WebSphere® WebSphere Extended Deployment Version 6.0.x

# ObjectGrid programming model guide

**Note**

Before using this information, be sure to read the general information under “Notices” on page 359.

**Compilation date: February 13, 2006**

**© Copyright International Business Machines Corporation 2004, 2005. All rights reserved.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

© Copyright IBM Corp. 2004, 2005

## How to send your comments

Your feedback is important in helping to provide the most accurate and highest quality information.

- To send comments on articles in the WebSphere Extended Deployment Information Center, available at: `http://www.ibm.com/software/webservers/appserv/extend/library/`
  1. Display the article in your Web browser and scroll to the end of the article.
  2. Fill out the **Feedback** link at the bottom of the article and submit.
- To send comments on this or another PDF books, you can e-mail your comments to: **wasdoc@us.ibm.com**.

  Be sure to include the document name and number, and, if applicable, the specific page, table, or figure number on which you are commenting.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2004, 2005

# Chapter 1. Getting started with ObjectGrid by running the sample application

Use this topic to get started with ObjectGrid, a distributed computing framework that makes objects available to a set of applications.

WebSphere Extended Deployment Version 6.0 or later and WebSphere Application Server Version 6.0.2 or later must be installed on at least one machine in your environment.

**Restriction:** If you are using ObjectGrid with WebSphere Extended Deployment Version 6.0, additional licensing arrangements are required to also use ObjectGrid in a Java 2 Platform, Standard Edition (J2SE) Version 1.4.2 or higher environment or in a WebSphere Application Server Version 6.02 or higher environment. Contact your sales representative for details.

If you want to develop ObjectGrid applications without accessing server machines that have WebSphere Extended Deployment installed, you can run them on your local machine. The local machine requires the installation of an IBM Software Developer Kit (SDK) or Eclipse.

To develop ObjectGrid applications on your local machine, copy the following directories from your installation to your local machine:

- If you are using WebSphere Extended Deployment Version 6.0.1, copy the /lib/wsobjectgrid.jar file and the /optionalLibraries/ObjectGrid/objectgridSamples.jar file to your working directory.
- If you installed ObjectGrid through the mixed server environment installation, copy the /ObjectGrid/lib/objectgrid.jar and the /ObjectGrid/samples/objectgridSamples.jar files to your working directory.

For more information about the Java archive (JAR) files that are installed with ObjectGrid, see ObjectGrid packaging.

Use this task to run and step through ObjectGrid sample applications. You can run the applications in this task in a Java command line, Eclipse, or Java 2 Platform, Enterprise Edition (J2EE) environment.

- To get the ObjectGrid sample application running on the command line, see Running the ObjectGrid sample application on the command line.
- To run the ObjectGrid sample application in Eclipse, see Importing and using the ObjectGrid sample application in Eclipse.
- To run the ObjectGrid sample application on WebSphere Extended Deployment, see Loading and running the ObjectGrid sample application with WebSphere Extended Deployment

You got started with ObjectGrid by running the sample application and loading the sample into your development environment.

## Running the ObjectGrid sample application on the command line

Use this topic to run ObjectGrid-enabled applications on a Java command line and test your ObjectGrid configuration.

© Copyright IBM Corp. 2004, 2005

Before you begin this task, install the mixed server environment, including the standalone ObjectGrid.

You must have a Software Development Kit (SDK) installed. You also must have access to the ObjectGrid sample applications. See Getting started with ObjectGrid for more information.

Use this task to quickly run an application with ObjectGrid enabled.

1. Check your Software Development Kit (SDK) version. ObjectGrid requires an IBM SDK 1.4.2 or higher. To test your Java environment before running the ObjectGrid sample application, perform the following steps:
   a. Open a command-line prompt.
   b. Type the following command:
      ```
      java -version
      ```
      If the command runs correctly, text similar to the following example displays:
      ```
      java version "1.4.2"
      Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.2)
      Classic VM (build 1.4.2, J2RE 1.4.2 IBM Windows 32 build cn142-20040820
      (JIT enabled: jitc))
      ```
      **Note:** You can also run these samples using a Java 2 Platform, Standard Edition (J2SE) Version 1.3.x Software Development Kit (SDK). For more information, see ObjectGrid packaging.

      If an error displays, ensure that the SDK is installed and is in your CLASSPATH.
2. Run the ObjectGrid sample application. The sample application illustrates a simple case that involves employees, offices, and work locations. The sample application creates an ObjectGrid instance with maps for each object type. Each map has entries inserted and manipulated to demonstrate the ObjectGrid caching function.
   a. Open a command line and navigate to the working directory. Copy the objectgrid.jar, asm.jar, and cglib.jar files from the /ObjectGrid/lib folder to a working directory. Copy the /ObjectGrid/samples/objectgridSamples.jar to the working directory.
   b. Issue the following command:
      ```
      cd working_directory
      java -cp "objectgrid.jar;objectgridSamples.jar;asm.jar;cglib.jar"
       com.ibm.websphere.samples.objectgrid.basic.ObjectGridSample
      ```
      The system displays output that is similar to the following text. This output has been shortened for publishing purposes:
      ```
      Initializing ObjectGridSample ...
      resourcePath: META-INF/objectgrid-definition.xml
      objectgridUrl:
       jar:file:/C:/temp/objg/objectgridSample.jar!/
       META-INF/objectgrid-definition.xml
      EmployeeOptimisticCallback returning version object for employee
       = Perry Cheng, version = 0
      EmployeeOptimisticCallback returning version object for employee =
       Hao Lee, version = 0
      EmployeeOptimisticCallback returning version object for employee =
       Ken Huang, version = 0
      EmployeeOptimisticCallback returning version object for employee =
       Jerry Anderson, version = 0
      EmployeeOptimisticCallback returning version object for employee =
       Kevin Bockhold, version = 0
      ```

```
---------------------------------------------
com.ibm.websphere.samples.objectgrid.basic.ObjectGridSample status:
ivObjectGrid Name = clusterObjectGrid
ivObjectGrid = com.ibm.ws.objectgrid.ObjectGridImpl@187b81e4
ivSession = com.ibm.ws.objectgrid.SessionImpl@6b0d81e4
ivEmpMap = com.ibm.ws.objectgrid.ObjectMapImpl@6b1841e4
ivOfficeMap = com.ibm.ws.objectgrid.ObjectMapImpl@6ba081e4
ivSiteMap = com.ibm.ws.objectgrid.ObjectMapImpl@6bae01e4
ivCounterMap = com.ibm.ws.objectgrid.ObjectMapImpl@697b41e4
---------------------------------------------
interactiveMode = false
Action = populateMaps
CounterOptimisticCallback returning version object for
 counter name = Counter1, version = 0
CounterOptimisticCallback returning version object for
 counter name = Counter2, version = 0
CounterOptimisticCallback returning version object for
 counter name = Counter3, version = 0
ivCounterMap operations committed
ivOfficeMap operations committed
... ending with:
CounterOptimisticCallback returning version object for
 counter name = Counter1, version = 0
EmployeeOptimisticCallback returning version object for employee =
 Ken Huang, version = 0
CounterOptimisticCallback returning version object for
 counter name = Counter2, version = 0
EmployeeOptimisticCallback returning version object for employee =
 Perry Cheng, version = 0
CounterOptimisticCallback returning version object for counter name =
 Counter3, version = 0
EmployeeOptimisticCallback returning version object for employee =
 Jerry Anderson, version = 0
CounterOptimisticCallback returning version object for
  counter name = Counter4, version = 0
EmployeeOptimisticCallback returning version object for employee =
 Hao Lee, version = 0
EmployeeOptimisticCallback returning version object for employee =
 Kevin Bockhold, version = 1
DONE cleanup
```

3. Run the distributed ObjectGrid sample application.

   The com.ibm.websphere.samples.objectgrid.basic.ObjectGridSample program uses a local ObjectGrid instance as the data cache. All objects are cached in the local Java virtual machine (JVM). To use a distributed ObjectGrid that is deployed in an ObjectGrid cluster, use the com.ibm.websphere.samples.objectgrid.distributed.DistributedObjectGridSample program. The DistributedObjectGridSample program is included in the objectgridSamples.jar .

   a. Start an ObjectGrid cluster. For more information about starting a standalone ObjectGrid cluster to use with the distributed ObjectGrid sample, see Starting the standalone sample ObjectGrid cluster.
   b. After you have the ObjectGrid server started, you can run the distributed ObjectGrid sample application with the following command:

      ```
       java -cp "objectgrid.jar;objectgridSamples.jar;asm.jar;cglib.jar"
      com.ibm.websphere.samples.objectgrid.distributed.DistributedObjectGridSample
      ```

   After the required ObjectGrid cluster is started, the DistributedObjectGridSample program has similar output to the ObjectGridSample program.

You ran the ObjectGrid sample application on a Java command line to test the ObjectGrid functionality.

The source for this sample is in the objectgridSamples.jar file, specifically in the com\ibm\websphere\samples\objectgrid\basic\ObjectGridSample.java and com\ibm\websphere\samples\objectgrid\distributed\DistributedObjectGridSample.java files.

## Starting the standalone sample ObjectGrid cluster

To run the distributed ObjectGrid sample, you must start an ObjectGrid cluster that hosts the required ObjectGrid.

Verify that WebSphere Extended Deployment for Mixed Server Environment, Version 6.0.x is installed.

Use this task to start an ObjectGrid server that is based on the `cluster-config-1.xml` and `cluster-objectgrid-definition.xml` files. This task is required to run the distributed ObjectGrid sample. See Running the ObjectGrid sample application on the command line and Importing and using the ObjectGrid sample application in Eclipse for more information. The `cluster-config-1.xml` only has one ObjectGrid server definition. This ObjectGrid server represents the sample ObjectGrid cluster.

1. Locate the `objectgridSamples.jar` file in the *mse_install_root*`/ObjectGrid/samples` directory.
2. Extract the `META-INF/cluster-config-1.xml` file and the `META-INF/cluster-objectgrid-definition.xml` file from the `objectgridSamples.jar` file to the *mse_install_root*`/ObjectGrid/samples` directory.
3. Verify that the JAVA_HOME environment variable is set and that the Java version meets the requirement. The ObjectGrid Server requires a Java 2 Platform, Standard Edition (J2SE) Version 1.4.2 or later environment. To check your Java environment, perform the following steps:
   a. Check the JAVA_HOME environment variable. On a command line prompt, issue the following command:
      ```
      echo %JAVA_HOME%
      ```
      This command displays the path to Java. If you need to set the JAVA_HOME environment variable, run the following command:
      ```
      set JAVA_HOME=JDK_INSTALL_ROOT
      ```
      Set the *JDK_INSTALL_ROOT* to your Java installation directory, for example, `c:\java`.
   b. Check your Java version. Run the following command:
      ```
      java -version
      ```
      Verify that your version is Java 2 Platform, Standard Edition (J2SE) Version 1.4.2 or later.
4. Start the ObjectGrid server. On a command line prompt, issue the following commands:
   ```
   cd mse_install_root/ObjectGrid/bin

   startOgServer.bat server1 -objectgridFile mse_install_root/ObjectGrid/
    samples/META-INF/cluster-objectgrid-definition.xml
   -clusterFile mse_install_root/ObjectGrid/samples/META-INF/
    cluster-config-1.xml
   -jvmArgs -cp mse_install_root/ObjectGrid/samples/objectgridSamples.jar
   ```

**Important:** You must specify the `objectgridSamples.jar` file in the classpath through the -jvmArgs option. The `objectgridSamples.jar` file contains classes that the sample ObjectGrid server needs for the plug-in implementations that are defined in the `cluster-objectgrid-definition.xml` file. This JAR file is also used for serializing and deserializing the objects that are stored in maps.

The system displays output that is similar to the following text. This output has been shortened for publishing purposes:

```
************ Start Display Current Environment ************
[1/17/06 14:04:34:144 CST] 7daee176 Launcher
I CWOBJ2501I: Launching ObjectGrid server server1.
:
[1/17/06 14:04:37:719 CST] 7daee176 ServerRuntime
I CWOBJ1001I: ObjectGrid Server server1 is ready to process requests.
```

See Running the ObjectGrid sample application on the command line or Importing and using the ObjectGrid sample application in Eclipse to run the distributed ObjectGrid sample application. For more details about starting and stopping the standalone ObjectGrid server on the command line, see Chapter 8, “Command line support,” on page 79.

## Importing and using the ObjectGrid sample application in Eclipse

Use this task to import and use the ObjectGrid sample application in Eclipse.

Before you begin this task, install the mixed server environment, including the standalone ObjectGrid.

For this sample application, use Eclipse Version 3.1 or later to import and run the sample. You can obtain Eclipse from the Application Server Toolkit that is included with WebSphere Application Server, from installing Rational Application Developer, or by downloading it directly from Eclipse.org.

By using Eclipse, you can easily debug your applications. You can perform a step-by-step walk through of the sample application.

1. Import the project into Eclipse:
   a. Run the Eclipse program. Use the `eclipse.exe` file in the Eclipse installation directory.
   b. Using Eclipse, create a new project.
      1) Click **File > New > Project > Java > Java Project**. Click **Next**.
      2) Type a project name. For example, type `ObjectGridSamples`.
      3) Select **Create new project in workspace**.
      4) In the Project Layout section, click **Configure default**.
      5) For the source and output folder, select **Project** and click **OK**.
      6) Click **Next**.
      7) Click the **Libraries** tab.
      8) Click **Add External JARs.**
      9) Navigate to the `/ObjectGrid/lib` folder and select the **objectgrid.jar**, **asm.jar**, and **cglib.jar** files. Click **Open** in the **JAR selection** wizard.
      10) Click **Finish**.
2. Import the `objectgridSamples.jar` file into the Java Project.
   a. Right-click on the Java project and select **Import**.
   b. Select **Zip file** under **Select an import source**.

   c. Click **Next**.
   d. Click **Browse** to open the Import From Zip File wizard.
   e. Open the **objectgridSamples.jar** file. Navigate to the `/ObjectGrid/samples` directory. Select the **objectgridSamples.jar** file and click **Open**.
   f. Verify that the check box of the root file tree is selected.
   g. Verify that the **Into folder** contains the Java project that you created in the previous step, for example, the `ObjectGridSamples` project.
   h. Click **Finish**.
3. Check the properties of the Java Project.
   a. Open the Java Perspective. Click **Window > Open Perspective > Java**.
   b. Go to the console view. Click **Window > Show view > Console**.
   c. Verify that the Package Explorer view is available and selected. Click **Window > Show View > Package Explorer**.
   d. Right-click on the Java project and select **Properties**.
   e. Click **Java Build Path** on the left panel.
   f. Click the **Source** tab in the right panel.
   g. Verify that the project root is listed in the Source folders on the Build path panel.
   h. Click the **Libraries** tab in the right panel.
   i. Verify that the `objectgrid.jar`, `asm.jar`, and `cglib.jar` files and a JRE System Library are listed in the JAR and class folders on the Build path panel.
   j. Click **OK**.
4. Run the ObjectGrid sample.
   a. From the Package Explorer view, expand the Java project.
   b. Expand the com.ibm.websphere.samples.objectgrid.basic package.
   c. Right-click on the **ObjectGridSample.java** file. Click **Run > Java Application**.
   d. The console displays similar output to when you run the application on the Java command line. For an example of the output, see Running the ObjectGrid sample application on the command line.
5. Run the distributed ObjectGrid sample. To run the distributed ObjectGrid sample, you must configure an ObjectGrid cluster. For running this sample, you can use the predefined XML configuration files that are provided in the objectgridSamples.jar file. See Starting the standalone sample ObjectGrid cluster for more information.

   After the ObjectGrid server is started, you can run the distributed ObjectGrid sample application with the following steps:
   a. From the Package Explorer view, expand the Java project.
   b. Expand the com.ibm.websphere.samples.objectgrid.distributed package.
   c. Right-click on the **DistributedObjectGridSample.java** file. Click **Run > Java Application**.
   d. The console displays output that is similar to the ObjectGrid sample.

The steps to load the project and run the debugger are also in the `SamplesGuide.htm` file. The `SamplesGuide.htm` file is in the `doc` directory in the `objectgridSamples.jar` file.

**Related reference**

ObjectGrid packaging
You can access the ObjectGrid packages in two ways: by installing WebSphere Extended Deployment, or by installing the mixed server environment.

## Loading and running the ObjectGrid sample application with WebSphere Extended Deployment

Use this task to load and run the Java 2 Platform, Enterprise Edition (J2EE) ObjectGrid sample within WebSphere Extended Deployment.

WebSphere Application Server and WebSphere Extended Deployment must be installed.

Use this task to understand and test the integration of ObjectGrid with WebSphere Extended Deployment.For more information, see Chapter 10, "Integrating ObjectGrid with WebSphere Application Server," on page 279.

1. Install the `ObjectGridSample.ear` file. You can install the enterprise archive (EAR) file on a single application server or a cluster. To install the `ObjectGridSample.ear` file in the administrative console, perform the following steps:
   a. In the administrative console, click **Applications > Install New Application**.
   b. On the **Preparing for application installation** page, specify the location of the ObjectGrid sample application. For example, browse to: `<install_root>/installableApps/ObjectGridSample.ear`. Click **Next**.
   c. On the second **Preparing for application installation** page, take the default settings and click **Next**.
   d. On the **Select installation options** page, take the default settings and click **Next**.
   e. On the **Map modules to servers** page, specify deployment targets where you want to install the modules that are contained in your application. Select a target server or cluster from the **Clusters and servers** list for every module. Select the **Module** check box to select all of the application modules or select individual modules.
   f. On the following pages, use the default values and click **Finish**.
   g. Click **Save to Master Configuration** after finishing the application installation.
   h. Click the **Synchronize changes with Nodes** option. On the **Enterprise Applications > Save** page, click **Save**.
   i. Click **OK**.
2. Check the HTTP port of the default_host of the servers and add a host alias. By default, Web modules are bound to the default_host virtual host name, unless you modify the host name during installation. If you are installing the application on a cluster, you must configure at least one host alias for the HTTP port of the default_host for each cluster member. You also must check the HTTP port of the default_host for each cluster member and add the corresponding host alias into the Host aliases list in the administrative console. To check the HTTP port of the default_host of a server, perform the following steps:
   a. In the administrative console, click **Servers > Application Servers >** ***server_name***.
   b. Expand the ports in the Communication section. The **WC_defaulthost** port is the default_host virtual host port.

   To add a host alias, perform the following steps:

   a. In the administrative console, click **Environment > Virtual hosts > default_host > Host aliases > New**.
   b. Use the default value of the host name and specify the port.
   c. Click **OK**.
3. Start the ObjectGrid sample application.
   - To start the application on a server, click **Servers > Application servers**. Select the server that has the `ObjectGridSample.ear` file installed. Click **Start**.
   - To start the application on a cluster, click **Servers > Clusters**. Select the cluster that has the `ObjectGridSample.ear` file installed. Click **Start**.

   After you start the application on a server or cluster, you can stop and start the application independently from the host server or cluster. To stop or start the ObjectGrid sample application, perform the following steps:
   a. In the administrative console, click **Applications > Enterprise applications**.
   b. Select the ObjectGrid sample application.
   c. Click **Start** or **Stop**.
4. Access the ObjectGrid sample. After you install the `ObjectGridSample.ear` file on a single server or cluster and start the application, you can access the ObjectGrid sample at the following Web address:

   ```
   http://hostname:port/ObjectGridSample
   ```

   For example, if your host name is `localhost` and the port value is `9080`, use the `http://localhost:9080/ObjectGridSample` Web address.
5. Test the functionality of distributed ObjectGrid in the WebSphere Application Server environment. The `ObjectGridSample.ear` file also contains the DistributedObjectGridServlet servlet that demonstrates the use of a distributed ObjectGrid in the WebSphere Application Server environment. The application server that hosts the DistributedObjectGridServlet servlet must also host the ObjectGrid server, which is a member of the required ObjectGrid cluster.
   - For more information about configuring an ObjectGrid cluster to get the DistributedObjectGridServlet running, see Starting a sample ObjectGrid cluster in the WebSphere environment.
   - For more information about starting ObjectGrid servers in application servers, see Starting an ObjectGrid server in an application server.

   After the application server with the `ObjectGridSample.ear` file installed also hosts the required ObjectGrid server, the DistributedObjectGridServlet servlet behaves the same way as other servlets. You can access the servlet at the following Web address: `http://hostname:port/ObjectGridSample/DistributedObjectGridServlet`. For example, if your host name is `localhost` and the port value is 9080, use the `http://localhost:9080/ObjectGridSample/DistributedObjectGridServlet` Web address.

   You can enable ObjectGrid tracing by using the following trace string: `ObjectGrid*=all=enabled`.

You installed and configured the ObjectGrid sample application and the distributed ObjectGrid sample application on a WebSphere Extended Deployment server.

After you install the application on a server or cluster, you can access the sample documentation after starting the application at the following Web address:

```
http://hostname:port/ObjectGridSample/docs/introduction.html
```

For example, if your hostname is **localhost** and the port value is **9080**, use the `http://localhost:9080/ObjectGridSample/docs/introduction.html` Web address.

## Starting a sample ObjectGrid cluster in the WebSphere environment

Use this task to start a simple ObjectGrid cluster to test the functionality of the distributed ObjectGrid in the WebSphere Application Server environment.

WebSphere Extended Deployment must be installed. You must have the ObjectGridSample.ear file installed on your application server. For more information about installing the ObjectGridSample.ear file, see Loading and running the ObjectGrid sample application with WebSphere Extended Deployment.

Use this task to set up an application server to host an ObjectGrid server that is based on the cluster-config-1.xml and cluster-objectgrid-definition.xml files.

The cluster-config-1.xml file only has one ObjectGrid server definition. This ObjectGrid server represents the sample ObjectGrid cluster. You can use either one standalone application server or a cluster with one cluster member to host the sample ObjectGrid server.

1. Extract both the META-INF/cluster-config-1.xml and META-INF/cluster-objectgrid-definition.xml files from the /optionalLibraries/ObjectGrid/objectgridSamples.jar file to the /optionalLibraries/ObjectGrid directory.
2. Define the necessary generic JVM arguments.
   a. In the administrative console, click **Servers > Application servers >** ***server_name*** **> Process Definition > Java Virtual Machine**.
   b. In the Generic JVM arguments panel, type the following text:
   ```
   -Dobjectgrid.server.name=server1
   -Dobjectgrid.xml.url=file:///<INSTALL_ROOT>\optionalLibraries\ObjectGrid\
    META-INF\cluster-objectgrid-definition.xml
   -Dobjectgrid.cluster.xml.url=file:///<INSTALL_ROOT>\optionalLibraries\
    ObjectGrid\META-INF\cluster-config-1.xml
   ```
   The `INSTALL_ROOT` is your WebSphere Application Server install root directory.
   c. Click **Save**.
   d. Click **Save to Master Configuration**.
   e. Select the **Synchronize changes with Nodes** option. Click **Save**.
3. Copy the /optionalLibraries/ObjectGrid/objectgridSamples.jar file to the /classes or the `lib/ext` directory. The objectgridSamples.jar contains classes that the sample ObjectGrid server needs for the plug-in implementations that are defined in the cluster-objectgrid-definition.xml file. This JAR file is also used for serializing and deserializing the objects that are stored in maps .
4. Restart server to make the change takes effect.

For more details about starting and stopping ObjectGrid servers in application servers, see Starting an ObjectGrid server in an application server.

## Starting an ObjectGrid server in an application server

An ObjectGrid server can be configured to start within an application server. WebSphere Application Server detects the ObjectGrid component and automatically starts the ObjectGrid server.

You can configure ObjectGrid servers in WebSphere Application Server Version 6.0.2 and later, including when add-ons such as WebSphere Extended Deployment or WebSphere Business Integration Server are installed. Earlier versions of WebSphere Application Server, such as WebSphere Application Server Version 5.0.2, can have applications that use the ObjectGrid as clients, but the ObjectGrid server function cannot be collocated with the earlier application server versions.

If you are using cluster configurations that enable replication, the high availability manager is required. ObjectGrid servers use the high availability manager differently than normal application servers. When the ObjectGrid server is in an application server, the ObjectGrid server does not configure, initialize or create the high availability manager service, but uses the existing high availability service in the application server. For replication between ObjectGrid servers, the ObjectGrid servers must be running in application servers that are members of the same core group.

All other functions of the ObjectGrid server are the same when the server runs in WebSphere Application Server. If your ObjectGrid cluster specification includes three servers, any three application servers in a single core group can host these ObjectGrid servers. The application servers can also span clusters, as long as the clusters belong to the same core group. The most important step is to correlate the server TCP/IP host name and port information in the cluster.xml file.

Use this task to run ObjectGrid servers within the application servers in your WebSphere Application Server environment.

1. Add the required custom properties on the Java Virtual Machine (JVM). In the administrative console, click **Servers > Application servers > *server_name* > Java and Process Management > Process Definition > Java Virtual Machine > Custom Properties**. Click **New**. Create the following custom properties:

*Table 1. JVM custom properties for ObjectGrid servers*

| Custom property name | Description | Example value |
|---|---|---|
| `objectgrid.server.name` | Specifies the ObjectGrid server name to be used within this application server. The name provided must be one of the server names that is defined in the ObjectGrid cluster XML file. | server1 |
| `objectgrid.xml.url` | Specifies the Universal Resource Locator (URL) for the ObjectGrid XML file. This property is required. | `file:///d:/was/etc/test/objectGridMatch.xml` |
| `objectgrid.cluster.xml.url` | Specifies the URL for the ObjectGrid cluster XML file. This property is required | `file:///d:/was/etc/test/csCluster0.xml` |

*Table 1. JVM custom properties for ObjectGrid servers (continued)*

| Custom property name | Description | Example value |
|---|---|---|
| `objectgrid.security.server.props` | Specifies the URL for the ObjectGrid server security properties file. This property is required only if security is enabled in the ObjectGrid cluster xml file. To determine if security is enabled in your cluster XML file, look for the following text:<br>`<cluster name="cluster1"`<br>` securityEnabled="true"`<br>`....`<br><br>If the securityEnabled attribute is set to `false`, you do not need to define this property.<br><br>Use the `security.ogserver.props` file as a template. See the "ObjectGrid security" on page 131 for the meaning of these properties in this file and how they can be used. | `file:///d:/was/optionalLibraries/ObjectGrid/properties/security.ogserver.props` |

You can also define these JVM properties in the **Generic JVM Arguments** field on the **Java Virtual Machine** panel in the administrative console. Following is an example value for the Generic JVM Arguments field:

```
-Dobjectgrid.server.name=server1
-Dobjectgrid.xml.url=file:///<INSTALL_ROOT>\optionalLibraries\ObjectGrid\
META-INF\cluster-objectgrid-definition.xml
-Dobjectgrid.cluster.xml.url=file:///<INSTALL_ROOT>\optionalLibraries\
ObjectGrid\META-INF\cluster-config-1.xml
```

2. Save the changes and restart the application server. WebSphere Application Server detects the ObjectGrid component and automatically starts the ObjectGrid server.

   The ObjectGrid in the application server uses the channel framework to interact with ObjectGrid clients, specifically called the Client Access port. When the ObjectGrid server is started, it detects collocation with WebSphere Application Server and uses the channel framework that is already running in the application server. The ObjectGrid server creates and starts its own channel framework only if a channel framework is not created or started in the application server.
3. Stop the ObjectGrid server. Stop the ObjectGrid server by stopping the associated application server. You cannot stop the ObjectGrid sever by using the ObjectGrid system management commands.

The application servers in your WebSphere Application Server environment are running ObjectGrid servers.

# Chapter 2. ObjectGrid

ObjectGrid is an extensible transactional object caching framework for Java 2 Platform, Standard Edition (J2SE) and Java 2 Platform, Enterprise Edition (J2EE) applications.

You can use the ObjectGrid API when developing your applications to retrieve, store, delete, and update objects in the ObjectGrid framework. You can also implement customized plug-ins that monitor updates to the cache, retrieve and store data with external data sources, manage eviction of entries from the cache, and handle background cache functionality for your own ObjectGrid application environment.

**Map-based API**

The ObjectGrid provides an API that is based on the java.util.Map interface. The API is extended to support the grouping of operations into transactional blocks. This interface is a superset of the java.util.Map interface and adds support for batch operations, invalidation, keyword association, and explicit insert and update. The Java Map semantics are enhanced with extension points so that you can implement the following enhancements:

- Cache evictors to fine-tune cache entry lifetimes
- Transaction callback interfaces to carefully control transaction management and optionally integrate with the WebSphere transaction manager in J2EE environments
- Loader implementations that automatically retrieve and place data to and from a database when an application programmer uses the ObjectGrid Map get and put operations
- Listener interfaces that can provide information about all committed transactions as they occur and are applied towards the ObjectGrid framework as a whole or are applied for particular Map instances.
- Object transformer interfaces that allow for more efficient copying and serializing of keys and values.

**The ObjectGrid environment**

You can use ObjectGrid framework by installing one of the existing offerings:

- ObjectGrid is integrated with WebSphere Extended Deployment Version 6.0.1 and is a part of the full installation.
- Standalone ObjectGrid is a part of the Mixed Server Environment (MSE) installation.

In both offerings, ObjectGrid supports client/server features. The server runtime supports full clustering, replication, and partitioning of distributed object caches. The client runtime supports the concept of a near cache and workload management routing logic to remote clusters. The client runtime also supports local object map creation.

The level of support varies depending on if you are running the client runtime, server runtime, integrated ObjectGrid, or the standalone ObjectGrid.

© Copyright IBM Corp. 2004, 2005

**ObjectGrid integrated with WebSphere Extended Deployment offering**

Server runtime: The server runtime is integrated. For WebSphere Extended Deployment Version 6.0.1, the integrated runtime is not supported on the z/OS platform.

Client runtime: The client runtime is supported on J2SE and J2EE at JDK level 1.3.1 and greater, including WebSphere Application Server Version 5.0.2 and later. The client runtime is fully supported on the z/OS platform.

**Standalone ObjectGrid offering**

Server runtime: The server runtime can run in standalone Java Virtual Machines (JVM) as a single server or as a cluster of servers. The standalone server is supported on most J2SE and J2EE platforms at JDK level 1.4.2 and greater. The standalone server is supported on WebSphere Application Server Version 6.0.2 and later. The standalone server runtime is not supported on the z/OS platform for WebSphere Extended Deployment Version 6.0.1.

Client runtime: The client runtime is supported on J2SE and J2EE platforms at JDK level 1.3.1 and greater, including WebSphere Application Sever Version 5.0.2 and later.

### Session management

A fully distributed HTTP Session management implementation is provided that stores HTTP Session objects in the ObjectGrid.

### Simple installation

You can install and configure ObjectGrid in a few simple steps. These steps include copying the Java archive (JAR) files to your class path and defining a few configuration directives.

### Transactional changes

All changes are made in the context of a transaction to ensure a robust programmatic interface. The transaction can either be explicitly controlled within the application, or the application can use the automatic commit programming mode. These transactional changes can be replicated across an ObjectGrid cluster in asynchronous and synchronous modes to provide scalable and fault tolerant access.

You can scale ObjectGrid from a simple grid running in a single Java virtual machine (JVM) to a grid that involves one or more ObjectGrid clusters of Java virtual machines. These servers make data available through the Map APIs to a large set of ObjectGrid-enabled clients. The ObjectGrid clients use the basic Java Map APIs. However, the application developer does not need to develop Java TCP/IP and remote method invocation (RMI) APIs because the ObjectGrid client can reach the other ObjectGrid servers that are holding information across the network. If your data set is too large for a single JVM, you can use ObjectGrid to partition the data.

ObjectGrid also offers your application solution added high availability capabilities. The object sharing is based on a replication model where a primary server, one or more replication servers, and one or more standby servers exist. This cluster of replication servers is referred to as a replication group. If the access to the replication group is a write operation, then the request is routed to the primary server. If the access is a read operation, or if the map is a read-only map, the request can route to the primary or replication servers. The standby servers are defined as potential replication servers if a server fails. If a primary server fails, then a replication server becomes the primary server to minimize any outage. This behavior is configurable and extensible based on your needs.

If you want to use a simpler object propagation approach, a lower quality of service peer to peer model is also available, as it was in Extended Deployment Version 6.0. With this simpler distributed transactional support, peers can be notified of changes by using a message transport. The message transport is built in if you are running WebSphere Application Server Version 6.0.2 or later. If you are not running WebSphere Application Server Version 6.0.2 or later, another message transport must be supplied, such as a Java Message Service (JMS) provider.

**Injection container compatible APIs**

Configure the ObjectGrid using a simple XML file or programmatically using Java APIs. The Java APIs are designed to also work in environments where you are using injection–based frameworks to configure your applications. The APIs and interfaces of the ObjectGrid objects can also be invoked by an Inversion of Control (IoC) container and then references to key ObjectGrid objects can be injected into the application.

**Extensible architecture**

You can extend most elements of the ObjectGrid framework by developing plug-ins. You can tune the ObjectGrid to allow an application to make trade-off decisions between consistency and performance. Plug-in customized code can also support the following application-specific behaviors:

- Listen to ObjectGrid instance events for initialization, transaction begin, transaction end, and destroy.
- Invoke transaction callbacks to enable transaction-specific processing.
- Implement specific common transaction policies with generic ObjectGrid transactions.
- Use loaders for transparent and common entry and exit points to external data stores and other information repositories.
- Handle non-serializable objects in a specific way with ObjectTransformer interfaces.

You can implement each of these behaviors without affecting the use of the basic ObjectGrid cache API interfaces. With this transparency, applications that are using the cache infrastructure can have data stores and transaction processing greatly changed without affecting these applications.

**Use ObjectGrid as a primary API or second-level cache**

The ObjectGrid APIs can be used directly by the application as a lookaside cache or as a write through cache. In write through mode, the application plugs in a Loader object so that the ObjectGrid can apply changes and fetch data directly and transparently to the application. ObjectGrid can also be used as a second-level cache for popular object relational mappers by writing an adapter. The cache is invisible to the application in this mode because the application uses the APIs from the object relational mapper as the primary API for accessing the data.

# Chapter 3. ObjectGrid overview

ObjectGrid provides a Java Map-based data access model and a distributed caching technology. With ObjectGrid, you can configure a highly available clustering environment. ObjectGrid clients can contact many different ObjectGrid clusters concurrently for large scale integration solutions. ObjectGrid also provides a rich, distributed data partitioning solution for large amounts of normalized information with data in more than one Java Virtual Machine. Fundamentally, ObjectGrid is a set of standardized Java APIs and network services that allow local and distributed caching. The solution scales from a single Java virtual machine (JVM) where a richer Java Map solution is required to a vast array of distributed and scalable data services are required from several ObjectGrid clusters throughout an entire enterprise.

## ObjectGrid in a single Java virtual machine (JVM)

The most basic usage of the ObjectGrid is in a single JVM.

You can use ObjectGrid to create a set of ObjectGrid instances. Each ObjectGrid instance can contain one or more Java Map-compatible instances. The Java Map instances provide the get and put interfaces that Java programmers are accustomed to, plus additional features that the current Java Map interface and functionality does not offer. The following diagram illustrates the most basic usage of ObjectGrid.

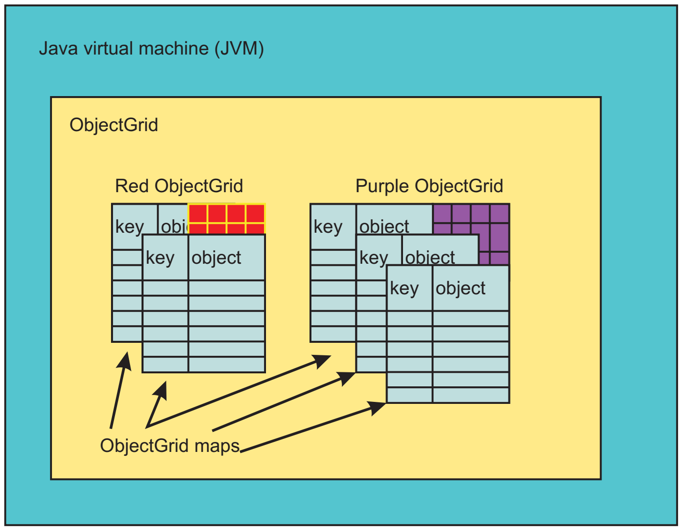


*Figure 1. ObjectGrid JVM Usage*

An ObjectGrid and ObjectGrid Map include many features that are not currently provided in the standard Java Map interface. These features include transactional access, various types of locking strategies (None, Optimistic and Pessimistic), Plug and Play Eviction, seamless interaction with databases as a side effect of using get and put APIs, and many other capabilities. You can also develop your own extensions to ObjectGrid. For example, you can develop a Map Listener that provides results for each transaction that is committed against a given Map instance. Users can log the changes, for example, to a file in a branch office location to ensure against lost transactions, or propagate the changes with Java message service (JMS) or some other infrastructure.

In the previous diagram, the JVM has two ObjectGrid instances, one with two Java Map-like objects for use and the other with three Map objects. The Map objects are two dimensional, allowing for a key and an object pairings to be manipulated like a normal Java Map. A single ObjectGrid instance can support many specific map instances.

The following configuration is a basic ObjectGrid configuration for the Red and Purple ObjectGrid instances:

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd
xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="Red">
   <backingMap name="FirstRedMap" readOnly="false" />
   <backingMap name="SecondRedMap readOnly="false" />
  </objetGrid>
  <objectGrid name="Purple">
   <backingMap name="FirstPurpleMap readOnly="false" />
   <backingMap name="SecondPurpleMap readOnly="false" />
   <backingMap name="ThirdPurpleMap readOnly="false" />
  </objectGrid>
 </objectGrids>
</objectGridConfig>
```

## Distributed ObjectGrid

In addition to using the ObjectGrid Java archive (JAR) file within a single JVM, you can use ObjectGrid in a distributed environment. In this environment, you can create an ObjectGrid cluster. An ObjectGrid cluster is made up of a set of ObjectGrid servers, each its own single JVM.

The “ObjectGrid in a single Java virtual machine (JVM)” on page 17 topic describes that ObjectGrid supports the concept of a Java Map. This concept is also supported locally in a single JVM and in a Java client that connects to one or more remote ObjectGrid cache clusters. The ObjectGrid servers offer the ability to distribute the basic functionality that is already described above in the single JVM case. For example, several clients can share the same ObjectGrid instance Map, using a locking strategy of None, Optimistic or Pessimistic. In addition, an evictor in the ObjectGrid cluster servers can manage eviction for the server side Map instance data. All clients can use the common get and put semantics, and the Loader that is configured on the ObjectGrid cluster server does all the interaction with the database instead of deploying and managing Java database connectivity (JDBC) drivers on each client.

In the following diagram, the JVM has two ObjectGrid instances: one with two Java Map-like objects for use and the other with three Java Map-like objects. The Maps each are two dimensional objects that allow a key and an object. A single ObjectGrid instance can support a large number of Maps, primarily depending on the application’s requirements. The difference in this case is that the Maps are housed within an ObjectGrid cluster server. The clients can be a normal Java application or Java 2 Platform, Enterprise Edition (J2EE) application servers.

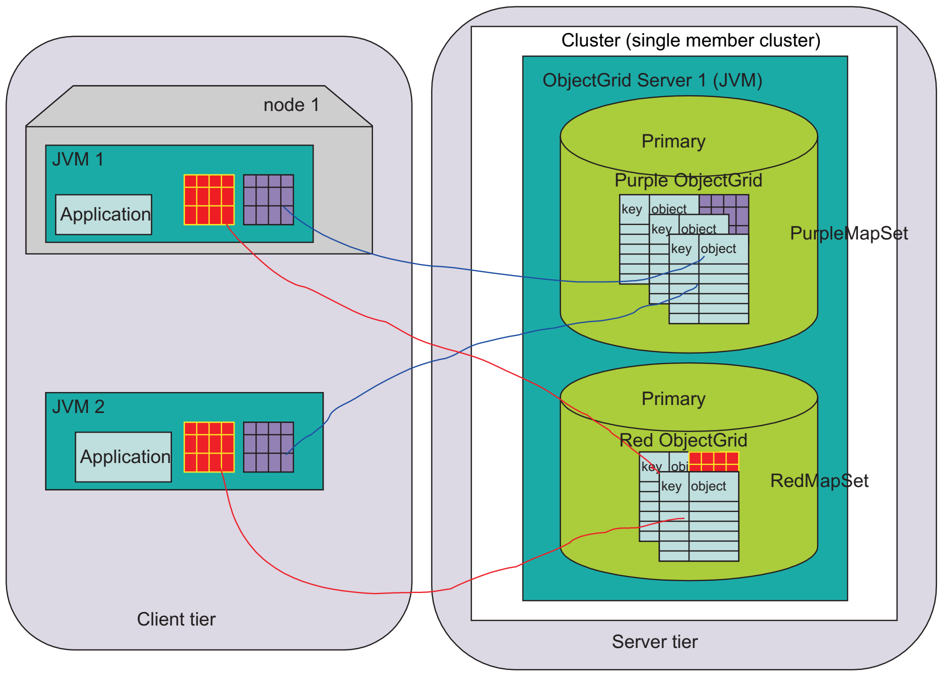


*Figure 2. Distributed ObjectGrid single server topology (with two MapSets)*

## ObjectGrid clients

ObjectGrid clients consist of a set of APIs to connect to an ObjectGrid cluster, bootstrap through the ObjectGrid cluster wide configuration, and then perform ObjectGrid map operations that are actually distributed. An ObjectGrid client is any Java application within its own JVM instance that is using the ObjectGrid in a distributed way. A distributed ObjectGrid client can also still use the non-distributed functionality in the same Java Virtual Machine. An ObjectGrid client usage can be as complicated as an entire application server with several parallel ObjectGrid connections, each with security enabled and acting on a different user's behalf.

To enable the distributed behavior, an ObjectGrid cluster (server side services of the ObjectGrid solution) must be created. The additional configuration required is an ObjectGrid Cluster XML file in addition to the ObjectGrid configuration file.

Following is the ObjectGrid Cluster XML that configures the ObjectGrid network deployment in the previous diagram:

```
<?xml version="1.0" encoding="UTF-8" ?>
<clusterConfig xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation=http://ibm.com/ws/objectgrid/config/cluster
../objectGridCluster.xsd
xmlns="http://ibm.com/ws/objectgrid/config/cluster">
<cluster name="cluster1">
 <!– single server ->
 <serverDefinition name="server1" host="localhost" clientAccessPort="12053"
  peerAccessPort="12500" />
</cluster>
 <objectGridBinding ref="Red">
  <mapSet name="RedMapSet" partitionSetRef="ColorMapsPartitioningSet">
   <map ref="FirstRedMap" />
   <map ref="SecondRedMap" />
  </mapSet>
 </objectGridBinding>
 <objectGridBinding ref="Purple">
  <mapSet name="PurpleMapSet" partitionSetRef="ColorMapsPartitioningSet">
   <map ref="FirstPurpleMap" />
   <map ref="SecondPurpleMap" />
   <map ref="ThirdPurpleMap" />
  </mapSet>
 </objectGridBinding>
 <partitionSet name="ColorMapsParitioningSet">
  <partition name="partition1" replicationGroupRef="ColorMapsReplicationGroup" />
 </partitionSet>
 <replicationGroup name="ColorMapsReplicationGroup">
  <replicationGroupMember serverRef="server1" priority="1" />
 </replicationGroup>
</clusterConfig>
```

This configuration describes a single cluster, ″cluster1″, which contains the server1 server. The server1 server hosts two ObjectGrids, ″Red″ and ″Purple″. The configuration file specifies information for partitioning and replication as well. The ObjectGrid client-server support in ObjectGrid requires the programmer to connect to one server defined in the ObjectGrid cluster. During the connect processing, the ObjectGrid and ObjectGrid cluster configuration is dynamically downloaded to the client, greatly simplifying preparing the client for usage and having to manage client side configuration content. Other than the ObjectGrid client performing a ″Connect″ operation, the programming APIs and concepts to use an ObjectGrid that is scoped to the local JVM and a JVM that is actually hosted in the ObjectGrid cluster are generally the same.

## ObjectGrid cluster initialization

You can start ObjectGrid servers within a cluster with the command line tools that are provided with ObjectGrid. An ObjectGrid application can include an ObjectGrid client and be integrated as any other Java API Library would be integrated into your application development framework. However, in both cases, ObjectGrid usage must be initialized.

To work within either the local Java virtual machine (JVM) usage scenario or in a distributed ObjectGrid cluster, you must obtain a valid configuration for bootstrap through a manageable approach. ObjectGrid clients and ObjectGrid cluster servers must use a uniform configuration. As a programmer, you might start with a very simple configuration, possibly bounded within a single Java application in a JVM.

Then, as you prepare to begin multi-client, concurrent user testing, create your first single server ObjectGrid cluster. After the initial client-server based testing is complete, you can work with the administrative staff and experiment with replication and other high availability solution requirements. Each of these normal progressions in development requirements require a richer configuration file.

The configuration file changes to enable each of these advanced features for each of the development stages described are relatively modest, but each stage in your solution development requires a different version of the configuration file. The intent is that the changes build upon each other. You can unit test a replicated solution on a single machine if the amount of data to develop the solution does not overwhelm a single system or can be artificially constrained for development purposes.

## ObjectGrid configuration with XML

A distributed ObjectGrid configuration, with one or more clients and one or more ObjectGrid servers, requires XML configuration. In addition to the base ObjectGrid XML configuration file, you must create an ObjectGrid cluster XML description.

A single ObjectGrid XML configuration description and an ObjectGrid Cluster XML configuration description provide the clients and the servers in a single ObjectGrid cluster the information that they need to function as expected. You can have any number of ObjectGrid clusters in your environment, however, a cluster-specific ObjectGrid cluster XML document must describe the particular cluster.

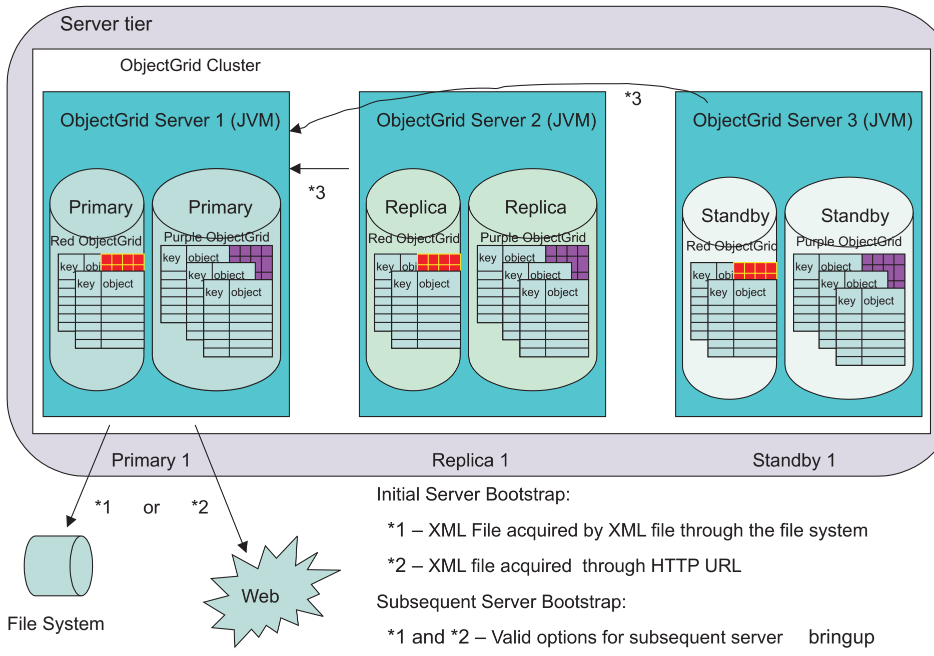
The configuration files that are required for the ObjectGrid to start can be acquired through any normal URL approach. For example, clients and servers can acquire the XML files with a physical file or an HTTP URL.

Within a distributed ObjectGrid environment as depicted in the following diagrams, an initial set of ObjectGrid servers can be configured with the command line to retrieve their configuration through a URL or, because the file URL can be complicated, a simple file on the file system. However, a better approach is to start subsequent servers within the same ObjectGrid cluster by bootstrapping them from other servers that are already operational within the cluster. This approach is much more manageable because administrators do not need to track configuration files on each machine that is hosting either an ObjectGrid client or server. In addition, a server that starts by bootstrapping can be assured that the XML has already been processed successfully, reducing XML configuration errors.

## Bootstrapping

### ObjectGrid server bootstrap

The following diagram depicts bootstrapping a typical ObjectGrid cluster environment hosting the same ObjectGrid configuration, but offering a richer replication cluster configuration. In this case, the first server bootstraps through an HTTP URL, and the second and third server are started from the first. The second and third servers can also be started from the same URL as the first server.



Initial Server Bootstrap:

*1 – XML File acquired by XML file through the file system

*2 – XML file acquired through HTTP URL

Subsequent Server Bootstrap:

*1 and *2 – Valid options for subsequent server bringup

*3 – Bootstrap from other servers in the same cluster

*Figure 3. Initial server bootstrap though XML file configuration or from an existing server*

As illustrated in the previous diagram, the server1 server in the cluster1 cluster is the initial server to bootstrap. The server1 server can bootstrap through an XML file on the file system, or through a URL to a local file, remote HTTP server or other valid URL option. The server2 server and server3 server can be started through these means or by targeting the server1 server as a configuration bootstrap host. In general, bootstrapping subsequent servers from other servers ensures that the configuration is consistent across cluster members.

In this particular scenario, if the server1 server fails, and server2 and server3 are still operational, server1 can be bootstrapped from server2 or server3, or again through the file or URL approaches. See "ObjectGrid client connect APIs" on page 94 for more details on bootstrapping and the specific configuration options.

### ObjectGrid client bootstrap

The ObjectGrid client, to use the ObjectGrid cluster server members services, must bootstrap from one of the ObjectGrid servers within the cluster. Each client can ″connect″ to any active member of the cluster. Administrators can configure specific servers to perform that service. For large client deployments, the sole purpose for any configured ObjectGrid cluster servers is to provide client bootstrap support. This approach is helpful if the number of clients is large and they connecting and disconnecting often. After the client ″connects″, they can get a distributed reference to the ObjectGrids defined in the cluster configuration. See "ObjectGridManager interface" on page 87 for more information.

Clients acquire their standard configuration from the ObjectGrid cluster, so that the administrator does not have to manage the XML for the client community. The ObjectGrid client can use a remote URL just as the ObjectGrid cluster servers do for bring-up to override specific settings that should be client specific.

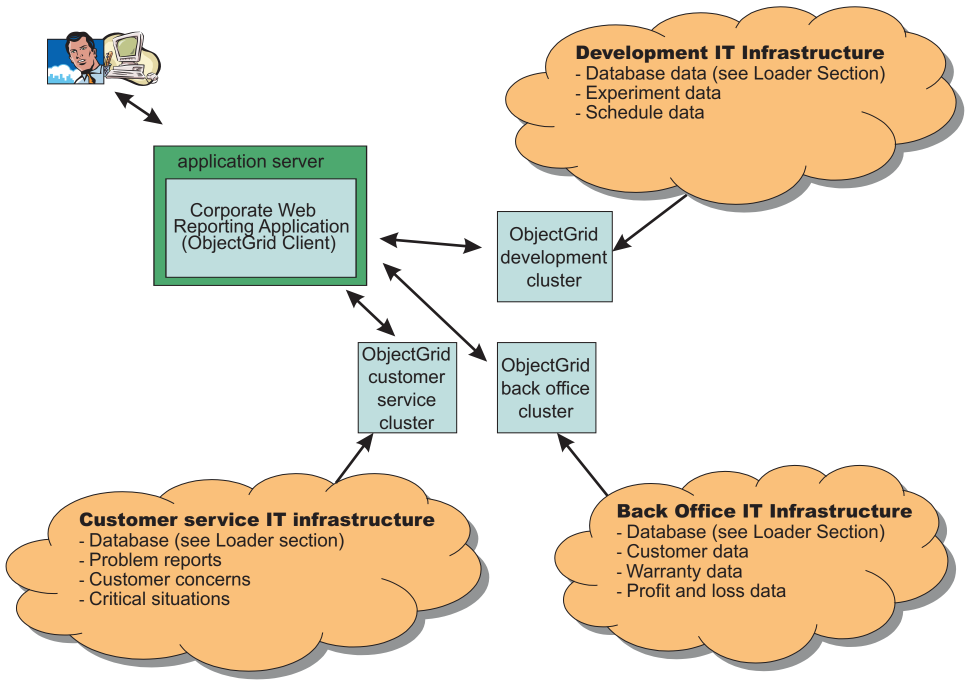
## ObjectGrid clients in a distributed ObjectGrid environment

ObjectGrid clients can connect to more than one ObjectGrid cluster concurrently. A single Java application within a Java virtual machine (JVM) can connect to the same remote cluster multiple times. This application can also attach to different remote clusters at the same time. This capability is important because it enables client functionality to access many different resources of information that are exported through one or more ObjectGrid clusters.

The first case, in which the same ObjectGrid client can contact the same ObjectGrid server is important for secure environments where the client might be an application server, and each connection from the application server to the remote ObjectGrid cluster uses different security credentials. Another example is an ObjectGrid client that needs to correlate data from several different ObjectGrid clusters for a single purpose.

The following diagram depicts a scenario where the corporate Web-based client user, through a Web application, is generating a report from three different corporate divisions. The servlet engine uses the application server ObjectGrid client functionality to contact three different ObjectGrid clusters, managed by each corporate division. In many corporations, data can be collected, and a key goal of ObjectGrid is to make the information more available in an easy way. After the information is externalized, other users who have the interest and security credentials can acquire and use the information in new ways. The data can be provided in a read-only mode, or when appropriate, for read-write update scenarios.

In this scenario, the data can be acquired in a secure manner. ObjectGrid caching in this scenario is not only enabling flexible data sharing in a common programmatic way within each corporate division, but also enabling cross-division data access for information acquired through a very secure, simple and clean programming model that many Java developers have used often.



Development IT Infrastructure
- Database data (see Loader Section)
- Experiment data
- Schedule data
application server
Corporate Web Reporting Application (ObjectGrid Client)
ObjectGrid development cluster
ObjectGrid customer service cluster
ObjectGrid back office cluster
Customer service IT infrastructure
- Database (see Loader section)
- Problem reports
- Customer concerns
- Critical situations
Back Office IT Infrastructure
- Database (see Loader Section)
- Customer data
- Warranty data
- Profit and loss data

*Figure 4. A Web-based client user generates a report from three different corporate divisions.*

## ObjectGrid clustering concepts

The term *distributed ObjectGrid* includes the concept that clients that can interact with one or more ObjectGrid clusters. An ObjectGrid cluster consists of one to many ObjectGrid servers.

### ObjectGrid client

An ObjectGrid client can be thought of in two ways. You can think of a client as a Java virtual machine (JVM) that uses the ObjectGrid API to connect to an ObjectGrid cluster and perform Java Map operations against that cluster. The second, more formal way to think about a client is to consider the concept of multiple clients within the same JVM. If you fully use the provided ObjectGrid function, you can use multiple clients within the same JVM.

Each time a programmer runs the ObjectGrid client connect operation in a JVM, a cluster context returns. This context is actually one client instance. Under the covers, asynchronous threads handle many aspects of the caching per context. For each context, ObjectGridManager can be used to acquire ObjectGrids that are hosted in the specific remote ObjectGrid cluster instances. So, in general, if you connect to three remote clusters in the same JVM, you implement a three client solution within the same JVM.

Important considerations for this scenario are the following. A single transaction session cannot span a Map set within the same cluster. Users cannot have a single transaction across different clients attached to the same or different ObjectGrid cluster. However, for users trying to integrate silos of information, users can use a transaction to pull information from each of the remote ObjectGrid clusters, and print consolidation reports or join the information and send through a ObjectGrid transaction data to another ObjectGrid cluster, or simply update the individual ObjectGrid clusters in a customer specific fashion. This is primarily because ObjectGrid offers single phase transaction support, as opposed to two phase transaction support separate transaction managers typically offer. For more information on this topic, see "ObjectGrid transaction demarcation" on page 34.

## Replication

You can replicate between ObjectGrid servers that are within the same ObjectGrid cluster. With replication, you can recover from a failure more quickly when the primary ObjectGrid server that has the particular information the user requires fails or is shutdown for maintenance. In the following diagram, the Red ObjectGrid and Purple ObjectGrid are in two different ObjectGrid replication group members. In ObjectGrid, each MapSet, a subset of an ObjectGrid can be replicated as a unit. PartitionSets are an exception to this rule,as discussed in the following section. The single server configuration described is modified in the following diagram to describe replication.

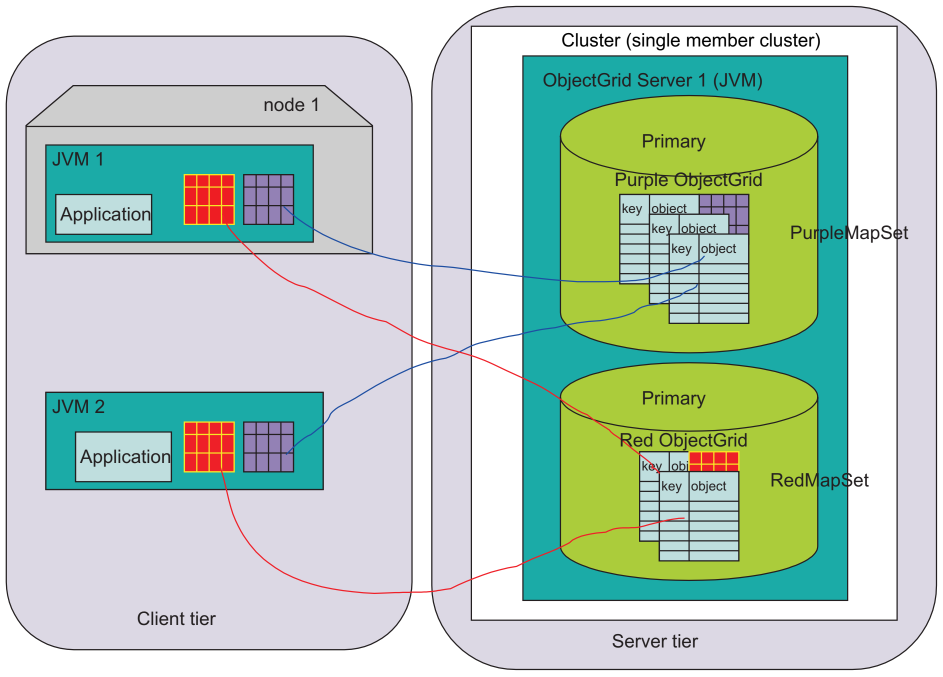


*Figure 5. Distributed ObjectGrid single server topology with two MapSets*

The diagram depicts an application server as a client application and standalone Java application. Both clients require access to two ObjectGrid instances, the Red and Purple instances in a single server ObjectGrid cluster. Each of these instances actually is contained in a replication group member. A replication group member is a key concept, and is the boundary for ObjectGrid transaction demarcation. A transaction can only commit changes to a single replication group member.

In an ObjectGrid cluster, the Java client can start an ObjectGrid transaction (a Session), and update data within a single replication group member. Each replication group member can be replicated as a unit, both synchronously or asynchronously or not all depending on your requirements. Each ObjectGrid client request is routed to a specific replication group member within the ObjectGrid Cluster servers. The ObjectGrid instance within the replication group member that is receiving the requests processes the request and returns a result to the client. For *synchronous* replication, each request, before returning to the client, is sent to the replica, or ObjectGrid server 2 in the following diagram, to confirm that the replica replication group member correctly applied the update, and then returns the result to the client. In *asynchronous* mode, the ObjectGrid client is able to apply a change, and the ObjectGrid servers primary replication group member returns the result to the client and does not wait for the replica to confirm that the changes were received and applied correctly. In asynchronous mode, the update will be sent to the remote server's replica replication group member after the transaction was committed successfully on the primary replication group member.

The following diagram is a different version of the bootstrap example. In this case, three servers, each having a unique role in the replication of the two ObjectGrid instances the users expect to interact with. The ObjectGrid Cluster is made of three servers, each hosting two replication group members. The server1 server hosts two primaries, the server2 server hosts two replicas, and the server 3 server hosts two standbys.

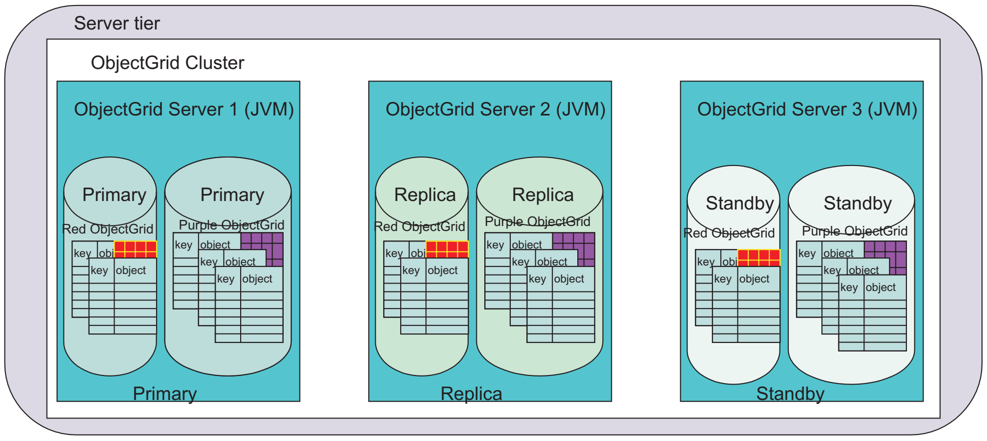


*Figure 6. Replicating the basic sample configuration*

The "High availability overview" on page 27 describes these concepts, however, a key concept to understand is the configuration differences required to move from the single server to three server replicated solution depicted in the previous diagram.

## Multi-Server replication configuration overview

Following configuration is a basic ObjectGrid configuration for the Red and Purple ObjectGrid instances.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="Red">
```

```
    <backingMap name="FirstRedMap" readOnly="false" />
    <backingMap name="SecondRedMap" readOnly="false" />
   </objectGrid>
   <objectGrid name="Purple">
    <backingMap name="FirstPurpleMap" readOnly="false" />
    <backingMap name="SecondPurpleMap" readOnly="false" />
    <backingMap name="ThirdPurpleMap" readOnly="false" />
   </objectGrid>
  </objectGrids>
</objectGridConfig>
```

Converting this configuration to a distributed ObjectGrid cluster requires an additional configuration file, the Cluster XML file. To convert the original configuration for the Red and Purple Object instances on a single server requires only the additions displayed in the following example. Specifically, only two server references were added. The replication group was already present from the initial configuration file, which cross referenced to the ColorMapsReplicationGroup replication group, as illustrated in the following sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:schemaLocation=http://ibm.com/ws/objectgrid/config/cluster ../objectGridCluster.xsd
 xmlns="http://ibm.com/ws/objectgrid/config/cluster">
 <cluster name="cluster1">
  <!- single server –>
  <serverDefinition name="server1" host="localhost" clientAccessPort="12503"
   peerAccessPort="12504" />
  <serverDefinition name="server2" host="localhost" clientAccessPort="12504"
   peerAccessPort="12506" />
  <serverDefinition name="server3" host="localhost" clientAccessPort="12507"
   peerAccessPort="12508" />
 </cluster>
 <objectgridBinding ref="Red">
  <mapSet name="RedMapSet" partitionSetRef="ColorMapsPartitioningSet">
   <map ref="FirstRedMap" />
   <map ref="SecondRedMap" />
  </mapSet>
 </objectgridBinding>
 <objectgridBinding ref="Purple">
  <mapSet name="PurpleMapSet" partitionSetRef="ColorMapsPartitioningSet">
   <map ref="FirstPurpleMap" />
   <map ref="SecondPurpleMap" />
   <map ref="ThirdPurpleMap" />
  </mapSet>
 </objectgridBinding>
 <partitionSet name="ColorMapsPartitioningSet">
  <partition name="partition1" replicationGroupRef="ColorMapsReplicationGroup" />
 </partitionSet>
 <replicationGroup name="ColorMapsReplicationGroup" maxReplicas="1"
  minReplicas="1">
  <replicationGroupMember serverRef="server1" priority="1" />
  <replicationGroupMember serverRef="server2" priority="2" /><!–New–>
  <replicationGroupMember serverRef="server3" priority="3" /><!–New->
 </replicationGroup>
</clusterConfig>
```

In the previous example, both MapSets (described below) refer to the ColorMapsReplicationGroup ReplicationGroup, which defines the servers to be included in the replication group. The configuration file could have been expanded to include another ReplicationGroup, with each of the MapSets having the same servers in different orders or different servers to meet customer requirements. ObjectGrid cluster configuration supports the reuse of stanzas. By default, because the MapSet replication attributes are not set, and the replication group has more than one server, replication is enabled and the mode is asynchronous.

## High availability overview

Replication enables high availability within an ObjectGrid cluster.

To understand replication and high availability, you must understand the ObjectGrid replication group member types. The replication group member types that ObjectGrid supports include *primary*, *replica*, and *standby*. Each of these types have a particular role in high availability configurations.

**ObjectGrid replication group member types**

**Primary replication group member**
: The primary replication group member holds the client's latest view of the data that is in use. As the data is updated, the data is propagated to the replicas. The primary is the instance that communicates with any connection database through the ObjectGrid Loader interface, propagates commits synchronously, asynchronously, or not all depending on the replication configuration.

**Replica replication group member**
: A replica replication group member holds a version of the data that has been propagated from the primary. The primary can be configured to send over the changes in various ways. The replication group must have at least two servers listed to have a primary and a replica, otherwise replication is not enabled.

**Standby replication group member**
: A standby replication group member does not receive updates as changes are made to the primary like a replica does. It simply is configured and ready to receive updates if the primary or replica fail. If the primary fails, the replica becomes the new primary and the standby needs to be converted to a replica.

### High availability scenario

In general, replication enables high availability within an ObjectGrid cluster. The two following illustrations depict primary failure scenarios and recovery. If a primary replication group member is replicated, and a failure occurs, one of the replicas is chosen to become the new primary. In this scenario one replica exists.

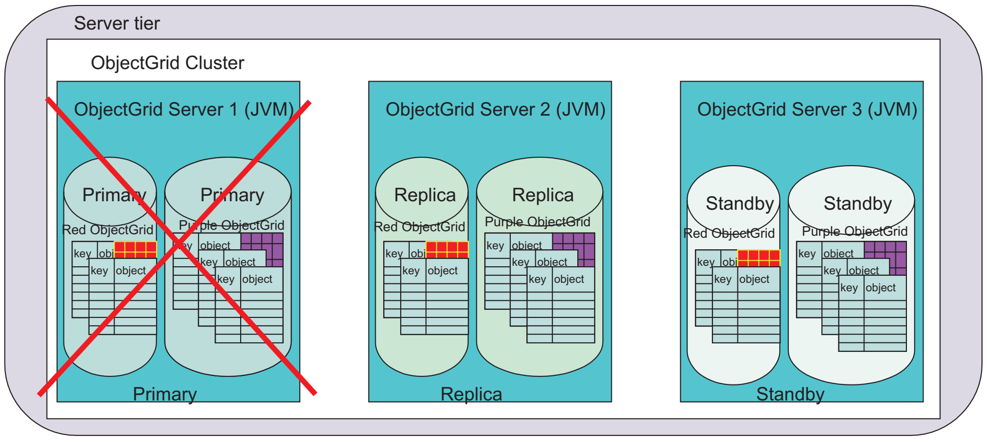


*Figure 7. ObjectGrid high availability scenario*

When a failure is detected, the primary becomes unavailable. The replica becomes the primary. If a standby exists, it becomes a Replica, similar to the example recovery in the following diagram:

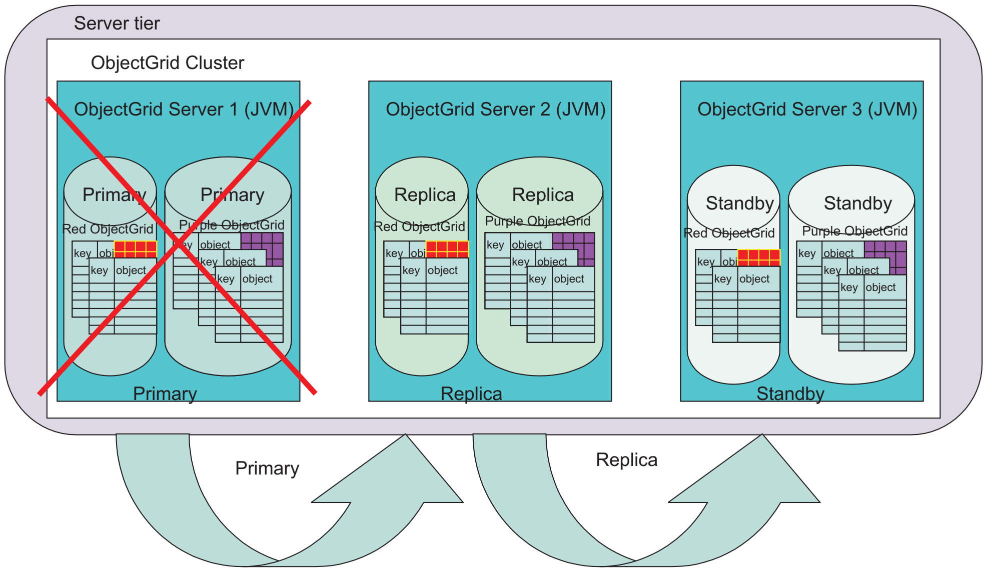


*Figure 8. ObjectGrid failover*

The ObjectGrid clients become aware of this adjustment during their next connection to any of the affected servers. Clients that contact the failed server use their runtime configuration, and can try the other servers in the cluster dynamically. The client contacts the next server in the configuration. If the server is up and not operational, the client waits for a timeout period. The clients assume the replicated replication group members are recovering from a failure. After an amount of time, the clients retry, and after the replication group, now having two members, is operational, a new routing table is provided to the client. The routing table describes where the current primary is located, its replica locations and which replication group members of this group are currently standbys.

## ObjectGrid clustering configuration sets

When configuring ObjectGrid clustering, you can separate ObjectGrids into MapSets. This separation is important because an ObjectGrid can contain many maps. MapSets can be partitioned with a PartitionSet and replicated with a ReplicationGroup. Each of these configuration options affects how many replication group members are created during the ObjectGrid server startup. A quick overview of each type of Set helps to explain the role of each type.

### ObjectGrid MapSet

Each ObjectGrid map might have different usage and availability requirements, yet is correlated by typical application usage. For example, one map might be read only with no changes after preload is complete, and another might be read-write and partitioned for scalability purposes. In this case, each map is included within a unique MapSet. In the previous example, the PurpleMapSet and RedMapSet hold all the maps for each given ObjectGrid, which is the simplest option.

A MapSet is a unit that can replicate across ObjectGrid Servers, and correlates to a replication group member that is not partitioned. Each replication group server associated to a MapSet through a PartitionSet hosts a replication group member as appropriate to support the requested configuration. A replication group member is a unique end point within the ObjectGrid Cluster, and hosts all maps a given MapSet dictates in the configuration.

For example, in the previous diagram, the server1 server has the primary, the server2 server has the replica, and the server3 server has a standby unit that can become a replica or a primary depending on the replication recovery scenarios. The Mapsets correlate to the PartitionSet that describes three servers. Therefore, both MapSets, even though they have a different number of maps in each, are mapped to the same server because the PartitionSet and ReplicationGroup stanzas are the same.

## ObjectGrid PartitionSet

Typically, the Mapset and the enumerated replication group servers determine the number of servers that support a particular MapSet in an ObjectGrid cluster. A replication group member is created within in each hosting replication group server. However, partitioning can affect the number of replication group members for a MapSet. Partitioning is managed in the configuration file through the PartitionSet relationship between the MapSet and the ReplicationGroup.

A PartitionSet divides a MapSet into portions so that one Java virtual machine (JVM) does not have hold the entire MapSet in a single primary replication group member. For example, imagine a database of 1,000,000 keys. If each object referred to by each key was large, there is a reasonable chance that a single 32-bit JVM could not hold the map in memory in a single primary, replica or standby replication group member. However, large data sets are often required. To avoid having to artificially partition the data yourself, for example to avoid manually partitioning the Purple ObjectGrid first map instance into PurpleFirstMapMap1, PurpleFirstMap2, PurpleFirstMapN maps, and put each in a different Mapset, ObjectGrid can do this work to a great extent.

Later in this document, the concept of a PartitionKey is defined. This effectively amounts to an API that ObjectGrid can invoke to determine what the key hashcode is for a particular entry during insert. If a MapSet has two partitions, then two replication group members are created for that MapSet. Often, because the data is large, these replication group members are located on different servers. For developers, these members might be on the same server during early prototyping. Each replication group member holds keys that hash to the same value across the partitioned replication group members that are available. As a simple example, assume that the MapSet was partitioned in three ways, 0, 1, and 2. Three primary replication group members are established, and one holds all keys that hash to a given value modulus the number of the primary replication group members. If a key’s hash value is 7 for example, 7 modulus 3 is 1, so the primary replication group member with the partition index of 1 would contain the instance.

## PartitionSet example

The following diagram illustrates separating the purple map into two partitions. Each key in the map is hashed to an integer, and assigned to a specific partition set upon insert into the appropriate replication group member. Each partition is in a different replication group member because it might exist in a different JVM, and ObjectGrid lets the programmer in general treat the PurpleFirstMap as a non-partitioned, logical single map instance. ObjectGrid client and server support manages routing the requests correctly between the replication group members.

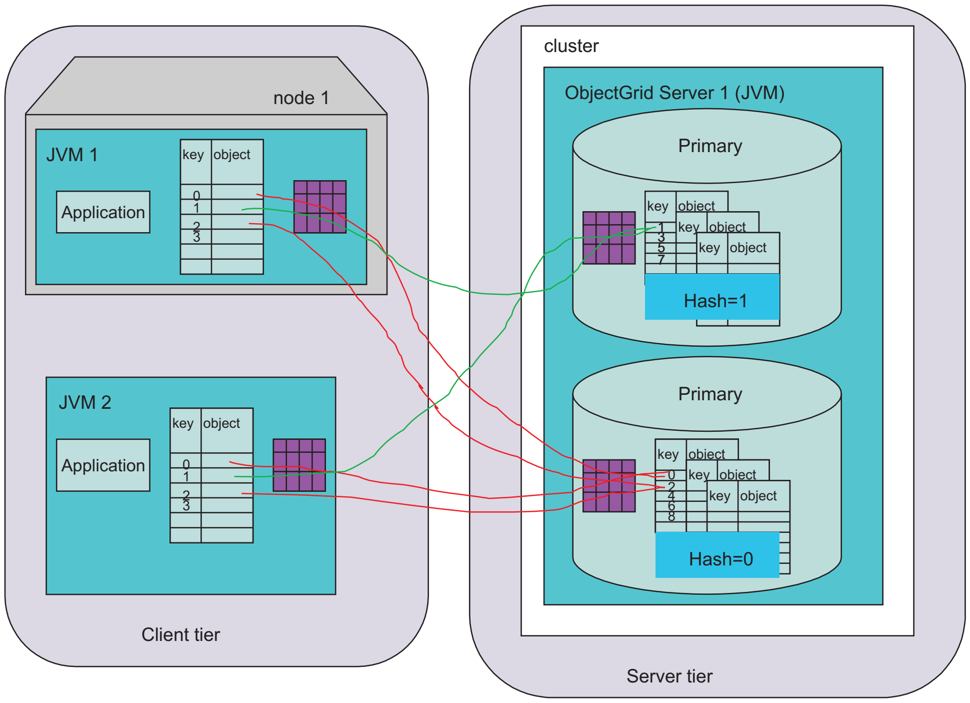


*Figure 9. Distributed ObjectGrid topology: single server with partitioning*

The partition set configuration change to enable this configuration follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:schemaLocation=http://ibm.com/ws/objectgrid/config/cluster
 ../objectGridCluster.xsd xmlns="http://ibm.com/ws/objectgrid/config/cluster">
 <cluster name="cluster1">
  <!- single server –>
  <serverDefinition name="server1" host="localhost" clientAccessPort="12503"
  peerAccessPort="12504" />
 </cluster>
 <objectGridBinding ref="Purple">
  <mapSet name="PurpleMapSet" partitionSetRef="ColorMapsPartitioningSet">
   <map ref="FirstPurpleMap" />
   <map ref="SecondPurpleMap" />
   <map ref="ThirdPurpleMap" />
  </mapSet>
 </objectGridBinding>
 <partitionSet name="ColorMapsPartitioningSet">
  <partition name="partition1" replicationGroupRef="ColorMapsReplicationGroup" />
  <partition name="partition2" replicationGroupRef="ColorMapsReplicationGroup" />
 </partitionSet>
 <replicationGroup name="ColorMapsReplicationGroup" maxReplicas="1"
  minReplicas="1">
  <replicationGroupMember serverRef="server1" priority="1" />
 </replicationGroup>
</clusterConfig>
```

The example configuration reflects how to establish a replicated, partitioned Purple ObjectGrid. In the following case three servers exist, and each of the primary replication group members are mapped in the same manner to the set of three servers. They could just as easily be mapped differently if another ReplicationGroup stanza was used.

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:schemaLocation=http://ibm.com/ws/objectgrid/config/cluster
 ../objectGridCluster.xsd xmlns="http://ibm.com/ws/objectgrid/config/cluster">
 <cluster name="cluster1">
  <!- single server –>
  <serverDefinition name="server1" host="localhost" clientAccessPort="12503"
  peerAccessPort="12504" />
  <serverDefinition name="server2" host="localhost" clientAccessPort="12504"
  peerAccessPort="12506" />
  <serverDefinition name="server3" host="localhost" clientAccessPort="12507"
  peerAccessPort="12508" />
 </cluster>
 <objectGridBinding ref="Purple">
  <mapSet name="PurpleMapSet" partitionSetRef="ColorMapsPartitioningSet">
   <map ref="FirstPurpleMap" />
   <map ref="SecondPurpleMap" />
   <map ref="ThirdPurpleMap" />
  </mapSet>
 </objectGridBinding>
 <partitionSet name="ColorMapsPartitioningSet">
  <partition name="partition1" replicationGroupRef="ColorMapsReplicationGroup" />
  <partition name="partition2" replicationGroupRef="ColorMapsReplicationGroup" />
 </partitionSet>
 <replicationGroup name="ColorMapsReplicationGroup" maxReplicas="1"
  minReplicas="1">
  <replicationGroupMember serverRef="server1" priority="1" />
  <replicationGroupMember serverRef="server2" priority="2" />
  <replicationGroupMember serverRef="server3" priority="3" />
 </replicationGroup>
</clusterConfig>
```

Notice in the previous example, again, only a small change to the configuration file yields a new level of capability and does require application modification to achieve the result. The following diagram illustrates the ObjectGrid cluster view and how the replication group members are laid out to support the previous partitioning configuration:

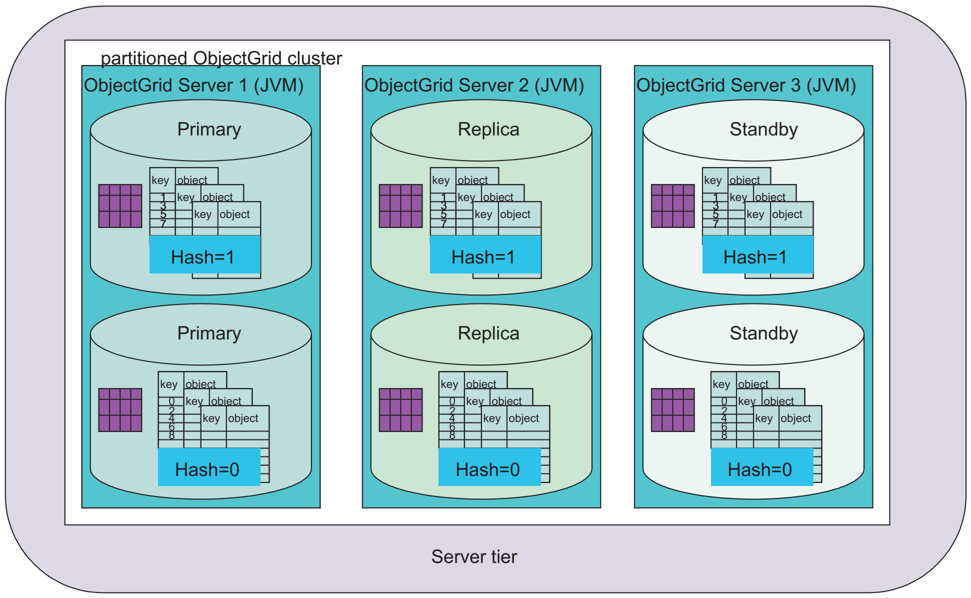

partitioned ObjectGrid cluster
ObjectGrid Server 1 (JVM)
ObjectGrid Server 2 (JVM)
ObjectGrid Server 3 (JVM)
Primary
Replica
Standby
key object
Hash=1
Hash=0
Server tier

Figure 10. Distributed ObjectGrid topology: multi-server with partitioning

To complete the PartitionSet discussion, following is another variation of the previous example. In this case, a second ReplicationGroup was created. Each PartitionSet is now included in its own replication group, on separate servers.

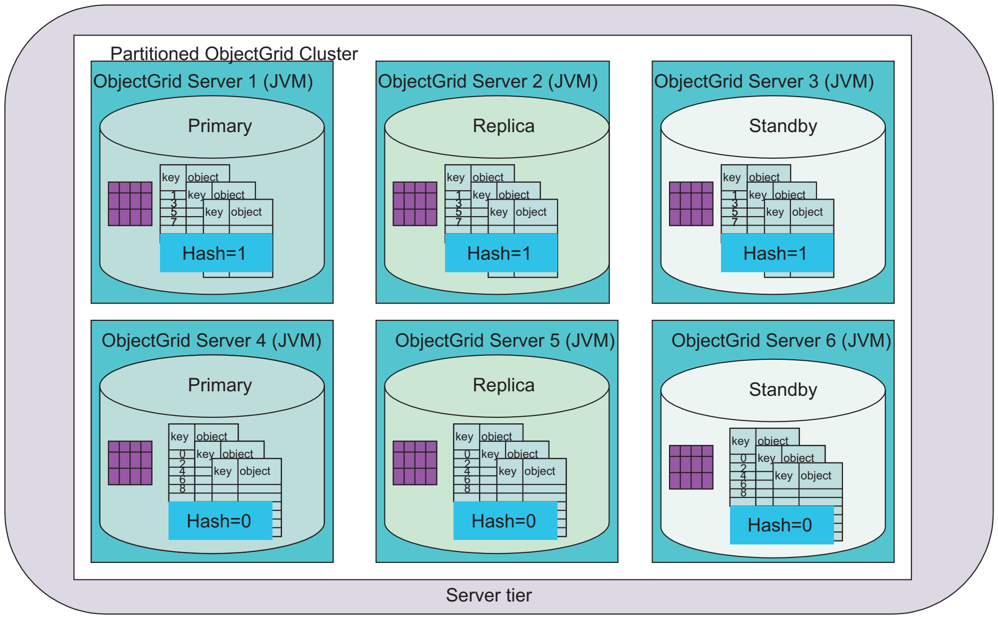

Partitioned ObjectGrid Cluster
ObjectGrid Server 1 (JVM)
Primary
Hash=1
ObjectGrid Server 2 (JVM)
Replica
Hash=1
ObjectGrid Server 3 (JVM)
Standby
Hash=1
ObjectGrid Server 4 (JVM)
Primary
Hash=0
ObjectGrid Server 5 (JVM)
Replica
Hash=0
ObjectGrid Server 6 (JVM)
Standby
Hash=0
key
object
Server tier

*Figure 11. Distributed ObjectGrid topology: multi-server with partitioning*

The configuration for this topology is very similar to the previous examples. The changes include three more server instances, and a new ReplicationGroup that is referenced by the second PartitionSet.

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:schemaLocation=http://ibm.com/ws/objectgrid/config/cluster
 ../objectGridCluster.xsd xmlns="http://ibm.com/ws/objectgrid/config/cluster">
 <cluster name="cluster1">
  <!- single server –>
  <serverDefinition name="server1" host="localhost" clientAccessPort="12503"
  peerAccessPort="12504" />
  <serverDefinition name="server2" host="localhost" clientAccessPort="12504"
  peerAccessPort="12506" />
  <serverDefinition name="server3" host="localhost" clientAccessPort="12507"
  peerAccessPort="12508" />
  <serverDefinition name="server4" host="localhost" clientAccessPort="12513"
  peerAccessPort="12514" /><!-*New*->
  <serverDefinition name="server5" host="localhost" clientAccessPort="12514"
  peerAccessPort="12516" /><!-*New*->
  <serverDefinition name="server6" host="localhost" clientAccessPort="12517"
  peerAccessPort="12518" /><!-*New*->
 </cluster>
 <objectGridBinding ref="Purple">
  <mapSet name="PurpleMapSet" partitionSetRef="ColorMapsPartitioningSet">
   <map ref="FirstPurpleMap" />
   <map ref="SecondPurpleMap" />
   <map ref="ThirdPurpleMap" />
  </mapSet>
 </objectGridBinding>
 <partitionSet name="ColorMapsPartitioningSet">
  <partition name="partition1" replicationGroupRef="ColorMapsReplicationGroup" />
  <partition name="partition2" replicationGroupRef="ColorMapsReplicationGroup" />
  <!-NEW->
 </partitionSet>
 <replicationGroup name="ColorMapsReplicationGroup" maxReplicas="1"
```

```
  minReplicas="1">
  <replicationGroupMember serverRef="server1" priority="1" />
  <replicationGroupMember serverRef="server2" priority="2" />
  <replicationGroupMember serverRef="server3" priority="3" />
 </replicationGroup>
 <replicationGroup name="ColorMapsReplicationGroupNew" maxReplicas="1"
  minReplicas="1"> <!-*NEW*->
  <replicationGroupMember serverRef="server4" priority="1" />
  <replicationGroupMember serverRef="server5" priority="2" />
  <replicationGroupMember serverRef="server6" priority="3" />
 </replicationGroup>
</clusterConfig>
```

This configuration results in the same number of replication group members, but forces the second replication group to be specifically configured through its own XML stanza, and at the same time attributes each of the instances to different server instances rather than collocating as was done in the earlier example.

The advantage of this configuration is that each partition, in this case can hold upwards of 1.5+ gigabytes of data, a total of 3 gigabytes or more because the two replication group members are in their own JVM instances. This then makes best use of 32-bit JVM 2 gigabytes of addressable memory space.

## ObjectGrid clients contacting multiple ObjectGrid clusters

ObjectGrid was specifically designed not only to scale in terms of partitioned support across Java virtual machines (JVMs), but also to extend the reach in which a normal Java Map interface can be taken to acquire information. You can contact many ObjectGrid clusters with a single client.

In the following scenario, a server tier contains two ObjectGrid clusters. One of the Java application clients and one of the application servers need to contact multiple clusters. This is a powerful feature and allows for a great deal of scalability.

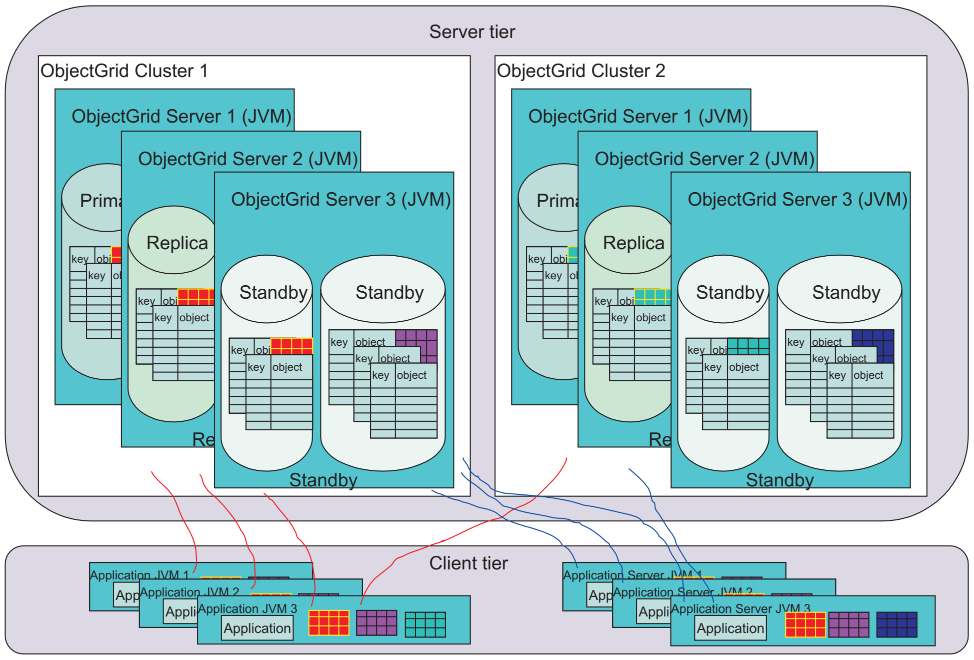


*Figure 12. ObjectGrid clients interacting with multiple ObjectGrid clusters*

An ObjectGrid Cluster can support many ObjectGrids, each containing many MapSet and PartitionSet configurations. Each ObjectGrid cluster can be made of one or more JVMs - possibly many more. For a large enterprise, the ability for the ObjectGrid Client to contact not only one, but several ObjectGrid clusters at the same time is very valuable feature.

A note of caution however, the ObjectGrid Client cannot reference data from multiple Clusters within a single transaction. The ObjectGrid Client application must run one or more transactions to cache the data in the Java virtual machine instance, and correlate the information retrieved as Java object. Updates based on the information must be also contained within in a single transaction for each cluster. See "ObjectGrid transaction demarcation" for more details on this issue.

## ObjectGrid client near caching support

The ObjectGrid client is actually a caching tier. You can design your application to leverage the local caching capability if you know if the data that was previously acquired from the remote server is not stale. For example, if the data has been updated in the ObjectGrid cluster but not in this client (this would take a new get(...) request), the client should update the local cache to be consistent (not all applications require this) with the ObjectGrid cluster.

After a get operation is performed, a subsequent request to get operation for the same key and object pair results in the ObjectGrid client detecting that the data has already been retrieved and use the ″in Java virtual machine (JVM)″ cached version instead of going across the network to the ObjectGrid cluster to access the data. After the data is retrieved once over the network, the data continues to be provided from the local cache until the local entry is evicted, manually or through a normal configured evictor.

For example, if you understand that the data on the server is refreshed once every six hours, you can control when the local cache, or near cache client update occurs. The user invalidates the near cache entry, then issue the get request. The get request contacts the server and acquires the information if all goes well. Assume the object is an image file. The first time the image is downloaded after the update window, every subsequent request does not result in a remote procedure call to the server to get the image.

The near caching support does not apply in pessimistic mode because the client might require locking on the ObjectGrid cluster data to enforce the requested locking strategy. Review the beginNoWriteThrough() method for more details on clearing or invalidating near cache entries that should be removed without modifying the ObjectGrid cluster's view of the information.

## ObjectGrid transaction demarcation

A key concept to keep in mind as a programmer is the concept of transaction demarcation. ObjectGrid does not support two phase commit protocol for transaction commit processing across replication group members in an ObjectGrid cluster. In a single ObjectGrid cluster based session, read-write updates must be applied to a single primary replication group member. If multiple replication group members were involved, it would not be possible to make the updates atomic during the transaction commit processing. This is slightly different than the local ObjectGrid JAR programming model, which allows committing against all maps in a single ObjectGrid.

A special case to take note of is a PartitionSet in which more than one partition is defined. In these scenarios, key 1 could be on server 1, key 2 on server 2, and so on. If, within one transaction there are updates to key 1 and key 2, the updates would fail because an ObjectGrid session cannot commit against two replication group members. As noted previously, a PartitionSet that supports two or more partitions must be used carefully in the application. Take care to ensure that the transaction sequence does not update data from more than a single transaction.

## ObjectGrid relationship to databases

This overview has not addressed specifically ObjectGrid servers that acquire initialized data from places other than from other Java clients. When an ObjectGrid is started, a loader is initialized for each Map in a MapSet. This loader allows user requests in the form of a Java Map get or put to be retrieved or written to the database as appropriate. The database operations are invisible to the Java Map user, and no special coding is required in their application. However, the loader functionality must be developed by a programmer and configured for use in the ObjectGrid before the end user programmer can use this function. See "Loaders" on page 191 for more details.

In addition, preload support exists to preload from a database after an ObjectGrid Cluster initializes, as well as a partitioned preload to ensure that the correct data is read for the particular replication group member partition for a given MapSet. Users can access and update the read-write information with varying locking strategies, including none, optimistic and pessimistic. Read-only data access is supported, and is the fastest model as several optimizations can be provided.

# Chapter 4. ObjectGrid tutorial : application programming model

Use this task to learn about the ObjectGrid application programming model.

Prepare your environment to run ObjectGrid applications. See Chapter 1, "Getting started with ObjectGrid by running the sample application," on page 1 to learn about the Java archive (JAR) file locations, Java requirements, and how to run a simple file to verify that your environment is set up properly.

Decide what programming environment to use for this task. You can use an integrated development environment (IDE) such as Eclipse, but the command line Java environment works also. Incorporate the ObjectGrid into enterprise beans and servlets after you are more familiar with ObjectGrid. The examples in the Tutorial do not assume any particular Java environment, so you can use any familiar environment.

At its most basic definition, ObjectGrid is a cache and an in-memory repository for objects. Using a java.util.Map map to store and access objects is similar to using ObjectGrid. At the same time, ObjectGrid is more than just a cache. By exploring the various features and plug-ins in this task, you discover that ObjectGrid is very extensible and flexible. You can use ObjectGrid as a simple *look-aside* cache or a more elaborate cache backed by a resource manager.

The examples in this tutorial are not complete programs. Imports, exception processing, and even some of the variables are not fully declared in every example. You can use the samples to write your own programs.

Use this task to use ObjectGrid from a Java program.

1. Locate the ObjectGrid APIs and exceptions. All of the public ObjectGrid APIs and Exceptions are contained in the `com.ibm.websphere.objectgrid` package. For more advanced system or configuration topics, see the additional APIs and Exceptions in the `com.ibm.websphere.objectgrid.plugins` package. Where there are provided plug-in implementations, locate those classes in the `com.ibm.websphere.objectgrid.plugins.builtins` package. For the ObjectGrid security features, look for packages with **security** in the name, such as `com.ibm.websphere.objectgrid.security`, `com.ibm.websphere.objectgrid.security.plugins`, and `com.ibm.websphere.objectgrid.security.plugins.builtins`.

   This task focuses on the APIs that are in the `com.ibm.websphere.objectgrid` package. The complete JavaDoc for ObjectGrid can be found at the following location: `<install_root>/web/xd/apidocs` .

   ```
   com.ibm.websphere.objectgrid
   com.ibm.websphere.objectgrid.plugins
   com.ibm.websphere.objectgrid.plugins.builtins
   com.ibm.websphere.objectgrid.security
   com.ibm.websphere.objectgrid.security.plugins
   com.ibm.websphere.objectgrid.security.plugins.builtins
   ```
2. Get or create an ObjectGrid instance. Use the ObjectGridManagerFactory to get the ObjectGridManager singleton instance. Then, create an ObjectGrid instance with the following statements:

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid objectGrid =
 objectGridManager.createObjectGrid("someGrid");
```

The ObjectGridManager interface has several methods for creating, retrieving, and removing ObjectGrid instances. See the “ObjectGridManager interface” on page 87 topic to choose a variation for your situation. You can also set trace settings with the ObjectGridManager interface. If you are running within WebSphere Extended Deployment or WebSphere Application Server, these methods are not necessary because trace is managed by the included facilities. If you are running outside of WebSphere Application Server, these methods can be useful. See the “Trace ObjectGrid” on page 93 topic for more complete information for these methods.

3. Initialize the ObjectGrid.
   a. Set a name for the ObjectGrid, if you did not set the name with the `create` methods.
   b. Define the BackingMaps, by using the default configuration for a BackingMap for your initial applications.
   c. After you have defined your BackingMaps, initialize the ObjectGrid. Initializing the ObjectGrid signals that all of the configuration is complete and you want to start using the ObjectGrid.
   d. After the ObjectGrid has been initialized, get a Session object. Reference “ObjectGrid interface” on page 100 and the JavaDoc for more information.

   Use the following example as guidance in this step:

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid objectGrid =
 objectGridManager.createObjectGrid("someGrid");
objectGrid.defineMap("someMap");
objectGrid.initialize();
Session session = objectGrid.getSession();
```

4. Use sessions to manage transactional operations. All access to an ObjectGrid cache is transactional: multiple accesses, inserts, updates, and removals of Objects from the cache are contained within a single unit of work, referred to as a session. At the end of a session, you can either commit all of the changes within this unit of work, or roll back and forget all of the changes within the unit of work.

   You can also use automatic commit for single atomic operations against the cache. In the absence of an active session context, individual accesses to the cache contents are enclosed in their own automatically committed sessions.

   Another important aspect of the Session interface is to get transactional access, or handle, to the BackingMap with the ObjectMap interface. You can use the getMap method to create an ObjectMap handle to a predefined BackingMap. All operations against the cache, such as inserts, updates, deletes, are completed with the ObjectMap instance. Reference the “Session interface” on page 109 topic for more information. Use the following example to obtain and manage a session:

```
Session session = objectGrid.getSession();
ObjectMap objectMap = session.getMap("someMap");
session.begin();
objectMap.insert("key1", "value1");
objectMap.insert("key2", "value2");
session.commit();
objectMap.insert("key3", "value3"); // auto-commit
```

5. Use the ObjectMap interface to access and update the cache. As you look at the ObjectMap interface, notice several methods for accessing and updating the cache. the ObjectMap interface is modeled as a map-like interface. However, checked exceptions are introduced as an aid with developing ObjectGrid applications with an IDE, such as Eclipse. If you want to use a java.util.Map interface without checked exceptions, you can use the getJavaMap method. See "ObjectMap and JavaMap interfaces" on page 113 for more information.

   Explicit insert and update methods get around the vague put operation. You can still use the put method, but using the explicit insert and update methods convey your intent much more clearly. The use of the put method is clarified by defining a put method without a preceding get operation as an insert method. If a preceding get operation is attempted before the put operation, then the put operation is treated as an insert or an update depending on whether the entry exists in the cache.

   You can perform the following basic ObjectMap operations: get, put, insert, update, remove, touch, invalidate, and containsKey. Various details and variations can be found in the "System programming model overview" on page 40 topic or the ObjectMap API documentation. The following example demonstrates the use of the ObjectMap to modify the cache:

   ```
   ObjectGridManager objectGridManager =
    ObjectGridManagerFactory.getObjectGridManager();
   ObjectGrid objectGrid =
    objectGridManager.createObjectGrid("someGrid");
   objectGrid.defineMap("someMap");
   objectGrid.initialize();
   Session session = objectGrid.getSession();
   ObjectMap objectMap = session.getMap("someMap");
   // Start a transaction/session...
   session.begin();
   objectMap.insert("key1", "value1");
   objectMap.put("key2", "value2");
   session.commit();
   // Verify changes did commit
   String value1 = (String)objectMap.get("key1");
   String value2 = (String)objectMap.get("key2");
   System.out.println("key1 = " + value1 + ", key2 = " + value2);
   //Start a new transaction/session...
   session.begin();
   objectMap.update("key2", "newValue2");
   objectMap.remove("key1");
   session.rollback();
   // Verify changes didn't commit
   String newValue1 = (String)objectMap.get("key1");
   String newValue2 = (String)objectMap.get("key2");
   System.out.println("key1 = " + newValue1 + ", key2 = " + newValue2);
   ```

6. Use the index to search for cached objects. By using the index, your applications can find objects by a specific value or a range of values. The BackingMap map must have the index plug-in configured before applications can use the index function. Applications must obtain the index object from the getIndex() method of the ObjectMap interface and cast it to the right index interface such as the MapIndex interface, the MapRangeIndex interface, or a custom index interface.

   Currently, the indexing feature is supported in the local cache only. The indexing feature is not supported in the distributed cache. If you try to perform any indexing operation against a distributed cache, the `UnsupportedOperationException` exception results.

   The following example demonstrates how to use the index:

```
MapRangeIndex myIndex = (MapRangeIndex ) objectMap.getIndex("indexName");
Object searchCriteria = "targetAttributeValue";
Iterator iter = myIndex.findAll(searchCriteria);
while (iter.hasNext()) {
    Object key = iter.next();
    System.out.println(objectMap.get(key));
}
```

As you finish reading this section and experiment with the example code, you become more comfortable with the essential ObjectGrid programming model.

For more specific information, see the Chapter 9, “ObjectGrid application programming interface overview,” on page 87.

## Getting started with remote ObjectGrid

Put your short description here; used for first paragraph and abstract.

The Chapter 4, “ObjectGrid tutorial : application programming model,” on page 37 generally dealt with a ″local″ or within an application usage of an ObjectGrid. An application created an instance of an ObjectGrid and used that instance. When the application Java virtual machine (JVM) terminated, the ObjectGrid cache also terminated. Remote ObjectGrid, as the name suggests, allows access to an ObjectGrid residing on different JVM. Multiple clients can connect to the remote ObjectGrid and access the ObjectGrid using the same API transparently.

1. Review the following sections to get started:
   - Chapter 3, “ObjectGrid overview,” on page 17
   - “ObjectGrid configuration” on page 249
   - “ObjectGrid client connect APIs” on page 94
   - Chapter 8, “Command line support,” on page 79
2. To start the servers, you must define the ObjectGrid XML file and the Cluster XML file. See “Distributed ObjectGrid configuration” on page 261. This topic refers to the `university.xml` and `universityCluster.xml` files. You can use these files as an example, modifying the host and port to launch or start the server. See Chapter 8, “Command line support,” on page 79 for details on launching ObjectGrid servers.
3. When the server is running, a client can connect to this running server. See “ObjectGrid client connect APIs” on page 94 for details on how a client can connect and perform ObjectGrid operations.

## System programming model overview

The system programming model provides several additional features and extension points for the ObjectGrid.

The following diagram illustrates how the system programming model provides several additional features and extension points.

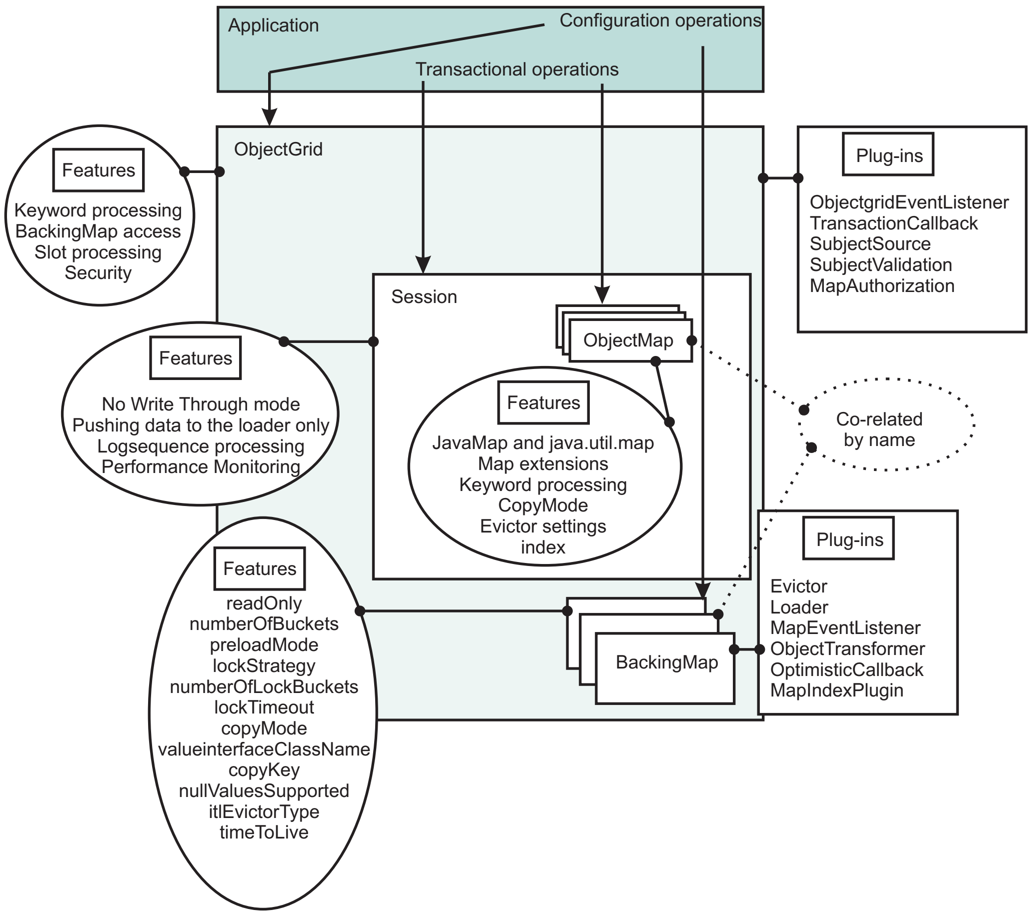


*Figure 13. ObjectGrid overview*

A plug-in in ObjectGrid is a component that provides a certain type of function to the pluggable ObjectGrid components that include ObjectGrid and BackingMap. A feature represents a specific function or characteristic of an ObjectGrid component, including ObjectGrid, Session, BackingMap, ObjectMap, and so on. If a feature represents a function, it can be used to achieve a specific computing objective. If a feature is a characteristic, it can be used to tune the behavior of the ObjectGrid components.

Each of the following sections describes some of the features and extensions that are illustrated in the preceding diagram:

- "System programming model overview: ObjectGrid interface plug points and features" on page 42

  The ObjectGrid interface has several plug points and features for more extensible interactions with the ObjectGrid.
- "System programming model overview: BackingMap interface plug points and features" on page 44

  The BackingMap interface has several optional plug points and features for more extensible interactions with the ObjectGrid.
- "System programming model overview: Session interface features" on page 52

  The Session interface has several features for more extensible interactions with the ObjectGrid. Each of the sections in this topic describe the feature and provide some brief code snippets for the usage scenario.
- "System programming model overview: ObjectMap interface features" on page 53

The ObjectMap interface has several features for more extensible interactions with the ObjectGrid. Each of the sections in this topic describe the feature and provide some brief code snippets for the usage scenario.

Fore more information about the individual features and plug-ins, see Chapter 9, “ObjectGrid application programming interface overview,” on page 87.

## System programming model overview: ObjectGrid interface plug points and features

The ObjectGrid interface has several plug points and features for more extensible interactions with the ObjectGrid.

Each of the following sections describe the feature and provide some brief code snippets for the usage scenario. Where appropriate, an XML snippet is provided to show the alternative XML configuration. For more extensive information, see the “ObjectGrid interface” on page 100 and “ObjectGrid configuration” on page 249 topics.

**Keyword processing**

The ObjectGrid interface provides a flexible invalidation mechanism that is based around keywords. A keyword is a non-null instance of any serializable object. You are free to associate keywords with BackingMap entries in any way. Most of the keyword processing is performed at the ObjectMap level, but the association of one keyword to another keyword to form a hierarchical tree of keywords is performed at the ObjectGrid level.

The associateKeyword(java.io.Serializable parent, java.io.Serializable child) method links the two keywords together in a directional relationship. If a parent is invalidated, then the child is also invalidated. Invalidating the child has no impact on the parent. For example, this method is used to add a New York map entry as a child of the USA map entry, so that if USA is invalidated then all New York entries are also invalidated. See the following code sample:

```
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
// associate several cities with "USA" keyword
objectGrid.associateKeyword("USA", "New York");
objectGrid.associateKeyword("USA", "Rochester");
objectGrid.associateKeyword("USA", "Raleigh");
:
// insert several entries with various keywords
objectMap.insert("key1", "value1", "New York");
objectMap.insert("key2", "value2", "Mexico");
objectMap.insert("key3", "value3", "Raleigh");
objectMap.insert("key4", "value4", "USA");
objectMap.insert("key5", "value5", "Rochester");
objectMap.insert("key6", "value6", "France");
:
// invalidate all entries associated with "USA" keyword, leaving
// "key2" and "key6" entries
objectMap.invalidateUsingKeyword("USA", true);
:
```

For more information, see “Keywords” on page 117.

**BackingMap access**

The ObjectGrid provides access to the BackingMap objects. You can get access to a BackingMap with either the defineMap or getMap methods. See “BackingMap interface” on page 105 for more information. The following example creates two BackingMap references:

```
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
BackingMap backingMap = objectGrid.getMap("someMap");
BackingMap newBackingMap = objectGrid.defineMap("newMap");
:
```

**Slot processing**

You can reserve a slot for storing objects that are used in the course of the transaction, such as the transaction ID object (TxID) or a database connection object (Connection). These stored objects are then referenced with a specific index, which is provided by the reserveSlot method. You can find additional information about using slots in the “Loaders” on page 191 and “TransactionCallback plug-in” on page 207 topics. The following code snippet demonstrates slot processing:

```
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
int index = objectGrid.reserveSlot
   (com.ibm.websphere.objectgrid.TxID.SLOT_NAME);
:
// Use the index later when storing or retrieving objects from
//the TxID object ...
TxID tx = session.getTxID();
tx.putSlot(index, someObject);
:
Object theTxObject = tx.getSlot(index);
:
```

**Security processing**

Maps can be protected using security mechanisms. The following methods are available on an ObjectGrid for configuring and using the security features.

- getSession(Subject)
- SubjectSource
- SubjectValidation
- AuthorizationMechanism
- MapAuthorization
- PermissionCheckPeriod

See “ObjectGrid security” on page 131 for more information on the available security mechanisms.

**ObjectGridEventListener**

The ObjectGridEventListener listener provides a way for applications to receive notification in the event of a transaction begin or commit. An instance of an ObjectGridEventListener can be set on the ObjectGrid. Reference the “Listeners” on page 177 topic for more information. Following is an example of how to implement the ObjectGridEventListener interface programmatically:

```
class MyObjectGridEventListener implements
com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener { ... }
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
objectGrid.addEventListener(new MyObjectGridEventListener());
:
```

You can also perform the same configuration with XML:

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <bean id="ObjectGridEventListner" className=
          "com.somecompany.MyObjectGridEventListener" />
  :
 </objectGrid>
</objectGrids>
:
```

**TransactionCallback plug-in**

Calling methods on the session sends corresponding events to the TransactionCallback plug-in. An ObjectGrid can have zero or one TransactionCallback plug-ins. BackingMaps that are defined on an ObjectGrid with a TransactionCallback plug-in must have a corresponding Loader. See "TransactionCallback plug-in" on page 207 for more information. The following code snippet demonstrates how to implement the TransactionCallback plug-in programmatically:

```
class MyTransactionCallback implements
com.ibm.websphere.objectgrid.plugins.TransactionCallback { ... }
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
objectGrid.setTransactionCallback(new MyTransactionCallback());
:
```

You can perform the same configuration with XML:

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <bean id="TransactionCallback" className=
         "com.somecompany.MyTransactionCallback" />
 </objectGrid>
</objectGrids>
:
```

## System programming model overview: BackingMap interface plug points and features

The BackingMap interface has several optional plug points for more extensible interactions with the ObjectGrid.

Each of the following sections describe the feature and provide some brief code snippets for the usage scenario. Where appropriate, an XML snippet is provided to show the alternative XML configuration. More extensive information is the "BackingMap interface" on page 105 and "ObjectGrid configuration" on page 249 topics or in the API documentation.

### Configuration attributes

Several configuration items are associated with BackingMaps:

- **ReadOnly** (defaults to `false`): Setting this attribute to `true` makes the backing map read-only. Setting to `false` will makes the backing map a read and write. If you do not specify a value, the default of read and write results.
- **NullValuesSupported** (defaults to `true`): Supporting null value means a null value can be put in a map. If this attribute is set to true, null values are supported in the ObjectMap; otherwise null values are not supported. If null values are supported, a `get` operation that returns null can mean that the value is null or the map does not contain the passed-in key.
- **NumberOfBuckets** (defaults to `503`): Specifies the number of buckets that are used by this BackingMap. The BackingMap implementation uses a hash map for its implementation. If many entries exist in the BackingMap then more buckets

lead to better performance because the risk of collisions is lower as the number of buckets grows. More buckets also lead to more concurrency.

- **NumberOfLockBuckets** (defaults to 383): Specifies the number of lock buckets that are used by the lock manager for this BackingMap. When the lockStrategy attribute is set to OPTIMISTIC or PESSIMISTIC, a lock manager is created for the BackingMap. The lock manager uses a hash map to keep track of entries that are locked by one or more transactions. If many entries exist in the hash map, then more lock buckets lead to better performance as the risk of collisions is lower as the number of buckets grows. More lock buckets also means more concurrency. When the lockStrategy is NONE, no lock manager is used by this BackingMap. In this case, setting the numberOfLockBuckets attribute has no effect.

### Programmatic configuration example

The following example configures properties on a backing map:

```
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
BackingMap backingMap = objectGrid.getMap("someMap");
 // override default of read/write
backingMap.setReadOnly(true);
// override default of allowing Null values
backingMap.setNullValuesSupported(false);
// override default (prime numbers work best)
backingMap.setNumberOfBuckets(251);
// override default (prime numbers work best)
backingMap.setNumberOfLockBuckets(251);
:
```

### XML configuration example

The following XML configuration example configures the same properties that are demonstrated in the preceding programmatic sample.

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <backingMap name="someMap" readOnly="true" nullValuesSupported="false"
    numberOfBuckets="251" numberOfLockBuckets="251" />
 </objectGrid>
</objectGrids>
:
```

## Lock strategy

When the lock strategy is set to OPTIMISTIC or PESSIMISTIC, a lock manager is created for the BackingMap. To prevent deadlocks from occurring, the lock manager has a default timeout value for waiting for a lock to be granted. If this timeout limit is exceeded, a LockTimeoutException exception results. The default value of 15 seconds is sufficient for most applications, but on a heavily loaded system, a timeout might occur when no actual deadlock exists. In that case, the setLockTimeout method can be used to increase the lock timeout value from the default to whatever is needed to prevent false timeout exceptions from occurring. When the lock strategy is NONE, no lock manager is used by this BackingMap. In this case, setting the lockTimeout attribute has no effect. For more information, see the “Locking” on page 123 topic.

### Programmatic configuration example

The following example sets the lock strategy:

```
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
BackingMap backingMap = objectGrid.getMap("someMap");
// override default value of OPTIMISTIC
backingMap.setLockStrategy(LockStrategy.PESSIMISTIC);
backingMap.setLockTimeout(30); // sets lock timeout to 30 seconds
:
```

### XML configuration example

The following example sets the same lock strategy that is defined in the preceding programmatic example.

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <backingMap name="someMap" lockStrategy="PESSIMISTIC" lockTimeout="30" />
 </objectGrid>
</objectGrids>
:
```

## Copy keys and values

Making copies of keys and values can be expensive, both from a resource and performance perspective. Without the capability to make these copies, strange and difficult-to-debug problems can occur. ObjectGrid has provided the ability to configure whether to and when to make copies of keys or values. Normally, keys are considered immutable so there is no need to copy the key objects. The default mode for key objects is not to make copies. Value objects are more likely to be modified by the application. When to provide a copy of the Value object versus the actual reference to the Value object is a configurable option. Reference the Chapter 11, “ObjectGrid performance best practices,” on page 315 topic and the JavaDoc for additional details on the CopyKey and CopyMode settings.

### Programmatic configuration example

Following is an example of setting the copy mode and copy key settings:

```
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
BackingMap backingMap = objectGrid.getMap("someMap");
backingMap.setCopyKey(true); // make a copy of each new key
backingMap.setCopyMode(NO_COPY); // Most efficient - trust the application
:
```

### XML configuration example

The following example results in the same configuration as in the preceding programmatic configuration example:

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <backingMap name="someMap" copyKey="true" copyMode="NO_COPY" />
 </objectGrid>
</objectGrids>
:
```

## Evictors

Evictors are used to periodically clean out unnecessary entries in the map. The entries that are removed are defined by the Evictor. The built-in Evictors are time-based, so the eviction strategy is based on the amount of time that an entry has been alive in the map. Other eviction strategies are based on usage, size, or a combination of factors.

- **Built-in Time To Live (TTL) Evictor**: The built-in Time To Live evictor provides for a couple of configuration items that are set on the BackingMap with the setTtlEvictorType and setTimeToLive methods. By default, this built-in TimeToLive evictor is not active. You can activate it by calling the setTtlEvictorType method with one of three values: CREATION_TIME, LAST_ACCESS_TIME, or NONE (default). Then, depending on the type of TimeToLive evictor selected, the value for the setTimeToLive method is used to set the lifetime for each map entry.
- **Evictor plug-ins:** In addition to the built-in Time To Live Evictor, an application can provide its own Evictor implementation plug-in. You can use any algorithm periodically to invalidate map entries.

### Programmatic configuration

The following class creates an evictor:

```
class MyEvictor implements com.ibm.websphere.objectgrid.plugins.Evictor { ... }
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
BackingMap backingMap = objectGrid.getMap("someMap");
// timer starts when entry is first created
backingMap.setTtlEvictorType(CREATION_TIME);
// Allow each map entry to live 30 seconds before invalidation
backingMap.setTimeToLive(30);
// Both builtin and custom Evictors will be active
backingMap.setEvictor(new MyEvictor()); :
```

### XML configuration

The following XML code creates a configuration that is identical to the preceding programmatic configuration:

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <backingMap name="someMap" pluginCollection="default"
    ttlEvictorType="CREATION_TIME" timeToLive="30" />
 </objectGrid>
</objectGrids>
:
<backingMapPluginCollections>
 <backingMapPluginCollection id="default">
  <bean id="Evictor" className="com.somecompany.MyEvictor" />
 </backingMapPluginCollection>
</backingMapPluginCollections>

:
```

For more information, see “Evictors” on page 182.

## Loaders

An ObjectGrid Loader is a pluggable component that enables an ObjectGrid map to behave as a memory cache for data that is normally kept in a persistent store on either the same system or another system. Typically, a database or file system is used as the persistent store. A loader has the logic for reading and writing data from and to persistent store.

A Loader is an optional plug-in for an ObjectGrid backing map. Only one Loader can ever be associated with a given backing map and each backing map has its own Loader instance. The backing map requests any data that it does not have from its Loader. Any changes to the map are pushed out to the Loader. The Loader plug-in provides a way for the backing map to move data between the map and its persistent store.

### Programmatic configuration

Following is an example of a loader implementation:

```
class MyLoader implements com.ibm.websphere.objectgrid.plugins.Loader { .. }
:
Loader myLoader = new MyLoader();
myLoader.setDataBaseName("testdb");
myLoader.setIsolationLevel("ReadCommitted");
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
BackingMap backingMap = objectGrid.getMap("someMap");
backingMap.setLoader(myLoader);
backingMap.setPreloadMode(true);
:
```

### XML configuration

The following XML sample results in the same configuration as the preceding programmatic example:

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <backingMap name="someMap" pluginCollectionRef="default" preloadMode="true" />
 </objectGrid>
</objectGrids>
:
<backingMapPluginCollections>
 <backingMapPluginCollection id="default">
  <bean id="Loader" classname="com.somecompany.MyLoader">
   <property name="dataBaseName" type="java.lang.String" value="testdb" />
   <property name="isolationLevel" type="java.lang.String" value="ReadCommitted" />
  </bean>
 </backingMapPluginCollection>
</backingMapPluginCollections>
:
```

For more information, see the “Loaders” on page 191 topic.

## MapEventListener interface

The MapEventListener callback interface is implemented by the application when it wants to receive events about a Map such as the eviction of a map entry or data preload completion. The following code example demonstrates how to set a MapEventListener instance on a BackingMap instance:

### Programmatic configuration

```
class MyMapEventListener implements
 com.ibm.websphere.objectgrid.plugins.MapEventListener { ... }
:
```

```
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
BackingMap backingMap = objectGrid.getMap("someMap");
backingMap.addMapEventListener(new MyMapEventListener() );
```

### XML configuration

The following example results in the same configuration as the preceding programmatic example:

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <backingMap name="someMap" pluginCollectionRef="default" />
 </objectGrid>
</objectGrids>
:
<backingMapPluginCollections>
 <backingMapPluginCollection id="default">
  <bean id="MapEventListener" classname="com.somecompany.MyMapEventListener" />
 </backingMapPluginCollection>
</backingMapPluginCollections>
:
```

See the “Listeners” on page 177 topic for more information.

## ObjectTransformer interface

The ObjectTransformer can be used to serialize cache entry keys and values that are not defined as serializable so that you can define your own serialization scheme without extending or implementing the Serializable interface directly. This interface also provides methods for performing the copy function on keys and values. Following is a class that implements the ObjectTransformer interface:

### Programmatic configuration

```
class MyObjectTransformer implements
 com.ibm.websphere.objectgrid.plugins.ObjectTransformer { ... }
:
ObjectTransformer myObjectTransformer = new MyObjectTransformer();
myObjectTransformer.setTransformType("full");
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
BackingMap backingMap = objectGrid.getMap("someMap");
backingMap.setObjectTransformer(myObjectTransformer);
:
```

### XML configuration

The following XML example results in the same configuration as the preceding programmatic sample:

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <backingMap name="someMap" pluginCollectionRef="default" />
 </objectGrid>
</objectGrids>
:
<backingMapPluginCollections>
 <backingMapPluginCollection id="default">
  <bean id="ObjectTransformer" className="com.somecompany.MyObjectTransformer">
   <property name="transformType" type="java.lang.String" value="full"
     description="..." />
  </bean>
 </backingMapCollection>
</backingMapCollections>
:
```

For more information, see the "ObjectTransformer plug-in" on page 202 topic.

## OptimisticCallback interface

The OptimisticCallback interface can be used to create and process a version field that is associated with a given Value object. In many cases, using the Value object directly to determine if another cache client has modified the value since it was retrieved is very inefficient and error-prone. An alternative is to provide another field that represents the state of the Value object. The intent of the OptimisticCallback interface is to provide an alternative Versioned Value object that represents the Value object. Following is a sample configuration of the OptimisticCallback interface:

### Programmatic configuration

```
class MyOptimisticCallback implements
 com.ibm.websphere.objectgrid.plugins.OptimisticCallback { ... }
:
OptimisticCallback myOptimisticCallback = new MyOptimisticCallback();
myOptimisticCallback.setVersionType("Integer");
backingMap.setOptimisticCallback(myOptimisticCallback);
:
```

### XML configuration

The following example results in the same configuration as the preceding programmatic example:

```
:
<objectGrids>
 <objectGrid name="someGrid">
  <backingMap name="someMap" pluginCollectionRef="default" />
 </objectGrid>
</objectGrids>
:
<backingMapPluginCollections>
 <backingMapPluginCollection id="default">
  <bean id="OptimisticCallBack" classname="com.somecompany.MyOptimisticCallback">
   <property name="versionType" type="java.lang.string" value="Integer"
    description="..." />
  </bean>
 </backingMapPluginCollection>
</backingMapPluginCollections>
:
```

## Indices

A MapIndexPlugin, or an Index in short, is an option that is used by the BackingMap to build index based on the specified attribute of the stored object. The index allows applications to find objects by a specific value or a range of values. To use the index, Applications have to obtain the index object from the getIndex() method of the ObjectMap interface and cast it to the right index interface such as MapIndex or MapRangeIndex or a custom index interface.

Currently, the indexing feature is only supported in the local cache, not the distributed cache. If trying to perform any indexing operation against a distributed cache, the UnsupportedOperationException will be raised.

There are two types of index: static and dynamic index. Static indices can be created via both Programmatic configuration and XML configuration. Dynamic indices can only be created programmatically.

**Programmatic configuration**

The following code example demonstrates how to add static index into a BackingMap instance:

```
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("indexSampleGrid");
BackingMap personBackingMap= objectGrid.getMap("person");

//use the builtin com.ibm.websphere.objectgrid.plugins.index.HashIndex
//class as the index plugin class.
HashIndex mapIndexPlugin = new HashIndex();
mapIndexPlugin.setName("CODE");
mapIndexPlugin.setAttributeName("EmployeeCode");
mapIndexPlugin.setRangeIndex(true);
personBackingMap.addMapIndexPlugin(mapIndexPlugin);
//Note: the previous Index configuration assumes that the stored object has
// an attribute named EmployeeCode and a method named getEmployeeCode()
//that returns the value of the EmployeeCode attribute.
:
```

The following code example demonstrates how to create dynamic index on a BackingMap instance:

```
class DynamicIndexCallbackImpl implements
com.ibm.websphere.objectgrid.plugins.index.DynamicIndexCallback { ... }
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("indexSampleGrid");
BackingMap personBackingMap= objectGrid.getMap("person");
objectGrid.initiallize();
:
//insert, update, or remove data
//Dynamic index can be created after the containing ObjectGrid instance has
//been initialized
//If there is a need to create a dynamic index, create it without
//DynamicIndexCallback
personBackingMap.createDynamicIndex("CODE2", true, "employeeCode", null);
:
//Another option is to create dynamic index with DynamicIndexCallback
//Assuming there is a DynamicIndexCallbackImpl class implements
//DynamicIndexCallback interface
personBackingMap.createDynamicIndex("CODE3", true, "employeeCode",
new DynamicIndexCallbackImpl());
:
```

**XML configuration**

The following example results in the same configuration as the preceding programmatic example of static index:

```
:
<objectGrids>
 <objectGrid name="indexSampleGrid">
  <backingMap name="person" pluginCollectionRef="person" />
 </objectGrid>
</objectGrids>
<backingMapPluginCollections>
 <backingMapPluginCollection id="person">
  <bean id="MapIndexPlugin
  className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
   <property name="Name" type="java.lang.String" value="CODE"
    description="index name" />
   <property name="RangeIndex" type="boolean" value="true"
    description="true for MapRangeIndex />
   <property name="AttributeName" type="java.lang.String"
    value="employeeCode" description="attriubte name" />
  </bean>
 </backingMapPluginCollection>
</backingMapPluginCollections>
:
```

See the Indexing topic for more information.

## System programming model overview: Session interface features

The Session interface has several features for more extensible interactions with the ObjectGrid. Each of the following sections describe the feature and provide some brief code snippets for the usage scenario.

For more information about the Session interface, see “Session interface” on page 109.

### No write through mode

Sometimes, applications just want to apply changes to the base map but not the Loader. The beginNoWriteThrough method of Session interface is designed to achieve this objective. The isWriteThroughEnabled method of Session interface can be used to verify if the current session is writing to the back end Loader. This might be useful to other users of the Session object to know what type of session is currently being processed. The following example enables the no write through mode:

```
:
ObjectGrid objectGrid = objectGridManager.createObjectGrid("someGrid");
objectGrid.defineMap("someMap");
objectGrid.initialize();
Session session = objectGrid.getSession();
session.beginNoWriteThrough();
boolean isWriteThroughEnabled = session.isWriteThroughEnabled();
// make updates to the map ...
session.commit();
:
```

### Push data to the Loader only

Applications can apply local changes in the session to the Loader without committing these changes permanently by invoking the flush method, as in the following example:

```
:
Session session = objectGrid.getSession();
session.begin();
// make some changes ...
session.flush(); // push these changes to the Loader, but don't commit yet
// make some more changes ...
session.commit();
:
```

### processLogSequence method

The processLogSequence method is used to process a LogSequence. Each LogElement within the LogSequence is examined and the appropriate operation, such as the insert, update, invalidate operations, is performed against the BackingMap identified by the LogSequence MapName. An ObjectGrid Session must be active before this method is invoked. The caller is then responsible for issuing the appropriate commit or rollback calls to complete the Session. Autocommit processing is not available for this method invocation.

The main use of this method is to process a LogSequence that was received by a remote JVM. For example, using the distributed commit support, the LogSequences that are associated with a given committed session are then distributed to other listening ObjectGrids in other Java virtual machines (JVM). After receiving the LogSequences at the remote JVM, the listener can start a Session using the beginNoWriteThrough method, invoke this processLogSequence method, and then perform the commit method on the Session. An example follows:

```
:
session.beginNoWriteThrough();
try {
 session.processLogSequence(inputSequence);
}
catch (Exception e) {
 session.rollback();
 throw e;
}
session.commit();
:
```

### Performance monitoring

Maps can optionally be instrumented for performance monitoring while running within WebSphere Application Server. The `setTransactionType` method is available on a Session for configuring and using the performance monitoring features. See “Monitoring ObjectGrid performance with WebSphere Application Server performance monitoring infrastructure (PMI)” on page 283 for more information.

## System programming model overview: ObjectMap interface features

The ObjectMap interface has several features for more extensible interactions with the ObjectGrid.

For more information about the ObjectMap interface, see “ObjectMap and JavaMap interfaces” on page 113.

### The JavaMap interface and the java.util.Map interface

For applications that want to use the java.util.Map interface, the ObjectMap has the getJavaMap method, so that applications can get the implementation of the java.util.Map interface that is backed by the ObjectMap. The returned Map instance can then be cast to the JavaMap interface, which extends the java.util.Map interface. The JavaMap interface has the same method signatures as ObjectMap, but with different exception handling. The JavaMap interface extends the java.util.Map interface, so all exceptions are instances of the java.lang.RuntimeException class. Because the JavaMap interface extends the java.util.Map interface, it is easy to quickly use ObjectGrid with an existing application that uses a java.util.Map interface for object caching. A code snippet follows:

```
:
JavaMap javaMap = (JavaMap)objectMap.getJavaMap();
:
```

## Map extensions

The ObjectMap interface also provides additional functional capabilities in addition to the checked exceptions capabilities. For example, a user can specify that a given map entry is updated with the getForUpdate method, which indicates to the ObjectGrid runtime and Loader plug-in that the entry can be locked during the processing, if appropriate. Batch processing is another additional capability with the getAll, putAll, and removeAll methods. For more information about these methods, see the API documentation.

## Keyword processing

Most map operations have the keyword parameter version, such as insert, get, getForUpdate, put, remove, and invalidate. For ease of use, the setDefaultKeyword method is also provided. This method associates entries with a keyword without using the keyword version of the map operation. A keyword example follows:

```
:
// setDefaultKeyword
session.begin();
objectMap.setDefaultKeyword("New York");
Person p = (Person) objectMap.get("Billy"); // "Billy" entry has "New York" keyword
p = (Person) objectMap.get("Bob", "Los Angeles"); // "Bob" entry
//has "Los Angeles" keyword
objectMap.setDefaultKeyword(null);
p = (Person) objectMap.get("Jimmy"); // "Jimmy" entry has no keyword
session.commit();
:
// keyword parameter version of insert operation
session.begin();
Person person = new Person("Joe", "Bloggs", "Manhattan");
objectMap.insert("BillyBob", person, "Rochester"); // "BillyBob" has
//"Rochester" keyword
session.commit();
:
```

See “Keywords” on page 117 for more information.

## Copy mode method

The setCopyMode method allows the copy mode for the Map to be overridden on this map for this session or transaction only. This method allows an application to use an optimal copy mode on a per session basis, as its needs dictate. The copy mode cannot be changed during an active session. A corresponding clearCopyMode method exists that resets the copy mode back to the one defined on the BackingMap. You can call this method only when no active sessions exist.An example of setting the copy mode follows:

```
:
objectMap.setCopyMode(CopyMode.COPY_ON_READ, null);
session.begin();
// modify objectMap ...
session.commit();
objectMap.clearCopyMode(); // reset CopyMode to BackingMap setting
session.begin();
// modify objectMap ...
session.commit();
:
```

For more information, see the “ObjectTransformer plug-in” on page 202 and Chapter 11, “ObjectGrid performance best practices,” on page 315 topics.

## Evictor settings

You can override the TimeToLive timeout value for the built-in TimeToLive evictor at the ObjectMap level. The setTimeToLive method establishes the number of seconds that any given cache entry can live. When modified, the previous TimeToLive value is returned. This TimeToLive value is the minimum time an entry remains in the cache before being considered for eviction and indicates to the built-in TimeToLive evictor how long an entry should remain after last access time. The new TimeToLive value only applies to ObjectMap entries that are accessed by the transaction started by the Session object that was used to obtain the ObjectMap instance. The new TimeToLive value applies to any transaction that is in progress for the Session and future transactions that are run by the Session. The new TimeToLive value does not affect entries of an ObjectMap instance that are accessed by a transaction started by some other Session. By calling this method on the ObjectMap, any previous value set by the setTimeToLive method on the BackingMap is overridden for this ObjectMap instance. An example follows:

```
:
session.begin();
int oldTTL = objectMap.setTimeToLive(60); // set TTL to 60 seconds
Person person = new Person("Joe", "Bloggs", "Manhattan");
objectMap.insert("BillyBob", person); // "BillyBob" entry will have a TTL
//of 60 seconds
session.commit();
:
objectMap.setTimeToLive(oldTTL); // reset TTL to original value
Person person2 = new Person("Angelina", "Jolie", "somewhere");
objectMap.insert("Brad", person2); // "Brad" entry will use original TTL value
:
```

For more information, see the “Evictors” on page 182 topic.

# Chapter 5. ObjectGrid samples

This topic describes the ObjectGrid samples that are provided when installing the WebSphere Extended Deployment product.

## Overview

Several ObjectGrid samples illustrate the integration with Java 2 Platform, Enterprise Edition (J2EE) applications and the partitioning facility (WPF). This topic describes each of the samples, the features that each sample demonstrates, the location of each sample, and the environments where the sample runs.

This topic describes samples that are provided when installing WebSphere Extended Deployment. Other samples will be provided related to using Java Message Service (JMS) integration and integration of ObjectGrid with other open source frameworks at the following Web address:http://www-1.ibm.com/support/docview.wss?uid=swg27006432 .

## Samples

- **ObjectGridSamplesSA :** This sample is a set of Java 2 Platform, Standard Edition (J2SE) examples that are packaged in the `objectgridSamples.jar` file for demonstrating the ObjectGrid functions. These J2SE samples can be run in a J2SE environment. The `objectgridSamples.jar` file contains the `SamplesGuide.htm` file, which has instructions for running these samples.
- **ObjectGridSample :** This sample is a J2EE example that demonstrates how servlets and Session enterprise beans use the ObjectGrid functions. This sample is shipped in the `ObjectGridSample.ear` enterprise archive (EAR) file. The `ObjectGridSample.ear` file contains the `readme.txt` file, which has instructions for setting up and running this sample.
- **ObjectGridPartitionCluster :** This sample is a J2EE sample for demonstrating how the WPF and ObjectGrid work together and how to use the ObjectGridEventListener to propagate object changes and how to enable context-based routing to maintain ObjectGrid integrity and consistency. This sample is shipped in the `D_ObjectGridPartitionClusterSample.ear` EAR file. The `D_ObjectGridPartitionClusterSample.ear` file contains the `readme.txt` file, which has instructions for setting up and running this sample.
- **ObjectGridJMSSamples:** This is a set of J2EE samples packaged in the ObjectGridJMSSamples.zip file that demonstrate how to use JMS function to transmit changes in one ObjectGrid instance to another ObjectGrid instance in a single JVM or a cluster environment. These J2EE samples are only available on the Web at the following Web address:http://www-1.ibm.com/support/docview.wss?uid=swg27006432 .

## Sample functionality

*Table 2. Sample functionalities*

| Functional Area | ObjectGrid SamplesSA sample | ObjectGrid Sample sample | ObjectGridPartition Cluster sample | ObjectGrid JMSSamples sample |
|---|---|---|---|---|
| ObjectGrid EventListener | | | x | x |
| Transaction callback | x | x | x | |

*Table 2. Sample functionalities (continued)*

| Functional Area | ObjectGrid SamplesSA sample | ObjectGrid Sample sample | ObjectGridPartition Cluster sample | ObjectGrid JMSSamples sample |
|---|---|---|---|---|
| Loader | x | x | x | |
| MapEvent Listener | x | | | |
| Object Transformer | x | x | x | x |
| Optimistic callback | x | x | x | |
| BackingMap copy mode | x | x | | |
| Distributed invalidation | | | x | x |
| Distributed update | | | x | x |
| LogSequence processing | | | | x |
| Partitioning facility (WPF) | | | x | |
| Java Message Service (JMS) | | | | x |
| Map index | x | | | |
| ObjectGrid cluster | x | x | | |
| ObjectGrid ClusterClient Context | x | x | | |
| Distributed ObjectGrid | x | x | | |
| ObjectGrid management | x | | | |
| ObjectGrid security | x | | x | |

## Location

After WebSphere Extended Deployment has been installed, the following .jar files are located in the following directories:

*Table 3. Sample locations*

| Sample | Location |
|---|---|
| ObjectGridSamplesSA | `install_root\optionalLibraries\ObjectGrid\objectgridSamples.jar` |
| ObjectGridSample | `install_root\installableApps\ObjectGridSample.ear` |
| ObjectGridPartitionCluster | `install_root\installableApps\D_ObjectGridPartitionClusterSample.ear` |

Updated versions of the listed shipped samples, and additional samples such as ObjectGridJMSSamples, can be found on the Web at the following Web address:http://www-1.ibm.com/support/docview.wss?uid=swg27006432 . You can also find articles on IBM DeveloperWorks that describe topics of interest at the following Web address: http://www.ibm.com/developerworks. Search for **ObjectGrid**.

## Sample environments

Some samples can run in a J2SE environment, but some have to run in a J2EE environment. Some can run in a single server instance, others have to run in a cluster. The following table shows the running environment of the samples.

**Restriction:** If you are using ObjectGrid in a WebSphere Extended Deployment Version 6.0 environment, you can also use ObjectGrid in a Java 2 Platform, Standard Edition (J2SE) Version 1.4.2 or higher environment or in a WebSphere Application Server Version 6.02 or higher environment with additional licensing arrangements. Contact your sales representative for details.

*Table 4. Sample running environments*

| | | **ObjectGrid SamplesSA** | **ObjectGrid Sample** | **ObjectGrid Partition Cluster** | **ObjectGrid JMSSamples** |
|---|---|---|---|---|---|
| J2SE | Eclipse | x | | | |
| | command line | x | | | |
| WebSphere Application Sever Version 6.0.*x* | single server | | x | | x |
| | cluster | | x | | x |
| | Rational Application Developer unit test environment (UTE) | | x | | |
| WebSphere Application Server Version 5.0.2.*x* and Version 5.1.*x* | single server | | x | | |
| | cluster | | x | | |
| WebSphere Extended Deployment Version 6.0.*x* | single server | | x | | x |
| | cluster | | x | x | x |

# Chapter 6. ObjectGrid packaging

You can access the ObjectGrid packages in two ways: by installing WebSphere Extended Deployment, or by installing the mixed server environment.

## WebSphere Extended Deployment Version 6.0.1 ObjectGrid package

When you install WebSphere Extended Deployment Version 6.0.1 or later, the following runtime files are installed:

*Table 5. WebSphere Extended Deployment ObjectGrid runtime files*

| File name | Runtime environment | Description |
|---|---|---|
| /lib/asm.jar<br><br>/lib/cglib.jar | Local, client and server | These jars are for the cglib utility function when using the copy on write copy mode. |
| /lib/wsobjectgrid.jar | Local, client and server | This Java archive (JAR) file contains the ObjectGrid local, client and server runtime for use in the WebSphere Extended Deployment Version 6.0.1 and later environment. |

## WebSphere Extended Deployment for Mixed Server Environment Version 6.0.1 ObjectGrid package

When you install WebSphere Extended Deployment for Mixed Server Environment, the following runtime files are installed:

*Table 6. WebSphere Extended Deployment for Mixed Server Environment ObjectGrid runtime files*

| File name | Runtime environment | Description |
|---|---|---|
| /ObjectGrid/lib/asm.jar<br><br>/ObjectGrid/lib/cglib.jar | Local, client and server | These JAR files are for the cglib utility function when you are using the copy on write copy mode. Include these JAR files in your CLASSPATH if you are using the copy on write copy mode and you want to use the cglib proxy function. These JAR files are automatically included in the server runtime. Add these files to your client or local ObjectGrid runtime. |

Table 6. WebSphere Extended Deployment for Mixed Server Environment ObjectGrid runtime files (continued)

| File name | Runtime environment | Description |
|---|---|---|
| /ObjectGrid/lib/mx4j.jar<br>/ObjectGrid/lib/mx4j-remote.jar<br>/ObjectGrid/lib/mx4j-tools.jar | Management gateway client and server | These JAR files are for the mx4j utility function that is used by the ObjectGrid management gateway server as well as the management gateway client programs. Add these JAR files to the management gateway client CLASSPATH when you are connecting to the management gateway server. |
| /ObjectGrid/lib/objectgrid.jar | Local, client and server | This JAR file is used by the standalone server runtime for Java 2 Platform, Standard Edition (J2SE) Version 1.4.2 and later. You can also use this JAR file for client and local runtime for J2SE version 1.3 and later. |
| /ObjectGrid/lib/ogclient.jar | Local and client | This JAR file contains only the local and client ObjectGrid runtimes when running outside of a WebSphere process. It may be more desireable to use this JAR file over objectgrid.jar due to the smaller footprint. You can use this jar with J2SE version 1.3 and later. |
| /ObjectGrid/lib/wsobjectgrid.jar | Local, client and server | Use this JAR file on WebSphere Application Server Version 6.0.2 or later. This JAR file is the same JAR file that is installed with WebSphere Extended Deployment. |
| /ObjectGrid/lib/wsogclient.jar | Local and client | Use this JAR file for WebSphere Application Server Version 5.0.2 and later. This JAR file contains only the local and client ObjectGrid runtimes. |

## Considerations for using ObjectGrid with J2SE Version 1.3

When using the ogclient.jar or objectgrid.jar file in a J2SE Version 1.3.x environment, you must add the following requirements to your J2SE 1.3.x environment to make it functional with ObjectGrid:

- **Java Authentication and Authorization Service (JAAS) implementation**. J2SE Version 1.3 did not include the javax.security.Subject object, a part of the

JAAS specificiation. The ObjectGrid and Session interfaces require this object. Place the JAAS implementation in the jre/lib/ext Java extensions directory.

- **Java API for XML Processing (JAXP) implementation**. If you are passing XML files to the ObjectGrid runtime, a JAXP implementation is necessary for the ObjectGrid runtime to parse the XML file. ObjectGrid uses XML schema definition syntax validation so an implementation that supports schema validation is required. The Apache Xerces product is an example of an implementation that supports schema validation.
- **Java Secure Socket Extension (JSSE) implementation**. When you are using the client runtime, a JSSE implementation is required. Verify that the JSSE implementation used is compatible with the ObjectGrid server Java Development Kit (JDK) implementation if you are running with security enabled.

If your local or client ObjectGrid runtime is included in a J2EE Version 1.3 compatible environment that uses J2SE Version 1.3, all of these requirements are met because all the required specification implementations were required as part of J2EE Version 1.3.

# Chapter 7. System management overview

With the release of WebSphere Extended Deployment Version 6.0.1, ObjectGrid provides a system management infrastructure to allow users to monitor and administer ObjectGrid environments. The system management architecture is a three-tiered approach: a user client connects to the Management Gateway server, which makes an ObjectGrid client connection to an ObjectGrid cluster.

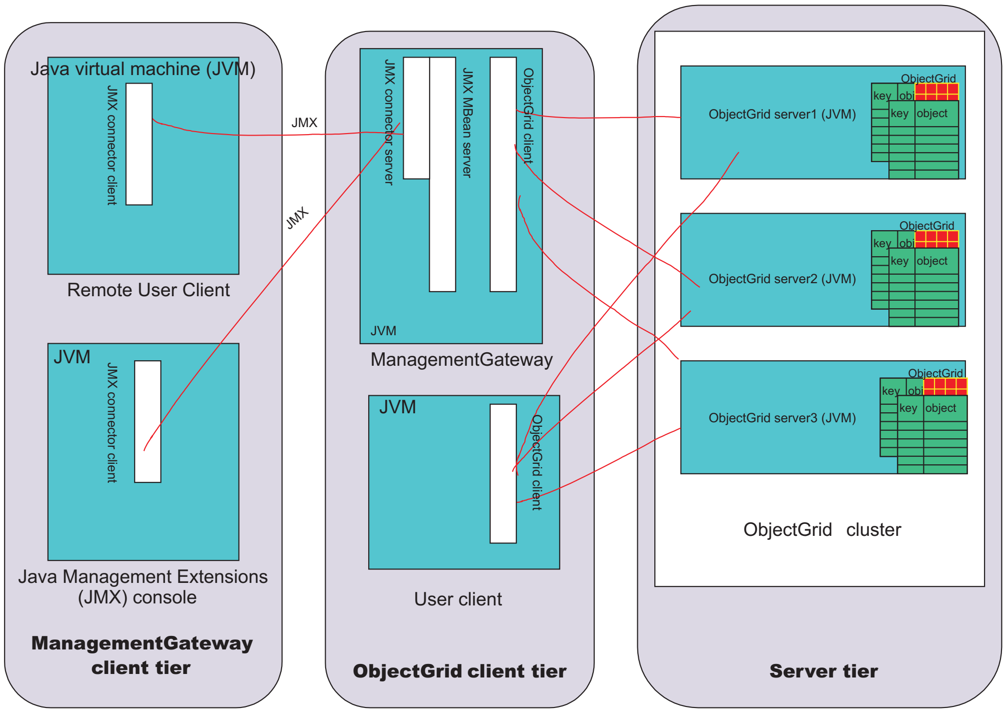


*Figure 14. System management diagram*

The *management gateway client tier* contains any program that uses Java Management Extensions (JMX) to connect to the Management Gateway server. Any third-party JMX console as well as a client program that uses MX4J APIs are included. The *ObjectGrid client tier* consists of the management gateway server. The management gateway acts as a server for the management gateway client tier and as a client to a functioning ObjectGrid cluster in the *server tier*. Also, an ObjectGrid client program can call the same APIs that the management gateway server calls if the user does not want to involve JMX. Finally the Server Tier consists of an ObjectGrid cluster.

The management gateway houses a set of managed beans (MBeans) and uses JMX to administer and monitor the ObjectGrid environment and is implemented by the MX4J open source project . MX4J is shipped with ObjectGrid.

The ObjectGrid JMX and MBean administration model was created to take advantage of the various JMX consoles that are available for administering JMX environments. You can put together dashboards using the JMX console of your choice. Consoles can be attached to the MBeans running on the

ManagementGateway Java virtual machine (JVM) and dashboards can be assembled using these MBeans. Consoles offer graphical histories or charts of numerical and string values.

There are two options for executing system management commands.

- Call any command through the client-server infrastructure currently in place using the ObjectGridAdministrator interface.
- Use JMX to call these same commands, with the ObjectGrid MBeans acting as a wrapper to the ObjectGridAdministrator.

## Start the ManagementGateway process

After a cluster (or single server) is started, the ManagementGateway process can be started. The ManagementGateway acts as a server to user client requests, and an ObjectGrid client to the cluster to which it is connected.

### Options

Following is list of options that can be passed to the ManagementGateway process:

- **connectorPort** (required) - Specifies the port number for the JMX connector.
- **clusterHost** (required) - Specifies the host name of one of the servers in the ObjectGrid cluster.
- **clusterPort** (required) - Specifies the client access port of one of the servers in the ObjectGrid cluster.
- **clusterName** (required) - Specifies the name of the ObjectGrid cluster.
- **traceEnabled** - Specifies if trace is enabled for the ManagementGateway process.
- **traceSpec** - Indicates the trace specification of the ManagementGateway.
- **traceFile** - Specifies the file to which trace output is printed.
- **sslEnabled** - Specifies if SSL is enabled on the ManagementGateway.
- **csConfig** - Specifies the ClientSecurityConfiguration object for secure ManagementGateway.
- **refreshInterval** - Specifies the time interval at which the management gateway refreshes the MBean attributes.

### ManagementGateway interface

The ManagementGateway process needs to be started to make the MBeans available. The ManagementGateway interface shows what options can be passed in when starting the ManagementGateway.

```
public interface ManagementGateway {

    /**
     * Start the JMX MBean connector server
     */
    void startConnector();

    /**
     * Stop the JMX MBean connector server
     */
    void stopConnector();

    /**
     * @param JMX connector port
     */
```

```
void setConnectorPort(int port);
/**
 * @return JMX connector port
 */
int getConnectorPort();

/**
 * @param a {@link com.ibm.websphere.objectgrid.security.config.
ClientSecurityConfiguration} object.
 */
void setCsConfig(ClientSecurityConfiguration csConfig);

/**
 * @return a {@link com.ibm.websphere.objectgrid.security.config.
ClientSecurityConfiguration} object.
 */
ClientSecurityConfiguration getCsConfig();

/**
 * @param port of server to which gateway client connects
 */
void setPort(String port);

/**
 * @return port of server to which gateway client connects
 */
String getPort();

/**
 * @param host of server to which gateway client connects
 */
void setHost(String host);

/**
 * @return host of server to which gateway client connects
 */
String getHost();

/**
 * @param boolean true if SSL enabled on gateway
 */
void setSSLEnabled(boolean sslEnabled);

/**
 * @return boolean true if SSL enabled on gateway
 */
boolean getSSLEnabled();

/**
 * @param cluster to which gateway client connects
 */
void setClusterName(String clusterName);

/**
 * @return cluster to which gateway client connects
 */
String getClusterName();

/**
 * @param true if trace is enabled on gateway
 */
void setTraceEnabled(boolean traceEnabled);

/**
 * @return true if trace is enabled on gateway
 */
boolean getTraceEnabled();
```

```
    /**
     * @param trace specification on gateway
     */
    void setTraceSpec(String traceSpec);

    /**
     * @return trace specification on gateway
     */
    String getTraceSpec();

    /**
     * @param trace output file for gateway trace
     */
    void setTraceFile(String traceFile);

    /**
     * @return trace output file for gateway trace
     */
    String getTraceFile();

    /**
     * @param interval (in seconds) to refresh cluster MBean attributes
     */
    void setRefreshInterval(int refreshInterval);

    /**
     * @return interval (in seconds) to refresh cluster MBean attributes
     */
    int getRefreshInterval();

}
```

## Options for starting the ManagementGateway process

**Programmatically using the ManagementGatewayFactory**

Here is sample code for using this option:

```
ManagementGateway gw = ManagementGatewayFactory.getManagementGateway();
gw.setConnectorPort(1099);
gw.setClusterName("cluster1");
gw.setHost("localhost");
gw.setPort("12503");
gw.startConnector();
```

This code should be in a user program that is run after the ObjectGrid cluster that you are trying to connect to has been started.

**On the command line with the startManagementGateway batch file**

An example follows:

```
 startManagementGateway.bat -connectorPort 1099 -clusterName cluster1
    -clusterHost localhost -clusterPort 12503
```

For more information about the startManagementGateway scripts, see "Start the management gateway server" on page 84.

The ManagementGateway acts as a server for a client process that wants to make JMX calls, but also as an ObjectGrid client to the cluster to which the user wants to connect. After the ManagementGateway is started, a connection is established to the cluster and the JMX Connector service becomes available. You can then access the JMX Connector service through MX4J or Java 2 Platform, Standard Edition (J2SE) Version 5 APIs.

## Example

Here is sample code of how to get a MapStatsModule from a server called Server1 through a ManagementGateway with connector port 1.

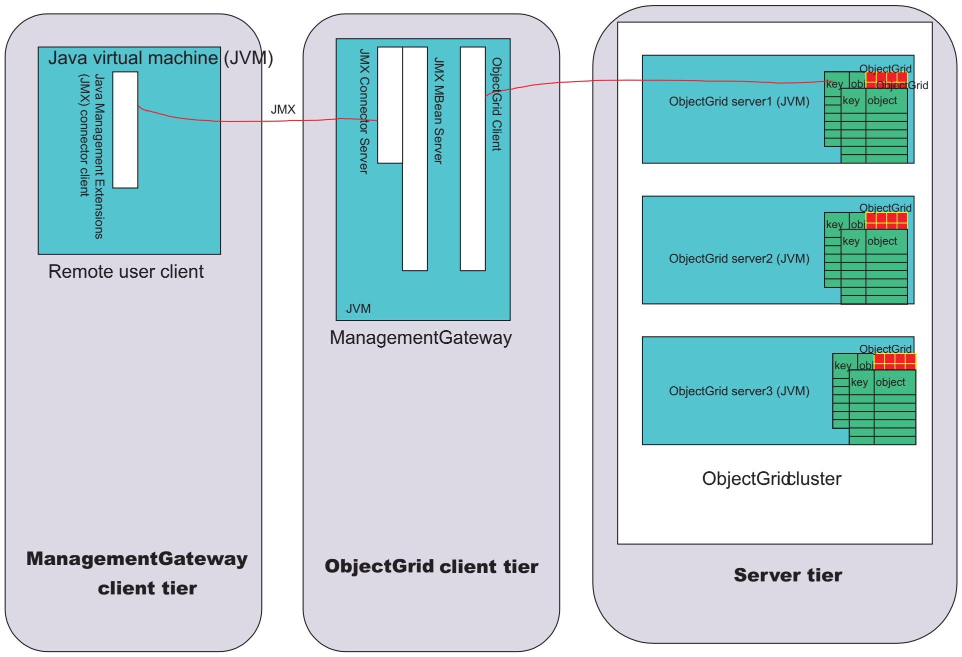


*Figure 15. Get map statistics from the server1 server*

Run the following code in a user program that is run in the remote user client section of the previous diagram:

```
JMXServiceURL url = new
 JMXServiceURL("service:jmx:rmi://host/jndi/rmi://localhost:1099/jmxconnector");
JMXConnector c = JMXConnectorFactory.connect(url);
MBeanServerConnection mbsc = c.getMBeanServerConnection();
Iterator it = mbsc
 .queryMBeans(new ObjectName
  ("ManagementMap:type=ObjectGrid,OG=OG1,Map=map1,S=server1"), null)
   .iterator();
ObjectInstance oi = (ObjectInstance) it.next();
ObjectName mapMBean = oi.getObjectName();
MapStatsModule stats = (MapStatsModule) mbsc.invoke(
 mapMBean,
 "retrieveStatsModule",
 new Object[] { },
 new String[] { });
```

To stop the server1 server through the ManagementGateway:

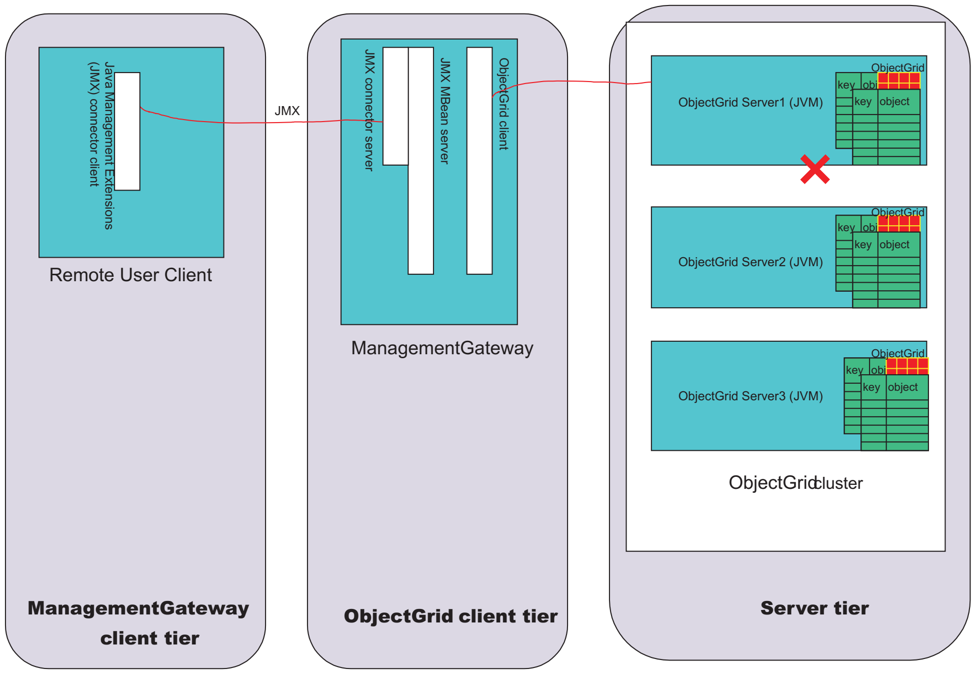


*Figure 16. Stop the server1 server*

Run the following code in a user program that is run in the remote user client in the previous diagram:

```
JMXServiceURL url = new JMXServiceURL(
  "service:jmx:rmi://host/jndi/rmi://localhost:1099/jmxconnector");
JMXConnector c = JMXConnectorFactory.connect(url);
 MBeanServerConnection mbsc = c.getMBeanServerConnection();
Iterator it = mbsc
 .queryMBeans(new ObjectName("ManagementServer:type=ObjectGrid,S=Server1"), null)
 .iterator();
ObjectInstance oi = (ObjectInstance) it.next();
ObjectName server1MBean = oi.getObjectName();
boolean stop = ((Boolean) mbsc.invoke(
 server1MBean,
 "stopServer",
 new Object[] { },
  new String[] { })).booleanValue();
```

After running the previous code sample, the server1 server stops. After the server1 server is stopped, it cannot be restarted with the ManagementGateway. The server can be restarted using the command line. See “Stop ObjectGrid servers” on page 83 for more information.

## ObjectGrid managed beans (MBeans)

Five types of MBeans exist in the ObjectGrid environment. Each MBean refers to a specific entity, such as a map, object grid, server, replication group, or replication group member, and has attributes and operations.

Each MBean in ObjectGrid has getxxx methods that represent attribute values. These getxxx methods cannot be called from a user program directly. This is because the Java management extensions (JMX) specification treats attributes differently from operations. Attributes can be viewed through any third-party JMX console, and operations can be performed either through a user program or a third-party JMX console.

## MapMbean Mbean

The MapMBean allows the user to monitor the statistics of each map defined for the cluster. Each map has the following statistics associated with it:

- Batch update time (min/max/mean/total)
- Count
- Hit rate

Also, because maps can be partitioned across servers, you can scope the map statistics to a particular server or replication group member. You can also map statistics for the entire cluster. The ObjectName for a MapMBean can be specified in several ways:

- `"ManagementMap:type=ObjectGrid,OG=ObjectGridName,Map=MapName"`
- `"ManagementMap:type=ObjectGrid,OG=ObjectGridName,Map=MapName, S=ServerName"`
- `"ManagementMap:type=ObjectGrid,OG=ObjectGridName,Map=MapName, RG=ReplicationGroup,IDX=Index"`

Take an example configuration with the OG1 ObjectGrid, a Map1 map, with two servers in replication group RG1, server1 and server2. Also assume the server1 server is the primary and the server2 server is a replica. To get the statistics for the Map1 map on the primary, use either of these ObjectNames:

- `"ManagementMap:type=ObjectGrid,OG=OG1,Map=Map1,S=server1"`
- `"ManagementMap:type=ObjectGrid,OG=OG1,Map=Map1,RG=RG1,IDX=0"`

In any ObjectName for ObjectGrid MBeans, when IDX=0, it refers to the primary of the replication group. IDX=1-10 refers to replicas for the replication group.

A listing of the MapMBean interface follows:

```
public interface MapMBean {

    /**
     * Operation to get MapStatsModule associated with the MBean.
     *
     * @return MapStatsModule
     */
    MapStatsModule retrieveStatsModule();

    /**
     * Operation will only go to server to get StatsModule if the
     * StatsModule is not cached in the ObjectGridAdministrator.
     *
     */
    void refreshStatsModule();

    /**
     * Map.
     *
     * @return name of map
     */
    String getMapName();

    /**
     * ObjectGrid containing the map.
     *
     * @return name of the object grid
     */
    String getObjectGridName();

    /**
     * Server name of the replication group member for the map.
     *
     * @return name of server of the replication group member
     */
    String getServerName();

    /**
     * Name of replication group for the map.
     *
     * @return name of replication group
     */
    String getReplicationGroup();

    /**
     * Index of replication group member for the map.
     *
     * @return index of replication group member
     */
    int getIndex();

    /**
     * MapStatsModule attribute loaded up by the
  * retrieveStatsModule call.
     *
     * @return String form of MapStatsModule
     */
    String getMapStatsModule();

    /**
     * Map count attribute loaded up by the
   * retrieveStatsModule call.
     *
     * @return number of entries in map
     */
    long getMapCountStatistic();

    /**
     * Hit rate attribute loaded up by the
   * retrieveStatsModule call.
     *
     * @return hit rate for map
     */
    double getMapHitRateStatistic();

    /**
     * Mean batch update time attribute loaded up by the
  * retrieveStatsModule call.
     *
     * @return mean batch update time for map
     */
    double getMapBatchUpdateMeanTime();

    /**
     * Maximum batch update time attribute loaded up by the
  * retrieveStatsModule call.
     *
     * @return maximum batch update time for map
     */
    double getMapBatchUpdateMaxTime();

    /**
     * Minimum batch update time attribute loaded up by the
  * retrieveStatsModule call.
     *
     * @return minimum batch update time for map
     */
    double getMapBatchUpdateMinTime();
    /**
     * Total batch update time attribute loaded up by the
   * retrieveStatsModule call.
     *
     * @return total batch update time for map
     */
    double getMapBatchUpdateTotalTime();
}
```

## ObjectGridMBean MBean

The ObjectGridMBean MBean allows the user to monitor the statistics for all the maps in each ObjectGrid that is defined for the cluster. Each ObjectGrid has the following statistics associated with it:

- Transaction time (min/max/mean/total)
- Count

Also, because ObjectGrids can be partitioned across servers, you can scope the ObjectGrid statistics to a particular server or replication group member. You can also get ObjectGrid statistics for the entire cluster. The ObjectName for a ObjectGridMBean can be specified in several ways:

- "ManagementObjectGrid:type=ObjectGrid,OG=ObjectGridName"
- "ManagementObjectGrid:type=ObjectGrid,OG=ObjectGridName, S=ServerName"
- "ManagementObjectGrid:type=ObjectGrid,OG=ObjectGridName, RG=ReplicationGroup,IDX=Index"

Following is a listing of the ObjectGridMbean interface:

```
public interface ObjectGridMBean {

    /**
     * Operation to get OGStatsModule associated with the MBean.
     *
     * @return OGStatsModule
     */
    OGStatsModule retrieveStatsModule();

    /**
     * Operation will only go to server to get StatsModule if the
     * StatsModule is not cached in the ObjectGridAdministrator.
     *
     */
    void refreshStatsModule();

    /**
     * ObjectGrid.
     *
     * @return name of the object grid
     */
    String getObjectGridName();

    /**
     * Server name of the replication group member for the ObjectGrid.
     *
     * @return name of server of the replication group member
     */
    String getServerName();

    /**
     * Name of replication group for the ObjectGrid.
     *
     * @return name of replication group
     */
    String getReplicationGroup();

    /**
     * Index of replication group member for the ObjectGrid.
     *
     * @return index of replication group member
     */
    int getIndex();

    /**
     * OGStatsModule attribute loaded up by the retrieveStatsModule call.
     *
     * @return String form of OGStatsModule
     */
    String getOGStatsModule();

    /**
     * ObjectGrid count attribute loaded up by the retrieveStatsModule call.
     *
     * @return number of transactions
     */
    long getOGCount();

    /**
     * Maximum transaction time attribute loaded up by the retrieveStatsModule call.
     *
     * @return maximum transaction time for the ObjectGrid
     */
    long getOGMaxTranTime();

    /**
     * Minimum transaction time attribute loaded up by the retrieveStatsModule call.
     *
     * @return minimum transaction time for the ObjectGrid
     */
    long getOGMinTranTime();

    /**
     * Mean transaction time attribute loaded up by the retrieveStatsModule call.
     *
     * @return mean transaction time for the ObjectGrid
     */
    double getOGMeanTranTime();

    /**
     * Total transaction time attribute loaded up by the retrieveStatsModule call.
     *
     * @return total transaction time for the ObjectGrid
     */
    long getOGTotalTranTime();
}
```

## ServerMBean MBean

The ServerMBean Mbean allows the user to perform operations on servers in the cluster. The ObjectName for a ServerMBean can be specified in the following way:

- ″ManagementServer:type=ObjectGrid,S=ServerName″

A listing of the ServerMBean interface follows:

```
public interface ServerMBean {

    /**
     * Operation to load the replication status for the server.
     *
     */
    void retrieveReplicationStatus();

    /**
     * Return the name of the server.
     *
     * @return server name
     */
    String getServerName();

    /**
     * Operation to get the status of the server.
     *
     * @return status of server (true if running, false if not)
     */
    boolean retrieveServerStatus();

    /**
     * Operation to stop the server.
     *
     * @return true if server was stopped, false if not
     */
    boolean stopServer();

    /**
     * Operation to force the server stop.
     *
     * @return true if server was stopped, false if not
     */
    boolean forceStopServer();

    /**
     * Operation to stop the cluster the server is a part of.
     *
     * @param determines if servers are stopped with force
     * @return true if cluster was stopped, false if not
     */
    boolean stopCluster(Boolean force);

    /**
     * Operation to modify the trace spec for all servers in
     * the cluster the server is a part of.
     *
     * @param trace specification
     */
    void modifyClusterTraceSpec(String spec);

    /**
     * Operation to modify the trace spec for the server.
     *
     * @param trace specification
     */
    void modifyServerTraceSpec(String spec);
}
```

## ReplicationGroupMBean Mbean

The ReplicationGroupMBean allows you to monitor the status for all the replication group members associated with a specific replication group including which server is the primary and up to ten replicas. The ObjectName for a ReplicationGroupMBean can be specified:

- ″ManagementReplicationGroup:type=ObjectGrid,RG=ReplicationGridName″

A listing of the ReplicationGroupMBean interface follows:

```
public interface ReplicationGroupMBean {

    /**
     * Operation to load up the status of the replication group attributes.
     *
     */
    String[] retrieveReplicationGroupStatus();

    /**
     * ReplicationGroupName attribute.
     *
     * @return name of the ReplicationGroup
     */
    String getReplicationGroupName();

    /**
     * Primary attribute.
     *
     * @return name of the Primary
     */
    String getPrimary();

    /**
     * Replica1 attribute.
     *
     * @return server name of Replica1
     */
    String getReplica1();

    /**
     * Replica2 attribute.
     *
     * @return server name of Replica2
     */
    String getReplica2();

    /**
     * Replica3 attribute.
     *
     * @return server name of Replica3
     */
    String getReplica3();

    /**
     * Replica4 attribute.
     *
     * @return server name of Replica4
     */
    String getReplica4();

    /**
     * Replica5 attribute.
     *
     * @return server name of Replica5
     */
    String getReplica5();

    /**
     * Replica6 attribute.
     *
     * @return server name of Replica6
     */
    String getReplica6();

    /**
     * Replica7 attribute.
     *
     * @return server name of Replica7
     */
    String getReplica7();

    /**
     * Replica8 attribute.
     *
     * @return server name of Replica8
     */
    String getReplica8();

    /**
     * Replica9 attribute.
     *
     * @return server name of Replica9
     */
    String getReplica9();

    /**
     * Replica10 attribute.
     *
     * @return server name of Replica10
     */
    String getReplica10();

    /**
     * All replicas for this replication group comma delimited
     *
     * @return server names of all replicas
     */
    String getReplicas();
}
```

## ReplicationGroupMemberMBean Mbean

The ReplicationGroupMemberMBean allows you to monitor the following statistics for a replication group member:

- Status of a replication group member. You can monitor primary or replica members.
- Replica weight ratio. This statistic only applies to replication group members that are replicas. This ratio is a quantification of how close a replica's maps are to being synchronized with the primary's maps. The higher the ratio, the closer a replica is to having the primary's up-to-date information.

The ObjectName for a ReplicationGroupMemberMBean can be specified in the following ways:

- `"ManagementReplicationGroupMember:type=ObjectGrid, RG=ReplicationGridName,S=ServerName"`
- `"ManagementReplicationGroupMember:type=ObjectGrid, RG=ReplicationGridName,IDX=Index"`

Specifying IDX=0 returns the primary of the replication group and IDX=1 up to 10 are replicas. A listing of the ReplicationGroupMBean interface follows:

```
public interface ReplicationGroupMemberMBean {

    /**
     * Operation to load up the status of the replication group member attributes.
     *
     */
    void retrieveReplicationGroupMemberStatus();

    /**
```

```
      * Operation to load up the status of the replication group member
    * attributes.
      * Will use the cache as opposed to the retrieveReplicationGroupMemberStatus
   * method which will go to the server to get status.
      *
      */
     void refreshReplicationGroupMemberStatus();

     /**
      * ReplicationGroupName attribute.
      *
      * @return name of the ReplicationGroup this member belongs to
      */
     String getReplicationGroupName();

     /**
      * Status of the ReplicationGroupMember: primary/replica/standby.
      *
      * @return status of the ReplicationGroupMember
      */
     String getStatus();

     /**
      * Statistic representing the percentage how close a replica is
      * to being up to date with the primary maps.
      *
      * @return Replica statistic of the ReplicationGroupMember
      */
     double getReplicaWeightRatio();

     /**
      * Name of server on which this ReplicationGroupMember resides.
      *
      * @return server name
      */
     String getServerName();

     /**
      * Index of ReplicationGroupMember.
      *
      * @return index of replica
      */
     int getIndex();
}
```

# Chapter 8. Command line support

Use command line scripts to manage your ObjectGrid servers.

A set of script files is provided in the `/ObjectGrid/bin` directory of a mixed server environment installation. These scripts can be used to start or stop an ObjectGrid server, start a management gateway server, and encode passwords in a property file. Before attempting to use the scripts, verify that the JAVA_HOME environment variable is set, and that its value is an ObjectGrid-supported version of Java. You can update JAVA_HOME in the `setupCmdLine.bat|sh` file to point to an appropriate version of Java if you do not want to change your environment variable globally.

See the following topics for more information about the command line scripts:
- "Start ObjectGrid servers"
- "Stop ObjectGrid servers" on page 83
- "Start the management gateway server" on page 84
- "Password encoding" on page 86

## Start ObjectGrid servers

The startOgServer script is provided to start an ObjectGrid server.

### Usage

Use the `startOgServer.bat` file to start a server on a Windows machine. Use the `startOgServer.sh` file to start an ObjectGrid server on Linux and Unix platforms.

### Using XML files

A valid ObjectGrid XML file must be paired with a valid cluster XML file to successfully start an ObjectGrid server. XML files can be passed into the startOgServer script using a regular file name or a Uniform resource locator (URL). The URL option allows for different protocols besides the file protocol, for example http, ftp, or jarfile.

The startOgServer script arguments for starting a server using XML files follows:

```
startOgServer.bat <server> -objectgridFile <XML file> | -objectgridUrl
 <XML file URL> -clusterFile <XML file> | -clusterUrl <XML file URL> [options]
```

### Example

Following are a few examples of starting the server1 ObjectGrid server. These examples make use of the `startOgServer.bat` file.

```
startOgServer.bat server1 -objectgridFile c:\objectgrid\xml\university.xml
-clusterFile c:\objectgrid\xml\universityCluster3Servers.xml

startOgServer.bat server1 -objectgridFile ..\xml\university.xml
-clusterUrl file:///c:/objectgrid/xml/universityCluster3Servers.xml

startOgServer.bat server1 -objectgridUrl file:///c:/objectgrid/xml/university.xml
 -clusterFile ..\xml\universityCluster3Servers.xml

startOgServer.bat server1 -objectgridUrl file:///c:/objectgrid/xml/university.xml
-clusterUrl file:///c:/objectgrid/xml/universityCluster3Servers.xml
```

The examples use the `universityCluster3Servers.xml` file. Because the server1 server is specified as the server to start, the `universityCluster3Servers.xml` file must have a serverDefinition value with the name `server1`.

The `universityCluster3Servers.xml` file follows. Notice the server1 serverDefinition, and that its host is lion.ibm.com. The server1 server must be started on the lion.ibm.com host. This file also defines the server2 and server3 servers. These servers must be started on the tiger.ibm.com and bear.ibm.com hosts, respectively.

*universityCluster3Servers.xml* file

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config/cluster
  ../objectGridCluster.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config/cluster">

 <cluster name="universityCluster">
  <serverDefinition name="server1" host="lion.ibm.com" clientAccessPort="12501"
   peerAccessPort="12502" />
  <serverDefinition name="server2" host="tiger.ibm.com" clientAccessPort="12503"
   peerAccessPort="12504" />
  <serverDefinition name="server3" host="bear.ibm.com" clientAccessPort="12505"
   peerAccessPort="12506" />
 </cluster>

 <objectgridBinding ref="academics">
  <mapSet name="academicsMapSet" partitionSetRef="partitionSet1">
   <map ref="faculty" />
   <map ref="student" />
   <map ref="course" />
  </mapSet>
 </objectgridBinding>

 <objectgridBinding ref="athletics">
  <mapSet name="athleticsMapSet" partitionSetRef="partitionSet1">
   <map ref="athlete" />
   <map ref="equipment" />
  </mapSet>
 </objectgridBinding>

 <partitionSet name="partitionSet1">
  <partition name="partition1" replicationGroupRef="replicationGroup1" />
 </partitionSet>

 <replicationGroup name="replicationGroup1">
  <replicationGroupMember serverRef="server1" priority="1" />
  <replicationGroupMember serverRef="server2" priority="2" />
  <replicationGroupMember serverRef="server3" priority="3" />
 </replicationGroup>
</clusterConfig>
```

## Bootstrapping

After one ObjectGrid server in the cluster is available, other servers in the cluster can bootstrap to the available server. The startOgServer script must be supplied with the host and client access port of an already available server to bootstrap to it.

Because the first server in a cluster is launched with XML, it already has the configuration information for all of the servers in the cluster. Bootstrapping allows the launching server to connect to the available server and download the configuration.

Here are the startOgServer script arguments for starting a server by bootstrapping to an available server.

```
startOgServer.bat <server> -bootstrap  <host:port,host:port> [options]
```

Following are a few examples of starting a server by bootstrapping to another server. For the first example, assume that the server1 server from the `universityCluster3Servers.xml` file has been starting using XML files and is available. This example shows how to bootstrap to the server1 server to start the server2 server.

```
startOgServer.bat server2 -bootstrap lion.ibm.com:12501
```

For the next example, assume that the server2 server started successfully, but the server1 server becomes unavailable. A comma-separated list of host:port combinations can be used when bootstrapping to another server. An attempt is made to contact each host and port in the list until an available server is found. In the following example, the server3 attempts to contact the host and port of server1 server. However, because the server1 server is unavailable in this scenario, the connection fails. Taking the next item in the list, the server3 server attempts to bootstrap to the host and port for the server2 server. This bootstrap attempt should succeed because the server2 server is available.

```
startOgServer.bat server3 -bootstrap lion.ibm.com:12501,tiger.ibm.com:12503
```

## Optional arguments

Several optional arguments exist that can be passed to the startOgServer script. The valid startOgServer arguments follow.

**Options:**

- -traceSpec <trace specification>
- -traceFile <trace file>
- -serverSecurityFile <server security properties file>
- -timeout <seconds>
- -script <script file name>
- -jvmArgs <JVM arguments>

**-traceSpec**

The -traceSpec argument can be used to set a trace specification that takes effect almost immediately during server startup. During normal server startup, the trace specification is not set until it can be read from the cluster XML file or from the bootstrapped configuration. If problems occur during server startup, it might be helpful to set the trace specification earlier.

Following is an example of how to set the -traceSpec option:

```
startOgServer.bat server1 -objectgridFile c:\objectgrid\xml\university.xml
 -clusterFile c:\objectgrid\xml\universityCluster3Servers.xml
 -traceSpec ObjectGrid=all=enabled
```

**-traceFile**

The -traceFile argument can be used to specify a location for trace that is output during server startup. After the configuration for this server is read, its trace settings as specified by the cluster XML file takes effect.

Following is an example of how to set the -traceFile option:

```
startOgServer.bat server2 -bootstrap lion.ibm.com:12501 -traceFile
 c:\objectgrid\trace.log
```

**-serverSecurityFile**

The -serverSecurityFile argument can be used to pass a server its security properties file. This option is required when security is enabled on the server. Following is an example of how to set the -serverSecurityFile option:

```
startOgServer.bat server1 -objectgridUrl file:///c:/objectgrid/xml/
   university.xml
     -clusterFile ..\xml\universityCluster3Servers.xml
  -serverSecurityFile c:\objectgrid\props\serverSecurity.props
```

**-timeout**

The -timeout argument can be used to specify the amount of time, in seconds, that is allowed to pass before the launching of the server is aborted. By default, the server is allowed 90 seconds to become available from the time it was launched. If this time is too short for a particular scenario, use the -timeout argument to set it to a more appropriate value. An example of how to use the timeout argument follows:

```
startOgServer.bat server1 -objectgridFile ..\xml\university.xml
    -clusterUrl file:///c:/objectgrid/xml/universityCluster3Servers.xml
  -timeout 120
```

**-script**

The -script argument can be used to create a script that launches an ObjectGrid server process and keeps its output in the current command prompt. In normal circumstances, when an ObjectGrid server is launched, the startOgServer script displays output from the server process to the command prompt until the server is available. After the server is available, startOgServer stops displaying the output from the server process and exits. In some cases, you might want to launch a server process that outputs to the current command prompt.

When specifying a file name for the script, do not give a path to the file. The file is placed in the `bin` directory of the OBJECTGRID_HOME path. Supply the name of the file. The script file that is created includes the arguments that were passed to the startOgServer script so it is not necessary to supply those same arguments when running the created script.

Following is an example of how to use the -script option:

```
startOgServer.bat server1 -objectgridUrl
  file:///c:/objectgrid/xml/university.xml
     -clusterUrl file:///c:/objectgrid/xml/universityCluster3Servers.xml
  -script universityClusterServer1.bat
```

This example creates a `universityClusterServer1.bat` script in the `OBJECTGRID_HOME/bin` directory. To run the newly created script, navigate to the proper directory on the command prompt, type the name of the script, and press **Enter**.

**-jvmArgs**

The -jvmArgs argument can be used to send arguments to the ObjectGrid server Java virtual machine (JVM) that is being launched. Any argument that can be passed to the JVM normally can be passed to the server using the -jvmArgs argument.

The -jvmArgs argument must be the last ObjectGrid optional argument specified as an argument to the startOgServer script. Everything that comes after the -jvmArgs argument is passed to the server JVM as a JVM argument. An example of how to set the -jvmArgs argument follows:

```
startOgServer.bat server2 -bootstrap lion.ibm.com:12501
  -jvmArgs -Xms768M -DmyProp=value1
```

If the -jvmArgs argument includes a -classpath or a -cp JVM argument, the classpath specified is appended to the ObjectGrid classpath. Following is an example of using the -jvmArgs argument to include the Xerces Java archive (JAR) files in the classpath that are used to launch an ObjectGrid server.

```
startOgServer.bat server2 -bootstrap lion.ibm.com:12501 -jvmArgs -cp
 C:\xerces2_7_1\xml-apis.jar;c:\xerces2_7_1\xercesImpl.jar
```

## Stop ObjectGrid servers

Use the stopOgServer script to stop ObjectGrid servers.

### Usage

Use the `stopOgServer.bat` file to stop a server on a Windows machine. Use the `stopOgServer.sh` file to stop an ObjectGrid server on Linux and Unix platforms. The stopOgServer script creates a client that can stop a server by connecting to any available server in the cluster. The behavior of this script is similar to bootstrapping to an available server to start another server. Following are the stopOgServer script arguments for stopping a server.

```
stopOgServer.bat <server> -bootstrap <host:port,host:port> [options]
```

### Examples

Following are a few of examples of stopping different ObjectGrid servers. These examples make use of the `stopOgServer.bat` file. For these examples, assume that three servers are up and running: the server1, server2, and server3 servers as defined by the `universityCluster3Servers.xml` file in “Start ObjectGrid servers” on page 79.

This first example stops the server1 server by bootstrapping to its host and client access port.

```
stopOgServer.bat server1 -bootstrap lion.ibm.com:12501
```

Assume that the server1 server stopped successfully. The next example stops the server2 by first attempting to bootstrap to the server1 server. Because the server1 server has already been stopped, the bootstrap is unsuccessful. The next host and port in the list belongs to the server3 server. Because the server3 server is available, the bootstrap to the server3 server is successful and the server2 is stopped.

```
stopOgServer.bat server2 -bootstrap lion.ibm.com:12501,bear.ibm.com:12505
```

### Optional arguments

There are a few optional arguments that can be passed to the stopOgServer script. This section will show how to use each of these optional arguments. Here are the valid stopOgServer arguments followed by the optional arguments.

```
stopOgServer.bat <server> -bootstrap  <host:port,host:port> [options]
```

**Options:**

- -traceSpec <trace specification>
- -traceFile <trace file>
- -clientSecurityFile <client security properties file>

**-traceSpec**

The -traceSpec argument can be used to set a trace specification on the client that attempts to stop an ObjectGrid server. Following is an example of how to set the -traceSpec argument:

```
stopOgServer.bat server1 -bootstrap lion.ibm.com:12501 -traceSpec
ObjectGrid=all=enabled
```

**-traceFile**

The -traceFile argument can be used to specify a location for the client trace that is output during server shutdown. Following is an example of how to set the traceFile argument:

```
stopOgServer.bat server2 -bootstrap lion.ibm.com:12501,bear.ibm.com:12505
-traceFile c:\objectgrid\trace.log
```

**-clientSecurityFile**

The -clientSecurityFile argument can be used to pass the client its security properties file. This argument is required when attempting to connect to a server with security enabled.

Following is an example of how to set the -clientSecurityFile argument:

```
stopOgServer.bat server1 -bootstrap lion.ibm.com:12501 -clientSecurityFile
   c:\objectgrid\props\clientSecurity.props
```

## Start the management gateway server

To monitor and administer an ObjectGrid cluster using Java management extensions (JMX), the management gateway must be started either through the command line script or programmatically.

### Purpose

To start the Management Gateway through the command line, use the startManagementGateway script. Use the `startManagementGateway.bat` file to start a ManagementGateway server on a Windows machine. Use the `startManagementGateway.sh` file to start a ManagementGateway server on Linux and Unix platforms. For more information on Management Gateway function, ObjectGrid MBeans, and JMX, see Chapter 7, “System management overview,” on page 65.

The startManagementGateway script creates a JMX connector server and an ObjectGrid client that connects to an ObjectGrid cluster to stop servers, gather status and statistics, and perform several more functions.

Following are the startManagementGateway script arguments for starting a management gateway server.

```
startManagementGateway.bat -connectorPort <port> -clusterHost <host>
 -clusterPort <port> -clusterName <cluster> [options]
```

### Optional arguments

A few optional arguments can be passed to the startManagementGateway script. The valid startManagementGateway arguments are followed by the optional arguments.

```
startManagementGateway.bat -connectorPort <port> -clusterHost <host>
 -clusterPort <port> -clusterName <cluster> [options]
```

**Options**

- -traceEnabled <true/false trace enabled>
- -traceSpec <trace specification>
- -traceFile <trace file>
- -refreshInterval <MBean attribute refresh interval>
- -sslEnabled <true/false SSL enabled for management gateway>
- -clientSecurityFile <path to client security file>

**-traceEnabled**
: The -traceEnabled argument can be used to set whether trace is turned on for the Management Gateway server. The default is false, so the only way to see ObjectGrid trace is to enable it by setting -traceEnabled to ″true″ and providing valid -traceSpec and -traceFile values.

**-traceSpec**
: The -traceSpec argument can be used to set a trace specification for the management gateway server.

**-traceFile**
: The -traceFile argument can be used to specify a location for the Management Gateway trace output. Following is an example of how to set the traceEnabled, traceSpec, and traceFile arguments.

```
startManagementGateway.bat -connectorPort 1099 -clusterHost lion.ibm.com
 -clusterPort 12501 -clusterName universityCluster -traceEnabled true
 -traceSpec ObjectGrid=all=enabled -traceFile \\objectgrid\\trace.log
```

**-refreshInterval**
: The -refreshInterval argument can be used to pass the amount of time (in seconds) that the management gateway waits between refreshes of the MBean attribute values. The default value is 120 seconds. Following is an example of how to set the refreshInterval argument:

```
startManagementGateway.bat -connectorPort 1099 -clusterHost lion.ibm.com
 -clusterPort 12501 -clusterName universityCluster -refreshInterval 60
```

**-sslEnabled**
: The -sslEnabled argument can be used to set whether SSL is enabled for the management gateway. If the value for this argument is `true`, any user client that connects to the management gateway server needs to provide SSL properties:

  - -Djavax.net.ssl.trustStore
  - -Djavax.net.ssl.trustStorePassword

  The default value if the -sslEnabled argument is not provided is ″false″.

**-clientSecurityFile**
: The -clientSecurityFile argument can be used to pass the file name that contains the client security properties for secure client access between the Management Gateway server and the ObjectGrid cluster. This argument is required when attempting to connect to a cluster with security enabled. ObjectGrid ships the following client security property file template: `security.ogclient.props`.

Here is an example of how to set the sslEnabled and clientSecurityFile properties:

```
startManagementGateway.bat -connectorPort 1099 -clusterHost lion.ibm.com
-clusterPort 12501 -clusterName universityCluster -sslEnabled true
-clientSecurityFile ..\\properties\\security.ogclient.props
```

## Password encoding

Password encoding deters the casual observation of passwords in the ObjectGrid security property files.

### Usage

ObjectGrid contains several encoded passwords that are not encrypted. ObjectGrid provides the FilePasswordEncoder utility, which you can use to encode these passwords. Use the `FilePasswordEncoder.bat` file to encode passwords on a Windows machine. Use the `FilePasswordEncoder.sh` file to encode passwords on Linux and Unix platforms.

The command syntax is as follows:

```
FilePasswordEncoder.bat file_name password_properties_list [file_type]
```

### Options

The following options are available for the FilePasswordEncoder command:

**file_name**
: The file_name is used to specify the file name which has passwords to be encoded. For example, `security.ogserver.props`.

**password_prop_list**
: The password_prop_list is a list of password property names separated by commas, for example, ″trustStorePassword,keyStorePassword″.

**file_type**
: This argument is optional. The file_type can either be an xml or property value, indicating whether the supplied file is a property file or an XML file. The default value is `property`. Currently, ObjectGrid does not store any passwords in an XML file, so this option is not required. The following examples demonstrate the correct syntax:

-
  ```
  FilePasswordEncoder.bat security.ogclient.props
  ″trustStorePassword,keyStorePassword″
  ```

-
  ```
  FilePasswordEncoder.bat security.ogserver.props
  ″trustStorePassword,keyStorePassword,secureTokenKeyStorePassword,
  secureTokenKeyPairPassword,secureTokenSecretKeyPassword″
  ```

This FilePasswordEncoder utility is not shipped with WebSphere Extended Deployment. You can use the PropFilePasswordEncoder utility provided by the WebSphere Application Server to encode these passwords. Refer to the PropFilePasswordEncoder command reference for more details.

# Chapter 9. ObjectGrid application programming interface overview

This section discusses the how to configure the ObjectGrid with XML or through programmatic interfaces. In addition, information is included to implement the external interfaces that ObjectGrid provides. In all cases, an overview, API interfaces, and examples are described.

### API documentation

The JavaDoc for ObjectGrid is the definitive source of information about the APIs. Find the JavaDoc in the following directory of your WebSphere Extended Deployment installation: *install_root*\web\xd\apidocs

## ObjectGridManager interface

The ObjectGridManagerFactory class and the ObjectGridManager interface provide a mechanism to create, access, and cache ObjectGrid instances. The ObjectGridManagerFactory class is a static helper class to access the ObjectGridManager interface, a singleton. The ObjectGridManager interface includes several convenience methods to create instances of an ObjectGrid object. The ObjectGridManager interface also facilitates creation and caching of ObjectGrid instances that can be accessed by several users.

## createObjectGrid methods

Use this topic to learn about the seven createObjectGrid methods that are in the ObjectGridManager interface.

### createObjectGrid methods

The ObjectGridManager interface has seven createObjectGrid methods. Following is a simple scenario:

#### Simple case with default configuration

Following is a simple case of creating an ObjectGrid to share among many users.

```
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
final ObjectGridManager oGridManager=
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid employees = oGridManager.createObjectGrid("Employees",true);
employees.initialize();
employees.
/*sample continues..*/
```

The preceding Java code snippet creates and caches the Employees ObjectGrid. The Employees ObjectGrid is initialized with the default configuration and is ready to be used. The second parameter in the createObjectGrid method is set to `true`, which instructs the ObjectGridManager to cache the ObjectGrid instance it creates. If this parameter is set to `false`, the instance is not cached. Every ObjectGrid instance has a `name`, and the instance can be shared among many clients or users based on that name.

If the objectGrid instance is used in peer-to-peer sharing, the caching must be set to `true`. For more information on peer-to-peer sharing, see Chapter 12, “Distributing changes between peer Java virtual machines,” on page 325.

## XML configuration

ObjectGrid is highly configurable. The previous example demonstrates how to create a simple ObjectGrid without any configuration. With this example, you can create a pre-configured ObjectGrid instance that is based on an XML configuration file. You can configure an ObjectGrid instance programmatically or using an XML-based configuration file. You can also configure ObjectGrid using a combination of both approaches.

The ObjectGridManager interface allows creation of an ObjectGrid instance based on the XML configuration. The ObjectGridManager interface has several methods that take a URL as an argument. Every XML file that is passed into the ObjectGridManager must be validated against the schema. XML validation can be disabled only when the file has been previously validated and no changes have been made to the file since its last validation. Disabling validation saves a small amount of overhead but introduces the possibility of using an invalid XML file. The IBM Java Developer Kit (JDK) 1.4.2 has support for XML validation. When using a JDK that does not have this support, Apache Xerces might be required to validate the XML.

The following Java code snippet demonstrates how to pass in an XML configuration file to create an ObjectGrid.

```
import java.net.MalformedURLException;
import java.net.URL;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
boolean validateXML = true; // turn XML validation on
boolean cacheInstance = true; // Cache the instance
String objectGridName="Employees"; // Name of Object Grid URL
allObjectGrids = new URL("file:test/myObjectGrid.xml");
final ObjectGridManager oGridManager=
 ObjectGridManagerFactory.getObjectGridManager();
 ObjectGrid employees = oGridManager.createObjectGrid(objectGridName,
                        allObjectGrids,
                        validateXML,
                        cacheInstance);
```

The XML file can contain configuration information for several ObjectGrids. The previous code snippet specifically returns ObjectGrid ″Employees″, assuming that the ″Employees″ configuration is defined in the file. For the XML syntax, see “ObjectGrid configuration” on page 249.

Seven createObjectGrid methods exist. The methods are documented in the following code block.

```
/**
* A simple factory method to return an instance of an
* Object Grid. A unique name is assigned.
* The instance of ObjectGrid is not cached.
* Users can then use {@link ObjectGrid#setName(String)} to change the
* ObjectGrid name.
*
* @return ObjectGrid an instance of ObjectGrid with a unique name assigned
* @throws ObjectGridException any error encountered during the ObjectGrid creation
* @ibm-api
```

```
*/
public ObjectGrid createObjectGrid() throws ObjectGridException;
/**
* A simple factory method to return an instance of an ObjectGrid with the
* specified name. The instances of ObjectGrid can be cached. If an ObjectGrid
* with the this name has already been cached, an ObjectGridException
* will be thrown.
*
* @param objectGridName the name of the ObjectGrid to be created.
* @param cacheInstance true, if the ObjectGrid instance should be cached
* @return an ObjectGrid instance
* @this name has already been cached or
* any error during the ObjectGrid creation.
* @ibm-api
*/
public ObjectGrid createObjectGrid(String objectGridName, boolean cacheInstance)
 throws ObjectGridException;
/**
* Create an ObjectGrid instance with the specified ObjectGrid name. The
* ObjectGrid instance created will be cached.
* @param objectGridName the Name of the ObjectGrid instance to be created.
* @return an ObjectGrid instance
* @throws ObjectGridException if an ObjectGrid with this name has already
* been cached, or any error encountered during the ObjectGrid creation
* @ibm-api
*/
public ObjectGrid createObjectGrid(String objectGridName)
 throws ObjectGridException;
/**
* Create an ObjectGrid instance based on the specified ObjectGrid name and the
* XML file. The ObjectGrid instance defined in the XML file with the specified
* ObjectGrid name will be created and returned. If such an ObjectGrid
* cannot be found in the xml file, an exception will be thrown.
*
* This ObjecGrid instance can be cached.
*
* If the URL is null, it will be simply ignored. In this case, this method behaves
* the same as {@link #createObjectGrid(String, boolean)}.
*
* @param objectGridName the Name of the ObjectGrid instance to be returned. It
* must not be null.
* @param xmlFile a URL to a wellformed xml file based on the ObjectGrid schema.
* @param enableXmlValidation if true the XML is validated
* @param cacheInstance a boolean value indicating whether the ObjectGrid
* instance(s)
* defined in the XML will be cached or not. If true, the instance(s) will
* be cached.
*
* @throws ObjectGridException if an ObjectGrid with the same name
* has been previously cached, no ObjectGrid name can be found in the xml file,
* or any other error during the ObjectGrid creation.
* @return an ObjectGrid instance
* @see ObjectGrid
* @ibm-api
*/
public ObjectGrid createObjectGrid(String objectGridName, final URL xmlFile,
final boolean enableXmlValidation, boolean cacheInstance) throws
 ObjectGridException;
/**
* Process an XML file and create a List of ObjectGrid objects based
* upon the file.
* These ObjecGrid instances can be cached.
* An ObjectGridException will be thrown when attempting to cache a
* newly created ObjectGrid
* that has the same name as an ObjectGrid that has already been cached.
*
* @param xmlFile the file that defines an ObjectGrid or multiple
* ObjectGrids
* @param enableXmlValidation setting to true will validate the XML
* file against the schema
* @param cacheInstances set to true to cache all ObjectGrid instances
* created based on the file
* @return an ObjectGrid instance
* @throws ObjectGridException if attempting to create and cache an
* ObjectGrid with the same name as
* an ObjectGrid that has already been cached, or any other error
* occurred during the
* ObjectGrid creation
* @ibm-api
*/
public List createObjectGrids(final URL xmlFile,
final boolean enableXmlValidation,
boolean cacheInstances)
throws ObjectGridException;
/*** Create all ObjectGrids that are found in the XML file. The XML file will be
* validated against the schema. Each ObjectGrid instance that is created will
* be cached. An ObjectGridException will be thrown when attempting to cache a
* newly created ObjectGrid that has the same name as an ObjectGrid that has
* already been cached.
* @param xmlFile The XML file to process. ObjectGrids will be created based
* on what is in the file.
* @return A List of ObjectGrid instances that have been created.
* @throws ObjectGridException if an ObjectGrid with the same name as any of
* those found in the XML has already been cached, or
* any other error encounterred during ObjectGrid creation.
* @ibm-api
*/
public List createObjectGrids(final URL xmlFile) throws ObjectGridException;
/**
* Process the XML file and create a single ObjectGrid instance with the
* objectGridName specified only if an ObjectGrid with that name is found in
* the file. If there is no ObjectGrid with this name defined in the XML file,
* an ObjectGridException
* will be thrown. The ObjectGrid instance created will be cached.
* @param objectGridName name of the ObjectGrid to create. This ObjectGrid
* should be defined in the XML file.
* @param xmlFile the XML file to process
* @return A newly created ObjectGrid
* @throws ObjectGridException if an ObjectGrid with the same name has been
* previously cached, no ObjectGrid name can be found in the xml file,
* or any other error during the ObjectGrid creation.
* @ibm-api
*/
public ObjectGrid createObjectGrid(String objectGridName, URL xmlFile)
throws ObjectGridException;
```

## getObjectGrid methods

Use the getObjectGrid methods to retrieve cached instances.

### Retrieve a cached instance

Since the Employees ObjectGrid instance was cached by the ObjectGridManager interface, any other user can access it with the following code snippet:

```
ObjectGrid myEmployees = oGridManager.getObjectGrid("Employees");
```

Following are the two getObjectGrid methods that return cached ObjectGrid instances.

```
/**
* Get a List of the ObjectGrid instances that have been previously cached.
* Returns null if no ObjectGrid instances have been cached.
* @return a List of ObjectGrid instances that have been previously cached
* @ibm-api
*/
public List getObjectGrids();
/**
* Use this if a ObjectGrid already exists. It returns a cached
* ObjectGrid instance by name. This method returns null if no
* ObjectGrid with this objectGridName has been cached.
*
* @param objectGridName the cached objectgrid name.
* @return a cached ObjectGrid which currently exists.
*
* @since WAS XD 6.0
* @ibm-api
*
*/
public ObjectGrid getObjectGrid(String objectGridName);
```

## removeObjectGrid methods

This topic describes how to use the two removeObjectGrid methods.

### Remove an ObjectGrid instance

To remove ObjectGrid instances from the cache, use one of the removeObjectGrid methods. The ObjectGridManager does not keep a reference of the instances that are removed. Two remove methods exist. One method takes a boolean parameter. If the boolean parameter is set to **true**, the destroy method is called on the ObjectGrid. The call to the destroy method on the ObjectGrid shuts down the ObjectGrid and frees up any resources it is using. Following is a description of how to use the two removeObjectGrid methods:

```
/**
* Remove an ObjectGrid from the cache of ObjectGrid instances
* @param objectGridName the name of the ObjectGrid instance to remove
* from the cache
* @throws ObjectGridException if an ObjectGrid with the objectGridName
* was not found in the cache
* @ibm-api
*/
public void removeObjectGrid(String objectGridName) throws ObjectGridException;
/**
* Remove an ObjectGrid from the cache of ObjectGrid instances and
* destroy its associated resources
* @param objectGridName the name of the ObjectGrid instance to remove
* from the cache
* @param destroy destroy the objectgrid instance and its associated
* resources
* @throws ObjectGridException if an ObjectGrid with the objectGridName
* was not found in the cache
* @ibm-api
*/
public void removeObjectGrid(String objectGridName, boolean destroy)
throws ObjectGridException;
```

## getObjectGridAdministrator method

### Return an ObjectGridAdministrator instance for the cluster

```
public ObjectGridAdministrator getObjectGridAdministrator(ClientClusterContext ctx)

/**
* Return an ObjectGridAdministrator instance for this cluster. Each
* cluster will require the use of a different ObjectGridAdministrator.
*
* @param clientClusterContext. A unique cluster context, with which the client
* needs to interact.
* @param objectGridName the cached objectgrid name.
* @return an ObjectGrid
*
* @since WAS XD 6.0.1
* @ibm-api
*
*/

public ObjectGridAdministrator getObjectGridAdministrator(ClientClusterContext
ontext);
```

See Chapter 7, “System management overview,” on page 65 for more information on this method.

## Use the ObjectGridManager interface to control the life cycle of an ObjectGrid instance

This topic demonstrates how the ObjectGridManager interface can be used to control the life cycle of an ObjectGrid instance using startup beans and a servlet.

### Manage an ObjectGrid instance life cycle in a startup bean

A startup bean can be used to control the life cycle of an ObjectGrid instance. A startup bean loads when an application starts. With a startup bean, code can run whenever an application starts or stops as expected. To create a startup bean, use the home `com.ibm.websphere.startupservice.AppStartUpHome` interface and use the remote `com.ibm.websphere.startupservice.AppStartUp` interface. Implement the `start` and `stop` methods on the bean. The `start` method is invoked whenever the application starts up. The `stop` method is invoked when the application shuts down. The `start` method can be used to create ObjectGrid instances. The`stop` method can be used to destroy ObjectGrid instances. Following is a code snippet that demonstrates this ObjectGrid life cycle management in a startup bean.

```
public class MyStartupBean implements javax.ejb.SessionBean {
private ObjectGridManager objectGridManager;
/*
* The methods on the SessionBean interface have been
* left out of this example for the sake of brevity
*/
public boolean start(){
 // Starting the startup bean
 // This method is called when the application starts
 objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
 try {
  // create 2 ObjectGrids and cache these instances
  ObjectGrid bookstoreGrid =
   objectGridManager.createObjectGrid("bookstore", true);
  bookstoreGrid.defineMap("book");
  ObjectGrid videostoreGrid =
   objectGridManager.createObjectGrid("videostore", true);
  // within the JVM,
  // these ObjectGrids can now be retrieved from the
  //ObjectGridManager using the getObjectGrid(String) method
 } catch (ObjectGridException e) {
  e.printStackTrace();
  return false;
 }
 return true;
}
public void stop(){
 // Stopping the startup bean
 // This method is called when the application is stopped
 try {
  // remove the cached ObjectGrids and destroy them
  objectGridManager.removeObjectGrid("bookstore", true);
  objectGridManager.removeObjectGrid("videostore", true);
  } catch (ObjectGridException e) {
   e.printStackTrace();
  }
 }
}
```

After the `start` method is called, the newly created ObjectGrid instances can be retrieved from the ObjectGridManager. For example, if a servlet is included in the application, the servlet can access these ObjectGrids using the following code snippet:

```
ObjectGridManager objectGridManager =
  ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.getObjectGrid("bookstore");
ObjectGrid videostoreGrid = objectGridManager.getObjectGrid("videostore");
```

### Managing an ObjectGrid life cycle in a servlet

One method to manage the life cycle of an ObjectGrid in a Servlet is to create the ObjectGrid instance in the `init` method and destroy the ObjectGrid in the `destroy` method. If the ObjectGrid instance is cached, it can be retrieved and manipulated in the servlet code. Following is some sample code that demonstrates ObjectGrid creation, manipulation, and destruction within a servlet.

```
public class MyObjectGridServlet extends HttpServlet implements Servlet {
 private ObjectGridManager objectGridManager;
 public MyObjectGridServlet() {
 super();
}
public void init(ServletConfig arg0) throws ServletException {
 super.init();
 objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
 try {
  // create and cache an ObjectGrid named bookstore
  ObjectGrid bookstoreGrid =
   objectGridManager.createObjectGrid("bookstore", true);
  bookstoreGrid.defineMap("book");
 } catch (ObjectGridException e) {
  e.printStackTrace();
 }
}
protected void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException {
 ObjectGrid bookstoreGrid = objectGridManager.getObjectGrid("bookstore");
 BackingMap bookMap = bookstoreGrid.getMap("book");
 // perform operations on the cached ObjectGrid
 // ...
}
public void destroy() {
 super.destroy();
 try {
  // remove and destroy the cached bookstore ObjectGrid
  objectGridManager.removeObjectGrid("bookstore", true);
  } catch (ObjectGridException e) {
  e.printStackTrace();
  }
 }
}
```

## Trace ObjectGrid

This topic explains how to set up tracing for ObjectGrid.

### Java 2 Platform, Standard Edition (J2SE) environment

When it is necessary to send debug information to IBM, use the tracing mechanism to get the debug trace. Following is an example of how to get the debug trace in a J2SE environment:

```
oGridManager.setTraceFileName("debug.log");
oGridManager.setTraceSpecification("ObjectGrid=all=enabled");
```

The previous example does not include tracing of built-in evictor plug-ins for ObjectGrid. If you are using one or more of the evictor plug-ins that are provided by ObjectGrid and you are having problems that might be related to eviction, enable tracing for both ObjectGrid plus the evictors of ObjectGrid as the following example illustrates:

```
oGridManager.setTraceFileName("debug.log");
oGridManager.setTraceSpecification
 ("ObjectGridEvictors=all=enabled:ObjectGrid=all=enabled");
```

### WebSphere Application Server environment

It is not necessary to use ObjectGridManager to set the trace within an WebSphere Application Server environment. You can use the administrative console to set the trace specification.

## ObjectGrid client connect APIs

### The basics

A client connects to an active, or running, server process within a cluster. A client minimally needs the host name and port number of the server to which it connects. The host name and port information is available from the cluster definition XML file that was initially used to start the server. See “ObjectGrid configuration” on page 249 for the XML configuration details. Following is a snippet of the cluster XML definition, which is used as a sample for this section. By using the APIs documented in this section, the client connects to a remote ObjectGrid that is configured to receive and process client requests. Note that the client ″downloads″ both the ObjectGrid and Cluster XML definitions to bootstrap itself. Client configuration is based on the server configuration to which it connects.

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config/cluster ../
 objectGridCluster.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config/cluster">
 <cluster name="cluster1" securityEnabled="false"  clientMaxRetries="15"
  tcpConnectionTimeout="180"
     singleSignOnEnabled="true" loginSessionExpirationTime="300"
  statisticsEnabled="true" statisticsSpec="all=enabled">
  <serverDefinition name="server1" host="s1.myco.com" clientAccessPort="12503"
         peerAccessPort="12500" workingDirectory="/tmp/s1/"
         traceSpec="ObjectGrid=all=disabled"
         systemStreamToFileEnabled="true" />
  <serverDefinition name="server2" host="s2.myco.com" clientAccessPort="12504"
         peerAccessPort="12501" workingDirectory="/tmp/s2/"
         traceSpec="ObjectGrid=all=disabled"
         systemStreamToFileEnabled="true" />
  <serverDefinition name="server3" host="10.5.1.22" clientAccessPort="12505"
         peerAccessPort="12502" workingDirectory="/tmp/s3/"
         traceSpec="ObjectGrid=all=disabled"
         systemStreamToFileEnabled="true"/>
 </cluster>
 <objectgrid-binding
 .
 .
 .
```

This cluster definition is incomplete, but sufficient enough for this example. Within the cluster1 cluster are three servers, the server1, server2 and server3 servers. The clientAccessPort attribute specifies the listener port that the server is listening to and the port to which the client first establishes a connection. From the previous XML snippet, the ports for the server1, server2, and server3 servers are `12503`, `12504`, and `12504` respectively.

## Connect APIs

The ObjectGridManager interface has connect methods that are documented in the following sample. The following connect APIs are available from the ObjectGridManager interface. See the API documentation for a description of these methods.

```
/**
* This allows a client to connect to a remote ObjectGrid
* The remote ObjectGrid is hosted as specified by the parameters:
* @param clusterName: The name of the cluster to which this client
* attaches itself
* @param host: The host on which to connect
* @param port: The clientAceess port that is listening
* @param ClientSecurityConfiguration: Security configuration,can be
* null if security is not configured
* @param overRideObjectGrid xml. This parameter can be null. If it is not null,
* the client side configuration of ObjectGrid plug-in is overridden.
* Not all plug-ins can be overridden. For details see the
* ObjectGrid documents
* @throws ConnectException
*
*/

public ClientClusterContext connect(String clusterName,
          String host,
          String port,
          ClientSecurityConfiguration securityProps,
          URL overRideObjectGrid) throws ConnectException ;


/**
*
* @param clusterName
* @param attributes Host and Port pair attributes that are tried in sequential
* order to connect. If an attempt to connect fails to one server, the next pair
* of host and port attributes is picked to retry the connect.
* @param ClientSecurityConfiguration: Security configuration. It can be null if
* security is not configured.
* @param overRideObjectGrid xml. This parameter can be null. If it is not null,
* the client side configuration of ObjectGrid plug-in is overridden.
* Not all plug-ins can be overridden. For details see the ObjectGrid documents.
* @return ClientClusterContext
* @throws ConnectException
*
*
*/

public ClientClusterContext connect(String clusterName,
             HostPortConnectionAttributes[] attributes,
             ClientSecurityConfiguration securityProps,
             URL overRideObjectGrid)  throws ConnectException ;
```

```
/**
* This method can be used only if a client is colocated with
* an ObjectGrid server, especially in a Java 2 Platform,
* Enterprise Edition (J2EE) environment with IBM WebSphere
* Application Server, which supports the
* embedded ObjectGrid server.
* This method connects the client to the Server which is running
* in the same Java virtual machine (JVM).
* @param securityProps. It can be null if not running in secure mode.
* @param overRideObjectGrid xml. This parameter can be null. If it is
* not null, the client side configuration of ObjectGrid plug-in is overridden.
* Not all plug-ins can be overridden. For details  see the
* ObjectGrid documents
* @return ClientClusterContext
* @throws ConnectException
*
*/

public ClientClusterContext connect(ClientSecurityConfiguration securityProps,
                      URL overRideObjectGrid) throws ConnectException;



/**
* This allows a client to connect to a Remote ObjectGrid
* @param clusterConfigFile  A URL to the clusterConfig File. This is the
* same file that is used to start servers.
* This is used to retrieve host port information. It cannot be null. If it is
* null a IllegalArgumentException exception results.
* @param serverName  A String, the name of the specific server to connect to.
*If the server name is not in the configuration, IllegalArgumentException
* results.
* This parameter can be null, in which case an attempt is made to connect
* to one of servers specified in the cluster
* XML file. If an attempt fails to connect to one, another server is picked,
* It is done, until such time the list is exhausted.
* @param securityProps
* @param overRideObjectGrid xml. This parameter can be null. If it is not
* null, the client side configuration of ObjectGrid plug-in is overridden.
* Not all plug-ins can be overridden. For details, see the ObjectGrid
* documents
* @return ClientClusterContext
* @throws ConnectException
*
* @ibm-api
*/

public ClientClusterContext connect(URL clusterConfigFile,
                 String serverName,
                 ClientSecurityConfiguration securityProps,
                 URL overRideObjectGrid) throws ConnectException ;
```

### Example using Host and Port parameters

The following code uses the cluster XML documented in "The basics" on page 94. This client connects to host `s1.myco.com` at port `12503`.

```
import com.ibm.websphere.objectgrid.ClientClusterContext;
import com.ibm.websphere.objectgrid.ConnectException;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;

public class C1 {
  /**
  * @param args
  */
  public static void main(String[] args) {
    final ObjectGridManager oGridManager=ObjectGridManagerFactory.
   getObjectGridManager();   //step 1
    ClientClusterContext ctx = null;

    try {
      ctx=oGridManager.connect("cluster1","s1.myco.com","12503",null,null);
   //step 2
      ObjectGrid employees = oGridManager.getObjectGrid(ctx,"employees");
   // step 3
      // Do objectGrid operations
      //  get
      //  update
      //  commit...etc..
    } catch (ConnectException e) {
      //connect failed
      e.printStackTrace();
      //terminate
    }finally {
      if(ctx !=null) {
        oGridManager.disconnect(ctx);       // step 4
      }
    }

  }

}
```

1. Get the ObjectGridManager singleton object from the ObjectGridManagerFactory.
2. Call the connect API.
3. Assuming objectGrid employees exist on the remote ObjectGrid, call the getObjectGrid method, by passing the ClientClusterContext parameter.
4. Call the disconnect method. As a last step, all clients must call disconnect, if the work is complete. This is a very important step.

### Providing multiple hosts to automatically re-try connect, in case of a ConnectException exception

In this example, HostPortConnectionAttributes attributes are used to provide an array of host port attributes, that a client can connect to. The API uses this host and port pair attributes in sequential order to connect. If an attempt to connect fails to one server, the next pair of host and port attributes is picked to retry the connect.

```
import com.ibm.websphere.objectgrid.ClientClusterContext;
import com.ibm.websphere.objectgrid.ConnectException;
import com.ibm.websphere.objectgrid.HostPortConnectionAttributes;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;

public class C2 {

  /**
  * @param args
  */
  public static void main(String[] args) {
    final ObjectGridManager oGridManager=ObjectGridManagerFactory.
   getObjectGridManager();
    ClientClusterContext ctx = null;
    HostPortConnectionAttributes[] hca = new HostPortConnectionAttributes[3];
    hca[0]=new HostPortConnectionAttributes("s1.myco.com","12503");
    hca[1]=new HostPortConnectionAttributes("s2.myco.com","12504");
    hca[2]=new HostPortConnectionAttributes("10.5.1.22","12505");

    try {
      ctx=oGridManager.connect("cluster1",hca,null,null);
      ObjectGrid employees = oGridManager.getObjectGrid(ctx,"employees");
      // Do objectGrid operations such as
      //  get
      //  update
      //  commit.... etc...
    } catch (ConnectException e) {

      e.printStackTrace();
    }finally {
      if(ctx !=null) {
        oGridManager.disconnect(ctx);
      }
    }

  }

}
```

**Client and server in the same process**

If client is in the same JVM as the server, the following connect method can be used.

```
ctx=oGridManager.connect(null,null);
```

**Specify cluster XML**

If the client has access to the Cluster XML file, you do not need to specify the host name or port number, This API retrieves the server name and port number, and uses those to connect. Server name is optional and can be null, in which case, the API tries to connect to one of the servers defined in the cluster XML file.

```
ctx=oGridManager.connect(urlToClusterxml,"server1",null,null);
// connect to server1
//                         or
ctx=oGridManager.connect(urlToClusterxml,null,null,null);
//connect to any server in the cluster
```

## Client security with the connect API

In all the examples, the as ClientSecurityConfiguration parameter was null. Passing the null value implies security is disabled. If security is enabled, pass the ClientSecurityConfiguration object as an argument. See "ObjectGrid security" on page 131 for more information.

## Override ObjectGrid XML configuration

The client ″downloads″ the ObjectGrid definitions from the server to configure itself. All plug-ins that are defined in the ObjectGrid are made available to the client. Essentially, a local ObjectGrid exists on the client side that communicates with server side ObjectGrid. By providing an overriding XML file on the connect API, you can ″override″ the plug-in configuration, which are specific for a client use only. These plug-ins are:

**ObjectGrid** plug-ins:

- TransactionCallback plug-in

- ObjectGridEventListener plug-in

**BackingMap** plug-ins:

- Evictor plug-in
- MapEventListener plug-in

Any other plug-ins that are defined in the override XML are ignored.

### Example

Assume that the client needs to override Evictor configuration for a specific BackingMap. That is, on the client side, the Evictor needs to be different than the one configured on the server side.

Assume the server side Evictor is used as follows. It uses the built-in LFUEvictor evictor:

```
<bean id="Evictor"
 className="com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor">
 <property name="maxSize" type="int" value="100" description="..." />
</bean>
```

The client side requirements are different. It is required that a user defined, myco.og.MyEvictor Evictor be used instead. The override XML can include snippet shown below. All the backingMaps configured to use LFUEvictor, use the user defined:

```
<bean id="Evictor" className="myco.og.MyEvictor">
 <property name="name" type="java.lang.String" value="MyEvictor"
   description="..." />
</bean>
```

### Complete XML

The following XML code displays two XML files: one used for server side, and the second for client. This configuration allows a client to override the Evictor configuration for the dow BackingMap.

### Server side ObjectGrid XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="market">
   <backingMap name="dow" ttlEvictorType="NONE" readOnly="false"
    pluginCollectionRef="default" />
  </objectGrid>
 </objectGrids>
 <backingMapPluginCollections>
  <backingMapPluginCollection id="default">
   <bean id="Evictor" className="com.ibm.websphere.objectgrid.plugins.
    builtins.LRUEvictor">
     <property name="maxSize" type="int" value="2"
      description="set max size for LRU Evictor" />
     <property name="numberOfLRUQueues" type="int" value="1"
      description="set number of LRU queues" />
     <property name="sleepTime" type="int" value="2" description="evictor
      thread sleep time" />
   </bean>
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

### Client side ObjectGrid XML file to override during connect

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="market">
   <backingMap name="dow" pluginCollectionRef="default" />
  </objectGrid>
 </objectGrids>
 <backingMapPluginCollections>
  <backingMapPluginCollection id="default">
   <bean id="Evictor" className="myco.og.MyEvictor">
    <property name="name" type="java.lang.String" value="MyEvictor"
     description="" />
   </bean>
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

### Application design consideration

The client connect() API is an expensive operation. Depending on the work, a client establishes one or more physical connections to a single server. The number of client connections vary between the values specified by the tcpMinConnections and tcpMaxConnections attributes that are defined within cluster element of the cluster configuration XML definition. This is the same cluster XML definition that was used to start the server. The connection manager pools these physical connections, and ObjectGrid reuses them as necessary. The tcpMinConnections and tcpMaxConnections attributes specify the number of client connections to a single server only. If a client connects to more than one server, the maximum number of client connections is less than or equal to the tcpMaxConnections attribute times number of servers the client connects to. For example, if the client connects to three servers and tcpMaxConnections is specified to be five, then the client has a maximum of `(5*3)=15` connections and a minimum of three connections, assuming setting of tcpMinConnection is `1`. Connections are shared among all clients.

The **threadsPerClientConnect** attribute specifies the number of worker threads. These worker threads dispatch the work through the physical connections. They process configuration data, client requests, server responses and system administration requests. Its default value is `5`. This attribute is available in the first iFix. If the iFix is not available, use the -Dthreads Java virtual machine (JVM) system property to specify the number of worker threads. Depending on your application, increasing this number can help with performance. As a rule, this number should not be less than the number of physical connections that the client uses.

If your application design permits, a client can create multiple threads to complete the work within one connect call by reusing the *ClientClusterContext* method.

## ObjectGrid interface

Use this topic to reference the methods that are needed to modify an ObjectGrid.

## Introduction

ObjectGrid is an extensible, transactional object caching framework that is based on the Java Map interface. The ObjectGrid API operations are grouped into a transactional unit of work and allow extensibility through custom designed plug-in support. ObjectGrid is a named logical container that contains a number of BackingMaps. For more information about backing maps, see “BackingMap interface” on page 105.

## Create and initialize

See the ObjectGridManager interface topic for the steps that are required for creating an ObjectGrid instance. Two distinct methods exist to create an ObjectGrid: programmatically or with XML configuration files. See “ObjectGridManager interface” on page 87 for more information.

## Get or set and factory methods

**Attention:** Any *set* methods must be called before you initialize the ObjectGrid instance. If you call a set method after the initialize method is called, a `java.lang.IllegalStateException` results. Each of the getSession methods of the ObjectGrid interface also implicitly call the initialize method. Therefore, you must call the set methods before calling any of the getSession methods. The only exception this rule is with the setting, adding, and removing of the EventListener objects. These objects are allowed to be processed after the ″initialize″ processing has completed.

The ObjectGrid interface contains the following methods:

*Table 7. ObjectGrid interface methods*

| Method | Description |
|---|---|
| BackingMap defineMap(String name); | *defineMap*: is a factory method to define a uniquely named BackingMap. For more information about backing maps, see “BackingMap interface” on page 105. |
| BackingMap getMap(String name); | *getMap*: Returns a BackingMap previously defined by calling *defineMap*. By using this method, you can configure the BackingMap, if it is not already configured through XML configuration. |
| BackingMap createMap(String name); | *createMap*: Creates a BackingMap, but does not cache it for use by this ObjectGrid. Use this method with the in tandem with the setMaps(List) method of the ObjectGrid interface, which caches BackingMaps for use with this ObjectGrid. Use these methods when you are configuring an ObjectGrid with the Spring Framework. |
| void setMaps(List mapList); | *setMaps*: Clears any BackingMaps that have been previously defined on this ObjectGrid and replaces them with the list of BackingMaps that is provided. |

*Table 7. ObjectGrid interface methods (continued)*

| **Method** | **Description** |
|---|---|
| public Session getSession() throws ObjectGridException, TransactionCallbackException; | *getSession*: Returns a Session, which provides begin, commit, rollback functionality for a Unit of Work. For more information about Session objects, see "Session interface" on page 109. |
| Session getSession(CredentialGenerator cg); | *getSession(CredentialGenerator cg)*: Get a session with a CredentialGenerator object. This method can only be called by the ObjectGrid client in a client server environment. |
| Session getSession(Subject subject); | *getSession(Subject subject)*: Allows the use of a specific Subject object rather than the one configured on the ObjectGrid to get a Session. |
| void initialize() throws ObjectGridException; | *initialize*: ObjectGrid is initialized and available for general use. This method is called implicitly when the `getSession` method is called, if the ObjectGrid is not in an initialized state. |
| void destroy(); | *destroy*: The framework is disassembled and cannot be used after this method is called. |

Table 7. ObjectGrid interface methods (continued)

| Method | Description |
|---|---|
| void setTxTimeout(int timeout); | *setTxTimeout*: Use this method to set the amount of time, in seconds, that a transaction that is started by a Session that this ObjectGrid instance created is allowed for completion. If a transaction does not complete within the specified amount of time, the Session that started the transaction is marked as being ″timed out″.<br><br>Marking a Session as timed out causes the next ObjectMap method that is invoked by the timed out Session to result in a `com.ibm.websphere.objectgrid.TransactionTimeoutException`<br><br>exception. The Session is marked as *rollback only*, which causes the transaction to be rolled back even if the application calls the commit method instead of the rollback method after the `TransactionTimeoutException` exception is caught by the application.<br><br>A timeout value of `0` indicates that the transaction is allowed unlimited amount of time to complete. The transaction does not time out if a time out value of `0` is used. If this method is not called, then any Session that is returned by the getSession method of this interface has a transaction timeout value set to `0` by default. An application can override the transaction timeout setting on a per Session basis by using the setTransactionTimeout method of the com.ibm.websphere.objectgrid.Session interface. |
| int getTxTimeout(); | *getTxTimeout*: Returns the transaction timeout value in seconds. This method returns the same value that is passed as the timeout parameter on the setTxTimeout method. If the setTxTimeout method was not called, then the method returns `0` to indicate that the transaction is allowed an unlimited amount of time to complete. |
| //Keywords. | |
| void associateKeyword(Serializable parent, Serializable child); | *associateKeyword*: ObjectGrid keyword provides a flexible invalidation mechanism based on keywords. For more information about keywords, see “Keywords” on page 117. This method links the two keywords together in a directional relationship. If parent is invalidated, then the child is also invalidated. Invalidating the child has no impact on the parent. |
| //Security | |

*Table 7. ObjectGrid interface methods (continued)*

| **Method** | **Description** |
|---|---|
| void setSecurityEnabled() | *setSecurityEnabled*: Enables security. Security is disabled by default. |
| void setPermissionCheckPeriod(long period); | *setPermissionCheckPeriod*: This method takes a single parameter that indicates how often to check the permission that is used to allow a client access. If the parameter is `0`, all methods ask the authorization mechanism, either JAAS authorization or custom authorization, to check if the current subject has permission. This strategy might cause performance issues depending on the authorization implementation. However, this type of authorization is available if it is required. Alternatively, if the parameter is less than `0`, it indicates the number of milliseconds to cache a set of permissions before returning to the authorization mechanism to refresh them. This parameter provides much better performance, but if the backend permissions are changed during this time the ObjectGrid might allow or prevent access even though the backend security provider has been modified. |
| void setAuthorizationMechanism(int authMechanism); | *setAuthorizationMechanism*: Set the authorization mechanism. The default is `SecurityConstants.JAAS_AUTHORIZATION`. |
| setMapAuthorization(MapAuthorization ma); | *setMapAuthorization*: Sets the MapAuthorization plug-in for this ObjectGrid instance. This plug-in can be used to authorize ObjectMap or JavaMap accesses to the principals that are contained in the Subject object. A typical implementation of this plug-in is to retrieve the principals from the Subject object, and then check if the specified permissions are granted to the principals. |
| setSubjectSource(SubjectSource ss); | *setSubjectSource*: Sets the SubjectSource plugin. This plug-in can be used to get a Subject object that represents the ObjectGrid client. This subject is used for ObjectGrid authorization. The SubjectSource.getSubject method is called by the ObjectGrid runtime when the ObjectGrid.getSession method is used to get a session and the security is enabled. This plug-in is useful for an already authenticated client: it can retrieve the authenticated Subject object and then pass to the ObjectGrid instance. Another authentication is not necessary. |

Table 7. ObjectGrid interface methods (continued)

| Method | Description |
|---|---|
| setSubjectValidation(SubjectValidation sv); | *setSubjectValidation*: Sets the SubjectValidation plugin for this ObjectGrid instance. This plug-in can be used to validate that a javax.security.auth.Subject subject that is passed to the ObjectGrid is a valid subject that has not been tampered with. An implementation of this plug-in needs support from the Subject object creator, because only the creator knows if the Subject object has been tampered with. However, a subject creator might not know if the Subject has been tampered with. In this case, this plug-in should not be used. |

## ObjectGrid interface: plug-ins

ObjectGrid interface has several optional plug-in points for more extensible interactions.

```
void addEventListener(ObjectGridEventListener cb);
void setEventListeners(List cbList);
void removeEventListener(ObjectGridEventListener cb);
void setTransactionCallback(TransactionCallback callback);
int reserveSlot(String);
// Security related plug-ins
void setSubjectValidation(SubjectValidation subjectValidation);
void setSubjectSource(SubjectSource source);
void setMapAuthorization(MapAuthorization mapAuthorization);
```

- *ObjectGridEventListener*: An ObjectGridEventListener interface is used to receive notifications when significant events occur on the ObjectGrid. These events include ObjectGrid initialization, beginning of a transaction, ending a transaction, and destroying an ObjectGrid. To listen for these events, create a class that implements the ObjectGridEventListener interface and add it to the ObjectGrid. These listeners are associated with each Session. See “Listeners” on page 177 and “Session interface” on page 109 for more information.
- *TransactionCallback*: A TransactionCallback listener interface allows transactional events such as begin, commit and rollback signals to send to this interface. Typically, a TransactionCallback listener interface is used with a Loader. For more information, see “TransactionCallback plug-in” on page 207 and “Loaders” on page 191. These events can then be used to coordinate transactions with an external resource or within multiple loaders.
- *reserveSlot*: Allows plug-ins on this ObjectGrid to reserve slots for use in object instances that have slots like TxID.
- *SubjectValidation*. If security is enabled, this plug-in can be used to validate a `javax.security.auth.Subject` class that is passed to the ObjectGrid.
- *MapAuthorization*. If security is enabled, this plug-in can be used to authorize ObjectMap accesses to the principals that are represented by the Subject object.
- *SubjectSource* If security is enabled, this plug-in can be used to get a Subject object that represents the ObjectGrid client. This subject is then used for ObjectGrid authorization.

## BackingMap interface

Each ObjectGrid instance contains a collection of BackingMap objects.

Each BackingMap is named and is added to an ObjectGrid instance by using the defineMap method or the createMap method of the ObjectGrid interface. These methods return a BackingMap instance that is then used to define the behavior of an individual Map. See "ObjectGrid interface" on page 100 for more information.

The Session interface is used to begin a transaction and to obtain the ObjectMap or JavaMap that is required for performing transactional interaction between an application and a BackingMap object. However, the transaction changes are not applied to the BackingMap object until the transaction is committed. A BackingMap can be considered as an in-memory cache of committed data for an individual Map. For more information about the Session interface, see Session interface.

The com.ibm.websphere.objectgrid.BackingMap interface provides methods for setting BackingMap attributes. Some of the set methods allow extensibility of a BackingMap through several custom designed plug-ins. Following is a list of the set methods for setting attributes and providing custom designed plug-in support:

```
 // For setting BackingMap attributes.
public void setReadOnly(boolean readOnlyEnabled);
public void setNullValuesSupported(boolean nullValuesSupported);
public void setLockStrategy( LockStrategy lockStrategy );
public void setCopyMode(CopyMode mode, Class valueInterface);
public void setCopyKey(boolean b);
public void setNumberOfBuckets(int numBuckets);
public void setNumberOfLockBuckets(int numBuckets);
public void setLockTimeout(int seconds);
public void setTimeToLive(int seconds);
public void setTtlEvictorType(TTLType type);

// For setting an optional custom plug-in provided by application.
public abstract void setObjectTransformer(ObjectTransformer t);
public abstract void setOptimisticCallback(OptimisticCallback checker);
public abstract void setLoader(Loader loader);
public abstract void setPreloadMode(boolean async);
public abstract void setEvictor(Evictor e);
public void setMapEventListeners( List /*MapEventListener*/ eventListenerList );
public void addMapEventListener(MapEventListener eventListener );
public void removeMapEventListener(MapEventListener eventListener );
public void addMapIndexPlugin(MapIndexPlugin index);
public void setMapIndexPlugins(List /* MapIndexPlugin */ indexList );
public void createDynamicIndex(String name, boolean isRangeIndex,
String attributeName, DynamicIndexCallback cb);
public void createDynamicIndex(MapIndexPlugin index, DynamicIndexCallback cb);
public void removeDynamicIndex(String name);
```

A corresponding get method exists for each of the set methods listed.

## BackingMap attributes

Each BackingMap has the following attributes that can be set to modify or control the BackingMap behavior:

- *ReadOnly* attribute. This attribute indicates if the Map is a read-only Map or a read and write Map. If this attribute is never set for the Map, then the Map is defaulted to be a read and write Map. When a BackingMap is set to be read only, ObjectGrid optimizes performance for read only when possible.
- *NullValuesSupported* attribute. This attribute indicates if a null value can be put into the Map. If this attribute is never set, the Map does not support null values. If null values are supported by the Map, a get operation that returns null can mean that either the value is null or the map does not contain the key specified by the get operation.

- *LockStrategy* attribute. This attribute determines if a lock manager is used by this BackingMap. If a lock manager is used, then the LockStrategy attribute is used to indicate whether an optimistic locking or pessimistic locking approach is used for locking the map entries. If this attribute is not set, then the optimistic LockStrategy is used. See the “Locking” on page 123 topic for details on the supported lock strategies.
- *CopyMode* attribute. This attribute determines if a copy of a value object is made by the BackingMap when a value is read from the map or is put into the BackingMap during the commit cycle of a transaction. Various copy modes are supported to allow the application to make the trade-off between performance and data integrity. If this attribute is not set, then the `COPY_ON_READ_AND_COMMIT` copy mode is used. This copy mode does not have the best performance, but it has the greatest protection against data integrity problems. For more information about the copy modes, see copyMode method best practices.
- *CopyKey* attribute. This attribute determines if the BackingMap makes a copy of a key object when an entry is first created in the map. The default action is to not make a copy of key objects because keys are normally unchangeable objects.
- *NumberOfBuckets* attribute. This attribute indicates the number of hash buckets to be used by the BackingMap. The BackingMap implementation uses a hash map for its implementation. If a lot of entries exist in the BackingMap, then more buckets means better performance. The number of keys that have the same bucket becomes lower as the number of buckets grows. More buckets also mean more concurrency. This attribute is useful for fine tuning performance. A default value of `503` is used if the application does not set the NumberOfBuckets attribute.
- *NumberOfLockBuckets* attibute. This attribute indicates the number of lock buckets that are be used by the lock manager for this BackingMap. When the LockStrategy is set to OPTIMISTIC or PESSIMISTIC, a lock manager is created for the BackingMap. The lock manager uses a hash map to keep track of entries that are locked by one or more transactions. If a lot of entries exist in the hash map, more lock buckets lead to better performance because the number of keys that collide on the same bucket is lower as the number of buckets grows. More lock buckets also means more concurrency. When the LockStrategy attribute is set to NONE, no lock manager is used by this BackingMap. In this case, setting numberOfLockBuckets has no effect. If this attribute is not set, a default value of `383` is used .
- *LockTimeout* attribute. This attribute is used when the BackingMap is using a lock manager. The BackingMap uses a lock manager when the the LockStrategy attribute is set to either OPTIMISTIC or PESSIMISTIC. The attribute value is in seconds and determines how long the lock manager waits for a lock to be granted. If this attribute is not set, then 15 seconds is used a the LockTimeout value. See Pessimistic locking for details regarding the lock wait timeout exceptions that can occur.
- *TtlEvictorType* attribute. Every BackingMap has its own built in time to live evictor that uses a time-based algorithm to determine which map entries to evict. By default, the built in time to live evictor is not active. You can activate the time to live evictor by calling the setTtlEvictorType method with one of three values: `CREATION_TIME`, `LAST_ACCESS_TIME`, or `NONE`. A value of `CREATION_TIME` indicates that the evictor adds the TimeToLive attribute to the time that the map entry was created in the BackingMap to determine when the evictor should evict the map entry from the BackingMap. A value of `LAST_ACCESS_TIME` indicates that the evictor adds the TimeToLive attribute to the time that the map entry was last accessed by some transaction that the application is running to determine when evictor should evict the map entry. The map entry is evicted only if a map entry is

never accessed by any transaction for a period of time that is specified by the TimeToLive attribute. A value of NONE indicates the evictor should remain inactive and never evict any of the map entries. If this attribute is never set, then NONE is used as the default and the time to live evictor is not active. See Evictors for details regarding the built-in time to live evictor.

- *TimeToLive* attribute. This attribute is used to specify the number of seconds that the built in time to live evictor needs to add to the creation or last access time for each entry as described for the TtlEvictorType attribute. If this attribute is never set, then the special value of zero is used to indicate the time to live is infinity. If this attribute is set to infinity, map entries are never evicted by the evictor.

The following example illustrates defining The someMap BackingMap in the someGrid ObjectGrid instance and setting various attributes of the BackingMap by using the set methods of the BackingMap interface:

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;

...

ObjectGrid og =
 ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("someGrid");
BackingMap bm = objectGrid.getMap("someMap");
bm.setReadOnly( true ); // override default of read/write
bm.setNullValuesSupported(false); // override default of allowing Null values
bm.setLockStrategy( LockStrategy.PESSIMISTIC ); // override default of OPTIMISTIC
bm.setLockTimeout( 60 ); // override default of 15 seconds.
bm.setNumberOfBuckets(251); // override default (prime numbers work best)
bm.setNumberOfLockBuckets(251); // override default (prime numbers work best)

...
```

## BackingMap plug-ins

The BackingMap interface has several optional plug points for more extensible interactions with the ObjectGrid:

- **ObjectTransformer plug-in**: For some map operations, a BackingMap might need to serialize, deserialize, or copy a key or value of an entry in the BackingMap. The BackingMap can perform these actions by providing a default implementation of the ObjectTransformer interface. An application can improve performance by providing a custom designed ObjectTransformer plug-in that is used by the BackingMap to serialize, deserialize, or copy a key or value of an entry in the BackingMap. See “ObjectTransformer plug-in” on page 202 for more information.
- **Evictor plug-in**: The built in time to live evictor uses a time-based algorithm to decide when an entry in BackingMap must be evicted. Some applications might need to use a different algorithm for deciding when an entry in a BackingMap needs to be evicted. The Evictor plug-in makes a custom designed Evictor available to the BackingMap to use. The Evictor plug-in is in addition to the built in time to live evictor. It does not replace the time to live evictor. ObjectGrid provides a custom Evictor plug-in that implements well-known algorithms such as ″least recently used″ or ″least frequently used″. Applications can either plug-in one of the provided Evictor plug-ins or it can provide its own Evictor plug-in. See “Evictors” on page 182 for more information.
- **MapEventListener plug-in**: An application might want to know about BackingMap events such as a map entry eviction or a preload of a BackingMap completion. A BackingMap calls methods on the MapEventListener plug-in to

notify an application of BackingMap events. An application can receive notification of various BackingMap events by using the setMapEventListener method to provide one or more custom designed MapEventListener plug-ins to the BackingMap. The application can modify the listed MapEventListener objects by using the addMapEventListener method or the removeMapEventListener method. See "MapEventListener interface" on page 180 for more information.

- **Loader plug-in**: A BackingMap is an in-memory cache of a Map. A Loader plug-in is an option that is used by the BackingMap to move data between memory and is used for a persistent store for the BackingMap. For example, a Java database connectivity (JDBC) Loader can be used to move data in and out of a BackingMap and one or more relational tables of a relational database. A relational database does not need to be used as the persistent store for a BackingMap. The Loader can also be used to moved data between a BackingMap and a file, between a BackingMap and a Hibernate map, between a BackingMap and a Java 2 Platform, Enterprise Edition (J2EE) entity bean, between a BackingMap and another application server, and so on. The application must provide a custom-designed Loader plug-in to move data between the BackingMap and the persistent store for every technology that is used. If a Loader is not provided, the BackingMap becomes a simple in-memory cache. See "Loaders" on page 191 for more information about this plug-in.
- **OptimisticCallback plug-in**: When the LockStrategy attribute for a BackingMap is set to OPTIMISTIC, either the BackingMap or a Loader plug-in must perform comparison operations for the values of the map. The OptimisticCallback plug-in is used by the BackingMap and the Loader to perform the optimistic versioning comparison operations. See "OptimisticCallback interface" on page 213 for more information.
- **MapIndexPlugin plug-in**: A MapIndexPlugin plug-in, or an Index in short, is an option that is used by the BackingMap to build an index that is based on the specified attribute of the stored object. The index allows the application to find objects by a specific value or a range of values. There are two types of index: static and dynamic. Refer to "Indexing" on page 226 for detailed information.

## Session interface

This section describes how applications begin and end transactions using the Session interface. The Session interface also provides access to the application based ObjectMap and JavaMap interfaces.

### Introduction

Each ObjectMap or JavaMap instance is directly tied to a specific Session object. Each thread that wants access to an ObjectGrid must first obtain a Session from the ObjectGrid object. A Session instance cannot be shared concurrently between threads. ObjectGrid does not use any thread local storage, but platform restrictions might limit the opportunity to pass a Session from one thread to another.

### Methods

The following methods are available with the Session interface. See the API documentation for more information about the following methods:

```
public interface Session {
 ObjectMap getMap(String cacheName)
 throws UndefinedMapException;

 void begin()
 throws TransactionAlreadyActiveException, TransactionException;
 void beginNoWriteThrough()
 throws TransactionAlreadyActiveException, TransactionException;

 public void commit()
 throws NoActiveTransactionException, TransactionException;

 public void rollback()
 throws NoActiveTransactionException, TransactionException;

 public void flush()
 throws TransactionException; ObjectGrid getObjectGrid();

 TxID getTxID()
 throws NoActiveTransactionException;

 boolean isWriteThroughEnabled();

 void setTransactionType(String tranType);

 public void processLogSequence(LogSequence logSequence)
 throws NoActiveTransactionException, UndefinedMapException, ObjectGridException;


public ObjectGrid getObjectGrid();

public void setTransactionTimeout(int timeout);
public int getTransactionTimeout();
public boolean transactionTimedOut();

public boolean isCommitting();
public boolean isFlushing();

public void markRollbackOnly(Throwable t) throws NoActiveTransactionException;
public boolean isMarkedRollbackOnly();
}
```

## Get method

An application obtains a Session instance from an ObjectGrid object using the ObjectGrid.getSession method. The following code snippet demonstrates how to obtain a Session instance:

```
ObjectGrid objectGrid = ...;
Session sess = objectGrid.getSession();
```

After a Session is obtained, the thread keeps a reference to the session for its own use. Calling the getSession method multiple times returns a new Session object each time.

## Transactions and sessions methods

A Session can be used to begin, commit, or rollback transactions. Operations against BackingMaps using ObjectMaps and JavaMaps are most efficiently performed within a Session transaction. After a transaction has started, any changes to one or more BackingMaps in that transaction scope are stored in a special transaction cache until the transaction is committed. When a transaction is committed, the pending changes are applied to the BackingMaps and Loaders and become visible to any other clients of that ObjectGrid.

ObjectGrid also supports the ability to automatically commit transactions, also known as auto-commit. If any ObjectMap operations are performed outside of the context of an active transaction, an implicit transaction is started before the operation and the transaction is automatically committed before returning control to the application.

```
Session session = objectGrid.getSession();
ObjectMap objectMap = session.getMap("someMap");
session.begin();
objectMap.insert("key1", "value1");
objectMap.insert("key2", "value2");
session.commit();
objectMap.insert("key3", "value3"); // auto-commit
```

## Session.flush method

The Session.flush method only makes sense when a Loader is associated with a BackingMap. The flush method invokes the Loader with the current set of changes in the transaction cache. The Loader applies the changes to the backend. These changes are not committed when the flush is invoked. If a Session transaction is committed after a flush invocation, only updates that happen after the flush invocation are applied to the Loader. If a Session transaction is rolled back after a flush invocation, the flushed changes are discarded with all other pending changes in the transaction. Use the Flush method sparingly because it limits the opportunity for batch operations against a Loader. Following is an example of the usage of the Session.flush method:

```
Session session = objectGrid.getSession();
session.begin();
// make some changes
...
session.flush(); // push these changes to the Loader, but don't commit yet
// make some more changes
...
session.commit();
```

## No write through method

Some ObjectGrid maps are backed by a Loader, which provides persistent storage for the data in the map. Sometimes it is useful to commit data just to the ObjectGrid map and not push data out to the Loader. The Session interface provides the beginNoWriteThough method for this purpose. The beginNoWriteThrough method starts a transaction like the begin method. With the beginNoWriteThrough method, when the transaction is committed, the data is only committed to the ObjectGrid in-memory map and is not committed to the persistent storage that is provided by the Loader. This method is very useful when performing data preload on the map.

When using a distributed ObjectGrid instance, the beginNoWriteThrough method is useful for making changes to the near cache only, without modifying the far cache on the server. If the data is known to be stale in the near cache, using the beginNoWriteThrough method can allow entries to be invalidated on the near cache without invalidating them on the server as well.

The Session interface also provides the isWriteThroughEnabled method to determine what type of transaction is currently active.

```
Session session = objectGrid.getSession();
session.beginNoWriteThrough();
// make some changes ...
session.commit(); // these changes will not get pushed to the Loader
```

## Obtain the TxID object method

The TxID object is an opaque object that identifies the active transaction. Use the TxID object for the following purposes:

- For comparison when you are looking for a particular transaction.
- To store shared data between the TransactionCallback and Loader objects.

See “TransactionCallback plug-in” on page 207 and “Loaders” on page 191 for additional information about the Object slot feature.

## Set the transaction type for performance monitoring method

If you are using ObjectGrid within a WebSphere Application Server application server, it might be necessary to reset the transaction type for performance monitoring. You can set the transaction type with the setTransactionType method. See “Monitoring ObjectGrid performance with WebSphere Application Server performance monitoring infrastructure (PMI)” on page 283 for more information about the setTransactionType method.

## Process a complete LogSequence method

ObjectGrid can propagate sets of map changes to other ObjectGrid listeners as a means of distributing maps from one Java Virtual Machine (JVM) to another. To make it easier for the listener to process the received LogSequences, the Session interface provides the processLogSequence method. This method examines each LogElement within the LogSequence and performs the appropriate operation, for example, insert, update, invalidate, and so on, against the BackingMap that is identified by the LogSequence MapName. An ObjectGrid Session must be active before the processLogSequence method is invoked. The application is also responsible for issuing the appropriate commit or rollback calls to complete the Session. Autocommit processing is not available for this method invocation.

Normal processing by the receiving ObjectGridEventListener at the remote JVM would be to start a Session using the beginNoWriteThrough method, which prevents endless propagation of changes, followed by a call to this processLogSequence method, and then committing or rolling back the transaction.

```
// Use the Session object that was passed in during
//ObjectGridEventListener.initialization...
session.beginNoWriteThrough();
// process the received LogSequence
try {
 session.processLogSequence(receivedLogSequence);
} catch (Exception e) {
 session.rollback(); throw e;
}
// commit the changes
session.commit();
```

## markRollbackOnly method

This method is used to mark the current transaction as ″rollback only″. Marking a transaction ″rollback only″ ensures that even if the commit method is called by application, the transaction is rolled back. This method is typically used by ObjectGrid itself or by the application when it knows that data corruption could occur if the transaction was allowed to be committed.

After this method is called, the Throwable object that is passed to this method is chained to the `com.ibm.websphere.objectgrid.TransactionException` exception that results by the commit method if it is called on a Session that was previously marked a ″rollback only″. Any subsequent calls to this method for a transaction that is already marked as ″rollback only″ is ignored. That is, only the first call that passes a non-null Throwable reference is used. Once the marked transaction is completed, the ″rollback only″ mark is removed so that the next transaction that is started by the Session can be committed.

### isMarkedRollbackOnly method

Returns if Session is currently marked as ″rollback only″. Boolean true is returned by this method if and only if markRollbackOnly method was previously called on this Session and the transaction started by the Session is still active.

### setTransactionTimeout method

Set transaction timeout for next transaction started by this Session to a specified number of seconds. This method does not affect the transaction timeout of any transactions previously started by this Session. It only affects transactions that are started after this method is called. If this method is never called, then the timeout value that was passed to the setTxTimeout method of the com.ibm.websphere.objectgrid.ObjectGrid method is used.

### getTransactionTimeout method

This method returns the transaction timeout value in seconds. The last value that was passed as the timeout value to the setTransactionTimeout method is returned by this method. If the setTransactionTimeout method is never called, then the timeout value that was passed to the setTxTimeout method of the com.ibm.websphere.objectgrid.ObjectGrid method is used.

### transactionTimedOut

This method returns boolean true if the current transaction that was started by this Session has timed out.

### isFlushing method

This method returns boolean true if and only if all transaction changes are being flushed out to the Loader plugin as a result of the flush method of Session interface being invoked. A Loader plugin may find this method useful when it needs to know why its batchUpdate method was invoked.

### isCommitting method

This method returns boolean true if and only if all transaction changes are being committed as a result of the commit method of Session interface being invoked. A Loader plug-in might find this method useful when it needs to know why its batchUpdate method was invoked.

## ObjectMap and JavaMap interfaces

This topic describes how applications interact with ObjectGrid using the ObjectMap and JavaMap interfaces. These two interfaces are used for transactional interaction between applications and BackingMaps.

## ObjectMap interface

An ObjectMap instance is obtained from a Session object that corresponds to the current thread. The ObjectMap interface is the main vehicle that applications use to make changes to entries in a BackingMap.

### Obtain an ObjectMap instance

An application gets an ObjectMap instance from a Session object using the Session.getMap(String) method. The following code snippet demonstrates how to obtain an ObjectMap instance:

```
ObjectGrid objectGrid = ...;
BackingMap backingMap = objectGrid.defineMap("mapA");
Session sess = objectGrid.getSession();
ObjectMap objectMap = sess.getMap("mapA");
```

Each ObjectMap instance corresponds to a particular Session object. Calling the getMap method multiple times on a particular Session object with the same BackingMap name always returns the same ObjectMap instance.

### Autocommit Transactions

As previously stated, operations against BackingMaps that use ObjectMaps and JavaMaps are most efficiently performed within a Session transaction. ObjectGrid provides autocommit support when methods on the ObjectMap and JavaMap interfaces are called outside of a Session transaction. The methods start an implicit transaction, perform the requested operation, and commit the implicit transaction.

### Method Semantics

Following is an explanation of the semantics behind each method on the ObjectMap and JavaMap interfaces. The setDefaultKeyword method, the invalidateUsingKeyword method, and the methods that have a Serializable argument are discussed in the “Keywords” on page 117 topic. The setTimeToLive method is discussed in the “Evictors” on page 182 topic. See the API documentation for more information on these methods.

**containsKey method**
: Determines if a key has a value in the BackingMap or Loader. If null values are supported by an application, this method can be used to determine if a null reference that is returned from a get operation refers to a null value or indicates that the BackingMap and Loader do not contain the key.

**flush method**
: The semantics of this method are similar to the flush method on the Session interface. The notable difference is that the Session flush applies the current pending changes for all of the maps that have been modified in the current session. With this method, only the changes in this ObjectMap are flushed to the loader.

**get method**
: Fetches the entry from the BackingMap. If the entry is not found in the BackingMap but a Loader is associated with the BackingMap, it attempts to fetch the entry from the Loader. The getAll method is provided to allow batch fetch processing.

**getForUpdate method**
: The getforUpdate method is the same as the get method, but using the

getForUpdate method tells the BackingMap and Loader that the intention is to update the entry. A Loader can use this hint to issue a `SELECT for UPDATE` query to a database backend. If a Pessimistic LockingStrategy is defined for the BackingMap, the lock manager locks the entry. The getAllForUpdate method is provided to allow batch fetch processing.

**insert method**
Inserts an entry into the BackingMap and the Loader. Using this method tells the BackingMap and Loader that you want to insert a previously nonexistent entry. When you invoke this method on an existing entry, an exception occurs when the method is invoked or when the current transaction is committed.

**invalidate method**
The semantics of the invalidate method depend on the value of the **isGlobal** parameter that is passed to the method. The invalidateAll method is provided to allow batch invalidate processing.

Local invalidation is specified when the value *false* is passed as the **isGlobal** parameter of the invalidate method. Local invalidation discards any changes to the entry in the transaction cache. If the application issues a `get` method, the entry is fetched from the last committed value in the BackingMap. If no entry is present in the BackingMap, the entry is fetched from the last flushed or committed value in the Loader. When a transaction is committed, any entries that are marked as being locally invalidated have no impact on the BackingMap. Any changes that were flushed to the Loader are still committed even if the entry was invalidated.

Global invalidation is specified when *true* is passed as the **isGlobal** parameter of the invalidate method. Global invalidation discards any pending changes to the entry in the transaction cache and bypasses the BackingMap value on subsequent operations that are performed on the entry. When a transaction is committed, any entries that are marked as globally invalidated are evicted from the BackingMap.

Consider the following use case for invalidation as an example: The BackingMap is backed by a database table that has an auto increment column. Increment columns are useful for assigning unique numbers to records. The application inserts an entry. After the insert, the application needs to know the sequence number for the inserted row. It knows that its copy of the object is old, so it uses global invalidation to get the value from the Loader. The following code demonstrates this use case:

```
Session sess = objectGrid.getSession();
ObjectMap map = sess.getMap("mymap");
sess.begin();
map.insert("Billy", new Person("Joe", "Bloggs", "Manhattan"));
sess.flush();
map.invalidate("Billy", true);
Person p = map.get("Billy");
System.out.println("Version column is: " + p.getVersion());
map.commit();
```

This code sample adds an entry for *Billy*. The version attribute of Person is set using an auto-increment column in the database. The application does an insert command first. It then issues a flush, which causes the insert to be sent to the Loader and database. The database sets the version column to the next number in the sequence, which makes the Person object in the transaction outdated. To update the object, the application performs a global invalidate. The next get method that is issued gets the entry from the

Loader ignoring the transaction's value. The entry is fetched from the database with the updated version value.

**put method**
: The semantics of the put method are dependent on whether a previous get method was invoked in the transaction for the key. If the application issues a get operation that returns an existent entry in the BackingMap or Loader, the put method invocation is interpreted as an update and returns the previous value in the transaction. A put method invocation without a previous get method invocation or a previous get method invocation that did not find an entry is interpreted as an insert. The semantics of the insert and update methods apply when the put operation is committed. The putAll method is provided to enable batch insert and update processing.

**remove method**
: Removes the entry from the BackingMap and the Loader, if one is plugged in. The value of the object that was removed is returned by this mehtod. If the object does not exist, this method returns a null value. The removeAll method is provided to enable batch deletion processing without the return values.

**setCopyMode method**
: Specifies a CopyMode for this ObjectMap. With this method, an application can override the CopyMode that is specified on the BackingMap. The specified CopyMode is in effect until clearCopyMode method is invoked. Both methods are invoked outside of transactional bounds. A CopyMode cannot be changed in the middle of a transaction.

**touch method**
: Updates the last access time for an entry. This method does not retrieve the value from the BackingMap. Use this method in its own transaction. If the provided key does not exist in the BackingMap due to invalidation or removal, an exception occurs during commit processing.

**update method**
: Explicitly updates an entry in the BackingMap and the Loader. Using this method indicates to the BackingMap and Loader that you want to update an existing entry. An exception occurs if you invoke this method on an entry that does not exist when the method is invoked or during commit processing.

**getIndex method**
: Attempts to obtain a named index that is built on the BackingMap. The index cannot be shared between threads and works on the same rules as a Session. The returned index object should be cast to the right application index interface such as the MapIndex interface, the MapRangeIndex interface, or a custom index interface.

## JavaMap interface

A JavaMap instance is obtained from an ObjectMap object. The JavaMap interface has the same method signatures as ObjectMap, but with different exception handling. JavaMap extends the java.util.Map interface, so all exceptions are instances of the java.lang.RuntimeException class. Because JavaMap extends the java.util.Map interface, it is easy to quickly use ObjectGrid with an existing application that uses a java.util.Map interface for object caching.

### Obtain a JavaMap instance

An application gets a JavaMap instance from an ObjectMap object using the ObjectMap.getJavaMap method. The following code snippet demonstrates how to obtain a JavaMap instance.

```
ObjectGrid objectGrid = ...;
BackingMap backingMap = objectGrid.defineMap("mapA");
Session sess = objectGrid.getSession();
ObjectMap objectMap = sess.getMap("mapA");
java.util.Map map = objectMap.getJavaMap();
JavaMap javaMap = (JavaMap) javaMap;
```

A JavaMap is backed by the ObjectMap from which it was obtained. Calling getJavaMap multiple times using a particular ObjectMap always returns the same JavaMap instance.

**Supported methods**

The JavaMap interface only supports a subset of the methods on the java.util.Map interface. The the java.util.Map interface supports the following methods:

- `containsKey(java.lang.Object)`
- `get(java.lang.Object)`
- `put(java.lang.Object, java.lang.Object)`
- `putAll(java.util.Map)`
- `remove(java.lang.Object)`

All other methods inherited from the java.util.Map interface result in the `java.lang.UnsupportedOperationException` exception.

## Keywords

ObjectGrid provides a flexible invalidation mechanism based around keywords. A *keyword* is a non-null instance of any serializable object. You can associate keywords with BackingMap entries in any way you choose.

### Associate keywords with entries

A set of entries can be associated with zero or more keywords. The methods on ObjectMap and JavaMap that manipulate entries, including the `get`, `update`, `put`, `insert`, and `touch` methods, all have versions that allow a single keyword to be associated with all of the entries that the method alters. New keyword associations are only visible in the current transaction until the transaction is committed. After a commit, the new association is applied to the BackingMap and is visible to other transactions. If an error occurs during commit processing resulting in a rollback or if a user rolls back an active transaction, the new keyword associations are rolled back. The following code demonstrates how a new entry is associated with a keyword:

```
Session sess = objectGrid.getSession();
ObjectMap map = sess.getMap("MapA");
sess.begin();
map.insert("Billy", new Person("Joe", "Bloggs", "Manhattan"), "New York");
sess.commit();
```

The previous example code inserts a new entry into the BackingMap and associates it with the keyword ″`New York`″. An application that inserts entries must also associate keywords when the entries are retrieved. The application must associate keywords with entries every time it gets them. Consider the following code sample:

```
sess.begin();
Person p = (Person)map.get("Billy", "New York");
sess.commit();
```

The previous example code ensures that the retrieved entry is associated with the ″New York″ keyword. An application can associate multiple keywords with an entry, but only one keyword per method invocation. To associate more keywords issue another method invocation, like the following sample:

```
sess.begin();
Person p = (Person)map.get("Billy", "New York");
map.touch("Billy", "Another keyword");
map.get("Billy", "Yet another keyword");
sess.commit();
```

## Default keywords

The `setDefaultKeyword` method on the ObjectMap and JavaMap interfaces provides a way to associate entries with a particular keyword without using the keyword version of the `get`, `insert`, `put`, `update`, or `touch` methods. If the keyword version of a method is used, the default keyword is ignored, and the supplied keyword object is used.

```
sess.begin();
map.setDefaultKeyword("New York");
Person p = (Person)map.get("Billy");
p = (Person)map.get("Bob", "Los Angeles");
map.setDefaultKeyword(null);
p = (Person)map.get("Jimmy");
sess.commit();
```

In the preceding example `Billy` is associated with the default keyword, ″New York″. `Bob` is not associated with the default keyword because an explicit keyword was passed to the get invocation to retrieve the Bob entry. No keywords are associated with ″Jimmy″ because the default keyword was reset and no explicit keyword argument was passed to the `get` method invocation.

## Invalidate entries with keywords

Using the `invalidateUsingKeyword` method on the ObjectMap and JavaMap interfaces invalidates all entries that are associated with a keyword in the corresponding BackingMap. With this approach you can efficiently invalidate related entries in a single operation.

```
sess.begin();
map.insert("Billy", new Person("Joe", "Bloggs", "Manhattan"), "New York");
map.invalidateUsingKeyword("New York", false);
map.insert("Bob", new Person("Paul", "Henry", "Albany"), "New York");
sess.commit();
```

In the preceding example, the entry for ″Billy″ is invalidated and is not inserted into the BackingMap. The entry for ″Bob″ is not invalidated because it was inserted after the `invalidateUsingKeyword` method invocation. The `invalidateUsingKeyword` method invalidates entries based on the keyword associations when the method is invoked.

## Keyword grouping

Keywords can also be grouped together in a parent-child relationship. A parent keyword can have multiple children, and a child keyword can have multiple parents.

For example, if an application uses the keywords ″Dublin″, ″Paris″, ″New York″, and ″Los Angeles″, it can add the following keyword groupings:

- ″USA″ groups ″New York″ and ″Los Angeles″
- ″Europe″ groups ″Dublin″ and ″Paris″
- ″World″ groups ″USA″ and ″Europe″

Invalidating the keyword ″USA″ invalidates all entries that are associated with the ″New York″ and ″Los Angeles″ keywords. Invalidating the ″World″ keyword invalidates all entries that are associated with the ″USA″ and ″Europe″ groupings. Keyword associations are defined using the `associateKeyword` method on the ObjectGrid interface. Adding child keywords to a parent keyword after an `invalidateUsingKeyword` method invocation does not cause the entries associated with the child keyword to be invalidated. The following example code defines the set of keyword associations that are described:

```
ObjectGrid objectGrid = ...;
objectGrid.associateKeyword("USA", "New York");
objectGrid.associateKeyword("USA", "Los Angeles");
objectGrid.associateKeyword("Europe", "Dublin");
objectGrid.associateKeyword("Europe", "Paris");
objectGrid.associateKeyword("World", "USA");
objectGrid.associateKeyword("World", "Europe");
```

## LogElement and LogSequence objects

When an application is making changes to a Map during a transaction, a LogSequence object tracks those changes. If the application changes an entry in the map, a corresponding LogElement exists to provide the details of the change. Loaders are given a LogSequence object for a particular map whenever an application calls for a flush or commit to the transaction. The Loader iterates over the LogElements within the LogSequence and applies each LogElement to the backend.

ObjectGridEventListeners registered with an ObjectGrid also make use of LogSequence objects. These listeners are given a LogSequence object for each map in a committed transaction. Applications can use these listeners to wait for certain entries to change, like a trigger in a conventional database.

This topic describes four log-related interfaces or classes that are provided by the ObjectGrid framework:

- `com.ibm.websphere.objectgrid.plugins.LogElement`
- `com.ibm.websphere.objectgrid.plugins.LogSequence`
- `com.ibm.websphere.objectgrid.plugins.LogSequenceFilter`
- `com.ibm.websphere.objectgrid.plugins.LogSequenceTransformer`

### LogElement interface

A LogElement represents an operation on an entry during a transaction. A LogElement object has the following attributes. The most commonly used attributes the *type* and the *current value* attributes:

***type* attribute**

A log element *type* indicates the kind of operation that this log element represents. The *type* can be one of the following constants that are defined in the `LogElement` interface: `INSERT`, `UPDATE`, `DELETE`, `EVICT`, `FETCH`, or `TOUCH`.

*undo type* **attribute**
: Returns what operation must be performed to "undo" a prior change that the transaction made to the map entry.

*current value* **attribute**
: The *current value* represents the new value for the operation `INSERT`, `UPDATE` or `FETCH`. If the operation is `TOUCH`, `DELETE`, or `EVICT`, the current value is null. This value can be cast to ValueProxyInfo when a ValueInterface is in use.

*CacheEntry* **attribute**
: You can get a reference to the CacheEntry object from the LogElement and use the methods defined on the CacheEntry object to retrieve needed information.

*pending state* **attribute**
: If the *pending state* is `true`, the change represented by this log element has not been applied to the loader yet. If it is `false`, the change has been applied to the loader, most likely by the flush operation.

*versioned value* **attribute**
: Versioned value is a value that can be used for versioning.

*new keywords* **attribute**
: The new keyword collection contains any new keywords that have been associated with this entry.

*last access time* **attribute**
: Represents the last access time for the entry.

*before image / after image* **attributes**
: Getter methods are available to get the image of the value object before or after the changes were applied to the map.

## LogSequence interface

In most transactions, operations to more than one entry in a map occur, so multiple LogElement objects are created. It makes sense to have an object that acts as a composite of multiple LogElement objects. The LogSequence interface serves this purpose by containing a list of LogElement objects. The LogSequence interface has the following methods:

**size method**
: Returns the number of LogElement objects in the specified sequence.

**getAllChanges method**
: Returns an iterator of all the changes in the specified log sequence.

**getPendingChanges method**
: Returns an iterator of all the pending changes. This is most likely to be used by a loader to only apply pending changes to the persistent store.

**getChangesByKeys method**
: Returns an iterator of the LogElement objects that have the target key, based on the input parameter.

**getChangesByTypes method**
: Returns an iterator of the LogElement objects that are of the specified LogElement type.

**getMapName method**
: Returns the name of the backing map to which the changes apply. The caller can use this name as input to the `Session.getMap(string)` method.

**isDirty method**
: Returns whether this LogSequence has any LogElements that would *dirty* a Map. That is, if the LogSequence contains any LogElement objects that are of any type other than Fetch or Get, then the LogSequence is considered ″dirty″.

**isRollback method**
: Returns if this LogSequence was generated to roll back a transaction.

**getObjectGridName method**
: Returns the name of the ObjectGrid that houses the map for which these changes apply.

LogElement and LogSequence are widely used in ObjectGrid and by ObjectGrid plug-ins that are written by users when operations are propagated from one component or server to another component or server. For example, a LogSequence object can be used by the distributed ObjectGrid transaction propagation function to propagate the changes to other servers, or it can be applied to the persistence store by the loader. LogSequence is mainly used by the following interfaces.

- `com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener`
- `com.ibm.websphere.objectgrid.plugins.Loader`
- `com.ibm.websphere.objectgrid.plugins.Evictor`
- `com.ibm.websphere.objectgrid.Session`

For more details about these interfaces, please refer to the API documentation.

## Loader example

This section demonstrates how the LogSequence and LogElement objects are used in a Loader. A *Loader* is used to load data from and persist data into a persistent store. The `batchUpdate` method of the Loader interface uses LogSequence:

```
void batchUpdate(TxID txid, LogSequence sequence)
throws LoaderException, OptimisticCollisionException;
```

The `batchUpdate` method is called whenever an ObjectGrid needs to apply all current changes to the Loader. The Loader is given a list of LogElement objects for the map, encapsulated in a LogSequence object. The implementation of the `batchUpdate` method must iterate over the changes and apply them to the backend. The following code snippet shows how a Loader uses a LogSequence object. The snippet iterates over the set of changes and builds up three batch Java database connectivity (JDBC) statements: one that has inserts, one that has updates and, a third statement that has deletes:

```
public void batchUpdate(TxID tx, LogSequence sequence)
throws LoaderException
{
 // Get a SQL connection to use.
 Connection conn = getConnection(tx);
 try
 {
  // Process the list of changes and build a set of prepared
  // statements for executing a batch update, insert, or delete
  // SQL operations. The statements are cached in stmtCache.
  Iterator iter = sequence.getPendingChanges();
  while ( iter.hasNext() )
  {
   LogElement logElement = (LogElement)iter.next();
   Object key = logElement.getCacheEntry().getKey();
   Object value = logElement.getCurrentValue();
   switch ( logElement.getType().getCode() )
  {
   case LogElement.CODE_INSERT:
   buildBatchSQLInsert( key, value, conn );
   break;
   case LogElement.CODE_UPDATE:
   buildBatchSQLUpdate( key, value, conn );
   break;
   case LogElement.CODE_DELETE:
   buildBatchSQLDelete( key, conn );
   break;
  }
 }
 // Run the batch statements that were built by above loop.
 Collection statements = getPreparedStatementCollection( tx, conn );
 iter = statements.iterator();
 while ( iter.hasNext() )
 {
  PreparedStatement pstmt = (PreparedStatement) iter.next();
  pstmt.executeBatch();
 }
}
catch (SQLException e)
{
 LoaderException ex = new LoaderException(e);
 throw ex;
}
}
```

The previous sample illustrates the high level logic of processing the LogSequence argument and the details of how an SQL `insert`, `update`, or `delete` statement is built are not illustrated. This example illustrates that the `getPendingChanges` method is called on LogSequence argument to obtain an iterator of LogElement objects that a Loader needs to process, and the `LogElement.getType().getCode()` method is used to determine whether a LogElement is for a SQL `insert`, `update`, or `delete` operation.

## Evictor sample

This example explores how LogSequence and LogElement are used in an Evictor. An Evictor is used to evict the map entries from the backing map based on certain criteria. The apply method of the Evictor interface uses LogSequence:

```
/**
* This is called during cache commit to allow the evictor to track object usage
* in a backing map. This will also report any entries that have been successfully
* evicted.
*
* @param sequence LogSequence of changes to the map
*/
void apply(LogSequence sequence);
```

For information on how the apply method uses LogSequence, refer to the code sample in the “Evictors” on page 182 topic.

## LogSequenceFilter and LogSequenceTransformer interfaces

Sometimes, it is necessary to filter the LogElement objects so that only LogElement objects with certain criteria are accepted, and reject other objects. For example, you might want to serialize a certain LogElement based on some criterion. LogSequenceFilter solves this problem with the following method:

```
public boolean accept (LogElement logElement);
```

This method returns true if the given LogElement should be used in the operation, and returns false if the given LogElement should not be used.

LogSequenceTransformer is a class which utilizes the LogSequenceFilter function described above. It uses the LogSequenceFilter to filter out some LogElement objects and then serialize the accepted LogElement objects. This class has two methods. The first method follows:

```
public static void serialize(Collection logSequences, ObjectOutputStream stream,
LogSequenceFilter filter, DistributionMode mode)
throws IOException
```

This method allows the caller to provide a filter for determining which LogElements to include in the serialization process. The **DistributionMode** parameter allows the caller to control the serialization process. For example, if the distribution mode is invalidation only, then there is no need to serialize the value. The second method of this class follows:

```
public static Collection inflate(ObjectInputStream stream, ObjectGrid objectGrid)
throws IOException, ClassNotFoundException.
```

This method reads the log sequence serialized form, which was created by the `serialize` method, from the provided object input stream.

## Locking

This topic describes the locking strategy that is supported by an ObjectGrid BackingMap.

Each BackingMap can be configured to use one of the following locking strategies:

- Pessimistic locking
- Optimistic locking
- None

Following is an example of how the lock strategy can be set on the `map1`, `map2`, and `map3` BackingMaps, where each map is using a different locking strategy:

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
...
ObjectGrid og =
 ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("test");
BackingMap bm = og.defineMap("map1");
bm.setLockStrategy( LockStrategy.PESSIMISTIC );
bm = og.defineMap("map2");
bm.setLockStrategy( LockStrategy.OPTIMISTIC );
bm = og.defineMap("map3");
bm.setLockStrategy( LockStrategy.NONE );
```

To avoid a `java.lang.IllegalStateException` exception, the `setLockStrategy` method must be called before calling the `initialize` or `getSession` methods on the ObjectGrid instance.

When either PESSIMISTIC or OPTIMISTIC lock strategy is used, a lock manager is created for the BackingMap. The lock manager uses a hash map to keep track of entries that are locked by one or more transactions. If many map entries exist in the hash map, more lock buckets means better performance. The risk of Java synchronization collisions is lower as the number of buckets grows. More lock buckets also lead to more concurrency. The following example shows how an application can set the number of lock buckets to use for a given BackingMap:

```
bm.setNumberOfLockBuckets( 503 );
```

Again, to avoid a `java.lang.IllegalStateException` exception, the `setNumberOfLockBuckets` method must be called before calling the `initialize` or `getSession` methods on the ObjectGrid instance. The `setNumberOfLockBuckets` method parameter is a Java primitive integer that specifies the number of lock buckets to use. Using a prime number ensures a uniform distribution of map entries over the lock buckets. A good starting point for best performance is set the number of lock buckets to about ten percent of the expected number of BackingMap entries.

## Pessimistic locking

Use the pessimistic locking strategy for read and write maps when other locking strategies are not possible.

When an ObjectGrid Map is configured to use the PESSIMISTIC locking strategy, a pessimistic transaction lock for a map entry is obtained when a transaction first gets the entry from the BackingMap. The pessimistic lock is held until the application completes the transaction. Typically, the pessimistic locking strategy is used in the following situations:

- The BackingMap is configured with or without a loader and versioning information is not available.
- The BackingMap is used directly by an application that needs help from the ObjectGrid for concurrency control.
- Versioning information is available, but update transactions frequently collide on the backing entries, resulting in optimistic update failures.

Because the pessimistic locking strategy has the greatest impact on performance and scalability, this strategy should only be used for read and write maps when other locking strategies are not viable. For example, optimistic update failures occur frequently, or recovery from optimistic failure is difficult for an application to handle.

### ObjectMap methods and lock modes

When an application uses the methods of the ObjectMap interface, ObjectGrid automatically attempts a pessimistic lock for the map entry being accessed. ObjectGrid uses the following lock modes based on which method the application calls in the ObjectMap interface:

- The `get` and `getAll` methods acquire an *S lock*, or a shared lock mode for the key of a map entry. The S lock is held until the transaction completes. An S lock mode allows concurrency between transactions that attempt to acquire an S or an upgradeable lock (U lock) mode for the same key, but blocks other transactions that attempt to get an exclusive lock (X lock) mode for the same key.
- The `getForUpdate` and `getAllForUpdate` methods acquire a *U lock*, or an upgradeable lock mode for the key of a map entry. The U lock is held until the transaction completes. A U lock mode allows concurrency between transactions

that acquire an S lock mode for the same key, but blocks other transactions that attempt to acquire a U lock or X lock mode for the same key.

- The `put`, `putAll`, `remove`, `removeAll`, `insert`, `update`, and `touch` acquire an *X lock*, or exclusive lock mode for the key of a map entry. The X lock is held until the transaction completes. An X lock mode ensures that only one transaction is inserting, updating, or removing a map entry of a given key value. An X lock blocks all other transactions that attempt to acquire a S, U, or X lock mode for the same key.
- The `global invalidate` and `global invalidateAll` methods acquire an X lock for each map entry that is invalidated. The X lock is held until the transaction completes. No locks are acquired for the `local invalidate` and `local invalidateAll` methods because none of the BackingMap entries are invalidated by local invalidate method calls.

From the preceding definitions, it is obvious that an S lock mode is weaker than a U lock mode because it allows more transactions to run concurrently when accessing the same map entry. The U lock mode is slightly stronger than the S lock mode because it blocks other transactions that are requesting either a U or X lock mode. The S lock mode only blocks other transactions that are requesting an X lock mode. This small difference is important in preventing some deadlocks from occurring. The X lock mode is the strongest lock mode because it blocks all other transactions attempting to get an S, U, or X lock mode for the same map entry. The net affect of an X lock mode is to ensure that only one transaction can insert, update, or remove a map entry and to prevent updates from being lost when more than one transaction is attempting to update the same map entry.

The following table is a lock mode compatibility matrix that summarizes the described lock modes and is used to determine which lock modes are compatible with each other. To read this matrix, the row in the matrix indicates a lock mode that is already granted. The column indicates the lock mode that is requested by another transaction. If **Yes** is displayed in the column, the lock mode requested by the other transaction is granted because it is compatible with the lock mode that is already granted. **No** indicates that the lock mode is not compatible and the other transaction must wait for the first transaction to release the lock that it owns.

*Table 8. Locking mode compatibility and strength*

| **lock** | **compatible locks** | | | **strength** |
|---|---|---|---|---|
| | S (shared) | U (upgradeable) | X (exclusive) | |
| S (Shared) | Yes | Yes | No | weakest |
| U (Upgradeable) | Yes | No | No | normal |
| X (Exclusive) | No | No | No | strongest |

## Lock wait timeout

Each ObjectGrid BackingMap has a default lock wait timeout value. The timeout value is used to ensure that an application does not wait forever for a lock mode to be granted because of a deadlock condition that occurs due to an application error. The application can use the BackingMap interface to override the default lock wait timeout value. The following example illustrates how to set the lock wait timeout value for the map1 backing map to 60 seconds:

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
```

```
...
ObjectGrid og =
 ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("test");
BackingMap bm = og.defineMap("map1");
bm.setLockStrategy( LockStrategy.PESSIMISTIC );
bm.setLockTimeout( 60 );
```

To avoid a `java.lang.IllegalStateException` exception, call both the setLockStrategy method and the setLockTimeout method before calling either the initialize or getSession methods on the ObjectGrid instance. The setLockTimeout method parameter is a Java primitive integer that specifies the number of seconds that ObjectGrid waits for a lock mode to be granted. If a transaction waits longer than the lock wait timeout value configured for the BackingMap, a `com.ibm.websphere.objectgrid.LockTimeoutException` exception results.

When a LockTimeoutException occurs, the application must determine if the timeout is occurring because the application is running slower than expected or if the timeout occurred because of a deadlock condition. If an actual deadlock condition occurred, then increasing the lock wait timeout value does not eliminate the exception. Increasing the timeout results in the exception taking longer to occur. However, if increasing the lock wait timeout value does eliminate the exception, then the problem occurred because the application was running slower than expected. The application in this case must determine why performance is slow. See Chapter 14, “Troubleshooting,” on page 333 and Chapter 11, “ObjectGrid performance best practices,” on page 315 for more information.

## Deadlocks

Consider the following sequence of lock mode requests:

```
X lock is granted to transaction 1 for key1.
X lock is granted to transaction 2 for key2.
X lock requested by transaction 1 for key2.
 (Transaction 1 blocks waiting for lock owned by transaction 2.)
X lock requested by transaction 2 for key1.
 (Transaction 2 blocks waiting for lock owned by transaction 1.)
```

The preceding sequence is the classic deadlock example of two transactions that attempt to acquire more than a single lock and each transaction acquires the locks in a different order. To prevent this deadlock, each transaction must obtain the multiple locks in the same order. If the OPTIMISTIC lock strategy is used and the flush method on the ObjectMap interface is never used by the application, then lock modes are requested by the transaction only during the commit cycle. During the commit cycle, the ObjectGrid determines the keys for the map entries that need to be locked and requests the lock modes in key sequence. With this method, ObjectGrid prevents the large majority of the classic deadlocks. However, ObjectGrid does not and cannot prevent all possible deadlock scenarios. A couple of scenarios exist that the application needs to consider. Following are the scenarios that the application must be aware of and take preventative action against.

One scenario exists where ObjectGrid is able to detect a deadlock without having to wait for a lock wait timeout to occur. If this scenario does occur, a `com.ibm.websphere.objectgrid.LockDeadlockException` exception results. Consider the following code snippet:

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
sess.begin();
Person p = (IPerson)person.get("Billy");
```

```
// Billy had a birthday, so we make him 1 year older.
p.setAge( p.getAge() + 1 );
person.put( "Billy", p );
sess.commit();
```

In this situation, Billy's wife wants to make him older than he is, and both Billy and his wife run this transaction concurrently. In this situation, both transactions own an S lock mode on the **Billy** entry of the PERSON map as a result of the person.get("Billy") method invocation. As a result of the person.put ("Billy", p) method call, both transactions attempt to upgrade the S lock mode to an X lock mode. Both transactions block waiting for the other transaction to release the S lock mode it owns. As a result, a deadlock occurs because a circular wait condition exists between the two transactions. A circular wait condition results when more than one transaction attempts to promote a lock from a weaker to a stronger mode for the same map entry. In this scenario, the ObjectGrid throws a `LockDeadlockException` exception rather than a `LockTimeoutException` exception. See "LockDeadlockException" on page 337 for more information.

The application can prevent the `LockDeadlockException` exception for the preceding example by using the OPTIMISTIC lock strategy rather than the PESSIMISTIC lock strategy. Using the OPTIMISTIC lock strategy is the preferred solution when the map is mostly read and updates to the map are infrequent. See "Optimistic locking" on page 129 for more details on the optimistic strategy. If the PESSIMISTIC lock strategy must be used, the getForUpdate method can be used instead of the get method in the above example. By doing so, the first transaction to call the getForUpdate method acquires a U lock mode rather than a S lock mode. This lock mode causes the second transaction to block when it calls the getForUpdate method because only one transaction is granted a U lock mode. Because the second transaction is blocked, it does not own any lock mode on the Billy map entry . The first transaction does not block when it attempts to upgrade the U lock mode to an X lock mode as a result of the put method call from the first transaction. This feature demonstrates why U lock mode is called the ″upgradeable″ lock mode. When the first transaction is completed, the second transaction unblocks and is granted the U lock mode. An application can prevent the lock promotion deadlock scenario by using the getForUpdate method instead of the get method when PESSIMISTIC lock strategy is being used.

**Important:** This solution does not prevent read only transactions from being able to read a map entry. Read only transactions call the get method, but never call the put, insert, update, or remove methods. Concurrency is just as high as when the regular get method is used. The only reduction in concurrency occurs when the getForUpdate method is called by more than one transaction for the same map entry.

You must take care when a transaction calls the getForUpdate method on more than one map entry to ensure that the U locks are acquired in the same order by each transaction. For example, suppose that the first transaction calls the getForUpdate method for the key 1 and the getForUpdate method for key 2. Another concurrent transaction calls the getForUpdate method for the same keys, but in reverse order. This sequence causes the classic deadlock because multiple locks are obtained in different orders by different transactions. The application still needs to ensure that every transaction accesses multiple map entries in key sequence to ensure that deadlock does not occur. Because the U lock is obtained at the time that the getForUpdate method is called rather than at commit time, the ObjectGrid cannot order the lock requests like it does during the commit cycle. The application must control the lock ordering in this case.

Using the flush method on the ObjectMap interface before a commit can introduce additional lock ordering considerations. The flush method is typically used to force changes made to the map out to the backend through the Loader plug-in. In this situation, the backend uses its own lock manager to control concurrency, so the lock wait condition and deadlock can occur in backend rather than in the ObjectGrid lock manager. Consider the following transaction:

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
boolean activeTran = false;
try
{
 sess.begin();
 activeTran = true;
 Person p = (IPerson)person.get("Billy");
 p.setAge( p.getAge() + 1 );
 person.put( "Billy", p );
 person.flush();
 ...
 p = (IPerson)person.get("Tom");
 p.setAge( p.getAge() + 1 );
 sess.commit();
 activeTran = false;
}
finally
{
 if ( activeTran ) sess.rollback();
}
```

Suppose that some other transaction also updated the Tom person, called the flush method, and then updated the Billy person. If this situation occurred, the following interleaving of the two transactions results in a database deadlock condition:

```
X lock is granted to transaction 1 for "Billy" when flush is executed.
X lock is granted to transaction 2 for "Tom" when flush is executed..
X lock requested by transaction 1 for "Tom" during commit processing.
(Transaction 1 blocks waiting for lock owned by transaction 2.)
X lock requested by transaction 2 for "Billy" during commit processing.
(Transaction 2 blocks waiting for lock owned by transaction 1.)
```

This example demonstrates that the use of the flush method can cause a deadlock to occur in the database rather than in ObjectGrid. This deadlock example can occur regardless of what lock strategy is used. The application must take care to prevent this kind of deadlock from occurring when using the flush method and when a Loader is plugged into the BackingMap. The preceding example also illustrates another reason why ObjectGrid has a lock wait timeout mechanism. A transaction that is waiting for a database lock might be waiting while it owns an ObjectGrid map entry lock. Consequently, problems at database level can cause excessive wait times for an ObjectGrid lock mode and result in a `LockTimeoutException` exception.

## Exception handling

The examples in this topic do not have any exception handling. To prevent locks from being held for excessive amounts of time when a `LockTimeoutException` exception or a `LockDeadlockException` occurs, an application needs to ensure that it catches unexpected exceptions and calls the rollback method when something unexpected occurs. Change the preceding code snippet as demonstrated in the following example:

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
boolean activeTran = false;
try
```

```
{
 sess.begin();
 activeTran = true;
 Person p = (IPerson)person.get("Billy");
 // Billy had a birthday, so we make him 1 year older.
 p.setAge( p.getAge() + 1 );
 person.put( "Billy", p );
 sess.commit();
 activeTran = false;
}
finally
{
 if ( activeTran ) sess.rollback();
}
```

The `finally` block in the snippet of code ensures that a transaction is rolled back when an unexpected exception occurs. It not only handles a `LockDeadlockException` exception, but any other unexpected exception that might occur. The `finally` block handles the case where an exception occurs during a commit method invocation. This example is not the only way to deal with unexpected exceptions, and there might be cases where an application wants to catch some of the unexpected exceptions that can occur and display one of its application exceptions. You can add catch blocks as appropriate, but the application must ensure that the snippet of code does not exit without completing the transaction.

## Optimistic locking

The optimistic locking strategy believes that no two transactions might attempt to update the same map entry while running concurrently. Because of this belief, it is not necessary to hold a lock mode for the life of the transaction because it is unlikely that more than one transaction might update the map entry concurrently.

The optimistic locking strategy is typically used when:

- A BackingMap is configured with or without a loader and versioning information is available.
- A BackingMap is mostly read. That is, transactions frequently read map entries, and only occasionally insert, update, or remove a map entry.
- A BackingMap is inserted, updated, or removed more frequently than it is read, but transactions rarely collide on the same map entry.

Like the pessimistic locking strategy, the methods on the ObjectMap interface determine how ObjectGrid automatically attempts to acquire a lock mode for the map entry being accessed. However, here are some very important differences between the pessimistic and optimistic strategies:

- Like the pessimistic locking strategy, a *S lock mode* is acquired by the `get` and `getAll` methods when the method is invoked. However, with optimistic locking, the S lock mode is not held until the transaction is completed. Instead, the S lock mode is released before the method returns to the application. The purpose of acquiring the lock mode is so that the ObjectGrid can ensure only committed data from other transactions is visible to the current transaction. After ObjectGrid has verified the data is committed, the S lock mode is released. At commit time an optimistic versioning check is performed to ensure that no other transaction changed the map entry after the current transaction released its S lock mode. If an entry is not fetched from the map before it is updated, invalidated, or deleted, the ObjectGrid runtime implicitly fetches the entry from the map. This implicit get operation is performed to get the current value at the time the entry was requested to be modified.

- Unlike pessimistic locking strategy, the `getForUpdate` and `getAllForUpdate` methods are handled exactly like the `get` and `getAll` methods when the optimistic locking strategy is used. That is, a S lock mode is acquired at the start of the method and the S lock mode is released before returning to the application.
- All other ObjectMap methods are handled exactly like they are handled for the pessimistic locking strategy. That is, when the commit method is invoked, an X lock mode is obtained for any map entry that is inserted, updated, removed, touched, or invalidated and the X lock mode is held until the transaction completes commit processing.

This locking strategy is called optimistic because an optimistic outlook exists. The optimistic locking strategy is that no two transactions might attempt to update the same map entry while running concurrently. Because of this belief, it is not necessary to hold a lock mode for the life of the transaction because it is unlikely that more than one transaction might update the map entry concurrently. However, since a lock mode was not held, another concurrent transaction could potentially update the map entry after the current transaction has released its S lock mode. To handle this possibility, ObjectGrid gets an X lock at commit time and performs an optimistic versioning check to verify that no other transaction has changed the map entry since the current transaction read the map entry from the BackingMap. If another transaction changes the map entry, the version check fails and an `OptimisticCollisionException` exception occurs. This exception forces the current transaction to be rolled back and the entire transaction must be retried by the application. The optimistic locking strategy is very useful when a map is mostly read and it is unlikely that updates for the same map entry might occur.

## None BackingMap locking strategy

When a BackingMap is configured to use a locking strategy of NONE, no transaction locks for a map entry are obtained. A scenario where this strategy is useful is when an application is a persistence manager such as a Java 2 Platform, Enterprise Edition (J2EE) Enterprise JavaBeans (EJB) container or uses Hibernate to obtain persistent data. In this scenario, the BackingMap is configured without a loader and is being used as a data cache by the persistence manager. The persistence manager in this scenario provides concurrency control between transactions that are accessing the same ObjectGrid Map entries. The ObjectGrid does not need to obtain any transaction locks for the purpose of concurrency control. This is assuming that the persistence manager does not release its transaction locks prior to updating the ObjectGrid map with committed changes. If that is not the case, then a PESSIMISTIC or OPTIMISTIC lock strategy must be used. For example, suppose the persistence manager of an EJB container is updating ObjectGrid map with data that was committed in the EJB container-managed transaction. If the update of ObjectGrid map occurs before the persistence manager transaction locks are released, then NONE lock strategy can be used. If theObjectGrid map update occurs after the persistence manager transaction locks are released, then either the OPTIMISTIC or PESSIMISTIC lock strategy is required.

Another scenario where the NONE locking strategy can be used is when the application uses a BackingMap directly and a Loader is configured for the map. In this scenario, the loader uses the concurrency control support provided by a relational database management system (RDBMS) by using either Java database connectivity (JDBC) or Hibernate to access data in a relational database. The loader implementation can use either an optimistic or pessimistic approach.

A loader that uses an optimistic locking or versioning approach helps to achieve the greatest amount of concurrency and performance. For more information about implementing an optimistic locking approach, see the "OptimisticCallback" on page 200 section in the "Loader considerations" on page 197 topic.

A loader that makes use of the pessimistic locking support of the underlying backend may want to make use of the **forUpdate** parameter that is passed on the get method on the Loader interface. This parameter is set to *true* if the `getForUpdate` method of the ObjectMap interface was used by the application to get the data. The loader can use this parameter to determine whether to request an upgradeable lock on the row being read. For example, DB2 obtains an upgradeable lock when an SQL select statement contains a `for update` clause. This approach offers the same deadlock prevention that is described in the Pessimistic locking topic.

## ObjectGrid security

Use ObjectGrid security mechanisms to secure map data access and management tasks through configuration or programming.

ObjectGrid provides security mechanisms to secure accesses to map data and management tasks. ObjectGrid security is built upon Java Authentication and Authorization Services (JAAS) mechanism. JAAS is an integral part of Java 2 Security.

This section describes ObjectGrid security mechanisms and how to use the ObjectGrid security APIs.

- "ObjectGrid security overview" gives an overview of the ObjectGrid security.
- "Client server security" on page 136 describes the client server security for the distributed ObjectGrid programming model.
- "Local ObjectGrid security" on page 155 describes the local ObjectGrid security.
- "Authorization" on page 161 describes the authorization mechanism and related plug-ins that apply to both distributed and local ObjectGrid programming model.
- "ObjectGrid cluster security" on page 170 describes ObjectGrid cluster security mechanism and related plug-ins.
- "Gateway security" on page 174 discusses the gateway security.
- "Security integration with WebSphere Application Server" on page 176 highlights the integration with WebSphere Application Server.

Most of the sample code shown in this section is from the ObjectGrid shipped samples. You can find the security sample overview in Chapter 5, "ObjectGrid samples," on page 57.

## ObjectGrid security overview

ObjectGrid is a distributed caching system. The access to the cache data can be secured. Generally, security is based on three key concepts:

- *Trustable authentication*: reliably determine the identity of the requester.
- *Authorization*: grant access rights to the requestor with permissions.
- *Secure transport*: safely transmit the data over the networks.

ObjectGrid provides security on the following aspects:

- “Client server security” addresses the authentication and client server communication security using Secure Sockets Layer (SSL).
- “Authorization” mechanisms guarantee that only authorized clients can access the ObjectGrid map data and management tasks.
- “ObjectGrid cluster security” verifies that only authorized servers can join the ObjectGrid cluster.
- “Gateway security” on page 133 addresses the gateway client authentication.
- “Local ObjectGrid security” on page 133 provides security mechanism when the application directly instantiates the ObjectGrid instance.

ObjectGrid security is built upon on open architecture and provides several plug-in points for customization. The plug-in mechanism plays an important role. ObjectGrid also provides some built-in implementation for these plug-ins. Some implementations are for out-of-box production use, and others are for testing or sample purposes. See “Security plug-ins” on page 133 for a summary of plug-ins and built-in implementations.

## Client server security

ObjectGrid supports distributed client server framework. A client server security infrastructure is in place to secure the access to ObjectGrid servers.

An ObjectGrid client can use any credential it wants to authenticate to the ObjectGrid server. A contract must be established between clients and servers so that this credential is understood by the server authentication mechanism. When Secure Sockets Layer (SSL) is used, the client can also use SSL certificates to authenticate to the ObjectGrid server.

To secure the client server communication, ObjectGrid supports SSL. The SSL protocol provides transport layer security with authenticity, integrity, and confidentiality, for a secure connection between an ObjectGrid client and server. Some of the security features that are provided by SSL are: data encryption to prevent the exposure of sensitive information while data flows, data signing to prevent unauthorized modification of data while data flows, and client and server authentication to ensure that you talk to the appropriate person or machine. SSL can be effective in securing an enterprise environment.

See “Client server security” on page 136 for more information.

## Authorization

ObjectGrid authorizations are based on subjects and permissions. In ObjectGrid, two categories of permissions exist: permissions for data access, and permissions for management tasks. You can use the Java Authentication and Authorization Services (JAAS) to authorize the access or plug in your own mechanisms to handle the authorizations.

See “Authorization” on page 161 for more information.

## ObjectGrid cluster security

In a secure environment, a server must be able to check the authenticity of another server. ObjectGrid uses a shared secret key string mechanism for this purpose. This secret key mechanism is similar to a shared password. All the ObjectGrid servers agree on a shared secret. When a server joins the cluster, it is challenged to present the secret string. If the secret string of the joining server matches the one in the master server, the joining server can join the cluster; otherwise the join request is rejected.

Sending a clear text secret is not secure. ObjectGrid security infrastructure provides a SecureTokenManager plug-in to allow the server to “secure” this secret before sending it. How you implement the “secure” operation is open. ObjectGrid provides an out-of-box implementation, in which the “secure” operation is implemented to encrypt and sign the secret.

See “ObjectGrid cluster security” on page 170 for more information

## Gateway security

An ObjectGrid gateway serves as a point to delegate the client management requests to the ObjectGrid server. The management gateway houses a set of mbeans. The gateway client invokes these mbeans to administer or monitor ObjectGrid servers.

The management gateway and server communication uses the ObjectGrid client server communication mechanism, in which the gateway is treated as an ObjectGrid client. The gateway client and gateway (MBean server) communication can be secured by SSL. This capability is provided by the JMX connector layer, which is implemented by the open source project mx4j. ObjectGrid requires mx4j in place to make gateway work.

For the authentication, the gateway propagates the credential presented by the gateway client to the ObjectGrid server. Both authentication and authorization are enforced on ObjectGrid servers.

See “Gateway security” on page 174 for more information.

## Local ObjectGrid security

In WebSphere Extended Deployment Server release 6.0, the local ObjectGrid programming model was introduced. In this model, the application directly instantiates and uses an ObjectGrid instance. Your application and ObjectGrid instances are in the same Java virtual machine (JVM). No client or server concept exists in this model.

Authentication is not supported in the local ObjectGrid programming model. Your applications must manage their own authentication, and then pass the authenticated Subject object to the ObjectGrid.

The same authorization mechanism is used for the local ObjectGrid programming model as that used for the client server model.

See “Local ObjectGrid security” on page 155 for more information.

## Security plug-ins

ObjectGrid security framework is supplemented by security plug-ins. These plug-ins can be implemented to extend or customize the security framework.

One example is the CredentialGenerator plug-in, represented by the com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator interface. When an ObjectGrid client connects to an ObjectGrid server, the getCredential() method of this plug-in is called to generate a Credential object. This Credential object is then sent to the server. The server then uses this Credential object to authenticate using the Authenticator plug-in, which is represented by thecom.ibm.websphere.objectgrid.security.plugins.CredentialGenerator.Authenticator interface.

Plug-ins play an important role in the ObjectGrid security framework. You can implement the CredentialGenerator to generate a specific credential, for example, user ID and password pair, a kerberos ticket, or a security token. You can implement the Authenticator plug-in to authenticate the client. If you want, you can implement the Authenticator plug-in to support both the user password or a security token.

All the plug-ins for security that you can use are in the following table:

*Table 9. Security plug-ins*

| Category | Plug-in class name | Instance |
|---|---|---|
| Authentication | `com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator` | client |
| | `com.ibm.websphere.objectgrid.security.plugins.Credential` | client |
| | `com.ibm.websphere.objectgrid.security.plugins.Authenticator` | server |
| Authorization | `com.ibm.websphere.objectgrid.security.plugins.MapAuthorization` | ObjectGrid |
| | `com.ibm.websphere.objectgrid.security.plugins.AdminAuthorization` | cluster |
| ObjectGrid cluster security | `com.ibm.websphere.objectgrid.security.plugins.SecureTokenManager` | server |
| other | `com.ibm.websphere.objectgrid.security.plugins.SubjectSource` | local ObjectGrid |
| | `com.ibm.websphere.objectgrid.security.plugins.SubjectValidation` | local ObjectGrid |

The following diagram shows the plug-ins and the instances applied. For example, the MapAuthorization plug-in applies on the ObjectGrid instances, but the AdminAuthorization applies on the server instances.

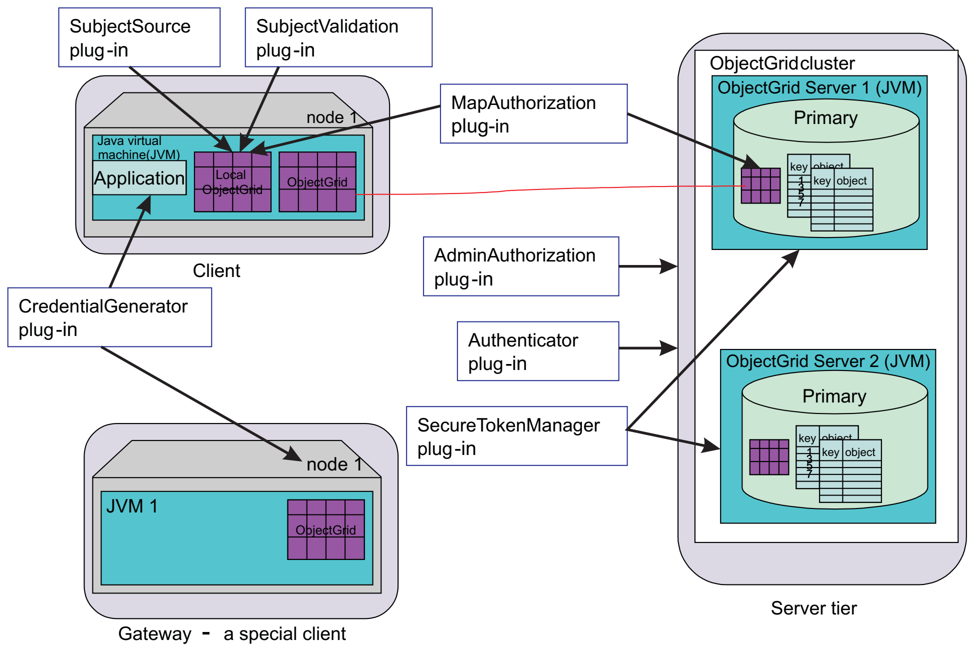


*Figure 17. Security plug-ins*

The following table displays the built-in implementation. The purpose column displays the purpose. The purpose can be for out-of-box production or for testing.

*Table 10. Security built-in implementations*

| Plug-in | built-in class name | purpose |
|---|---|---|
| Credential Generator Credential | `com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator` | production |
| | `com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredentialGenerator` | production |
| | `com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential` | production |
| | `com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredential` | production |
| | `com.ibm.websphere.objectgrid.security.plugins.builtins.ClientCertificateCredential` | production |
| Authenticator | `com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenAuthenticator` | production |
| | `com.ibm.websphere.objectgrid.security.plugins.builtins.KeyStoreLoginAuthenticator` | testing |
| | `com.ibm.websphere.objectgrid.security.plugins.builtins.CertificateMappingAuthenticator` | testing |
| | `com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPAuthenticator` | testing |
| Map Authorization | `com.ibm.websphere.objectgrid.security.plugins.builtins.JAASMapAuthorizationImpl` | production |
| | `com.ibm.websphere.objectgrid.security.plugins.builtins.TAMMapAuthorizationImpl` | testing |
| SubjectSource | `com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectSourceImpl` | production |

*Table 10. Security built-in implementations (continued)*

| Plug-in | built-in class name | purpose |
|---|---|---|
| Subject Validation | `com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectValidationImpl` | production |

## Client server security

This topic describes the authentication mechanism and how to secure the client server communication.

The client server security has the following important aspects:

- How to "Enable client server security"
- How to get a credential representing the client with the "Credential and credential generator" on page 137
- How to configure parameters used for the SSL configuration using "Secure communication" on page 153
- How to authenticate the client on the server side using an "Authenticator" on page 143

### Enable client server security

**Enable client security**

To enable the security on the client security, set the securityEnabled property in the `security.ogclient.props` file to **true**. ObjectGrid ships a client security property template file, the `security.ogclient.props` file, in the `[WAS_HOME]/optionalLibraries/ObjectGrid/properties` directory for a WebSphere installation, or the `/ObjectGrid/properties` directory in a mixed server installation. You can modify this template file with appropriate values.

The description of the securityEnabled property follows:

**securityEnabled (true, false+)**

This property indicates if security is enabled. When a client connects to a server, the securityEnabled value on the client and server side must be both true or both false. For example, if the connected server security is enabled, the client has to set this property to true to connect to the server.

The com.ibm.websphere.objectgrid.security.config.ClientSecurityConfiguration interface represents the `security.ogclient.props` file. You can use the com.ibm.websphere.objectgrid.security.config. ClientSecurityConfigurationFactory public API to create an instance of this interface with default values, or you can create an instance by passing the ObjectGrid client security property file. The `security.ogclient.props` file contains other properties.

**Enable server security**

To enable the security on the server side, you can set the securityEnabled property in the cluster XML to true. Here is an example:

```
<cluster>
 <objectGrid name="cluster" securityEnabled="true"
  singleSignOnEnabled="true" loginSessionExpirationTime="300">
```

## Credential and credential generator

When connecting a server, a client needs to present its own credential. A client credential is represented by a com.ibm.websphere.objectgrid.security.plugins.Credential interface. The credential can contain a user password pair, a Kerberos ticket, and so on.

The credential interface follows:

```
package com.ibm.websphere.objectgrid.security.plugins;

import java.io.Serializable;

/**
 * This interface represents a credential used by an ObjectGrid client. It
 * represents one client identity. This credential is sent to the
 * ObjectGrid server for authentication. It must be serializable.
 *
 * A credential has to implement the equals(Object)and
 * hashCode() methods.  Two Credential objects are considered equal
 * if and only if they represent the same identity and security information. For
 * example, if the credential contains a user ID and password. Two credentials
 * are equal if and only if both their user IDs and passwords are equal.
 *
 * ObjectGrid provides three built-in implementations for this interface:
 * com.ibm.websphere.objectgrid.security.plugins.builtins.
 * ClientCertificateCredential:
 * A credential containing an SSL certificate chain.
 * com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential:
 * A credential containing a user ID and password pair.
 * com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredential:
 * A credential containing WebSphere Application Server specific authentication
 * and authorization tokens.
 *
 * Refer to the respective API documentation for more details.
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see CredentialGenerator
 */
public interface Credential extends Serializable {

    /**
     * Checks two Credential objects for equality.
     *
     * Two Credential objects are considered equal if and only if
     * they represent the same identity and security information.
     *
     * @param o the object we are testing for equality with this object.
     *
     * @return true if both Credential objects are equivalent.
     */
    boolean equals(Object o);

    /**
     * Returns the hashcode of the Credential object
     *
     * @return the hash code of the Credential object
     */
    int hashCode();
}
```

This interface explicitly defines the equals(Object) and hashCode() methods. These methods are important to guarantee the behavior. The authenticated Subject objects are cached based on the Credential objects on the server side.

ObjectGrid provides three default implementations for the Credential interfaces:

1. The com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential implemenation. This credential contains a user ID and password pair.
2. The com.ibm.websphere.objectgrid.security.plugins.builtins.ClientCertificateCredential implementation. This credential contains a client certificate chain. This credential can be used for ObjectGrid client certificate authentication. You cannot create this credential on the client side. It has to be generated by the server as part of the SSL handshake.
3. The com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredential implementation. This credential contains WebSphere Application Server-specific authentication and authorization tokens. These tokens can be used to propagate the security attributes across the application servers in the same security domain.

Refer to the API documentation for more details.

ObjectGrid also provides a plug-in to generate a credential. This plug-in is represented by the com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator interface. Following are the CredentialGenerator interfaces:

```
/**
  * This plug-in is used to get a Credential representing this client. It is a
 * factory for the Credential object.
 * One example implementation is to return a Credential object
 * containing a user ID and password pair. The implementation of the Credential
 * generated by an implementation of this class must to be understood by the
 * server's Authenticator plug-in.
 *
 * An implementation class of this interface must have a default constructor.
 * When launching the client in a secure environment, set the
 * implementation class name (credentialGeneratorClass) in the client security
 * configuration property file. The client runtime constructs an object of
 * this implementation class and calls getCredential() to get the Credential
 * to connect to an ObjectGrid cluster.
 *
 *Users can also specify the additional properties for this factory using the
 * credentialGeneratorProps property in the client security configuration
 * property file.
 * These properties are be passed to this
 * factory by using the setProperties(String) method. This way, you can
 * customize your factory.
 *
 * You can also set CredentialGenerator programmatically by calling
 * ClientSecurityCinfiguration.setCredentialGenerator
 * (CredentialGenerator) method.
 *
 *
 * For example, you can have the following settings in the client security
 * configuration property file:
  * credentialGeneratorClass=com.myco.CredGenFactory
  *
  * credentialGeneratorProps=user1 password1
  *
  *
  * , a String "user1 password1" is passed to the setProperties(String)
  * method, with the "user1" indicating the user name, and "password1"
 * indicating the password.
  *
  * ObjectGrid provides two built-in implementations for this interface:
   * com.ibm.websphere.objectgrid.security.plugins.builtins.
 *   UserPasswordCredentialGenerator:
   * A credential generator generating a UserPasswordCredential
 * containing a user ID and password pair.
 * com.ibm.websphere.objectgrid.security.plugins.builtins.
 *  WSTokenCredentialGenerator:
   * A credential generator generating a WSTokenCredential containing WebSphere
 * Application Server
  * specific authentication and authorization tokens.
  *
  *
 * The relationship between CredentialGenerator and Credential can be
 * one to one relationship or one to many relationship. For example, the
 * UserPasswordCredentialGenerator
 * has a one to one relationship with UserPasswordCredential, but the
 * WSTokenCredentialGenerator
 * has a one to many relationship with WSTokenCredential because it could generate
 * different WSTokenCredential based on what Subject is associated with the current
 * thread.
 *
 * Refer to the respective API documentation for more details.
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see Authenticator
 * @see ClientSecurityConfiguration#setCredentialGenerator(CredentialGenerator)
 * @see Credential
 * @see CredentialGeneratorFactory#getCredentialGenerator()
 */
public interface CredentialGenerator {

    /**
     * Gets a Credential which represents the client.
     *
     * @return the Credential representing the client
     *
     * @throws CannotGenerateCredentialException if a failure occurs when
     *         generating the Credential for the client.
     *
     * @see Credential
     */
    Credential getCredential() throws CannotGenerateCredentialException;

    /**
     * Set the user defined properties to the factory
*
* This method is used to add addtional CredentialGenerator properties
* to the object. These properties can be set using the credentialGeneratorProps
* property in the client security configuration property file.
* This way, you can customize your factory.
*
* @param properties user defined properties
*/
void setProperties(String properties);
}
```

ObjectGrid provides two default built-in implementations:

1. The com.ibm.websphere.objectgrid.security.plugins.builtins. UserPasswordCredentialGenerator constructor takes a user ID and a password. When the getCredential() method is called, it returns a UserPasswordCredential object that contains the user ID and password.

2. The com.ibm.websphere.objectgrid.security.plugins.builtins. WSTokenCredentialGenerator represents a credential (security token) generator when running in WebSphere Application Server. When the getCredential() method is called, the Subject associated with the current thread is retrieved. Then the security information in this Subject object is converted into a WSTokenCredential object. You can specify whether to retrieve a runAs subject or a caller subject from the thread by using the constant WSTokenCredentialGenerator.RUN_AS_SUBJECT or WSTokenCredentialGenerator.CALLER_SUBJECT.

Refer to the API documentation for more details.

## Connect

If an ObjectGrid client wants to connect to a server securely, you can use the any connect method in the ObjectGridManager interface. Take the following connect method as an example:

```
/**
* This allows client to connect to a Remote ObjectGrid
* The RemoteObject Grid is hosted as specified by the paramaters
* @param clusterName: The name of the cluster to which this client
* will attach iteself
* @param host: The host on which to connect to
* @param port: The clientAceess port which is listening.
* @param ClientSecurityConfiguration: Security configuration. It can be null
* if security is not configured
* @param overRideObjectGrid xml. This parameter can be null. If it is not
* null, the client side configuration of objectgrid plug-in is overridden.
* Not all plug-ins can be overridden. For details see the ObjectGrid documents
* @throws ConnectException
* @ibm-api
*/
public ClientClusterContext connect(String clusterName, String host, String port,
ClientSecurityConfiguration securityProps, URL overRideObjectGrid) throws
ConnectException ;
```

This method takes a parameter of type ClientSecurityConfiguration among others. This interface represents a client security configuration. You can use the com.ibm.websphere.objectgrid.security.config.ClientSecurityConfigurationFactory public API to create an instance of this with default values, or you can create an instance by passing the ObjectGrid client security property file. The `security.ogclient.props` file contains the following properties related to authentication. The value marked with + is the default value.

- `securityEnabled (true, false+)`: This property indicates if the security is enabled. When a client connects to a server, the securityEnabled value on the client and server side must be both true or both false. For example, if the connected server security is enabled, the client has to set this property to true to connect to the server.
- `credentialAuthentication (Never, Supported+, Required)`: This property indicates if the client supports credential authentication.
  – If the property value is **Never**, no credential authentication is supported by this client.
  – If the property value is **Supported**, client authentication is performed when communicating with any server that supports or requires credential authentication. Client credential authentication transmits a credential or a single sign-on (SSO) token
  – If the property value is **Required**, the client must send a credential to the server for authentication.

- `authenticationRetryCount (an integer value, 0+)`. This property determines how many retries are attempted for login when a credential is expired. If the value is `0`, no retries are attempted. The authentication retry only applies to the case when the credential is expired. If the credential is not valid, there is not any retry. Your application is responsible for retrying the operation.
- `clientCertificateAuthentication (Never+, Supported, Required)`: This property indicates if the client supports client certificate authentication.
  - If the property value is **Never**, no client certification authentication is supported on the client side.
  - If the property value is **Supported**, transport layer client authentication can be performed and the client sends digital certificate to the server during the authentication stage.
  - If the property value is **Required**, the client only authenticates with servers that support transport-layer client authentication.
- `transportType (TCP/IP, SSL-Supported+, SSL-Required)`: This indicates which transport protocol the client wants to connect to the server. Which protocol a client connects to a server also depends on the transportType setting on the server side. See "Secure communication" on page 153 for more details. .
  - If the value is **TCP/IP**, the client has to use TCP/IP to connect to the server.
  - If the value is **SSL-Supported**, the client may use TCP/IP or SSL to connect to the server. The client first tries to use SSL to connect to the server. If the SSL connect fails, the client tries to use TCP/IP.
  - If the value is **SSL-Required**, the client must use SSL to connect to the server.
- `SSOEnabled`: Specifies if the client supports passing single sign-on tokens to the server. Set this property to `false` if the client authenticates to every server. Set this property to `true` if the client only authenticates to one server. If you set SSOEnabled true on the client, verify that the single-sign-on-enabled property in the cluster XML configuration is also set to `true`.

You can also set these properties using setters in the ClientSecurityConfiguration interface.

After you create a ClientSecurityConfiguration type object, set the credentialGenerator on the object using the following method:

```
/**
* Set the {@link CredentialGenerator} object for this client.
* @param generator the CredentialGenerator object associated with this client
*/
void setCredentialGenerator(CredentialGenerator generator);
```

You can set the CredentialGenerator in the ObjectGrid client security property file too. Here are the properties:

- **credentialGeneratorClass**: the class implementation name for the CredentialGenerator. It must have a default constructor.
- **credentialGeneratorProps**: the properties for the CredentialGenerator class. If the value is not null, it is set to the constructed CredentialGenerator object using the setProperties(String) method.

Here is a sample to instantiate a ClientSecurityConfiguration and then use it to connect to the server.

```
/**
* Get a secure ClientClusterContext
* @return a secure ClientClusterContext object
*/
```

```
protected ClientClusterContext connect() throws ConnectException {
  ClientSecurityConfiguration csConfig = ClientSecurityConfigurationFactory
  .getClientSecurityConfiguration("/properties/security.ogclient.props");

  UserPasswordCredentialGenerator gen= new
  UserPasswordCredentialGenerator("manager", "manager1");

  csConfig.setCredentialGenerator(gen);

  return objectGridManager.connect(csConfig, null);
}
```

When the connect is called, the ObjectGrid client calls the CredentialGenerator.getCredential() method to get the client credential. This credential is sent along with the connect request to the server for authentication.

## Use a different CredentialGenerator per session

In some cases, an ObjectGrid client represents just one client identity; in other cases, it might represent multiple identities. Here is one scenario for the latter case: An ObjectGrid client is created and shared in a Web server. All servlets in this web server use this one ObjectGrid client. Because every servlet represents a different web client, use different credentials when sending requests to ObjectGrid servers.

ObjectGrid provides changing the credential on the session level. That is, every session can uses a different CredentialGenerator. Therefore, the previous scenarios can be done by letting the servlet get a session with a different CredentialGenerator. Following is the method in the ObjectGridManager interface.

```
/**
* Get a session with a CredentialGenerator. This method can only be called
* by the ObjectGrid client in a client server environment.
*
* If ObjectGrid is used in a core model, that is, within the same JVM with
* no client or server existing, getSession(Subject) should be used to secure
* the ObjectGrid.
*
* @since WAS XD 6.0.1
*/
Session getSession(CredentialGenerator credGen) throws
 ObjectGridException, TransactionCallbackException;
```

Here is an example:

```
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();

CredentialGenerator credGenManager = new UserPasswordCredentialGenerator
("manager", "xxxxxx");
CredentialGenerator credGenEmployee = new UserPasswordCredentialGenerator
("employee", "xxxxxx");

ObjectGrid og = ogManager.getObjectGrid(ctx, "accounting");

// Get a session with CredentialGenerator;
Session session = og.getSession(credGenManager );

// Get the employee map
ObjectMap om = session.getMap("employee");

// start a transaction.
session.begin();

Object rec1 = map.get("xxxxxx");

session.commit();
```

```
// Get another  session with a different CredentialGenerator;
session = og.getSession(credGenEmployee );

// Get the employee map
om = session.getMap("employee");

// start a transaction.
session.begin();

Object rec2 = map.get("xxxxx");

session.commit();
```

If you use ObjectGird.getSession() method to get a Session object, the session uses the CredentialGenerator set on the ClientConfigurationSecurity object. Therefore, you can treat the CredentialGenerator passed to the ObjectGird.getSession(CredentialGenerator) method overrides the CredentialGenerator set in the ClientConfigurationSecurity object.

If you can reuse the Session object, a performance gain results. However, calling the ObjectGrid.getSession(CredentialGenerator) method is not very expensive; the major overhead is the increased object garbage collection time. Make sure you release the references after you are done with the Session objects. In summary, if your Session object can share the identity, try to reuse the Session object; if your Session object can not share the identity, use the ObjectGrid.getSession(CredentialGenerator) method.

## Authenticator

After the ObjectGrid client retrieves the Credential object using the CredentialGenerator object, the Credential object is sent along with the client request to the ObjectGrid server. The ObjectGrid server authenticates the Credential object before processing the request. If the Credential object is authenticated successfully, a Subject object is returned to represent this Credential object. This Subject object is then used for authorizing the request.

This Subject object is also cached. It expires after its lifetime reaches the session time out value. The login session timeout value can be set using the loginSessionExpirationTime property in the cluster XML file. For example, setting loginSessionExpirationTime=″300″ makes the Subject object expire in 300 seconds.

ObjectGrid server uses the Authenticator plug-in to authenticate the Credential object. Following is the Authenticator interface:

```
/**
 * This plug-in can be used to authenticate an ObjectGrid client to an ObjectGrid
 * server based on the credential provided by the client. A Subject
 * object is returned as a result of authentication.
 *
 * This plug-in is used in an ObjectGrid server. It can be configured in the
 * ObjectGrid cluster XML file.
 *
 * The Credential passed in the authenticate(Credential)
 * method can contain any credential information users desire. For example, it could be a
 * Credential object containing a user password pair.
 *
 * ObjectGrid provides several built-in implementations for this interface:
 *  * com.ibm.websphere.objectgrid.security.plugins.builtins.
 * CertificateMappingAuthenticator:
 * An authenticator that simply maps a SSL certficate to a Subject.
 * com.ibm.websphere.objectgrid.security.plugins.builtins.KeyStoreLoginAuthenticator:
```

```
 * An authenticator that authenticates a user ID and password to a key file.
 * com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPAuthenticator:
 * An authenticator that authenticates a user ID and password to a LDAP server.
 * com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenAuthenticator:
 * An authenticator that authenticates a WebSphere Application Server securty token.
 *
 * Refer to the respective API documentation for more details.
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see Credential
 */
public interface Authenticator {

    /**
     * Authenticates a user represented by the credential object.
     *
     * @param credential the user Credential
     *
     * @return a Subject object representing the user
     *
     * @throws InvalidCredentialException if the credential is invalid
     * @throws ExpiredCredentialException if the credential is expired
     *
     * @see Credential
     */
    Subject authenticate(Credential credential)
        throws InvalidCredentialException, ExpiredCredentialException;
}
```

This is where the implementation gets the Credential object and then authenticate it to a user registry, for example, a Lightweight Directory Access Protocol (LDAP) server, and so on. ObjectGrid does not provide an out-of-box user registry configuration. Connecting to a user registry and authenticating to it must be implemented in this plug-in.

For example, one Authenticator implementation extracts the user ID and password from the credential, uses them to connect and validate to an LDAP server, and creates a Subject object as a result of the authentication. The implementation could utilize JAAS login modules. A Subject object is returned as a result of authentication.

Notice that this method throws two exceptions: `InvalidCredentialException` and `ExpiredCredentialException`. The `InvalidCredentialException` exception indicates the credential is not valid. The `ExpiredCredentialException` exception indicates the credential is expired. If one of these two exceptions result from the authenticate method, the exceptions are sent back to the client. However, the client runtime deals with these two exceptions differently:

- If the exception is an InvalidCredentialException, the client runtime displays this exception. Your application is expected to handle the exception. You can correct the CredentialGenerator, for example, and then retry the operation.
- If the exception is an ExpiredCredentialException, and the retry count is not 0, the client runtime calls the CredentialGenerator.getCredential() method again, and sends the new Credential object to the server. If the new credential authentication succeeds, the server processes the request. If the new credential authentication fails, the exception is sent back to the client. If the number of authentication retry reaches the allowed value and the client still gets an ExpiredCredentialException , the ExpiredCredentialException results. Your application must handle the exception.

The Authenticator interface provides a great flexibility. You can implement the Authenticator interface in any way. For example, you can implement this interface to do both credential authentication and client certificate authentication, to support both authentications. Or you can implement the interface to support two different user registries.

ObjectGrid supports two kinds of authentications: credential authentication and client certificate authentication. Which mechanism to use depends on the client and server side security property setting. These properties follow:

- credentialAuthentication on the `security.ogclient.props` file
- credentialAuthentication on the `security.ogserver.props` file
- clientCertificateAuthentication on the `security.ogclient.props` file
- clientCertificateAuthentication on the `security.ogserver.props` file

Remember that you can also setting these properties using programming APIs.

The following two tables display which authentication mechanism is used under different settings.

*Table 11. Credential authentication under client and server settings*

| Client credentialAuthentication | Server credentialAuthentication | Result |
|---|---|---|
| No | Never | disabled |
| | Supported | disabled |
| | Required | Error case |
| Supported | Never | disabled |
| | Supported | enabled |
| | Required | enabled |
| Required | Never | Error case |
| | Supported | enabled |
| | Required | enabled |

When no credential authentication exists (the result is disabled), the client certificate authentication could happen.

The following table shows whether client certificate authentication are used under different settings. Notice client certificate authentication is only possible if SSL is used as the communication protocol, and the credential authentication is not used.

*Table 12. Client certificate authentication under client and server settings.*

| Client clientCertificate Authentication | Server clientCertificate Authentication | Result |
|---|---|---|
| No | Never | disabled |
| | Supported | disabled |
| | Required | Error case |
| Supported | Never | disabled |
| | Supported | enabled* |
| | Required | enabled* |

*Table 12. Client certificate authentication under client and server settings. (continued)*

| Client clientCertificate Authentication | Server clientCertificate Authentication | Result |
|---|---|---|
| Required | Never | Error case |
| | Supported | enabled* |
| | Required | enabled* |

* ClientCertificateAuthentication only happens when SSL is used as the protocol and CredentialAuthentication is not used.

Notice that subtlety exists: When both credential authentication and client certificate authentication are used, but the credential sent from the client is null, the client certificate authentication is used.

The authenticator can be configured in the cluster XML file. An example follows:

```
<cluster name="cluster1" securityEnabled="true" singleSignOnEnabled="true"
 loginSessionExpirationTime="300" statisticsEnabled="true"
 statisticsSpec="map.all=enabled">
 <serverDefinition name="server1" host="localhost" clientAccessPort="12503"
  peerAccessPort="12500" workingDirectory="" traceSpec="ObjectGrid=all=disabled"
  systemStreamToFileEnabled="true" />
 <serverDefinition name="server2" host="localhost" clientAccessPort="12504"
  peerAccessPort="12501" workingDirectory=""
  traceSpec="ObjectGrid=all=disabled"
  systemStreamToFileEnabled="true" />
 <authenticator
 className ="com.ibm.websphere.objectgrid.security.plugins.builtins.
  WSTokenAuthenticator">
 </authenticator>
</cluster>
```

ObjectGrid provides four default authentication built-in implementations for the following: user ID and password authentication to a key file user registry, user ID and password authentication to an LDAP server, SSL client certificate simple mapping authentication, and WebSphere Application Server Security mechanism. Except the Authenticator implementation for the WebSphere Application Server security mechanism, the built-in implementations are for testing purposes only. The main purpose of these two built-ins is to allow you to do simple testing without writing any code. WebSphere Application Server Authenticator implementation is an out-of-box implementation that can be plugged in when both ObjectGrid servers and clients are in the same security domain.

For ObjectGrid servers that want to use WebSphere Application Server user registries, you can use WebSphere Application Server APIs to get the user registry configured in the application server, and then use that in your Authenticator implementation. However, this implementation is out of scope of this programming guide.

**Key file registry authenticator implementation**

You can store user ID and password in a file called a key store file. You can use the keytool tool to create a keystore file and entries. For example, the following command creates an entry with alias user1:

```
keytool -genkey -v -keystore ./keys.jks -storepass password -alias user1
-keypass password -dname CN=user1,O=MyCompany,L=MyCity,ST=MyState
```

For testing purposes, ObjectGrid provides the com.ibm.websphere.objectgrid.security.plugins.builtins. KeyStoreLoginAuthenticator default implementation for this plug-in to handle the user name and password authentication. This implementation uses login name KeyStoreLogin to log the user into a key store file.

Following is a code snippet shows the implementation of the authenticate(Credential) method in the KeyStoreLoginAuthenticator class.

```
public Subject authenticate(Credential credential)throws
 InvalidCredentialException,
ExpiredCredentialException {
  UserPasswordCredential cred = (UserPasswordCredential) credential;
  LoginContext lc = null;

  lc = new LoginContext("KeyStoreLogin",
  new UserPasswordCallbackHandlerImpl(cred.getUserName(),
  cred.getPassword().toCharArray()));

  lc.login();

  Subject subject = lc.getSubject();
```

This snippet first casts the Credential to a UserPasswordCredential, which is an implementation of the Credential interface, because it has a contract with the client that the client can only pass a UserPasswordCredential type object. It then calls the KeyStoreLogin login module to log in.

ObjectGrid ships a com.ibm.websphere.objectgrid.security.plugins.builtins.KeystoreLoginModule login module for this purpose. You need to provide a key store file that contains the user name and password pair for each user. The key store file is configured as an option to the login module.

Following is the code snippet that shows how the login model log into the key file.

```
/**
* Authenticates a user based on the keystore file.
*
* @see javax.security.auth.spi.LoginModule#login()
*/
public boolean login() throws LoginException {

  if (debug) {
    System.out.println("[KeyStoreLoginModule] login: entry");
  }

  String name = null;
  char pwd[] = null;

  if (keyStore == null || subject == null || handler == null) {
    throw new LoginException("Module initialization failed");
  }

  NameCallback nameCallback = new NameCallback("Username:");
  PasswordCallback pwdCallback = new PasswordCallback("Password:", false);

  try {
    handler.handle(new Callback[] { nameCallback, pwdCallback });
  }
  catch (Exception e) {
    throw new LoginException("Callback failed: " + e);
  }
  name = nameCallback.getName();
  char[] tempPwd = pwdCallback.getPassword();

  if (tempPwd == null) {
    // treat a NULL password as an empty password
    tempPwd = new char[0];
  }
  pwd = new char[tempPwd.length];
  System.arraycopy(tempPwd, 0, pwd, 0, tempPwd.length);

  pwdCallback.clearPassword();

  if (debug) {
    System.out.println("[KeyStoreLoginModule] login: "
    + "user entered user name: " + name);
  }
  if (ObjectGridManagerImpl.isTraceEnabled && TC.isDebugEnabled())
  Tr.debug(TC, "login", "userName="+name);

  // Validate the user name and password
  try {
    validate(name, pwd);
  }
  catch (SecurityException se) {

    principals.clear();
    publicCreds.clear();
    privateCreds.clear();
    LoginException le = new LoginException(
    "Exception encountered during login");
    le.initCause(se);

    throw le;
  }

  if (debug) {
    System.out.println("[KeyStoreLoginModule] login: exit");
  }
  return true;
}

/**
* Validate the user name and password based on the keystore.
*
* @param userName user name
* @param password password
* @throws SecurityException if any exceptions encountered
*/
protected void validate(String userName, char password[])
throws SecurityException {

  PrivateKey privateKey = null;

  // Get the private key from the keystore
  try {
    privateKey = (PrivateKey) keyStore.getKey(userName, password);
  }
  catch (NoSuchAlgorithmException nsae) {

    SecurityException se = new SecurityException();
    se.initCause(nsae);
    throw se;
  }
  catch (KeyStoreException kse) {
    SecurityException se = new SecurityException();
    se.initCause(kse);
```

```
      throw se;
    }
    catch (UnrecoverableKeyException uke) {
      SecurityException se = new SecurityException();
      se.initCause(uke);
      throw se;
    }

    if (privateKey == null) {
      throw new SecurityException("Invalid name: " + userName);
    }

    // Check the certificats
    Certificate certs[] = null;
    try {
      certs = keyStore.getCertificateChain(userName);
    }
    catch (KeyStoreException kse) {
      SecurityException se = new SecurityException();
      se.initCause(kse);
      throw se;
    }


    if (certs != null && certs.length > 0) {

      // If the first certificate is an X509Certificate
      if (certs[0] instanceof X509Certificate) {
        try {
          // Get the first certificate which represents the user
          X509Certificate certX509 = (X509Certificate) certs[0];

          // Create a principal
          X500Principal principal = new X500Principal(certX509
          .getIssuerDN()
          .getName());
          principals.add(principal);

          if (debug) {
            System.out.println("  Principal added: " + principal);
          }

        }
        catch (CertificateException ce) {
          SecurityException se = new SecurityException();
          se.initCause(ce);
          throw se;
        }
      }
    }
  }
```

You must create a login name “KeyStoreLogin” in the JAAS authentication configuration file. If you are not familiar with the JAAS authentication configuration file, see the JAAS Authentication Tutorial for more details.

```
KeyStoreLogin {
  com.ibm.websphere.objectgrid.jaas.KeystoreLoginModule required
  keyStoreFile="${user.dir}${/}security${/}.keystore";
};
```

This implementation is for testing purposes only.

### LDAP authenticator implementation

ObjectGrid provides the com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPAuthenticator default implementation for this plug-in to handle the user name and password authentication to an LDAP server. This implementation uses the LDAPLogin login module to log the user into an LDAP server.

The following snippet demonstrates how the authenticate method is implemented:

```
/**
* @see com.ibm.ws.objectgrid.security.plugins.Authenticator#
*   authenticate(LDAPLogin)
*/
public Subject authenticate(Credential credential) throws
InvalidCredentialException, ExpiredCredentialException {

  UserPasswordCredential cred = (UserPasswordCredential) credential;
  LoginContext lc = null;
  try {
    lc = new LoginContext("LDAPLogin",
    new UserPasswordCallbackHandlerImpl(cred.getUserName(),
    cred.getPassword().toCharArray()));

    lc.login();

    Subject subject = lc.getSubject();

    return subject;
  }
  catch (LoginException le) {
    throw new InvalidCredentialException(le);
  }
  catch (IllegalArgumentException ile) {
    throw new InvalidCredentialException(ile);
  }
}
```

ObjectGrid ships a login module com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPLoginModule for this purpose. You must provide the following two options in the JAAS login configuration file:

- **providerURL**: The LDAP server provider URL
- **factoryClass**: The LDAP context factory implementation class

The LDAPLoginModule calls the com.ibm.websphere.objectgrid.security.plugins.builtins. LDAPAuthentcationHelper.authenticate method. The following code snippet shows how the authenticate method of LDAPAuthentcationHelper is implemented:

```
/**
* Authenticate the user to the LDAP directory.
* @param user the user ID, e.g., uid=xxxxxx,c=us,ou=bluepages,o=ibm.com
* @param pwd the password
*
* @throws NamingException
*/
public String[] authenticate(String user, String pwd)
  throws NamingException {
  Hashtable env = new Hashtable();
  env.put(Context.INITIAL_CONTEXT_FACTORY, factoryClass);
  env.put(Context.PROVIDER_URL, providerURL);
  env.put(Context.SECURITY_PRINCIPAL, user);
  env.put(Context.SECURITY_CREDENTIALS, pwd);
  env.put(Context.SECURITY_AUTHENTICATION, "simple");

  InitialContext initialContext = new InitialContext(env);
```

```
    // Look up for the user
    DirContext dirCtx = (DirContext) initialContext.lookup(user);

    String uid = null;
    int iComma = user.indexOf(",");
    int iEqual = user.indexOf("=");
    if (iComma > 0 && iComma > 0) {
      uid = user.substring(iEqual + 1, iComma);
    }
    else {
      uid = user;
    }

    Attributes attributes = dirCtx.getAttributes("");

    // Check the UID
    String thisUID = (String) (attributes.get(UID).get());

    String thisDept = (String) (attributes.get(HR_DEPT).get());

    if (thisUID.equals(uid)) {
      return new String[] { thisUID, thisDept };
    }
    else {
      return null;
    }
  }
```

If authentication succeeds, the ID and password are considered valid. Then the login module gets the UID info and department info from this authenticate method. The login module creates two principals: SimpleUserPrincipal and SimpleDeptPrincipal. You can use the authenticated subject for group authorization (in this case, the department is a group) and individual authorization.

Following is a login module configuration example that is used to log in to the LDAP server:

```
LDAPLogin { com.ibm.websphere.objectgrid.security.plugins.builtins.
 LDAPLoginModule required
providerURL="ldap://directory.acme.com:389/"
factoryClass="com.sun.jndi.ldap.LdapCtxFactory";
};
```

In the previous configuration, the LDAP server points to the `ldap://directory.acme.com:389/` server. Change this setting to your LDAP server. This login module uses the provided user ID and password to connect to the LDAP server. This implementation is for testing purposes only.

**WebSphere Application Server authenticator implementation**

ObjectGrid also provides the com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenAuthenticator built-in implementation to use the WebSphere Application Server security infrastructure. This built-in implementation can be used when the following conditions exist:

- The WebSphere Application Server global security is turned on.
- Both ObjectGrid clients and ObjectGrid servers are launched in the WebSphere application server Java virtual machines.
- These application servers are in the same security domain.

- The ObjectGrid client is already authenticated in the WebSphere Application Server.

The ObjectGrid client can use the com.ibm.websphere.objectgrid.security.plugins.builtins. WSTokenCredentialGenerator class to generate a credential and the ObjectGrid server uses this Authenticator implementation class to authenticate the credential. If the token is authenticated successfully, a Subject object returns.

This scenario takes advantage of the fact that the ObjectGrid client has already been authenticated. Because the application servers that have the ObjectGrid servers are in the same security domain as the application servers housing the ObjectGrid clients, the security tokens can be propagated from the ObjectGrid client to the ObjectGrid server so the same user registry does not need to be re-authenticated.

**Simple certificate mapping authenticator implementation**

ObjectGrid also provides a built-in implementation com.ibm.websphere.objectgrid.security.plugins.builtins. CertificateMappingAuthenticator to map the certificate to an Subject object. The implementation extracts the Distinguished Name (DN) of the first certificate in the chain and creates a principal with that name. This implementation is for testing purposes only.

**Tivoli Access Manager Authenticator Implementation**

Tivoli Access Manager has been widely as a security server. You can also implement Authenticator using the Tivoli Access Manager-provided login modules.

To authenticate a user using Tivoli Access Manager, the Tivoli provided LoginModule, com.tivoli.mts.PDLoginModule, requires that the calling application provide the following:

- A principal name, specified as either a short name or an X.500 name (DN)
- A password

The LoginModule authenticates the principal and returns the Tivoli Access Manager credential. The LoginModule expects the calling application to provide the following information:

- The user name, through a javax.security.auth.callback.NameCallback
- The password, through a javax.security.auth.callback.PasswordCallback.

When the Tivoli Access Manager credential is successfully retrieved, the JAAS LoginModule creates a Subject and a PDPrincipal. No built-in for TAM authentication is provided, because it is just trivial with the PDLoginModule. Refer to the IBM Tivoli Access Manager Authorization Java Classes Developer Reference for more details.

## Single sign-on

After an ObjectGrid client successfully authenticates to a server, the ObjectGrid server creates a Subject object. If both the client and server support single sign-on (SSO), this Subject object is then converted to an SSO token. This token is passed back to the client side to associate with the socket. This SSO token can be passed to a new server for authentication so there is no need to re-authenticate on a different server.

SSO token is implemented using ObjectGrid secure token manager mechanism. For more details about secure token manager, see “ObjectGrid cluster security” on page 170. Basically, the secure token mechanism uses cryptographic keys (secret keys) to encrypt and decrypt user data that passes between the servers, and public-private keys to sign the data.

The SSO token also contains an expiration time. All product servers participating in a protection domain must have their time, date, and time zone synchronized. If not, the SSO tokens appear prematurely expired and cause authentication or validation failures. (This is not necessary if universal time is used).

When an ObjectGrid client connects to a different server, this SSO token can be passed to the new server. This server will validate the SSO token to make sure it hasn’t been tampered with by unsigning and decrypting it. It also checks its timestamp to make sure it has not been expired. If the token is valid, the client does not need to authenticate to this server.

If an SSO token is expired, the server has to re-authenticate the client. The server asks the client provide the credential again.

**Enable single-sign-on for client**

Enabling client single sign on can be done in two ways:

- **Configuration**. Use the SSOEnabled property in the `security.ogclient.props` file to enable single sign-on on the client side.
- **Programming**. Use the ClientSecurityConfiguration to enable single sign-on with the following method.

```
/**
* Set whether single sign on is enabled.
* @param enabled whether single sign on is enabled for this
* client or not.
*/
void setSingleSignOnEnabled(boolean enabled);
```

**Enable single-sign-on for server**

To enable single sign-on on the server side, set the singleSignOnEnabled attribute to true in the cluster XML file. An example follows:

```
<cluster>
 <objectGrid name="cluster" securityEnabled="true"
  singleSignOnEnabled="true" loginSessionExpirationTime="300">
```

Notice that single sign-on is only enabled if security is enabled.

## Secure communication

ObjectGrid supports both TCP/IP and SSL for secure communication. SSL provides a secure communication between client and server. Which communication mechanism is used depends on the settings of the following properties:

- The transportType property in the security.ogclient.props file
- The transportType property in the security.ogserver.props file

*Table 13. Transport protocol to use under client transport and server transport settings*

| Client transportType | Server transportType | Resulted protocol |
|---|---|---|
| TCP/IP | TCP/IP | TCP/IP |
| | SSL supported | TCP/IP |
| | SSL required | Error |

*Table 13. Transport protocol to use under client transport and server transport settings (continued)*

| Client transportType | Server transportType | Resulted protocol |
|---|---|---|
| SSL supported | TCP/IP | TCP/IP |
| | SSL supported | SSL (if SSL fails, then TCP/IP) |
| | SSL required | SSL |
| SSL required | TCP/IP | Error |
| | SSL supported | SSL |
| | SSL required | SSL |

When SSL is used, the SSL configuration must be provided on both the client and server side.

**Configure SSL parameters for ObjectGrid clients**

SSL parameters on the client side can be configured in the following ways:

- Create a com.ibm.websphere.objectgrid.security.config.SSLConfiguration object by using the factory class com.ibm.websphere.objectgrid.security.config. ClientSecurityConfigurationFactory. For more details, refer to the API documentation.
- Configure the parameters in the `security.ogclient.props` file, and then use ClientSecurityConfigurationFactory.getClientSecurityConfiguration(String) method to populate the object instance.

The following properties are for SSL configurations in the `security.ogclient.props` file.

- **provider**: Specifies the SSL JSSE provider. Possible values are IBMJSSE+, IBMJSSE2, SunJSSE, and so on. Set this value based on the Java Development Kit (JDK) that you use.
- **protocol**: Specifies the SSL protocol. Possible values are SSL+, SSLV2, SSLV3, TLS, TLSv1, and so on. Set this protocol value based on which Java Secure Socket Extension (JSSE) provider you use.
- **alias**: The string represents the alias in the key store. No default value exists. This property is used if the key store has multiple key pair certificates and you want to select one of the certificates.
- **keyStoreType**: Specifies the SSL key store type. Possible values are JKS+, JCEK, PKCS12 etc. Set this value based on which Java Secure Socket Extension (JSSE) provider you use.
- **keyStore**: Specifies the key store path file name that has the client public certificates and private keys. For example, `[OBJECTGRID_HOME]/properties/DummyClientKeyFile.jks`. In this release, hardware support is not supported.
- **keyStorePassword**: Specifies the password to protect the key store path. The password is encoded simply using "xor" algorithm by ObjectGrid. Use the PropFilePasswordEncoder tool to encode this property file. Here is an example of encoded password: `{x0r}CDo9Hgw\\`.
- **trustStoreType**: Specifies the trust store type. Possible values are JKS+, JCEK, PKCS12 etc. You can set this value based on which JSSE provider they use.

- **trustStore**: Specifies the trust store path file name which has the server public certificates. For example, `[OBJECTGRID_HOME]/properties/DummyClientTrustFile.jks`
- **trustStorePassword**: Specifies the password to protect the trust store path. The password is encoded simply using xor algorithm by ObjectGrid. Use tool PropFilePasswordEncoder to encode this property file. Here is an example of encoded password: `{x0r}CDo9Hgw\`
- **certReqSubjectDN**: This is the string that is required in the certificate subject distinguished name (DN) from the server. A client is allowed to connect to the server only if the server certificate DN contains this string. If the value is null, the client does not require a particular subject DN in the server certificate. For example, if the certificate subject DN is ″CN=Server1, OU=Your Organizational Unit, O=Your Organization, S=Your State,C=Your Country″, then ″CN=server1″, ″O=Your Organization″, ″OU=Your Organizational Unit, O=Your Organization, S=Your State,C=Your Country″ results in a match, but ″CN=server2″ and ″OU=Your Organizational Unit, L=smething, O=Your Organization, S=Your State,C=Your Country″ does not match. Wild card matching is not supported.

**Configure SSL Parameters for Object Servers**

SSL parameters on the client side can be configured in the `security.ogserver.props` file. This property file can be passed as an parameter when you launch an ObjectGrid server.

Except the previous SSL properties, the server side SSL configuration has an additional property:

- **clientAuthentication** (true+, false). If this property is set to true, the SSL client must be authenticated. This is different from the client certificate authentication. Client certificate authentication means authenticating a client to a user registry based on the certificate chain, while this property ensures the server connects to the right client.

## Local ObjectGrid security

This topic describes the security of local ObjectGrid programming model. In the local ObjectGrid programming model, the main security function is the authorization. The local ObjectGrid programming model does not support any authentication. You must authenticate outside of ObjectGrid. However, ObjectGrid does provide plug-ins to get and validate Subject objects.

Enabling ObjectGrid security can be done in two ways:

- **Configuration**. You can use the ObjectGrid XML file to define an ObjectGrid and enable the security for that ObjectGrid. Following is the `secure-objectgrid-definition.xml` file that is used in the ObjectGridSample enterprise application sample. In this XML file, security is enabled by setting the securityEnabled attribute to `true`.

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
  authorizationMechanism="AUTHORIZATION_MECHANISM_JASS">
  <bean id="TransactionCallback"
   classname="com.ibm.websphere.samples.objectgrid.HeapTransactionCallback" />
  ...
</objectGrids>
```

- **Programming**. If you want to create an ObjectGrid using APIs, call the following method on the ObjectGrid interface to enable the security:

```
/**
* Enable the ObjectGrid security
*/
void setSecurityEnabled();
```

In the local ObjectGrid programming model, no authentication exists. When you set security with this method, you are configuring authorization. This condition is consistent with the client server model. Enabling security for an ObjectGrid in the client server model only enables the authorization on that ObjectGrid instance.

## Authentication

In the local ObjectGrid programming model, ObjectGrid does not provide any authentication mechanism. ObjectGrid relies on the environment, either application servers or applications, for authentications. When an ObjectGrid is used in WebSphere Application Server or WebSphere Extended Deployment, applications can use the WebSphere Application Server security authentication mechanism. When an ObjectGrid is running in a Java 2 Platform, Standard Edition (J2SE) environment, the application has to manage authentications with Java Authentication and Authorization Service (JAAS) authentication or other authentication mechanisms. For more information about using JAAS authentication, see the JAAS reference guide.

The contract between an application and an ObjectGrid instance is the javax.security.auth.Subject object. After the client is authenticated by the application server or the application, the application can retrieve the authenticated javax.security.auth.Subject object and use this Subject object to get a session from the ObjectGrid instance by calling the ObjectGrid.getSession(Subject) method. This Subject object is used to authorize accesses to the map data. This contract is called a subject passing mechanism. Following is the ObjectGrid.getSession(Subject) API:

```
/**
* This API allows the cache to use a specific subject rather than the one
* configured on the ObjectGrid to get a session.
* @param subject
* @return An instance of Session
* @throws ObjectGridException
* @throws TransactionCallbackException
* @throws InvalidSubjectException the subject passed in is invalid based
* on the SubjectValidation mechanism.
*/
public Session getSession(Subject subject)
throws ObjectGridException, TransactionCallbackException, InvalidSubjectException;
```

The getSession method in the ObjectGrid interface can also be used to get a Session object:

```
/**
* This returns a Session object that can be used by a single thread at a time.
* It's not allowed to share this Session object between threads without placing a
* critical section around it. While the core framework allows the object to move
* between threads, the TransactionCallback and Loader may prevent this usage,
* especially in J2EE environments. When security is enabled, this will use the
* SubjectSource to get a Subject object.
*
* If the initialize() method has not been invoked prior to the first
* getSession invocation, then an implicit initialization will occur.  This ensures
* that all of the configuration is complete before any runtime usage is required.
*
* @see #initialize()
* @return An instance of Session
* @throws ObjectGridException
* @throws TransactionCallbackException
* @throws IllegalStateException if this method is called after the
*        destroy() method is called.
*/
public Session getSession()
throws ObjectGridException, TransactionCallbackException;
```

As the API documentation specifies, when security is enabled, this method uses the SubjectSource plug-in to get a Subject object. The SubjectSource plug-in is one of the security plug-ins defined in ObjectGrid to support propagating Subject objects. See "Security-related plug-ins" for more information.

The getSession(Subject) method can only be called on the local ObjectGrid instance. If you call the getSession(Subject) method on a client side in a distributed ObjectGrid configuration, an exception results.

## Security-related plug-ins

ObjectGrid provides two security plug-ins that are related to the subject passing mechanism: the SubjectSource and SubjectValidation plug-ins.

**SubjectSource plug-in**

The SubjectSource plug-in, represented by the com.ibm.websphere.objectgrid.security.plugins.SubjectSource interface, is a plug-in that is used to get a Subject object from an ObjectGrid running environment. This ObjectGrid environment can be an application that uses the ObjectGrid or an application server that hosts the application. The interface follows:

```
/**
* This plug-in can be used to get a Subject object which represents the
* ObjectGrid client.
* This subject is then used for ObjectGrid authorization. The method
* getSubject is called by the ObjectGrid runtime when the
* ObjectGrid.getSession() method is used to get a session and the
* security is enabled.
*
* This plug-in is useful for an already authenticated client: it
* can retrieve the authenticated Subject object and then pass to the
* ObjectGrid instance. Therefore, there is no need for another
* authentication.
*
* For example, use
* Subject.getSubject(AccessControlContext)
* to get the subject associated with the AccessControlContext and
* then return it in the getSubject implementation.
*
* This plug-in can only be used in a secure domain, such as in a
* ObjectGrid server.
*
* @ibm-api
* @since WAS XD 6.0
*/
public interface SubjectSource {

  /**
  * Get a Subject object which can represent the ObjectGrid client.
  *
  * @return a Subject object
  * @throws ObjectGridSecurityException any exception during the subject
```

```
 * retrieving
  */
  Subject getSubject() throws ObjectGridSecurityException;
}
```

Consider the SubjectSource plug-in an alternative to the subject passing mechanism. Using the subject passing mechanism, the application retrieves the Subject object and uses it to get the ObjectGrid session object. With the SubjectSource plug-in, the ObjectGrid runtime that retrieves the Subject object and uses it to get the session object. The subject passing mechanism gives the control of Subject objects to applications, while the SubjectSource plug-in mechanism frees applications from retrieving the Subject object.

This SubjectSource plug-in can be used to get a Subject object that represents an ObjectGrid client that is used for ObjectGrid authorization. When the ObjectGrid.getSession() method is called, the Subject getSubject() throws ObjectGridSecurityException() method is called by the ObjectGrid runtime, if security is enabled.

ObjectGrid provides a default implementation of this plug-in: com.ibm.websphere.objectgrid.security.plugins.builtins. WSSubjectSourceImpl. This implementation can be used to retrieve a caller subject or RunAs subject from the thread when an application is running in WebSphere Application Server. You can configure this class as the SubjectSource implementation class when using ObjectGrid in WebSphere Application Server. Following is a code snippet that shows the main flow of the WSSubjectSourceImpl.getSubject():

```
Subject s = null;
try {
  if (finalType == RUN_AS_SUBJECT) {
    // get the RunAs subject
    s = com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();
  }
  else if (finalType == CALLER_SUBJECT) {
    // get the callersubject
    s = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();
  }
}
catch (WSSecurityException wse) {
  throw new ObjectGridSecurityException(wse);
}

return s;
```

For other details, refer to the API Documentation for the SubjectSource plug-in and the WSSubjectSourceImpl implementation.

**SubjectValidation plug-in**

The SubjectValidation plug-in, represented by the com.ibm.websphere.objectgrid.security.plugins.SubjectValidation interface, is another security plug-in. The SubjectValidation plug-in can be used to validate that a javax.security.auth.Subject, either passed to the ObjectGrid or retrieved by the SubjectSource plug-in, is a valid Subject which has not been tampered with. Here is the interface.

```
/**
* This plug-in can be used to validate a javax.security.auth.Subject
* passed to the ObjectGrid is a valid subject which has not been
* tampered with.
*
```

```
* An implementation of this plug—in needs support from the Subject
* object creator, because only the creator knows whether the Subject object
* has been tampered with. However, a subject creator may
* not know whether the Subject has been tampered with. In this
* case, this plug-in should not be used.
*
* This plug-in can only be used in a secure domain, such as in a
* application server. Do not put this plug-in on the client side, it is
*  ignored.
*
* @ibm-api
*
* @since WAS XD 6.0
*/
public interface SubjectValidation {

  /**
  * Validate the Subject has not been tampered with.
  * @param subject a subject to be validated
  * @return the validated Subject object
  * @throws InvalidSubjectException
  */
  Subject validateSubject(Subject subject) throws
   InvalidSubjectException;

}
```

The Subject validateSubject (Subject subject) throws InvalidSubjectException; method in the SubjectValidation interface takes a Subject object and returns a Subject object. Whether a Subject object is considered valid and what Subject object is returned are all up to your implementations. If the Subject object is not valid, an `InvalidSubjectException` results.

You can use this plug-in if you do not trust the Subject object passed to this method. This case is rare considering that we trust the application developers who develop the code to retrieve the Subject object.

An implementation of this plug-in needs support from the Subject object creator because only the creator knows if the Subject object has been tampered with. However, some subject creator might not know if the Subject has been tampered with. In this case, this plug-in is not useful.

ObjectGrid provides a default implementation of SubjectValidation: com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectValidationImpl. This implementation can be used to validate the WebSphere authenticated subject. Users can configure this class as the SubjectValidation implementation class when using ObjectGrid in WebSphere Application Server. The WSSubjectValidationImpl implementation considers a Subject object valid if and only if the credential token associated with this Subject has not been tampered with. In other words, you could change other parts of the Subject object. The WSSubjectValidationImpl implementation asks WebSphere Application Server for the original Subject corresponding to the credential token and returns the original Subject object as the validated Subject object. Therefore, the changes made to the Subject contents other than the credential token have no effects. The code snippet below shows the basic flow of the WSSubjectValidationImpl.validateSubject(Subject):

```
// Create a LoginContext with scheme WSLogin and
// pass a Callback handler.
LoginContext lc = new LoginContext("WSLogin",
new WSCredTokenCallbackHandlerImpl(subject));
```

```
// When this method is called, the callback handler methods
// will be called to log the user in.
lc.login();

// Get the subject from the LoginContext
return lc.getSubject();
```

In the previous code snippet, a credential token callback handler object, WSCredTOkenCallbackHandlerImpl, is created with the Subject object to be validated. Then a LoginContext is created with login scheme ″WSLogin″. When the lc.login() method is called, WebSphere Application Server security retrieves the credential token from the Subject object and then returns the correspondent Subject as the validated Subject object.

For other details, refer to the API documentation of SubjectValidation and WSSubjectValidationImpl.

**Plug-in configuration**

The SubjectValidation plug-in and SubjectSource plug-in can be configured in two ways:

- **Configuration**. You can use the ObjectGrid XML file to define an ObjectGrid and set these two plug-ins. Here is an example, in which the WSSubjectSourceImpl class is configured as the SubjectSource plug-in and the WSSubjectValidation class is configured as the SubjectValidation plug-in.

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
  authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS">
        <bean id="SubjectSource"
className="com.ibm.websphere.objectgrid.security.plugins.builtins.
 WSSubjectSourceImpl" />
    <bean id="SubjectValidation"
 className="com.ibm.websphere.objectgrid.security.plugins.builtins.
 WSSubjectValidationImpl" />
    <bean id="TransactionCallback"
className="com.ibm.websphere.samples.objectgrid.
 HeapTransactionCallback" />
...
</objectGrids>
```

- **Programming.** If you want to create an ObjectGrid through APIs, you can call the following methods to set the SubjectSource or SubjectValidation plug-ins.

```
/**
* Set the SubjectValidation plug-in for this ObjectGrid instance. A
* SubjectValidation plug-in can be used to validate the Subject object
* passed in is a valid Subject. Refer to {@link SubjectValidation}
* for more details.
* @param subjectValidation the SubjectValidation plug-in
*/
void setSubjectValidation(SubjectValidation subjectValidation);


/**
* Set the SubjectSource plug-in. A SubjectSource plug-in can be used
* to get a Subject object from the environment to represent the
* ObjectGrid client.
*
* @param source the SubjectSource plug-in
*/
void setSubjectSource(SubjectSource source);
```

### Write your own JAAS authentication code

You can write you own JAAS authentication code to handle the authentication. You need to write your own login modules and then configure the login modules for your authentication module.

The login module receives information about a user and authenticates the user. This information can be anything that can identify the user. For example, it can be a user ID and password, client certificate, and so on. After receiving the information, the login module verifies that it represents a valid subject and then creates a Subject object. Currently, several implementations of login modules are available to public.

After a login module is written, configure this login module so it can be used by the runtime. A JAAS login module configuration file must be configured. This login module contains the login module and its authentication scheme. For example:

```
FileLogin
{
    com.acme.auth.FileLoginModule required
};
```

The authentication scheme is ″FileLogin″ and the login module is com.acme.auth.FileLoginModule. The required token indicates that the FileLoginModule module must validate this login or the scheme as a whole fails.

Setting the JAAS login module configuration file can be done by one of the following ways:

- Set the JAAS login module configuration file in the **login.config.url** in the `java.security` file, for example, `login.config.url.1=file:${java.home}/lib/security/file.login`
- Set the JAAS login module configuration file from the command line by using JVM arguments **-Djava.security.auth.login.config**, for example `-Djava.security.auth.login.config ==$JAVA_HOME/lib/security/file.login`

For more information about how to write and configure login modules, see the JAAS Authentication Tutorial.

If your code is running in WebSphere Application Server, you must configure the JAAS login in the administrative console and store this login configuration in the application server configuration. See Login configuration for Java Authentication and Authorization Service for details.

## Authorization

After the client is authenticated, you can use ObjectGrid authorization mechanisms to authorize access to the ObjectGrid map data and the management tasks. ObjectGrid authorization is based on the Subject object. ObjectGrid supports two kinds of authorization mechanisms: Java Authentication and Authorization Service (JAAS) authorization and custom authorization.

### Permission class

Two different kinds of ObjectGrid authorization exist: authorization to the data in the map, and authorization to the management tasks. Each authorization uses a Permission class. The permission to access the map is represented by the MapPermission class, and the permission to run the management tasks is represented by the AdminPermission class.

**MapPermission class**

In ObjectGrid, the com.ibm.websphere.objectgrid.security.MapPermission public class represents permissions to the ObjectGrid resources, specifically the methods of ObjectMap or JavaMap interfaces. ObjectGrid defines the following permission strings to access the methods of ObjectMap and JavaMap:

- **read**: Grants permission to read the data from the map. The integer constant is defined as MapPermission.READ
- **write**: Grants permission to update the data in the map. The integer constant is defined as MapPermission.WRITE.
- **insert**: Grants permission to insert the data into the map. The integer constant is defined as MapPermission.INSERT.
- **remove**: Grants permission to remove the data from the map. The integer constant is defined as MapPermission.REMOVE.
- **invalidate**: Grants permission to invalidate the data from the map. The integer constant is defined as MapPermission.INVALIDATE.
- **all**: Grants all permissions: read, write, insert, remote, and invalidate. The integer constant is defined as MapPermission.ALL.

You can construct a MapPermission object by passing the fully qualified ObjectGrid map name (in format [ObjectGrid_name].[ObjectMap_name]) and the permission string or integer value. A permission string can be a comma-delimited string of the above permissions strings such as ″read, insert″, or it can be ″all″ which means all permissions are granted. A permission integer value can be any above permission integer constants or a mathematical “or” value of several integer permission constants, such as DGMapPermission.GET|DGMapPermission.PUT.

The authorization occurs when a client calls a method of ObjectMap or JavaMap. The ObjectGrid runtime checks different permissions for different methods. If the required permissions are not granted to the client, an AccessControlException results.

*Table 14. List of methods and their required permissions*

| | **com.ibm.websphere.objectgrid.ObjectMap com.ibm.websphere.objectgrid.JavaMap** |
|---|---|
| read | boolean containsKey(Object) |
| | boolean equals(Object) |
| | Object get(Object) |
| | Object get(Object, Serializable) |
| | List getAll(List) |
| | List getAll(List keyList, Serializable) |
| | List getAllForUpdate(List, Serializable) |
| | Object getForUpdate(Object) |
| | Object getForUpdate(Object, Serializable) |

Table 14. List of methods and their required permissions (continued)

| | **com.ibm.websphere.objectgrid.ObjectMap com.ibm.websphere.objectgrid.JavaMap** |
|---|---|
| write | Object put(Object key, Object value) |
| | void put(Object, Object, Serializable) |
| | void putAll(Map) |
| | void putAll(Map, Serializable) |
| | void update(Object, Object) |
| | void update(Object, Object, Serializable) |
| insert | public void insert (Object, Object) |
| | void insert(Object, Object, Serializable) |
| | remove Object remove (Object) |
| | void removeAll(Collection) |
| invalidate | public void invalidate (Object, boolean) |
| | void invalidateAll(Collection, boolean) |
| | void invalidateUsingKeyword(Serializable) |
| | int setTimeToLive(int) |

Authorization is based solely on which method is used, instead on what the method really does. For example, a put method can insert or update an record based on whether the record exists. However, the insert or update case are not distinguished at this moment.

Notice also that an operation type could be achieved by combinations of other types. For example, an update can be achieved by a remove and then an insert. Design your authorization policies with this in mind.

**AdminPermission**

The administration permission is represented by the com.ibm.websphere.objectgrid.security.AdminPermission class. ObjectGrid defines two permission actions for administration permissions:

- **admin**: Grant permissions to do any administration tasks.
- **monitor**: Grant permissions to actions that only are read-access-only administration tasks.

The detailed operations granted to users with different permissions are listed in the following table. These operations correspond to the methods in the ManagementMBean interface:

Table 15. Relationship between management tasks and admin permissions

| **operations** | **admin** | **monitor** |
|---|---|---|
| startServer | Y | N |
| stopServer | Y | N |
| forceStopServer | Y | N |
| setServerTrace | Y | N |
| retrieveServerStatus | Y | Y |
| getMapStats | Y | Y |
| getOGStats | Y | Y |
| getReplicationStats | Y | Y |

If the client has admin permission, it can run the startServer task; if the client has monitor permission, it cannot run the startServer task.

## Authorization mechanisms

ObjectGrid supports two kinds of authorization mechanisms: JAAS authorization and custom authorization. This applies to both map data access authorization and admin authorization. JAAS authorization augments the Java security policies with user-centric access controls. Permissions can be granted based not just on what code is running but also on who (principal) is running it. It is part of the JDK 1.4.

ObjectGrid also supports custom authorization with the com.ibm.websphere.objectgrid.security.plugins.MapAuthorization plug-in and com.ibm.websphere.objectgrid.security.plugins.AdminAuthorization plug-in. You can implement your own authorization mechanism if you do not want to use JAAS authorization. By using custom authorization mechanism, you can use the policy database, policy server, or Tivoli Access Manager to manage the ObjectGrid authorizations.

ObjectGrid authorization mechanism can be configured in two ways:

- **Configuration**. You can use the ObjectGrid XML file to define an ObjectGrid and set the authorization mechanism to either AUTHORIZATION_MECHANISM_JAAS or AUTHORIZATION_MECHANISM_CUSTOM. Here is the `secure-objectgrid-definition.xml` file that is used in the ObjectGridSample enterprise application sample.

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
  authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS">
  <bean id="TransactionCallback"
classname="com.ibm.websphere.samples.objectgrid.HeapTransactionCallback" />
...
</objectGrids>
```

- **Programming**. If you want to create an ObjectGrid using APIs, you can call the following method to set the authorization mechanism. This only applies to the local ObjectGrid programming model when you directly instantiates the ObjectGrid instance.

```
/**
* Set the authorization Mechanism. The default is
* com.ibm.websphere.objectgrid.security.SecurityConstants.
* AUTHORIZATION_MECHANISM_JAAS.
* @param authMechanism the map authorization mechanism
*/
void setAuthorizationMechanism(int authMechanism);
```

**JAAS Authorization**

A javax.security.auth.Subject object represents an authenticated user. A Subject is comprised of a set of principals, and each Principal represents an identity for that user. For example, a Subject could have a name principal (″Joe Smith″) and a group principal (″manager″).

Using JAAS authorization policy, permissions can be granted to specific Principals. ObjectGrid associates the Subject with the current access control context. For each method call to the ObjectMap or Javamap, the Java runtime automatically determines if the policy grants the required permission only to a specific Principal and if so, the operation is allowed only if the Subject associated with the access control context contains the designated Principal.

You must be familiar with the policy syntax of the policy file. For detailed description of JAAS authorization, refer to theJAAS Authorization Tutorial.

ObjectGrid has a special codebase used for checking the JAAS authorization to the ObjectMap and JavaMap method calls. This special codebase is `http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction`. Use this code base when granting ObjectMap or JavaMap permissions to principals. This special code was created because the ObjectGrid Java archive (JAR) file is granted with all permissions.

The template of the policy to grant MapPermission is:

```
grant codeBase "http://www.ibm.com/com/ibm/ws/objectgrid/security/
 PrivilegedAction"
   <Principal field(s)>{
    permission com.ibm.websphere.objectgrid.security.MapPermission
   "[ObjectGrid_name].[ObjectMap_name]", "action";
    ....
    permission com.ibm.websphere.objectgrid.security.MapPermission
   "[ObjectGrid_name].[ObjectMap_name]", "action";
  };
```

A Principal field looks like the following:

```
Principal Principal_class "principal_name"
```

That is, it is the word ″Principal″ followed by the fully qualified name of a Principal class and a principal name. The map_name is the fully qualified map name in the format of [ObjectGrid Name].[Map Name], for example, ″secureClusterObjectGrid.employees″. The action is a comma-delimited string of the permissions strings defined in MapPermission class, such as “read, insert”, or “all”.

Limited wildcard function is supported. You can replace the ObjectGrid name or map name with “*” to indicate “any”. However, ObjectGrid does not support replacing part of the ObjectGrid name or map name with “*”. Therefore, “*.employees”, “clusterObjectGrid.*”, and “*.*” are all valid names, but “cluster*.employees” is not valid

For example, in the `ObjectGridSample.ear` sample application, two authorization policy files are defined: `fullAccessAuth.policy` and `readInsertAccessAuth.policy`. The content of `readInsertAccessAuth.policy` is as follows:

```
grant codebase "http://www.ibm.com/com/ibm/ws/objectgrid/security/
 PrivilegedAction"
    Principal com.ibm.ws.security.common.auth.WSPrincipalImpl
   "principal_name" {
    permission com.ibm.websphere.objectgrid.security.MapPermission
        "secureClusterObjectGrid.employees", "read,insert";
    permission com.ibm.websphere.objectgrid.security.MapPermission
        "secureClusterObjectGrid.offices", "read,insert";
    permission com.ibm.websphere.objectgrid.security.MapPermission
        "secureClusterObjectGrid.sites", "read,insert";
    permission com.ibm.websphere.objectgrid.security.MapPermission
        "secureClusterObjectGrid.counters", "read,insert";
};
```

In this policy, only ″insert″ and ″read″ permissions are granted to these four maps to a certain principal. The other policy file, `fullAccessAuth.policy`, grants ″all″ permissions to these maps to a principal. Before running the application, change the principal_name and principal class to appropriate values. The value of the principal_name depends on the user registry. For example, if local OS is used as user registry , the machine name is MACH1, and the user ID is user1, the principal_name is ″MACH1/user1″.

JAAS authorization policy can be directly put in the Java policy file, or it can be put in a separate JAAS authorization file and then set it by using the `-Djava.security.auth.policy=file:[JAAS_AUTH_POLICY_FILE]` JVM argument or using `auth.poliyc.url.x=file:[JAAS_AUTH_POLICY_FILE]` in the `java.security` file.

The description of JAAS authorization also applies when you want to write and configure the policies for authorizing access to the management tasks. The only difference will be that instead of using ″com.ibm.websphere.objectgrid.security.MapPermission map name, actions;″ format, you use ″com.ibm.websphere.objectgrid.security.AdminPermission action;″. The action could be either ″admin″ or ″monitor″.

**Custom map authorization**

ObjectGrid also supports the custom map authorization by the MapAuthorization plug-in. The interface follows:

```
/**This plugin can be used to authorize ObjectMap/JavaMap accesses to the
 * principals represented by the Subject object.
  *
 * A typical implementation of this plug-in is to retrieve the
 * principals from the Subject object, and then check whether
 * the specified permissions are granted to the principals.
 *


 *
 * @ibm-api
 * @since WAS XD 6.0
 */
public interface MapAuthorization {

    /**
     * Check whether the principals represented by the Subject object
     * in the subject has the specified MapPermission. If
     * the permissions are granted, true is returned; otherwise a false
     * is returned.
     *
     * @param subject the subject
     * @param permission the permission to access ObjectMap
     *
     * @return true if the permission is granted; false otherwise.
     */
    boolean checkPermission(Subject subject, MapPermission permission);

}
```

This plug-in can be used to authorize ObjectMap and JavaMap accesses to the principals contained in the Subject object. The following method:

```
boolean checkPermission(Subject subject, MapPermission permission)
```

The MapAuthorization interface is called by the ObjectGrid runtime to check whether the passed-in subject object has the passed-in permission. The implementation of the MapAuthorization interface returns true if so and false otherwise.

A typical implementation of this plug-in is to retrieve the principals from the Subject object and check whether the specified permissions are granted to the principals by consulting specific policies. These policies are defined by users. For example, the policies can be defined in a database, a plain file, or a Tivoli Access Manager policy server.

ObjectGrid provides two default implementations for this plug-in. The com.ibm.websphere.objectgrid.security.plugins.builtins. JAASMapAuthorizationImpl class is an implementation of MapAuthorization that uses JAAS mechanism for authorization. Another implementation class is the com.ibm.websphere.objectgrid.security.plugins.builtins. TAMMapAuthorizationImpl class. It shows how Tivoli Access Manager can be used to manage the ObjectGrid authorizations. Following is a code snippet that shows the basic flow of the JAASMapAuthorizationImpl.checkPermission(Subject, MapPermission):

```
// Creates a PrivilegedExceptionAction to check the permssions.
PrivilegedExceptionAction action =
 MapPermissionCheckAction.getInstance(permission);

Subject.doAsPrivileged(subject, action, null);
```

See the IBM Tivoli Access Manager Authorization Java Classes Developer Reference for more details.

Do not use this TAMMapAuthorizationImpl plug-in in an out-of-box scenario. Use this plug-in for testing purposes only. It requires certain restrictive preconditions:

- The Subject object contains a com.tivoli.mts.PDPrincipal principal.
- The TAM policy server has defined the following permissions for the ObjectMap or JavaMap name object. The object defined in the policy server should have the same name as the ObjectMap or JavaMap name in the format of [ObjectGrid_name].[ObjectMap_name]. The permission is the first character of the permission strings defined in the MapPermission. For example, the permission ″r″ defined in the policy server represents the ″read″ permission to the ObjectMap.

The following snippet demonstrates how to implement the checkPermission method:

```
/**
* @see com.ibm.websphere.objectgrid.security.plugins.
*   MapAuthorization#checkPermission
* (javax.security.auth.Subject, com.ibm.websphere.objectgrid.security.
*   MapPermission)
*/
public boolean checkPermission(final Subject subject,
 MapPermission permission) {

  String[] str = permission.getParsedNames();

  StringBuffer pdPermissionStr = new StringBuffer(5);
  for (int i=0; i<str.length; i++) {
    pdPermissionStr.append(str[i].substring(0,1));
  }
  PDPermission pdPerm = new PDPermission(permission.getName(),
  pdPermissionStr.toString());

  Set principals = subject.getPrincipals();

  Iterator iter= principals.iterator();
  while(iter.hasNext()) {
    try {
      PDPrincipal principal = (PDPrincipal) iter.next();
      if (principal.implies(pdPerm)) {
        return true;
      }
    }
    catch (ClassCastException cce) {
      // Handle exception
    }
  }
  return false;
}
```

The MapAuthorization plug-in can be configured in the following ways:

- **Configuration**. You can use the ObjectGrid XML file to define an MapAuthorization plug-in. Here is an example:

  ```
  <objectGrids>
   <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
    authorizationMechanism="AUTHORIZATION_MECHANISM_CUSTOM">
  ...
    <bean id="MapAuthorization"
  className="com.ibm.websphere.objectgrid.security.plugins.builtins.
  JAASMapAuthorizationImpl" />
  </objectGrids>
  ```

- **Programming**. If you want to create an ObjectGrid using APIs, you can call the following method to set the authorization plug-in. This only applies to the local ObjectGrid programming model when you directly instantiate the ObjectGrid instance.

  ```
  /**
  * Sets the MapAuthorization plug-in for this ObjectGrid instance.
  *
  * A {@link MapAuthorization} plug-in can be used to authorize
  * access to the maps. Refer to {@link MapAuthorization}
  * for more details.
  * @param mapAuthorization the MapAuthorization plug-in
  */
  void setMapAuthorization(MapAuthorization mapAuthorization);
  ```

### Custom admin authorization

Like the custom map data access authorization support, ObjectGrid supports the custom admin authorization. The plug-in is com.ibm.websphere.objectgrid.security.plugins.AdminAuthorization.

```
/**
  * This plug-in can be used to authorize management operations to the
 * principals contained in the Subject object. The permissions for the
 * management operations are represented by AdminPermission
 * objects.
  *
  * This plug-in is used in an ObjectGrid server. It can be configured in the
 * ObjectGrid cluster XML file.


  *
  * A typical implementation of this plug-in is to retrieve the
 * Principal set from the Subject object, and then
 * check whether the specified permissions are granted to these principals.
```

```
*

 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see AdminPermission
 */
public interface AdminAuthorization {

    /**
     * Checks whether the user represented by the Subject object has the
     * specified AdminPermission or not.
     *
* If the permissions are granted, true is returned; otherwise
* false is returned.
*
* @param subject the Subject object representing the user
* @param permission the administration permission to check
*
* @return true if the permission is granted; false otherwise.
*
* @see AdminPermission
*/
boolean checkPermission(Subject subject, AdminPermission permission);
}
```

This plug-in can be used to authorize admin accesses to the principals contained in the Subject object. The method

```
boolean checkPermission(Subject subject, AdminPermission permission)
```

in the AdminAuthorization interface, is called by the ObjectGrid runtime to check whether the passed-in Subject object has the passed-in admin permission. The implementation of the AdminAuthorization interface should return true if so and false otherwise.

You can implement this interface based on your security requirements. ObjectGrid does not ship an implementation class for this interface.

You can set the AdminAuthorization plug-in on the cluster level in the cluster XML. An example follows:

```
<cluster name="cluster1" securityEnabled="true"
 singleSignOnEnabled="true" loginSessionExpirationTime="300"
 statisticsEnabled="true"
 statisticsSpec="map.all=enabled">
 <serverDefinition name="server1" host="localhost"
  clientAccessPort="12503" peerAccessPort="12500" workingDirectory=""
  traceSpec="ObjectGrid=all=disabled"
  systemStreamToFileEnabled="true" />
 <serverDefinition name="server2" host="localhost"
  clientAccessPort="12504" peerAccessPort="12501" workingDirectory=""
  traceSpec="ObjectGrid=all=disabled"
  systemStreamToFileEnabled="true" />
 <authenticator className ="com.ibm.websphere.objectgrid.security.plugins.
  builtins.WSTokenAuthenticator"></authenticator>
 <adminAuthorization className= "com.ibm.ws.objectgrid.test.security.util.
  TestAdminAuthorization"></adminAuthorization>
</cluster>
```

## Permission checking period

ObjectGrid supports caching the map permission checking results for performance reason. Without this mechanism, when a method listed on Table 14 on page 162 is called, ObjectGrid runtime calls the configured authorization mechanism to authorize the access. With this permission checking period being set, the authorization mechanism is called periodically based on the permission checking period.

We cache the permission authorization information based on the Subject object. When a client tries to access the methods, the ObjectGrid runtime will look up the cache based on the Subject object. If it cannot be found in the cache, the runtime will check the permissions granted for this Subject object, and then store the permissions in a cache.

The permission checking period must be defined before the ObjectGrid is initialized. The permission checking period can be configured in two ways:

- **Configuration**. You can use the ObjectGrid XML file to define an ObjectGrid and set the permission check period. Here is an example to set the permission check period to 45 seconds.

```
<objectGrids>
 <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
 authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS"
 permissionCheckPeriod="45">
  <bean id="bean id="TransactionCallback"
className="com.ibm.websphere.samples.objectgrid.HeapTransactionCallback" />
...
</objectGrids>
```

- **Programming**. If you want to create an ObjectGrid with APIs, call the following method to set the permission checking period. This method can only be called before the ObjectGrid instance is initialized. This method only applies to the local ObjectGrid programming model when you instantiate the ObjectGrid instance directly.

```
/**
* This method takes a single parameter indicating how often the customer
* wants to check the permission used to allow a client access. If the
* parameter is 0 then every single get/put/update/remove/evict call will
* ask the authorization mechanism, either JAAS authorization or custom
* authorization to check if the current subject has permission. This may be
* prohibitively expensive from a performance point of view depending on
* the authorization implementation but if this is required then you can do it.
* Alternatively, if the parameter is > 0 then it indicates the number
* of seconds to cache a set of permissions before returning to
* the authorization mechanism to refresh them. This provides much
* better performance but you run the risk that if the backend
* permissions are changed during this time then the ObjectGrid will
* possibly allow or prevent access even though the backend security
* provider has been modified.
*
* @param period the permission check period in seconds.
*/
void setPermissionCheckPeriod(int period);
```

## ObjectGrid cluster security

The ObjectGrid cluster security ensures that a joining server has the right credential, so a malicious server cannot join the cluster. ObjectGrid uses a shared secret string mechanism for this purpose.

All the ObjectGrid servers agree on a shared secret string. When a server joins the cluster, it is challenged to present the secret string. If the secret string of the joining server matches the one in the president server, the joining server can join the cluster; otherwise the join request is rejected.

Sending a clear text secret is not secure. ObjectGrid security infrastructure provides a secure token manager plug-in to allow the server to “secure” this secret before sending it. You must decide how to implement the “secure” operation. ObjectGrid provides an out-of-box implementation, in which the “secure” operation is implemented to encrypt and signing the secret.

The secret string (authenticationSecret) is set on the `security.ogserver.props` file:

- `authenticationSecret`: the secret string to challenge the server. When a server starts up, it needs to present this string to the president server. If the secret string matches what in the president server, this server is allowed to join the cluster.

## SecureTokenManager plug-in

A secure token manager plug-in is represented by the com.ibm.websphere.objectgrid.security.plugins.SecureTokenManager interface. The interface follows:

```
package com.ibm.websphere.objectgrid.security.plugins;

import com.ibm.websphere.objectgrid.security.ObjectGridSecurityException;
import com.ibm.websphere.objectgrid.security.SecurityConstants;

/**
 * This interface is used by ObjectGrid servers to transform an object to a
 * secure token and vice versa. A secure token is a byte array.
* Here is one example of a possible usage: When a server joins the cluster,
* the joining server needs to present a password to the president server in the
* cluster. Before sending the password out, the joining server calls the
* generateToken(Object) method to generate a token for this
* password. The token should be hard to break so the password can be protected
* securely. The token will then be sent across the wire. Usually the token is
* associated with a time stamp so the malicious replay attack will be difficult.
* On the receving side, the server calls the verifyToken(byte[])
* method to verfiy the token and reconstruct the corresponding object from the
* token.
*
*ObjectGrid utilizes JCE to provide a default implementation of this
* interface. In this implementation, when generating the token, the object is
* encrypted with a time stamp and then signed. To verify a token, the token's
* signature is verified and then decrypted. This implementation will need a key
* store configured in the ObjectGrid servers to support the data
* encrypting and decrypting and signature signing and verifying. Please use
* security.ogserver.props for the secure token key settings.
*
* An implementation class should have a default constructor. Users can set the
* CustomSecureTokenManagerProps property in the server security configuration
* property file. This property will be set on the object using the
* setProperties(String) method.
*
* @ibm-api
* @since WAS XD 6.0.1
*
* @see SecurityConstants#SECURE_TOKEN_MANAGER_CUSTOM_STRING
* @see SecurityConstants#SECURE_TOKEN_MANAGER_DEFAULT_STRING
*/
public interface SecureTokenManager {

/**
*

Set the user defined properties to the factory
*
*
This method is used to set addtional SecureTokenManager properties
* to the object. These properties can be set using the SecureTokenManagerProps
* property in the server security configuration property file.
* This way, you can customize your factory.
*
* @param properties user defined properties
*/
void setProperties(String properties);

/**
* Generates the token for the specified object.
*


* The generated token should be hard to break.
*
* @param o the object to be protected
*
* @return a token representing the object to be protected
*
* @throws ObjectGridSecurityException if any exception occurs during
* generation of the token byte array
*/
byte[] generateToken(Object o) throws ObjectGridSecurityException;

/**
* Verifies the token and reconstruct the object.
*
* @param bytes the token byte array representing the protected object.
*
* @return the protected object
*
* @throws ObjectGridSecurityException if any exception occurs during
* verification of the token byte array
*/
Object verifyToken(byte[] bytes) throws ObjectGridSecurityException;
}
```

The generateToken(Object) method takes an object to be protected, and then generates a token that cannot be understood by others. The verifyTokens(byte[]) method does the reverse process: it converts the token back to the original object.

A simple SecureTokenManager implementation is to use a simple encoding algorithm (such as the XOR algorithm) to encode the object in serialized form, and then use corresponding decoding algorithm to decode the token. This implementation is not secure and is easy to break.

ObjectGrid provides an out-of-box implementation for this interface. The implementation is not a public API and is transparent to you.

The default implementation uses a key pair to sign and verify the signature, and uses a secret key to encrypt the content. To do this, every server has a JCKES type keystore to store the key pair (private key and public key) and a secret key. The keystore has to be JCKES type to store secret keys.

These keys are used to encrypt and sign or verify the secret string on the sending end. Also, the token is associated with an expiration time, so it expires after certain amount of time. On the receiving end, the data is verified, decrypted, and compared to the receiver’s secret string. SSL-like communication protocols are not required between a pair of servers for authentication, because the private keys and public keys serves the same purpose. However, if server communication is not encrypted, the data could be stolen by poking at the communication. Because the token expires soon, the replay attack threat is minimized. This possibility is significantly decreased if all servers are deployed behind a firewall.

The disadvantage of this approach is that the ObjectGrid administrators have to generate keys and transport them to all servers, which could cause security problems.

## Configurations

To use the secure token manager, the following properties should be configured in the `security.ogserver.props` file:

- **secureTokenManagerType** property: This property indicates which secure token manager to use.
  - If the value is **none**, no secure token manager is used.
  - If the value is **default**, the default out-of-box provided secure token manager is used.
  - If the value is **custom**, the user-provided secure token manager is used.
- **customSecureTokenManagerClass** property: This property specifies the SecureTokenManager implementation class. It is only used if the secureTokenManagerType value is ″custom″. The implementation class must have a default constructor to be instantiated.
- **customSecureTokenManagerProps** property: This property specifies the custom SecureTokenManager properties. It is only used if the secureTokenManagerType value is ″custom″. The value is set to the SecureTokenManager Object with the setProperties(String) method.
- If the secureTokenManagerType value is set to default, then the following configurations for the signing and ciphering keys are needed:
  - `secureTokenKeyStore`: Specifies the file path name for the key store that stores the public-private key pair and the secret key.
  - `secureTokenKeyStoreType`: Specifies the key store type, for example, JCKES. You can set this value based on the Java Secure Socket Extension (JSSE) provider that you use. However, this key store should be able to support secret keys.
  - `secureTokenKeyStorePassword`: Specifies the password to protect the key store.
  - `secureTokenKeyPairAlias`: Specifies the alias of the public-private key pair used for the singing and verifying.
  - `secureTokenKeyPairPassword`: Specifies the password to protect the key pair alias used for signing and verifying.
  - `secureTokenSecretKeyAlias`: Specifies the secret key alias used for ciphering.
  - `secureTokenSecretKeyPassword`: Specifies the password to protect the secret key.
  - `secureTokenCipherAlgorithm`: Specifies the algorithm used for the ciphering. You can set this value based on the JSSE provider you use.
  - `secureTokenSignAlgorithm`: Specifies the algorithm used for signing the object. You can set this value based on the JSSE provider you use.

## Gateway security

ObjectGrid management gateway serves as a point to delegate the client administration requests to the ObjectGrid server. This topic describes how to secure the gateway access.

The following diagram is an example. If the ObjectGrid client wants to get the statistics from the a cluster, it first sends a request to the gateway. The gateway sends this request to both servers to get the statistics and then combines the statistics. The combined statistics send back to the client.

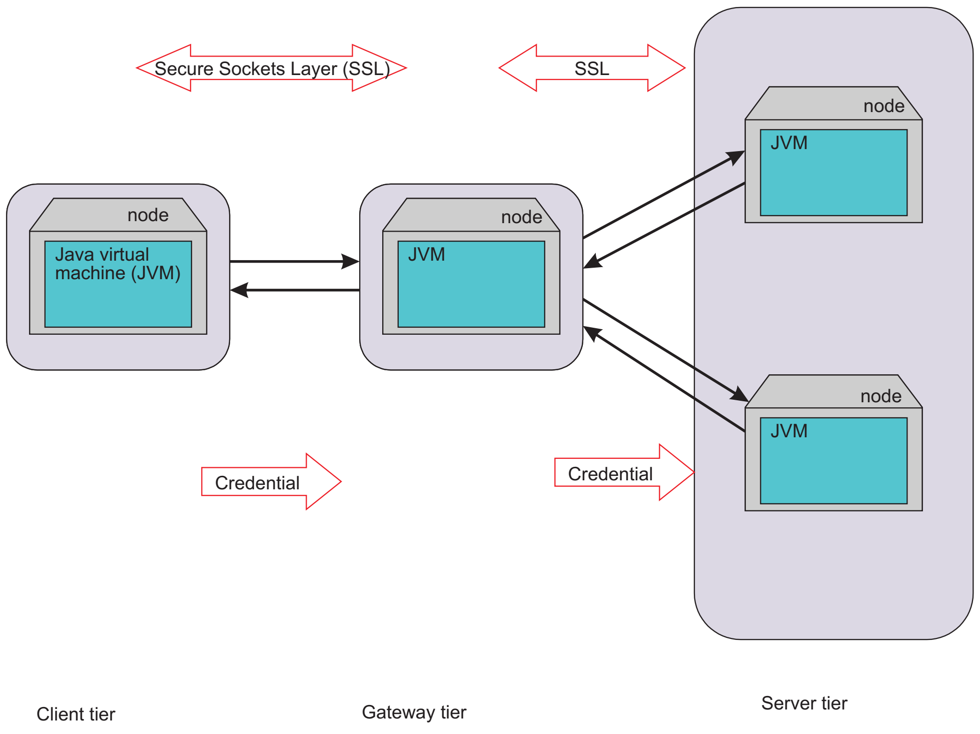


*Figure 18. Gateway security*

The gateway and server communication uses the ObjectGrid client server communication mechanism. The gateway is treated as an ObjectGrid client. The client and gateway communication can be secured by SSL. This capability is provided by the JMX connector layer, which is the open source project mx4j. ObjectGrid requires mx4j in place to make gateway work.

For the authentication, the gateway propagates the credential, for example, a user ID and password, that is presented by the client to the server. Both authentication and authorization are enforced on ObjectGrid servers.

Client certificate authentication for the gateway client is not supported.

### Gateway server security

A gateway server is an ObjectGrid client. All the security aspects are the same as an ObjectGrid client. Refer to “Start the management gateway server” on page 84 for more details on how to start a gateway server from a command line.

The following code snippet demonstrates how to start the secure gateway programmatically:

```
// Get the ClientSecurityConfiguration from the client security property file
ClientSecurityConfiguration csConfig = ClientSecurityConfigurationFactory
.getClientSecurityConfiguration("etc/test/security/security.client.props");
CredentialGenerator creGen = new UserPasswordCredentialGenerator("admin",
 "xxxxxx");
csConfig.setCredentialGenerator(credGen);
// Initialize the gateway
ManagementGateway gateway = ManagementGatewayFactory.getManagementGateway();
gateway.setConnectorPort(namingPort);
gateway.setClusterName("cluster1");
gateway.setHost("localhost");
gateway.setPort("12503");
gateway.setTraceEnabled(true);
gateway.setTraceSpec("ObjectGrid=all=enabled");
gateway.setTraceFile("logs/GatewayTrace.log");

// Set the ClientSecurityConfiguration object
gateway.setCsConfig(csConfig);

// Start the gateway
gateway.startConnector();
```

In the previous code, a ClientSecurityConfiguration object is created and set on the ManagementGateway instance.

## Gateway client security

The gateway client needs to pass a credential to a gateway server at the connect time. The following code snippet demonstrates how to pass a credential:

```
/**
* retrieve the server status from the gateway
*/
public boolean retrieveServerStatus()
throws Exception {

  String serverProtocol = "rmi";
  String serverHost = "host";
  String namingHost = "localhost";

  String jndiPath = "/jmxconnector";

  JMXServiceURL url = new JMXServiceURL("service:jmx:" + serverProtocol + "://"
  + serverHost + "/jndi/rmi://" + namingHost + ":" + namingPort + jndiPath);

  // Create the JMXCconnectorServer
  JMXConnector cntor = JMXConnectorFactory.newJMXConnector(url, null);

  // The connection environment map
  Map environment = new HashMap();

  // create a credential
  UserPasswordCredential gatewayClientCred =
   new UserPasswordCredential("admin", "admin1");

  environment.put(JMXConnector.CREDENTIALS, gatewayClientCred);

  // Connect and invoke an operation on the remote MBeanServer
  try {
    cntor.connect(environment);
  }
  catch (SecurityException x) {
    // Uh-oh ! Bad credentials !
    throw x;
  }

  // Obtain a stub for the remote MBeanServer
  mbsc = cntor.getMBeanServerConnection();

  Iterator it = mbsc.queryMBeans(
  new ObjectName("ManagementServer:type=ObjectGrid,S=server1"),
  null).iterator();
  ObjectInstance oi = (ObjectInstance) it.next();
  server1MBean = oi.getObjectName();

  boolean status = ((Boolean) mbsc.invoke(
  server1MBean,
  "retrieveServerStatus",
  new Object[] {},
  new String[] {})).booleanValue();
  return status;
}
```

In this code snippet, a gatewayClientCred object is created and put in the environment. This environment is then used to connect to the gateway server.

If you want use SSL to connect from the gateway client to the gateway server, you have to use system properties to store the trust store and the trust store password. For example, you can pass in the following properties when you start a gateway client.

- `-Djavax.net.ssl.trustStore=etc/test/security/client.public`
- `-Djavax.net.ssl.trustStorePassword=public`

See the MX4J - Open Source Java Management Extensions Web site for more information.

## Security integration with WebSphere Application Server

ObjectGrid provides several security features to integrate with the WebSphere Application Server security infrastructure.

### Distributed ObjectGrid security integration with WebSphere Application Server

For the distributed ObjectGrid model, the security integration can be done by using the following classes:

- `com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredentialGenerator.`
- `com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenAuthenticator`
- `com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredential`

  .

These classes are discussed in “Client server security” on page 136. Following is an example of how to use the WSTokenCredentialGenerator class.

```
/**
* connect to the ObjectGrid Server.
*/
protected ClientClusterContext connect() throws ConnectException {
  ClientSecurityConfiguration csConfig = ClientSecurityConfigurationFactory
  .getClientSecurityConfiguration(proFile);
  CredentialGenerator gen = getWSCredGen();

  csConfig.setCredentialGenerator(gen);

  return objectGridManager.connect(csConfig, null);
}

/**
* Get a WSTokenCredentialGenerator
*
private CredentialGenerator getWSCredGen() {
WSTokenCredentialGenerator gen = new WSTokenCredentialGenerator(
WSTokenCredentialGenerator.RUN_AS_SUBJECT);
return gen;
}
```

On the server side, WSTokenAuthentication can be used as the authenticator to authenticate the WSTokenCredential object.

### Local ObjectGrid security integration with WebSphere Application Server

For the local ObjectGrid model, the security integration can be done by using the following two classes:

- `com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectSourceImpl`
- `com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectValidationImpl`

For more information about these classes, see “Local ObjectGrid security” on page 155. You can configure the WSSubjectSourceImpl class as the SubjectSource plug-in, and the WSSubjectValidationImpl class as the SubjectValidation plug-in.

## Listeners

ObjectGrid provides two listener-type interfaces that you can extend. The extensions can advise you through the extension interface and describe operations that are run on an ObjectGrid instance or a map instance.

### ObjectGridEventListener interface

Use the ObjectGridEventListener interface to receive notifications when significant events occur on an ObjectGrid. These events include ObjectGrid initialization, beginning of a transaction, ending a transaction, and destroying an ObjectGrid. To listen for these events, create a class that implements the ObjectGridEventListener interface and add it to the ObjectGrid.

**The ObjectGridEventListener interface**

The ObjectGridEventListener interface has the following methods. These methods are called when certain significant events occur on the ObjectGrid.

```
/**
* This method is invoked when the ObjectGrid itself is initialized.
* A usable Session instance is passed into this Listener to allow the
* optional replaying of a received LogSequence into a Map.
*
* @param session The Session instance that this Listener is associated with.
*/
void initialize(Session session);
```

```
/**
* This event signals the beginning of a transaction (session).
* A stringified version of the TxID is provided for
* correlating with the end of the transaction
* (session), if you want to use this version. The type of
* transaction (session) is
* also provided via the isWriteThroughEnabled boolean parameter.
*
* @param txid Stringified version of the TxID
* @param isWriteThroughEnabled Boolean flag indicating whether the
* Session was started via beginNoWriteThrough
*/
void transactionBegin(String txid, boolean isWriteThroughEnabled);
/**
* This signals the ending of a transaction (session). A stringified
* version of the TxID is provided for correlating with the
* begin of the transaction
* (session), if so desired. Changes are also reported. Typical uses of
* this event are for custom peer invalidation
* or peer commit push. This event
* listener outputs the changes. Calls to this method are made
* after commit and are sequenced so that they are delivered one by one,
* not in parallel. The event order is the commit order.
*
* @param txid Stringified version of the TxID
* @param isWriteThroughEnabled a boolean flag indicating
* whether the Sesison was
* started via beginNoWriteThrough
* @param committed a boolean flag indicating whether the Session
* was committed
* (true) or rolled back (false)
* @param changes A Collection of LogSequences that have been
* processed for the current Session.
*/
void transactionEnd(String txid, boolean isWriteThroughEnabled,
boolean committed, Collection /*/* <LogSequence> */*/ changes);
/**
* This method will be invoked when the ObjectGrid is destroyed. It's the
* opposite of initialize. When this method is called, the
* ObjectGridEventListener can free up any resource it uses.
*/
void destroy();
```

### Add and remove ObjectGridEventListeners objects

An ObjectGrid can have multiple ObjectGridEventListeners. Two methods exist on the ObjectGrid that allow ObjectGridEventListeners to be added. ObjectGridEventListeners that have been added can also be removed from an ObjectGrid.

The addEventListener method can be used to add an ObjectGridEventListener to an ObjectGrid.

```
/**
* Add an EventListener to the Session. Significant events
* will be communicated to interested listeners via this callback.
* Multiple event listeners are allowed to be registered, with no
* implied ordering of event notifications.
*
* Note, this method is allowed to be invoked before and after the
* {@link ObjectGrid#initialize()} method.
*
* @param cb An instance of ObjectGridEventListener
*/
void addEventListener(ObjectGridEventListener cb);
```

To add a list of ObjectGridEventListeners, use the setEventListeners method:

```
/**
* This overwrites the current list of callbacks and replaces it with the
* supplied list of callbacks.
*
* Note, this method is allowed to be invoked before and after the
* {@link ObjectGrid#initialize()} method.
* @param callbacks
*/
void setEventListeners(List callbacks);
```

To remove an ObjectGridEventListener from an ObjectGrid use the removeEventListener method:

```
/**
* Removes an EventListener from the Session. If the desired EventListener
*is not found on the Session, no error will be returned.
*
* Note, this method is allowed to be invoked before and after the
* {@link ObjectGrid#initialize()} method.
* @param cb An instance of ObjectGridEventListener
*/
void removeEventListener(ObjectGridEventListener cb);
```

**Create a custom ObjectGrid event listener**

To use a custom ObjectGrid event listener, first create a class that implements the com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener interface. Add the custom listener to an ObjectGrid to receive notification of significant events. An ObjectGridEventListener can be configured programmatically or with XML:

- **Programatically.** Assume that the class name of the ObjectGrid event listener is the com.company.org.MyObjectGridEventListener class. This class implements the ObjectGridEventListener interface. The following code snippet creates the custom ObjectGridEventListener and adds it to an ObjectGrid:

  ```
  ObjectGridManager objectGridManager =
   ObjectGridManagerFactory.getObjectGridManager();
  ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
  MyObjectGridEventListener myListener = new MyObjectGridEventListener();
  myGrid.addEventListener(myListener);
  ```

- **With XML.** An ObjectGridEventListner can also be configured using XML. The following XML creates a configuration that is equivalent to the described program-created ObjectGrid event listener. The following text must be in the `myGrid.xml` file:

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
    "http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
     <objectGrid name="myGrid">
      <bean id="ObjectGridEventListener"
         className="com.company.org.MyObjectGridEventListener" />
     </objectGrid>
    </objectGrids>
  </objectGridConfig>
  ```

  Provide this file to the ObjectGridManager to facilitate the creation of this configuration. The following code snippet demonstrates how to create an ObjectGrid using this XML file. The ObjectGrid that is created has an ObjectGridEventListener set on the myGrid ObjectGrid.

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid =
 objectGridManager.createObjectGrid("myGrid", new URL(
  "file:etc/test/myGrid.xml"), true, false);
```

**Watch for changes in a Map**

The transactionEnd method on the ObjectGridEventListener interface is very useful for applications that are interested in watching entries in the local Maps. An application can add one of these listeners and then use the transactionEnd method to see when the entries are changed. For example, if the ObjectGrid is working in distributed mode, an application can watch for incoming changes. Suppose that the replicated entries were for latest stock prices. This listener can watch for these changes arriving and update a second Map that keeps the value of a position in a portfolio. The listener must make all changes using the Session provided to the listener in the initialize method on the ObjectGridEventListener interface. The listener can distinguish between local changes and incoming remote changes usually by checking if the transaction is write through. The incoming changes from peer ObjectGrids are always write through.

## MapEventListener interface

Use the MapEventListener interface to receive significant events about a map. Events are sent to the MapEventListener when an entry is evicted from the map and when the preload of a map completes.

**MapEventListener interface**

The MapEventListener interface has the following methods. Implement the com.ibm.websphere.objectgrid.plugins.MapEventListener interface to create a custom MapEventListener.

```
/**
* This method is invoked when the specified entry is evicted from
* the map. The eviction could have occurred either by Evictor
* processing or by invoking one of the invalidate methods on the
* ObjectMap.
*
* @param key The key for the map entry that was evicted.
* @param value The value that was in the map entry evicted. The value
* object should not be modified.
*
*/
void entryEvicted(Object key, Object value);
/**
* This method is invoked when preload of this map has completed.
*
* @param t A Throwable object that indicates if preload completed without
* any Throwable occurring during the preload of the map. A null reference
* indicates preload completed without any Throwable objects occurring
* during the preload of the map.
*/
void preloadCompleted( Throwable t );
```

**Add and remove MapEventListeners**

The following BackingMap methods allow MapEventListeners to be added to and removed from a map:

```
/**
* Adds a MapEventListener to this BackingMap.
*
* Note, this method is allowed to be invoked before and after the
* ObjectGrid.initialize() method.
* @param eventListener A non-null reference to a MapEventListener to add
* to the list.
*
* @throws IllegalArgumentException if eventListener is null.
*
* @see MapEventListener
*/
public void addMapEventListener(MapEventListener
 eventListener );
/**
* Sets the list of MapEventListener objects.
*
* If this BackingMap already has a List of
* MapEventListeners, that list is replaced by the
* List passed as an argument to the current invocation
* of this method. This method can be called before and
* after the ObjectGrid.initialize() method.
*
* @param eventListenerList A non-null reference to a List of
* MapEventListener objects.
*
* @throws IllegalArgumentException is thrown if
* eventListenerList is null
* or the eventListenerList contains either a null
* reference or an object that is not an instance of
* MapEventListener.
*
* @see MapEventListener
*/
public void setMapEventListeners( List /*MapEventListener*/
 eventListenerList );
/**
* Removes a MapEventListener from this BackingMap.
*
* Note, this method is allowed to be invoked before and after the
* ObjectGrid.initialize() method.
*
* @param eventListener A non-null reference to an event listener
* that was previously added by invoking either the
* addMapEventListener(MapEventListener) or
* setMapEventListeners(List) method of this interface.
*
* @throws IllegalArgumentException if eventListener is null.
*
* @see MapEventListener
*/
public void removeMapEventListener(MapEventListener eventListener );
```

## Create a MapEventListener

To create a custom MapEventListener, implement the com.ibm.websphere.objectgrid.plugins.MapEventListener interface. To use the MapEventListener, add it to a BackingMap. A MapEventListener can be created and configured programmatically or with XML:

- **Programmatically**. The class name for the custom MapEventListener is the com.company.org.MyMapEventListener class. This class implements the MapEventListener interface. The following code snippet creates the custom MapEventListener and adds it to a BackingMap:

  ```
  ObjectGridManager objectGridManager =
   ObjectGridManagerFactory.getObjectGridManager();
  ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
  BackingMap myMap = myGrid.defineMap("myMap");
  MyMapEventListener myListener = new MyMapEventListener();
  myMap.addMapEventListener(myListener);
  ```

- **XML creation**. A MapEventListner can also be configured using XML. The following XML achieves a configuration that is equivalent to the preceding programmatic creation. The following XML must be in the `myGrid.xml` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config
  ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
   <objectGrids>
    <objectGrid name="myGrid">
     <backingMap name="myMap" pluginCollectionRef="myPlugins" />
    </objectGrid>
   </objectGrids>
   <backingMapPluginCollections>
    <backingMapPluginCollection id="myPlugins">
     <bean id="MapEventListener"
        classname="com.company.org.MyMapEventListener" />
    </backingMapPluginCollection>
   </backingMapPluginCollection>
</objectGridConfig>
```

  Providing this file to the ObjectGridManager facilitates the creation of this configuration. The following code snippet shows how to create an ObjectGrid using this XML file. The newly created ObjectGrid has a MapEventListener set on the myMap BackingMap.

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid =
 objectGridManager.createObjectGrid("myGrid", new URL(
 "file:etc/test/myGrid.xml"), true, false);
```

## Evictors

ObjectGrid provides a default evictor mechanism. You can also provide a pluggable evictor mechanism.

An *evictor* controls the membership of entries in each BackingMap. The default evictor uses a *time to live* eviction policy for each BackingMap. If you provide a pluggable evictor mechanism, it typically uses an eviction policy that is based on the number of entries instead of on time. This topic describes both types of evictors.

### Default time to live evictor

ObjectGrid provides a time to live (TTL) evictor for every BackingMap. The TTL evictor maintains an expiration time for each entry that is created. When the expiration time for an entry comes, the evictor removes the entry from the BackingMap. To minimize performance impact, the TTL evictor might wait to evict an entry after the expiration time, but never before the entry expires.

The BackingMap has attributes that are used to control how the time to live evictor computes the expiration time for each entry. Applications set the ttlType attribute to specify how the TTL evictor should calculate the expiration time. The ttlType attribute can be set to one of the following values:

- **None** indicates that an entry in the BackingMap never expires. The TTL evictor does not evict these entries.
- **Creation time** indicates that the time an entry is created is used in the expiration time calculation.

- **Last access time** indicates that the time that an entry has been last accessed is used in the expiration time calculation.

If the ttlType attribute is not set on a BackingMap, the default type of **None** is used so that the TTL evictor does not evict any entries. If the ttlType attribute is set to either **creation time** or **last access time**, the value of the time to live attribute on the BackingMap is added to either the creation time or last access time to compute the expiration time. The time precision of the time to live map attribute is in seconds. A value of `0` for the time to live attribute is a special value that is used to indicate that the map entry can live forever, that is, the entry stays in the map until the application explicitly removes or invalidates the map entry.

**Specify attributes for TTL evictors**

TTL evictors are associated with BackingMap instances. The following snippet of code demonstrates how the BackingMap interface can be used to set the needed attributes so that when each entry is created, it has an expiration time set to ten minutes after it was created.

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.TTLType;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid" );
BackingMap bm = og.defineMap( "myMap" );
bm.setTtlEvictorType( TTLType.CREATION_TIME );
bm.setTimeToLive( 600 );
```

The setTimeToLive method argument is `600` because it indicates the time to live value is in seconds. The preceding code must run before the initialize method is invoked on the ObjectGrid instance. These BackingMap attributes cannot be changed after the ObjectGrid instance is initialized. After the code runs, any entry that is inserted into the myMap BackingMap has an expiration time. After the expiration time is reached, the TTL evictor purges the entry.

If an application requires that the expiration time be set to the last access time plus ten minutes, one line of the preceding code must be changed. The argument that is passed to the setTtlEvictorType method is changed from `TTLType.CREATION_TIME` to `TTLType.LAST_ACCESS_TIME`. With this value, the expiration time is computed as the last access time plus 10 minutes. When an entry is first created, the last access time is the creation time.

When `TTLType.LAST_ACCESS_TIME` is used, the ObjectMap and JavaMap interfaces can be used to override the BackingMap time to live value. This mechanism allows an application to use a different time to live value for each entry that is created. Assume the preceding snippet of code was used to set the ttlType attribute to `LAST_ACCESS_TIME` and the time to live value was set to ten minutes on the BackingMap. An application can then override the time to live value for each entry by running the following code prior to creating or modifying an entry:

```
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.ObjectMap;
Session session = og.getSession();
ObjectMap om = session.getMap( "myMap" );
int oldTimeToLive1 = om.setTimeToLive( 1800 );
om.insert("key1", "value1" );
int oldTimeToLive2 = om.setTimeToLive( 1200 );
om.insert("key2", "value2" );
```

In the previous snippet of code, the entry with the key1 key has an expiration time of the insert time plus 30 minutes as a result of the setTimeToLive( 1800 ) method invocation on the ObjectMap. The oldTimeToLive1 variable is set to 600 because the time to live value from the BackingMap is used as a default value if the setTimeToLive method was not previously called on the ObjectMap.

The entry with the key2 key has an expiration time of insert time plus 20 minutes as a result of the setTimeToLive( 1200 ) method call on the ObjectMap. The oldTimeToLive2 variable is set to 1800 because the time to live value from the previous ObjectMap.setTimeToLive method invocation set the time to live to 1800.

The previous example shows two map entries being inserted in the myMap map for key values key1 and key2. At a later point in time the application from a new thread might want to update these map entries with new map values. However, the application wants to retain the time-to-live values that are used at insert time for each map entry. The following example illustrates how to retain the time-to-live values by using a constant defined in the ObjectMap interface for this very purpose:

```
Session session = og.getSession();
ObjectMap om = session.getMap( "myMap" );
om.setTimeToLive( ObjectMap.USE_DEFAULT );
session.begin();
om.update("key1", "updated value1" );
om.update("key2", "updated value2" );
om.insert("key3", "value3" );
session.commit();
```

Because the ObjectMap.USE_DEFAULT special value is used on the setTimeToLive method call, key1 retains its time-to-live value of 1800 seconds and key2 retains its time-to-live value of 1200 seconds because those values were used when these map entries were inserted by the prior transaction.

The previous example also shows a new map entry for key3 being inserted. In this case, the USE_DEFAULT special value indicates to use the default setting of time-to-live value for this map. The default value is defined by the time-to-live BackingMap attribute. See “BackingMap attributes” on page 106 for information about how the time-to-live attribute is defined on the BackingMap.

See the API documentation for the setTimeToLive method on the ObjectMap and JavaMap interfaces. It warns you that an IllegalStateException exception results if the BackingMap.getTtlEvictorType() method returns anything other than the TTLType.LAST_ACCESS_TIME value. ObjectMap and JavaMap can only be used to override the time to live value when you are using the LAST_ACCESS_TIME TTL evictor type. This method cannot be used to override the time to live value when you are using the CREATION_TIME TTL evictor type or the NONE TTL evictor type.

**Use an XML file to specify attributes for the TTL evictor**

Instead of using the BackingMap interface to programmatically set the BackingMap attributes to be used by the TTL evictor, an XML file can be used to configure each BackingMap. The following code demonstrates how to set these attributes for three different BackingMaps:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
        xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
 <objectGrid name="grid1">
  <backingMap name="map1" ttlEvictorType="NONE" />
  <backingMap name="map2" ttlEvictorType="LAST_ACCESS_TIME" timeToLive="1800" />
  <backingMap name="map3" ttlEvictorType="CREATION_TIME" timeToLive="1200" />
 </objectgrid>
</objectGrids>
```

The preceding example shows that the map1 BackingMap uses a `NONE` TTL evictor type. The map2 BackingMap uses a `LAST_ACCESS_TIME` TTL evictor type and has a time to live value of 1800 seconds, or 30 minutes. The map3 BackingMap is defined to use a `CREATION_TIME` TTL evictor type and has a time to live value of 1200 seconds, or 20 minutes.

## Optional pluggable evictors

The default TTL evictor uses an eviction policy that is based on time, and the number of entries in the BackingMap has no affect on the expiration time of an entry. An optional pluggable evictor can be used to evict entries based on the number of entries that exist instead of based on time. The following optional pluggable evictors provide some commonly used algorithms for deciding which entries to evict when a BackingMap grows beyond some size limit.

- **LRUEvictor** is an evictor that uses a *least recently used* algorithm to decide which entries to evict when the BackingMap exceeds a maximum number of entries.
- **LFUEvictor** is an evictor that uses a *least frequently used* algorithm to decide which entries to evict when the BackingMap exceeds a maximum number of entries.

The BackingMap informs an evictor as entries are created, modified, or removed in a transaction. The BackingMap keeps track of these entries and chooses when to evict one or more entries from the BackingMap.

A BackingMap has no configuration information for a maximum size. Instead, evictor properties are set to control the evictor behavior. Both the LRUEvictor and the LFUEvictor have a maximum size property that is used to cause the evictor to begin to evict entries after the maximum size is exceeded. Like the TTL evictor, the LRU and LFU evictors might not immediately evict an entry when the maximum number of entries is reached to minimize impact on performance.

If the LRU or LFU eviction algorithm is not adequate for a particular application, you can write your own evictors to achieve the eviction strategy that you want.

### Specify a pluggable evictor

Because evictors are associated with BackingMaps, the BackingMap interface is used to specify the pluggable evictor to use. The following code snippet is an example of specifying a LRUEvictor evictor for the map1 BackingMap and a LFUEvictor evictor for the map2 BackingMap:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor;
import com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid" );
BackingMap bm = og.defineMap( "map1" );
LRUEvictor evictor = new LRUEvictor();
evictor.setMaxSize(1000);
evictor.setSleepTime( 15 );
evictor.setNumberOfLRUQueues( 53 );
bm.setEvictor(evictor);
bm = og.defineMap( "map2" );
LFUEvictor evictor2 = new LFUEvictor();
evictor2.setMaxSize(2000);
evictor2.setSleepTime( 15 );
evictor2.setNumberOfHeaps( 211 );
bm.setEvictor(evictor2);
```

The preceding snippet shows an LRUEvictor evictor being used for map1 BackingMap with a maximum number of entries of `1000`. The LFUEvictor evictor is used for the map2 BackingMap with a maximum number of entries of `2000`. Both the LRU and LFU evictors have a sleep time property that indicates how long the evictor sleeps before waking up and checking to see if any entries need to be evicted. The sleep time is specified in seconds. A value of 15 seconds is a good compromise between performance impact and preventing BackingMap from growing too large. The goal is to use the largest sleep time possible without causing the BackingMap to grow to an excessive size.

The setNumberOfLRUQueues method sets the LRUEvictor property that indicates how many LRU queues the evictor uses to manage LRU information. A collection of queues is used so that every entry does not keep LRU information in the same queue. This approach can improve performance by minimizing the number of map entries that need to synchronize on the same queue object. Increasing the number of queues is a good way to minimize the impact that the LRU evictor can cause on performance. A good starting point is to use ten percent of the maximum number of entries as the number of queues. Using a prime number is typically better than using a number that is not prime.

The setNumberOfHeaps method sets the LFUEvictor property to set how many binary heap objects the LFUEvictor uses to manage LFU information. Again, a collection is used to improve performance. Using ten percent of the maximum number of entries is a good starting point and a prime number is typically better than using a number that is not prime.

**Use XML to specify a pluggable evictor**

Instead of using various APIs to programmatically plug in an evictor and set its properties, an XML file can be used to configure each BackingMap as illustrated in the following sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="grid">
   <backingMap name="map1" ttlEvictorType="NONE" pluginCollectionRef="LRU" />
   <backingMap name="map2" ttlEvictorType="NONE" pluginCollectionRef="LFU" />
  </objectGrid>
 </objectGrids>
 <backingMapPluginCollections>
  <backingMapPlugincollection id="LRU">
   <bean id="Evictor"
      className="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor">
    <property name="maxSize" type="int" value="1000"
         description="set max size for LRU evictor">
    <property name="sleepTime" type="int" value="15"
         description="evictor thread sleep time" />
    <property name="numberOfLRUQueues" type="int" value="53"
         description="set number of LRU queues" />
   </bean>
  </backingMapPluginCollection>
  <backingMapPluginCollection id="LFU">
   <bean id="Evictor"
      className="com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor">
    <property name="maxSize" type="int" value="2000"
         description="set max size for LFU evictor">
    <property name="sleepTime" type="int" value="15"
         description="evictor thread sleep time" />
    <property name="numberOfHeaps" type="int" value="211"
         description="set number of LFU heaps" />
   </bean>
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

## Write a custom evictor

ObjectGrid can be extended to use any eviction algorithm. You must create a custom evictor that implements the com.ibm.websphere.objectgrid.plugins.Evictor interface. The interface follows:

```
public interface Evictor
{
void initialize(BackingMap map, EvictionEventCallback callback);
void destroy();
void apply(LogSequence sequence);
}
```

- The initialize method is invoked during initialization of the BackingMap object. This method initializes an Evictor plug-in with a reference to the BackingMap and a reference to an object that implements the com.ibm.websphere.objectgrid.plugins.EvictionEventCallback interface.
- The apply method is invoked when transactions that access one or more entries of the BackingMap are committed. The apply method is passed a reference to an object that implements the com.ibm.websphere.objectgrid.plugins.LogSequence interface. The LogSequence interface allows an Evictor plug-in to determine which BackingMap entries were created, modified, or removed by the transaction. An Evictor uses this information in deciding when and which entries to evict.
- The destroy method is invoked when the BackingMap is being destroyed. This method allows an Evictor to terminate any threads that it might have created.

The EvictionEventCallback interface has the following methods:

```
public interface EvictionEventCallback
{
 void evictMapEntries(List evictorDataList) throws ObjectGridException;
 void evictEntries(List keysToEvictList) throws ObjectGridException;
 void setEvictorData(Object key, Object data);
 Object getEvictorData(Object key);
}
```

The EvictionEventCallback methods are used by an Evictor plug-in to call back to the ObjectGrid framework as follows:

- The setEvictorData method is used by an evictor to request the framework that is used to store and associate some evictor object it creates with the entry indicated by the key argument. The data is evictor specific and is determined by the information the evictor needs to implement the algorithm it is using. For example, in a least frequently used algorithm, the evictor maintains a count in the evictor data object for tracking how many times the apply method is invoked with a LogElement that refers to an entry for a given key.

- The getEvictorData method is used by an evictor to retrieve the data it passed to the setEvictorData method during a prior apply method invocation. If evictor data for the specified key argument is not found, a special `KEY_NOT_FOUND` object that is defined on the EvictorCallback interface is returned.
- The evictMapEntries method is used by an evictor to request the eviction of one or more map entries. Each object in the evictorDataList parameter must implement the com.ibm.websphere.objectgrid.plugins.EvictorData interface. Also, the same EvictorData instance that is passed to the setEvictorData method must be in the evictor data list parameter of this method. The getKey method of the EvictorData interface is used to determine which map entry to evict. The map entry is evicted if the cache entry currently contains the exact same EvictorData instance that is in the evictor data list for this cache entry.
- The evictEntries method is used by an evictor to request eviction of one or more map entries. This method is used only if the object that is passed to the setEvictorData method does not implement the com.ibm.websphere.objectgrid.plugins.EvictorData interface.

ObjectGrid calls the apply method of the Evictor interface *after* a transaction completes. All transaction locks that were acquired by the completed transaction are no longer held. Potentially, multiple threads can call the apply method at the same time, and each thread can complete its own transaction. Because transaction locks are already released by the completed transaction, the apply method must provide its own synchronization to ensure the apply method is thread safe.

The reason to implement the EvictorData interface and use the evictMapEntries method instead of the evictEntries method is to close a potential timing window. Consider the following sequence of events:

1. Transaction 1 completes and calls the apply method with a LogSequence that deletes the map entry for key 1.
2. Transaction 2 completes and calls the apply method with a LogSequence that inserts a new map entry for key 1. In other words, transaction 2 recreates the map entry that was deleted by transaction 1.

Because the evictor runs asynchronously from threads that run transactions, it is possible that when the evictor decides to evict key 1, it might be evicting either the map entry that existed prior to transaction 1 completion, or it might be evicting the map entry that was recreated by transaction 2. To eliminate timing windows and to eliminate uncertainty as to which version of the key 1 map entry the evictor intended to evict, implement the EvictorData interface by the object that is passed to the setEvictorData method. Use the same EvictorData instance for the life of a map entry. When that map entry is deleted and is then recreated by another transaction, the evictor should use a new instance of the EvictorData implementation. By using the EvictorData implementation and by using the evictMapEntries method, the evictor can ensure that the map entry is evicted if and only if the cache entry that is associated with the map entry contains the correct EvictorData instance.

The Evictor and EvictonEventCallback interfaces allow an application to plug in an evictor that implements a user-defined algorithm for eviction. The following snippet of code illustrates how you can implement the initialize method of Evictor interface:

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.EvictionEventCallback;
import com.ibm.websphere.objectgrid.plugins.Evictor;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;
import java.util.LinkedList;
```

```
// Instance variables
private BackingMap bm;
private EvictionEventCallback evictorCallback;
private LinkedList queue;
private Thread evictorThread;
public void initialize(BackingMap map, EvictionEventCallback callback)
{
 bm = map;
 evictorCallback = callback;
 queue = new LinkedList();
 // spawn evictor thread
 evictorThread = new Thread( this );
 String threadName = "MyEvictorForMap-" + bm.getName();
 evictorThread.setName( threadName );
 evictorThread.start();
}
```

The preceding code saves the references to the map and callback objects in instance variables so that they are available to the apply and destroy methods. In this example, a linked list is created that is used as a *first in, first out* queue for implementing a least recently used (LRU) algorithm. A thread is spawned off and a reference to the thread is kept as an instance variable. By keeping this reference, the destroy method can interrupt and terminate the spawned thread.

Ignoring synchronization requirements to make code thread safe, the following snippet of code illustrates how the apply method of the Evictor interface can be implemented:

```
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.EvictionEventCallback;
import com.ibm.websphere.objectgrid.plugins.Evictor;
import com.ibm.websphere.objectgrid.plugins.EvictorData;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;
public void apply(LogSequence sequence)
{
 Iterator iter = sequence.getAllChanges();
 while ( iter.hasNext() )
 {
  LogElement elem = (LogElement)iter.next();
  Object key = elem.getCacheEntry().getKey();
  LogElement.Type type = elem.getType();
  if ( type == LogElement.INSERT )
  {
   // do insert processing here by adding to front of LRU queue.
   EvictorData data = new EvictorData(key);
   evictorCallback.setEvictorData(key, data);
   queue.addFirst( data );
  }
  else if ( type == LogElement.UPDATE || type == LogElement.FETCH ||
   type == LogElement.TOUCH )
  {
   // do update processing here by moving EvictorData object to
   // front of queue.
   EvictorData data = evictorCallback.getEvictorData(key);
   queue.remove(data);
   queue.addFirst(data);
  }
  else if ( type == LogElement.DELETE || type == LogElement.EVICT )
  {
   // do remove processing here by removing EvictorData object
   // from queue.
   EvictorData data = evictorCallback.getEvictorData(key);
   if ( data == EvictionEventCallback.KEY_NOT_FOUND )
   {
    // Assumption here is your asynchronous evictor thread
    // evicted the map entry before this thread had a chance
    // to process the LogElement request. So you probably
    // need to do nothing when this occurs.
   }
   else
   {
    // Key was found. So process the evictor data.
    if ( data != null )
    {
     // Ignore null returned by remove method since spawned
     // evictor thread may have already removed it from queue.
     // But we need this code in case it was not the evictor
     // thread that caused this LogElement to occur.
     queue.remove( data );
    }
    else
    {
     // Depending on how you write you Evictor, this possibility
     // may not exist or it may indicate a defect in your evictor
     // due to improper thread synchronization logic.
    }
   }
  }
 }
}
```

Insert processing in the apply method typically handles the creation of an evictor data object that is passed to the setEvictorData method of the EvictionEventCallback interface. Because this evictor illustrates a LRU implementation, the EvictorData is also added to the front of the queue that was created by the initialize method. Update processing in the apply method typically updates the evictor data object that was created by some prior invocation of the apply method (for example, by the insert processing of the apply method). Because this evictor is an LRU implementation, it needs to move the EvictorData object from its current queue position to the front of the queue. The spawned evictor thread removes the last EvictorData object in the queue because the last queue element represents the least recently used entry. The assumption is that the EvictorData object has a getKey method on it so that the evictor thread knows the keys of the entries that need to be evicted. Keep in mind that this example is ignoring synchronization requirements to make code thread safe. A real custom evictor is more complicated because it deals with synchronization and performance bottlenecks that occur as a result of the synchronization points.

The following snippets of code illustrate the destroy method and the run method of the runnable thread that the initialize method spawned:

```
// Destroy method simply interrupts the thread spawned by the initialize method.
public void destroy()
{
 evictorThread.interrupt();
}

// Here is the run method of the thread that was spawned by the initialize method.
public void run()
{
 // Loop until destroy method interrupts this thread.
 boolean continueToRun = true;
 while ( continueToRun )
 {
  try
  {
   // Sleep for a while before sweeping over queue.
   // The sleepTime is a good candidate for a evictor
   // property to be set.
   Thread.sleep( sleepTime );
   int queueSize = queue.size();
   // Evict entries if queue size has grown beyond the
   // maximum size. Obviously, maximum size would
   // be another evictor property.
   int numToEvict = queueSize - maxSize;
   if ( numToEvict > 0 )
   {
    // Remove from tail of queue since the tail is the
    // least recently used entry.
    List evictList = new ArrayList( numToEvict );
    while( queueSize > ivMaxSize )
    {
     EvictorData data = null;
     try
     {
      EvictorData data = (EvictorData) queue.removeLast();
      evictList.add( data );
      queueSize = queue.size();
     }
     catch ( NoSuchElementException nse )
     {
      // The queue is empty.
      queueSize = 0;
     }
    }
    // Request eviction if key list is not empty.
    if ( ! evictList.isEmpty() )
    {
     evictorCallback.evictMapEntries( evictList );
    }
   }
  }
  catch ( InterruptedException e )
  {
   continueToRun = false;
  }
 } // end while loop
} // end run method.
```

### Optional RollBackEvictor interface

The com.ibm.websphere.objectgrid.plugins.RollbackEvictor interface can be optionally implemented by an Evictor plug-in. By implementing this interface, an evictor can be invoked not only when transactions are committed, but also when transactions are rolled back.

```
public interface RollbackEvictor
{
    void rollingBack( LogSequence ls );
}
```

The apply method is called only if a transaction is committed. If a transaction is rolled back and the RollbackEvictor interface is implemented by the evictor, the rollingBack method is invoked. If the RollbackEvictor interface is not implemented and the transaction rolls back, the apply method and the rollingBack method are not called.

## Loaders

An ObjectGrid loader is a pluggable component that allows an ObjectGrid map to behave as a memory cache for data that is typically kept in a persistent store on either the same system or some other system.

Typically, a database or file system is used as the persistent store. A remote Java virtual machine (JVM) can also be used as the source of data allowing hub based caches to be built using ObjectGrid. A loader has the logic for reading and writing data from and to a persistent store.

A Loader is a plug-in for an ObjectGrid backing map. Only one Loader can ever be associated with a given backing map. Each backing map has its own Loader instance. The backing map requests any data that it does not contain from its loader. Any changes to the map are pushed out to the loader. The loader plug-in allows the backing map to move data between the map and its persistent store.

## Plug in a loader

The following snippet of code illustrates how an application-provided Loader is plugged into the backing map for map1 using the ObjectGrid API:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid" );
BackingMap bm = og.defineMap( "map1" );
MyLoader loader = new MyLoader();
loader.setDataBaseName("testdb");
loader.setIsolationLevel("read committed");
bm.setLoader( loader );
```

The assumption is that `MyLoader` is the application-provided class that implements the com.ibm.websphere.objectgrid.plugins.Loader interface. Because the association of a Loader with a backing map cannot be changed after ObjectGrid is initialized, the code must be run before invoking the `initialize` method of the ObjectGrid interface that is being called. An `IllegalStateException` exception occurs on a `setLoader` method call if it is called after initialization has occurred.

The application-provided Loader can have set properties. In the example, the MyLoader loader is used to read and write data from a table in a relational database. The loader must have the name of the database and the SQL isolation level to use. The MyLoader loader has the `setDataBaseName` and `setIsolationLevel` methods that allow the application to set these two Loader properties.

An application provided Loader could also be plugged in by using an XML file. The following example illustrates how the MyLoader loader is plugged into the map1 backing map with the same database name and isolation level Loader properties being set:

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="grid">
   <backingMap name="map1" pluginCollectionRef="map1" lockStrategy="OPTIMISTIC" />
  </objectGrid>
 </objectGrids>
 <backingMapPluginCollections>
  <backingMapPluginCollection id="map1">
   <bean id="Loader" className="com.myapplication.MyLoader">
    <property name="dataBaseName" type="java.lang.String" value="testdb"
     description="database name" />
    <property name="isolationLevel" type="java.lang.String"
     value="read committed" description="iso level" />
   </bean>
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

## Implement the Loader interface

An application provided Loader must implement the *com.ibm.websphere.objectgrid.plugins.Loader* interface. The Loader interface has the following definition:

```
public interface Loader
{
 static final SpecialValue KEY_NOT_FOUND;
 List get(TxID txid, List keyList, boolean forUpdate)
 throws LoaderException;
 void batchUpdate(TxID txid, LogSequence sequence)
 throws LoaderException, OptimisticCollisionException;
 void preloadMap(Session session, BackingMap backingMap)
 throws LoaderException;
}
```

Each of the following sections gives an explanation and considerations when implementing each of the methods on the Loader interface.

**get method**

The backing map calls the Loader `get` method to get the values associated with a key list that is passed as the **keyList** argument. The get method is required to return a java.lang.util.List list of values, one for each key that is in the key list. The first value returned in the value list corresponds to the first key in the key list, the second value returned in the value list corresponds to the second key in the key list, and so on. If the loader does not find the value for a key in the key list, the Loader is required to return the special `KEY_NOT_FOUND` value object that is defined in the Loader interface. Because a backing map can be configured to allow `null` as a valid value, it is very important for the Loader to return the special `KEY_NOT_FOUND` object when the Loader is unable to find the key. This value allows the backing map to distinguish between a `null` value and a value that does not exist because the key was not found. If a backing map does not support `null` values, a Loader that returns null instead of the `KEY_NOT_FOUND` object for a key that does not exist results in an exception.

The **forUpdate** argument tells the Loader if the application called a `get` method on the map or a `getForUpdate` method on the map. See the com.ibm.websphere.objectgrid.ObjectMap interface for more information. The Loader is responsible for implementing a concurrency control policy that controls concurrent access to the persistent store. For example, many relational database management systems support the `for update` syntax on the SQL select statement that is used to read data from a relational table. The Loader can choose to use the `for update` syntax on the SQL `select` statement based on whether boolean `true` is passed as the argument value for the **forUpdate** parameter of this method. Typically, the Loader uses the for update syntax only when using a pessimistic concurrency control policy. For an optimistic concurrency control, the Loader never uses for update syntax on the SQL select statement. The Loader is responsible to decide to use the `forUpdate` argument based on the concurrency control policy that is being used by the Loader.

For an explanation of the **txid** parameter, see the “TransactionCallback plug-in” on page 207 topic.

## batchUpdate method

The batchUpdate method is critical on the Loader interface. This method is called whenever the ObjectGrid needs to apply all current changes to the Loader. The Loader is given a list of changes for this Map. The changes are iterated and applied to the backend. The method receives the current TxID value and the changes to apply. The following sample iterates over the set of changes and batches three Java database connectivity (JDBC) statements, one with insert, another with update, and one with delete.

```
import java.util.Collection;
import java.util.Map;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
import com.ibm.websphere.objectgrid.plugins.Loader;
import com.ibm.websphere.objectgrid.plugins.LoaderException;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;
public void batchUpdate(TxID tx, LogSequence sequence)
throws LoaderException
{
 // Get a SQL connection to use.
 Connection conn = getConnection(tx);
 try
 {
  // Process the list of changes and build a set of prepared
  // statements for executing a batch update, insert, or delete
  // SQL operation.
  Iterator iter = sequence.getPendingChanges();
  while ( iter.hasNext() )
  {
   LogElement logElement = (LogElement)iter.next();
   Object key = logElement.getCacheEntry().getKey();
   Object value = logElement.getCurrentValue();
   switch ( logElement.getType().getCode() )
   {
    case LogElement.CODE_INSERT:
    buildBatchSQLInsert( tx, key, value, conn );
    break;
    case LogElement.CODE_UPDATE:
    buildBatchSQLUpdate( tx, key, value, conn );
    break;
    case LogElement.CODE_DELETE:
    buildBatchSQLDelete( tx, key, conn );
    break;
   }
  }
  // Execute the batch statements that were built by above loop.
  Collection statements = getPreparedStatementCollection( tx, conn );
  iter = statements.iterator();
  while ( iter.hasNext() )
  {
   PreparedStatement pstmt = (PreparedStatement) iter.next();
   pstmt.executeBatch();
  }
 }
 catch (SQLException e)
 {
  LoaderException ex = new LoaderException(e);
  throw ex;
 }
}
```

The preceding sample illustrates the high level logic of processing the LogSequence argument, but the details of how a SQL `insert`, `update`, or `delete` statement is built are not illustrated. Some of the key points that are illustrated include:

- The `getPendingChanges` method is called on the LogSequence argument to obtain an iterator over the list of LogElements that the Loader needs to process.
- The `LogElement.getType().getCode()` method is used to determine if the LogElement is for a SQL `insert`, `update`, or `delete` operation.
- An `SQLException` exception is caught and is chained to a `LoaderException` exception that prints to report that an exception occurred during the batch update.
- JDBC batch update support is used to minimize the number of queries to the backend that must be made.

**`preloadMap` method**

During the ObjectGrid initialization, each backing map that is defined is initialized. If a Loader is plugged into a backing map, the backing map invokes the `preloadMap` method on the Loader interface to allow the loader to pre-fetch data from its backend and load the data into the map. The following sample assumes the first 100 rows of an Employee table is read from the database and is loaded into the map. The EmployeeRecord class is an application provided class that holds the employee data read from the employee table.

```
import java.util.Map;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.TxID;
import com.ibm.websphere.objectgrid.plugins.Loader;
import com.ibm.websphere.objectgrid.plugins.LoaderException;
public void preloadMap(Session session, BackingMap backingMap)
throws LoaderException
{
 boolean tranActive = false;
 ResultSet results = null;
 Statement stmt = null;
 Connection conn = null;
 try
 {
  session.beginNoWriteThrough();
  tranActive = true;
  ObjectMap map = session.getMap( backingMap.getName() );
  TxID tx = session.getTxID();
  // Get a auto-commit connection to use that is set to
  // a read committed isolation level.
  conn = getAutoCommitConnection(tx);
  // Preload the Employee Map with EmployeeRecord
  // objects. Read all Employees from table, but
  // limit preload to first 100 rows.
  stmt = conn.createStatement();
  results = stmt.executeQuery( SELECT_ALL );
  int rows = 0;
  while ( results.next() && rows < 100 )
  {
   int key = results.getInt(EMPNO_INDEX);
   EmployeeRecord emp = new EmployeeRecord( key );
   emp.setLastName( results.getString(LASTNAME_INDEX) );
   emp.setFirstName( results.getString(FIRSTNAME_INDEX) );
   emp.setDepartmentName( results.getString(DEPTNAME_INDEX) );
   emp.updateSequenceNumber( results.getLong(SEQNO_INDEX) );
```

```
      emp.setManagerNumber( results.getInt(MGRNO_INDEX) );
      map.put( new Integer(key), emp );
      ++rows;
     }
     // Commit the transaction.
     session.commit();
     tranActive = false;
    }
    catch (Throwable t)
    {
     throw new LoaderException("preload failure: " + t, t);
    }
    finally
    {
     if ( tranActive )
     {
      try
      {
       session.rollback();
      }
      catch ( Throwable t2 )
      {
       // Tolerate any rollback failures and
       // allow original Throwable to be thrown.
      }
     }
     // Be sure to clean up other databases resources here
     // as well such a closing statements, result sets, etc.
    }
   }
```

This sample illustrates the following key points:

- The preloadMap backing map uses the Session object that is passed to it as the session argument.
- The `Session.beginNoWriteThrough()` method is used to begin the transaction rather than the `begin` method. The Loader cannot be called for each `put` operation that occurs in this method for loading the map.
- The Loader can map columns of employee table to a field in the EmployeeRecord java object.
- The Loader catches all throwable exceptions that occur and throws a `LoaderException` exception with the caught throwable exception chained to it.
- The `finally` block ensures that any throwable exception that occurs between the time the `beginNoWriteThrough` method is called and the commit `method` is called cause the `finally` block to roll back the active transaction. This action is critical to ensure that any transaction that has been started by the `preloadMap` method is completed before returning to the caller. The `finally` block is a good place to perform other clean up actions that might be needed, like closing the JDBC connection and other JDBC objects.

The `preloadMap` sample is using a SQL select statement that selects all rows of the table. In your application provided Loader, you might need to set one or more Loader properties to control how much of the table needs to be preloaded into the map.

Because the `preloadMap` method is only called one time during the BackingMap initialization, it is also a good place to run the one time Loader initialization code. Even if a Loader chooses not to pre-fetch data from the backend and load the data into the map, it probably needs to perform some other one time initialization to make other methods of the Loader more efficient. The following is an example of caching the TransactionCallback object and OptimisticCallback object as instance variables of the Loader so that the other methods of the Loader do not have to make method calls to get access to these objects. This caching of the ObjectGrid plug-in values can be done because after the BackingMap is initialized, the TransactionCallback and the OptimisticCallback objects cannot be changed or replaced. It is acceptable to cache these object references as instance variables of the Loader.

```
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.OptimisticCallback;
import com.ibm.websphere.objectgrid.plugins.TransactionCallback;

// Loader instance variables.
MyTransactionCallback ivTcb; // MyTransactionCallback

// extends TransactionCallback
MyOptimisticCallback ivOcb; // MyOptimisticCallback
// implements OptimisticCallback
...
public void preloadMap(Session session, BackingMap backingMap)
 throws LoaderException
{
 // Cache TransactionCallback and OptimisticCallback objects
 // in instance variables of this Loader.
 ivTcb = (MyTransactionCallback)
 session.getObjectGrid().getTransactionCallback();
 ivOcb = (MyOptimisticCallback) backingMap.getOptimisticCallback();
 // The remainder of preloadMap code (such as shown in prior example).
}
```

For information on preloading and recoverable preloading as it pertains to replication failover, see “Replication programming” on page 217.

## Loader considerations

Use the following considerations when implementing a loader.

### Preload considerations

Each backing map has a boolean preloadMode attribute that can be set to indicate if preload of a map completes asynchronously. By default, the preloadMode attribute is set to `false`, which indicates that the backing map initialization does not complete until the preload of the map is complete. For example, backing map initialization is not complete until the `preloadMap` method returns. If the `preloadMap` method is going to read a large amount of data from its back end and load it into the map, it might take a relatively long time to complete. In this case, you can configure a backing map to use asynchronous preload of the map by setting the preloadMode attribute to true. This setting causes the backing map initialization code to spawn a thread that invokes the `preloadMap` method, allowing initialization of a backing map to complete while the preload of the map is still in progress.

The following snippet of code illustrates how the preloadMode attribute is set to enable asynchronous preload:

```
BackingMap bm = og.defineMap( "map1" );
bm.setPreloadMode( true );
```

The preloadMode attribute can also be set by using a XML file as illustrated in the following example:

```
<backingMap name="map1" preloadMode="true"
 pluginCollectionRef="map1" lockStrategy="OPTIMISTIC" />
```

## TxID and use of the TransactionCallback interface

Both the `get` method and `batchUpdate` methods on the Loader interface are passed a TxID object that represents the Session transaction that requires the `get` or `batchUpdate` operation to be performed. It is possible that the `get` and `batchUpdate` methods are called more than once per transaction. Therefore, transaction-scoped objects that are needed by the Loader are typically kept in a slot of the TxID object. A Java database connectivity (JDBC) Loader is used to illustrate how a Loader uses the TxID and TransactionCallback interfaces.

It is also possible that several ObjectGrid maps are stored in the same database. Each map has its own Loader and each Loader might need to connect to the same database. When connecting to the same database, each Loader wants to use the same JDBC connection so that the changes to each table are committed as part of the same database transaction. Typically, the same person who writes the Loader implementation also writes the TransactionCallback implementation. The best method is if the TransactionCallback interface is extended to add methods that the Loader needs for getting a database connection and for caching prepared statements. The reason for this methodology becomes apparent as you look at how the TransactionCallback and TxID interfaces are used by the Loader.

As an example, the Loader might need the TransactionCallback interface to be extended as follows:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
public interface MyTransactionCallback extends TransactionCallback
{
 Connection getAutoCommitConnection(TxID tx, String databaseName)
 throws SQLException;
 Connection getConnection(TxID tx, String databaseName,
 int isolationLevel ) throws SQLException;
 PreparedStatement getPreparedStatement(TxID tx, Connection conn,
 String tableName, String sql) throws SQLException;
 Collection getPreparedStatementCollection( TxID tx, Connection conn,
 String tableName );
}
```

Using these new methods, the Loader `get` and `batchUpdate` methods can get a connection as follows:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
private Connection getConnection(TxID tx, int isolationLevel)
{
 Connection conn = ivTcb.getConnection(tx, databaseName, isolationLevel );
 return conn;
}
```

In the previous example and the examples that follow, *ivTcb* and *ivOcb* are Loader instance variables that were initialized as described in the “Preload considerations” on page 197 section. The *ivTcb* variable is a reference to the MyTransactionCallback instance and the *ivOcb* is a reference to the MyOptimisticCallback instance. The *databaseName* variable is an instance variable of the Loader that was set as a Loader property during the initialization of the backing map. The `isolationLevel` argument is one of the JDBC Connection constants that are defined for the various isolation levels that JDBC supports. If the Loader is using an optimistic implementation, the `get` method typically uses a JDBC auto–commit connection to fetch the data from the database. In that case, the Loader might have a `getAutoCommitConnection` method that is implemented as follows:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
private Connection getAutoCommitConnection(TxID tx)
{
 Connection conn = ivTcb.getAutoCommitConnection(tx, databaseName);
 return conn;
}
```

Recall that the `batchUpdate` method has the following switch statement:

```
switch ( logElement.getType().getCode() )
{
 case LogElement.CODE_INSERT:
 buildBatchSQLInsert( tx, key, value, conn );
 break;
 case LogElement.CODE_UPDATE:
 buildBatchSQLUpdate( tx, key, value, conn );
 break;
 case LogElement.CODE_DELETE:
 buildBatchSQLDelete( tx, key, conn );
 break;
}
```

Each of the `buildBatchSQL` methods uses the MyTransactionCallback interface to get a prepared statement. Following is a snippet of code that shows the `buildBatchSQLUpdate` method building an SQL update statement for updating an EmployeeRecord entry and adding it for the batch update:

```
private void buildBatchSQLUpdate( TxID tx, Object key, Object value, Connection
 conn )
throws SQLException, LoaderException
{
 String sql = "update EMPLOYEE set LASTNAME = ?, FIRSTNAME = ?, DEPTNO = ?,
 SEQNO = ?, MGRNO = ? where EMPNO = ?";
 PreparedStatement sqlUpdate = ivTcb.getPreparedStatement( tx, conn, "employee",
 sql );
 EmployeeRecord emp = (EmployeeRecord) value;
 sqlUpdate.setString(1, emp.getLastName());
 sqlUpdate.setString(2, emp.getFirstName());
 sqlUpdate.setString(3, emp.getDepartmentName());
 sqlUpdate.setLong(4, emp.getSequenceNumber());
 sqlUpdate.setInt(5, emp.getManagerNumber());
 sqlUpdate.setInt(6, key);
 sqlUpdate.addBatch();
}
```

After the `batchUpdate` loop has built all of the prepared statements, it calls the `getPreparedStatementCollection` method. This method can be implemented as follows:

```
private Collection getPreparedStatementCollection( TxID tx, Connection conn )
{
 return ( ivTcb.getPreparedStatementCollection( tx, conn, "employee" ) );
}
```

When the application invokes the `commit` method on the Session, the Session code calls the `commit` method on the `TransactionCallback` method after it has pushed all the changes made by the transaction out to the Loader for each map that was changed by the transaction. Because all of the Loaders used the `MyTransactionCallback` method to get any connection and prepared statements they needed, the `TransactionCallback` method knows which connection to use to request that the back end commits the changes. So, extending the TransactionCallback interface with methods that are needed by each of the Loaders has the following advantages:

- The TransactionCallback object encapsulates the use of TxID slots for transaction-scoped data, and the Loader does not require information about the TxID slots. The Loader only needs to know about the methods that are added to TransactionCallback using the MyTransactionCallback interface for the supporting functions needed by the Loader.
- The TransactionCallback object can ensure that connection sharing occurs between each Loader that connects to the same backend so that a two phase commit protocol can be avoided.
- The TransactionCallback object can ensure that connecting to the backend is driven to completion through a commit or rollback invoked on the connection when appropriate.
- TransactionCallback can ensure that the cleanup of database resources occurs when a transaction completes.
- TransactionCallback can hide if it is obtaining a managed connection from a managed environment such as WebSphere Application Server or some other Java 2 Platform, Enterprise Edition (J2EE) compliant application server. This advantage allows the same Loader code to be used in both a managed and unmanaged environments. Only the TransactionCallback plug-in must be changed.

For detailed information about how the TransactionCallback implementation uses the TxID slots for transaction-scoped data, see "TransactionCallback plug-in" on page 207.

## OptimisticCallback

As mentioned earlier, the Loader might decide to use an optimistic approach for concurrency control. If that is the case, the `buildBatchSQLUpdate` method example needs to be modified slightly for implementing an optimistic approach. Several possible ways exist for using an optimistic approach. A typical way is to have either a timestamp column or sequence number counter column for versioning each update of the row. Assume that the employee table has a sequence number column that increments each time the row is updated.

You then modify the signature of the `buildBatchSQLUpdate` method so that it is passed the LogElement object instead of the key and value pair. It also needs to use the OptimisticCallback object that is plugged into the backing map for getting both the initial version object and for updating the version object. The following is an example of a modified `buildBatchSQLUpdate` method that uses the *ivOcb* instance variable that was initialized as described in the preloadMap section:

```
private void buildBatchSQLUpdate( TxID tx, LogElement le,
 Connection conn )throws SQLException, LoaderException
{
 // Get the initial version object when this map entry was last read
 // or updated in the database.
 Employee emp = (Employee) le.getCurrentValue();
 long initialVersion = ((Long) le.getVersionedValue()).longValue();
 // Get the version object from the updated Employee for the SQL update
 //operation.
 Long currentVersion = (Long)ivOcb.getVersionedObjectForValue( emp );
 long nextVersion = currentVersion.longValue();
 // Now build SQL update that includes the version object in where clause
 // for optimistic checking.
 String sql = "update EMPLOYEE set LASTNAME = ?, FIRSTNAME = ?,
 DEPTNO = ?,SEQNO = ?, MGRNO = ? where EMPNO = ? and SEQNO = ?";
 PreparedStatement sqlUpdate = ivTcb.getPreparedStatement( tx, conn,
 "employee", sql );
 sqlUpdate.setString(1, emp.getLastName());
 sqlUpdate.setString(2, emp.getFirstName());
 sqlUpdate.setString(3, emp.getDepartmentName());
 sqlUpdate.setLong(4, nextVersion );
 sqlUpdate.setInt(5, emp.getManagerNumber());
 sqlUpdate.setInt(6, key);
 sqlUpdate.setLong(7, initialVersion);
 sqlUpdate.addBatch();
}
```

The example shows that the LogElement is used to obtain the initial version value. When the transaction first accesses the map entry, a LogElement is created with the initial Employee object that is obtained from the map. The initial Employee object is also passed to the getVersionedObjectForValue method on the OptimisticCallback interface and the result is saved in the LogElement. This processing happens before an application is given a reference to the initial Employee object and has a chance to call some method that changes the state of the initial Employee object.

The example shows that the Loader uses the getVersiondObjectForValue method to obtain the version object for the current updated Employee object. Before calling the batchUpdate method on the Loader interface, ObjectGrid calls the updateVersionedObjectForValue method on the OptimisticCallback interface to cause a new version object to be generated for the updated Employee object. After the batchUpdate method returns to the ObjectGrid, the LogElement is updated with the current version object so it becomes the new initial version object. This step is necessary because the application might have called the flush method on the map instead of the commit method on the Session. It is possible for the Loader to be called multiple times by a single transaction for the same key. For that reason, ObjectGrid ensures that the LogElement is updated with the new version object each time the row is updated in the employee table.

Now that the Loader has both the initial version object and the next version object, it can run an SQL update statement that sets the SEQNO column to the next version object value and uses the initial version object value in the `where` clause. This approach is sometimes referred to as being an overqualified `update` statement. The use of the overqualified `update` statement allows the relational database to verify that the row was not changed by some other transaction in between the time that this transaction read the data from the database and the time that this transaction updates the database. If another transaction modified the row, then the count array that is returned by the batch update indicates that zero rows were updated for this key. The Loader is responsible for verifying that the SQL `update` operation did in fact update the row. If it does not, the Loader displays a `com.ibm.websphere.objectgrid.plugins.OptimisticCollisionException` exception to inform the Session that the batchUpdate method failed due to more than one concurrent transaction trying to update the same row in the database table. This exception causes the Session to roll back and the application must retry the entire transaction. The rationale is that the retry will be successful, which is why this approach is called optimistic. The optimistic approach does in fact perform better if data is infrequently changed or concurrent transactions rarely try to update the same row.

It is important for the Loader to use the *key* parameter of the OptimisticCollisionException constructor to identify which key or set of keys caused the optimistic batchUpdate method to fail. The key parameter can either be the key object itself or an array of key objects if more than one key resulted in optimistic update failure. ObjectGrid uses the getKey method of the OptimisticCollisionException constructor to determine which map entries contain stale data and caused the exception to result. Part of the rollback processing is to evict each stale map entry from the map. Evicting stale entries is necessary so that any subsequent transaction that accesses the same key or keys results in the get method of the Loader interface being called to refresh the map entries with the current data from the database.

Other ways for a Loader to implement an optimistic approach include:

- No timestamp or sequence number column exists. In this case, the getVersionObjectForValue method on the OptimisticCallback interface simply returns the value object itself as the version. With this approach, the Loader needs to build a `where` clause that includes each of the fields of the initial version object. This approach is not very efficient, and not all column types are eligible to be used in the `where` clause of an overqualified SQL update statement. This approach is typically not used.
- No timestamp or sequence number column exists. However, unlike the prior approach, the `where` clause only contains the value fields that were modified by the transaction. One way to detect which fields are modified is to set the copy mode on the backing map to be `CopyMode.COPY_ON_WRITE` mode. This copy mode requires that a value interface to be passed to the `setCopyMode` method on the BackingMap interface. The BackingMap creates dynamic proxy objects that implement the provided value interface. With this copy mode, the Loader can cast each value to a com.ibm.websphere.objectgrid.plugins.ValueProxyInfo object. The ValueProxyInfo interface has a method that allows the Loader to obtain the List of attribute names that were changed by the transaction. This method enables the Loader to call the `get` methods on the value interface for the attribute names to obtain the changed data and to build an SQL update statement that only sets the changed attributes. The `where` clause can now be built to have the primary key column plus each of the changed attribute columns. This approach is more efficient than the prior approach, but it requires more code to be written in the Loader and leads to the possibility that the prepared statement cache needs to be larger to handle the different permutations. However, if transactions typically only modify a few of the attributes, this limitation might not be a problem.
- Some relational databases might have an API to assist in automatically maintaining column data that is useful for optimistic versioning. Consult your database documentation to determine if this possibility exists.

## ObjectTransformer plug-in

Use the ObjectTransformer plug-in when you require high performance. If you see performance issues with CPU usage, add an ObjectTransformer plug-in to each map. If you do not provide an ObjectTransformer plug-in, up to 60-70% of the total CPU time is spent serializing and copying entries.

### Purpose

The purpose of the ObjectTransformer plug-in is to allow applications to provide custom methods for the following operations:

- Serialize or deserialize the key for an entry

- Serialize or deserialize the value for an entry
- Copy a key or value for an entry

If no ObjectTransformer plug-in is provided, you must be able to serialize the keys and values because the ObjectGrid uses a serialize and deserialize sequence to copy the objects. This method is expensive, so use an ObjectTransformer plug-in when performance is critical. The copying occurs when an application looks up an object in a transaction for the first time. You can avoid this copying by setting the copy mode of the Map to `NO_COPY` or reduce the copying by setting the copy mode to `COPY_ON_READ`. Optimize the copy operation when needed by the application by providing a custom copy method on this plug-in. Such a plug-in can reduce the copy overhead from 65–70% to 2/3% of total CPU time.

The default copyKey and copyValue method implementations first attempt to use the clone() method, if provided. If no clone() method implementation is provided, the implementation defaults to serialization.

Object serialization is also used directly when the ObjectGrid is running in distributed mode. The LogSequence uses the ObjectTransformer plug-in to help it serialize keys and values before transmitting the changes to peers in the ObjectGrid. You must take care when providing a custom serialization method instead of using the built-in JDK serialization. Object versioning is a complex issue and you might encounter problems with version compatibility if you do not ensure that your custom methods are designed for versioning.

The following list details how the ObjectGrid tries to serialize both keys and values:
- If a custom ObjectTransformer plug-in is written and plugged in, ObjectGrid calls methods in the ObjectTransformer methods to serialize keys and values and get copies of object keys and values..
- If a custom ObjectTransformer plug-in is not used, ObjectGrid serializes and deserializes according to the default. If the default is used, each object is implemented as externalizable or is implemented as serializable.
  – If the object supports the Externalizable interface, the `writeExternal` method is called. Objects that are implemented as externalizable lead to better performance.
  – If the object does not support the Externalizable interface and does implement Serializable, the object is saved using the `ObjectOutputStream` method.

## ObjectTransformer interface

See the API documentation for more information about the ObjectTransformer interface. The ObjectTransformer interface contains the following methods that serialize and deserialize keys or values and copy keys or values:

```
public interface ObjectTransformer
{
 void serializeKey(Object key, ObjectOutputStream stream)
  throws IOException;
 void serializeValue(Object value, ObjectOutputStream stream)
  throws IOException;
 Object inflateKey(ObjectInputStream stream)
  throws IOException, ClassNotFoundException;
 Object inflateValue(ObjectInputStream stream)
  throws IOException, ClassNotFoundException;
 Object copyKey(Object value);
 Object copyValue(Object value);
}
```

## ObjectTransformer interface usage

You can use the ObjectTransformer interface in the following situations:

- non-serializable object
- serializable object but improve serialization performance
- key or value copy

In the following example, ObjectGrid is used to store the Stock class:

```
/**
* Stock object for ObjectGrid demo
*
*
*/
public class Stock implements Cloneable {
 String ticket;
 double price;
 String company;
 String description;
 int serialNumber;
 long lastTransactionTime;
 /**
 * @return Returns the description.
 */
 public String getDescription() {
  return description;
 }
 /**
 * @param description The description to set.
 */
 public void setDescription(String description) {
  this.description = description;
 }
 /**
 * @return Returns the lastTransactionTime.
 */
 public long getLastTransactionTime() {
  return lastTransactionTime;
 }
 /**
 * @param lastTransactionTime The lastTransactionTime to set.
 */
 public void setLastTransactionTime(long lastTransactionTime) {
  this.lastTransactionTime = lastTransactionTime;
 }
 /**
 * @return Returns the price.
 */
 public double getPrice() {
  return price;
 }
 /**
 * @param price The price to set.
 */
 public void setPrice(double price) {
  this.price = price;
 }
 /**
 * @return Returns the serialNumber.
 */
 public int getSerialNumber() {
  return serialNumber;
 }
 /**
 * @param serialNumber The serialNumber to set.
 */
 public void setSerialNumber(int serialNumber) {
  this.serialNumber = serialNumber;
 }
 /**
 * @return Returns the ticket.
 */
 public String getTicket() {
  return ticket;
 }
 /**
 * @param ticket The ticket to set.
 */
 public void setTicket(String ticket) {
  this.ticket = ticket;
 }
 /**
 * @return Returns the company.
 */
 public String getCompany() {
  return company;
 }
 /**
 * @param company The company to set.
 */
 public void setCompany(String company) {
  this.company = company;
 }
 //clone
 public Object clone() throws CloneNotSupportedException
 {
  return super.clone();
 }
}
```

You can write a custom object transformer class for the Stock class:

```
/**
* Custom implementation of ObjectGrid ObjectTransformer for stock object
*
*/
public class MyStockObjectTransformer implements ObjectTransformer {
 /* (non-Javadoc)
 * @see
 * com.ibm.websphere.objectgrid.plugins.ObjectTransformer#serializeKey
 * (java.lang.Object,
 * java.io.ObjectOutputStream)
 */
 public void serializeKey(Object key, ObjectOutputStream stream)
 throws IOException {
  String ticket= (String) key;
  stream.writeUTF(ticket);
 }

 /* (non-Javadoc)
 * @see com.ibm.websphere.objectgrid.plugins.
 ObjectTransformer#serializeValue(java.lang.Object,
 java.io.ObjectOutputStream)
 */
 public void serializeValue(Object value, ObjectOutputStream stream)
 throws IOException {
  Stock stock= (Stock) value;
  stream.writeUTF(stock.getTicket());
  stream.writeUTF(stock.getCompany());
  stream.writeUTF(stock.getDescription());
  stream.writeDouble(stock.getPrice());
  stream.writeLong(stock.getLastTransactionTime());
  stream.writeInt(stock.getSerialNumber());
 }

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#inflateKey(java.io.ObjectInputStream)
*/
 public Object inflateKey(ObjectInputStream stream) throws IOException,
 ClassNotFoundException {
  String ticket=stream.readUTF();
  return ticket;
 }

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#inflateValue(java.io.ObjectInputStream)
*/

 public Object inflateValue(ObjectInputStream stream) throws IOException,
 ClassNotFoundException {
  Stock stock=new Stock();
  stock.setTicket(stream.readUTF());
  stock.setCompany(stream.readUTF());
  stock.setDescription(stream.readUTF());
  stock.setPrice(stream.readDouble());
  stock.setLastTransactionTime(stream.readLong());
  stock.setSerialNumber(stream.readInt());
  return stock;
 }

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#copyValue(java.lang.Object)
*/
 public Object copyValue(Object value) {
  Stock stock = (Stock) value;
  try{
   return stock.clone();
  }
  catch (CloneNotSupportedException e)
  {
   //streamize one
  }
 }

/* (non-Javadoc)
* @see com.ibm.websphere.objectgrid.plugins.
ObjectTransformer#copyKey(java.lang.Object)
*/
 public Object copyKey(Object key) {
  String ticket=(String) key;
  String ticketCopy= new String (ticket);
  return ticketCopy;
 }
}
```

Then, plug in this custom MyStockObjectTransformer class into the BackingMap:

```
ObjectGridManager ogf=ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogf.getObjectGrid("NYSE");
BackingMap bm = og.defineMap("NYSEStocks");
MyStockObjectTransformer ot = new MyStockObjectTransformer();
bm.setObjectTransformer(ot);
```

## TransactionCallback plug-in

An application usually plugs in both a TransactionCallback plug-in and a Loader as a pair. The Loader is responsible for fetching data from the back end as well as applying changes to the back end. This fetching and flushing usually take place within the context of an ObjectGrid transaction.

The TransactionCallback plug-in has the following responsibilities:

- Reserves slots for a transaction-specific state that is needed for the transaction and the Loader
- Translates or maps an ObjectGrid transaction to a platform transaction
- Sets up the per transaction state when the ObjectGrid begins a transaction
- Commits the transaction when the ObjectGrid transaction commits
- Rolls back the transaction when the ObjectGrid transaction rolls back

The ObjectGrid is not a XA transaction coordinator. The ObjectGrid relies on the platform to provide that capability. The ObjectGrid `begin`, `commit`, and `rollback` methods that are presented on a Session are lifecycle calls. The TransactionCallback plug-in must receive these events and make the platform provide the transactional capability for the resources used by the Loaders. This topic examines various scenarios and discusses how the TransactionCallback plug-in can be written to work for these scenarios.

### TransactionCallback plug-in overview

The TransactionCallback plug-in is a POJO that implements the TransactionCallback interface. The TransactionCallback interface looks like the following sample:

```
public interface TransactionCallback
{
 void initialize(ObjectGrid objectGrid) throws TransactionCallbackException;
 void begin(TxID id) throws TransactionCallbackException;
 void commit(TxID id) throws TransactionCallbackException;
 void rollback(TxID id) throws TransactionCallbackException;
 boolean isExternalTransactionActive(Session session);
}
```

**`initialize` method**

The `initialize` method is called when the ObjectGrid is initialized. The callback reserves slots for the TxID object that it needs. Usually, it reserves a slot for each piece of the state or Object that it wants to create in the `begin` method when a transaction starts. For example, you want to use a Persistence Manager with the ObjectGrid as a Loader. Assuming that this persistence manager has session and transaction state objects, the TransactionCallback would obtain a session and transaction and keep references to those two objects in slots on the TxID. In this case, the `initialize` method looks like the following sample:

```
/**
* This is called when the grid first initializes. We'll just
* reserve our slots in the TxID.
*/
public void initialize(ObjectGrid objectGrid) throws
 TransactionCallbackException
{
 // reserve a slot for the persistence manager transaction
 TXslot = objectGrid.reserveSlot(TxID.SLOT_NAME);
 // reserve a slot for the persistence manager session
 SessionSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
}
```

A TxID has slots. The slots are entries on an ArrayList array. Plug-ins can reserve an entry in the ArrayList array by calling the `ObjectGrid.reserveSlot` method and indicating that it wants a slot on the TxID object. The method then returns the next entry index to the application. The application can then store information in this slot. The next methods demonstrate this technique.

**`begin` method**

The ObjectGrid calls this method when it starts a new transaction. The plug-in maps this event to a real transaction that the Loaders can then use for the `get` and `update` method calls that arrive before the `commit` method is called. Following is an example `begin` method that maps an ObjectGrid begin to a persistence manager transaction begin:

```
/**
* This is called when the grid starts a new transaction. We just create a
* persistence manager transaction and call begin on it. We then store
* the transaction in the TxID slot so we can get it again later
* without needing ThreadLocal etc.
*/
public void begin(TxID id) throws TransactionCallbackException
{
 Session PMsession = getPMcurrentSession();
 Transaction tx = PMsession.beginTransaction();
 id.putSlot(TXslot, tx);
 id.putSlot(SessionSlot, PMsession);
}
```

This sample relies on the fact that the `initialize` method has reserved two slots on the TxID object. One slot is for the persistence manager session and the other slot is for the persistence manager Transaction. The `begin` method calls the persistence manager to get a session, stores it in the indexed SessionSlot slot, and creates a Transaction on the session and stores a reference to this transaction using the indexed TXSlot slot.

**`commit` method**

The `commit` method is called when an ObjectGrid transaction is committing. All Loaders have already been flushed. The plug-in responsibility is to communicate this commit event to the platform.

```
/**
* This is called when the grid wants to commit a transaction.
* We just pass it on to persistence manager.
*/
public void commit(TxID id) throws TransactionCallbackException
{
 Transaction tx = (Transaction)id.getSlot(TXslot);
 tx.commit();
}
```

The method looks up the persistence manager transaction stored in the slot and then calls the `commit` method.

**`rollback` method**

This method is called when an ObjectGrid transaction wants to roll back a transaction. The plug-in forwards this to the platform transaction manager. Following is the code snippet:

```
/**
* This is called when the grid wants to rollback a transaction.
* We just pass it on to persistence manager.
*/
public void rollback(TxID id) throws TransactionCallbackException
{
 Transaction tx = (Transaction)id.getSlot(TXslot);
 tx.rollback();
}
```

This method is very similar to the `commit` method. It gets a reference to the persistence manager transaction from a slot and then invokes the `rollback` method.

**`isExternalTransactionActive` method**

An ObjectGrid session normally works in autocommit mode or in transaction mode. Autocommit mode means an implicit transaction is created around every method call to the ObjectMap instances for the session. If no transaction is active and an application makes a call on an ObjectMap method, the framework calls this method on the TransactionCallback plug-in to check if there is a proper transaction active. If this method returns true then the framework does an automatic begin otherwise, it does autocommit. This method allows the ObjectGrid to be integrated in environments where the application invokes `begin`, `commit`, or `rollback` methods on the platform APIs instead of the ObjectGrid APIs.

## Scenario: Simple Java database connectivity (JDBC)-based Java 2 Platform, Standard Edition (J2SE) environment

This example uses a J2SE environment where the application has a JDBC-based Loader. Two Maps exist, each with a Loader that backs each Map by a different table in the database. The TransactionCallback plug-in gets a JDBC connection and then invokes the `begin`, `commit`, and `rollback` methods on the connection. Following is the sample TransactionCallback implementation:

```
public class JDBCTCB implements TransactionCallback
{
 DataSource datasource;
 int connectionSlot;
 public JDBCTCB(DataSource ds)
 {
  datasource = ds;
 }
 public void initialize(ObjectGrid objectGrid)
  throws TransactionCallbackException
  {
   connectionSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
  }
 public void begin(TxID id) throws TransactionCallbackException
 {
  try
  {
   Connection conn = datasource.getConnection();
   conn.setAutoCommit(false);
   id.putSlot(connectionSlot, conn);
  }
  catch(SQLException e)
  {
   throw new TransactionCallbackException("Cannot start transaction", e);
  }
 }
 public void commit(TxID id) throws TransactionCallbackException
 {
  Connection conn = null;
  try
 {
   conn = (Connection)id.getSlot(connectionSlot);
   conn.commit();
   conn.close();
  }
  catch(SQLException e)
  {
   throw new TransactionCallbackException("Cannot commit transaction", e);
  }
  finally {
   if (conn!=null) {
    try {
    conn.close();
    }
    catch (SQLException closeE) {
   }
  }
 }
 }
 public void rollback(TxID id) throws TransactionCallbackException
 {
  Connection conn = null;
  try
  {
   conn = (Connection)id.getSlot(connectionSlot);
   conn.rollback();
   conn.close();
  }
  catch(SQLException e)
  {
   throw new TransactionCallbackException("Cannot rollback transaction", e);
  }
  finally {
   if (conn!=null) {
    try {
     conn.close();
    }
    catch (SQLException closeE) {
    }
   }
  }
 }
 public boolean isExternalTransactionActive(Session session)
 {
  return false;
 }
 public int getConnectionSlot()
 {
  return connectionSlot;
 }
}
```

This example shows a TransactionCallback plug-in that converts the ObjectGrid transaction events to a JDBC connection. When the plug-in is initialized, it reserves a single slot to keep a JDBC connection reference. The `begin` method then obtains a JDBC connection for the new transaction, turns auto commit off, and then stores a reference to the connection in the TxID slot. The `commit` and `rollback` methods retrieve the connection from the TxID slot and call the appropriate method on the connection. The `isExternalTransaction` method always returns false, indicating that the application must use the ObjectGrid transaction APIs explicitly to control transactions. A Loader that is paired with this plug-in obtains the JDBC connection from the TxID. A Loader looks like the following example:

```
public class JDBCLoader implements Loader
{
 JDBCTCB tcb;
 public void preloadMap(Session session, BackingMap backingMap)
 throws LoaderException
 {
 tcb = (JDBCTCB)session.getObjectGrid().getTransactionCallback();
 }
 public List get(TxID txid, List keyList, boolean forUpdate)
 throws LoaderException
 {
  Connection conn = (Connection)txid.getSlot(tcb.getConnectionSlot());
  // implement get here
 return null;
 }
 public void batchUpdate(TxID txid, LogSequence sequence)
 throws LoaderException, OptimisticCollisionException
 {
  Connection conn = (Connection)txid.getSlot(tcb.getConnectionSlot());
  // TODO implement batch update here
 }
}
```

The Loader obtains a reference to the JDBCTCB instance when the `initialize` method is called. It then obtains the Connection obtained by the JDBCTCB when it is required in the `get` and `batchUpdate` methods. TransactionCallback implementations and Loaders are typically written in pairs that cooperate with each other. The TransactionCallback implementation handles the Transaction and stores objects needed by the Loaders in slots in the TxID. The Loaders then implement `get` and `batchUpdate` methods in the context of a transaction managed by the TransactionCallback using resources obtained by the TCB usually.

## Scenario: Servlet engine environment

In this scenario, the ObjectGrid is using a JDBC-based Loader but in a managed servlet engine. The container expects us to use the `UserTransaction` method to begin and commit transactions. This is slightly different from the J2SE case because storing a reference to the JDBC connection in a TxID slot is not necessary. The container manages the JDBC connection. When a container transaction is active, a connection that is looked up using a data source results in the same connection each time because the container remembers which connections are used by this transaction and returns the same connection each time the `DataSource.getConnection` method is called. Assume that the data source reference is configured as `Shareable` in the following example:

```
public class ManagedJDBCTCB implements TransactionCallback {
 UserTransaction tx;
 public void initialize(ObjectGrid objectGrid)
 throws TransactionCallbackException
 {
  try
  {
   InitialContext ic = new InitialContext();
   tx = (UserTransaction)ic.lookup("java:comp/UserTransaction");
  }
  catch(NamingException e)
  {
   throw new TransactionCallbackException("Cannot find UserTransaction", e);
  }
 }
 public void begin(TxID id) throws TransactionCallbackException
 {
  try
  {
   tx.begin();
  }
  catch(SystemException e)
  {
   throw new TransactionCallbackException("Cannot begin tx", e);
  }
  catch(NotSupportedException e)
  {
   throw new TransactionCallbackException("Cannot begin tx", e);
  }
 }
 public void commit(TxID id) throws TransactionCallbackException
 {
  try
  {
   tx.commit();
  }
  catch(SystemException e)
  {
   throw new TransactionCallbackException("Cannot commit tx", e);
  }
  catch(HeuristicMixedException e)
  {
   throw new TransactionCallbackException("Cannot commit tx", e);
  }
  catch(RollbackException e)
  {
   throw new TransactionCallbackException("Cannot commit tx", e);
  }
  catch(HeuristicRollbackException e)
  {
   throw new TransactionCallbackException("Cannot commit tx", e);
  }
 }

 public void rollback(TxID id) throws TransactionCallbackException
 {
  try
  {
   tx.rollback();
  }
  catch(SystemException e)
  {
   throw new TransactionCallbackException("Cannot commit tx", e);
  }
 }
 public boolean isExternalTransactionActive(Session session) {
  return false;
 }
}
```

This example obtains a reference to the `UserTransaction` method in the `initialize` method and then maps `begin`, `commit`, and `rollback` on to the appropriate `UserTransaction` methods. Slots are not needed because the container verifies that the correct connection information is retrieved for this transaction. Following is the JDBC Loader that works with this TransactionCallback implementation:

```
public class ManagedJDBCLoader implements Loader
{
 DataSource myDataSource;
 ManagedJDBCLoader(DataSource ds)
 {
  myDataSource = ds;
 }
 public void preloadMap(Session session, BackingMap backingMap)
 throws LoaderException
 {
 }
 public List get(TxID txid, List keyList, boolean forUpdate)
 throws LoaderException
 {
  try
  {
   Connection conn = myDataSource.getConnection();
   // TODO implement get here with this connection
   return null;
  }
  catch(SQLException e)
  {
   throw new LoaderException("Cannot get objects", e);
  }
 }
 public void batchUpdate(TxID txid, LogSequence sequence)
 throws LoaderException, OptimisticCollisionException
 {
  try
  {
   Connection conn = myDataSource.getConnection();
   // TODO implement update here using this connection
  }
  catch(SQLException e)
  {
   throw new LoaderException("Cannot update objects", e);
  }
 }
}
```

This example can be simpler than the basic JDBC version because the container manages the connections and verifies that within the same transaction, the `DataSource.getConnection` method always returns the same connection when it is called with the same transaction active each time. Do not try to cache the connection in a slot as a result, although the application can cache the connection if it chooses to.

## OptimisticCallback interface

You can provide a pluggable optimistic callback object that implements the com.ibm.websphere.objectgrid.plugins.OptimisticCallback interface.

### Purpose

The OptimisticCallback interface is used to provide optimistic comparison operations for the values of a map. An OptimisticCallback is required when the optimistic lock strategy is being used as described in "Optimistic locking" on page 129. ObjectGrid provides a default OptimisticCallback implementation. However, usually the application must plug in its own implementation of the OptimisticCallback interface.

### Plug in an application-provided OptimisticCallback object

The following example demonstrates how an application can plug in an OptimisticCallback object for the employee backing map in the grid1 ObjectGrid instance:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
```

```
ObjectGrid og = ogManager.createObjectGrid( "grid1" );
BackingMap bm = dg.defineMap("employees");
EmployeeOptimisticCallbackImpl cb = new EmployeeOptimisticCallbackImpl();
bm.setOptimisticCallback( cb );
```

The EmployeeOptimisticCallbackImpl object in the preceding example must implement the OptimisticCallback interface. The application can also use an XML file to plug in its OptimisticCallback object as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
 <objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
   <objectGrid name="grid1">
    <backingMap name="employees" pluginCollectionRef="employees"
     lockStrategy="OPTIMISTIC" />
   </objectGrid>
  </objectGrids>

 <backingMapPluginCollections>
  <backingMapPluginCollection id="employees">
   <bean id="OptimisticCallback"
    className="com.xyz.EmployeeOptimisticCallbackImpl" />
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

## Default implementation

The ObjectGrid framework provides a default implementation of the OptimisticCallback interface that is used if the application does not plug in an application-provided OptimisticCallback object, as demonstrated in the previous section. The default implementation always returns the special value of NULL_OPTIMISTIC_VERSION as the version object for the value and never updates the version object. This action makes optimistic comparison a ″no operation″ function. In most cases, you do not want the ″no operation″ function to occur when you are using the optimistic locking strategy. Your applications must implement the OptimisticCallback interface and plug in their own OptimisticCallback implementations so that the default implementation is not used. However, at least one scenario exists where the default provided OptimisticCallback implementation is useful. Consider the following situation:

- A loader is plugged in for the backing map.
- The loader knows how to perform the optimistic comparison without assistance from an OptimisticCallback plug-in.

How can the Loader know how to deal with optimistic versioning without assistance from an OptimisticCallback object? The Loader has knowledge of the value class object and knows which field of the value object is used as an optimistic versioning value. For example, suppose the following interface is used for the value object for the employees map:

```
public interface Employee
{
// Sequential sequence number used for optimistic versioning.
public long getSequenceNumber();
public void setSequenceNumber(long newSequenceNumber);
// Other get/set methods for other fields of Employee object.
}
```

In this case, the Loader knows that it can use the getSequenceNumber method to get the current version information for an Employee value object. It increments the returned value to generate a new version number before updating the persistent storage with the new Employee value. For a Java database connectivity (JDBC) Loader, the current sequence number in the where clause of an overqualified SQL update statement is used, and it uses the new generated sequence number to set the sequence number column to the new sequence number value. Another possibility is that the Loader makes use of some backend-provided function that automatically updates a hidden column that can be used for optimistic versioning. In some cases, a stored procedure or trigger can possibly be used to help maintain a column that holds versioning information. If the Loader is using one of these techniques for maintaining optimistic versioning information, then the application does not need to provide an OptimisticCallback implementation. The default OptimisticCallback is usable in this case because the Loader is able to handle optimistic versioning without any assistance from an OptimisticCallback object.

## Implement the OptimisticCallback interface

The OptimisticCallback interface contains the following methods and special values:

```
public interface OptimisticCallback
{
 final static Byte NULL_OPTIMISTIC_VERSION;
 Object getVersionedObjectForValue(Object value);
 void updateVersionedObjectForValue(Object value);
 void serializeVersionedValue(Object versionedValue,
 ObjectOutputStream stream) throws IOException;
 Object inflateVersionedValue(ObjectInputStream stream) throws
 IOException, ClassNotFoundException;
}
```

The following list provides a description or consideration for each of the methods in the OptimisticCallback interface:

**NULL_OPTIMISTIC_VERSION**
: This special value is returned by getVersionedObjectForValue method if the default OptimisticCallback implementation is used instead of an application-provided OptimisticCallback implementation.

**getVersionedObjectForValue method**
: This method might return a copy of the value or it might return an attribute of the value that can be used for versioning purposes. This method is called whenever an object is associated with a transaction. When no Loader is plugged into a backing map, the backing map uses this value at commit time to perform an optimistic version comparison. The optimistic version comparison is used by the backing map to ensure that the version has not changed since this transaction first accessed the map entry that was modified by this transaction. If another transaction had already modified the version for this map entry, the version comparison fails and the backing map displays an OptimisticCollisionException exception to force rollback of the transaction. If a Loader is plugged in, the backing map does not use the optimistic versioning information. Instead, the Loader is responsible for performing the optimistic versioning comparison and updating the versioning information when necessary. The Loader typically gets the initial versioning object from the LogElement passed to the Loader's batchUpdate method, which is called when a flush operation occurs or a transaction is committed.

  The following code shows the implementation used by the EmployeeOptimisticCallbackImpl object:

```
public Object getVersionedObjectForValue(Object value)
{
 if (value == null)
 {
  return null;
 }
 else
 {
  Employee emp = (Employee) value;
  return new Long( emp.getSequenceNumber() );
 }
}
```

As demonstrated in the previous example, the sequenceNumber attribute is returned in a java.lang.Long object as expected by the Loader, which implies that the same person that wrote the Loader either wrote the EmployeeOptimisticCallbackImpl implementation or worked closely with the person that implemented the EmployeeOptimisticCallbackImpl - for example, agreed on the value returned by the getVersionedObjectForValue method.

As previously described, the default OptimisticCallback returns the special value NULL_OPTIMISTIC_VERSION as the version object.

**updateVersionedObjectForValue method**

This method is called whenever a transaction has updated a value and a new versioned object is needed. If the getVersionedObjectForValue returns an attribute of the value, this method typically updates the attribute value with a new version object. If getVersionedObjectForValue returns a copy of the value, this method typically would do nothing. The default OptimisticCallback does nothing since the default implementation of getVersionedObjectForValue always returns the special value NULL_OPTIMISTIC_VERSION as the version object.

The following shows the implementation used by the EmployeeOptimisticCallbackImpl object that is used in the OptimisticCallback section:

```
public void updateVersionedObjectForValue(Object value)
{
 if ( value != null )
 {
  Employee emp = (Employee) value;
  long next = emp.getSequenceNumber() + 1;
  emp.updateSequenceNumber( next );
 }
}
```

As demonstrated in the previous example, the sequenceNumber attribute is incremented by one so that the next time the getVersionedObjectForValue method is called, the java.lang.Long value that is returned has a long value that is the original sequence number value plus one, for example, is the next version value for this employee instance. Again, this example implies that the same person that wrote the Loader either wrote EmployeeOptimisticCallbackImpl or worked closely with the person that implemented the EmployeeOptimisticCallbackImpl.

**serializeVersionedValue method**

This method writes the versioned value to the specified stream. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate the version of the value.

Since the actual implementation is unknown, this method is provided to perform the proper serialization. The default implementation does a writeObject call.

**inflateVersionedValue method**

This method takes the serialized version of the versioned value and returns the actual versioned value object. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate the version of the value. Because the actual implementation is unknown, this method is provided to perform the proper deserialization. The default implementation does a readObject.

## Replication programming

Replication is configured by associating a MapSet with a ReplicationGroup and replication policy attributes. The ReplicationGroup defines the server members that are used for the primary and associated replicas and standbys. It also defines the minimum and maximum number of replicas that are required for this configuration. The replication policy attributes indicate whether synchronous or asynchronous replication is required, whether to allow read access to the replicas, and whether to use compression when sending replication data to the replicas. Replication has a minimal impact on the programming model. The main impact is on the applications that preload data into their Maps.

### Map preloading

You can associate a Loader with each Map. A Loader is used to fetch objects when they cannot be found in the Map and also to write changes to a back end when a transaction commits. Loaders can also be used for pre-loading data into a map. The preload method of the Loader interface is called when the Java virtual machine (JVM) becomes a primary for the replication group. The preload method is not called on replicas or standbys. The preload method attempts to load all the intended referenced data from the back end into the Map using the provided Session. The Map to be used is identified by the BackingMap argument that is passed to the preload method.

```
void preloadMap(Session session, BackingMap backingMap) throws LoaderException;
```

### Preloading in a partitioned MapSet

Maps can be partitioned in to *N* partitions. Maps can be stored across multiple servers, with each entry identified by a key that is only stored on one of those servers. Very large Maps can be held in an ObjectGrid because the application is no longer limited by the heap size of a single JVM to hold all the entries of a Map. Applications that want to preload with the preload method of the Loader interface must identify the subset of the data that it should preload. A fixed number of partitions always exists. This can be determined using the following code snippet:

```
 int numPartitions = backingMap.getPartitionManager().getNumOfPartitions();

int myPartition = backingMap.getPartitionId();
```

This code snippet shows how an application can identify the subset of the data to preload from the database. Applications must always use these methods even when the map is not initially partitioned. These methods allow flexibility: if the Map is later partitioned by the administrators, then the Loader continues to work correctly.

The application must issue queries to retrieve the subset myPartition from the backend. If a database is being used then it might be easier to have a column with the partition identifier for a given record unless there is some natural query that allows the data in the table to be partitioned easily.

## Performance

The preload implementation should copy data from the back end into the Map by storing multiple objects in the Map in a single transaction. The next question is, ″How many records to store per transaction?″ and, unfortunately, the answer is, ″It depends.″ After the transaction includes more than blocks of 100 entries then the performance benefit diminishes. The optimal number depends on a number of factors including object complexity and size. Start with 100 entries and then increase the number until no more performance gains are seen. Larger transactions result in better replication performance. Remember, only the primary runs the preload code. The preloaded data is replicated from the primary to any replicas that are online.

## Preloading MapSets

If the application uses a MapSet with multiple Maps then each Map has its own Loader. Each Loader has a preload method. Each Map is loaded serially by the ObjectGrid. It might be more efficient to preload all the Maps by designating a single Map as the preloading Map. This is just an application convention. For example, two Maps, department and employee, might use the department Loader to preload both the department and the employee Maps. This ensures that, transactionally, if an application wants a department then the employees for that department are in the cache. Of course, this means that when the department Loader preloads a department from the back end then it also fetches the employees for that department. The department object and its associated employee objects should then be added to the Map using a single transaction for this to be true.

## Recoverable preloading

Some customers have very large data sets that need to be cached. Preloading this data can be very time consuming. Sometimes, the preloading must complete before the application can go online. This might mean that you want to make preloading recoverable. Suppose there were a million records to preload. The primary is preloading them and fails at the 800,000th record. Normally, the replica chosen to be the new primary clears any replicated state and start from the beginning. ObjectGrid can do better than that by using a ReplicaPreloadController. The Loader for the application would also need to implement the ReplicaPreloadController interface. This adds a single method to the Loader:

```
Status checkPreloadStatus(Session session, BackingMap bmap);
```

This method is called by the ObjectGrid runtime before preload method of the Loader interface is normally called. The ObjectGrid tests the result of this method (Status) to determine its behavior whenever a replica is promoted to a primary.

| Returned status value | ObjectGrid behavior in reaction |
|---|---|
| Status.PRELOADED_ALREADY | ObjectGrid does not call the preload method at all because this status value indicates that the Map is fully preloaded. |
| Status.FULL_PRELOAD_NEEDED | ObjectGrid clears the Map and calls the preload method normally. |

| Returned status value | ObjectGrid behavior in reaction |
|---|---|
| Status.PARTIAL_PRELOAD_NEEDED | ObjectGrid leaves the Map as-is and calls preload. This strategy allows the application's Loader to continue preloading from that point onwards. |

Clearly, while a primary is preloading the Map, it must leave some state in a Map in the MapSet being replicated so that the replica can figure out what status to return. You can use an extra Map called, for example, RecoveryMap. This RecoveryMap must be part of the same MapSet that is being preloaded. This ensures that it is replicated consistently with the data being preloaded.

A suggested implementation follows. As the preload commits each block of records, it should also update a counter/value in the RecoveryMap as part of that transaction. This means the preloaded data and the RecoveryMap data are replicated atomically to the replicas. When the replica is promoted to primary, it can now check the RecoveryMap to see what has happened. The RecoveryMap may simply hold a single entry with key 'state'. If no object exists for this key then we need a full preload (checkPreloadStatus returns FULL_PRELOAD_NEEDED). If an object exists for this 'state' key then if the value is 'COMPLETE' then the preload is done and the checkPreloadStatus returns PRELOADED_ALREADY. Otherwise, the value object indicates where preload should restart from and the checkPreloadStatus method should return PARTIAL_PRELOAD_NEEDED. The Loader may store the recovery point in an instance variable for the Loader so that when preload is called, it knows the starting point. The RecoveryMap could also hold an entry per Map if each Map is preloaded independently.

## Handling recovery in synchronous replication mode with a Loader

The ObjectGrid runtime is designed to not lose committed data when the primary fails. The following section shows the algorithms used to achieve this. These algorithms apply only when a replication group uses synchronous replication. A Loader is optional.

The ObjectGrid runtime can be configured to replicate all changes from a primary to the replicas synchronously. When a JVM is promoted to be a replica, the primary first sends a snapshot of the Map to the replica. Once the replica has processed this snapshot, the primary starts sending all the changes (completed transactions) since the generation of the snapshot. Eventually, the replica will catch up with the primary. This initial replication processing is asynchronous. Once a replica catches up with the primary then the pair enters peer mode and, finally, synchronous replication begins. From this point on, each transaction committed on the primary will be sent to all replicas in peer mode and the primary waits for an acknowledge message. This slows down the primary when compared with an asynchronous replication scenario because of the latency involved in receiving acknowledge messages. A synchronous commit sequence on the primary looks like this:

| Step with Loader | Step without Loader |
|---|---|
| Get locks for entries | same |
| Flush changes to the Loader | NOOP |
| Save changes to the cache | same |

| Step with Loader | Step without Loader |
|---|---|
| Sent changes to replicas and wait for acknowledgement | same |
| Commit to the loader through the TransactionCallback plug-in | The TransactionCallBack plug-in commit is still called but typically does not do anything. |
| Release locks for entries | same |

Notice that the changes are sent to the replica before they are committed to the Loader. When are the changes committed on the replica? Revise this sequence:

At initialize time, initialize the tx lists on the primary.

- Set CommitedTx = {}, RolledBackTx = {}

During synchronous commit processing:

| Step with Loader | Step without loader |
|---|---|
| Get locks for entries | same |
| Flush changes to the Loader | NOOP |
| Save changes to the cache | same |
| Send changes with a committed transaction and rolled back transaction to replica and wait for acknowledgement | same |
| Clear list of committed transactions and rolled back transactions | same |
| Commit the Loader through the TransactionCallBack plug-in | TransactionCallBack plug-in commit is still called but typically does not do anything |
| If commit succeeds, add the transaction to the committed transactions, otherwise add to the rolled back transactions | NOOP |
| Release locks for entries | same |

Replica processing

- Receive changes
- Commit all received transactions in the committed transaction list
- Roll back all received transactions in the rolled back transaction list
- Start a transaction or session
- Apply changes to the transaction or session
- Save the transaction or session to the pending list
- Send back reply

Notice that on the replica there are no Loader interactions while it is in replica mode. The primary must push all changes through the Loader. The replica is a drone.

A side effect of this algorithm is that the replica always has the transactions but they are not committed until the next primary transaction sends the commit status of those transactions. They are then committed or rolled back on the replica. But, until then, the transactions are not committed. We may add a timer on the primary that will send the transaction outcome after a small period of time (a few seconds). This will limit any staleness to that time window, but it will not eliminate it completely.

This staleness is only a problem when using replica read mode. Otherwise, it’s invisible and has no impact on the application.

When the primary fails, it’s likely there are a few transactions that were committed/rolled back on the primary but the message never made it to the replica with these outcomes. When a replica is promoted to the new primary, one of it’s first actions is to handle this condition. Each pending transaction is reprocessed against the new primary’s set of Maps. If there is a Loader then each transaction is given to the Loader. These transactions are applied in strict FIFO order. If a transactions fails then it’s ignored. If there are 3 transactions pending, A B and C, then A may commit, B may rollback and C may also commit. No one transaction has any impact on the others. We assume they are independent.

A Loader may want to use slightly different logic when in ’failover recovery’ mode versus ’normal’ mode. The Loader can easily know when it’s in failover recovery mode by implementing the ReplicaPreloadController interface. The checkPreloadStatus method is only called when failover recovery completes. Therefore, if the apply method of the Loader interface is called before checkPreloadStatus then it is a recovery transaction. After the checkPreloadStatus method is called then failover recovery has completed.

## Stateful singletons using replication

WebSphere Extended Deployment added support for singletons in its first release with the partitioning facility. This allowed applications to create singletons in a cluster. The ObjectGrid runtime enables a similar feature using replicated MapSets. While the ObjectGrid singleton pattern has many advantages, it also has a few disadvantages. The partition facility provides an event to the application when the singleton/partition is activated locally, this is communicated using the partitionLoad method of the partition facility. A replicated MapSet also has a singleton, the primary. The application is notified when it becomes the primary by the ReplicaPreloadController#checkPreloadStatus method on the Loader. This can be used in a similar way to the partitioning facility but has the advantage of being portable across different versions of WebSphere Application Server or competitive application servers.

The partition facility has a deactivate event, but the ObjectGrid runtime does not offer this capability. A primary in the ObjectGrid normally runs until it fails. You cannot move it around. This is an advantage of the partitioning facility over the ObjectGrid. Here is a table of capabilities:

*Table 16.*

| Capability | Partitioning facility | ObjectGrid singletons |
|---|---|---|
| Singleton start event | Yes | Yes |
| Singleton stop event | Yes | No |
| Replication of singleton state | No | Yes |
| Variable quality of service (QoS) for replication | No | Yes |
| Flexible singleton placement | Yes | No |
| Can move singleton at runtime | Yes | No |
| IIOP routing of work to singleton | Yes | No |

*Table 16. (continued)*

| Capability | Partitioning facility | ObjectGrid singletons |
|---|---|---|
| Requires a Java 2 Platform, Enterprise Environment (J2EE) server | Yes | No |
| Requires a full version of WebSphere Extended Deployment | Yes | No |
| Requires Enterprise JavaBeans (EJB) | Yes | No |
| Application is portable to other application servers | No | Yes |

## Singleton state

The partitioning facility offered no built-in support for state management. Applications were left to their own devices if the singleton required state. Typically, this meant the state was pushed to a database. If the partition failed then the server that was elected to host and recover the partition needed to retrieve this state from that database. If an application uses the ObjectGrid instead, then the singleton can keep its state in the Map associated with the ReplicaPreloadController managing the singleton. If the primary or singleton fails, then the replica that is elected to be the new primary already has the state locally because of the replication. Synchronous replication should be used unless data loss is acceptable to the application.

## Flexible singleton placement

The partitioning facility uses the high availability manager policy mechanism to determine where a partition will be hosted and these policies can be changed at runtime with immediate effect. The ObjectGrid replication group policies are not as flexible as those with the high availability manager and cannot be changed without restarting all the servers. You lose the ability to move around singletons at runtime if you are using the ObjectGrid.

## Variable QoS replication

The partition facility doe not offer state management. The ObjectGrid offers a variety of replication approaches:

- No replication
- Asynchronous replication
- Synchronous replication

The replication policy of the MapSet associated with the Map you are using for the state determines the policy. Synchronous replication means no data loss, but it is slower. Asynchronous replication is fast but means one or more transactions committed on the primary can be lost if the primary fails.

## Load balancing across replicas

The ObjectGrid, unless configured otherwise, sends all read and write requests to the primary server for a given replication group. This means the primary alone must service all requests from clients. You might want to allow read requests to be sent to replicas of the primary. This allows the load of the read requests to be shared by multiple JVMs, however, but sending read requests to the replicas are at the expense of consistency.

This is typically only used when clients are caching data which is changing all the time or clients are using pessimistic locking.

If the data is continually changing then being invalidated in client near caches and the primary should see a relatively high get request rate from clients as a result. Likewise, in pessimistic locking mode, there is no local cache so all requests are sent to the primary.

If the data is relatively static or pessimistic mode is not used then replica read does not have a big impact on performance as the frequency of get requests from clients with warm caches will not be high.

However, when a client first starts, its near cache is empty and cache requests to that empty cache are forwarded to the primary. The client cache gets data over time, causing this request load will drop. If there is a large number of clients and many of them start concurrently, then this load might be significant and replica read may be an appropriate performance choice.

## Reads from replicas and asynchronous replication

If the data in the replication group does not change often then this is usually a good trade off. This allows get requests from clients to be directed to the data on any replicas that are online. A get request might be sent to a replica that does not have a copy, and the key/value might not have been replicated to the replica at that point. If the data is not on the replica then the get request is redirected to the primary.

If the data changes, then it is very likely that gets from the replicas return stale data. This might or might not be acceptable to the application. If it is not acceptable then do not enable reads from replicas.

## Reads from replicas in synchronous replication mode

Synchronous replication tries to keep the replica exactly the same as the primary. If the primary fails then all the committed data on the primary is guaranteed to be available on all replicas that were in peer mode when the failure occurred. While this is the case when failures occur, allowing reads from replicas exposes some side effects of the algorithms used.

When the primary is about to commit a transaction, a copy of the changes is sent to the replica and the replica commits this transaction in the following two cases:

- The primary fails
- The next transaction on the primary is sent

When the primary fails, all pending transactions on the replica are committed.

Pending transactions are only committed when a subsequent transaction is committed on the primary. The primary piggy backs on this replica message the outcome of committing. When the replica receives one of these messages, it commits or rolls back any pending transactions that had outcomes specified in that message.

Pending transactions only become visible to read on a replica when they are committed. Obviously, if the primary is loaded and has regular modifications made to it, then these pending transactions are committed very quickly. If the modification load on the primary is low then there are periods where pending transactions are not committed, until the next primary modification is made.

Clearly, the replica for a primary that is taking modifications is normally at least one transaction behind the primary from a read point of view. No data is lost, these transactions are physically on the replica, they are simply not committed until the outcome of those pending transactions is sent from the primary. This commit happens when the next read and write transaction runs.

### Summary

If read from replica is enabled, then the application must be prepared to tolerate some gets returning stale data. This issue is true whether synchronous or asynchronous replication is being used.

## Partitioning

Use partitioning when the objects in your MapSet require more memory than is available in a single Java virtual machine (JVM), or if the JVM is not able to provide the required throughput for updates.

### Where are entries held?

A hashing algorithm determines which server holds each entry. The administrator specifies the number of partitions to use with the PartitionSet definition. This configuration cannot be changed after the JVMs start. A simple hash value is obtained from the key for an entry and the result of this value modulo (%) the number of partitions indicates which server ″owns″ that entry.

Normally, the Java hashCode method on the key object is used. Override this value by overriding the hashCode implementation.

Sometimes, an application might not want to modify the value for normal hashing but might still want to use a different hash algorithm for entry distribution. The com.ibm.websphere.objectgrid.plugins.PartitionableKey interface allows this situation. This interface has a single method:

```
Object ibmGetPartition();
```

If the key implements this interface, the ObjectGrid runtime uses the hash of the object that is returned by this method rather than the hash on the key object.

### Partitioning at runtime

The com.ibm.websphere.objectgrid.PartitionManager interface provides APIs to allow an application to determine information about partitioning at runtime. An application can obtain a reference to an instance of this interface by using the getPartitionManager method of the BackingMap interface. A reference to the BackingMap for a Map can be obtained using the getMap(String) method of the ObjectGrid interface on any ObjectGrid instance. Or, it is passed as a parameter on some of the plug-in callbacks, such as the preload(Session, BackingMap) of the Loader interface.

## Methods of the PartitionManager interface

The PartitionManager instance allows an application to determine the following facts about partitioning:

| Method name | Description |
|---|---|
| int getNumOfPartitions() | Returns the number of partitions that the Map is being split into. |
| int getPartition(Object key) | This returns the 0-based partition number that is used for the entry with the specified key. |
| List /*Integer*/ getPartitions(List /*Object*/ keys) | This method is the same as the getPartition method but operates on a List of keys instead. The returned List of Integers contain the partition number for each corresponding input key. |
| List /*List Integer*/ getPartitionLists(List /* Object */ keys) | This method is the same as the getPartitions method but returns an ordered List of partition Lists. For example, the first entry in the returned List contains a List of the input keys that correspond to partition 0. The next entry would contain a List of input keys that correspond to partition 1, and so on. |
| List /*LogSequence*/ partitionLogSequence(LogSequence ls) | This method splits a LogSequence into a list of LogSequences for specified partitions. The input LogSequence is examined and the appropriate partition is determined for each LogElement within it. After the sequence has been examined then for each partition which has a LogElement, a LogSequence of those LogElements is returned. |

## Partitioning limitations

A transaction can only modify entries in a single partition per transaction. If a transaction modifies multiple entries in a MapSet and those entries hash to different partitions, the transaction rolls back when an attempt is made to commit the transaction. A transaction can read objects from different partitions. However, a transaction can only modify entries within a single partition.

## Application events when the primary for a partition is elected

If a Loader is supplied for a Map and the ReplicaPreloadController is also implemented by the Loader then the application can use the checkPreloadStatus callback to receive an event indicating that the JVM receiving that method call is now the primary for that partition. The partition ID can be identified using the getPartitionId method of the BackingMap interface. See “Loaders” on page 191 for more information about preload.

## Partitioning on a client versus running on a server

Partitioning only works when the application is using an ObjectGrid that is obtained using the connect methods of the ObjectGrid interface. If the ObjectGrid is provided to the application by a callback on a plug-in, then it is a local ObjectGrid that does not do routing. If you are running on a server and want to take advantage of the partitioning capabilities transparently, then use an ObjectGrid that is obtained using a connect method for all transactions. However, there is a performance loss when compared with using the local ObjectGrid reference supplied by the framework. If you do not need the partitioning capability then use the local reference that is provided to the plug-in when possible.

## Indexing

The indexing feature can be used to build an index or several indices on a BackingMap. An index is built from an attribute of an object in the BackingMap. This feature provides a way for applications to find certain objects more quickly. Without an index, applications have to locate objects by their keys. The indexing feature allows applications to find objects with a specific value or within a range of values. This is similar to Enterprise JavaBeans (EJB) Query that can locate EJB objects by querying with a specified criteria. Indexing provides applications the convenience of finding objects more easily and a performance improvement in the object searching process.

There are two types of indexing: static and dynamic. With *Static* indexing, you must configure the index plug-in on the BackingMap before initializing the ObjectGrid instance. You can do this configuration with XML or programmatic configuration of the BackingMap. Static indexing starts building an index during ObjectGrid initialization. The index is always synchronized with the BackingMap and ready for use. After the static indexing process has started, the maintenance of the index is part of ObjectGrid transaction management process. When transactions commit changes, these changes also update the static index. The index changes are rolled back if the transaction is rolled back.

Dynamic indexing allows an index to be created on a BackingMap before or after the initialization of the containing ObjectGrid instance. Applications have life cycle control over the dynamic indexing process. A dynamic index can be removed when it is no longer needed. When an application creates a dynamic index, the index might not be ready for immediate use because of the time it takes to complete the index building process. Because the amount of time is dependent upon the amount of data indexed, the DynamicIndexCallback interface is provided for applications that want to receive notifications when certain indexing event occur. These events include ready, error, and destroy. Applications can implement this callback interface and register with the dynamic indexing process.

The indexing feature is represented by the MapIndexPlugin plug-in, or Index for short. The MapIndexPlugin is a BackingMap plug-in. A BackingMap can have multiple Index plug-ins configured as long as they follow the Index configuration rules.

If a BackingMap has an index plug-in configured, the index proxy object can be retrieved from the corresponding ObjectMap. Calling the getIndex method on the ObjectMap and passing in the name of the index plug-in returns the index proxy object. The index proxy object has to be cast to an appropriate application index interface, such as MapIndex, MapRangeIndex, or customized index interface.

Currently, the indexing feature is only supported in the local cache, not the distributed cache. If an indexing operation is attempted against a distributed cache, the `UnsupportedOperationException` exception results.

## Index plug-in implementation

The HashIndex class in the com.ibm.websphere.objectgrid.plugins.index package is the built in index plug-in implementation that can support both built-in application index interfaces: MapIndex and MapRangeIndex.

Applications can provide their own index plug-in implementation to allow more complex indices to be programmed. The index implementation class needs to implement com.ibm.websphere.objectgrid.plugins.index.MapIndexPlugin interface. The MapIndexPlugin has the following definition:

```
/**
* An index implementation must implement this interface so that modifications
* to the Map are propagated to it so that it can maintain the index as
* transactions are committed.  Only attributes that implement the
* {@link java.lang.Comparable} interface are eligible to be indexed.
*
* @see com.ibm.websphere.objectgrid.plugins.index.MapIndex
* @see com.ibm.websphere.objectgrid.plugins.index.MapRangeIndex
*/
public interface MapIndexPlugin
{
  /**
  * This should be the name of the attribute to be indexed. If the object
  * has an attribute called EmployeeName then the index will call the
  * "getEmployeeName" method. The attribute name must be the name
  * as that in the get method and the attribute must implement the
  * {@link java.lang.Comparable} interface.
  *
  * @param attributeName
  *           The name of the attribute to set.
  */
  public void setAttributeName(String attributeName);

  /**
  * This index name.
  *
  * @return The name of the index.
  *
  * @see com.ibm.websphere.objectgrid.ObjectMap#getIndex
  */
  String getName();

  /**
  * Gets an index proxy object for performing index lookup operations. The
  * caller must cast the object returned to either a MapIndex or MapRangeIndex
  * object to perform the lookup operations.
  *
  * @param map The MapIndexInfo object required for maintaining the index.
  * .
  * @return a proxy to either an object that implements MapIndex or MapRangeIndex.
  */
  Object getIndexProxy(MapIndexInfo map);

  /**
  * This is called by the core to allow the index to be updated as the result
  * of changes applied to map during the commit cycle of a transaction.
  * Use  the {@link LogElement#getType()} method to determine what operation is
  * required to for updating the index. Use the {@link LogElement#getBeforeImage()}
  * to get the value object that existed prior to committing transaction applying
  * a change to the map and the {@link LogElement#getAfterImage()} to get the value
  * object after the committing transaction applied the change to the map entry.
  *

* Note, the {@link #undoBatchUpdate(TxID, LogSequence)} method may be called
* later to undo these changes if an exception occurs that causes committing
* transactions to be rolled back instead.
 *
 * @param txid The transaction for the changes.
 * @param sequence The log sequence that contains changes from transaction.
 *
 * @throws ObjectGridRuntimeException is a failure occurs that requires transaction
 * to be rolled back.
 */
 void doBatchUpdate(TxID txid, LogSequence sequence) throws
 ObjectGridRuntimeException;

/**
 * This is called by the core to undo any changes made to the index as a result of
 * a prior call to the {@link #doBatchUpdate(TxID, LogSequence)} method. This
 * method is called when an exception or error condition that requires all
 * changes made by transaction to be rolled back. For this reason, the
 * implementation of this method should catch all Throwable and continue with
 * next LogElement in the LogSequence until all LogElements are processed so that
 * as many changes to the index is undone as possible. An ObjectGridException
 * should only be thrown after processing the entire LogSequence and this method
 * was unable to successfully undo 1 or more changes in the LogSequence.
 *

* Use the {@link LogElement#getUndoType()} method to determine what operation is
* required to undo any change made to the index. Use the
* {@link LogElement#getBeforeImage()} to get the value object that existed prior
* to committing transaction applying a change to the map and the {@link
* LogElement#getAfterImage()} to get the value object after the committing
* transaction applied the change to the map entry.
*
* @param txid The transaction for the changes.
* @param sequence The log sequence that contains changes from transaction.
*
*/
void undoBatchUpdate( TxID txid, LogSequence sequence) throws ObjectGridException;
}
```

The setAttributeName and getName methods are straightforward and revealed by their names. The other methods require more attention.

## getIndexProxy method

The getIndexProxy method should return an index proxy object that implements either the MapIndex interface, the MapRangeIndex interface, or a custom Index interface. The implementation of the index proxy object is the core part of the index plug-in.

A MapIndexInfo object is passed into this method to provide transactional change information. This is the data that is visible only to the current transaction that invokes the getIndexProxy method. The index proxy object can use this MapIndexInfo object to search this transactional data.

The following is the definition of the MapIndexInfo interface:

```
/**
* This interface is used to provide an index with detailed change information
* for a specific Map in a transaction.
*/
public interface MapIndexInfo
{
  /**
  * An index contains the key values of a set of map entries that have a
  * a specific attribute value.  This method returns the ObjectMap the
  * index is referring to ObjectMap that the index is associated with.
  *
  * @return ObjectMap this index is associated with.
  */
  ObjectMap getMap();

  /**
  * Returns the set of all changes made by the current transaction to the
  * ObjectMap that is returned by the {@link #getMap()} method.
  *
  * @param includeRemoved must be set to true to include LogElement.DELETE types
  *        in the list returned by this method.
  *
  * @return a List of LogElement created for each ObjectMap entry that was
  *         either inserted, updated, or removed by current transaction.
  *
  * @throws ObjectGridRuntimeException
  */
  List getTransactionChanges(boolean includeRemoved) throws
  ObjectGridRuntimeException;

  /**
  * This returns the set of changes as they apply to a particular set of keys
  * in the current transaction for the ObjectMap that is returned by the
  * {@link #getMap()} method.  If a key has not been referenced
  * in the transaction then null is returned.
  *
  * @param keys The list of keys for which the data is required.
  * @return a List of LogElement corresponding to the keys or null if the keys
 * was not referenced.
  *
  * @throws ObjectGridRuntimeException
  *
  * @see com.ibm.websphere.objectgrid.plugins.LogElement
  * @see com.ibm.websphere.objectgrid.ObjectMap
  */
  List getTransactionChanges(List keys) throws ObjectGridRuntimeException;

}
```

The getIndexProxy method is designed to support the getIndex(String name) method of the ObjectMap interface. The returned index proxy object will be the one returned by the the getIndex method of the ObjectMap. For example, the application invokes the getIndex method of the ObjectMap, which then invokes this getIndexProxy method and returns the Object that is returned by this getIndexProxy method. The application has to cast the returned index proxy object to an application index interface, such as MapIndex, MapRangeIndex, or another customized index interface.

The following code example illustrates some index proxy object implementations that can be returned by the getIndexProxy method:

```
/**
* A class used to return a proxy to this map index
* so that applications can perform query operations
* using MapIndex interface.
*/
class Proxy implements MapIndex
{
  /**
  * The MapIndexInfo object associated with this index proxy object.
  */
  protected MapIndexInfo ivMap;

  /**
  * Maximum number of retries when concurrent transactions
  * modify index during a query operation.
  */
  protected static final int RETRY_LIMIT = 10;

  /**
  * EQUAL comparator to use.
  */
  final protected ProxyEQComparator ivEQ = new ProxyEQComparator();

  final protected ProxyGTComparator ivGT = new ProxyGTComparator();

  final protected ProxyRangeComparator ivRange = new ProxyRangeComparator();

  /**
  * Construct a proxy object for a given ObjectMap.
  *
  * @param map
  *           is the MapIndexInfo object.
  */
  Proxy(MapIndexInfo map)
  {
    ivMap = map;
  }

  /**
  *
  * @see com.ibm.websphere.objectgrid.plugins.index.MapIndex#findAll
  */
  public Iterator findAll(Object attributeValue) throws FinderException
  {
    if ( attributeValue == null )
    {
      throw new IllegalArgumentException(
      "the attributeValue must be a non null reference" );
    }

    // Use the greater than comparator for range check.
    ivEQ.ivAttribute = (Comparable) attributeValue;

    ArrayList resultList = null;
    int retryCount = 0;
    boolean retry;
    do
    {
      // Variables that need to be re-initialize each time thru loop.
      retry = false;
      resultList = new ArrayList();

      // Use index to obtains the Set of keys for map entries that
      // contain the specified attribute value.
      Set s = (Set) index.get( attributeValue );
      Set keySet = processSet( s, ivEQ );
      if ( keySet != null )
      {
        resultList.addAll( keySet );
      }
      else
      {
        // Whoops, another transaction modified Set obtained from index
        // while the above was iterating over the Set to perform the
        // addAll operation. Therefore, we need to retry by starting
        // over beginning with getting Set from index to pickup changes
        // from the transaction that just modified the Set.
        ++retryCount;
        if ( retryCount >= RETRY_LIMIT )
        {
          throw new FinderException( "query retry limit exceeded" );
        }
        retry = true;
      }
    } while ( retry );

    // Return iterator for result list created by above loop.
    Iterator result = resultList.iterator();
    return result;
  }


  /**
  * Process a Set obtained from index to determine if which of the keys
  * are for map entries that meet the query select criteria.
  *
  * @param s
  *            is the Set of key values for entries in BackingMap
  *            this index is built over. A null reference indicates only
  *            changes from current transaction needs to be processed.
  *
  * @param comparator
  *            is the comparator to use for making range check.
  *
  * @return Set of keys that met the select criteria or a null reference
  *             if a Exception occurs while iterating over the Set.
  *
  * @throws FinderException
  *             if an error condition prevents processing of Set
  *             from being performed.
  */
  protected Set processSet(Set s, ProxyComparator comparator)
  throws FinderException {
    HashSet resultSet =  new HashSet();

    //...
    //process the s Set, use comparator and prepare the resultSet.
    //...

    return resultSet;
  }

} // end class Proxy


/**
* A class used to return a proxy to this map index so that applications can
* perform query operations using MapRangeIndex interface.
*/
class RangeProxy extends Proxy implements MapRangeIndex
{
  /**
  * Various comparators needed by proxy to perform the
  * the range check of attribute value.
  */
  final private ProxyLTComparator ivLT = new ProxyLTComparator();
  final private ProxyLEComparator ivLE = new ProxyLEComparator();
  final private ProxyGEComparator ivGE = new ProxyGEComparator();

  /**
  * Index is a synchronized SortedMap.
  */
  final SortedMap ivIndexSortedMap;

  /**
  * Construct a MapRangeIndex proxy.
  */
  RangeProxy(MapIndexInfo map)
```

```
{
  super( map );
  ivIndexSortedMap = (SortedMap) index;
}

/**
* Execute query operation on a specified Map and ProxyComparator object.
*
* @param map
*           is a subset of the index to perform finder operation on.
* @param proxyComparator
*           is a comparator used to perform range check on attribute value.
*
* @return Set of keys required to be returned by finder method.
*
* @throws FinderException
*            is any failure occurs during execution of the query.
*/
private Set executeQuery(Map map, ProxyComparator proxyComparator)
throws FinderException {
  HashSet resultList = null;
  int retryCount = 0;
  boolean retry;
  do
  {
    // Variables that need to be re-initialize each time thru loop.
    retry = false;
    resultList = new HashSet();

    // Use index to obtains the Set of keys for map entries that
    // contain the specified attribute value.
    SortedMap treeMap = (SortedMap) index;
    Collection values = map.values();
    if ( values.isEmpty() )
    {
      // Nothing in range currently in index, so we only
      // need to check changes from current transaction.
      Set keySet = processSet( null, proxyComparator );
      if ( keySet != null )
      {
        resultList.addAll( keySet );
      }
    }
    else
    {
      // Index does contains some keys in range, so we need to query
      // both index entries as well as current transaction changes.
      Iterator iter = values.iterator();
      while ( iter.hasNext() )
      {
        Set keySet;
        try
        {
          Set s = (Set) iter.next();
          keySet = processSet( s, proxyComparator );
        }
        catch (ConcurrentModificationException  e)
        {
          // Indicate unable to get keySet.
          keySet = null;
        }

        if ( keySet != null )
        {
          resultList.addAll( keySet );
        }
        else
        {
          ++retryCount;
          if ( retryCount >= RETRY_LIMIT )
          {
            throw new FinderException( "query retry limit exceeded" );
          }
          retry = true;
        }
      }
    }

  } while ( retry );

  return resultList;
}

/*
* (non-Javadoc)
*
* @see com.ibm.websphere.objectgrid.plugins.index.MapRangeIndex#findGreater
*/
public Iterator findGreater(Object attributeValue)
throws FinderException {
  if ( attributeValue == null )
  {
    throw new IllegalArgumentException(
    "the attributeValue must be a non null reference" );
  }

  // Use the greater than comparator for range check.
  ivGT.ivAttribute = (Comparable) attributeValue;
  SortedMap tailMap = ivIndexSortedMap.tailMap( attributeValue );
  Set resultSet = executeQuery( tailMap, ivGT );
  Iterator result = resultSet.iterator();

  return result;
}

/*
* (non-Javadoc)
*
* @see com.ibm.websphere.objectgrid.plugins.index.MapRangeIndex#findGreaterEqual
*/
public Iterator findGreaterEqual(Object attributeValue)
throws FinderException {
  if ( attributeValue == null )
  {
    throw new IllegalArgumentException(
    "the attributeValue must be a non null reference" );
  }

  // Use the greater than comparator for range check.
  ivGE.ivAttribute = (Comparable) attributeValue;
  SortedMap tailMap = ivIndexSortedMap.tailMap( attributeValue );
  Set resultSet = executeQuery( tailMap, ivGE );
  Iterator result = resultSet.iterator();

  return result;
}

/*
* (non-Javadoc)
*
* @see com.ibm.websphere.objectgrid.plugins.index.MapRangeIndex#findLess
*/
public Iterator findLess(Object attributeValue) throws FinderException
{
  if ( attributeValue == null )
  {
    throw new IllegalArgumentException(
    "the attributeValue must be a non null reference" );
  }

  // Use the greater than comparator for range check.
  ivLT.ivAttribute = (Comparable) attributeValue;
  SortedMap headMap = ivIndexSortedMap.headMap( attributeValue );
  Set resultSet = executeQuery( headMap, ivLT );
  Iterator result = resultSet.iterator();

  return result;
}

/*
* (non-Javadoc)
*
* @see com.ibm.websphere.objectgrid.plugins.index.MapRangeIndex#findLessEqual
*/
public Iterator findLessEqual(Object attributeValue) throws FinderException
{
  if ( attributeValue == null )
  {
    throw new IllegalArgumentException(
    "the attributeValue must be a non null reference" );
  }

  // Use the greater than comparator for range check.
  ivLE.ivAttribute = (Comparable) attributeValue;
  Set resultSet;
  int retryCount = 0;
  boolean retry;
  do
  {
    // re-initialize for each retry that occurs.
    retry = false;
    SortedMap headMap = ivIndexSortedMap.headMap( attributeValue );
    resultSet = executeQuery( headMap, ivLE );
    Set s = (Set) ivIndexSortedMap.get( attributeValue );
    ivEQ.ivAttribute = (Comparable) attributeValue;
    Set equalSet = processSet( s, ivEQ );
    if ( equalSet != null )
    {
      if ( ! equalSet.isEmpty() )
      {
        resultSet.addAll( equalSet );
      }
    }
    else
    {
      // Whoops, another transaction modified index while processSet
      // was executing.  Therefore, we need to retry the entire query.
      ++retryCount;
      retry = true;
      if ( retryCount >= RETRY_LIMIT )
      {
        throw new FinderException( "query retry limit exceeded" );
      }
    }
  } while ( retry );

  // Return iterator for result list created by above loop.
  Iterator result = resultSet.iterator();

  return result;
}
```

```
  /*
  * (non-Javadoc)
  *
  * @see com.ibm.websphere.objectgrid.plugins.index.MapRangeIndex#findRange
  */
  public Iterator findRange(Object lowAttributeValue, Object highAttributeValue)
  throws FinderException {
    if ( lowAttributeValue == null )
    {
      throw new IllegalArgumentException(
      "the lowAttributeValue must be a non null reference" );
    }

    if ( highAttributeValue == null )
    {
      throw new IllegalArgumentException(
      "the highAttributeValue must be a non null reference" );
    }

    // Use the greater than comparator for range check.
    ivRange.ivLowAttribute = (Comparable) lowAttributeValue;
    ivRange.ivHighAttribute = (Comparable) highAttributeValue;
    SortedMap subMap = ivIndexSortedMap.
    subMap( lowAttributeValue, highAttributeValue );
    Set resultSet = executeQuery( subMap, ivRange );
    Iterator result = resultSet.iterator();

    return result;
  }

}

/**
* Abstract base class used for determining if attribute value is in range.
*/
abstract class ProxyComparator
{
  abstract boolean inRange(Object attribute);
}

/**
* Performs less than range check.
*/
class ProxyLTComparator extends ProxyComparator
{
  Comparable ivAttribute;

  boolean inRange(Object attribute)
  {
    if ( attribute == null )
    {
      return false;
    }
    else
    {
      Comparable attr = (Comparable) attribute;
      return ( attr.compareTo( ivAttribute ) < 0 );
    }
  }
}

/**
* Performs less than or equal range check.
*/
class ProxyLEComparator extends ProxyComparator
{
  Comparable ivAttribute;

  boolean inRange(Object attribute)
  {
    if ( attribute == null )
    {
      return false;
    }
    else
    {
      Comparable attr = (Comparable) attribute;
      return ( attr.compareTo( ivAttribute ) <= 0 );
    }
  }
}

/**
* Performs equal range check.
*/
class ProxyEQComparator extends ProxyComparator
{
  Comparable ivAttribute;

  boolean inRange(Object attribute)
  {
    if ( attribute == null )
    {
      return false;
    }
    else
    {
      return ( ivAttribute.compareTo( attribute ) == 0 );
    }
  }
}

/**
* Performs greater than range check.
*/
class ProxyGTComparator extends ProxyComparator
{
  Comparable ivAttribute;

  boolean inRange(Object attribute)
  {
    if ( attribute == null )
    {
      return false;
    }
    else
    {
      Comparable attr = (Comparable) attribute;
      return ( attr.compareTo( ivAttribute ) > 0 );
    }
  }
}

/**
* Performs greater than or equal range check.
*/
class ProxyGEComparator extends ProxyComparator
{
  Comparable ivAttribute;

  boolean inRange(Object attribute)
  {
    if ( attribute == null )
    {
      return false;
    }
    else
    {
      Comparable attr = (Comparable) attribute;
      return ( attr.compareTo( ivAttribute ) >= 0 );
    }
  }
}

/**
* Performs lowAttribute <= attribute < highAttribute range check.
*/
class ProxyRangeComparator extends ProxyComparator
{
  Comparable ivLowAttribute;

  Comparable ivHighAttribute;

  boolean inRange(Object o)
  {
    if ( o == null )
    {
      return false;
    }

    Comparable attribute = (Comparable) o;
    if ( attribute.compareTo( ivLowAttribute ) < 0 )
    {
      return false; // attribute < ivLowAttribute
    }
    else
    {
      // ivLowAttribute <= attribute
      if ( attribute.compareTo( ivHighAttribute ) < 0 )
      {
        return true;  // ivLowAttribute <= attribute < ivHighAttribute
      }
      else
      {
        return false; // attribute >= ivHighAttribute
      }
    }
  }
}
```

## doBatchUpdate and undoBatchUpdate methods

The doBatchUpdate and undoBatchUpdate methods are critical methods in the MapIndexPlugin interface. The doBatchUpdate method is invoked as the result of changes applied to map during the commit cycle of a transaction. The undoBatchUpdate method is used to undo any changes made to the index as a result of a prior call to the doBatchUpdate method. It is called when an exception or error condition occurs that requires all changes made by transaction to be rolled back. Both methods are given the current TxID and a list of changes for this Map. They should iterate over the changes and process them.

The following code example shows how to implement these two methods and supporting methods.

```
/**
* The synchronized Map used as the index implementation where
* the attribute value object is the key and a Java Set is the value.
* A Set member is the key of a BackingMap entry that matches attribute value.
```

```
*/
Map index; //<Object attribute, Set keys>

public void doBatchUpdate(TxID txid, LogSequence sequence)
throws ObjectGridRuntimeException
{
  Iterator iter = sequence.getAllChanges();
  while ( iter.hasNext() )
  {
    LogElement elem = (LogElement) iter.next();
    Object key = elem.getCacheEntry().getKey();
    LogElement.Type doType = elem.getType();
    if ( doType == LogElement.INSERT )
    {
      Object newAttribute = getAttribute( elem.getAfterImage() );
      insertIntoIndex( key, newAttribute );
    }
    else if ( doType == LogElement.UPDATE )
    {
      Object newAttribute = getAttribute( elem.getAfterImage() );
      Object oldAttribute = getAttribute( elem.getBeforeImage() );
      updateIndex( key, oldAttribute, newAttribute );
    }
    else if ( doType == LogElement.DELETE )
    {
      Object oldAttribute = getAttribute( elem.getBeforeImage() );
      removeFromIndex( key, oldAttribute );
    }
    else if ( doType == LogElement.EVICT )
    {
      Object beforeImage = elem.getBeforeImage();
      if ( beforeImage != null )
      {
        Object oldAttribute = getAttribute( beforeImage );
        removeFromIndex( key, oldAttribute );
      }
    }
  }
}

public void undoBatchUpdate(TxID txid, LogSequence sequence)
throws ObjectGridException
{
  int errors = 0;
  Iterator iter = sequence.getAllChanges();
  while ( iter.hasNext() )
  {
    try
    {
      LogElement elem = (LogElement) iter.next();
      Object key = elem.getCacheEntry().getKey();
      LogElement.Type undoType = elem.getUndoType();
      if ( undoType == LogElement.INSERT )
      {
        Object newAttribute = getAttribute( elem.getBeforeImage() );
        insertIntoIndex( key, newAttribute );
      }
      else if ( undoType == LogElement.UPDATE )
      {
        Object oldAttribute = getAttribute( elem.getAfterImage() );
        Object newAttribute = getAttribute( elem.getBeforeImage() );
        updateIndex( key, oldAttribute, newAttribute );
      }
      else if ( undoType == LogElement.DELETE )
      {
        Object oldAttribute = getAttribute( elem.getAfterImage() );
        removeFromIndex( key, oldAttribute );
      }
    }
    catch ( Throwable t )
    {
      ++errors;
    }
  }

  if ( errors > 0 )
  {
    throw new ObjectGridException( errors
    + " exceptions occurred during rollback of index changes.");
  }
}

/**
* Extracts the attribute from a specified value Object.
*
* @param value  The value Object.
*
* @return attribute from the value Object, which may be a null reference.
*
* @throws ObjectGridRuntimeException is thrown if any exception occurs
*         attempting to extract the attribute value from the value Object.
*/
private Object getAttribute(Object value) throws ObjectGridRuntimeException
{
  try
  {
    Object attribute = null;
    if ( value != null )
    {
      Method m = getAttributeMethod( value );
      attribute = getAttributeMethod.invoke( value, emptyArray );
    }

    return attribute;
  }
  catch ( InvocationTargetException e )
  {
    Throwable t = e.getTargetException();
    throw new ObjectGridRuntimeException( "Caught unexpected Throwable", t );
  }
  catch ( Throwable t )
  {
    throw new ObjectGridRuntimeException( "Caught unexpected Throwable", t );
  }
}

private void updateIndex(Object key, Object oldAttribute, Object newAttribute)
{
  // Was attributed changed by the update?
  if ( newAttribute != null && oldAttribute != null &&
  oldAttribute.equals( newAttribute ) )
  {
    // Nope, then nothing needs to be changed in index.
    return;
  }

  // Unless we restrict Loader to only access tables with non-nullable columns,
  // we have to handle the possibility that the attribute is null.
  Set oldKeys = null;
  if ( oldAttribute != null )
  {
    // Remove oldAttribute from index entry.
    oldKeys = (Set) index.get( oldAttribute );
    if ( oldKeys != null )
    {
      oldKeys.remove( key );
      if ( oldKeys.isEmpty() )
      {
        index.remove( oldAttribute );
      }
    }
  }

  // Unless we restrict Loader to only access tables with non-nullable columns,
  // we have to handle the possibility that the attribute is null.
  Set keys = null;
  if ( newAttribute != null )
  {
    keys = (Set) index.get( newAttribute );

    // Add newAttribute to index.
    if ( keys == null )
    {
      // Since different transactions can be updating different BackingMap
      // entries and multiple map entries can have same attribute value,
      // we need to use a synchronized Set object to ensure only
      // one transaction at a time can make changes to the Set.
      keys = Collections.synchronizedSet( new HashSet() );
      index.put( newAttribute, keys );
    }

    // Add key for this map entry to the Set of keys for the new attribute value.
    keys.add( key );
  }
}

private void insertIntoIndex( Object key, Object newAttribute )
{
  // Unless we restrict Loader to only access tables with non-nullable columns,
  // we have to handle the possibility that the attribute is null.
  if ( newAttribute != null )
  {
    Set keys = (Set) index.get( newAttribute );
    if ( keys == null )
    {
      // Since different transactions can be updating different
      // Map entries and multiple map entries can have same attribute
      // value, we need to use a synchronized Set object to ensure only
      // one transaction at a time can make changes to the Set.
      keys = Collections.synchronizedSet( new HashSet() );
      index.put( newAttribute, keys );
    }

    // Add key for this map entry to the Set of keys for the new attribute value.
    keys.add( key );
  }
}

private void removeFromIndex(Object key, Object oldAttribute )
{
  // Extract the old attribute value
  Set oldKeys = null;

  // Unless we restrict Loader to only access tables with non-nullable columns,
  // we have to handle the possibility that the attribute is null.
  if ( oldAttribute != null )
  {
    oldKeys = (Set) index.get( oldAttribute );
    if ( oldKeys != null )
    {
      oldKeys.remove( key );
      if ( oldKeys.isEmpty() )
      {
        index.remove( oldAttribute );
      }
    }
  }
}
```

## Application index interfaces

Application index interfaces are designed to support query methods. Currently, there are two application index interfaces defined: MapIndex and MapRangeIndex.

**MapIndex**

The MapIndex is a simple index for looking up objects by an attribute value. It allows any attribute value on a Map to be indexed. This lets the application quickly find all objects in the Map that have a specific attribute value. The following is the definition of the MapIndex interface:

```
/**
* This is an abstract index that can be created on an empty Map. The
* index can be used to perform efficient look ups and possibly other
* operations such as relational operations on an attribute in a Map.
* The MapIndex is provided with all update events and maintains an
* index that can be used to issue simple queries against the index
* later. The index could use an index defined callback to make an
* index on composite attributes.
*/
public interface MapIndex
{
  /**
  * Returns the Keys for the entries that have the specified attribute
 * value.
  *
  * @param attributeValue
  *            a non-null reference to the attribute value to search for.
  *
  * @return A list of the keys for the entries with that attribute.
  *
  * @throws IllegalArgumentException if attributeValue argument is null.
  * @throws FinderException is thrown if exception or retry limit is
 *     reached when concurrent transactions updating the index
 *    prevent findAll from completing.
  */
  Iterator findAll(Object attributeValue) throws FinderException;

}
```

**MapRangeIndex**

The MapRangeIndex is a simple index for looking up objects with an attribute value in a certain range. It allows any attribute value on a Map to be indexed. It differs from MapIndex in that it allows queries using value ranges and value comparison operations. This allows queries to find all objects with an attribute value less or greater than a specific value. The following is the definition of the MapRangeIndex interface:

```
/**
* This is an index that allows comparison type searches.
*/
public interface MapRangeIndex extends MapIndex
{
  /**
  * This find all keys with entries with an attribute greater than the
 * specified value.
  *
  * @param attributeValue  is the low endpoint of range excluding the
 * low attribute value.
  *
  * @return The set of keys with values greater than the attribute.
  *
  * @throws IllegalArgumentException if attributeValue argument is null.
  * @throws FinderException is thrown if exception or retry
  * limit is reached
  * when concurrent transactions updating the index prevent
 * findAll from completing.
  */
  Iterator findGreater(Object attributeValue) throws FinderException;

  /**
  * This find all keys with entries with an attribute greater or equal
 * to the specified value.
  *
  * @param attributeValue is the low endpoint of range including the
 * low attribute value.
  *
  * @return The set of keys with attributes meeting the criteria
  *
  * @throws IllegalArgumentException if attributeValue argument is null.
  * @throws FinderException is thrown if exception or retry
 * limit is reached
  * when concurrent transactions updating the index prevent findAll
  * from completing.
  */
  Iterator findGreaterEqual(Object attributeValue) throws FinderException;


  /**
  * This find all keys with entries with an attribute less than the
 * specified value.
  *
  * @param attributeValue  is the high endpoint of range excluding high
 * endpoint value.
  *
  * @return The set of keys with attributes meeting the criteria
  *
  * @throws IllegalArgumentException if attributeValue argument is null.
  * @throws FinderException is thrown if exception or retry limit
 * is reached
  * when concurrent transactions updating the index prevent
 *  findAll from completing.
  */
  Iterator findLess(Object attributeValue) throws FinderException;

  /**
  * This find all keys with entries with an attribute less than or equal
 * to the specified value.
  *
  * @param attributeValue is the high endpoint of range including high
 * endpoint value.
  *
  * @return The set of keys with attributes meeting the criteria
  *
  * @throws IllegalArgumentException if attributeValue argument is null.
  * @throws FinderException is thrown if exception or retry limit
 * is reached
  * when concurrent transactions updating the index prevent
 * findAll from completing.
  */
  Iterator findLessEqual(Object attributeValue) throws FinderException;

  /**
  * This returns all keys for the entries with the attribute inclusively
```

```
     * within the specified range such that lowAttributeValue <= attribute
     * < highAattributeValue.
      *
      * @param lowAttributeValue  is the low endpoint of range including the
     * low attribute value.
      * @param highAttributeValue is the high endpoint of range excluding
     * high attribute value.
      *
      * @return The list of keys with entries in that range, in ascending order.
      *
      * @throws IllegalArgumentException if either lowAttributeValue or
     * highAttributeValue
      *         argument is null or lowAttributeValue > highAttributeValue.
      * @throws FinderException is thrown if exception or retry limit is reached
      *         when concurrent transactions updating the index prevent
     *      findAll from completing.
      */
      Iterator findRange(Object lowAttributeValue, Object highAttributeValue)
      throws FinderException;

    }
```

Applications need to cast the obtained index object from the getIndex method of ObjectMap instance to the desired application index interface. If the index plug-in is designed to support the MapRangeIndex interface, the index object can be cast to the MapRangeIndex type; otherwise, it should be cast to the MapIndex type.

You can define a customized application index interface. Implement the custom application index as the index proxy object that can be returned by the getIndexProxy method of the MapIndexPlugin. Cast the obtained index object from the getIndex method of ObjectMap instance to this customized application index interface and use it.

## Adding static index plug-ins

There are two approaches to add static index plug-ins into BackingMap configuration: XML configuration and programmatic configuration. The following example illustrates the XML Configuration approach:

```
<backingMapPluginCollection id="person">
 <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.
  plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="CODE"
   description="index name" />
  <property name="RangeIndex" type="boolean" value="true" description="true
   for MapRangeIndex" />
  <property name="AttributeName" type="java.lang.String" value="employeeCode"
   description="attribute name" />
 </bean>
</backingMapPluginCollection>
```

The BackingMap interface has two methods that can be used to add static index plug-ins: addMapIndexPlugin and setMapIndexPlugins method. The following is the definition of these two methods.

```
/**
* This method adds an index plugin to this Map. We assume the index implementation
* was constructed
* with the name of the attribute to index. The name of the index is specified when
* the index is constructed.
*
* Note, to avoid an {@link IllegalStateException}, this method must be called
* prior to {@link ObjectGrid#initialize()} method.  Also, keep in mind that the
* {@link ObjectGrid#getSession()} method implicitly calls the
* {@link ObjectGrid#initialize()} method if it has yet to be called by the
* application.
*
* @param index The index implementation.
*
* @throws IndexAlreadyDefinedException This index already exists.
* @throws IllegalStateException if this method is called after the
*         {@link ObjectGrid#initialize()} method is called.
*/
public void addMapIndexPlugin(MapIndexPlugin index)
throws IndexAlreadyDefinedException;

/**
* This method sets the list of MapIndexPlugin objects for this BackingMap.
* If the BackingMap already has a List of MapIndexPlugin objects,
* that list is replaced by the List passed as
* an argument to the current invocation of this method.
*
* Note, to avoid an {@link IllegalStateException}, this method must be called
* prior to {@link ObjectGrid#initialize()} method.  Also, keep in mind that the
* {@link ObjectGrid#getSession()} method implicitly calls the
* {@link ObjectGrid#initialize()} method if it has yet to be called by the
* application.
*
* @param indexList  A non-null reference to a List of {@link MapIndexPlugin}
* objects.
*
* @throws IllegalArgumentException is thrown if indexList is null
*         or the indexList contains an object that is not
*         an instanceof {@link MapIndexPlugin}.
*/
public void setMapIndexPlugins(List indexList );
```

The following snippet of code illustrates the programmatic configuration approach:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;

ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid ivObjectGrid = ogManager.createObjectGrid( "grid" );
BackingMap personBackingMap = ivObjectGrid.getMap("person");
//use the builtin HashIndex class as the index plugin class.
HashIndex mapIndexPlugin = new HashIndex();
mapIndexPlugin.setName("CODE");
mapIndexPlugin.setAttributeName("EmployeeCode");
mapIndexPlugin.setRangeIndex(true);
personBackingMap.addMapIndexPlugin(mapIndexPlugin);
```

## Using Static Indices

After a static index plug-in has been added to a BackingMap configuration and the containing ObjectGrid instance has been initialized, applications can get the index object by name from the ObjectMap instance for the BackingMap. Cast the index object to the application index interface. Index operations supported by the application index interface can now run. The following is the definition of the getIndex method of the ObjectMap interface:

```
/**
* This returns a reference to the named index that can be used with this Map.
* This index cannot be shared between threads and works on the same rules as
* Session. The returned value should be cast to the right index interface
* such as MapIndex or MapRangeIndex or a custom index interface such as geo
* spatial index.
```

```
*
* @param name The index name
*
* @return A reference to the index, it must be cast to the appropriate index
* interface.
*
* @throws IndexUndefinedException if the index is not defined on the BackingMap
* @throws IndexNotReadyException if the index is not ready
* @throws UnsupportedOperationException if the map is a distributed map
*/
Object getIndex(String name)
throws IndexUndefinedException, IndexNotReadyException,
 UnsupportedOperationException;
```

The following snippet of code illustrates the way to get and use static indices:

```
Session session = ivObjectGrid.getSession();
ObjectMap map = session.getMap("person ");
MapRangeIndex codeIndex = (MapRangeIndex) m.getIndex("CODE");
Iterator iter = codeIndex.findLessEqual(new Integer(15));
while (iter.hasNext()) {
  Object key = iter.next();
  Object value = map.get(key);
}
```

## Adding and removing dynamic indices

Dynamic indices can be created on and removed from a BackingMap instance programmatically at anytime. A dynamic index differs from a static index in that the dynamic index can be created even after the containing ObjectGrid instance has been initialized. Unlike the static indexing, the dynamic indexing is an asynchronous process and needs to be in ready state before serving its purpose. The way to get and use the dynamic indices is same as static indices. If a dynamic index is no longer needed, it can be removed. The BackingMap interface has methods to create and remove dynamic indices. The following is the definition of these methods:

```
/**
* Create a dynamic index on the BackingMap
*
* @param name the name of the index. The name can not be null.
* @param isRangeIndex Indicate whether to create a MapRangeIndex or a MapIndex.
*                    If set to true, the index will be type of MapRangeIndex.
* @param attributeName The name of the attribute to be indexed.
*                    The attributeName can not be null.
* @param dynamicIndexCallback The callback that will invoke upon dynamic
*  index events.
*                    The dynamicIndexCallback is optional and can be null.
*
* @throws IndexAlreadyDefinedException if a MapIndexPlugin with the specified
* name already exists.
* @throws UnsupportedOperationException if the map is a distributed map.
*
*/
public void createDynamicIndex(String name, boolean isRangeIndex,
String attributeName, DynamicIndexCallback cb)
throws IndexAlreadyDefinedException, UnsupportedOperationException;


/**
* Create a dynamic index on the BackingMap.
*
* @param index The index implementation. The index can not be null.
* @param dynamicIndexCallback The callback that will invoke upon dynamic
* index events.
*        The dynamicIndexCallback is optional and can be null.
```

```
*
* @throws IndexAlreadyDefinedException if a MapIndexPlugin with the
* specified name already exists.
* @throws UnsupportedOperationException if the map is a distributed map.
*/
public void createDynamicIndex(MapIndexPlugin index, DynamicIndexCallback
dynamicIndexCallback)
throws IndexAlreadyDefinedException, UnsupportedOperationException;


/**
* remove a dynamic index from the BackingMap
*
* @param name the name of the index. The name can not be null.
*
* @throws IndexUndefinedException if a MapIndexPlugin with the specified name
*         does not exists.
* @throws OperationNotSupportedException if the map is a distributed map.
*/
public void removeDynamicIndex(String name) throws IndexUndefinedException;
```

The following snippet of code illustrates the programmatic approach of creating, using and removing a dynamic index:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;

ObjectGridManager ogManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid" );
BackingMap bm = og.getMap("person");
og.initialize();

//create index after ObjectGrid initialization without DynamicIndexCallback.
bm.createDynamicIndex("CODE", true, "employeeCode", null);

try {
  //If not using DynamicIndexCallback, need to wait for the Index to be ready.
  //The waiting time depends on the current size of the map
  Thread.sleep(3000);
} catch (Throwable t) {
  //...
}

//Once the index is ready, applications can try to get application index
//interface instance.
//Applications have to find a way to ensure the index is ready to use,
//if not using DynamicIndexCallback interface.
//The following demonstrates the way to wait for the index to be ready
//The total waiting time should consider the size of the map

Session session = og.getSession();
ObjectMap m = session.getMap("person");
MapRangeIndex codeIndex = null;

int counter = 0;
int maxCounter = 10;
boolean ready = false;
while(!ready && counter < maxCounter){
 try {
    counter++;
    codeIndex = (MapRangeIndex) m.getIndex("CODE");
    ready = true;
  } catch (IndexNotReadyException e) {
    //implies index is not ready, ...
    System.out.println("Index is not ready. continue to wait.");
    try {
      Thread.sleep(3000);
    } catch (Throwable tt) {
      //...
    }
  } catch (Throwable t) {
    //unexpected exception
    t.printStackTrace();
  }
}

if(!ready){
  System.out.println("Index is not ready.  Need to handle this situation.");
}

//use the index to peform queries
//Refer to MapIndex or MapRangeIndex interface for supported operations.
//The object attribute that the index created on is the EmployeeCode.
//Assuming the EmployeeCode attribute is Integer type, which should be
//the data type of the parameter passed into index operations.

Iterator iter = codeIndex.findLessEqual(new Integer(15));

//remove the dynamic index when no longer needed

bm.removeDynamicIndex("CODE");
```

## DynamicIndexCallback interface

The DynamicIndexCallback interface is designed for applications that wish to get notifications at the indexing events of ready, error, or destroy. It is an optional parameter for the createDynamicIndex() method of the BackingMap. With a registered DynamicIndexCallback instance, applications can execute business logic upon receiving notification of an indexing event. For example, the ready event means the index is ready for use. When a notification for this event is received, an application can try to get the application index interface instance and use it. The following shows the definition of the DynamicIndexCallback Interface:

```
/**
* This is the callback interface for dynamic indexing process.
* If applications wish to get notification at the event of ready, error,
* or destroy, they can implement this callback interface and register with
* the dynamic indexing process when creating dynamic index.
*
*/

public interface DynamicIndexCallback {

  /**
  * This callback method will be invoked when the dynamic index is ready.
  *
  * @param indexName
  *           The index name
  *
  */
  public void ready(String indexName);

  /**
  * Invoked when the dynamic indexing process encounters an unexpected error.
  *
  * @param indexName the index name
  * @param t A Throwable object that causes the error situation in dynamic
 * indexing process.
  */
    public void error(String indexName, Throwable t);
  /**
  * This callback method will be invoked when the dynamic index is removed
  *
  * @param indexName
  *          The index name
  */
  public void destroy(String indexName);
}
```

The following code snippet illustrates the use of the DynamicIndexCallback interface:

```
BackingMap personBackingMap = ivObjectGrid.getMap("person");
DynamicIndexCallback callback = new DynamicIndexCallbackImpl();
personBackingMap.createDynamicIndex("CODE", true, "employeeCode", callback);


class DynamicIndexCallbackImpl implements DynamicIndexCallback {
  public DynamicIndexCallbackImpl() {
  }

  public void ready(String indexName) {
    System.out.println("DynamicIndexCallbackImpl.ready() -> indexName = " +
    indexName);

    ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
    ObjectGrid og = ogManager.createObjectGrid( "grid" );
    Session session = og.getSession();
    ObjectMap map = session.getMap("person");
    MapIndex codeIndex = (MapIndex) map.getIndex("CODE");
    Iterator iter = codeIndex.findAll(codeValue);
  }

  public void error(String indexName, Throwable t) {
        System.out.println("DynamicIndexCallbackImpl.error() ->
     indexName = " + indexName);
        t.printStackTrace();
   }

  public void destroy(String indexName) {
    System.out.println("DynamicIndexCallbackImpl.destroy() -> indexName = " +
    indexName);
  }
}
```

## Performance considerations

Although one of the main objectives of the indexing feature is to improve overall BackingMap performance, there are some factors that need to be considered before using this feature. If indexing is not used properly, the performance of the application might be compromised.

- The number of concurrent write transactions. Index processing can occur every time a transaction writes data into a BackingMap. Performance will degrade if there are many transactions writing data into the map concurrently when an application attempts index query operations.
- The size of the resultset returned by a query operation. As the size of the resultset increases, the query performance declines. One experiment has shown that performance degrades when the size of the resultset is 15% or more of the BackingMap.
- The number of indices built over the same BackingMap. Each index consumes system resources. As the number of the indices built over the BackingMap increases, the performance declines.

The conclusion is the indexing function can improve the BackingMap performance dramatically in certain environments. An ideal case for indexing is when the BackingMap is read-mostly, the query resultset is of a smaller percentage of the BackingMap entries, and only few indices are built over the BackingMap.

## ObjectGrid configuration

ObjectGrid can be configured to run in a distributed environment or as a local cache available only within a single JVM. A local ObjectGrid can be configured programmatically or with an ObjectGrid XML file. The ObjectGrid XML file is the place to define ObjectGrids, BackingMaps and their respective plug-ins.

A local ObjectGrid can be migrated to a distributed environment. To configure a distributed ObjectGrid, a cluster XML must be provided in conjunction with the ObjectGrid XML. The cluster XML file defines the servers in the ObjectGrid topology and how the ObjectGrid data is partitioned and replicated across the servers. This section details how to configure local and distributed ObjectGrids.

### Local ObjectGrid configuration

To configure a local ObjectGrid, see “Local ObjectGrid configuration.”

### Distributed ObjectGrid

To configure a distributed ObjectGrid, see “Distributed ObjectGrid configuration” on page 261.

## Local ObjectGrid configuration

A local ObjectGrid can be configured programmatically or with XML. The ObjectGridManager is the entry point for both means of configuration.

Several methods on the ObjectGridManager exist that can be used to create a local ObjectGrid. For a complete description of each method, see “ObjectGridManager interface” on page 87.

Use the following topics to configure a local ObjectGrid:

- “Basic ObjectGrid configuration” discusses how to create a very simple XML file with one ObjectGrid and one BackingMap defined.
- “Complete ObjectGrid configuration” on page 250 defines each element and attribute of the XML file and discusses how to achieve the same result as the XML file programmatically.
- “Mixed mode ObjectGrid configuration” on page 260 describes how to use a combination of XML and programmatic configuration methods.

### Basic ObjectGrid configuration

This topic demonstrates how to create a very simple ObjectGrid XML file, the `bookstore.xml` file, with one ObjectGrid and one BackingMap defined.

The first few lines of the file are the required header for each ObjectGrid XML file. The following XML defines the *bookstore* ObjectGrid with the *book* BackingMap:

*bookstore.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
   <objectGrid name="bookstore">
    <backingMap name="book" />
   </objectGrid>
  </objectGrids>
</objectGridConfig>
```

The XML file is sent to the ObjectGridManager interface to create an ObjectGrid instance based on the file. The following code snippet validates the `bookstore.xml` file against the XML schema, and creates the *bookstore* ObjectGrid. The newly created ObjectGrid instance is not cached.

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.createObjectGrid("bookstore",
new URL("file:etc/test/bookstore.xml"), true, false);
```

The following code accomplishes the same task without XML. Use this code to programmatically define a BackingMap on an ObjectGrid. This code creates the book BackingMap on the bookstoreGrid ObjectGrid:

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.createObjectGrid
 ("bookstore", false);
BackingMap bookMap = bookstoreGrid.defineMap("book");
```

## Complete ObjectGrid configuration

This topic is a complete guide to configuring an ObjectGrid. Each element and attribute of the XML file is defined. Sample XML files are given, along with code that accomplishes the same task programmatically.

The following XML file, `bookstore.xml`, is referred to throughout this topic. The elements and attributes of this file are described in detail following this example.

*bookstore.xml* file

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="bookstore">
   <bean id="ObjectGridEventListener"
     className="com.company.organization.MyObjectGridEventListener" />
   <backingMap name="books" pluginCollectionRef="collection1" />
  </objectGrid>
 </objectGrids>
 <backingMapPluginCollections>
  <backingMapPluginCollection id="collection1">
   <bean id="Evictor"
    classname="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor">
    <property name="maxSize" type="int" value="321" />
   </bean>
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

### objectGridConfig element

**Number of occurrences:** one

**Child elements:** objectGrids element and backingMapPluginCollections element

The objectGridConfig element is the top level element of the XML file. It must be written in the XML document as shown in the preceding example. This element sets up the namespace of the file and the schema location. The schema is defined in the `objectGrid.xsd` file. ObjectGrid looks for this file in the root directory of the ObjectGrid Java archive (JAR) files.

## objectGrids element

**Number of occurrences:** one

**Child element:** objectGrid element

The objectGrids element is a container for all the objectGrid elements. In the `sample1.xml` file, the objectGrids element contains one objectGrid that has the name `bookstore`.

## objectGrid element

**Number of occurrences:** one to many

**Child elements:** bean element and backingMap element

Use the objectGrid element to define an ObjectGrid in an XML file. Each of the attributes on the objectGrid element corresponds to a method on the ObjectGrid interface.

```
<objectGrid
(1) name="objectGridName"
(2) securityEnabled="true|false"
(3) authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS|
            AUTHORIZATION_MECHANISM_CUSTOM"
(4) permissionCheckPeriod="permission check period"
(5) txTimeout="seconds"
/>
```

**Attributes:**

1. **name** attribute (required): Specifies the name that is assigned to the ObjectGrid. If this attribute is missing, the XML validation fails.
2. **securityEnabled** attribute (optional, default is `false`): Setting this attribute to `true` enables security for the ObjectGrid. Enabling security on the ObjectGrid level means enabling the access authorizations to the data in the map. By default, security is disabled.
3. **authorizationMechanism** attribute (optional, defaults to `AUTHORIZATION_MECAHNISM_JAAS`): Sets the authorization mechanism for this ObjectGrid. This attribute can be set to one of two values: `AUTHORIZATION_MECHANISM_JAAS` or `AUTHORIZATION_MECHANISM_CUSTOM`. Set to `AUTHORIZATION_MECHANISM_CUSTOM` when using a custom MapAuthorization plug-in. This setting takes effect if the securityEnabled attribute is set to `true`.
4. **permissionCheckPeriod** (optional, defaults to `0`): Specifies an integer value in seconds that indicates how often to check the permission that is used to allow a client access. If the attribute value is `0`, then every get, put, update, remove, or evict method call asks the authorization mechanism, either JAAS authorization or custom authorization, to check if the current subject has permission. A value greater than `0` indicates the number of seconds to cache a set of permissions before returning to the authorization mechanism to refresh. This setting takes effect if the securityEnabled attribute is set to `true`. For more details, see “ObjectGrid security” on page 131.

5. **txTimeout** (optional, defaults to `0`): The amount of time in seconds, that a transaction is allowed for completion. If a transaction does not complete in this amount of time, the transaction is marked for roll back and a `TransactionTimeoutException` exception results. If the value is set to `0`, transactions never time out.

The following example XML file, the `bookstoreObjectGridAttr.xml` file, demonstrates one way to configure the attributes of an objectGrid. In this example, security is enabled, the authorization mechanism is set to JAAS, and the permission check period is set to 45 seconds.

*bookstoreObjectGridAttr.xml* file

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="bookstore" securityEnabled="true"
   authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS"
   permissionCheckPeriod="45">
  </objectGrid>
 </objectGrids>
</objectGridConfig>
```

The following code demonstrates the programmatic approach to achieve the same configuration as the `bookstoreObjectGridAttr.xml` file in the previous example.

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory
.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.createObjectGrid("bookstore", false);

bookstoreGrid.setSecurityEnabled();
bookstoreGrid.setAuthorizationMechanism(
SecurityConstants.AUTHORIZATION_MECHANISM_JAAS);
bookstoreGrid.setPermissionCheckPeriod(45);
```

### backingMap element

**Number of occurrences:** zero to many

**Child elements:** none

The backingMap element is used to define a BackingMap on an ObjectGrid. Each of the attributes on the backingMap element corresponds to a method on the BackingMap interface.

```
<backingMap
(1) name="backingMapName"
(2) readOnly="true|false"
(3) pluginCollectionRef="reference to backingMapPluginCollection"
(4) numberOfBuckets="number of buckets"
(5) preloadMode="true|false"
(6) lockStrategy="OPTIMISTIC|PESSIMISTIC|NONE"
(7) numberOfLockBuckets="number of lock buckets"
(8) lockTimeout="lock timeout"
(9) copyMode="COPY_ON_READ_AND_COMMIT|COPY_ON_READ|
  COPY_ON_WRITE|NO_COPY"
(10) valueInterfaceClassName="value interface class name"
(11) copyKey="true|false"
(12) nullValuesSupported="true|false"
(13) ttlEvictorType="CREATION_TIME|LAST_ACCESS_TIME|NONE"
(14) timeToLive="time to live"
/>
```

**Attributes:**

1. **name** attribute (required): Specifies the name that is assigned to the BackingMap. If this attribute is missing, XML validation fails.
2. **readOnly** attribute (optional, defaults to `false`): Setting this attribute to `true` makes a read-only BackingMap. Setting the attribute to `false` makes a read-write BackingMap. If a value is not specified, the default of read-write results.
3. **pluginCollectionRef** attribute (optional): Specifies a reference to a backingMapPluginCollection plug-in. The value of this attribute must match the id attribute of a backingMapCollection plug-in. Validation fails if no matching id exists. This reference is designed to be an easy way to reuse BackingMap plug-ins.
4. **numberOfBuckets** attribute (optional, defaults to `503`): The number of buckets to be used by the BackingMap. The BackingMap uses a hash map for its implementation. If a lot of entries exist in the BackingMap more buckets lead to better performance because the risk of collisions is lower as the number of buckets grows. More buckets also lead to more concurrency.
5. **preloadMode** attribute (optional, defaults to `false`): Sets the preload mode if a Loader plug-in is set for this BackingMap. If the attribute is set to `true`, the Loader.preloadMap(Session, BackingMap) method is invoked asynchronously. Otherwise it blocks running the method when loading data so that the cache is unavailable until preload completes. Preloading occurs during ObjectGrid initialization.
6. **lockStrategy** attribute (optional, defaults to `OPTIMISTIC`): Sets the LockStrategy that is used for the BackingMap. The locking strategy determines if the internal ObjectGrid lock manager is used whenever a map entry is accessed by a transaction. This attribute can be set to one of three values: `OPTIMISTIC`, `PESSIMISTIC`, or `NONE`.

   `OPTIMISTIC` is typically used for a map that does not have a Loader plug-in, the map is mostly read, and the locking is not provided by persistence manager using the objectGrid as a side cache or by the application. For the optimistic locking strategy, an exclusive lock is obtained on a map entry being inserted, updated, or removed at commit time. The lock ensures version information cannot be changed by another transaction while the transaction being committed is performing an optimistic versioning check.

   `PESSIMISTIC` is typically used for a map that does not have a Loader plug-in and locking is not provided by a persistence manager using the objectGrid as a side cache, by a Loader plug-in, or by the application. The pessimistic locking strategy is used when the optimistic approach fails too often because update transactions frequently collide on the same map entry. The optimistic approach can fail when the map is not mostly read, or a large number of clients access a small map.

   `NONE` indicates that internal LockManager use is not needed because concurrency control is provided outside of the ObjectGrid, either by the persistence manager using ObjectGrid as a side cache, application, or by Loader plug-in that uses database locks to control concurrency.
7. **numberOfLockBuckets** attribute (optional, defaults to `383`): Sets number of lock buckets that are used by the lock manager for this BackingMap. When the lockStrategy attribute is set to `OPTIMISTIC` or `PESSIMISTIC`, a lock manager is created for the BackingMap. The lock manager uses a hash map to keep track of entries that are locked by one or more transactions. If many entries exist, then more lock buckets lead to better performance because the risk of collisions is lower as the number of buckets grows. More lock buckets also

lead to more concurrency. When the lockStrategy attribute is set to NONE, no lock manager is used by this BackingMap. In this case, setting numberOfLockBuckets attribute is not needed.

8. **lockTimeout** attribute (optional, defaults to 15): Sets the lock timeout that is used by the lock manager for this BackingMap. When the lockStrategy attribute is set to OPTIMISTIC or PESSIMISTIC, a lock manager is created for the BackingMap. To prevent deadlocks from occurring, the lock manager has a default timeout value for waiting for a lock to be granted. If this timeout limit is exceeded, a LockTimeoutException exception occurs. The default value of 15 seconds is sufficient for most applications, but on a heavily loaded system, a timeout might occur when no deadlock exists. In that case, this method can be used to increase the lock timeout value from the default to prevent false timeout exceptions from occurring. When the lock strategy is NONE, no lock manager is used by this BackingMap. In this case, setting the lockTimeout attribute is not needed.
9. **copyMode** attribute (optional, defaults to COPY_ON_READ_AND_COMMIT): The copyMode attribute determines if a get operation of an entry in the BackingMap returns the actual value, a copy of the value, or a proxy for the value. The copyMode attribute can be set to one of four values: COPY_ON_READ_AND_COMMIT, COPY_ON_READ, COPY_ON_WRITE, or NO_COPY.

   The COPY_ON_READ_AND_COMMIT mode ensures that an application never has a reference to the value object that is in the BackingMap, and instead the application is always working with a copy of the value that is in the BackingMap.

   The COPY_ON_READ mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when a transaction is committed. To preserve integrity of BackingMap data, the application promises to destroy every reference that it has to an entry after the transaction is committed. This mode results in a ObjectMap.get method returning a copy of the value instead of a reference to the value to ensure that changes that are made by the application to the value does not affect the BackingMap value until the transaction is committed.

   The COPY_ON_WRITE mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when ObjectMap.get method is called for the first time by a transaction for a given key. Instead, the ObjectMap.get method returns a proxy to the value instead of a direct reference to the value object. The proxy ensures that a copy of the value is not made unless the application calls a set method on the value interface.

   The NO_COPY mode allows an application to promise that it never modifies a value object that is obtained using an ObjectMap.get method in exchange for performance improvements. If this mode is used, the value is not copied.
10. **valueInterfaceClassName** attribute (optional): When the copyMode attribute is set to COPY_ON_WRITE, a valueInterfaceClassName attribute is required. It is ignored for all other modes. Copy on write uses a proxy when ObjectMap.get method calls are made. The proxy ensures that a copy of the value is not made unless the application calls a set method on the class that is specified as the valueInterfaceClassName attribute.
11. **copyKey** attribute (optional, defaults to false): This attribute determines if the key needs to be copied when a map entry is created. Copying the key object allows the application to use the same key object for each ObjectMap operation. Setting to true copies the key object when a map entry is created.
12. **nullValuesSupported** attribute (optional, defaults to true): Supporting null values means that a null value can be put in a map. If set to true, null values

are supported in the ObjectMap, otherwise null values are not supported. If null values are supported, a `get` operation that returns `null` could mean that the value is null or that the map does not contain the passed-in key.

13. **ttlEvictorType** attribute (optional, defaults to `NONE`): The ttlEvictorType attribute determines how the expiration time of a BackingMap entry is computed. This attribute can be set to one of three values: `CREATION_TIME`, `LAST_ACCESS_TIME`, or `NONE`.

    `NONE` indicates that an entry has no expiration time and is allowed to live in the BackingMap until the application explicitly removes or invalidates the entry.

    `CREATION_TIME` indicates that an entry expiration time is the sum of the creation time of the entry plus the timeToLive attribute value.

    `LAST_ACCESS_TIME` indicates that an entry expiration time is the sum of the last access time of the entry plus the timeToLive attribute value.
14. **timeToLive** attribute (optional, defaults to `0`): The time to live of each map entry, in seconds. The default value of `0` means that the map entry lives forever, or until the application explicitly removes or invalidates the entry. If the attribute is not `0`, the TTL evictor is used to evict the map entry based on this value.

The following XML file, the `bookstoreBackingMapAttr.xml` file, demonstrates a sample backingMap configuration. This example makes use of all the optional attributes except the pluginCollectionRef attribute. For an example that shows how to use the pluginCollectionRef, see “backingMapPluginCollection element” on page 259.

*bookstoreBackingMapAttr.xml* file

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="bookstore">
   <backingMap name="book" readOnly="true" numberOfBuckets="641"
    preloadMode="false" lockStrategy="OPTIMISTIC"
    numberOfLockBuckets="409" lockTimeout="30" copyMode="COPY_ON_WRITE"
    valueInterfaceClassName=
     "com.ibm.websphere.samples.objectgrid.CounterValueInterface"
    copyKey="true" nullValuesSupported="false"
    ttlEvictorType="LAST_ACCESS_TIME" timeToLive="3000" />
  </objectGrid>
 </objectGrids>
</objectGridConfig>
```

The following sample code demonstrates the programmatic approach to achieve the same configuration as the `bookstoreBackingMapAttr.xml` file in the preceding example:

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.createObjectGrid("bookstore", false);

BackingMap bookMap = bookstoreGrid.defineMap("book");
bookMap.setReadOnly(true);
bookMap.setNumberOfBuckets(641);
bookMap.setPreloadMode(false);
bookMap.setLockStrategy(LockStrategy.OPTIMISTIC);
bookMap.setNumberOfLockBuckets(409);
bookMap.setLockTimeout(30);

// when setting copy mode to COPY_ON_WRITE, a valueInterface class is required
```

```
bookMap.setCopyMode(CopyMode.COPY_ON_WRITE,
com.ibm.websphere.samples.objectgrid.CounterValueInterface.class);
bookMap.setCopyKey(true);
bookMap.setNullValuesSupported(false);
bookMap.setTtlEvictorType(TTLType.LAST_ACCESS_TIME);
bookMap.setTimeToLive(3000); // set time to live to 50 minutes
```

## bean element

**Number of occurrences (within the objectGrid element)**: zero to many

**Number of occurrences (within the backingMapPluginCollection element)**: zero to many

**Child element**: property element

Use the bean element to define plug-ins. Plug-ins can be attached to ObjectGrids and BackingMaps.

The ObjectGrid plug-ins:
- TransactionCallback plug-in
- ObjectGridEventListener plug-in
- SubjectSource plug-in
- MapAuthorization plug-in
- SubjectValidation plug-in

The BackingMap plug-ins:
- Loader plug-in
- ObjectTransformer plug-in
- OptimisticCallback plug-in
- Evictor plug-in
- MapEventListener plug-in
- MapIndex plug-in

*bean element attributes*

```
<bean
(1) id="TransactionCallback|ObjectGridEventListener|SubjectSource|
  MapAuthorization|SubjectValidation|Loader|ObjectTransformer|
  OptimisticCallback|Evictor|MapEventListener|MapIndexPlugin"
(2) className="class name"
/>
```

1. **id** attribute (required): Specifies the type of plug-in to create. For a bean that is a child element of the objectGrid element, the valid values are `TransactionCallback`, `ObjectGridEventListener`, `SubjectSource`, `MapAuthorization`, and `SubjectValidation` plug-ins. For a bean that is a child element of the backingMapPluginCollection element, the valid values are `Loader`, `ObjectTransformer`, `OptimisticCallback`, `Evictor`, and `MapEventListener` plug-ins. Each of the valid values for the id attribute represent an interface.
2. **className** attribute (required): Specifies the name of the class to instantiate to create the plug-in. The class must implement the plug-in type interface.

The following `bean.xml` file sample demonstrates how to use the bean element to configure plug-ins. In this XML file, an ObjectGridEventListener plug-in is added to the bookstore ObjectGrid. The className attribute for this bean is the com.ibm.websphere.objectgrid.plugins.builtins.TranPropListener class. This class implements the com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener interface as required.

A BackingMap plug-in is also defined in the following `bookstoreBean.xml` file sample. An evictor plug-in is added to the book BackingMap. Because the bean id is *Evictor*, the className attribute must specify a class that implements the com.ibm.websphere.objectgrid.plugins.Evictor interface. The com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor class implements this interface. The backingMap references its plug-ins using the pluginCollectionRef attribute. See "BackingMap interface" on page 105 for more information on how to add plug-ins to a BackingMap.

*bookstoreBean.xml* file

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="bookstore">
   <bean id="ObjectGridEventListener"
     className="com.ibm.websphere.objectgrid.plugins.builtins.TranPropListener" />
    <backingMap name="book" pluginCollectionRef="bookPlugins" />
  </objectGrid>
 </objectGrids>
 <backingMapPluginCollections>
  <backingMapPluginCollection id="bookPlugins">
   <bean id="Evictor"
     classnName="com.ibm.websphere.objectGrid.plugins.builtins.LRUEvictor" />
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

The following code demonstrates the programmatic approach to achieve the same configuration as the previous `bookstoreBean.xml` file.

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.createObjectGrid
 ("bookstore", false);
TranPropListener tranPropListener = new TranPropListener();
bookstoreGrid.addEventListener(tranPropListener);

BackingMap bookMap = bookstoreGrid.defineMap("book");
Evictor lruEvictor = new
 com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor();
bookMap.setEvictor(lruEvictor);
```

### property element

**Number of occurrences:** zero to many

**Child element:** none

The property element is used to add properties to plug-ins. The name of the property corresponds to a set method on the className attribute of the bean that contains the property.

*property element attributes*

```
<property
(1) name="name"
(2) type="java.lang.String|boolean|java.lang.Boolean|int|
  java.lang.Integer|double|java.lang.Double|byte|
  java.lang.Byte|short|java.lang.Short|long|
  java.lang.Long|float|java.lang.Float|char|
  java.lang.Character"
(3) value="value"
(4) description="description"
/>
```

1. **name** attribute (required): Specifies the name of the property. The value that is assigned to this attribute must correspond to a set method on the class that is provided as the className attribute on the bean element. For example, if the className attribute of the bean is set to com.ibm.MyPlugin and the name of the property provided is `size`, then the com.ibm.MyPlugin class must have a setSize method.
2. **type** attribute (required): Specifies the type of the property. It is the type of the parameter that is passed to the set method that is identified by the name attribute. The valid values are the Java primitives, their `java.lang` counterparts, and `java.lang.String`. The name and type attributes must correspond to a method signature on the className attribute of the bean. For example, if name is `size` and type is `int`, then a `setSize(int)` method must exist on the class that is specified as the className attribute for the bean.
3. **value** attribute (required): Specifies the value of the property. This value is converted to the type that is specified by the type attribute, and is then used as a parameter in the call to the set method that is identified by the name and type attributes. The value of this attribute is not validated in any way. The plug-in implementor must verify that the value passed in is valid. The implementor can display an `IllegalArgumentException` exception in the set method if the parameter is not valid.
4. **description** attribute (optional): Use this attribute to write a description of the property.

The following `bookstoreProperty.xml` file demonstrates how to add a property element to a bean. In this example, a property with the name `maxSize` and type `int` is added to an Evictor. The com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor Evictor has a method signature that matches the setMaxSize(int) method. An integer value of `499` is passed to the setMaxSize(int) method on the com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor class.

*bookstoreProperty.xml file*

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="bookstore">
   <backingMap name="book" pluginCollectionRef="bookPlugins" />
  </objectGrid>
 </objectGrids>
 <backingMapPluginCollections>
  <backingMapPluginCollection id="bookPlugins">
   <bean id="Evictor"
    className="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor">
    <property name="MaxSize" type="int" value="449"
     description="The maximum size of the LRU Evictor" />
   </bean>
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

The following code achieves the same configuration as the `bookstoreProperty.xml` file:

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.createObjectGrid
 ("bookstore", false);

BackingMap bookMap = bookstoreGrid.defineMap("book");

LRUEvictor lruEvictor =
new com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor();
// if the XML file were used instead,
// the property that was added would cause the following call to be made
lruEvictor.setMaxSize(449);
bookMap.setEvictor(lruEvictor);
```

### backingMapPluginCollections element

**Number of occurrences**: zero to one

**Child element**: backingMapPluginCollection element

The backingMapPluginCollections element is a container for all the backingMapPluginCollection elements. In the `bookstore.xml` file, the backingMapPluginCollections element contains one backingMapPluginCollection element with the id `collection1`.

### backingMapPluginCollection element

**Number of occurrences**: zero to many

**Child element:** bean element

The backingMapPluginCollection element defines the BackingMap plug-ins. Each backingMapPluginCollection element is identified by its id attribute. Each backingMap element must reference its plug-ins using the pluginCollectionRef attribute on the backingMap element. If several BackingMaps exist that must have their plug-ins configured similarly, each of them can reference the same backingMapPluginCollection element.

*backingMapPluginCollection element attributes*

```
<backingMapPluginCollection
(1) id="id"
/>
```

1. **id** attribute (required): The identifier for the backingMapPluginCollection. Each id must be unique. The id is referenced by the pluginCollectionRef attribute of the backingMap element. If the value of a pluginCollectionRef attribute does not match the id of one backingMapPluginCollection element, XML validation fails. Any number of backingMap elements can reference a single backingMapPluginCollection element.

The following `bookstoreCollection.xml` file demonstrates how to use the backingMapPluginCollection element. In this file, three backingMap elements are defined. The `book` and `customer` BackingMaps both use the `collection1` backingMapPluginCollection. Each of these two BackingMaps have their own LRUEvictor evictor. The `employee` BackingMap references the `collection2` backingMapPluginCollection. This BackingMap has an LFUEvictor evictor set as an Evictor plug-in and the EmployeeOptimisticCallbackImpl class set as an OptimisticCallback plug-in.

*bookstoreCollection.xml file*

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="bookstore">
   <backingMap name="book" pluginCollectionRef="collection1" />
   <backingMap name="customer" pluginCollectionRef="collection1" />
   <backingMap name="employee" pluginCollectionRef="collection2" />
  </objectGrid>
 </objectGrids>
 <backingMapPluginCollections>
  <backingMapPluginCollection id="collection1">
   <bean id="Evictor"
     className="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor" />
  </backingMapPluginCollection>
  <backingMapPluginCollection id="collection2">
   <bean id="Evictor"
     className="com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor" />
   <bean id="OptimisticCallback"
     className="com.ibm.websphere.samples.objectgrid.
      EmployeeOptimisticCallBackImpl" />
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

The following code demonstrates how to programmatically achieve the same configuration as the `bookstoreCollection.xml` file.

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();

ObjectGrid bookstoreGrid = objectGridManager.createObjectGrid
 ("bookstore", false);
BackingMap bookMap = bookstoreGrid.defineMap("book");
LRUEvictor bookEvictor = new LRUEvictor();
bookMap.setEvictor(bookEvictor);

BackingMap customerMap = bookstoreGrid.defineMap("customer");
LRUEvictor customerEvictor = new LRUEvictor();
customerMap.setEvictor(customerEvictor);

BackingMap employeeMap = bookstoreGrid.defineMap("employee");
LFUEvictor employeeEvictor = new LFUEvictor();
employeeMap.setEvictor(employeeEvictor);
OptimisticCallback employeeOptCallback =
 new EmployeeOptimisticCallbackImpl();
employeeMap.setOptimisticCallback(employeeOptCallback);
```

## Mixed mode ObjectGrid configuration

ObjectGrid can be configured using a combination of XML configuration and programmatic configuration.

To accomplish a mixed configuration, first create an XML file to pass to the ObjectGridManager interface. After an ObjectGrid has been created based on the XML file, the ObjectGrid can be manipulated programmatically, as long as the

ObjectGrid.initialize() method has not been called. The ObjectGrid.getSession() method implicitly calls the ObjectGrid.initialize() method if it has not been called by the application.

### Example

Following is a demonstration of how to achieve a mixed mode configuration. The following `mixedBookstore.xml`, file is used.

*mixedBookstore.xml file*

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://ibm.com/ws/objectgrid/config/ ..objectGrid.xsd"
 xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="bookstore">
   <backingMap name="book" readOnly="true" numberOfBuckets="641"
    pluginCollectionRef="bookPlugins" />
  </objectGrid>
 </objectGrids>
 <backingMapPluginCollections>
  <backingMapPluginCollection id="bookPlugins">
   <bean id="Evictor"
     className="com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor" />
  </backingMapPluginCollection>
 </backingMapPluginCollections>
</objectGridConfig>
```

The following code snippet that shows the XML being passed to the ObjectGridManager, and the newly created ObjectGrid is further manipulated.

```
ObjectGridManager objectGridManager =
 ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.createObjectGrid("bookstore",
new URL("file:etc/test/document/mixedBookstore.xml"), true, false);
// at this point, we have the ObjectGrid that was defined in the XML

// now modify the BackingMap that was created and configured
BackingMap bookMap = bookstoreGrid.getMap("book");
// the XML set readOnly to true
// here the readOnly attribute is changed to false
bookMap.setReadOnly(false);

// the XML did not set nullValuesSupported, so
// it would default to true. Here the
// value is set to false
bookMap.setNullValuesSupported(false);

// get the Evictor that was set in the XML,
// and set its maxSize
LFUEvictor lfuEvictor = (LFUEvictor) bookMap.getEvictor();
lfuEvictor.setMaxSize(443);

bookstoreGrid.initialize();
// further configuration is not allowed
// to this ObjectGrid after the initialize call
```

## Distributed ObjectGrid configuration

To create a distributed ObjectGrid, a cluster XML file must be created and paired with an ObjectGrid XML file.

With the cluster XML and ObjectGrid XML files, you can start an ObjectGrid server.

Before creating a cluster XML file, create an ObjectGrid XML file as you would for a local ObjectGrid. For details on how to construct an ObjectGrid XML file see “Local ObjectGrid configuration” on page 249. The following `university.xml` file is used as the ObjectGrid XML for the examples in “Cluster configuration” on page 263.

*university.xml* file

```
<?xml version="1.0" encoding="UTF-8">
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config">
 <objectGrids>
  <objectGrid name="academics">
   <backingMap name="faculty" />
   <backingMap name="student" />
   <backingMap name="course" />
  </objectGrid>
  <objectGrid name="athletics">
   <backingMap name="athlete" />
   <backingMap name="equipment" />
  </objectGrid>
 </objectGrids>
</objectGridConfig>
```

Following is the `universityCluster.xml` cluster XML file that can be used with the `university.xml` file to start an ObjectGrid server. The `universityCluster.xml` file is a very basic cluster XML file with all of the optional XML attributes stripped away.

*universityCluster.xml* file

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config/cluster
  ../objectGridCluster.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config/cluster">

 <cluster name="universityCluster">
  <serverDefinition name="server1" host="lion.ibm.com" clientAccessPort="12501"
   peerAccessPort="12502" />
 </cluster>

 <objectgridBinding ref="academics">
  <mapSet name="academicsMapSet" partitionSetRef="partitionSet1">
   <map ref="faculty" />
   <map ref="student" />
   <map ref="course" />
  </mapSet>
 </objectgridBinding>

 <objectgridBinding ref="athletics">
  <mapSet name="athleticsMapSet" partitionSetRef="partitionSet1">
   <map ref="athlete" />
   <map ref="equipment" />
  </mapSet>
 </objectgridBinding>

 <partitionSet name="partitionSet1">
  <partition name="partition1" replicationGroupRef="replicationGroup1" />
 </partitionSet>

 <replicationGroup name="replicationGroup1">
  <replicationGroupMember serverRef="server1" priority="1" />
 </replicationGroup>
</clusterConfig>
```

Sample usage of many of the optional XML elements and attributes is in the "Cluster configuration" section.

## Cluster configuration

Each element and attribute of the cluster XML is described in this section. Examples are also provided that show how to use the cluster XML with the ObjectGrid XML to achieve a configuration. The `university.xml` file is used as the ObjectGrid XML for these examples.

### clusterConfig element

**Number of occurrences:** one

**Child elements:** cluster, objectgridBinding, partitionSet, and replicationGroup elements

The clusterConfig element is the top level element of the cluster XML file. It must be at the top of the file as demonstrated in the `universityCluster.xml` file. This element sets up the namespace of the file and the schema location. The schema is defined in the `objectGridCluster.xsd` file.

### cluster element

**Number of occurrences:** one

**Child elements:** serverDefinition, authenticator, and adminAuthorization elements

The cluster element is used to define an ObjectGrid cluster. Each of the servers in the cluster is defined within the cluster element. The cluster element is also used to define security and network-related attributes.

```
<cluster
(1) name="clusterName"
(2) securityEnabled="true|false"
(3) statisticsEnabled="true|false"
(4) statisticsSpec="statisticsSpecification"
(5) singleSignOnEnabled="true|false"
(6) loginSessionExpirationTime="seconds"
(7) adminAuthorizationEnabled="true|false"
(8) adminAuthorizationMechanism=""
(9) clientMaxRetries="numberOfRetries"
(10) clientMaxForwards="numberOfForwards"
(11) clientStartupRetries="numberOfRetries"
(12) clientRetryInterval="seconds"
(13) tcpConnectionTimeout="seconds"
(14) tcpMinConnections="numberOfConnections"
(15) tcpMaxConnections="numberOfConnections"
(16) tcpInactivityTimeout="seconds"
(17) tcpMaxWaitTime="seconds"
(18) peerHeartbeatInterval="seconds"
(19) peerTransportBufferSize="sizeInMBs"
(20) threadPoolMinSize="minThreads"
(21) threadPoolMaxSize="maxThreads"
(22) threadPoolInactivityTimeout="seconds"
(23) managementTimeout="seconds"
(24) threadsPerClientConnect="numberOfThreads"
/>
```

**Attributes:**

1. **name** attribute (required): This is the name that is assigned to the cluster. If this attribute is missing, XML validation fails.
2. **securityEnabled** attribute (optional, defaults to **false**): Enables security for the cluster when set to true. If it is set to false, cluster-wide security is disabled. For more information, see "ObjectGrid security" on page 131.
3. **statisticsEnabled** attribute (optional, defaults to **false**): Enables statistics for the cluster when set to true. When statistics is enabled, the statisticsSpec attribute is used to set the statistics specification.
4. **statisticsSpec** attribute (optional): Specifies the string that is used to set the statistics specification. This string determines what statistics are gathered.
5. **singleSignOnEnabled** attribute (optional, defaults to false): Setting the singleSignOnEnabled attribute to true allows a client to connect to any server after it has authenticated with one of the servers. When this attribute is set to false, a client must authenticate with each server before it is allowed to connect.
6. **loginSessionExpirationTime** attribute (optional): The amount of time in seconds before the login session expires. If the login session expires, the client must re-authenticate.
7. **adminAuthorizationEnabled** attribute (optional, defaults to false): This value is used to enable administrative authorization. If the value is true, all of the administrative tasks need authorization. The authorization mechanism used is specified by the value of adminAuthorizationMechanism attribute.
8. **adminAuthorizationMechanism** attribute (optional, defaults to AUTHORIZATION_MECHANISM_JAAS): This attribute indicates which authorization mechanism is used. ObjectGrid supports two authorization mechanisms: Java Authentication and Authorization Service (JAAS) and custom. The JAAS authorization mechanism uses the standard JAAS policy-based approach. To specify JAAS as the authorization mechanism, set the value to AUTHORIZATION_MECHANISM_JAAS. The custom authorization mechanism uses a user-plugged-in AdminAuthorization implementation. To specify a custom authorization mechanism, set the value to AUTHORIZATION_MECHANISM_CUSTOM. For more information on how these two mechanisms are used, see "ObjectGrid security" on page 131.
9. **clientMaxRetries** attribute (optional, defaults to 4): The maximum number of times a request for a server can be retried automatically when service is not available.
10. **clientMaxForwards** attribute (optional, defaults to 5): The maximum number of times a failed request is forwarded to another server.
11. **clientStartupRetries** attribute (optional, defaults to 8): The maximum number of times a request is automatically retried while waiting for server startup to complete. Because the high availability manager requires a significant amount of time to start, set this number to be sufficiently high. If the number is not large enough, client requests that are submitted before the server is completely started fail.
12. **clientRetryInterval** attribute (optional, defaults to 10): The time interval, in seconds, between a client retries. This is used for both the clientMaxRetries and clientStartupRetries attributes.
13. **tcpConnectionTimeout** attribute (optional, defaults to 180): The tcpConnectionTimeout attribute is the a socket connection timeout. The value is in seconds.
14. **tcpMinConnections** attribute (optional, defaults to 2): The minimum number of connections for the connection pool.

15. **tcpMaxConnections** attribute (optional, defaults to 20): The maximum number of connections for the connection pool.
16. **tcpInactivityTimeout** attribute (optional, defaults to infinity): The number of seconds of inactivity on a connection that must pass before the connection is removed from the connection pool.
17. **tcpMaxWaitTime** attribute (optional, defaults to 120): The maximum number of seconds a system waits for an available connection when all connections are in use and the number of connections has reached the tcpMaxConnections attribute value.
18. **peerHeartbeatInterval** attribute (optional, defaults to 120): The peerHeartbeatInterval attribute is the heartbeat interval used by the high availability manager. The value is in seconds.
19. **peerTransportBufferSize** attribute (optional, defaults to 10): The peerTransportBufferSize attributes represents the transportation message buffer size used by the high availability manager. This value is specified in megabytes.
20. **threadPoolMinSize** attribute (optional, defaults to 6): Specifies the minimum size of the high availability manager thread pool.
21. **threadPoolMaxSize** attribute (optional, defaults to 20): Specifies the maximum size of the high availability manager thread pool.
22. **threadPoolInactivityTimeout** attribute (optional, defaults to 6000): Specifies the thread inactivity timeout for the high availability manager thread pool. The timeout value is in seconds.
23. **managementTimeout** attribute (optional, defaults to 30): Several of the ObjectGrid MBean functions send messages to servers in the cluster to either gather information from or perform operations on the servers. The managementTimeout value dictates how long the client attempts to receive a message back from the server. If communication problems exist between client and server, or if the server is busy, the client retries for the amount of time that is specified by the managementTimeout value. The managementTimeout value is specified in seconds.
24. **threadsPerClientConnect** attribute (optional, defaults to 5): The number of threads that are created per ClientClusterContext. Each call to the connect method on the ObjectGridManager interface results in a new ClientClusterContext.

The following `universityClusterAttr.xml` file is a sample configuration that makes use of the various optional attributes on the cluster element. In this example, security is disabled. The various client, tcp, peer, and thread related attributes are also changed. The `universityClusterAttr.xml` is not a recommendation of what values to assign to attributes. It is an example of how to set attribute values.

*universityClusterAttr.xml* file

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config/cluster
  ../objectGridCluster.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config/cluster">

 <cluster name="universityCluster" securityEnabled="false" statisticsEnabled="true"
  statisticsSpec="map.all=enabled" singleSignOnEnabled="false"
  loginSessionExpirationTime="1800" adminAuthorizationEnabled="false"
     adminAuthorizationMechanism="AUTHORIZATION_MECHANISM_JAAS" clientMaxRetries="2"
  clientMaxForwards="2" clientStartupRetries="2" clientRetryInterval="5"
  tcpConnectionTimeout="160" tcpMinConnections="2" tcpMaxConnections="15"
  tcpInactivityTimeout="3600" tcpMaxWaitTime="160" peerHeartbeatInterval="130"
```

```
   peerTransportBufferSize="15" threadPoolMinSize="8" threadPoolMaxSize="25"
  threadPoolInactivityTimeout="6050" managementTimeout="60">

   <serverDefinition name="server1" host="lion.ibm.com" clientAccessPort="12501"
    peerAccessPort="12502" />
   <serverDefinition name="server2" host="tiger.ibm.com" clientAccessPort="12503"
    peerAccessPort="12504" />

 </cluster>

 <objectGridBinding ref="academics">
  <mapSet name="academicsMapSet" partitionSetRef="partitionSet1">
   <map ref="faculty" />
   <map ref="student" />
   <map ref="course" />
  </mapSet>
 </objectGridBinding>

 <objectGridBinding ref="athletics">
  <mapSet name="athleticsMapSet" partitionSetRef="partitionSet1">
   <map ref="athlete" />
   <map ref="equipment" />
  </mapSet>
 </objectGridBinding>

 <partitionSet name="partitionSet1">
  <partition name="partition1" replicationGroupRef="replicationGroup1" />
 </partitionSet>

 <replicationGroup name="replicationGroup1">
  <replicationGroupMember serverRef="server1" priority="1" />
 </replicationGroup>

</clusterConfig>
```

## serverDefinition element

**Number of occurrences:** one to many

**Child elements**: none

The serverDefinition element is used to define an ObjectGrid server. Each server runs in its own Java virtual machine (JVM) and requires two ports.

**Attributes:**

```
<serverDefinition
(1) name="serverName"
(2) host="hostname"
(3) clientAccessPort="portNumber"
(4) peerAccessPort="portNumber"
(5) traceSpec="traceSpecification"
(6) systemStreamToFileEnabled="true|false"
(7) workingDirectory="logsDirectory"
/>
```

1. **name** attribute (required): This is the name that is assigned to the server. If this attribute is missing, XML validation fails.
2. **host** attribute (required): The host name of the machine where the server JVM runs. Each machine can host multiple ObjectGrid servers. If this attribute is missing, XML validation fails.
3. **clientAccessPort** attribute (required): The port on the server that is used for client connections. If this attribute is missing, XML validation fails.

4. **peerAccessPort** attribute (required): The port on the server that is used for communication among ObjectGrid servers. If this attribute is missing, XML validation fails.
5. **traceSpec** attribute (optional, defaults to *=all=disabled): Setting the traceSpec attribute enables trace for the server using the specified string.
6. **systemStreamToFileEnabled** attribute (optional, defaults to **true**): If this attribute is set to true, System.out, System.err, and trace output steams go to a file. When this attribute is set to false, System.out goes to the stdout stream and System.err goes to the stderr stream. If trace is enabled, trace output goes to a file regardless of the value of the systemStreamToFileEnabled attribute.
7. **workingDirectory** attribute (optional): The workingDirectory attribute specifies where log files are written. If a workingDirectory attribute is not specified, logs are written to the current directory.

The `universityClusterServerAttr.xml` file demonstrates the use of the serverDefinition attributes. In this XML file, the server1 server is configured to run on the lion.ibm.com host. Port 12501 is used for client access to the server and port 12502 is used for server to server communications. Because the systemStreamToFileEnabled attribute is set to true, System.out, System.err and trace are output to a file within the directory specified with the workingDirectory attribute. In this example, the files are in the `/objectgrid/` directory. Because the traceSpec attribute is set to ″ObjectGrid=all=enabled″, all ObjectGrid related trace is captured and output to a file.

*universityClusterServerAttr.xml* file

```
<?xml version="1.0" encoding="UTF-8" ?>
<clusterConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config/cluster
  ../objectGridCluster.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config/cluster">

 <cluster name="universityCluster">
  <serverDefinition name="server1" host="lion.ibm.com" clientAccessPort="12501"
   peerAccessPort="12502" systemStreamToFileEnabled="true"
   workingDirectory="/objectgrid/" traceSpec="ObjectGrid=all=enabled" />
  <serverDefinition name="server2" host="tiger.ibm.com" clientAccessPort="12503"
   peerAccessPort="12504" />
 </cluster>
<objectGridBinding ref="academics">
  <mapSet name="academicsMapSet" partitionSetRef="partitionSet1">
   <map ref="faculty" />
   <map ref="student" />
   <map ref="course" />
  </mapSet>
 </objectGridBinding>

 <objectGridBinding ref="athletics">
  <mapSet name="athleticsMapSet" partitionSetRef="partitionSet1">
   <map ref="athlete" />
   <map ref="equipment" />
  </mapSet>
 </objectGridBinding>

 <partitionSet name="partitionSet1">
  <partition name="partition1" replicationGroupRef="replicationGroup1" />
 </partitionSet>

 <replicationGroup name="replicationGroup1">
  <replicationGroupMember serverRef="server1" priority="1" />
 </replicationGroup>
</clusterConfig>
```

## objectgridBinding element

**Number of occurrences:** one to many

**Child element:** mapSet element

The objectgridBinding element is used to bind the objectGrid elements in the ObjectGrid XML to the topology defined in the cluster XML. The value assigned to the ref attribute must match the name attribute of one of the objectGrid elements in the ObjectGrid XML. An objectGrid element from the ObjectGrid XML can be referenced in only one objectgridBinding in the cluster XML.

### Attributes

```
<objectgridBinding
(1) ref="objectGridReference"
(2) minThreadPoolSize="minSize"
(3) maxThreadPoolSize="maxSize"
/>
```

1. **ref** attribute (required): The ref attribute is used to reference an objectGrid element that is defined in the ObjectGrid XML file. Each objectgridBinding element must reference one of the objectGrid elements from the ObjectGrid XML. The ref attribute must match the name attribute of one of the objectGrid elements in the ObjectGrid XML.
2. **minThreadPoolSize** attribute (optional, defaults to `3`): The minThreadPoolSize attribute is the minimum number of threads that are allowed in the thread pool for each replication group member. The number of threads is controlled by a thread pool manager, but the number is not allowed to drop below the minThreadPoolSize value. In general, more threads allow the client to receive a response faster from the server. More threads also result in more contention. However, faster machines are able to handle additional concurrent threads effectively.
3. **maxThreadPoolSize** attribute (optional, defaults to 10): The maxThreadPoolSize attribute is the maximum number of threads that are allowed in the thread pool for each replication group member. The number of threads is controlled by a thread pool manager, but the number is not allowed to climb above the maxThreadPoolSize value. In general, more threads allow the client to receive a response faster from the server. More threads also result in more contention. However, faster machines are able to handle additional concurrent threads effectively.

The `universityClusterOGBinding.xml` file demonstrates how to use the objectgridBinding element and its attributes. In this example, one objectgridBinding element is defined. The objectgridBinding element is referring to the ″academics″ defined in the `university.xml` file. Notice that even though the ″athletics″ objectGrid is in the `university.xml` file, no objectgridBinding element is referring to the athletics ObjectGrid in the `universityClusterOGBinding.xml` file. The ″athletics″ ObjectGrid is not clustered because it is not included in the `universityClusterOGBinding.xml` filel. Only the ″academics″ ObjectGrid is created and clustered in this case because it is in the `university.xml` file and is referenced in the `universityClusterOGBinding.xml` file.

The minThreadPoolSize and maxThreadPoolSize attributes are also set in this example. The minThreadPoolSize value is set to `2`, and the maxThreadPoolSize value is set to `11`. The thread pool manager on each replication group member keeps the number of threads within these boundaries for all maps in this ObjectGrid.

*universityClusterOGBinding.xml* file

```
<?xml version="1.0" encoding="UTF-8" ?>
<clusterConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config/cluster
  ../objectGridCluster.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config/cluster">

 <cluster name="universityCluster">
  <serverDefinition name="server1" host="lion.ibm.com" clientAccessPort="12501"
   peerAccessPort="12502" />
 </cluster>

 <objectgridBinding ref="academics" minThreadPoolSize="2" maxThreadPoolSize="11">
  <mapSet name="academicsMapSet" partitionSetRef="partitionSet1">
   <map ref="faculty" />
   <map ref="student" />
   <map ref="course" />
  </mapSet>
 </objectgridBinding>

 <partitionSet name="partitionSet1">
  <partition name="partition1" replicationGroupRef="replicationGroup1" />
 </partitionSet>

 <replicationGroup name="replicationGroup1">
  <replicationGroupMember serverRef="server1" priority="1" />
 </replicationGroup>
</clusterConfig>
```

## mapSet element

**Number of occurrences**: one to many

**Child element**: map element

The mapSet element is used to group maps together. The maps within a mapSet are partitioned similarly. In a distributed ObjectGrid, each map must belong to one and only one mapSet.

### Attributes

```
<mapSet
(1) name="mapSetName"
(2) partitionSetRef="partitionSetReference"
(3) synchronousReplication="true|false"
(4) replicaReadEnabled="true|false"
(5) replicaDeliveryRate="deliveryRate"
(6) compression="true|false"
/>
```

1. **name** attribute (required): Specifies the name that is assigned to the mapSet.
2. **partitionSetRef** attribute (required): Each mapSet must be associated with a partitionSet through the partitionSetRef attribute. The partitionSetRef value must match the value of the name attribute of one of the partitionSet elements. By using the partitionSetRef attribute and its corresponding partitionSet, the maps in the mapSet are partitioned.
3. **synchronousReplication** attribute (optional, defaults to **false**): When this attribute is set to true, replication among replication group members occurs synchronously. When set to false, replication occurs asynchronously.
4. **replicaReadEnabled** attribute (optional, defaults to **false**): If the synchronousReplication value is set false and the replicaReadEnabled value is true, clients are allowed to read data from replicas. A best effort is made to

distribute read requests among the primary and its replicas. If the synchronousReplication attribute is set to true, the replicaReadEnabled attribute is ignored.

5. **replicaDeliveryRate** (optional, defaults to **1000**): The replicaDeliveryRate value represents the maximum number of records per LogSequence that are delivered to each replica.
6. **compressReplicationEnabled** attribute (optional, defaults to **true**): When compressReplicationEnabled is set to true, replication messages are compressed.

The `universityClusterMapSet.xml` file is a bit more complex than the previous XML file examples. In this file the academics ObjectGrid is divided into two map sets. The academicsMapSet1 map set contains the faculty and the course maps. These two maps are partitioned according to the partitionSet1 partitionSet. The replication settings for these maps are also the same because they are in the same mapSet.

The academics objectgridBinding also contains the academicsMapSet2 mapSet. This mapSet contains only the student map. The student map is partitioned differently than the maps in the academicsMapSet1 mapSet. The student map is partitioned according to the studentPSet partitionSet. Because the academicsMapSet2 has not explicitly stated values for the replication-related attributes, including the synchronousReplication, replicaReadEnabled, replicaDeliveryRate, and compressReplicationEnabled attributes, it is assigned the default values. This is another way that the behavior of the two mapSets within the academics objectgridBinding differ.

The athletics objectgridBinding contains the athleticsMapSet mapSet. Like the academicsMapSet1 mapSet in the academics objectgridBinding, it is partitioned according to the partitionSet1 partitionSet. The replication related attributes for this mapSet are set to the default values because they are not explicitly stated.

*universityClusterMapSet.xml* file

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config/cluster
  ../objectGridCluster.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config/cluster">
 <cluster name="universityCluster">
  <serverDefinition name="server1" host="lion.ibm.com" clientAccessPort="12501"
   peerAccessPort="12502" />
  <serverDefinition name="server2" host="tiger.ibm.com" clientAccessPort="12503"
   peerAccessPort="12504" />
 </cluster>

 <objectgridBinding ref="academics"
  <mapSet name="academicsMapSet1" partitionSetRef="partitionSet1"
   synchronousReplication="true" replicaReadEnabled="true"
   replicaDeliveryRate="1500" compressReplicationEnabled="true">
   <map ref="faculty" />
   <map ref="course" />
  </mapSet>
  <mapSet name="academicsMapSet2" partitionSetRef="studentPSet">
   <mapRef="student" />
  </mapSet>
 </objectGridBinding>

 <objectgridBinding ref="athletics">
  <mapSet name="athleticsMapSet" partitionSetREf="partitionSet1">
   <map ref="athlete" />
   <map ref="equipment" />
  </mapSet>
 </objectgridBinding>

 <partitionSet name="partitionSet1">
  <partition name="partition1" replicationGroupRef="replicationGroup1" />
 </partitionSet>

 <partitionSet name="studentPSet">
  <partition name="studentPartition1" replicationGroupRef="replicationGroup1" />
  <partition name="studentPartition2" replicationGroupRef="replicationGroup2" />
 </partitionSet>

 <replicationGroup name="replicationGroup1">
  <replicationGroupMember serverRef="server1" priority="1" />
 </replicationGroup>

 <replicationGroup name="replicationGroup2" minReplicas="1" maxReplicas="2">
  <replicationGroupMember serverRef="server1" priority="2" />
  <replicationGroupMember serverRef="server2" priority="1" />
 </replicationGroup>
</clusterConfig>
```

## map element

**Number of occurrences**: one to many

**Child elements**: none

Each map in a mapSet references one of the backingMap elements that is defined in the ObjectGrid XML file. When defining a distributed ObjectGrid, each backingMap within the objectGrid elment of the ObjectGrid XML must be referenced by a map in the cluster XML. Every map in a distributed ObjectGrid must belong to one and only one mapSet.

### Attributes

```
<map
(1)  ref="backingMapReference"
/>
```

1. **ref** attribute (required): A reference to a backingMap in the ObjectGrid XML. Each map in a mapSet must reference a backingMap from the ObjectGrid XML file. The value assigned to ref must match the name attribute of one of the backingMap elements in the ObjectGrid XML.

See the `universityClusterMapSet.xml` file for sample usage of the map element. Every backingMap from the academics objectGrid in the `university.xml` is referenced by a map in one and only one mapSet in the `universityClusterMapSet.xml`. The same is true of the athletics objectGrid. An `ObjectGridException` exception results if an objectgridBinding references an objectGrid from the ObjectGrid XML, but does not include all of its maps in a mapSet.

## partitionSet element

**Number of occurrences:** one to many

**Child element(s):** partition

The partitionSet element is used to define partitions for a mapSet. Each map in the mapSet is partitioned across the partitions of a partitionSet. A mapSet is associated with a partitionSet with the partitionSetRef attribute on the mapSet element. When only one partition is defined within a partitionSet, the data contained within the maps of an associated mapSet is not partitioned.

**Attributes**

```
<partition-set
(1) name="partitionSetName"
/>
```

1. **name** attribute (required): This attribute is used to assign a name to a partitionSet. The name of the partitionSet is referenced by the partitionSetRef attribute of the mapSet.

Refer to the `universityClusterMapSet.xml` file for sample usage of the partitionSet. In the `universityClusterMapSet.xml`, two partitionSets are defined: partitionSet1 and studentPSet. The partitionSet1 partitionSet has only one partition defined. Because only one partition is defined, any mapSet that makes use of the partitionSet1 partitionSet does have its data partitioned. Two mapSets exist in the `universityClusterMapSet.xml` file that are partitioned according to the partitionSet1 partitionSet. Through the partitionSetRef on the mapSet element, the academicsMapSet1 and athleticsMapSet mapSets are bound to the partitionSet1 partitionSet.

The studentPSet partitionSet contains two partitions. Any mapSet that uses this partitionSet has its map data divided across two partitions. In the `universityClusterMapSet.xml` file, the academicsMapSet2 mapSet uses the studentPSet partitionSet.

## partition element

**Number of occurrences**: one to many

**Child elements**: none

The partition element is used to define partitions within a partitionSet. Partitions are used to divide the data in the maps of a mapSet.

**Attributes**

```
<partition
(1) name="partitionName"
(2) replicationGroupRef="replicationGroupReference"
/>
```

1. **name** attribute (required): The name attribute is used to assign a name to a partition. A partition name must be unique within its partitionSet.
2. **replicationGroupRef** attribute (required): The replicationGroupRef attribute is used to associate a replicationGroup with a partition. The replicationGroupRef must match the name attribute of one of the replicationGroup elements.

Refer to the `universityClusterMapSet.xml` file for sample usage of the partition element. In the `universityClusterMapSet.xml` file, several partitions are defined. The partitionSet1 partitionSet has one partition named partition1. The studentPSet partitionSet contains two partitions: studentPartition1 and studentPartition2.

Each partition is associated with a replicationGroup through the replicationGroupRef attribute. In the `universityClusterMapSet.xml` file, the studentPartition1 partition is replicated across the replicationGroup1 replicationGroup. The studentPartition2 partition is replicated across the replicationGroup2 replicationGroup.

## replicationGroup element

**Number of occurrences**: one to many

**Child element**: replicationGroupMember element

A replicationGroup is used to define how a map or its partitions are replicated. The partitions of a map are replicated amongst the replication group members within a replicationGroup.

### Attributes

```
<replicationGroup
(1) name="replicationGroupName"
(2) minReplicas="minNumberOfReplicas"
(3) maxReplicas="maxNumberOfReplicas"
```

1. **name** attribute (required): The name attribute is used to assign a name to a replicationGroup.
2. **minReplicas** attribute (optional, defaults to **0** only one replicationGroupMember is in the replicationGroup; defaults to **1** more than one replicationGroupMember is in the replicationGroup): The minReplicas attribute is used to indicate how many replicationGroupMembers must be available before write access is allowed to map data in this replicationGroup. If the number of available replicas falls below the number of minReplicas specified, only read access to the maps is allowed. If minReplicas is set to 0, write access is still allowed on the primary even if all replicas are unavailable.

   To activate replication, at least two replication group members must be available, and the minReplicas attribute must be at least 1. It is important to be aware of how a replicationGroup behaves during the ″bringup″ stage of an ObjectGrid cluster. If you want map data to be available after starting only one server, then define a replicationGroup with only one replicationGroupMember. In a replicationGroup with only one replicationGroupMember, data is not replicated.

   Here are some rules for setting the minReplicas value.

   ```
   minReplicas >= 0
   minReplicas <= maxReplicas
   minReplicas <= # of members in the replicationGroup -1
   ```
3. **maxReplicas** attribute (optional, defaults to **0** if only one replicationGroupMember is in the replicationGroup; defaults to **1** more than one replicationGroupMember is in the replicationGroup): The maxReplicas attribute represents the maximum number of replicas that are activated in the replicationGroup. Within a replicationGroup, replication occurs amongst the number of maxReplicas specified if that many members are available. If maxReplicas is less than the number of replication group members in the group, the extra members are standbys; that is, they are dormant until one of the replicas becomes unavailable.

   Here are some rules for setting the maxReplicas value.

   ```
   maxReplicas >= 0
   maxReplicas >= minReplicas
   ```

Refer to the `universityClusterMapSet.xml` file for sample usage of the partition element. In the `universityClusterMapSet.xmll`, two replicationGroups are defined: replicationGroup1 and replicationGroup2. The replicationGroup1 replicationGroup contains only one replicationGroupMember. Any partitions that are bound to this replicationGroup are not replicated because more than one replicationGroupMember is required for replication.

The replicationGroup2 replicationGroup contains two replicationGroupMembers. The studentPartition2 partition of the studentPSet partitionSet is using this replicationGroup. Therefore, the studentPartition2 partition is replicated across the two replicationGroupMembers. The replicationGroup2 replicationGroup also has its minReplicas and maxReplicas attributes set. Because minReplicas is set to 1, the map data that is housed in this replicationGroup is read-only until a primary and at least one replica become available. The maxReplicas value of 1 indicates that the primary of this replicationGroup replicates its data to at most one replica. In the case of the replicationGroup2 replicationGroup, it is not possible to exceed one replica because the group only contains two members. One member is the primary and the other is a replica.

## replicationGroupMember element

**Number of occurrences**: one to many

**Child elements**: none

A replicationGroupMember element is used to refer to a server definition. Each replicationGroupMember element also has an associated priority. This priority is used to determine which replicationGroupMember is the primary server and which members are replicas.

### Attributes

```
<replicationGroupMember
(1) serverRef="serverDefinitionReference"
(2) priority="priority"
/>
```

1. **serverRef** attribute (required): The serverRef attribute is used to associate a server definition with a replicationGroupMember element. The serverRef attribute associates each replicationGroupMember with a specific server.
2. **priority** element (required): The priority attribute is used to determine which of the replication group members are the primary. The priority values range from `1` to the number of replication group members, with `1` being the highest priority. ObjectGrid makes a best effort to honor the priority for each replication group member. The replicationGroupMember element with a priority of `1` is the primary unless the circumstances prevent it. If all the servers and their replicationGroupMembers become available at approximately the same time, the priority settings are honored. However, if a replicationGroupMember with a priority of `2` is available long before any other repliationGroupMember, then it becomes the primary.

   If the primary starts successfully and fails after a period of time, a new primary must be selected. The replicationGroupMember element with the next highest priority is likely to become the new primary. However, a different replica might be selected as the new primary if the replica with the next highest priority is determined to be behind in its replication.

Refer to the `universityClusterMapSet.xml` file for sample usage of the partition element. In the `universityClusterMapSet.xml`, the replicationGroup1 replicationGroup has only one replicationGroupMember. Because of this defintion, this replicationGroup only has a primary. There are no replicas within this group. The replicationGroupMember defined is active on the server1 server as the serverRef value states.

The replicationGroup2 replicationGroup has more than one replicationGroupMember. The first replicationGroupMember listed is activated on the server1 server. This member has a priority of `2`. The second replicationGroupMember listed is activated on the sever2 server. Because the second member has a priority of `1`, it is the primary of this replicationGroup if the group members become available at approximately the same time. The first replicationGroupMember that is listed serves as a replica because it has a priority of `2`.

It is also important to understand how the minReplicas value affects the replicationGroup2 replicationGroup. Consider the scenario where both the server1 and server2 servers are running. In this case, both replicationGroupMembers are available. Therefore, the minReplicas and maxReplicas values are both satisfied and data is replicated between the primary and the replica of this group. If the server1 becomes unavailable, one of the replicationGroupMembers becomes unavailable. In this situation, the data in the replicationGroup2 replicationGroup becomes read-only because the minReplicas value is no longer met.

## authenticator element

**Number of occurrences**: zero to one

**Child element**: property element

An authenticator element is used to authenticate clients to ObjectGrid servers in the cluster. The class specified by className attribute must implement the com.ibm.websphere.objectgrid.security.plugins.Authenticator interface. The authenticator can use properties to call methods on the class specified by the className attribute. See “property element” on page 277 for more information on using properties.

### Attributes

```
<authenticator
(1) className="authenticatorClassName"
/>
```

1. **className** attribute (required): The className attribute is used to specify a class that implements the com.ibm.websphere.objectgrid.security.plugins.Authenticator interface. This class is used to authenticate clients to the servers in the ObjectGrid cluster.

The following `universityClusterSecurity.xml` file demonstrates how to use the authenticator element. In this example, the com.ibm.websphere.objectgrid.security.plugins.builtins.KeyStoreLoginAuthenticator class is specified as the authenticator. This class implements the com.ibm.websphere.objectgrid.security.plugins.Authenticator interface.

*universityClusterSecurity.xml* file

```
<?xml version="1.0" encoding="UTF-8"?>
<clusterConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config/cluster
  ../objectGridCluster.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config/cluster">

 <cluster name="universityCluster" securityEnabled="true"
   singleSignOnEnabled="true"
       loginSessionExpirationTime="1800" adminAuthorizationEnabled="true"
       adminAuthorizationMechanism="AUTHORIZATION_MECHANISM_CUSTOM">
   <serverDefinition name="server1" host="lion.ibm.com"
    clientAccessPort="12501" peerAccessPort="12502" />
   <authenticator
```

```
className ="com.ibm.websphere.objectgrid.security.plugins.builtins.
  KeyStoreLoginAuthenticator" />
   <adminAuthorization className= "com.ibm.MyAdminAuthorization">
     <property name="interval" type="int" value="60" description="Set the
     interval to 60 seconds" />
   </adminAuthorization>
 </cluster>

 <objectgridBinding ref="academics">
  <mapSet name="academicsMapSet" partitionSetRef="partitionSet1">
   <map ref="faculty" />
   <map ref="student" />
   <map ref="course" />
  </mapSet>
 </objectgridBinding>

 <objectgridBinding ref="athletics">
  <mapSet name="athleticsMapSet" partitionSetRef="partitionSet1">
   <map ref="athlete" />
   <map ref="equipment" />
  </mapSet>
 </objectgridBinding>

 <partitionSet name="partitionSet1">
  <partition name="partition1" replicationGroupRef="replicationGroup1" />
 </partitionSet>

 <replicationGroup name="replicationGroup1">
  <replicationGroupMember serverRef="server1" priority="1" />
 </replicationGroup>
</clusterConfig>
```

## adminAuthorization element

**Number of occurrences**: zero to one

**Child element**: property element

An adminAuthorization element is used to set up administrative access to the ObjectGrid cluster. Administrative tasks can be performed after administration access has been provided.

### Attributes

```
<adminAuthorization
(1) className="adminAuthClassName"
/>
```

1. **className** attribute (required): The className attribute is used to specify a class that implements the com.ibm.websphere.objectgrid.security.plugins.AdminAuthorization interface.

Refer to the `universityClusterSecurity.xml` file in the authenticator section for sample usage of the adminAuthorization element. In the universityClusterSecurity.xml, a custom adminAuthorization is used. The com.ibm.MyAdminAuthorization class is used as the adminAuthorization class. To use a custom adminAuthorization, the securityEnabled attribute must be `true`, adminAuthorizationMechanism must be set to `AUTHORIZATION_MECHANISM_CUSTOM`, and an adminAuthorization element must be provided. This adminAuthorization element also uses a property. For more information on how to use properties, see "property element" on page 277.

## property element

**Number of occurrences**: zero to many

**Child elements**: none

The property element is used to call set methods on the authenticator and adminAuthorization. The name of the property corresponds to a set method on the className of the authenticator or adminAuthorization element that contains the property.

### Attributes

```
<property
(1) name="propertyName"
(2) type="java.lang.String|boolean|java.lang.Boolean|int|
  java.lang.Integer|double|java.lang.Double|byte|
  java.lang.Byte|short|java.lang.Short|long|
     java.lang.Long|float|java.lang.Float|char|
     java.lang.Character"
(3) value="propertyValue"
(4) description="description"
/>
```

1. **name** attribute (required): The name of the property. The value assigned to this attribute must correspond to a set method on the class that is provided as the className for the authenticator or adminAuthorization. For example, if the className of the authenticator is set to com.ibm.MyAuthenticator and the name of the property provided is `interval`, then the com.ibm.MyAuthenticator class must have a setInterval method.
2. **type** attribute (required): The type of the property. It is the type of the parameter that is passed to the set method that is identified by the name attribute. The valid values are the Java primitives, their java.lang counterparts, and java.lang.String. The name and type must correspond to a method signature on the className of the bean. For example, if name is interval and type is int, then a setInterval(int) method must exist on the class that is specified as the className for the authenticator or adminAuthorization.
3. **value** attribute (required): The value of the property. This value is converted to the type that is specified by the type attribute, and then used as a parameter in the call to the set method that is identified by the name and type attributes. It is important to know that the value of this attribute is not be validated in any way. The plug-in implementer must verify that the value passed in is valid. The implementer can display an `IllegalArgumentException` exception in the set method if the parameter is not valid.
4. **description** attribute (optional): Use this attribute to write a description of the property.

Refer to the `universityClusterSecurity.xml` file for sample usage of the adminAuthorization element. The property element can be used within the authenticator or the adminAuthorization elements in the cluster XML. In the `universityClusterSecurity.xml` file, the property is used to call a set method on adminAuthorization. In this case, a setInterval method is called on the com.ibm.MyAdminAuthorization class. It is passed an integer value of `60`.

# Chapter 10. Integrating ObjectGrid with WebSphere Application Server

Use ObjectGrid with the features that are provided with WebSphere Application Server to enhance your applications with ObjectGrid capability.

Install WebSphere Application Server and WebSphere Extended Deployment. After WebSphere Extended Deployment is installed, you can add ObjectGrid functions to your Java 2 Platform, Enterprise Edition (J2EE) applications.

The ObjectGrid API can be used in a WebSphere Application Server-targeted J2EE application. The `wsobjectgrid.jar` file is in the `\base\lib` directory after WebSphere Extended Deployment is installed. In addition to integrating the ObjectGrid API with the J2EE application programming model, you can leverage the distributed transaction propagation support. With this support, you can configure ObjectGrid instances to coordinate transaction commit results across a WebSphere Application Server cluster.

1. Perform the basic programming steps for enabling a J2EE application with ObjectGrid. See “Integrating ObjectGrid in a Java 2 Platform, Enterprise Edition environment” for more information.
2. Monitor performance data for your ObjectGrid applications. See “Monitoring ObjectGrid performance with WebSphere Application Server performance monitoring infrastructure (PMI)” on page 283 for more information.
3. When ObjectGrid is embedded, the transactions might be started and ended by an external transaction coordinator. See “ObjectGrid and external transaction interaction” on page 289 for more information.
4. Use the partitioning facility with ObjectGrid. The ObjectGrid feature provides the capacity of caching key and value pairs in the transactional fashion, and the Partition Facility feature provides the capacity of context-based routing according to object characteristics. See “Integrating ObjectGrid and the partitioning facility” on page 292 for more information.
5. You can use container-managed persistence (CMP) beans in WebSphere Application Server Version 6.0.2 and later, taking advantage of ObjectGrid as an external cache instead of a built-in cache. See “Configuring ObjectGrid to work with container-managed beans” on page 313 for more information.

You can also use ObjectGrid with JMS to distribute changes between different tiers or in environments with mixed platforms. See “Java Message Service for distributing transaction changes” on page 328 for more information.

## Integrating ObjectGrid in a Java 2 Platform, Enterprise Edition environment

ObjectGrid supports both servlet and Enterprise JavaBeans (EJB) programming models in the Java 2 Platform, Enterprise Edition (J2EE) environment.

This topic explores the common programming steps for enabling a J2EE application with ObjectGrid.

© Copyright IBM Corp. 2004, 2005

## Local ObjectGrid scenario

1. Define an ObjectGrid configuration. Define an ObjectGrid configuration either with XML files, through programmatic interface or with a mixed usage of XML files and programmatic configuration. For more information, see ObjectGrid configuration.
2. Create a URL object. If the ObjectGrid configuration is in an XML file, create a URL object that points to that XML file. You can use this URL object to create ObjectGrid instances by using the ObjectGridManager API. If the ObjectGrid configuration XML file is included in a Web archive (WAR) or Enterprise JavaBeans (EJB) Java archive (JAR) file, it is accessible as a resource to the class loader for both the Web and EJB module. For example, if the ObjectGrid configuration XML file is in the WEB–INF folder of the Web module WAR file, servlets that are in that WAR file can create a URL object with the following pattern:

   ```
   URL url =className.class.getClassLoader().
   getResource("META-INF/objectgrid-definition.xml");
   URL objectgridUrl = ObjectGridCreationServlet.class.getClassLoader().
   getResource("WEB-INF/objectgrid-definition.xml");
   ```
3. Create or get ObjectGrid instances. Use the ObjectGridManager API to get and create ObjectGrid instances. With the ObjectGridManager API, you can create ObjectGrid instances with XML and use utility methods to quickly create a simple ObjectGrid instance. Applications must use the ObjectGridManagerFactory API to get a reference to the ObjectGridManager API. See the following coding example:

   ```
   import com.ibm.websphere.objectgrid.ObjectGridManager;
   import com.ibm.websphere.objectgrid.ObjectGridManagerFactory ;
   ...
   ObjectGridManager objectGridManager = ObjectGridManagerFactory.
   getObjectGridManager();
   ObjectGrid ivObjectGrid = objectGridManager.
   createObjectGrid(objectGridName, objectgridUrl, true, true);
   ```

   For more information about the ObjectGridManager API, see the ObjectGridManager interface topic.
4. Initialize the ObjectGrid instances. Use the initialize method in the ObjectGrid interface to begin bootstrapping the ObjectGrid and Session instances. This initialize method is considered optional because the first call to the getSession method performs an implicit initialization. After this method is invoked, the ObjectGrid configuration is considered complete and is ready for runtime usage. Any additional configuration method invocations, such as calling the defineMap(String mapName) method, result in an exception.
5. Get a Session and ObjectMap instance. A session is a container for ObjectMap instances. A thread must get its own Session object to interact with the ObjectGrid core. You can think of this technique as a session that can only be used by a single thread at a time. The session is shareable across threads if it uses only one thread at a time. However, if a J2EE connection or transaction infrastructure is used, the session object is not shareable across threads. A good analogy for this object is a Java Database Connectivity (JDBC) connection to a database.

   An ObjectMap map is a handle to a named map. Maps must have homogenous keys and values. An ObjectMap instance can be used only by the thread that is currently associated with the session that was used to get this ObjectMap instance. Multiple threads cannot share Session and ObjectMap objects concurrently. Keywords are applied within a transaction. A transaction rollback rolls back any keyword association that is applied during this transaction.The coding example follows:

```
Session ivSession = ivObjectGrid.getSession();
ObjectMap ivEmpMap = ivSession.getMap("employees");
ObjectMap ivOfficeMap = ivSession.getMap("offices");
ObjectMap ivSiteMap = ivSession.getMap("sites");
ObjectMap ivCounterMap = ivSession.getMap("counters");
```

6. Begin a session, read or write objects, and commit or roll back the session. Map operations must be within a transactional context. The begin method of the Session object is used to begin an explicit transactional context. After the session begins, applications can start performing map operations. The most common operations include get, `update`, insert, and remove method calls for objects against maps. At the end of map operations, the commit or rollback method of the Session object is called to either commit an explicit transactional context or roll back an explicit transactional context. A programming example follows:

```
ivSession.begin();
Integer key = new Integer(1);
if (ivCounterMap.containsKey(key) == false) {
 ivCounterMap.insert(key, new Counter(10));
}
ivSession.commit();
```

## Distributed ObjectGrid scenario

This distributed ObjectGrid scenario differs from local ObjectGrid scenario only in the way to get the ObjectGrid instance. The following code examples demonstrates how to get a distributed ObjectGrid instance:

```
 //Use ObjectGridManagerFactory to get the reference to the ObjectGridManager API
ObjectGridManager objectGridManager = ObjectGridManagerFactory.
  getObjectGridManager();

//Obtain the ClientClusterContext represents the ObjectGrid cluster from
//ObjectGridManager
//Assuming the WebSphere server also host an ObjectGrid server
//that is a member of the ObjectGrid cluster.
ClientClusterContext context = objectGridManager.connect(null, null);

//Get the ObejctGrid instance from the ObjectGridManager a specific
//ClientClusterContext.
ObjectGrid ivObjectGrid= objectGridManager.getObjectGrid(context,
 "objectgridName");
```

You performed the basic programming steps for enabling a J2EE application with ObjectGrid.

See Building ObjectGrid-enabled Java 2 Platform, Enterprise Edition (J2EE) applications and Considerations for the integration of Java 2 Platform, Enterprise Edition (J2EE) applications and ObjectGrid for more information.

## Building ObjectGrid-enabled Java 2 Platform, Enterprise Edition applications

Use this task to configure the build path, or class path, of ObjectGrid-enabled Java 2 Platform, Enterprise Edition (J2EE) applications. The class path must include the `wsobjectgrid.jar` file that is located in the `$install_root/lib` directory.

Develop an ObjectGrid enabled J2EE application. See Integrating ObjectGrid in a J2EE environment for more information.

This task demonstrates how to set the build path to include the `wsobjectgrid.jar` file in IBM Rational Software Development Platform Version 6.0.

1. In the Project Explorer view of the J2EE perspective, right-click the **WEB** or Enterprise JavaBeans (**EJB**) project, and select **Properties**. The Properties window is displayed.
2. Select the **Java build path** in the left panel, click the **Libraries** tab in the right panel, and click **Add Variable**. The **New Variable Classpath Entry** window is displayed.
3. Click **Configure Variables** to open the **Preference** window.
4. Add a new variable entry.
   a. Click **New**.
   b. Type `OBJECTGRID_JAR` in the **name** field. Click **File** to open the **JAR Selection** window.
   c. Browse to the `/lib` directory, click the **wsobjectgrid.jar** file, and click **Open** to close the **JAR Selection** window.
   d. Click **OK** to close the **New Variable Entry** window.

   The `OBJECTGRID_JAR` variable is displayed in the Classpath variables list.
5. Click **OK** to close the **Preference** window.
6. Select the `OBJECTGRID_JAR` variable from the variables list and click **OK** to close the **New Variable Classpath Entry** window. The `OBJECTGRID_JAR` variable is displayed in the **Libraries** panel.
7. Click **OK** to close the **Properties** window.

You set the build path to include the `wsobjectgrid.jar` file in IBM Rational Software Development Platform Version 6.0.

## Considerations for the integration of Java 2 Platform, Enterprise Edition applications and ObjectGrid

Use these considerations when integrating a Java 2 Platform, Enterprise Edition (J2EE) application with ObjectGrid.

### Startup beans and ObjectGrid

You can use startup beans for an application to bootstrap an ObjectGrid instance when an application starts and destroy the ObjectGrid instance when the application stops. A startup bean is a stateless session bean with a com.ibm.websphere.startupservice.AppStartUpHome remote home and a com.ibm.websphere.startupservice.AppStartUp remote interface. When WebSphere Application Server sees an Enterprise JavaBean (EJB), it recognizes the startup bean. The remote interface has two methods, the start method and the stop method. Use the start method to bootstrap the grid, and call the grid destroy method with the stop method. The application can keep a reference to the grid by using the ObjectGridManager.getObjectGrid method to get a reference when needed. For more information, see the ObjectGridManager interface topic.

### Class loaders and ObjectGrid instances

You must take care when sharing a single ObjectGrid instance between application modules that use different class loaders. Application modules that use different class loaders do not work and result in class cast exceptions in the application. An ObjectGrid must be shared only by application modules that use the same class loader or when the application objects, for example the plug-ins, keys, and values are on a common class loader.

### Manage the life cycle of ObjectGrid instances in a servlet

You can manage the life cycle of ObjectGrid instances with the init method and destroy method of a servlet. Use the init method to create and initialize ObjectGrid instances that are needed by the application. After the ObjectGrid instances are created and cached, you can obtain the instances by their names with the ObjectGridManager API. Use the destroy method to destroy these ObjectGrid instances and to release system resources. For more information, see the ObjectGridManager interface topic.

## Monitoring ObjectGrid performance with WebSphere Application Server performance monitoring infrastructure (PMI)

ObjectGrid supports performance monitoring infrastructure (PMI) when running in a WebSphere Application Server or WebSphere Extended Deployment application server. PMI collects performance data on runtime applications and provides interfaces that support external applications to monitor performance data.

For more information about the statistics that ObjectGrid provides, see ObjectGrid statistics.

ObjectGrid uses the custom PMI feature of WebSphere Application Server to add its own PMI instrumentation. With this approach, you can enable and disable ObjectGrid PMI with the administrative console or with Java Management Extensions (JMX) interfaces. In addition, you can access ObjectGrid statistics with the standard PMI and JMX interfaces that are used by monitoring tools, including the Tivoli Performance Viewer.

1. Enable ObjectGrid PMI. You must enable PMI to view the PMI statistics. See "Enabling ObjectGrid PMI" on page 286 for more information.
2. Retrieve ObjectGrid PMI statistics. View the performance of your ObjectGrid applications with the Tivoli Performance Viewer. See "Retrieving ObjectGrid PMI statistics" on page 288 for more information.

## ObjectGrid statistics

ObjectGrid provides two performance monitoring infrastructure (PMI) modules: the objectGridModule module and the mapModule module.

### objectGridModule module

The objectGridModule module contains one time statistic: transaction response time. An ObjectGrid transaction is defined as the duration between the Session.begin method call and the Session.commit method call. This duration is tracked as the transaction response time.

The root element of the objectGridModule module, the ObjectGrids element, serves as the entry point to the ObjectGrid statistics. This root element has ObjectGrid instances as its children that have transaction types as their children. The response time statistic is associated with each transaction type. The objectGridModule module structure is shown in the following diagram:

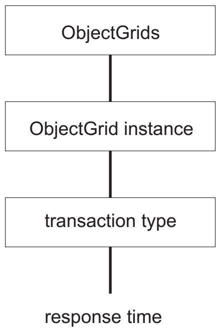


*Figure 19. ObjectGridModule module structure*

The following diagram shows an example of the ObjectGrid PMI module structure. In this example, two ObjectGrid instances exist in the system: the objectGrid1 ObjectGrid and the objectGrid2 ObjectGrid. The objectGrid1 instance has two types of transactions: update and read, and the objectGrid2 instance has only type of transaction: update.

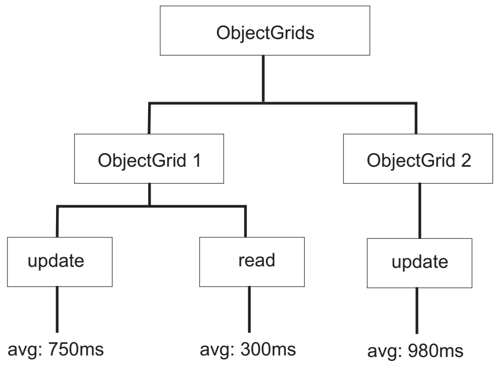


*Figure 20. ObjectGrid PMI module structure*

Transaction types are defined by application developers because they know what types of transactions their applications use. The transaction type is set using the following Session.setTransactionType(String) method:

```
/**
* Sets the transaction type for future transactions.
*
* After this method is called, all of the future transactions have the
* same type until another transaction type is set. If  no transaction
* type is set, the default  TRANSACTION_TYPE_DEFAULT transaction type
* is used.
*
* Transaction types are used mainly for statistical data tracking purpose.
* Users can predefine types of transactions that run in an
* application. The idea is to categorize transactions with the same characteristics
* to one category (type), so one transaction response time statistic can be
* used to track each transaction type.
*
* This tracking is useful when your application has different types of
* transactions.
* Among them, some types of transactions, such as update transactions, process
* longer than other transactions, such as read-only transactions. By using the
* transaction type, different transactions are tracked by different statistics,
* so the statistics can be more useful.
```

```
*
* @param tranType the transaction type for future transactions.
*/
void setTransactionType(String tranType);
```

The following example sets transaction type to `updatePrice`:

```
// Set the transaction type to updatePrice
// The time between session.begin() and session.commit() will be
// tracked in the time statistic for "updatePrice".
session.setTransactionType("updatePrice");
session.begin();
map.update(stockId, new Integer(100));
session.commit();
```

The first line indicates that the subsequent transaction type is `updatePrice`. An `updatePrice` statistic exists under the ObjectGrid instance that corresponds to the session in the example. Using Java Management Extensions (JMX) interfaces, you can get the transaction response time for updatePrice transactions. You can also get the aggregated statistic for all types of transactions on the specified ObjectGrid.

## mapModule module

The mapModule PMI module contains three statistics that are related to ObjectGrid maps:

- **Map hit rate:** This BoundedRangeStatistic statistic tracks the hit rate of a map. Hit rate is a float value between `0` and `100` inclusively, which represents the percentage of map hits in relation to map get operations.
- **Number of entries**: This CountStatistic statistic tracks the number of entries in the map.
- **Loader batch update response time**: This TimeStatistic statistic tracks the response time that is used for the loader batch update operation.

The root element of the mapModule module, the ObjectGrid Maps element, serves as the entry point to the ObjectGrid Map statistics. This root element has ObjectGrid instances as its children, which have map instances as their children. Every map instance has the three listed statistics. The mapModule structure is shown in the following diagram:
The following diagram shows an example of the mapModule structure:

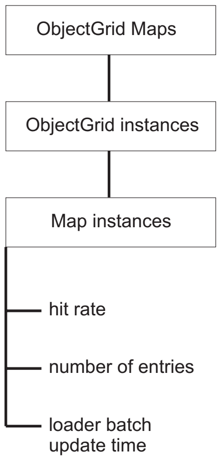


*Figure 21. mapModule module structure*

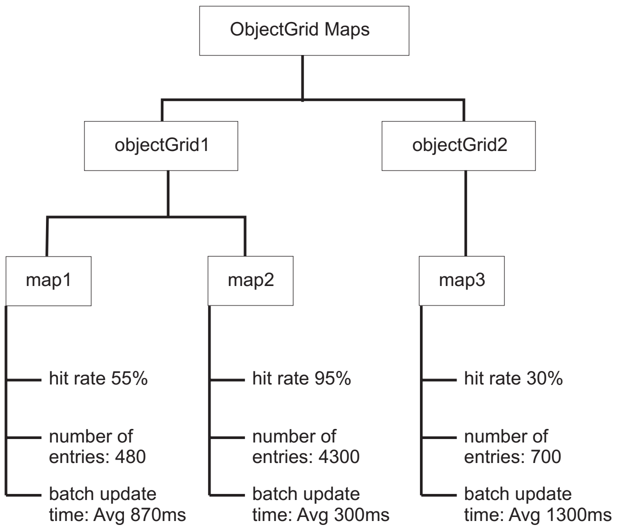


*Figure 22. mapModule module structure example*

## Enabling ObjectGrid PMI

You can use WebSphere Application Server Performance Monitoring Infrastructure (PMI) to enable or disable statistics at any level. For example, you can choose to just enable the map hit rate statistic for a particular map, but not the number of entry statistic or the loader batch update time statistic. This topic shows how to use the administrative console and wsadmin scripts to enable ObjectGrid PMI.

Use WebSphere Application Server PMI to provide a granular mechanism with which you can enable or disable statistics at any level. For example, you can choose to enable map hit rate statistic for a particular map, but not the number of entry statistic or the loader batch update time statistic. This section shows how to use the administrative console and wsadmin scripts to enable ObjectGrid PMI.

1. Open the administrative console, for example, http://localhost:9060/ibm/console.
2. Click **Monitoring and Tuning > Performance Monitoring Infrastructure >** ***server_name***.
3. Verify that **Enable Performance Monitoring Infrastructure (PMI)** is selected. This setting is enabled by default. If the setting is not enabled, select the check box and then restart the server.
4. Click **Custom**. In the configuration tree, select the **ObjectGrid** and **ObjectGrid Maps** module. Enable the statistics for each module.

The transaction type category for ObjectGrid statistics is created at run time. You can see only the subcategories of the ObjectGrid and Map statistics on the Runtime panel.

For example, you can perform the following steps to enable PMI statistics for the sample application:

1. Launch the application using the `http://host:port/ObjectGridSample` Web address, where host and port are the host name and HTTP port number of the server where the sample is installed.

2. In the sample application, click **ObjectGridCreationServlet**, and then click action buttons 1, 2, 3, 4, and 5 to generate some actions to the ObjectGrid and maps. Do not close this servlet page at this time.
3. Go back to the administrative console, click **Monitoring and Tuning > Performance Monitoring Infrastructure > *server_name***. Click the **Runtime** tab.
4. Click the **Custom** radio button.
5. Expand the **ObjectGrid Maps** module in the runtime tree, and then click the **clusterObjectGrid** link. Under **ObjectGrid Maps** group, one ObjectGrid instance exists that is called clusterObjectGrid, and under this clusterObjectGrid group, four maps exist: counters, employees, offices, and sites. In the **ObjectGrids** instance, one clusterObjectGrid instance exists, and under that instance is a transaction type called `DEFAULT`.
6. You can enable the statistics that you are interested in. For demonstration purpose, you can enable **number of map entries** for employees map, and **transaction response time** for the DEFAULT transaction type.

You can automate the task of enabling PMI with scripting. See Enabling ObjectGrid PMI with scripting for more information.

## Enabling ObjectGrid PMI with scripting

Automate the task of enabling ObjectGrid PMI with the wsadmin tool.

Your application server must be started and have an ObjectGrid enabled application installed. You also must be able to log in and use the wsadmin tool. For more information about the wsadmin tool, see Using scripting (wsadmin) in the WebSphere Extended Deployment Version 6.0.x information center.

Use this task to automate enabling PMI. To enable PMI with the administrative console, see Enabling ObjectGrid PMI.

1. Open a command line prompt. Navigate to the *`install_root`*`/bin` directory. Type type `wsadmin` to start the wsadmin command line tool.
2. Modify the ObjectGrid PMI runtime configuration. Check to see if PMI is enabled for the server with the following commands:

   ```
   wsadmin>set s1 [$AdminConfig getid /Cell:CELL_NAME/Node:NODE_NAME/Server:
    APPLICATION_SERVER_NAME/]
   wsadmin>set pmi [$AdminConfig list PMIService $s1]
   wsadmin>$AdminConfig show $pmi.
   ```

   If PMI is not enabled, run the following commands to enable PMI:

   ```
   wsadmin>$AdminConfig modify $pmi {{enable true}}
   wsadmin>$AdminConfig save
   ```

   If you need to enable PMI, restart the server.
3. Set variables for changing the statistic set to a custom set. Run the following commands:

   ```
   wsadmin>set perfName [$AdminControl completeObjectName type=
    Perf,process=APPLICATION_SERVER_NAME,*]
   wsadmin>set perfOName [$AdminControl makeObjectName $perfName]
   wsadmin>set params [java::new {java.lang.Object[]} 1]
   wsadmin>$params set 0 [java::new java.lang.String custom]
   wsadmin>set sigs [java::new {java.lang.String[]} 1]
   wsadmin>$sigs set 0 java.lang.String
   ```
4. Set statistic set to custom: Run the following command:

   ```
   wsadmin>$AdminControl invoke_jmx $perfOName setStatisticSet $params $sigs
   ```

5. Set variables to enable the objectGridModule PMI statistic. Run the following commands:

   ```
   wsadmin>set params [java::new {java.lang.Object[]} 2]
   wsadmin>$params set 0 [java::new java.lang.String objectGridModule=1]
   wsadmin>$params set 1 [java::new java.lang.Boolean false]
   wsadmin>set sigs [java::new {java.lang.String[]} 2]
   wsadmin>$sigs set 0 java.lang.String
   wsadmin>$sigs set 1 java.lang.Boolean
   ```
6. Set the statistics string. Run the following command:

   ```
   wsadmin>$AdminControl invoke_jmx $perfOName setCustomSetString $params $sigs
   ```
7. Set variables to enable the mapModule PMI statistic. Run the following commands:

   ```
   wsadmin>set params2 [java::new {java.lang.Object[]} 2]
   wsadmin>$params2 set 0 [java::new java.lang.String mapModule=*]
   wsadmin>$params2 set 1 [java::new java.lang.Boolean false]
   wsadmin>set sigs2 [java::new {java.lang.String[]} 2]
   wsadmin>$sigs2 set 0 java.lang.String
   wsadmin>$sigs2 set 1 java.lang.Boolean
   ```
8. Set the statistics string. Run the following command:

   ```
   wsadmin>$AdminControl invoke_jmx $perfOName setCustomSetString $params2 $sigs2
   ```

These steps enable ObjectGrid runtime PMI, but do not modify the PMI configuration. If you restart the application server, the PMI settings are lost, other than the main PMI enablement.

After PMI is enabled, you can view PMI statistics with the administrative console or through scripting. See “Retrieving ObjectGrid PMI statistics” and “Retrieving ObjectGrid PMI statistics with scripts” on page 289 for more information.

## Retrieving ObjectGrid PMI statistics

See the performance statistics of your ObjectGrid applications.

After the ObjectGrid statistics are enabled, you can retrieve them. To enable ObjectGrid PMI, see Enabling ObjectGrid PMI.

Use this task to see the performance statistics of your ObjectGrid applications.

1. Open the administrative console. For example, http://localhost:9060/ibm/console.
2. Click **Monitoring and Tuning > Performance Viewer > Current Activity**.
3. Click the server that you want to monitor using Tivoli Performance Viewer and enable the monitoring.
4. Click the server to view the Performance viewer page.
5. Expand the configuration tree. Click **ObjectGrid Maps > clusterObjectGrid**, and select **employees**. Expand **ObjectGrids > clusterObjectGrid**, and select **DEFAULT**.
6. In the ObjectGrid sample application, go back to the ObjectGridCreationServlet servlet , click button 1, **populate maps**. You can view the statistics in the viewer.

You can view ObjectGrid Statistics in the Tivoli Performance Viewer.

You can automate the task of retrieving statistics by using Java Management Extensions (JMX) or with the wsadmin tool. See “Retrieving ObjectGrid PMI statistics with scripts” on page 289

### Retrieving ObjectGrid PMI statistics with scripts

Use this task to retrieve performance statistics for your ObjectGrid applications.

Enable Performance Monitoring Infrastructure (PMI) in your application server environment. See Enabling ObjectGrid PMI or Enabling ObjectGrid PMI with scripts for more information. You also must be able to log in and use the wsadmin tool. For more information about the wsadmin tool, see Using scripting (wsadmin) in the WebSphere Extended Deployment Version 6.0.x Information Center.

Use this task to get performance statistics for your application server environment. For more information about the ObjectGrid statistics that can be retrieved, see "ObjectGrid statistics" on page 283.

1. Open a command-line prompt. Navigate to the `install_root/bin` directory. Type `wsadmin` to start the wsadmin command-line tool.
2. Set variables for the environment. Run the following commands:
```
wsadmin>set perfName [$AdminControl completeObjectName type=Perf,*]
wsadmin>set perfOName [$AdminControl makeObjectName $perfName]
wsadmin>set mySrvName [$AdminControl completeObjectName type=Server,
 name=APPLICATION_SERVER_NAME,*]
```
3. Set variables to get mapModule statistics. Run the following commands:
```
wsadmin>set params [java::new {java.lang.Object[]} 3]
wsadmin>$params set 0 [$AdminControl makeObjectName $mySrvName]
wsadmin>$params set 1 [java::new java.lang.String mapModule]
wsadmin>$params set 2 [java::new java.lang.Boolean true]
wsadmin>set sigs [java::new {java.lang.String[]} 3]
wsadmin>$sigs set 0 javax.management.ObjectName
wsadmin>$sigs set 1 java.lang.String
wsadmin>$sigs set 2 java.lang.Boolean
```
4. Get mapModule statistics. Run the following command:
```
wsadmin>$AdminControl invoke_jmx $perfOName getStatsString $params $sigs
```
5. Set variables to get objectGridModule statistics. Run the following commands:
```
wsadmin>set params2 [java::new {java.lang.Object[]} 3]
wsadmin>$params2 set 0 [$AdminControl makeObjectName $mySrvName]
wsadmin>$params2 set 1 [java::new java.lang.String objectGridModule]
wsadmin>$params2 set 2 [java::new java.lang.Boolean true]
wsadmin>set sigs2 [java::new {java.lang.String[]} 3]
wsadmin>$sigs2 set 0 javax.management.ObjectName
wsadmin>$sigs2 set 1 java.lang.String
wsadmin>$sigs2 set 2 java.lang.Boolean
```
6. Get objectGridModule statistics. Run the following command:
```
wsadmin>$AdminControl invoke_jmx $perfOName getStatsString $params2 $sigs2
```

See "ObjectGrid statistics" on page 283 for more information about the statistics that are returned.

## ObjectGrid and external transaction interaction

Usually, ObjectGrid transactions begin with the session.begin method and end with the session.commit method. However, when ObjectGrid is embedded, the transactions might be started and ended by an external transaction coordinator. In this case, you do not need to call the session.begin method and end with the session.commit method.

## External transaction coordination

The ObjectGrid TransactionCallback plug-in is extended with the isExternalTransactionActive(Session session) method that associates the ObjectGrid session with an external transaction. The method header follows:

```
public synchronized boolean isExternalTransactionActive(Session session)
```

For example, ObjectGrid can be set up to integrate with WebSphere Application Server and WebSphere Extended Deployment. The key to this seamless integration is the exploitation of the ExtendedJTATransaction API in WebSphere Application Server Version 5.x and Version 6.x. However, if you are using WebSphere Application Server Version 6.0.2, you must apply APAR PK07848 to support this method. Use the following sample code to associate an ObjectGrid session with a WebSphere Application Server transaction ID:

```
/**
* This methodis required to associate an objectGrid session with a WebSphere
* transaction ID.
*/
Map/**/ localIdToSession;
public synchronized boolean isExternalTransactionActive(Session session)
{
 // remember that this localid means this session is saved for later.
 localIdToSession.put(new Integer(jta.getLocalId()), session);
 return true;
}
```

## Retrieve an external transaction

Sometimes you might need to retrieve an external transaction service object for the ObjectGrid TransactionCallback plug-in to use. In the WebSphere Application Server server, you look up the ExtendedJTATransaction object from its namespace as shown in the following example:

```
public J2EETransactionCallback() {
 super();
 localIdToSession = new HashMap();
 String lookupName="java:comp/websphere/ExtendedJTATransaction";
 try
 {
  InitialContext ic = new InitialContext();
  jta = (ExtendedJTATransaction)ic.lookup(lookupName);
  jta.registerSynchronizationCallback(this);
 }
 catch(NotSupportedException e)
 {
  throw new RuntimeException("Cannot register jta callback", e);
 }
 catch(NamingException e){
  throw new RuntimeException("Cannot get transaction object");
 }
}
```

For other products, you can use a similar approach to retrieve the transaction service object.

## Control commit by external callback

The TransactionCallback plug-in needs to receive an external signal to commit or roll back the ObjectGrid session. To receive this external signal, use the callback from the external transaction service. You need to implement the external callback interface and register it with the external transaction service. For example, in the

WebSphere Application Server case, you need to implement the SynchronizationCallback interface, as shown in the following example:

```
public class J2EETransactionCallback implements
com.ibm.websphere.objectgrid.plugins.TransactionCallback,
SynchronizationCallback
{
 public J2EETransactionCallback() {
 super();
 String lookupName="java:comp/websphere/ExtendedJTATransaction";
 localIdToSession = new HashMap();
 try
 {
  InitialContext ic = new InitialContext();
  jta = (ExtendedJTATransaction)ic.lookup(lookupName);
  jta.registerSynchronizationCallback(this);
 }
 catch(NotSupportedException e)
 {
  throw new RuntimeException("Cannot register jta callback", e);
 }
 catch(NamingException e)
 {
  throw new RuntimeException("Cannot get transaction object");
 }
}
public synchronized void afterCompletion(int localId, byte[] arg1,
boolean didCommit)
{
 Integer lid = new Integer(localId);
 // find the Session for the localId
 Session session = (Session)localIdToSession.get(lid);
 if(session != null)
 {
  try
  {
   // if WebSphere Application Server is committed when
   // hardening the transaction to backingMap.
   // We already did a flush in beforeCompletion
   if(didCommit)
   {
    session.commit();
   }
   else
   {
    // otherwise rollback
    session.rollback();
   }
  }
  catch(NoActiveTransactionException e)
  {
   // impossible in theory
  }
  catch(TransactionException e)
  {
   // given that we already did a flush, this should not fail
  }
  finally
  {
   // always clear the session from the mapping map.
   localIdToSession.remove(lid);
  }
 }
}
public synchronized void beforeCompletion(int localId, byte[] arg1)
{
 Session session = (Session)localIdToSession.get(new Integer(localId));
 if(session != null)
 {
  try
  {
   session.flush();
  }
  catch(TransactionException e)
  {
   // WebSphere Application Server  does not formally define
   // a way to signal the
   // transaction has failed so do this
   throw new RuntimeException("Cache flush failed", e);
  }
 }
}
}
```

### Use ObjectGrid APIs with the TransactionCallback plug-in

This plug-in, when used as the TransactionCallback plug-in for an ObjectGrid, disables autocommit. The normal usage pattern for an ObjectGrid follows:

```
Session ogSession = ...;
ObjectMap myMap = ogSession.getMap("MyMap");
ogSession.begin();
MyObject v = myMap.get("key");
v.setAttribute("newValue");
myMap.update("key", v);
ogSession.commit();
```

When this TransactionCallback plug-in is in use, ObjectGrid assumes that the application uses the ObjectGrid when a container-managed transaction is present. The previous code snippet changes to the following code in this environment:

```
public void myMethod()
{
 UserTransaction tx = ...;
 tx.begin();
 Session ogSession = ...;
 ObjectMap myMap = ogSession.getMap("MyMap");
 MyObject v = myMap.get("key");
 v.setAttribute("newValue");
 myMap.update("key", v);
 tx.commit();
}
```

The myMethod method is similar to a web application case. The application uses the normal UserTransaction interface to begin, commit, and roll back transactions. The ObjectGrid automatically begins and commits around the container transaction. If the method is an Enterprise JavaBeans (EJB) method that uses the TX_REQUIRES attribute, then remove the UserTransaction reference and the calls to begin and commit transactions and the method works the same way. In this case, the container is responsible for starting and ending the transaction.

## Integrating ObjectGrid and the partitioning facility

Use the ObjectGridPartitionCluster sample application to learn about the combined functions of ObjectGrid and the partitioning facility (WPF).

See "ObjectGrid and the partitioning facility" on page 293 for a summary of how the ObjectGrid and partitioning facility work together.

To use ObjectGrid with the partitioning facility, you must have WebSphere Extended Deployment installed in your environment.

The ObjectGridPartitionCluster sample demonstrates the combined functions of ObjectGrid and Partitioning Facility (WPF). The ObjectGrid feature provides the capacity of caching key and value pairs by transaction, and the partitioning facility feature provides the capacity of context-based routing according to object characteristics.

- Install and run the ObjectGridPartitionCluster sample application. See "Installing and running the ObjectGridPartitionCluster sample application" on page 295 for more information.
- If you want to view or modify the source code of the sample application, you can load the enterprise archive (EAR) file into your development tool. See "Building an integrated ObjectGrid and partitioning facility application" on page 298 for more information.
- Learn about the sample application. See "Example: ObjectGrid and partitioning facility programming" on page 301 for explanation about the code that is in the sample application.

See Chapter 10, "Integrating ObjectGrid with WebSphere Application Server," on page 279 for more information about how to integrate ObjectGrid with other WebSphere Application Server features. For more information about the ObjectGrid programming model, see Chapter 9, "ObjectGrid application programming interface overview," on page 87.

## ObjectGrid and the partitioning facility

ObjectGrid and the partitioning facility (WPF) features can work together to provide the caching of key and value pairs and context-based routing based on object characteristics.

The ObjectGridPartitionCluster sample demonstrates the combined functions of ObjectGrid and the partitioning facility (WPF). ObjectGrid and partitioning facility are two features in the WebSphere Extended Deployment product. The ObjectGrid feature provides the capacity of caching key and value pairs in the transactional fashion, and the partitioning facility feature provides the capacity of context-based routing according to object characteristics.

In addition to demonstrating loader and TransactionCallback plug-in features, this sample also demonstrates how to use the ObjectGridEventListener, ObjectTransformer, and OptimisticCallback plug-ins. In particular, the sample demonstrates how to propagate local ObjectGrid transactions and how to invalidate the changed objects from one server to other servers with and without the optimistic version checker.

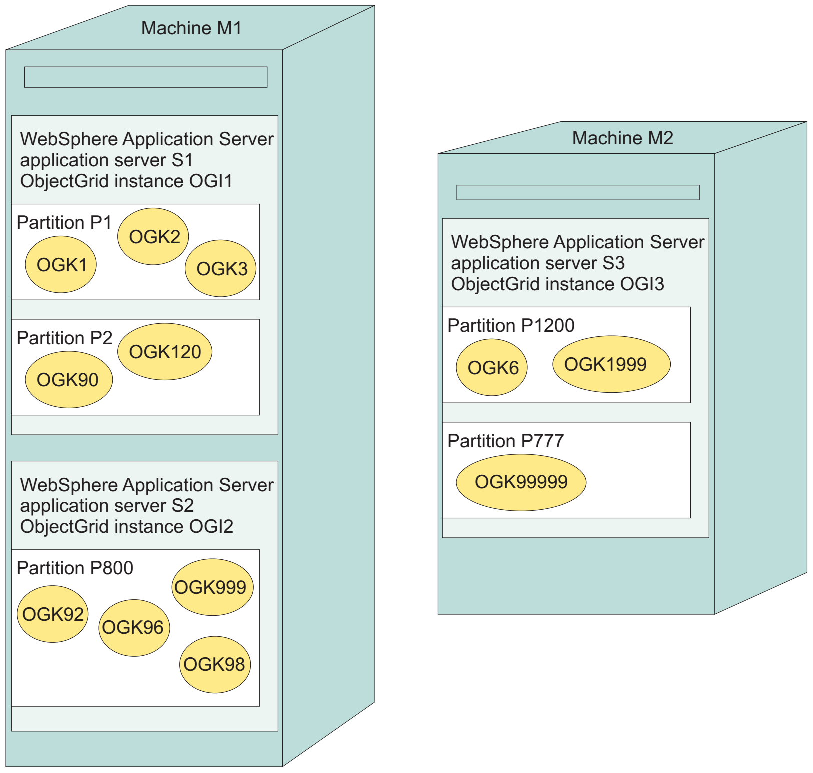
You must use the partitioning facility context-based routing feature to ensure that object update, insert, and remove requests for the same key are routed to the same Java virtual machine (JVM) and that the object retrieval requests can be distributed across all of the ObjectGrid JVMs with workload management. Using the partitioning facility maintains data integrity across the different cluster member ObjectGrid instances.

To maintain ObjectGrid consistency and integrity, you can use the partitioning facility to spread a large ObjectGrid out into many partitioned ObjectGrids, and the partitioning facility context-based routing directs requests according to ObjectGrid keys. For example, you need ObjectGrid to handle a large number of objects that cannot fit into a JVM ObjectGrid. You can use the partitioning facility feature to load data into different servers with the partitionLoadEvent method as preload and the partitioning facility context-based routing finds the right ObjectGrid for you.

The sample creates a set of hash-based partitions and partition cluster routing contexts:

- You can partition and map ObjectGrid keys to the WPF partitions with a many-to-many strategy.
- The WPF partitions can be hosted in the WebSphere Application Server cluster in a many-to-many strategy.

The following diagram shows the typical settings and configuration for the ObjectGridPartitionCluster sample :


Machine M1
WebSphere Application Server application server S1 ObjectGrid instance OGI1
Partition P1
OGK1
OGK2
OGK3
Partition P2
OGK90
OGK120
WebSphere Application Server application server S2 ObjectGrid instance OGI2
Partition P800
OGK92
OGK96
OGK999
OGK98
Machine M2
WebSphere Application Server application server S3 ObjectGrid instance OGI3
Partition P1200
OGK6
OGK1999
Partition P777
OGK99999

In the preceding diagram, the M1 machine and the M2 machine are used to deploy the ObjectGridPartitionCluster sample. Each physical machine can host one or more WebSphere Application Servers. For instance, the M1 machine hosts two application servers: the S1 application server and the S2 application server. The M2 machine hosts one server, which is the S3 application server. Each server has an ObjectGrid instance: OGI1ObjectGrid instance for the S1 application server, the OGI2 ObjectGrid instance for the S2 application server, and the OGI3 ObjectGrid instance for the S3 application server.

Each application server can host many partitions. For example, the S1 server hosts the P1 partition and the P2 partition and the S3 server hosts the P1200 partition and the P777 partition.

Each partition can host many ObjectGrid keys. For example, the P1 partition hosts the OGK1, OGK2, and OGK3 ObjectGrid keys and the P800 partition hosts the OGK92, OGK96, OGK98, and OGK9999 partitions.

All of the ObjectGrid update, insert, and remove requests are routed according to ObjectGrid keys. You have two options for object retrievals: from any server in a workload managed strategy, or from the particular server partition for this key.

## Installing and running the ObjectGridPartitionCluster sample application

Use this task to install and run the ObjectGridPartitionCluster sample application to test the functionality between ObjectGrid and the partitioning facility.

Install WebSphere Extended Deployment. See the WebSphere Extended Deployment Library page for instructions.

A good environment for running the ObjectGridPartitionCluster sample includes installing the WebSphere Extended Deployment into two physical machines, or creating two nodes and federating them together with the deployment manager.

1. To adequately demonstrate the features of this sample, configure a cluster that has three or more cluster members.
2. Install the `D_ObjectGridPartitionClusterSample.ear` file. The partitioning facility (WPF) deployed `D_ObjectGridPartitionClusterSample.ear` file is ready to install and run. If you modify the sample source code, follow the build and wpf-deploy instructions to build and deploy your enterprise archive (EAR) file.

   The common way to install application EAR files is to use the administrative console. Follow the enterprise application install procedure to install the `D_ObjectGridPartitionClusterSample.ear` file. To access this part of the administrative console, click **Applications > Install a new application**. Do not deploy the EAR file during the installation. Use the default settings except in the step where you are asked to select an installation location. On this step, select the cluster that you defined, instead of the default server1 server.
3. Run the ObjectGridPartitionClusterSample client.
   a. Start the cluster. In the administrative console, click **Servers > Clusters**. Select the cluster and click **Start**.
   b. Run the **WAS_INSTALL_ROOT\bin\wpfadmin balance** script command. Verify that the partition has an active status using the **WAS_INSTALL_ROOT\bin\wpfadmin list** command. For information on the wpfadmin script and its commands, see the Partitioning Facility Guide in the WebSphere Extended Deployment Information Center.
   c. To run the ObjectGridPartitionClusterSample client, run the following command:

      ```
      WAS_INSTALL_ROOT/bin/launchClient.bat|sh \
      WAS_INSTALL_ROOT/installableApps/D_ObjectGridPartitionClusterSample.ear \
      -CCBootstrapPort=PORT
      ```

      Where PORT is the server RMI port that you can find in the server `SystemOut.log` file after you start the server. Usually this port value is one of the following values: 9810, 9811, 9812.

      For example, you might run the following command:

      ```
      WAS_INSTALL_ROOT/bin/launchClient.bat|sh
      WAS_INSTALL_ROOT/installableApps/D_ObjectGridPartitionClusterSample.ear
        -CCBootstrapPort=9811
      ```

      For more advanced usage of this script, see "ObjectGridPartitionClusterSample application client options" on page 296.

4. Change the number of partitions. Change the number of partitions that the ObjectGridPartitionCluster session enterprise bean creates: The number of partitions that is created by the PFClusterObjectGridEJB session bean is decided by the NumberOfPartitions environment entry variable that is in the `META-INF\ejb-jar.xml` file. The default value is `10`. You can change the value of this environment variable and reinstall the application to create different numbers of partitions. Set the number of partitions to less than `999999`.
5. Change the distributed listener options. You can change the following ObjectGrid distributed listener options:

*Table 17. Distributed listener options*

| Variable name | Description |
|---|---|
| enableDistribution, | You can enable an ObjectGrid distributed listener with the enableDistribution environment entry variable that is in the EJB deployment descriptor. The default is `true` , which is enabled. Set the value to `false` to turn off the distributed listener. |
| propagationMode | You can change the propagation mode with the propagationMode environment entry variable that is in the EJB deployment descriptor. The default is `update`. You can change the value to `invalidate` if you do not want the default value. |
| propagationVersionOption, | You can change the propagation version option with the propagationVersionOption environment entry variable that is located in the EJB deployment descriptor. The default is `enable`. You can set the value to `disable`. |
| compressionMode | You can change the compression mode with the compressionMode environment entry variable that is located in the EJB deployment descriptor. The default is `enable`. You can set the value to `disable`. |

The default is to propagate the updates with the version check. You might want to set the value to the invalidate mode without the version check.

You installed and ran the ObjectGridPartitionCluster sample application.

## ObjectGridPartitionClusterSample application client options

Use these options for advanced use in running the `D_ObjectGridPartitionClusterSample.ear` file.

### Advanced sample usage

See "Installing and running the ObjectGridPartitionCluster sample application" on page 295 for more information about installing and running the `D_ObjectGridPartitionClusterSample.ear` file.

For advanced use of the sample, refer to the following full usage guide:

```
WAS_INSTALL_ROOT/bin/launchClient.bat|sh
WAS_INSTALL_ROOT/installableApps/D_ObjectGridPartitionClusterSample.ear
-CCproviderURL=corbaloc::HOSTNAME:SERVER_RMI_PORT [-loop LOOP] [-threads
NUMBER_OF_THREADS] [-add NUMBER_OF_STOCKS_PER_PARTITION] [-waitForPropagation
SECONDS_TO_WAIT_FOR_PROPAGATION] [-getIteration
NUMBER_OF_ITERATION_PER_OGKEY]
```

Fill in the following variables:

- HOSTNAME : Specifies the host name of the application server that is running.
- SERVER_RMI_PORT: Specifies the Bootstrap port of the application server.
- LOOP: Specifies how many loops the client runs. This parameter is optional. The default value is 1.
- NUMBER_OF_THREADS: Specifies how many threads the client runs. This parameter is optional. The default value is 1.
- NUMBER_OF_STOCKS_PER_PARTITION: Specifies the number of stocks for each partition to add. This parameter is optional. The default is 3.
- SECONDS_TO_WAIT_FOR_PROPAGATION : Specifies the seconds to wait for newly added or updated ObjectGrid objects to be propagated to other servers. The default is 2seconds.
- NUMBER_OF_ITERATION_PER_OGKEY : Specifies the number of iterations of retrieving objects in ObjectGrid in a workload managed fashion. The default is 6. With more iterations specified, a clear pattern is seen for objects of the same key in different WebSphere Application Server servers.

## Sample output

The output of this command looks like the following example:

```
C:\dev\xd6\bin>launchClient
D_ObjectGridPartitionClusterSample.ear -CCBootstrapPort=9812
IBM WebSphere Application Server, Release 6.0
J2EE Application Client Tool
Copyright IBM Corp., 1997-2004
WSCL0012I: Processing command line arguments.
WSCL0013I: Initializing the J2EE Application Client Environment.
WSCL0035I: Initialization of the J2EE Application Client Environment has completed.
WSCL0014I: Invoking the Application
Client class com.ibm.websphere.samples.objectgrid.partitionclust
er.client.PartitionObjectGrid
ObjectGrid Partition Sample has 10 partitions
PARTITION: ObjectGridHashPartition000007->clusterdevNode01/s2
PARTITION: ObjectGridHashPartition000003->clusterdevNode02/s3
PARTITION: ObjectGridHashPartition000005->clusterdevNode01/s2
PARTITION: ObjectGridHashPartition000010->clusterdevNode02/s3
PARTITION: ObjectGridHashPartition000006->clusterdevNode02/s3
PARTITION: ObjectGridHashPartition000009->clusterdevNode01/s2
PARTITION: ObjectGridHashPartition000008->clusterdevNode01/s1
PARTITION: ObjectGridHashPartition000002->clusterdevNode02/s3
PARTITION: ObjectGridHashPartition000001->clusterdevNode02/s3
PARTITION: ObjectGridHashPartition000004->clusterdevNode01/s2
************** Partition=ObjectGridHashPartition000004****************
------ObjectGrid Operations: Stock Ticket=Stock000104 --------
get on partition for ticket: Stock000104->clusterdevNode02/s2
update: Stock000104->clusterdevNode02/s2
sleep 2 seconds.....
Iteration 1 : Stock000104->clusterdevNode01/s2
> ObjectGrid Stock000104 price=43.35478459674703 lastTransaction=1121137456584
Iteration 2 : Stock000104->clusterdevNode01/s1
> ObjectGrid Stock000104 price=43.35478459674703 lastTransaction=1121137456584
Iteration 3 : Stock000104->clusterdevNode02/s3
> ObjectGrid Stock000104 price=43.35478459674703 lastTransaction=1121137456584
Iteration 4 : Stock000104->clusterdevNode01/s2
> ObjectGrid Stock000104 price=43.35478459674703 lastTransaction=1121137456584
Iteration 5 : Stock000104->clusterdevNode02/s3
> ObjectGrid Stock000104 price=43.35478459674703 lastTransaction=1121137456584
Iteration 6 : Stock000104->clusterdevNode02/s3
> ObjectGrid Stock000104 price=43.35478459674703 lastTransaction=1121137456584
------ObjectGrid Operations: Stock Ticket=Stock000114 --------
get on partition for ticket: Stock000114->clusterdevNode01/s2
```

```
update: Stock000114->clusterdevNode02/s2
sleep 2 seconds.....
Iteration 1 : Stock000114->clusterdevNode02/s3
> ObjectGrid Stock000114 price=39.70991373766818 lastTransaction=1121137458737
Iteration 2 : Stock000114->clusterdevNode01/s2
> ObjectGrid Stock000114 price=39.70991373766818 lastTransaction=1121137458737
Iteration 3 : Stock000114->clusterdevNode01/s1
> ObjectGrid Stock000114 price=39.70991373766818 lastTransaction=1121137458737
Iteration 4 : Stock000114->clusterdevNode02/s3
> ObjectGrid Stock000114 price=39.70991373766818 lastTransaction=1121137458737
Iteration 5 : Stock000114->clusterdevNode01/s2
> ObjectGrid Stock000114 price=39.70991373766818 lastTransaction=1121137458737
Iteration 6 : Stock000114->clusterdevNode02/s3
> ObjectGrid Stock000114 price=39.70991373766818 lastTransaction=1121137458737
------ObjectGrid Operations: Stock Ticket=Stock000124 --------
get on partition for ticket: Stock000124->clusterdevNode02/s2
update: Stock000124->clusterdevNode01/s2
sleep 2 seconds.....
Iteration 1 : Stock000124->clusterdevNode02/s3
> ObjectGrid Stock000124 price=35.37356414423455 lastTransaction=1121137460940
Iteration 2 : Stock000124->clusterdevNode02/s3
> ObjectGrid Stock000124 price=35.37356414423455 lastTransaction=1121137460940
Iteration 3 : Stock000124->clusterdevNode01/s2
> ObjectGrid Stock000124 price=35.37356414423455 lastTransaction=1121137460940
Iteration 4 : Stock000124->clusterdevNode01/s1
> ObjectGrid Stock000124 price=35.37356414423455 lastTransaction=1121137460940
Iteration 5 : Stock000124->clusterdevNode02/s3
> ObjectGrid Stock000124 price=35.37356414423455 lastTransaction=1121137460940
Iteration 6 : Stock000124->clusterdevNode01/s2
> ObjectGrid Stock000124 price=35.37356414423455 lastTransaction=1121137460940
C:\dev\xd6\bin>
```

## Building an integrated ObjectGrid and partitioning facility application

Open, modify, and install the ObjectGrid partitioning sample application.

Use these steps to modify, export, and install the `ObjectGridPartitionSample.ear` file in a WebSphere Extended Deployment environment. If you do not want to make changes to the sample file, you can use the deployed and partitioning facility (WPF)-enabled `D_ObjectGridPartitionClusterSample.ear` file. If you use the `D_ObjectGridPartitionClusterSample.ear` file, you can install and run the file without performing the following steps. Both enterprise archive (EAR) files are in the `WAS_INSTALL_ROOT/installableApps` directory.

1. Set up the`ObjectGridPartitionSample.ear` file in your build environment, such as IBM Rational Application Developer Version 6.0.x or the Application Server Toolkit Version 6.0.x. See "Getting started with building an ObjectGrid and partitioning facility application" on page 299 for more information.
2. Modify any of source code in the sample.
3. Export the ObjectGridPartitionClusterSample application from your build environment as an EAR file. "Exporting the ObjectGridPartitionClusterSample.ear file in IBM Rational Application Developer" on page 300 for more information.
4. Deploy the application so that it can work with the partitioning facility. See "Deploying the ObjectGridPartitionClusterSample.ear file to work with the partitioning facility" on page 301 for more information.
5. Install the `ObjectGridPartitionClusterSample.ear` file into WebSphere Extended Deployment. The common way to install application EAR files is to use the WebSphere Application Server administrative console. Follow the enterprise application installation procedure of the administrative console to install the `D_ObjectGridPartitionClusterSample.ear` file. Do not deploy the file

during the installation; use the default instead. Use the default settings for each step except when you are asked to select where to install. In this step, select the cluster that you defined instead, of the default server1 server.

You installed the `ObjectGridPartitionClusterSample.ear` file into a WebSphere Extended Deployment environment.

For more information about programming with ObjectGrid, partitioning facility, and the sample applications, see "Example: ObjectGrid and partitioning facility programming" on page 301.

## Getting started with building an ObjectGrid and partitioning facility application

Use the Application Server Toolkit Version 6.0.x or IBM Rational Application Developer Version 6.0.x to rebuild the sample application.

The `ObjectGridPartitionClusterSample.ear` file in the `WAS_INSTALL_ROOT/installableApps` directory contains all the source code. You can use the Application Server Toolkit Version 6.0.x or IBM Rational Application Developer Version 6.0.x to rebuild this sample application. This task uses Rational Application Developer as an example to establish the build environment for the `ObjectGridPartitionClusterSample.ear` file. You can also use the Application Server Toolkit; a free assembly tool that is shipped with WebSphere Application Server on a separate CD.

The deployed and WPF-enabled enterprise archive (EAR) file, the `D_ObjectGridPartitionClusterSample.ear` file, also in the `WAS_INSTALL_ROOT/installableApps` directory, is ready to install and run.

1. Import the `ObjectGridPartitionClusterSample.ear` file into Rational Application Developer.
   a. Start Rational Application Developer.
   b. **Optional:** Open the Java 2 Platform, Enterprise Edition (J2EE) perspective to work with J2EE projects. Click **Window > Open Perspective > Other > J2EE**.
   c. **Optional:** Open the Project Explorer view. Click **Window > Show View > Project Explorer**. Another helpful view is the Navigator view: **Window > Show View > Navigator**.
   d. Import the `ObjectGridPartitionClusterSample.ear` file. Click **File > Import > EAR file**, then click **Next**.
   e. Select the `ObjectGridPartitionClusterSample.ear` file from the `WAS_INSTALL_ROOT/installableApps` directory.
   f. **Optional:** Click **New** to open the New Server Runtime wizard and follow the instructions.
   g. In the Target server field, select the **WebSphere Application Server V6.0** type of Server Runtime.
   h. Click **Finish**.

   The ObjectGridPartitionClusterSample, ObjectGridPartitionClusterSampleEJB, and ObjectGridPartitionClusterSampleClient projects must be created and visible in the Project Explorer view.
2. Set up the ObjectGridPartitionClusterSampleEJB project.
   a. In the Project Explorer view of the J2EE perspective, right-click the **ObjectGridPartitionClusterSampleEJB** project in the EJB projects, and select **Properties**. The Properties window is displayed.

b. Click the **Java Build Path** in the left panel, click the **Libraries** tab in the right panel, and select **Add Variable**. The New Variable Classpath Entry window is displayed.
c. Click **Configure Variables** to open the Preference window.
d. Click **New** to open the New Variable Entry window.
e. Type `ObjectGridPartitionCluster_JAR` for the **Name** and click **File** to open the JAR Selection window.
f. Browse to the `WAS_INSTALL_ROOT/lib` directory, and select **wsobjectgrid.jar**. Click **Open** to close the JAR Selection window.
g. Click **OK** to close the New Variable Entry window. The ObjectGridPartitionCluster_JAR variable is displayed in the Class path variables list.
h. Click **OK** to close the Preference window.
i. Select the **ObjectGridPartitionCluster_JAR** variable from the variables list and click **OK** to close the New Variable Classpath Entry window. The ObjectGridPartitionCluster_JAR variable is displayed in the Libraries panel.
j. Repeat this procedure to add the `wpf.jar` file to your environment.
k. Verify that the `wpf.jar` file and the `wsobjectgrid.jar` file are in your build class path.

After your build environment is set up, you can modify source code and apply other changes. See “Building an integrated ObjectGrid and partitioning facility application” on page 298 for more information.

## Exporting the ObjectGridPartitionClusterSample.ear file in IBM Rational Application Developer

After you make changes to the sample file, you can export the ObjectGridPartitionClusterSample application to create an enterprise archive (EAR) file that you can install on WebSphere Extended Deployment servers.

You must have the `ObjectGridPartitionSample.ear` file imported into your development tools so that you can make changes to the source. See “Getting started with building an ObjectGrid and partitioning facility application” on page 299 for more information. Before exporting, make your changes to the sample application.

You can export the `ObjectGridPartitionClusterSample.ear` file from the ObjectGridPartitionClusterSample project in enterprise applications in the IBM Rational Application Developer. You can install the exported `ObjectGridPartitionClusterSample.ear` file on any WebSphere Extended Deployment Version 6.0 server after deploying the partitioning facility.

1. In the Project Explorer view of the Java 2 Platform, Enterprise Edition (J2EE) perspective, right-click the **ObjectGridPartitionClusterSample** application that is located under Enterprise Applications. Click **Export > EAR file**. The Export window is displayed.
2. Click **Browse** to open the **Save As** window. Locate the target output directory, specify the file name as `ObjectGridPartitionClusterSample`, and click **Save**.
3. Click **Browse** to open the Save As window. Locate the target output directory and specify the file name as `ObjectGridPartitionClusterSample`. Click **Save**.

The `ObjectGridPartitionClusterSample.ear` file is created in the specified target output directory.

After you deploy the `ObjectGridPartitionClusterSample.ear` file for the partitioning facility (WPF), you can run the file in WebSphere Extended Deployment. See “Deploying the ObjectGridPartitionClusterSample.ear file to work with the partitioning facility” for more information.

### Deploying the ObjectGridPartitionClusterSample.ear file to work with the partitioning facility

If you plan on installing the `ObjectGridPartitionClusterSample.ear` file on WebSphere Extended Deployment, you must perform a wpf-deploy operationon the file.

You must have an existing `ObjectGridPartitionClusterSample.ear` file. To modify the existing file, see “Building an integrated ObjectGrid and partitioning facility application” on page 298.

Perform the wpf-deploy operation to prepare the enterprise archive (EAR) file in a WebSphere Extended Deployment environment.

1. Create a `DEST_DIR` directory.
2. Copy the `ObjectGridPartitionClusterSample.ear` file nto the `DEST_DIR` directory. Rename the `ObjectGridPartitionClusterSample.ear` file to the `old_ObjectGridPartitionClusterSample.ear` file.
3. Run the following command, where `WORKING_DIR` is the working directory for the ejbdeloy tool, for example, the `c:\temp` directory.
   ```
   WAS_HOME\bin\ejbdeploy.bat|ejbdeploy.sh
    DEST_DIR\old_ObjectGridPartitionClusterSample.ear WORKING_DIR
    DEST_DIR\ObjectGridPartitionClusterSample.ear
   ```
4. Run the following command, where `TEMP_DIR` is a temporary directory for the tool. If the -keep argument is specified, the temporary directories that are created by the wpfStubUtil utility are not deleted.
   ```
   WAS_HOME\bin\wpfStubUtil.cmd|wpfStubUtil.sh
    DEST_DIR\ObjectGridPartitionClusterSample.ear
    ObjectGridPartitionClusterSampleEJB.jar com/ibm/websphere/samples/
    objectgrid/partitioncluster/ejb/PFClusterObjectGridEJB.class
    TEMP_DIR [-stubDebug|-keep]
   ```

The `ObjectGridPartitionClusterSample.ear` file is ready to run in a WebSphere Extended Deployment environment. The `D_ObjectGridPartitionClusterSample.ear` sample file that ships with WebSphere Extended Deployment is already deployed. You do not need to deploy this file before you install the application if you did not change the source code.

Install the `ObjectGridPartitionClusterSample.ear` file into the WebSphere Extended Deployment environment with the administrative console. See “Building an integrated ObjectGrid and partitioning facility application” on page 298 for more information.

## Example: ObjectGrid and partitioning facility programming

This example demonstrates how to use the combined functions of ObjectGrid and the partitioning facility in a Java 2 Platform, Enterprise Edition environment in WebSphere Extended Deployment.

## Purpose

In addition to demonstrating the combined ObjectGrid and partitioning facility (WPF) functions, this example also demonstrates ObjectGrid distributed listener propagation and invalidation.

Object update, insert, and remove requests are routed to specific servers where partitions are hosted for the corresponding ObjectGrid keys. ObjectGrid get method requests are workload managed among all the servers.

This example also illustrates how to partition a big ObjectGrid into many smaller ObjectGrids and use the partitionLoadEvent method to preload data so that the partitioned ObjectGrid can host an unlimited number of objects.

## Overview

The `ObjectGridPartitionClusterSampler.ear` file creates a stock object that illustrates how ObjectGrid and the partitioning facility work together. The stock object contains the following properties:

- ticket
- company
- serialNumber
- description
- lastTransaction
- price

Where the lastTransaction property is the time that the stock has been changed. Use the lastTransaction property to indicate the freshness of objects in the ObjectGrid of different Java virtual machines (JVM).

In the sample, the ObjectGrid instance is created in the Enterprise JavaBeans (EJB) setContext method with the ObjectGridFactory class.

Define a set of hash-based partitions. The default value is 10 partitions, but you can change the number of partitions. Hash the stock tickets into these partitions by using the `SampleUtility.java` file. Each partition can host many ObjectGrid key and value pairs.

The sample demonstrates how the ObjectGrid insert, update, and remove requests are routed to a specific partitioned server, and how the ObjectGrid get method requests route to either a particular server for its key or any server in a cluster. The sample compares object values for a key from different servers after value changes due to an update, insert, or remove operation for this key occurs in a particular server.

## Location

Use this sample in a cluster environment where each server can host many partitions and where each partition can host many objects with different keys.

Two ObjectGrid partition cluster sample files exist in the `<install_root>\installableApps\` directory:

- The `ObjectGridPartitionClusterSampler.ear` file contains the source code. To see the source, expand the EAR file into the file system or import the source into

a development environment. See “Building an integrated ObjectGrid and partitioning facility application” on page 298 for more information.
- The `D_ObjectGridPartitionClusterSample.ear` file is already deployed for the partitioning facility. Follow the readme file and instructions to get this file running quickly.

### Explanation

The following sections include explanation about the ObjectGrid partition cluster sample application:
- “ObjectGrid operation EJB Interface”
- “PartitionKey class” on page 305
- “SampleUtility class and partition mapping” on page 307
- “ObjectGrid creation in the enterprise bean setContext method” on page 309
- “Singleton ObjectGridFactory class” on page 310
- “ObjectGrid partition preload” on page 312

## ObjectGrid operation EJB Interface

This article demonstrates the ObjectGrid operation Enterprise JavaBeans (EJB) interface that performs get, get from partitioned server, insert, update, and remove operations.

### Purpose

The ObjectGrid operation EJB interface performs get, get from partitioned server, insert, update, and remove operations. The get from partitioned server method is routed to a partition that corresponds to the key it requests. The get method is routed in a worked load managed strategy to any server.

### The PFClusterObjectGridEJB interface

The PFClusterObjectGridEJB interface content follows:

```
/**
* Remote interface for Enterprise Bean: PFClusterObjectGridEJB
*/
public interface PFClusterObjectGridEJB extends javax.ejb.EJBObject {
 public String PARTITION_PREFIX = "ObjectGridHashPartition";
 /**
 * Get all Partitions
 *
 * @return Array of Strings
 * @throws java.rmi.RemoteException
 */

 public String [] getAllPartitions() throws java.rmi.RemoteException;
 /**
 * Get where partition is hosted
 *
 * @param partition
 * @return String
 * @throws java.rmi.RemoteException
 */

 public String getServer(String partition)
 throws java.rmi.RemoteException;
 /**
 * Get Stock object and its server information
 * (ServerIDResult) for a stock ticket
 * from any server in a cluster (that has had workload management)
 *
 * @param ticket
 * @return
 * @throws java.rmi.RemoteException
 */

 public ServerIDResult getStock(String ticket)
 throws java.rmi.RemoteException;
 /**
 * Get Stock object and its partitioned server
 * information for a stock ticket
 * from the partition this ticket key is hashed to
 *
 * @param ticket
 * @return ServerIDResult
 * @throws java.rmi.RemoteException
 */

 public ServerIDResult getStockOnPartitionedServer(String ticket)
 throws java.rmi.RemoteException;
 /**
 * Update stock in a particular server where the partition is
 * active for this stock ticket key.
 *
 * @param stock
 * @return ServerIDResult
 * @throws java.rmi.RemoteException
 */

 public ServerIDResult updateStock(Stock stock)
 throws java.rmi.RemoteException;
 /**
 * Remove stock in a particular server where the partition is
 * active for this stock ticket key.
 *
 * @param ticket
 * @return ServerIDResult
 * @throws java.rmi.RemoteException
 */

 public ServerIDResult removeStock(String ticket)
 throws java.rmi.RemoteException;
 /**
 * Insert stock in a particular server where the partition is
 * active for this stock ticket key.
 *
 * @param stock
 * @return ServerIDResult
 * @throws java.rmi.RemoteException
 */

 public ServerIDResult insertStock(Stock stock)
 throws java.rmi.RemoteException;
 /**
 * Retrieve data from all servers and compare values
 *
 * @param server
 * @return ServerObjectGridVerification
 * @throws java.rmi.RemoteException
 */

 public ServerObjectGridVerification verifyObjectGrid(String server)
 throws java.rmi.RemoteException;
}
```

## PartitionKey class

The PartitionKey class controls the behavior of the partitioning facility context-based routing.

The following code illustrates the sample partition key class. When the method returns `not null`, it is routed with the partitioning facility (WPF) router. When the method returns `null`, it is forwarded to the workload management (WLM) router.

```
/**
* PartitionKey for Partitioned Stateless Session Bean WPFKeyBasedPartition
*
*/
public class PFClusterObjectGridEJB_PartitionKey {
 /**
 * Number of Partitions
 *
 * Default is 10.
 *
 */
 static int numOfPartitions=10;
 /**
 * Only once to getPartitionNumbers
 */
 static boolean getNumOfPartitions=true;
 /**
 * Get the number of partitions
 *
 */
 static void getPartitionNumbers(){
 //get only once
  if (getNumOfPartitions){
   try {
    InitialContext ic = new InitialContext();
    PFClusterObjectGridEJBHome home =
    (PFClusterObjectGridEJBHome) PortableRemoteObject.narrow(
    ic.lookup("java:comp/env/ejb/PFClusterObjectGridEJB"),
    PFClusterObjectGridEJBHome.class);
    final PFClusterObjectGridEJB session = home.create();
    String[] PARTITIONS = session.getAllPartitions();
    numOfPartitions=PARTITIONS.length;
    getNumOfPartitions=false;
   }
   catch (ClassCastException e) {
    e.printStackTrace();
    numOfPartitions=10;
   }
   catch (RemoteException e) {
    e.printStackTrace();
    numOfPartitions=10;
   }
   catch (NamingException e) {
    e.printStackTrace();
    numOfPartitions=10;
   }
   catch (CreateException e) {
    e.printStackTrace();
    numOfPartitions=10;
   }
  }
 }
/**
* Return partition key
*
* @param partition
* @return String
*/
```

```
public static String getStock(String key) {
 return null;
}
/**
* Return partition key
*
* @param key
* @return String
*/
public static String getServer(String key) {
 return key;
}
/**
* Retrieve ObjectGrid data from a partitioned server where
* data's changes happen (the highest quality and integrity).
*
* @param ticket
* @return hashcode of stock ticket
*/
public static String getStockOnPartitionedServer(String ticket) {
 if (ticket==null){
  return null;
 }
 getPartitionNumbers();
 return SampleUtility.hashStockKeyToPartition(ticket, numOfPartitions);
}
/**
* Return partition key
*
* @param stock
* @return hashcode of stock ticket
*/
public static String updateStock(Stock stock) {
 getPartitionNumbers();
 String ticket=null;
 if (stock!=null){
  ticket=stock.getTicket();
 }
 return SampleUtility.hashStockKeyToPartition(ticket, numOfPartitions);
}
/**
* Return partition key
*
* @param stock
* @return hashcode of stock ticket
*/
public static String insertStock(Stock stock) {
 getPartitionNumbers();
 String ticket=null;
 if (stock!=null){
  ticket=stock.getTicket();
 }
 return SampleUtility.hashStockKeyToPartition(ticket, numOfPartitions);
}
/**
* Return partition key
*
* @param server
* @return String
*/
public static String verifyObjectGrid(String server) {
 return server;
}
/**
* Return partition key
*
* @param stock
* @return hashcode of stock ticket
*/
public static String removeStock(String ticket) {
if (ticket==null){
 return null;
}
getPartitionNumbers();
 return SampleUtility.hashStockKeyToPartition(ticket, numOfPartitions);
}
/**
* Return partition key
*
* @param partition
* @return
*/
public static String getAllPartitions() {
 return null;
}
}
```

Each remote method must have a corresponding method that returns a valid string or a null value.

## SampleUtility class and partition mapping

Use the `SampleUtility.java` file to manipulate keys, stock tickets, hash, and partitions. You can also use this file to map ObjectGrid keys to partitions. You can develop similar a utility class to map ObjectGrid keys to partitions that can meet your business needs. To use the partitioning facility with ObjectGrid, you must map different keys into different partitions.

### SampleUtility class

The utility class for the ObjectGridPartitionCluster sample follows:

```
/**
* Utility class for ObjectGridPartitionCluster sample
*
*
*/
public class SampleUtility {
 /**
 * Container for recording partitions.
 */
 static Map serverPartitions= new HashMap();
 /**
 * Partition name prefix
 */

 public static String PARTITION_PREFIX = "ObjectGridHashPartition";
 /**
 * Stock name prefix
 */

 public static String STOCK_PREFIX="Stock";
 /**
 * Retrieve the number part of partition name
 *
 * @param partition
 * @return int
 */

 public static int getIntFromPartition(String partition){
  int result=-1;
  int pre=PARTITION_PREFIX.length();
  int p=partition.length();
  String num=partition.substring(pre, p);
  result=Integer.parseInt(num);
  return result;
 }
 /**
 * Retrieve the number part of stock ticket
 *
 * @param ticket
 * @return
 */
 public static int getIntFromStockTicket(String ticket){
  int result=-1;
  int pre=STOCK_PREFIX.length();
  int p=ticket.length();
  String num=ticket.substring(pre, p);
  result=Integer.parseInt(num);
  return result;
 }
 /**
 * Hash stock ticket to a given hash base.
 *
 * @param ticket
 * @param base
 * @return int
 */
 public static int hashTicket(String ticket, int base){
  if (base<1){
   return 0;
  }
  int hash=0;
  int num=getIntFromStockTicket(ticket);
  hash= num % base;
  return hash;
 }

 /**
 * Hash stock key to a partition
 *
 * @param ticket
 * @param base
 * @return String - partition name
 */
 public static String hashStockKeyToPartition(String ticket, int base){
  String p=null;
  int hashcode=hashTicket(ticket, base)+1;
  p=PARTITION_PREFIX+ padZeroToString(hashcode+"", 6);
  return p;
 }
 /**
 * Record server/partition
 *
 * @param server
 * @param partition
 */
 public static void addServer(String server, String partition){
  serverPartitions.put(server, partition);
 }
 /**
 * Remove server/partition
 *
 * @param server
 */
 public static void removeServer(String server){
  serverPartitions.remove(server);
 }
 /**
 * Get all servers where partitions are active.
```

```
 *
 * @return Iterator - String
 */
 public static Iterator getAllServer(){
  return serverPartitions.values().iterator();
 }
}
```

You must use the same global hash base and parse variable to hash into the hash base. Consider the following example:

```
myKey.hashCode % hashBase
```

You need to parse out myKey as hash variable and you need to keep the same hash base among different servers. In the preceding example, the same variable from the Java environment is looked up. You cannot use `key1 % 100`, but you can use `key2 % 90`.

## ObjectGrid creation in the enterprise bean setContext method

Create the ObjectGrid instance in the enterprise bean setContext method as in the `PFClusterObjectGridEJBBean.java` file and retrieve the preload data.

```
/**
* setSessionContext
*
* with ObjectGrid instance
*/
public void setSessionContext(javax.ejb.SessionContext ctx) {
 mySessionCtx = ctx;
 try {
  InitialContext ic = new InitialContext();
  //get PartitionManager
  ivManager = (PartitionManager)
  ic.lookup("java:comp/websphere/wpf/PartitionManager");
  // get enableDistribution configuration
  boolean enableDistribution = ((Boolean)
  ic.lookup("java:comp/env/enableDistribution")).booleanValue();
  System.out.println("***** enableDistribution="+ enableDistribution);
  // get propagationMode configuration
  String propagationMode = (String) ic.lookup("java:comp/env/propagationMode");
  System.out.println("***** pMode="+ propagationMode);
  String pMode=null;
  if (propagationMode.equals(com.ibm.ws.objectgrid.Constants.
   OBJECTGRID_TRAN_PROPAGATION_MODE_DEFAULT_KEY)||
   propagationMode.equals(com.ibm.ws.objectgrid.Constants.
   OBJECTGRID_TRAN_PROPAGATION_MODE_INVALID_KEY) ){
   pMode=propagationMode;
  }
  // get propagationVersionOption configuration
  String propagationVersionOption = (String)
  ic.lookup("java:comp/env/propagationVersionOption");
  System.out.println("***** pVersionOption="+ propagationVersionOption);
  String pVersion=null;
  if (propagationVersionOption.equals(com.ibm.ws.objectgrid.Constants.
   OBJECTGRID_TRAN_PROPAGATION_MODE_VERS_KEY)||
   propagationMode.equals(com.ibm.ws.objectgrid.Constants.
   OBJECTGRID_TRAN_PROPAGATION_MODE_NOVERS_KEY) ){
    pVersion=propagationVersionOption;
  }
  // get compressionMode configuration
  String compressionMode = (String) ic.lookup("java:comp/env/compressionMode");
  System.out.println("***** compressMode="+ compressionMode);
  String compressMode=null;
  if (compressionMode.equals(com.ibm.ws.objectgrid.Constants.
   OBJECTGRID_TRAN_PROPAGATION_COMPRESS_DISABLED)||
   propagationMode.equals(com.ibm.ws.objectgrid.Constants.
   OBJECTGRID_TRAN_PROPAGATION_COMPRESS_ENABLED) ){
   compressMode=compressionMode;
  }
  // whethere preload is enabled
  bPreload = ((Boolean)
  ic.lookup("java:comp/env/preload")).booleanValue();
  System.out.println("***** enablePreload="+ bPreload);
  //whethere remove is enabled
  bRemove = ((Boolean)
  ic.lookup("java:comp/env/remove")).booleanValue();
  System.out.println("***** enableRemove="+ bRemove);
  // whethere Loader is enabled
  boolean bLoader = ((Boolean)
  ic.lookup("java:comp/env/loader")).booleanValue();
  System.out.println("***** enableLoader="+ bLoader);
  // get file path and name
  String filePathandName = (String)
  ic.lookup("java:comp/env/filePathandName");
  System.out.println("***** fileName="+ filePathandName);
  //get ObjectGrid instance
  og=ObjectGridFactory.getObjectGrid(ogName,
  enableDistribution,
  pMode, pVersion,
  compressMode, bLoader,
  filePathandName);
  if (og==null){
   throw new RuntimeException
    ("ObjectGrid insance is null in ObjectGridPartitionClusterSample");
  }
  System.out.println("Bean Context, getObjectGrid="
  + og + " for name="+ ogName);
  if (bPreload && !lock){
   System.out.println("Preload data");
   PersistentStore store=PersistentStore.getStore(filePathandName);
   store.preload(10);
   store.verify(10);
   lock=true;
   preloadData=store.getAllRecords();
  }
 }
 catch (Exception e) {
  logger.logp(Level.SEVERE, CLASS_NAME,
  "setSessionContext", "Exception: " + e);
  throw new EJBException(e);
 }
}
```

## Singleton ObjectGridFactory class

An ObjectGrid instance is created with a custom factory that caches the ObjectGrid instance with custom settings.

An example of how to create an ObjectGrid instance programatically, set the ObjectGridTransformer object, configure the propagation event listener, and set this listener to the ObjectGrid instance follows. You also can use an XML file to perform this configuration.

```
/**
*
* Create ObjectGrid instance and configure it.
*
*
*/
public class ObjectGridFactory {
 /**
 * ObjectGrid name
 */
 static String ogName="WPFObjectGridSample";
 /**
 * ObjectGrid instance
 */
 static ObjectGrid og=null;
 /**
 * ObjectGrid session
 */
 static Session ogSession=null;
 /**
 * Map name
 */
 static String mapName="SampleStocks";
 /**
 * ObjectGrid cache
 */
 static Map ogCache= new HashMap();
 /**
 * Get ObjectGrid instance
 *
 * @param ogn
 * @param enableDist
 * @param pMode
 * @param pVersion
 * @param compressMode
 * @return
 */

public static synchronized ObjectGrid getObjectGrid(String ogn,
 boolean enableDist,
 String pMode,
 String pVersion,
 String compressMode,
 boolean loader,
 String fileName){
  if (ogn!=null){
   ogName=ogn;
  }
  else {
   throw new IllegalArgumentException ("ObjectGrid name given is null");
  }
  if (ogCache.containsKey(ogName)){
   return (ObjectGrid) ogCache.get(ogName);
  }
  try {
   ObjectGridManager manager= ObjectGridManagerFactory.
   getObjectGridManager();
   og=manager.createObjectGrid(ogName);
   if (enableDist){
    TranPropListener tpl=new TranPropListener();
     if (pMode!=null){
      tpl.setPropagateMode(pMode);
     }
     if (pVersion!=null){
      tpl.setPropagateVersionOption(pVersion);
     }
     if (compressMode!=null) {
      tpl.setCompressionMode(compressMode);
     }
     og.addEventListener(tpl);
   }
   // Define BackingMap and set the Loader
  BackingMap bm = og.defineMap(mapName);
  ObjectTransformer myTransformer=
  new MyStockObjectTransformer();
  bm.setObjectTransformer(myTransformer);
  OptimisticCallback myOptimisticCallback=
  new MyStockOptimisticCallback();
  if (loader){
   TransactionCallback tcb=new MyTransactionCallback();
   Loader myLoader= new MyCacheLoader(fileName, mapName);
   og.setTransactionCallback(tcb);
   bm.setLoader(myLoader);
  }
  og.initialize();
  ogCache.put(ogName, og);
 }
 catch (Exception e) {
 }
 return og;
 }
}
```

## ObjectGrid partition preload

This topic discusses how to preload an ObjectGrid instance.

Use the partitionLoadEvent method to load objects that are related to this partition only when the partition is activated. By loading objects when the partitioning is activated, you partition ObjectGrid so that ObjectGrid can handle large numbers of objects.

```
/**
* This is called when a specific partition is assigned to this server process.
* @param partitionName
* @return
*/
public boolean partitionLoadEvent(String partitionName) {
 //preload data
 preloadDataForPartition(partitionName);
 logger.logp(
  Level.FINER,
  CLASS_NAME,
  "partitionLoadEvent",
  "Loading "+ partitionName );
  return true;
}
/**
*
* preload data
*
* @param partition
*/

private synchronized void preloadDataForPartition(String partition){
 if (bPreload && (preloadData!=null)){
  Iterator itr=preloadData.keySet().iterator();
  while (itr.hasNext()){
   String ticket= (String) itr.next();
   String p=SampleUtility.
   hashStockKeyToPartition(ticket, numOfPartitions);
   if (partition.equals(p)){
    Stock stock= (Stock) preloadData.get(ticket);
    System.out.println("preload in partition=" +
    partition + " with data ticket="+ ticket);
    insertStock(stock);
   }
  }
 }
}
```

You might need to disable the distributed updates if you use the partitioned preload of your big ObjectGrid to partition your big ObjectGrid. The current version of distributed updates cannot be partitioned. The partitioning facility (WPF) context-based routing finds the correct data at the correct partition.

## Configuring ObjectGrid to work with container-managed beans

With WebSphere Application Server Version 6.0.2 and later, you can use container-managed persistence (CMP) beans with an external cache product.

Use this task to use CMP beans, taking advantage of ObjectGrid as an external cache instead of a built-in cache. This functionality is provided by the persistence engine in WebSphere Application Server.

1. Define the JVM arguments to define the CacheFactoryManager adapter and the location of the ObjectGrid XML configuration file. The CacheFactoryManager is an adapter between the persistence engine and ObjectGrid.
   a. Click **Servers > Application servers >** *server_name* **> Java and process management > Process definition > Java Virtual Machine > Generic JVM arguments**.
   b. Add the following properties:
      - `-Dcom.ibm.ws.pmcache.manager=com.ibm.ws.objectgrid.adapter.pm.CacheFactoryManager`
      - `-Dcom.ibm.ws.pmcache.config=file:/d:/temp/objectGrid.xml`

      The `-Dcom.ibm.ws.pmcache.config` property specifies configuration file for the ObjectGrid. The value is a URL to the ObjectGrid configuration file.
2. Configure the ObjectGrid XML configuration file. The configuration is in the `objectGrid.xml` file. Consider the following example. Application and module information are needed for for the Java 2 Platform, Enterprise Edition (J2EE) application. The information is reflected in the `objectGrid.xml` file. An Accounts application has three CMP Enterprise JavaBeans: Savings, Checkin and MoneyMarket. These Enterprise JavaBeans are contained in the PersonalBankingEJB module. The display name is *Accounts* and *PersonalBankingEJB* is the EJB module of the application deployment descriptor. The Savings, Checkin and MoneyMarket are the names as specified in ejb-name element of the Enterprise Java Bean deployment descriptor for the container-managed Entity Beans (CMP). A sample snippet for this configuration follows:

   ```
   <ObjectGrids>
    <ObjectGrid name="Accounts">
     <BackingMap name="PersonalBankingEJB.jar#Savings" readOnly="true"
      pluginCollectionRef="default" />
     <BackingMap name="PersonalBankingEJB.jar#Checkin" readOnly="true"
      pluginCollectionRef="default" />
     <BackingMap name="PersonalBankingEJB.jar#MoneyMarket" readOnly="true"
      pluginCollectionRef="default" />
    </ObjectGrid>
   </ObjectGrids>
   ```

   The `PersonalBankingEJB.jar` file is specified within the EJB tags of the application deployment descriptor, as in the following example:

   ```
   <module id="module_1">
    <ejb>PersonalBankingEJB.jar</ejb>
   </module>
   ```
3. Enable the persistence manager LifeTimeInCache setting for each bean, within the application to use external cache. ObjectGrid requires that you enable this setting in the deployment descriptor, but it ignores the LifeTimeInCache setting. The ObjectGrid configuration takes precedence.

4. Explicitly configure a backingMap to evict objects from the cache. An XML code snippet for the `objectGrid.xml` file follows:

```
<backingMapPluginCollection id="TotalTimeToLive">
 <bean id="Evictor"
  className="com.ibm.websphere.objectgrid.plugins.builtins.TTLEvictor">
  <property name="pruneSize" type="int" value="2"
   description="set max size for TTL Evictor" />
  <property name="numberOfHeaps" type="int" value="1"
   description="set number of TTL heaps" />
  <property name="sleepTime" type="int" value="1"
   description="evictor thread sleep time" />
  </bean>
</backingMapPluginCollection>
<backingMapPluginCollection id="LifeTimeInCache">
 <bean id="Evictor"
  className="com.ibm.websphere.objectgrid.plugins.builtins.TTLEvictor">
  <property name="lifeTime" type="int" value="3"
   description="lifetime of map entry is 3 seconds" />
  <property name="pruneSize" type="int" value="2"
   description="set max size for TTL Evictor" />
  <property name="numberOfHeaps" type="int" value="1"
   description="set number of TTL heaps" />
  <property name="sleepTime" type="int" value="1"
   description="evictor thread sleep time" />
 </bean>
</backingMapPluginCollection>
```

The lifeTime property controls the evictor.

See Chapter 10, “Integrating ObjectGrid with WebSphere Application Server,” on page 279 for more information about how you can use ObjectGrid with WebSphere Application Server.

# Chapter 11. ObjectGrid performance best practices

You can improve the performance of a ObjectGrid Map with the following best practices. These best practices are implemented only in the context of the application and its architecture.

Every application and environment uses a different solution for performance. ObjectGrid provides built-in customizations to improve performance, but you can also improve performance within the application architecture. The following areas offer performance improvements:

- "Locking performance best practices"

  Choose between the different locking strategies that can affect the performance of your applications.
- "copyMode method best practices" on page 316

  Choose between the different copy modes that can be used to change how ObjectGrid maintains and copies entries.
- "ObjectTransformer interface best practices" on page 320

  Use the ObjectTransformer interface to allow callbacks to the application to provide custom implementations of common and expensive operations such as object serialization and a deep copy on an object.
- "Plug-in evictor performance best practices" on page 321

  Choose between least frequently used (LFU) and least recently used (LRU) eviction strategies.
- "Default evictor best practices" on page 323

  Properties for the default time to live (TTL) evictor, the default evictor that is created with every backingMap.

## Locking performance best practices

Locking strategies can affect the performance of your applications.

For more details about the following locking strategies, see the "Locking" on page 123 topic.

### Pessimistic locking strategy

You can use the pessimistic locking strategy for read and write map operations where keys often collide. The pessimistic locking strategy has the greatest impact on performance.

### Optimistic locking strategy

Optimistic locking is the default configuration. This strategy improves both on performance and scalability over the pessimistic strategy. Use this strategy when your applications can tolerate some optimistic update failures, while still performing better than pessimistic strategy. This strategy works great for read mostly, infrequent update applications.

© Copyright IBM Corp. 2004, 2005

### None locking strategy

Use the none locking strategy is good for applications that are read only. The none locking strategy does not obtain any locks. Therefore, it offers the most concurrency, performance and scalability.

## copyMode method best practices

ObjectGrid makes a copy of the value based on the CopyMode setting. You can use the BackingMap API setCopyMode(CopyMode, valueInterfaceClass) method to set the copy mode to one of the following final static fields that are defined in the com.ibm.websphere.objectgrid.CopyMode class.

When an application uses the ObjectMap interface to obtain a reference to a map value, it is recommended to use that reference only within the ObjectGrid transaction that obtained the reference. Using the reference in a different ObjectGrid transaction can lead to errors. For example, if you use the pessimistic locking strategy for the BackingMap, a get or getForUpdate method call acquires an S (shared) or U (update) lock respectively. The get method returns the reference to the value and the lock that is obtained is released when the transaction completes. The transaction must call the get or getForUpdate method to lock the map entry in a different transaction. Each transaction must obtain its own reference to the value by calling the get or getForUpdate method instead of reusing the same value reference in multiple transactions.

Use the following information to choose between the copy modes with the following information:

### COPY_ON_READ_AND_COMMIT mode

The COPY_ON_READ_AND_COMMIT mode is the default mode. The valueInterfaceClass argument is ignored when this mode is used. This mode ensures that an application does not contain a reference to the value object that is in the BackingMap, and instead the application is always working with a copy of the value that is in the BackingMap. The COPY_ON_READ_AND_COMMIT mode ensures the application can never inadvertently corrupt the data that is cached in the BackingMap. When an application transaction calls an ObjectMap.get method for a given key, and it is the first access of the ObjectMap entry for that key, a copy of the value is returned. When the transaction is committed, any changes committed by the application are copied to the BackingMap to ensure that the application does not have a reference to the committed value in the BackingMap.

### COPY_ON_READ mode

The COPY_ON_READ mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when a transaction is committed. The valueInterfaceClass argument is ignored when this mode is used. To preserve the integrity of the BackingMap data, the application ensures that every reference that it has for an entry is destroyed after the transaction is committed. With this mode, the ObjectMap.get method returns a copy of the value instead of a reference to the value to ensure that changes made by the application to the value does not affect the BackingMap value until the transaction is committed. However, when the transaction does commit, a copy of changes is not made. Instead, the reference to the copy that was returned by the ObjectMap.get method is stored in the BackingMap. The application destroys all map entry references after the transaction is committed. If application fails to do this, the application might cause the data cached in BackingMap to become corrupted. If an application is using this mode and is having problems, switch to COPY_ON_READ_AND_COMMIT mode to see if the problem still exists. If the problem goes away, then the application is failing to destroy all of its references after the transaction has committed.

## COPY_ON_WRITE mode

The COPY_ON_WRITE mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when the ObjectMap.get method is called for the first time by a transaction for a given key. The ObjectMap.get method returns a proxy to the value instead of a direct reference to the value object. The proxy ensures that a copy of the value is not made unless the application calls a set method on the value interface that is specified by the valueInterfaceClass argument. The proxy provides a *copy on write* implementation. When a transaction commits, the BackingMap examines the proxy to determine if any copy was made as a result of a set method being called. If a copy was made, then the reference to that copy is stored in the BackingMap. The big advantage of this mode is that a value is never copied on a read or at a commit when the transaction never calls a set method to change the value.

The COPY_ON_READ_AND_COMMIT and COPY_ON_READ modes both make a deep copy when a value is retrieved from the ObjectMap. If an application only updates some of the values that are retrieved in a transaction then this mode is not optimal. The COPY_ON_WRITE mode supports this behavior in an efficient manner but requires that the application uses a simple pattern. The value objects are required to support an interface. The application must use the methods on this interface when interacting with the value within a ObjectGrid Session. If this is the case, then the ObjectGrid creates proxies for the values that are returned to the application. The proxy has a reference to be real value. If the application just does reads, they always run against the real copy. If the application modifies an attribute on the object, the proxy makes a copy of the real object and then makes the modification on the copy. The proxy then uses the copy from that point on. This allows the copy operation to be avoided completely for objects that are only read by the application. All modify operations must start with the set prefix. Enterprise JavaBeans normally are coded to use this style of method naming for methods that modify the objects attributes. This convention must be followed. Any objects that are modified are copied at the time they are modified by the application. This is the most efficient read and write scenario supported by the ObjectGrid. You can configure a map to use COPY_ON_WRITE as follows. In this example, the application wants to store Person objects keyed using the name in the Map. The person object looks like the following code snippet:

```
class Person
{
 String name;
 int age;
 public Person()
 {
 }
 public void setName(String n)
 {
  name = n;
 }
 public String getName()
 {
  return name;
 }
 public void setAge(int a)
 {
  age = a;
 }
 public int getAge()
 {
  return age;
 }
}
```

The application uses IPerson interface only when interacts with values that are retrieved from a ObjectMap. Modify the object to use an interface as in the following example.

```
interface IPerson
{
 void setName(String n);
 String getName();
 void setAge(int a);
 int getAge();
}
// Modify Person to implement IPerson interface
class Person implements IPerson
{
 ...
}
```

The application then needs to configure the BackingMap to use COPY_ON_WRITE mode, like in the following example:

```
ObjectGrid dg = ...;
BackingMap bm = dg.defineMap("PERSON");
// use COPY_ON_WRITE for this Map with
// IPerson as the valueProxyInfo Class
bm.setCopyMode(CopyMode.COPY_ON_WRITE,IPerson.class);
// The application should then use the following
// pattern when using the PERSON Map.
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
...
sess.begin();
// the application casts the returned value to IPerson and not Person
IPerson p = (IPerson)person.get("Billy");
p.setAge(p.getAge()+1);
...
// make a new Person and add to Map
Person p1 = new Person();
p1.setName("Bobby");
p1.setAge(12);
person.insert(p1.getName(), p1);
sess.commit();
// the following snippet WON'T WORK. Will result in ClassCastException
sess.begin();
// the mistake here is that Person is used rather than
// IPerson
Person a = (Person)person.get("Bobby");
sess.commit();
```

The first section shows the application retrieving a value that was named Billy in the map. The application casts the returned value to IPerson object, and not the Person object because the proxy that is returned implements two interfaces:

- The interface specified in the BackingMap.setCopyMode method call.
- The com.ibm.websphere.objectgrid.ValueProxyInfo interface

You can cast the proxy to two types. The last part of the preceding code snippet demonstrates what is not allowed in COPY_ON_WRITE mode. The application retrieves the Bobby record and tries to cast it to a Person object. This action fails with a class cast exception because the proxy that is returned is not a Person object. The returned proxy implements the IPerson object and ValueProxyInfo.

**ValueProxyInfo interface and partial update support**

This interface allows an application to retrieve either the committed read-only value referenced by the proxy or the set of attributes that have been modified during this transaction.

```
public interface ValueProxyInfo
{
List /**/ ibmGetDirtyAttributes();
Object ibmGetRealValue();
}
```

The ibmGetRealValue method returns a read only copy of the object. The application must not modify this value. The ibmGetDirtyAttributes method returns a list of strings representing the attributes that have been modified by the application during this transaction. The main use case for ibmGetDirtyAttributes is in a Java database connectivity (JDBC) or CMP based loader. Only the attributes that are named in the list need be updated on either the SQL statement or object mapped to the table, which leads to more efficient SQL generated by the Loader. When a copy on write transaction is committed and if a loader is plugged in, the the loader can cast the values of the modified objects to the ValueProxyInfo interface to obtain this information.

**Handling the equals method when using COPY_ON_WRITE or proxies.**

For example, the following code constructs a Person object and then inserts it to a an ObjectMap. Next, it retrieves the same object using ObjectMap.get method. The value is cast to the interface. If the value is cast to the Person interface, a ClassCastException exception results because the returned value is a proxy that implements the IPerson interface and is not a Person object. The equality check fails when using the == operation because they are not the same object.

```
session.begin();
// new the Person object
Person p = new Person(...);
personMap.insert(p.getName, p);
// retrieve it again, remember to use the interface for the cast
IPerson p2 = personMap.get(p.getName());
if(p2 == p)
{
// they are the same
}
else
{
// they are not
}
```

Another consideration is when you must override the equals method. As illustrated in the following snippet of code, the equals method must verify that the argument is an object that implements IPerson interface and cast the argument to be a IPerson. Because the argument might be a proxy that implements the IPerson interface, you must use the getAge and getName methods when comparing instance variables for equality.

```
public boolean equals(Object obj)
{
 if ( obj == null ) return false;
 if ( obj instanceof IPerson )
 {
  IPerson x = (IPerson) obj;
  return ( age.equals( x.getAge() ) && name.equals( x.getName() ) )
 }
 return false;
}
```

### NO_COPY mode

The NO_COPY mode allows an application to ensure that it never modifies a value object that is obtained using an `ObjectMap.get` method in exchange for performance improvements. The valueInterfaceClass argument is ignored when this mode is used. If this mode is used, no copy of the value is ever made. If the application does modify values, then data in the BackingMap is corrupted. The NO_COPY mode is primarily useful for read-only maps where data is never modified by the application. If the application is using this mode and it is having problems, then switch to the COPY_ON_READ_AND_COMMIT mode to see if the problem still exists. If the problem goes away, then the application is modifying the value returned by ObjectMap.get method, either during transaction or after transaction has committed.

## ObjectTransformer interface best practices

ObjectTransformer uses callbacks to the application to provide custom implementations of common and expensive operations such as object serialization and a deep copy on an object.

For specific details about the ObjectTransformer interface, see the “ObjectTransformer plug-in” on page 202 topic. From a performance point of view and information from the CopyMode method in the “copyMode method best practices” on page 316 topic, it is clear that ObjectGrid copies the values for all cases except when in NO_COPY mode. The default copying mechanism that is employed within ObjectGrid is serialization, which is known as an expensive operation. The ObjectTransformer interface can be used in this situation. The ObjectTransformer interface uses callbacks to the application to provide a custom implementation of common and expensive operations such as object serialization and deep copies on objects.

An application can provide an implementation of the ObjectTransformer interface to a map. The ObjectGrid then delegates to the methods on this object and relies on the application to provide an optimized version of each method in the interface. The ObjectTransformer interface follows:

```
public interface ObjectTransformer
{
 void serializeKey(Object key, ObjectOutputStream stream)
 throws IOException;
 void serializeValue(Object value, ObjectOutputStream stream)
 throws IOException;
 Object inflateKey(ObjectInputStream stream)
 throws IOException, ClassNotFoundException;
 Object inflateValue(ObjectInputStream stream)
 throws IOException, ClassNotFoundException;
 Object copyValue(Object value);
 Object copyKey(Object key);
}
```

You can associate an ObjectTransformer interface with a BackingMap by using the following example code:

```
ObjectGrid g = ...;
BackingMap bm = g.defineMap("PERSON");
MyObjectTransformer ot = new MyObjectTransformer();
bm.setObjectTransformer(ot);
```

### Tune object serialization and inflation

Object serialization is usually the biggest performance issue with ObjectGrid. ObjectGrid uses the default serializable mechanism if an ObjectTransformer plug-in is not supplied by the application. An application can provide implementations of either the Serializable readObject and writeObject or it can have the objects implement the Externalizable interface, which is around 10 times faster. If the objects in the Map cannot be modified, then an application can associate an ObjectTransformer with the ObjectMap. The serialize and inflate methods are provided to allow the application to provide custom code to optimize these operations given their large performance impact on the system. The serialize methods serialize the object and provide a stream. The method serializes the method to the provided stream. The inflate methods provide the input stream and expect the application to create the object, inflate it using data in the stream and then return the object. The implementations of the serialize and inflate methods must mirror each other.

### Tune deep copy operations

After an application receives an object from an ObjectMap then the ObjectGrid performs a deep copy on the object value to ensure that the copy in the BaseMap map stays safe. The application can then modify the object value safely. When the transaction commits, the copy of the object value in the BaseMap map is updated to the new modified value and the application stops using the value from that point on. You could have copied the object again at the commit phase to make a private copy, but in this case the performance cost of this action was traded off against telling the application programmer to not use the value after the transaction commits. The default object copy mechanism attempts to use either a clone or a serialize and inflate pair to generate a copy. The serialize and inflate pair is the worst case performance scenario. If profiling reveals that serialize and inflate is a problem for your application, provide an ObjectTransformer plug-in and implement the `copyValue` and `copyKey` methods using a more efficient object copy.

## Plug-in evictor performance best practices

If you use plug-in evictors, they are not active until you create them and tell the backing map to use them. Use these best practices and performance tips for least frequently used (LFU) and least recently used (LRU) evictors.

### Least frequently used (LFU) evictor

The concept of a LFU evictor is to remove entries from the map that are used infrequently. The entries of the map are spread over a set amount of binary heaps. As the usage of a particular cache entry grows, it becomes ordered higher in the heap. When the evictor attempts a set of evictions it removes only the cache entries that are located lower than a specific point on the binary heap. As a result, the least frequently used entries are evicted.

## Least recently used (LRU) evictor

The LRU Evictor follows the same concepts of the LFU Evictor with a few differences. The main difference is that the LRU uses a first in, first out queue (FIFO) instead of a set of binary heaps. Every time a cache entry is accessed, it moves to the head of the queue. Consequently, the front of queue contains the most recently used map entries and the end becomes the least recently used map entries. For example, the A cache entry is used 50 times, and the B cache entry is used only once right after the A cache entry. In this situation, the B cache entry is at the front of the queue because it was used most recently, and the A cache entry is at the end of the queue. The LRU evictor evicts the cache entries that are at the tail of the queue, which are the least recently used map entries.

## LFU and LRU properties and best practices to improve performance

**Number of heaps**
When using the LFU evictor, all of the cache entries for a particular map are ordered over the number of heaps that you specify, improving performance drastically and preventing all of the evictions from synchronizing on one binary heap that contains all of the ordering for the map. More heaps also speeds up the time that is required for reordering the heaps because each heap has fewer entries. Set the number of heaps to 10% of the number of entries in your BaseMap.

**Number of queues**
When using the LRU evictor, all of the cache entries for a particular map are ordered over the number of LRU queues that you specify, improving performance drastically and preventing all of the evictions from synchronizing on one queue that contains all of the ordering for the map. Set the number of queues to 10% of the number of entries in your BaseMap.

**MaxSize property**
When an LFU or LRU evictor begins evicting entries, it uses the MaxSize evictor property to determine how many binary heaps or LRU queue elements to evict. For example, assume that you set the number of heaps or queues to have about 10 map entries in each map queue. If your MaxSize property is set to 7, the evictor evicts 3 entries from each heap or queue object to bring the size of each heap or queue back down to 7. The evictor only evicts map entries from a heap or queue when that heap or queue has more than the MaxSize property value of elements in it. Set the MaxSize to 70% of your heap or queue size. For this example, the value is set to 7. You can get an approximate size of each heap or queue by dividing the number of BaseMap entries by the number of heaps or queues that are used.

**SleepTime property**
An evictor does not constantly remove entries from your map. Instead it sleeps for a set amount of time, only waking every *n* number of seconds , where *n* refers to the SleepTime property. This property also positively affects performance: running an eviction sweep too often lowers performance because of the resources that are needed for processing them. However, not using the evictor enough can leave you with a map of unneeded entries. A map full of unneeded entries can negatively affect both the memory requirements and processing resources that required for your map. Setting the eviction sweeps to fifteen seconds is a good practice for most maps. If the map is written to frequently and is used at a high

transaction rate, it might be more useful to set the value to a lower time. However, if the map is accessed very infrequently, you can set the time to a higher value.

### Example

The following example defines a map, creates a new LFU evictor, sets the evictor properties, and sets the map to use the evictor:

```
//Use ObjectGridManager to create/get the ObjectGrid. Refer to
// the ObjectGridManger section
ObjectGrid objGrid = ObjectGridManager.create............
BackingMap bMap = objGrid.defineMap("SomeMap");

//Set properties assuming 50,000 map entries
LFUEvictor someEvictor = new LFUEvictor();
someEvictor.setNumberOfHeaps(5000);
someEvictor.setMaxSize(7);
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

Using the LRU evictor is very similar to using an LFU evictor. Following is an example:

```
ObjectGrid objGrid = new ObjectGrid;
BackingMap bMap = objGrid.defineMap("SomeMap");

//Set properties assuming 50,000 map entries
LRUEvictor someEvictor = new LRUEvictor();
someEvictor.setNumberOfLRUQueues(5000);
someEvictor.setMaxSize(7);
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

Notice that only 2 lines are different from the LFUEvictor example.

## Default evictor best practices

Best practices for the default *time to live* evictor.

In addition to the plug-in evictors that are described in the “Plug-in evictor performance best practices” on page 321 topic, a default TTL evictor is created with every backing map. The default evictor removes entries based on a *time to live* concept. This behavior is defined by the ttlType attribute. Three ttlTypes attributes exist:

- **None** : Specifies that entries never expire and therefore are never removed from the map.
- **Creation time** : Specifies that entries are evicted depending on when they were created.
- **Last accessed time** : Specifies that entries are evicted depending upon when they were last accessed.

### Default evictor properties and best practices for performance

#### TimeToLive property

This property, along with ttlType property, is the most crucial from a performance perspective. If you are using the CREATION_TIME ttlType, the evictor evicts an entry when its time from creation equals its TimeToLive attribute value. If you set the TimeToLive attribute value to 10 seconds, everything in the entire map is evicted after ten seconds. It is important to take caution when setting this value for the CREATION_TIME ttlType. This evictor is best used when reasonably high amounts of additions to the cache exist that are only used for a set amount of time. With this strategy, anything that is created is removed after the set amount of time.

Following is an example of where a TTL type of CREATION_TIME is useful. You are using a Web application that obtains stock quotes, and getting the most recent quotes is not critical. In this case, the stock quotes are cached in an ObjectGrid for 20 minutes. After 20 minutes, the ObjectGrid map entries expire and are evicted. Every twenty minutes or so the ObjectGrid map uses the Loader plug-in to refresh the map data with fresh data from the database. The database is updated every 20 minutes with the most recent stock quotes. So for this application, using a TimeToLive value of 20 minutes is ideal.

If you are using the LAST_ACCESSED_TIME ttlType attribute, set the TimeToLive to a lower number than if you are using the CREATION_TIME ttlType, because the entries TimeToLive attribute is reset every time it is accessed. In other words, if the TimeToLive attribute is equal to 15 and an entry has existed for 14 seconds but then gets accessed, it does not expire again for another 15 seconds. If you set the TimeToLive to a relatively high number, many entries might never be evicted. However, if you set the value to something like 15 seconds, entries might be removed when they are not often accessed.

Following is an example of where a TTL type of LAST_ACCESSED_TIME is useful. An ObjectGrid map is used to hold session data from a client. Session data must be destroyed if the client does not use the session data for some period of time. For example, the session data times out after 30 minutes of no activity by the client. In this case, using a TTL type of LAST_ACCESSED_TIME with the TimeToLive attribute set to 30 minutes is exactly what is needed for this application.

**Example**

The following example creates a backing map, set its default evictor ttlType attribute, and sets its TimeToLive property.

```
ObjectGrid objGrid = new ObjectGrid;
BackingMap bMap = objGrid.defineMap("SomeMap");
bMap.setTtlEvictorType(TTLType.LAST_ACCESSED_TIME);
bMap.setTimeToLive(15);
```

Most evictor settings should be set prior to ObjectGrid initialization. For a more in-depth understanding of the evictors, see “Evictors” on page 182.

# Chapter 12. Distributing changes between peer Java virtual machines

The LogSequence and LogElement objects communicate the changes that have occurred in an ObjectGrid transaction with a plug-in.

For more information about how Java Message Service (JMS) can be used to distribute transactional changes, see “Java Message Service for distributing transaction changes” on page 328.

A prerequisite is that the ObjectGrid instance must be cached by the ObjectGridManager. See“createObjectGrid methods” on page 87 for more information. The cacheInstance boolean value must be set to true.

The objects provide a means for an application to easily publish changes that have occurred in an object grid using a message transport to peer ObjectGrids in remote Java virtual machines (JVM) and then apply those changes on that JVM. The LogSequenceTransformer class is critical to enabling this support. This article examines how to write a listener using a Java Message Service (JMS) messaging system for propagating the messages.

ObjectGrid supports transmitting LogSequences that result from an ObjectGrid transaction commit across WebSphere Application Server cluster members with an IBM-provided plug-in. This function is not enabled by default, but can be configured to be operational. However, when either the consumer or producer is not a WebSphere Application Server, using an external JMS messaging system might be required.

1. Initialize the plug-in. The ObjectGrid calls the `initialize` method of the plug-in, part of the ObjectGridEventListener interface contract, when the ObjectGrid starts. The `initialize` method must obtain its JMS resources, including connections, sessions, and publishers, and start the thread that is the JMS listener. The initialize method looks like the following example:

```
public void initialize(Session session)
{
 mySession = session;
 myGrid = session.getObjectGrid();
 try
 {
  if(mode == null)
  {
   throw new ObjectGridRuntimeException("No mode specified");
  }
  if(userid != null)
  {
   connection = topicConnectionFactory.createTopicConnection(
   userid, password);
  }
  else
   connection = topicConnectionFactory.createTopicConnection();

  // need to start the connection to receive messages.
  connection.start();

  // the jms session is not transactional (false).
  jmsSession = connection.createTopicSession(false,
  javax.jms.Session.AUTO_ACKNOWLEDGE);
  if(topic == null)
   if(topicName == null)
```

```
    {
     throw new ObjectGridRuntimeException("Topic not specified");
    }
    else
    {
     topic = jmsSession.createTopic(topicName);
    }
   publisher = jmsSession.createPublisher(topic);
   // start the listener thread.
   listenerRunning = true;
   listenerThread = new Thread(this);
   listenerThread.start();
  }
  catch(Throwable e)
  {
   throw new ObjectGridRuntimeException("Cannot initialize", e);
  }
 }
```

The code to start the thread uses a Java 2 Platform, Standard Edition (J2SE) thread. If you are running a WebSphere Application Server Version 6.x or a WebSphere Application Server Version 5.x Enterprise server, use the asynchronous bean application programming interface (API) to start this daemon thread. You can also use the common APIs. Following is an example replacement snippet showing the same action using a work manager:

```
// start the listener thread.
listenerRunning = true;
workManager.startWork(this, true);
```

The plug-in must also implement the Work interface instead of the Runnable interface. You also need to add a release method to set the listenerRunning variable to false. The plug-in must be provided with a WorkManager instance in its constructor or by injection if using an Inversion of Control (IoC) container.

2. Transmit the changes. Following is a sample transactionEnd method for publishing the local changes that are made to an ObjectGrid. This uses JMS although clearly, you can use any message transport that is capable of reliable publish and subscribe messaging.

```
// This method is synchronized to make sure the
// messages are published in the order the transaction
// were committed. If we started publishing the messages
// in parallel then the receivers could corrupt the Map
// as deletes may arrive before inserts etc.
public synchronized void transactionEnd(String txid,
boolean isWriteThroughEnabled,
boolean committed, Collection changes)
{
 try
 {
  // must be write through and commited.
  if(isWriteThroughEnabled && committed)
  {
   // write the sequences to a byte []
   ByteArrayOutputStream bos = new ByteArrayOutputStream();
   ObjectOutputStream oos = new ObjectOutputStream(bos);
   if (publishMaps.isEmpty()) {
    // serialize the whole collection
    LogSequenceTransformer.serialize(changes, oos, this, mode);
   }
   else {
    // filter LogSequences based on publishMaps contents
    Collection publishChanges = new ArrayList();
    Iterator iter = changes.iterator();
    while (iter.hasNext()) {
      LogSequence ls = (LogSequence) iter.next();
      if (publishMaps.contains(ls.getMapName())) {
      publishChanges.add(ls);
     }
    }
    LogSequenceTransformer.serialize(publishChanges, oos, this,
mode);
  }
  // make an object message for the changes
  oos.flush();
  ObjectMessage om = jmsSession.createObjectMessage(
  bos.toByteArray());
  // set properties
  om.setStringProperty(PROP_TX, txid);
  om.setStringProperty(PROP_GRIDNAME, myGrid.getName());
  // transmit it.
  publisher.publish(om);
  }
 }
 catch(Throwable e)
 {
  throw new ObjectGridRuntimeException("Cannot push changes", e);
 }
}
```

This method uses several instance variables:

- **jmsSession** variable: A JMS session that is used to publish messages. It is created when the plug-in initializes
- **mode** variable: The distribution mode.
- **publishMaps** variable: A set that contains the name of each Map with changes to publish. If the variable is empty, then all the Maps are published.
- **publisher** variable: A TopicPublisher object that is created during the plug-in initialize method.

3. Receive and apply update messages. Following is the run method. This method runs in a loop until the application stops the loop. Each loop iteration attempts to receive a JMS message and apply it to the ObjectGrid.

```
private synchronized boolean isListenerRunning()
{
 return listenerRunning;
}
public void run()
{
 try
 {
  System.out.println("Listener starting");
  // get a jms session for receiving the messages.
  // Non transactional.
  TopicSession myTopicSession;
  myTopicSession = connection.createTopicSession(false,
  javax.jms.Session.AUTO_ACKNOWLEDGE);

  // get a subscriber for the topic, true indicates don't receive
  // messages transmitted using publishers
  // on this connection. Otherwise, we'd receive our own updates.
  TopicSubscriber subscriber = myTopicSession.createSubscriber(topic,
  null, true);
  System.out.println("Listener started");
  while(isListenerRunning())
  {
   ObjectMessage om = (ObjectMessage)subscriber.receive(2000);
   if(om != null)
   {
    // Use Session that was passed in on the initialize...
```

```
        // very important to use no write through here
        mySession.beginNoWriteThrough();
        byte[] raw = (byte[])om.getObject();
        ByteArrayInputStream bis = new ByteArrayInputStream(raw);
        ObjectInputStream ois = new ObjectInputStream(bis);
        // inflate the LogSequences
        Collection collection = LogSequenceTransformer.inflate(ois,
         myGrid);
        Iterator iter = collection.iterator();
        while (iter.hasNext()) {
         // process each Maps changes according to the mode when
         // the LogSequence was serialized
         LogSequence seq = (LogSequence)iter.next();
         mySession.processLogSequence(seq);
        }
        mySession.commit();
       } // if there was a message
      } // while loop
      // stop the connection
      connection.close();
     }
     catch(IOException e)
     {
      System.out.println("IO Exception: " + e);
     }
     catch(JMSException e)
     {
      System.out.println("JMS Exception: " + e);
     }
     catch(ObjectGridException e)
     {
      System.out.println("ObjectGrid exception: " + e);
      System.out.println("Caused by: " + e.getCause());
     }
     catch(Throwable e)
     {
      System.out.println("Exception : " + e);
     }
     System.out.println("Listener stopped");
    }
```

The LogSequenceTransformer class, and the ObjectGridEventListener, LogSequence and LogElement APIs allow any reliable publish and subscribe to be used to distribute the changes and filter the Maps that you want to distribute. The snippets in this task show how to use these APIs with JMS to build a peer-to-peer ObjectGrid that is shared by applications that are hosted on a diverse set of platforms that share a common message transport.

## Java Message Service for distributing transaction changes

Use Java Message Service (JMS) for distributed changes between different tiers or in environments on mixed platforms.

JMS is an ideal protocol for distributed changes between different tiers or in environments on mixed platforms. For example, some applications that use the ObjectGrid might be deployed on Gluecode or Tomcat, where as other applications might run on WebSphere Application Server Version 6.0. JMS is ideal for distributed changes between ObjectGrid peers in these different environments. The high availability manager message transport is very fast, but can only distribute changes to JVMs that are in a single core group. JMS is slower, but allows larger and a more diverse set of application clients to share an ObjectGrid. JMS is ideal for example when sharing data in an ObjectGrid between a fat Swing client and an application deployed on WebSphere Extended Deployment.

## Overview

JMS is implemented for distributing transaction changes by using a Java object that behaves as an ObjectGridEventListener listener. This object can propagate the state in the following four ways:

**Invalidate**
: Any entry that is evicted, updated or deleted is removed on all peer Java Virtual Machines (JVMs) when they receive the message.

**Invalidate conditional**
: The entry is evicted only if the local version is the same or older than the version on the publisher.

**Push**
: Any entry that was evicted, updated, deleted or inserted is added or overwritten on all peer JVMs when they receive the JMS message.

**Push conditional**
: The entry is only updated or added on the receive side if the local entry is less recent than the version that is being published.

## Listen for changes for publishing

The plug-in implements the ObjectGridEventListener interface to intercept the transactionEnd event. When the ObjectGrid invokes this method, the plug-in attempts to convert the LogSequence list for each Map that is touched by the transaction to a JMS message and then publish it. The plug-in can be configured to publish changes for all Maps or a subset of Maps. LogSequence objects are processed for the Maps that have publishing enabled. The LogSequenceTransformer ObjectGrid class serializes a filtered LogSequence for each Map to a stream. After all LogSequences are serialized to the stream then a JMS ObjectMessage is created and published to a well known topic.

## Listen for JMS messages and apply them to the local ObjectGrid

The same plug-in also starts a thread that spins in a loop, receiving all messages that are published to the well known topic. When a message arrives, it passes the message contents to the LogSequenceTransformer class to convert it to a set of LogSequence objects. Then, a no write through transaction is started. Each LogSequence object is provided to the Session.processLogSequence method, which updates the local Maps with the changes. The processLogSequence method understands the distribution mode. The transaction is committed and the local cache now reflects the changes.

For more information about using JMS to distribute transaction changes, see Chapter 12, “Distributing changes between peer Java virtual machines,” on page 325.

# Chapter 13. Injection-based container integration

You can use the Inversion of Control (IoC) framework to completely configure the ObjectGrid. As a result, you do not need to use the ObjectGrid XML configuration framework.

## Injection-based containers

Injection based containers, also known as Inversion of Control (IoC), is a common pattern that is used by applications on the client side and on the server side. Several open source implementations of such containers exist. The new Enterprise JavaBeans (EJB) Version 3.0 specification also borrows some of these concepts. Most of these frameworks are Enterprise JavaBean containers, and take the responsibility of creating an instance of a particular bean. These frameworks can also initialize a bean with a set of properties and wire other Enterprise JavaBeans that it requires by using getter and setter pairs on Enterprise JavaBean attributes. The ObjectGrid application programming interfaces (API) are designed to work well with such containers. Starting with the ObjectGridManager.createObjectGrid methods, you can configure these containers to bootstrap an ObjectGrid so that the application has a reference to either a working ObjectGrid or can ask for the container to provide an ObjectGrid session when required.

## Supported patterns

The following sections discuss what has been done to verify that the ObjectGrid APIs can be used cleanly by an application that employs an IoC framework.

### Use ObjectGridManager to create ObjectGrids

The ObjectGrid APIs are designed to work well with IoC frameworks. The root singleton used by such a framework is the ObjectGridManager interface, which has several `createObjectGrid` factory methods that return a reference to a named ObjectGrid. You can set this ObjectGrid reference as a singleton in the IoC framework, so that subsequent requests for the bean return the same ObjectGrid instance.

### ObjectGrid Plug-ins

The plug-ins on the ObjectGrid include:
- TransactionCallback
- ObjectGridEventListener
- SubjectSource
- MapAuthorization
- SubjectValidation

Each of the plug-ins are simple JavaBeans that implement an interface. You can use the IoC framework to create these plug-ins and wire them to the appropriate attributes on the ObjectGrid instance. Each of these plug-ins has a corresponding set method on the ObjectGrid interface for clean integration with the IoC framework.

### Create maps

The createMap factory method on the ObjectGrid interface can be used to create a newly named Map. Any plug-ins that are required by the BackingMap (the object returned by createMap) are constructed using the IoC framework and then wired to the BackingMap by using the appropriate attribute name. Because the BackingMap is not referenced by any other object, the IoC frameworks do not automatically construct it. You can wire each BackingMap to a dummy attribute on the main application bean as a work around. Use the setMaps() method to clear any BackingMaps that have been previously defined on this ObjectGrid and replace them with the list of BackingMaps that are provided.

**Backing map plug-ins**

The plug-ins on the BackingMap behave in the same manner as those on the ObjectGrid. Each plug-in has a corresponding `set` method on the BackingMap interface. The BackingMap plug-ins are:

- Loader
- ObjectTransformer
- OptimisticCallback
- Evictor
- MapEventListener

## Usage patterns

After the IoC framework has its configuration file set up to produce an ObjectGrid, create an enterprise bean that is called `gridName_Session` or something similar. Define it as an Enterprise JavaBean that is obtained by calling the `getSession` method on the ObjectGrid singleton Enterprise JavaBean. The application then uses the IoC framework to obtain a reference to a gridName_Session object whenever an object requires a new session.

## Summary

Using the ObjectGrid in environments that already use an IoC framework for bean instantiation and configuration is straightforward. You can use the IoC framework to completely configure the ObjectGrid, and as a result, you do not need to use the XML configuration framework. The ObjectGrid works seamlessly with your existing IoC framework.

# Chapter 14. Troubleshooting

This section describes scenarios for troubleshooting problems that are caused by an application error or an application design issue. If you suspect ObjectGrid has a defect, you might need to enable tracing of ObjectGrid as described in the tracing section of ObjectGridManager.

## Intermittent and unexplained errors

An application attempts to improve performance by using the COPY_ON_READ, COPY_ON_WRITE, or NO_COPY copy mode as described in the CopyMode section. The application encounters intermittent problems when the symptom of the problem is changing and the problem is unexplained or unexpected. Intermittent problems do not occur when the copy mode is changed to the COPY_ON_READ_AND_COMMIT mode.

### Problem

The problem might be due to corrupted data in the ObjectGrid map, which is a result of the application violating the programming contract of the copy mode that is being used. Data corruption can cause unpredictable errors to occur intermittently or in an unexplained or unexpected fashion.

### Solution

The solution is for the application to comply with the programming contract stated for the copy mode being used. For the COPY_ON_READ and COPY_ON_WRITE copy modes, this means the application uses a reference to a value object outside of the transaction scope where the value reference was obtained. To use these modes, the application must agree to destroy the reference to the value object after the transaction is completed and to obtain a new reference to the value object in each transaction that needs to access the value object. For the NO_COPY copy mode, the application must agree to never change the value object. In this case, the application must either be changed so that it does not change the value object or the application must use a different copy mode. See the CopyMode section for additional details regarding the copy mode setting.

## General exception handling technique

Knowing the root cause of a Throwable object is helpful in isolating the source of a problem. The following is an example of a utility method that can be used by an exception handler to find the root cause of the Throwable that occurred.

### Example

```
static public Throwable findRootCause( Throwable t )
{
     // Start with Throwable that occurred as the root.
     Throwable root = t;

     // Follow cause chain until last Throwable in chain is found.
     Throwable cause = root.getCause();
     while ( cause != null )
     {
        root = cause;
        cause = root.getCause();
     }
```

```
          // Return last Throwable in the chain as the root cause.
          return root;
     }
```

## Specific exception handling techniques

### Duplicate insert

This problem should typically only occurs in a distributed transaction propagation environment. It does not happen often.

### Message

```
[7/11/05 22:02:11:303 CDT] 00000032 SessionImpl < processLogSequence Exit
[7/11/05 22:02:11:303 CDT] 00000032 SessionImpl > commit for:
TX:08FE0C67-0105-4000-E000-1540090A5759 Entry
[7/11/05 22:02:11:303 CDT] 00000032 SessionImpl > rollbackPMapChanges for:
TX:08FE0C67-0105-4000-E000-1540090A5759
as result of Throwable: com.ibm.websphere.objectgrid.plugins.
CacheEntryException:
Duplicate key on an insert!
Entry com.ibm.websphere.objectgrid.plugins.CacheEntryException:
Duplicate key on an insert!
at com.ibm.ws.objectgrid.map.BaseMap.applyPMap(BaseMap.java:528)
at com.ibm.ws.objectgrid.SessionImpl.commit(SessionImpl.java:405)
at com.ibm.ws.objectgrid.plugins.TranPropWorkerThread.commitPropagatedLogSequence
(TranPropWorkerThread.java:553)
at com.ibm.ws.objectgrid.plugins.TranPropWorkerThread.processCommitRequest
(TranPropWorkerThread.java:449)
at com.ibm.ws.objectgrid.plugins.TranPropWorkerThread.run
(TranPropWorkerThread.java:200)
at java.lang.Thread.run(Thread.java:568)
```

### Problem

When the filtered log sequence is propagated from one JVM to another, the foreign log sequence is processed in the second JVM. The entry for this key may exist or the two log sequence operation codes should be different This problem happens occasionally.

### Impact and solution

When this problem occurs, the entry is not updated in another JVM which can cause an inconsistency in ObjectGrid. However, a workaround exists to avoid this problem. You can use the partitioning facility (WPF) on object retrieval in addition to the object insert/update/remove. Reference the Integrating WPF and ObjectGrid section for more information on this technique.

## Optimistic collision exception

You can receive an OptimisticCollisionException exception directly, or receive it while receiving an ObjectGridException exception. The following code is an example of how to catch the exception and then display its message:

```
try {
...
} catch (ObjectGridException oe) {
System.out.println(oe);
}
```

### Exception cause

An OptimisticCollisionException exception is created in a situation where two different clients try to update the same map entry at relatively the same time. One client's session is committed and updates the map entry. However, the other client has already read the data before the commit and contains old or incorrect data. The other client will then attempt to commit the incorrect data, which is when the exception is created.

### Retrieving the key that triggered the exception

It might be useful, when troubleshooting such an exception, to retrieve the key corresponding to the entry which triggered the exception. The benefit of the OptimisticCollisionException is that it has a built in method getKey that returns the object representing that key. Following is an example on how to retrieve and print the key when catching the OptimisticCollisionException :

```
try {
...
} catch (OptimisticCollisionException oce) {
 System.out.println(oce.getKey());
}
```

An OptimisticCollisionException might be the cause of an ObjectGridException. If this is the case, you can use the following code to determine the exception type and print out the key. The code below uses the findRootCause utility method as described in the General Exception Handling Technique section.

```
try {
...
}
catch (ObjectGridException oe) {
 Throwable Root = findRootCause( oe );
 if (Root instanceof OptimisticCollisionException) {
  OptimisticCollisionException oce = (OptimisticCollisionException)Root;
  System.out.println(oce.getKey());
 }
}
```

## LockTimeoutException exception

### Message

You can catch a LockTimeoutException exception directly, or while catching an ObjectGridException. The following code snippet shows how to catch the exception and display the message.

```
try {
 ...
}
catch (ObjectGridException oe) {
 System.out.println(oe);
}
```

The result is:

```
com.ibm.websphere.objectgrid.plugins.LockTimeoutException: %Message
```

%Message represents the string that is passed as a parameter when the exception is created and the exception properties and methods are used to display an accurate error message. It most likely describes the type of lock that was requested, and which map entry the transaction acted upon.

## Exception cause

When a transaction or client is asking for a lock to be granted for a specific map entry it will often have to wait for the current client to release the lock. If the lock request remains idle for an extended period of time, and a lock never gets granted, a `LockTimeoutException` is created. This is to prevent a deadlock, which is described in more detail in the following section. You are more likely to see this exception when using a pessimistic locking strategy because the lock is never released until the transaction is committed.

## Getting more details about the lock request and exception

The `LockTimeoutException` has a built in method called getLockRequestQueueDetails which returns a string that contains a more in-depth description of the situation that triggered the exception. The following is an example of some code that catches the exception, and displays an error message.

```
try {
 ...
}
catch (LockTimeoutException lte) {
 System.out.println(lte.getLockRequestQueueDetails());
}
```

The output result is:

```
lock request queue
->[TX:163C269E-0105-4000-E0D7-5B3B090A571D, state =
 Granted 5348 milli-seconds ago, mode = U]
->[TX:163C2734-0105-4000-E024-5B3B090A571D, state =
 Waiting for 5348 milli-seconds, mode = U]
->[TX:163C328C-0105-4000-E114-5B3B090A571D, state =
 Waiting for 1402 milli-seconds, mode = U]
```

If you get the exception in an `ObjectGridException` catch block, the following code determines the exception and print out the queue details. It uses the findRootCause utility method described in General Exception Handling Technique section.

```
try {
...
}
catch (ObjectGridException oe) {
 Throwable Root = findRootCause( oe );
 if (Root instanceof LockTimeoutException) {
  LockTimeoutException lte = (LockTimeoutException)Root;
  System.out.println(lte.getLockRequestQueueDetails());
 }
}
```

## Possible solution

The `LockTimeoutException` is to prevent possible deadlocks in your application. An exception of this type will be thrown when it has waited a set amount of time. The amount of time it waits, however, can be set using the `setLockTimeout(int)` method which is available for the BackingMap. Using the setLockTimeout method eliminates the `LockTimeoutException`. If a deadlock does not actually exist in your application, adjusting the lock timeout can help you avoid the LockTimeoutException.

The following code shows how to create an ObjectGrid, define a map, and set its LockTimeout value to 30 seconds

```
ObjectGrid objGrid = new ObjectGrid();
BackingMap bMap = objGrid.defineMap("MapName");
//This will set the amount of time that a
// lock request will wait before an exception is thrown
bMap.setLockTimeout(30);
```

The previous code can be used for hard-coding ObjectGrid and map properties. If you create ObjectGrid from an XML file, the LockTimeout property can be set within the backingMap tag. Following is an example of a backingMap tag that sets a map LockTimeout value to 30 seconds.

```
<backingMap name="MapName" lockStrategy="PESSIMISTIC" lockTimeout="30">
```

## LockDeadlockException

### Message

You can catch a LockDeadLockException directly, or get it while catching an ObjectGridException. Following is a code example that shows catching the exception, then the displayed message.

```
try {
...
} catch (ObjectGridException oe) {
System.out.println(oe);
}
```

The result is:

```
com.ibm.websphere.objectgrid.plugins.LockDeadlockException: %Message
```

%Message represents the string that is passed as a parameter when the exception is created and thrown.

### Exception cause

The most common type of deadlock happens when using the pessimistic lock strategy, and two separate clients each own a shared lock on a particular object. Then, both attempt to promote to an exclusive lock on that object. Following is a diagram that shows such a situation with transaction blocks that would cause the exception to be thrown.

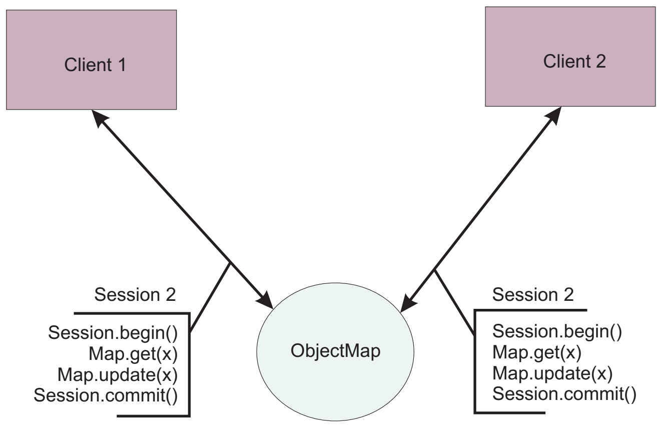


*Figure 23. Example of a potential deadlock situation*

This is an abstract view of what is occurring in your program when the exception occurs. In an application with many threads updating the same ObjectMap, it is possible to encounter this situation. The following is a step-by-step example of when two clients execute the transaction code blocks, described in the previous figure.

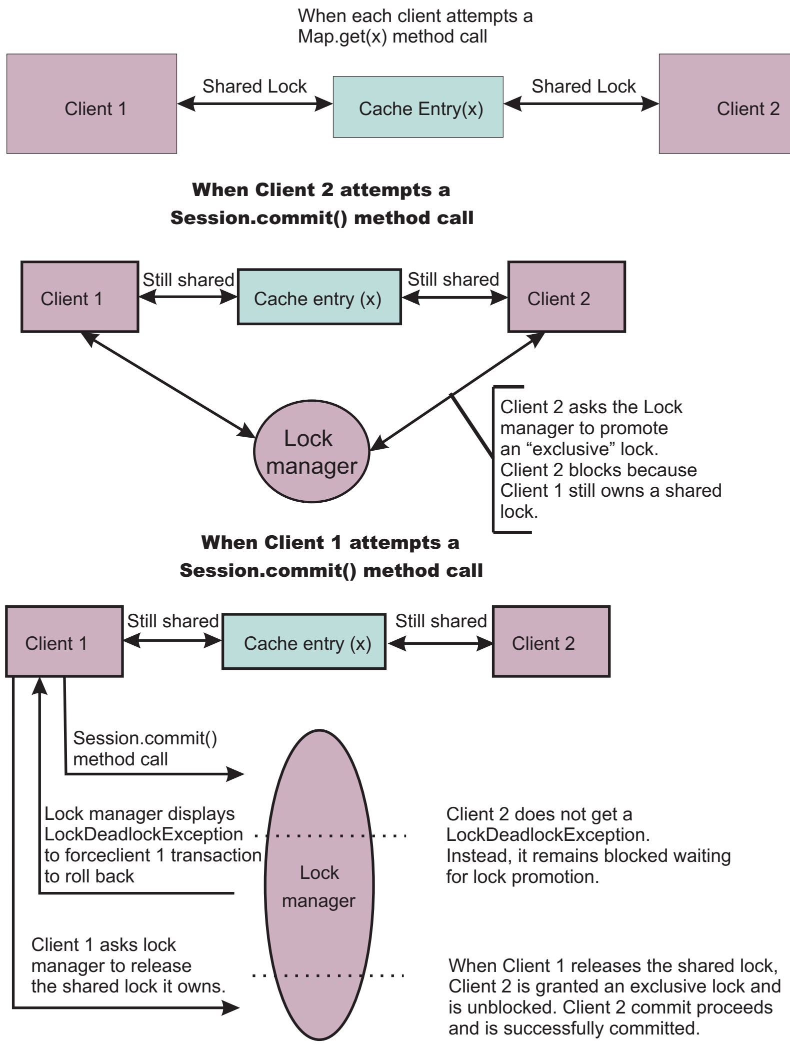

When each client attempts a Map.get(x) method call
Client 1
Shared Lock
Cache Entry(x)
Shared Lock
Client 2
When Client 2 attempts a Session.commit() method call
Client 1
Still shared
Cache entry (x)
Still shared
Client 2
Lock manager
Client 2 asks the Lock manager to promote an "exclusive" lock. Client 2 blocks because Client 1 still owns a shared lock.
When Client 1 attempts a Session.commit() method call
Client 1
Still shared
Cache entry (x)
Still shared
Client 2
Session.commit() method call
Lock manager displays LockDeadlockException to forceclient 1 transaction to roll back
Lock manager
Client 2 does not get a LockDeadlockException. Instead, it remains blocked waiting for lock promotion.
Client 1 asks lock manager to release the shared lock it owns.
When Client 1 releases the shared lock, Client 2 is granted an exclusive lock and is unblocked. Client 2 commit proceeds and is successfully committed.

*Figure 24. A deadlock situation*

As shown, when both clients are trying to promote to exclusive locks and still own the shared locks, it is impossible for either of them to actually get one. They always wait for the other client to release its shared lock, and thus a `LockDeadlockException` occurs.

## Possible solutions

Receiving this exception is positive on occasion. When there are many threads, all of which execute transactions on a particular map, it is possible that you will encounter the situation described previously (Figure1). This exception is thrown to keep your program from hanging. Catching this exception allows you to notify yourself and, if you want to, add code to the catch block so that you can get more details of the cause. Since you will only see this exception in a pessimistic locking strategy, one simple solution is to simply use an optimistic locking strategy. If pessimistic is required, however, you can use the getForUpdate method instead of the get method. This eliminates getting the exceptions for the situation described previously.

## XML configuration problem diagnosis

### Referencing a nonexistent plug-in collection

When using XML to define BackingMap plug-ins, the pluginCollectionRef attribute of the backingMap element must reference a backingMapPluginCollection. The pluginCollectionRef attribute must match the id of one of the backingMapPluginCollection elements.

### Message

If the pluginCollectionRef attribute does not match any id attributes of any of the backingMapPluginConfiguration elements, a message similar to the following message is displayed in the log.

```
[7/14/05 14:02:01:971 CDT] 686c060e XmlErrorHandl E CWOBJ9002E:
This is an English only Error message:
Invalid XML file. Line: 14; URI: null;
Message: Key 'pluginCollectionRef' with
value 'bookPlugins' not found for identity
constraint of element 'objectGridConfig'..
```

The following message is an excerpt from the log with trace enabled:

```
[7/14/05 14:02:01:971 CDT] 686c060e XmlErrorHandl E CWOBJ9002E: This is an
English only Error message:
Invalid XML file. Line: 14; URI: null; Message: Key
'pluginCollectionRef' with
value 'bookPlugins' not found for identity constraint
of element 'objectGridConfig'..
[7/14/05 14:02:01:991 CDT] 686c060e SystemErr R com.ibm.websphere.objectgrid.
ObjectGridException:
Invalid XML file: etc/test/document/bookstore.xml
[7/14/05 14:02:01:991 CDT] 686c060e SystemErr R at
com.ibm.ws.objectgrid.config.XmlConfigBuilder.<init>(XmlConfigBuilder.java:160)
[7/14/05 14:02:01:991 CDT] 686c060e SystemErr R at
com.ibm.websphere.objectgrid.ProcessConfigXML$2.run(ProcessConfigXML.java:99)
...
[7/14/05 14:02:02:001 CDT] 686c060e SystemErr R Caused by: org.xml.sax.
SAXParseException: Key 'pluginCollectionRef' with value 'bookPlugins'
not found for identity
constraint of element 'objectGridConfig'.
[7/14/05 14:02:02:001 CDT] 686c060e SystemErr R at org.apache.xerces.util.
ErrorHandlerWrapper.createSAXParseException(Unknown Source)
[7/14/05 14:02:02:001 CDT] 686c060e SystemErr R at org.apache.xerces.util.
ErrorHandlerWrapper.error(Unknown Source)
[7/14/05 14:02:02:001 CDT] 686c060e SystemErr R at org.apache.xerces.impl.
XMLErrorReporter.reportError(Unknown Source)
[7/14/05 14:02:02:001 CDT] 686c060e SystemErr R at org.apache.xerces.impl.
XMLErrorReporter.reportError(Unknown Source)
[7/14/05 14:02:02:011 CDT] 686c060e SystemErr R at org.apache.xerces.impl.xs.
XMLSchemaValidator$XSIErrorReporter.reportError(Unknown Source)
[7/14/05 14:02:02:011 CDT] 686c060e SystemErr R at org.apache.xerces.impl.xs.
XMLSchemaValidator.reportSchemaError(Unknown Source)
...
```

### Problem

The XML file that was used to produce this error is shown below. Notice that the book BackingMap has its pluginCollectionRef attribute set to bookPlugins, and the single backingMapPluginCollection has an id of collection1.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config
   ../objectGrid.xsd"  xmlns="http://ibm.com/ws/objectgrid/config">
   <objectGrids>
      <objectGrid name="bookstore">
         <backingMap name="book" pluginCollectionRef="bookPlugin" />
      </objectGrid>
   </objectGrids>
   <backingMapPluginCollections>
    <backingMapPluginCollection id="collection1">
     <bean id="Evictor"
       className="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor" />
     </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

### Solution

To fix the problem, ensure that the value of each pluginCollectionRef matches the id of one of the backingMapPluginCollection elements. In this example, simply changing the name of the pluginCollectionRef to collection1 avoids this error. Other ways to fix the problem include changing the id of the existing backingMapPluginCollection to match the pluginCollectionRef, or adding an additional backingMapPluginCollection with an id that matches the pluginCollectionRef.

## Missing a required attribute

Many of the elements in the XML file have several optional attributes. You can include or exclude optional attributes in the file. The XML will pass validation either way. However, there are some required attributes. If these required attributes are not present when their associated element is used, XML validation fails.

### Message

When a required attribute is missing, a message similar to the one that follows is found in the log. In this example, the type attribute is missing from the property element.

```
[7/15/05 13:41:41:267 CDT] 6873dcac XmlErrorHandl E CWOBJ9002E:
This is an English only
Error message: Invalid XML file.
Line: 12; URI: null; Message: cvc-complex-type.4:
Attribute 'type' must appear on element 'property'..
```

The following message is an excerpt from the log with trace enabled.

```
[7/15/05 14:08:48:506 CDT] 6873dff9 XmlErrorHandl E CWOBJ9002E: This is an English
only Error message: Invalid XML file.
Line: 12; URI: null; Message: cvc-complex-type.4: Attribute 'type'
must appear on element 'property'..
[7/15/05 14:08:48:526 CDT] 6873dff9 SystemErr R com.ibm.websphere.objectgrid.
ObjectGridException: Invalid XML file: etc/test/document/bookstore.xml
[7/15/05 14:08:48:536 CDT] 6873dff9 SystemErr R at com.ibm.ws.objectgrid.config.
XmlConfigBuilder.<init>(XmlConfigBuilder.java:160)
[7/15/05 14:08:48:536 CDT] 6873dff9 SystemErr R at com.ibm.websphere.objectgrid.
```

```
ProcessConfigXML$2.run(ProcessConfigXML.java:99)
...
[7/15/05 14:08:48:536 CDT] 6873dff9 SystemErr R Caused by: org.xml.sax.
SAXParseException: cvc-complex-type.4:
Attribute 'type' must appear on element 'property'.
[7/15/05 14:08:48:546 CDT] 6873dff9 SystemErr R at org.apache.xerces.util.
ErrorHandlerWrapper.createSAXParseException(Unknown Source)
[7/15/05 14:08:48:546 CDT] 6873dff9 SystemErr R at org.apache.xerces.util.
ErrorHandlerWrapper.error(Unknown Source)
[7/15/05 14:08:48:546 CDT] 6873dff9 SystemErr R at org.apache.xerces.impl.
XMLErrorReporter.reportError(Unknown Source)
[7/15/05 14:08:48:546 CDT] 6873dff9 SystemErr R at org.apache.xerces.impl.
XMLErrorReporter.reportError(Unknown Source)
[7/15/05 14:08:48:546 CDT] 6873dff9 SystemErr R at org.apache.xerces.impl.xs.
XMLSchemaValidator$XSIErrorReporter.reportError(Unknown Source)
[7/15/05 14:08:48:546 CDT] 6873dff9 SystemErr R at org.apache.xerces.impl.xs.
XMLSchemaValidator.reportSchemaError(Unknown Source)
...
```

### Problem

The following example is the XML file that was used to produce the previous error. Notice that the property on the Evictor has only two of the three required attributes. The name and value attributes are both present, but the type attribute is missing. This missing attribute causes XML validation to fail.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config">
   <objectGrids>
      <objectGrid name="bookstore">
         <backingMap name="book" pluginCollectionRef="collection1" />
      </objectGrid>
   </objectGrids>
   <backingMapPluginCollections>
    <backingMapPluginCollection id="collection1">
     <bean id="Evictor"
       className="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor" />
       <property name="maxSize" value="89" />
     </backingMapPluginCollection>
    </backingMapPluginCollections>
</objectGridConfig>
```

### Solution

To solve this problem, add the required attribute to the XML file. In the example XML file shown previously, you need to add the attribute type and assign the integer value.

## Missing a required element

A few of the XML elements are required by the schema. If they are not present, the XML fails validation.

### Message

When a required element is missing, a message similar to the one that follows is found in the log. In this case, the objectGrid element is missing.

```
[7/15/05 14:54:23:729 CDT] 6874d511 XmlErrorHandl E CWOBJ9002E:
This is an English only Error message: Invalid XML file.
Line: 5; URI: null; Message: cvc-complex-type.2.4.b: The content of
element 'objectGrids' is not complete.
One of '{"http://ibm.com/ws/objectgrid/config":objectGrid}' is expected..
```

Enable trace to see more information regarding this error. Section ObjectGridManager covers information on how to turn on the trace.

### Problem

The following example is the XML file that was used to produce this problem. Notice that the objectGrids element does not have objectGrid child elements. According to the XML schema, the objectGrid element must occur within the objectGrids tags at least once. This missing element causes XML validation to fail.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config">
   <objectGrids>
   </objectGrids>
</objectGridConfig>
```

### Solution

To fix this problem, make sure that the required elements are in the XML file. In the previous example, at least one objectGrid element must be placed within the objectGrids tag. Once the required elements are present, you can successfully validate the XML file.

The following valid XML file contains the required elements present.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config">
   <objectGrids>
      <objectGrid name="bookstore" />
   </objectGrids>
</objectGridConfig>
```

## XML value of attribute is not valid

### Message

Some of the attributes in the XML file can only be assigned certain values. These attributes have their acceptable values enumerated by the schema. These attributes include:

- `authorizationMechanism` • attribute on the objectGrid element
- `copyMode` attribute on the `backingMap` element
- `lockStrategy` attribute on the `backingMap` element
- `ttlEvictorType` attribute on the `backingMap` element
- `type` attribute on the `property` element

If one of these attributes is assigned an invalid value, XML validation fails.

When an attribute is set to a value that isn’t one of its enumerated values, the following message shows in the log:

```
[7/19/05 16:45:40:992 CDT] 6870e51b XmlErrorHandl E CWOBJ9002E: This is an English
only Error message: Invalid XML file. Line: 6; URI: null; Message:
cvc-enumeration-valid: Value 'INVALID_COPY_MODE' is not facet-valid with
respect to enumeration '[COPY_ON_READ_AND_COMMIT, COPY_ON_READ,
COPY_ON_WRITE, NO_COPY]'. It must be a value from the enumeration..
```

The following excerpt is from the log with trace enabled.

```
[7/19/05 16:45:40:992 CDT] 6870e51b XmlErrorHandl E CWOBJ9002E: This is an
  English only Error message: Invalid XML file. Line: 6; URI: null; Message:
     cvc-enumeration-valid: Value 'INVALID_COPY_MODE' is not facet-valid with
     respect to enumeration '[COPY_ON_READ_AND_COMMIT, COPY_ON_READ,
     COPY_ON_WRITE, NO_COPY]'. It must be a value from the enumeration..
[7/19/05 16:45:41:022 CDT] 6870e51b SystemErr R com.ibm.websphere.objectgrid
     .ObjectGridException: Invalid XML file: etc/test/document/backingMapAttrBad.xml
[7/19/05 16:45:41:022 CDT] 6870e51b SystemErr R at com.ibm.ws.objectgrid.config
     .XmlConfigBuilder.<init>(XmlConfigBuilder.java:160)
[7/19/05 16:45:41:022 CDT] 6870e51b SystemErr R at com.ibm.websphere.objectgrid
     .ProcessConfigXML$2.run(ProcessConfigXML.java:99)...
[7/19/05 16:45:41:032 CDT] 6870e51b SystemErr R Caused by:
  org.xml.sax.SAXParseException:
     cvc-enumeration-valid: Value 'INVALID_COPY_MODE' is not facet-valid
  with respect
     to enumeration '[COPY_ON_READ_AND_COMMIT, COPY_ON_READ, COPY_ON_WRITE,
  NO_COPY]'.
     It must be a value from the enumeration.
[7/19/05 16:45:41:032 CDT] 6870e51b SystemErr R at org.apache.xerces.util
     .ErrorHandlerWrapper.createSAXParseException(Unknown Source)
[7/19/05 16:45:41:032 CDT] 6870e51b SystemErr R at org.apache.xerces.util
     .ErrorHandlerWrapper.error(Unknown Source)
[7/19/05 16:45:41:032 CDT] 6870e51b SystemErr R at org.apache.xerces.impl
     .XMLErrorReporter.reportError(Unknown Source)
[7/19/05 16:45:41:032 CDT] 6870e51b SystemErr R at org.apache.xerces.impl
     .XMLErrorReporter.reportError(Unknown Source)
[7/19/05 16:45:41:032 CDT] 6870e51b SystemErr R at org.apache.xerces.impl
     .xs.XMLSchemaValidator$XSIErrorReporter.reportError(Unknown Source)
[7/19/05 16:45:41:032 CDT] 6870e51b SystemErr R at org.apache.xerces.impl
     .xs.XMLSchemaValidator.reportSchemaError(Unknown Source)
...
```

## Problem

An attribute that is assigned a value out of a specific set of values has been set improperly. In this case, the copyMode attribute is not set to one of its enumerated values. It was set to INVALID_COPY_MODE. The following is the XML file that was used to produce this error.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
   xmlns="http://ibm.com/ws/objectgrid/config">
   <objectGrids>
      <objectGrid name="bookstore" />
         <backingMap name="book" copyMode="INVALID_COPY_MODE"/>
            </objectGrid>
           </objectGrids>
</objectGridConfig>
```

## Solution

In this example, copyMode has an invalid value. Set the attribute to one of these valid values: COPY_ON_READ_AND_COMMIT, COPY_ON_READ, COPY_ON_WRITE, or NO_COPY.

## Validating XML without support of an implementation

The IBM Software Development Kit (SDK) version 1.4.2 contains an implementation of some JAXP function to use for XML validation against a schema.

### Message

When using a SDK that does not contain this implementation, attempts to validate may fail. If you would like to validate XML using a SDK that does not contain this implementation, download Xerces, and include its Java archive (JAR) files in the classpath.

When you attempt to validate XML with a SDK that does not have the necessary implementation, the following error is found in the log.

```
[7/19/05 10:50:45:066 CDT] 15c7850 XmlConfigBuil d XML validation is enabled
[7/19/05 10:50:45:086 CDT] 15c7850 SystemErr R com.ibm.websphere
     .objectgrid[7/19/05 10:50:45:086 CDT] 15c7850 SystemErr R at
   com.ibm.ws.objectgrid
     .ObjectGridManagerImpl.getObjectGridConfigurations
   (ObjectGridManagerImpl.java:182)
[7/19/05 10:50:45:086 CDT] 15c7850 SystemErr R at com.ibm.ws.objectgrid
     .ObjectGridManagerImpl.createObjectGrid(ObjectGridManagerImpl.java:309)
[7/19/05 10:50:45:086 CDT] 15c7850 SystemErr R at com.ibm.ws.objectgrid.test.
     config.DocTest.main(DocTest.java:128)
[7/19/05 10:50:45:086 CDT] 15c7850 SystemErr R Caused by: java.lang
     .IllegalArgumentException: No attributes are implemented
[7/19/05 10:50:45:086 CDT] 15c7850 SystemErr R at org.apache.crimson.jaxp.
     DocumentBuilderFactoryImpl.setAttribute(DocumentBuilderFactoryImpl.java:93)
[7/19/05 10:50:45:086 CDT] 15c7850 SystemErr R at com.ibm.ws.objectgrid.config
     .XmlConfigBuilder.<init>(XmlConfigBuilder.java:133)
[7/19/05 10:50:45:086 CDT] 15c7850 SystemErr R at com.ibm.websphere.objectgrid
     .ProcessConfigXML$2.run(ProcessConfigXML.java:99)
...
```

### Problem

The SDK that is used does not contain an implementation of JAXP function that is necessary to validate XML files against a schema.

### Solution

After you download Apache Xerces and include the JARs in the classpath, you can validate the XML file successfully.

## ObjectGrid messages

This reference information provides additional information about messages you might encounter while using the ObjectGrid. Messages are identified by the message key and have explanation and user response. They could be information, warning, or errors and are indicated by the last letter (I, W, or E) of the message key. The explanation part of the message explains why the message occurs. The user response part of the message describes what action should be taken in the case of warning or error message.

**CWOBJ0001E:** Method, {0}, was called after initialization completed.
**Explanation:** After initialization completes, certain method invocations are no longer accepted.
**User response:** Restructure your code so that the configuration completes before use of the runtime is initiated.

**ICWOBJ0002W**: ObjectGrid component is ignoring an unexpected exception: {0}.
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ0005W**: The thread created an InterruptedException: {0}
**Explanation:** An InterruptedException occurred.
**User response:** Check the exception message to see whether this interruption is expected.

**CWOBJ0006W:** An exception occurred: {0}
**Explanation:** An exception occurred during the runtime.
**User response:** Check the exception message to see whether this is an expected exception.

**CWOBJ0007W:** The value null was specified for {0}, a default value of {1} is used.
**Explanation:** A null value was specified for the variable. A default value is used.
**User response:** Set the appropriate value. Refer to ObjectGrid documentation to know the valid values for the variables or properties.

**CWOBJ0008E**: The value {0} provided for the property {1} is invalid.
**Explanation:** An invalid value was specified for the variable.
**User response:** Set the appropriate value. Refer to ObjectGrid document to know the valid values for the variables or properties.

**CWOBJ0010E**: Message key {0} is missing.
**Explanation:** A message key is missing in the message resource bundle
**User response:** CMSG0002

**CWOBJ0011W**: Cannot deserialize field {0} in class {1}; using the default value.
**Explanation:** During the deserialization of an object, an expected field was not found. This field was probably not found because the object was deserialized by a different version of the class than the one that serialized it.
**User response:** This warning indicates a potential problem. No user action is required unless further errors arise.

**CWOBJ0012E:** The LogElement type code, {0} ({1}), is not recognized for this operation.
**Explanation:** An internal error occurred in the ObjectGrid runtime.
**User response:** CMSG0002

**CWOBJ0013E**: An exception occurred while attempting to evict entries from the cache: {0}
**Explanation:** A problem occurred while attempting to apply the eviction entries to the cache.
**User response:** Check the exception message to see whether this is an expected exception.

**CWOBJ0014E:** The ObjectGrid runtime detected an attempt to nest transactions.
**Explanation:** The nesting of transactions is not permitted.
**User response:** Modify the code to avoid the nesting of transactions.

**CWOBJ0015E:** An exception occurred while attempting to process a transaction: {0}
**Explanation:** A problem occurred during transaction processing.
**User response:** Check the exception message to see whether this exception is expected.

**CWOBJ0016E:** No active transaction is detected for the current operation.
**Explanation:** An active transaction is necessary to perform this operation.
**User response:** Modify the code to start a transaction before performing this operation.

**CWOBJ0017E:** A duplicate key exception was detected during the processing of the ObjectMap operation: {0}
**Explanation:** The key for the entry already exists in the cache.
**User response:** Modify the code to avoid inserting the same key more than once.

**CWOBJ0018E:** The key was not found during the processing of the ObjectMap operation: {0}
**Explanation:** The key for the entry does not exist in the cache.
**User response:** Modify the code to ensure that the entry exists before attempting the operation.

**CWOBJ0019W:** Did not find data in the cache entry slot reserved for {0} to use for ObjectMap name {1}.
**Explanation:** An internal error occurred in the ObjectGrid runtime.
**User response:** CMSG0002

**CWOBJ0020E:** Cache entry is not in BackingMap {0}.
**Explanation:** Internal error in ObjectGrid runtime.
**User response:** CMSG0002

**CWOBJ0021E:** A usable ObjectTransformer instance was not found during the deserialization of the LogSequence object for {0} ObjectGrid and {1} ObjectMap.
**Explanation:** The receiving side of a LogSequence object does not have the proper configuration to support the required ObjectTransformer instance.
**User response:** Verify the configuration of the ObjectGrid instances for both the sending and receiving sides of the LogSequence object.

**CWOBJ0022E:** The caller does not own mutex: {0}.
**Explanation:** An internal error occurred in the ObjectGrid runtime.
**User response:** CMSG0002

**CWOBJ0023E:** The CopyMode ({0}) is not recognized for this operation.
**Explanation:** An internal error occurred in the ObjectGrid runtime.
**User response:** CMSG0002

**CWOBJ0024E:** Cannot deserialize field {0} in class {1}. Deserialization failed.
**Explanation:** During deserialization of an object, a required field was not found. This problem is likely an ObjectGrid runtime error.
**User response:** CMSG0002

**CWOBJ0025E:** The serialization of the LogSequence object failed. The number of serialized LogElement objects ({0}) does not match the number of read LogElement objects ({1}).
**Explanation:** An internal error occurred in the ObjectGrid runtime.
**User response:** CMSG0002

**CWOBJ0026E:** The JMX credential type is not right. It should be of type {0}.
**Explanation:** The JMX credential type is not right. If basic authentication is used, the expected type is String[] with the first element being user name and the second being password. If the client certificate is used, the expected type is Certificate[].
**User response:** Use the right credentials.

**CWOBJ0027E:** Internal runtime error. Clone method not supported: {0}
**Explanation:** An internal error occurred in the ObjectGrid runtime.
CLONE_METHOD_NOT_SUPPORTED_CWOBJ0027.useraction=CMSG0002

**CWOBJ0028E:** An error occurred in {0} for the map {1}. The key, {2}, was not found in the map. LogElement type is {3}.
**Explanation:** An internal error occurred when trying to evict an entry.
**User response:** CMSG0002

**CWOBJ0029E:** An error occurred in {0} for the map {1}. CacheEntry is missing a {2} object for key {3}. LogElement type is {4}.
**Explanation:** An internal error occurred when trying to evict an entry.
**User response:** CMSG0002

**CWOBJ0900I**: The ObjectGrid runtime component is started for server {0}.
**Explanation:** The ObjectGrid component is started.
**User response:** None. Informational entry.

**CWOBJ0901E**: "{0}" system property is required to start ObjectGrid component for server {1}.
**Explanation:** ObjectGrid runtime component requires "{0}" to be specified as a Java Virtual Machine system property.
**User response:** See Information Center for using WebSphere Administrator Console for providing ObjectGrid required custom properties.

**CWOBJ0902W**: Error prevented the ObjectGrid runtime component from starting for server {0}.
**Explanation:** A prior error prevented the ObjectGrid component from starting.
**User response:** See prior error messages to determine what prevented ObjectGrid component from starting.

**CWOBJ0910I**: The ObjectGrid runtime component is stopped for server {0}.
**Explanation:** The ObjectGrid component is stopped.
**User response:** None. Informational entry.

**CWOBJ0911I**: Starting the ObjectGrid runtime component for server {0}.
**Explanation:** The ObjectGrid component is starting.
**User response:** None. Informational entry.

**CWOBJ1001I**: ObjectGrid Server {0} is ready to process requests.
**Explanation:** ObjectGrid Server is ready to process requests.
**User response:** The services for this ObjectGrid Server are available.

**CWOBJ1002E**: Server port {0} is already in use.
**Explanation:** ObjectGrid server cannot be started due to port conflict.
**User response:** Users need to choose another port.

**CWOBJ1003I**: DCS Adapter service is disabled by configuration, to enable it, please change your configuration with an endpoint defined.
**Explanation:** DCS adapter is turned off.
**User response:** Users can turn on DCS adapter by changing the configuration.

**CWOBJ1004E**: Server topic is null
**Explanation:** Server topic is null
**User response:** CMSG0002

**CWOBJ1005E**: The incoming request queue is null.
**Explanation:** Client request handler cannot retrieve requests.
**User response:** CMSG0002

**CWOBJ1006E**: The outgoing result queue is null.
**Explanation:** Client request handler cannot give result to client.
**User response:** CMSG0002

**CWOBJ1007E**: ObjectGrid client request is null.
**Explanation:** Client request handler cannot handle request that does not contain any information about the request.
**User response:** Check your request

**CWOBJ1008E**: ObjectGrid client request TxID is null.
**Explanation:** We use TXID to match connections and do pooling, TXID cannot be null.
**User response:** CMSG0002

**CWOBJ1009E**: ObjectGrid client received a null response from the server.
**Explanation:** Encountered a null response from server.
**User response:** CMSG0002

**CWOBJ1010I**: Shutdown request is processing.
**Explanation:** Cluster servers are processing the shutdown request.
**User response:** none

**CWOBJ1011I**: Shutdown request is sending.
**Explanation:** Cluster servers are processing the shutdown request
**User response:** none

**CWOBJ1012I**: Shutdown request is performed.
**Explanation:** Cluster servers are processing the shutdown request.
**User response:** none

**CWOBJ1110I**: Starting the transport for ObjectGrid cluster {0} using IP Address {1}, port {2}, transport type {3}.
**Explanation:** The ObjectGrid cluster member transport is starting.
**User response:** None. Informational entry.

**CWOBJ1111W**: Resolution of IP Addresses for host name {0} found only the loopback address. The loopback address will be used.
**Explanation:** There may be a problem with the host name or DNS resolution. For production related implementation, a non-loopback address is normally expected.
**User response:** Modify the host name or determine if a DNS problem exits.

**CWOBJ1112E**: An error was encountered while looking up the IP address for the host name of an ObjectGrid cluster member. The host name is {0} and the server name is {1}. The member will be excluded from the cluster.
**Explanation:** Unable to resolve the IP address for the indicated host. The ObjectGrid cluster member for the specified host will be excluded.
**User response:** Correct the host name lookup problem and retry.

**CWOBJ1113E**: ObjectGrid cluster transport service on this process is not started. This cluster member is not defined in the configuration.
**Explanation:** This ObjectGrid cluster member is not a configured member of the cluster. If this cluster member should be a member of a ObjectGrid cluster, repair the configuration.
**User response:** Review the current configuration.

**CWOBJ1114E**: ObjectGrid cluster transport service on this process could not process the incoming message. The message is {0} and the exception is {1}.
**Explanation:** An unexpected internal error has been detected.
**User response:** Review the IBM ObjectGrid internet support web site for a similar problem or contact IBM service.

**CWOBJ1115E**: An unrecognized view change event was received from the ObjectGrid cluster transport. The view identifier is {0} and the event is {1}.
**Explanation:** The type of the event is not recognized. The HA Manager does not know how to respond to the event.
**User response:** Review the IBM ObjectGrid internet support web site for a similar problem or contact IBM service.

**CWOBJ1116E**: An attempt by another process to connect to this process via the ObjectGrid cluster transport has been rejected. The connecting process provided a name of {0}, a target of {1}, a member name of {2} and an IP address of {3}. The error message is {4}.
**Explanation:** The ObjectGrid cluster transport has rejected the connection attempt.
**User response:** This may be a connection attempt from an unauthorized party.

**CWOBJ1117E**: An attempt to authenticate a connection has failed. The exception is {0}.
**Explanation:** The ObjectGrid cluster transport has rejected the connection attempt.
**User response:** This may be a connection attempt from an unauthorized party.

**CWOBJ1118I**: ObjectGrid Server Initializing [Cluster: {0} Server: {1}].
**Explanation:** The ObjectGrid cluster member is initializing.
**User response:** None. Informational entry.

**CWOBJ1119I**: ObjectGrid client failed to connect to host: {0} port: {1}.
**Explanation:** ObjectGrid client failed to connect.
**User response:** None. Informational entry.

**CWOBJ1120I**: ObjectGrid Client connected successfully to host: {0} port: {1}.
**Explanation:** ObjectGrid Client connected successfully.
**User response:** None. Informational entry.

**CWOBJ1201E**: No valid client access end points are defined.
**Explanation:** =No valid client access end points are defined.
**User response:** Define a valid client access end point.

**CWOBJ1202E**: SSL Server Socket failed to initialize. The exception message is {0}
**Explanation:** SSL Server Socket fails to initialize. The SSL settings might be wrong or the port number is already in use.
**User response:** Examine the exception to see what went wrong.

**CWOBJ1203W**: Received a timeout event from the server for transaction: {0}
**Explanation:** Client did not receive expected response message from the server within a configured timeout limit.
**User response:** Look for prior messages that may explain the timeout. If none found, try increasing the timeout limit.

**CWOBJ1204W**: Received a message of unknown message type. The message is: {0}
**Explanation:** An unexpected internal error has been detected.
**User response:** Review the IBM ObjectGrid internet support web site for a similar problem or contact IBM service.

**CWOBJ1205E**: SSL Initialization failed. The exception message is {0}
**Explanation:** SSL Initialization failed. The SSL settings might be wrong.
**User response:** Examine the exception to see what went wrong.

**CWOBJ1206W**: SSL Initialization failed. The exception message is {0}
**Explanation:** SSL Initialization failed. The SSL settings might be wrong.
**User response:** Examine the exception to see what went wrong.

**CWOBJ1207W**: The property {0} on plug-in {1} is using an unsupported property type.
**Explanation:** Java primitives and their java.lang counterparts are the only supported property types. java.lang.String is also supported.
**User response:** Check the property type and change it to one of the supported types.

**CWOBJ1208W**: The specified plug-in type, {0}, is not one of the supported plug-in types.
**Explanation:** This type of plug-in is unsupported.
**User response:** Add one of the supported plug-in types.

**CWOBJ1211E**: The Performance Monitoring Infrastructure (PMI) creation of {0} failed. The exception is {1}.
**Explanation:** An attempt to create ObjectGrid PMI failed.
**User response:** Examine the exception message and the first failure data capture (FFDC) log.

The following messages are used to gather the user id and password on the panel or standard input.
```
LOGIN_PANEL_TITLE=Login at the target server
GENERIC_LOGIN_PROMPT=Enter login information
USER_ID=User identity
PASSWORD=Password
OK=OK
CANCEL=Cancel
```

**CWOBJ1215I**: ObjectGrid Transaction Propagation Event Listener is initializing [ObjectGrid {0}].
**Explanation:** This informational message indicates that the ObjectGrid Transaction Propagation Event Listener is initializing.
**User response:** None. Informational entry.

**CWOBJ1216I**: ObjectGrid Transaction Propagation Event Listener is initialized [ObjectGrid {0}].
**Explanation:** ObjectGrid Transaction Propagation Event Listener Initialized.
**User response:** =None. Informational entry.

**CWOBJ1217I**: ObjectGrid Transaction Propagation Service Point Initialized [ObjectGrid {0}].
**Explanation:** This informational message indicates that the ObjectGrid Transaction Propagation Event Listener is initialized.
**User response:** None. Informational entry.

**CWOBJ1218E:** ObjectGrid Transaction Propagation Event Listener failure occurred [ObjectGrid {0} Exception message {1}].
**Explanation:** ObjectGrid runtime encountered an ObjectGrid Transaction Propagation failure.
**User response:** Examine the exception to determine the failure.

**CWOBJ1219E:** ObjectGrid Transaction Propagation Service End Point failure occurred [ObjectGrid {0} Exception message {1}].
**Explanation:** ObjectGrid runtime encountered an ObjectGrid Transaction Propagation Service End Point failure.
**User response:** Examine the exception to determine the failure.

**CWOBJ1220E:** ObjectGrid Transaction Propagation Service is not supported in this environment.
**Explanation:** =ObjectGrid Transaction Propagation Service is not supported on z/OS or the standalone ObjectGrid server environment.
**User response:** Do not use ObjectGrid Transaction Propagation Service on z/OS or in the standalone ObjectGrid server environment

**CWOBJ1300I:** Adapter successfully initialized ObjectGrid.
**Explanation:** Adapter successfully initialized ObjectGrid.
**User response:** None. Informational entry.

**CWOBJ1301E:** Adapter failed to initialize ObjectGrid. Exception occurred [Exception message {0}].
**Explanation:** Adapters attempt to initialize ObjectGrid failed.
**User response:** Examine the exception to determine the failure.

**CWOBJ1302I:** Adapter stopped.
**Explanation:** Adapter stopped.
**User response:** None. Informational Only.

**CWOBJ1303I:** Adapter started.
**Explanation:** PMA_CWOBJ1303.explanation=Adapter started.
**User response:** None. Informational Only.

**CWOBJ1304I:** ObjectGrid security is enabled.
**Explanation:** ObjectGrid security is enabled.
**User response:** none

**CWOBJ1305I:** ObjectGrid security is disabled.
**Explanation:** ObjectGrid security is disabled.
**User response:** none

**CWOBJ1306W:** Cannot retrieve the client certificates from the SSL socket.
**Explanation:** For some reason, the runtime cannot retrieve the client certificates from the SSL socket.
**User response:** Check your SSL configurations.

**CWOBJ1307I:** Security of the ObjectGrid instance {0} is enabled.
**Explanation:** Security of the ObjectGrid instance {0} is enabled.
**User response:** none

**CWOBJ1308I:** Security of the ObjectGrid instance {0} is disabled.
**Explanation:** Security of the ObjectGrid instance {0} is disabled.
**User response:** none

**CWOBJ1309E:** Unexpected error occured in the connect token creation: {0}.
**Explanation:** An unexpected error occurs in the connection token creation.
**User response:** Check the security configuration

**CWOBJ1310E:** An attempt by another process to connect to this process via the core group transport has been rejected. The connecting process provided a source core group name of {0}, a target of {1}, a member name of {2} and an IP address of {3}. The error message is {4}.
**Explanation:** The High Availability Manager has rejected a connection attempt.
**User response:** This may be a connection attempt from an unauthorized party.

**CWOBJ1400W**: Detected multiple ObjectGrid runtime JARS files in the JVM. Using multiple ObjectGrid runtime JAR files may cause problems.
**Explanation:** Usually only one ObjectGrid runtime JAR should be found in a JVM.
**User response:** Use the appropriate ObjectGrid runtime JAR for your configuration.

**CWOBJ1401E**: Detected a wrong ObjectGrid runtime JAR file for this configuration. Detected configuration is {0}. Expected Jar file is {1}.
**Explanation:** Each ObjectGrid runtime JAR file corresponds to a particular supported configuration.
**User response:** Use the appropriate ObjectGrid runtime JAR for your configuration.

**CWOBJ1402E**: ObjectGrid connection link callback not found for id: {0}.
**Explanation:** Internal error in ObjectGrid runtime.
**User response:** CMSG0002

**CWOBJ1500E**: An exception occurred when attempting to create a GroupName for HA Group ({0}): {1}.
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1501E**: An exception occurred when member ({0}) attempted to join HA Group ({1}): {2}.
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1503E**: Cannot access ObjectGrid ({0}) for applying updates to replica member ({1}).
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1504E**: An exception occurred when attempting to process the LogSequences for replica ({0}): {1}.
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1505E**: More than one replication group member reported back as the primary. Only one primary can be active. ({0}).
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1506E**: More than one primary replication group member exists in this group ({1}). Only one primary can be active. ({0}).
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1507W**: An exception occurred when attempting to end the replication process for BackingMap ({0}): {1}.
**Explanation:** While attempting to shut down a primary replication group member, an exception occurred during the clean up processing.
**User response:** CMSG0002

**CWOBJ1508E**: An exception occurred when attempting to send message ({0}) from sender ({1}) to receiver ({2}): {3}.
**Explanation:** A problem occurred while attempting to send a message between replication group members.
**User response:** CMSG0002

**CWOBJ1509E**: An exception occurred when attempting to serialize message ({0}): {1}.
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1510E**: An exception occurred when attempting to inflate message ({0}): {1}.
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1511I**: {0} ({1}) is open for business.
**Explanation:** Specified replication group member is now ready to accept requests.
**User response:** None.

**CWOBJ1512W**: {0} already exists in replication group {1}.
**Explanation:** The specified replication group member is already active in this replication group.
**User response:** None.

**CWOBJ1513E**: Synchronous replication failed on {0} ({1}). This member is no longer active.
**Explanation:** A problem was encountered that prevented synchronous replication from successfully completing.
**User response:** Review previous messages in the log to help diagnose the problem. Stopping and restarting the specified server may be required.

**CWOBJ1514I**: Primary ({0}) is being downgraded to either a replica or standby.
**Explanation:** This is not a normal operation, but ObjectGrid processing can continue.
**User response:** CMSG0002

**CWOBJ1515I**: Minimum configuration requirements not satisfied for replication group ({0}).
**Explanation:** The necessary primary and replica configuration requirements were not met with the recent replication group member change.
**User response:** Wait for additional resources to be started and recognized for this configuration.

**CWOBJ1516E**: An exception occurred when attempting to activate the replication process for ObjectGrid ({0}): {1}.
**Explanation:** While attempting to start a primary replication group member, an exception occurred during the activation processing.
**User response:** CMSG0002

**CWOBJ1517E**: Synchronous replication failed for transaction {2} on {0} ({1}). This member is no longer active.
**Explanation:** A problem was encountered that prevented synchronous replication from successfully completing.
**User response:** Review previous messages in the log to help diagnose the problem. Stopping and restarting the specified server may be required.

**CWOBJ1518E**: An exception occurred when attempting to commit replica transaction ({0}) for primary transaction ({1}) on Replica ({2}): {3}.
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1519E**: An exception occurred when attempting to rollback the LogSequences for replica ({0}): {1}.
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ1610W**: Try to reset a null cluster for {0}.
**Explanation:** Replication group cluster data are not available.
**User response:** none

**CWOBJ1611I**: Replication group cluster {0} is open for business.
**Explanation:** Now replication group cluster can accept requests.
**User response:** none

**CWOBJ1612I**: Replication group cluster {0} is closed for business.
**Explanation:** Now replication group cluster cannot accept requests.
**User response:** =none

**CWOBJ1620I**: Replacing target for wrongly routed request due to changes in the server. The new target is {0}.
**Explanation:** Old routing target replaced with new target.
**User response:** If the intended replication group is out of service, you need to bring it back.

**CWOBJ1630I**: Replication group cannot serve this request {0}.
**Explanation:** Routing is refused due to the unavailable service such as the Domino effect
**User response:** Information only.

**CWOBJ1632E**: Original request does not have a valid ID; no way to forward this request.
**Explanation:** No way to forward this request because the original request does not have a valid ID.
**User response:** Report to IBM support

**CWOBJ1634I**: Router cannot find the forwarding target; using blind forwarding.
**Explanation:** Router cannot find the forwarding target.
**User response:** None

**CWOBJ1660I**: Replication group member has changed. This server does not host what is requested anymore. The original request is {0}.
**Explanation:** Replication group member has changed.
**User response:** If the intended replication group is out of service, you need to bring it back.

**CWOBJ1661I**: Cluster data are updated for replication group: {0}
**Explanation:** Cluster data are updated
**User response:** none

**CWOBJ1663E**: Server router cannot verify server routing for {0}, because cluster data for this replication group are null in the server.
**Explanation:** No replication group cluster data are available to verify.
**User response:** Report to IBM support

**CWOBJ1668W**: Request is coming to the server that has not completely started.
N**Explanation:** Server startup takes 60-120 seconds. Request will be automatically retried if you have configured so (by default the request will be automatically retried).
**User response:** Adjust your configuration or start your clients 60-120 seconds after you start your servers.

**CWOBJ1680W**: The configured TCP connection timeout is smaller than retryInterval * max(startupRetries, maxRetries), so there is possibility that connection will time out.
**Explanation:** The configured TCP connection timeout should be larger than retryInterval * max(startupRetries, maxRetries).
**User response:** Adjust your configuration.

**CWOBJ1682W**: The configured transaction timeout is smaller than maxForwards * retryInterval * max(startupRetries, maxRetries), so there is possibility that transaction will time out.
**Explanation:** The configured transaction timeout should be larger than maxForwards * retryInterval * max(startupRetries, maxRetries).
**User response:** Adjust your configuration.

**CWOBJ1700I**: Standalone HAManager is initialized with coregroup {0}.
**Explanation:** standalone HAManager is initialized successfully.
**User response:** none

**CWOBJ1701I**: Standalone HAManager is already initialized.
**Explanation:** Standalone HAManager is already initialized successfully.
**User response:** none

**CWOBJ1702E**: Standalone HAManager is not initialized, so it cannot be started.
**Explanation:** Standalone HAManager is not initialized.
**User response:** Initialize it before starting it.

**CWOBJ1710I**: Standalone HAManager is started successfully.
**Explanation:** Standalone HAManager is started successfully.
**User response:** none

**CWOBJ1711I**: Standalone HAManager is already started successfully.
**Explanation:** Standalone HAManager is already started successfully.
**User response:** none

**CWOBJ1712E**: Standalone HAManager is not started.
**Explanation:** Standalone HAManager is not started.
**User response:** Initialize and start it before using it.

**CWOBJ1713E**: Standalone HAManager failed to start.
**Explanation:** Standalone HAManager failed to start.
**User response:** Check if ports are used already.

**CWOBJ1720I**: HAManager Controller detected that ObjectGrid server is in the WebSphere environment, using WebSphere HAManager instead of initializing and starting standalone HAManager.
**Explanation:** ObjectGrid server is running in the WebSphere environment.
**User response:** None

**CWOBJ1730I**: HAManager Controller detected that the WebSphere external HAManager is null.
**Explanation:** Cannot get the external HAManager from WebSphere.
**User response:** None

**CWOBJ1790I**: Need to initialize and start the standalone HAManager.
**Explanation:** Cannot get the external HAManager from WebSphere. Need to initialize and start the standalone HAManager.
**User response:** None

**CWOBJ1792I**: The maximum number of threads is {0} and the minimum number of threads is {1}.
**Explanation:** Configure thread pool.
**User response:** Information only.

**CWOBJ1800I**: Forwarding is required for request {0} with response of {1}.
**Explanation:** Forward routing is required.
**User response:** None. Handled automatically

**CWOBJ1810I**: Forwarding is required for response {0}.
**Explanation:** Forwarding is required for response.
**User response:** None

**CWOBJ1811E**: Forwarding is required, but the original request cannot be found.
**Explanation:** Forwarding is required, but the original request cannot be found.
**User response:** None

**CWOBJ1820E**: Forwarding request does not have a replication group identifier.
**Explanation:** There is not any replication group identifier in this forwarding request.
**User response:** Contact IBM Support

**CWOBJ1870I**: Server service is not available for response {0}.
**Explanation:** Server service is not available due to the Domino effect or other events.
**User response:** Bring at least the minimum number of servers up.

**CWOBJ1871E**: Detected unavailable service, received null response, no way to retry.
**Explanation:** Null response from the unavailable service
**User response:** Contact IBM support

**CWOBJ1872I**: Service is unavailable with response of {0}.
**Explanation:** Service is not available
**User response:** Bring at least the minimum number of servers up or check if server startup is successful.

**CWOBJ1890I**: Re-routing request {0} due to
an un-responsive server.
**Explanation:** The request for intended server failed
to complete. Request was re-routed to another server.
**User response:** None. Handled automatically. If the intended
replication group is out of service, you need to bring it back.

**CWOBJ1891E**: All servers are not available in replication group {0}.
**Explanation:** All servers were either not started or have
failed. They are not available
**User response:** If the intended replication group is out of service,
you need to bring it back.

**CWOBJ1898W**: Forwarding is required, but router cannot find new
available target for response {0}
**Explanation:** Service is not available.
**User response:** Make service available.

**CWOBJ1899W**: Forwarding is required, but router cannot find right
replication group for response {0}
**Explanation:** Replication group ID is lost.
**User response:** Contact IBM Support

**CWOBJ1900I**: Client server remote procedure call service is initialized.
**Explanation:** Client server remote procedure call service is initialized.
**User response:** None

**CWOBJ1901I**: Client server remote procedure call service is started.
**Explanation:** Client server remote procedure call service is started.
**User response:** None

**CWOBJ1902I**: Client server remote procedure call handler
threads are started.
**Explanation:** Client server remote procedure call handler
threads are started.
**User response:** None

**CWOBJ1903I**: Configuration network service is initialized.
**Explanation:** Configuration network service is initialized.
**User response:** None

**CWOBJ1904I**: Configuration network service is started.
**Explanation:** Configuration network service is started.
**User response:** None

**CWOBJ1905I**: Configuration handler is started.
**Explanation:** Configuration handler is started.
**User response:** None

**CWOBJ1913I**: System administration network service is initialized.
**Explanation:** System administration network service is initialized.
**User response:** None

**CWOBJ1914I**: System administration network service is started.
**Explanation:** System administration network service is started.
**User response:** None

**CWOBJ1915I**: System administration handler is started.
**Explanation:** System administration handler is started.
**User response:** None

**CWOBJ2000E**: No member in this replication group {0}.
**Explanation:** No member can be found in this replication group.
**User response:** Check if servers are started or data are available

**CWOBJ2001W**: No available member in this replication group {0}.
**Explanation:** No available member can be found in this replication group.
**User response:** Check if server service is available

**CWOBJ2002W**: No available routing table for this replication group {0}.
**Explanation:** No available routing table for this replication group.
**User response:** Check if clients have brought in routing table

**CWOBJ2003I**: Cannot find routing cache for cache key {0}, creating
new routing cache.
**Explanation:** First time routing or cluster changes.
R**User response:** none

**CWOBJ2010E**: Target for this request is null.
**Explanation:** Request did not come with target information.
**User response:** contact IBM support.

**WOBJ2060I**: Client received new version of replication group cluster {0}.
**Explanation:** Client received new version of replication group cluster
**User response:** none

**CWOBJ2068I**: Reachability control detected problem in replication group member {0}.
**Explanation:** Some server cannot be reached, reachability mechanism will handle it.
**User response:** None.

**CWOBJ2069I**: Reachability control timer releases replication group member {0}.
**Explanation:** This member is available for routing.
**User response:** none

**CWOBJ2086I**: Routing thread control is activated due to overload for replication group {0}.
**Explanation:** Thread control is in action.
**User response:** none

**CWOBJ2088I**: Reachability control is activated to regulate the server availability for replication group {0}.
**Explanation:** Reachability is in action.
**User response:** none

**CWOBJ2090W**: Cannot find routing table for replication group {0}.
**Explanation:** Replication group cluster is null.
**User response:** none

**CWOBJ2091W**: Routing table is not null, but it does not contain any servers for replication group {0}.
**Explanation:** Replication group cluster is empty.
**User response:** none

**CWOBJ2092I**: Routing table is null in runtime for replication group {0}.
**Explanation:** Getting routing table from runtime.
**User response:** none

**CWOBJ2093I**: Routing table is not null in replication group cluster store for replication group {0}
**Explanation:** Getting routing table from cluster store.
**User response:** none

**CWOBJ2096I**: Routing table was obtained from replication group cluster store for replication group {0}.
**Explanation:** Obtained replication group cluster from replication group cluster store.
**User response:** none

**CWOBJ2097I**: Routing is based on round robin algorithm for replication group {0}.
**Explanation:** Routing is based on round robin algorithm.
**User response:** none

**CWOBJ2098I**: Routing is based on random selection for replication group {0}.
**Explanation:** Routing is based on random selection.
**User response:** none

**CWOBJ2100I**: Connection ({0}) is stale, it cannot be reused.
**Explanation:** Connection is stale.
**User response:** none

**CWOBJ2101W**: Connection cannot be acquired after the maximum wait time.
**Explanation:** There are not any connections left in the pool.
**User response:** Increase the maximum number of connections in the configuration.

**CWOBJ1600I**: ManagementGateway service started on port ({0}).
**Explanation:** ManagementGateway service is ready to process requests.
**User response:** ManagementGateway service is available.

**CWOBJ1601E**: ManagementGateway service failed to start on port ({0}).
**Explanation:** ManagementGateway service failed to start.
**User response:** Ensure specified port is not already in use.

**CWOBJ1602E**: ManagementGateway service failed to connect to server at ({0}):({1}).
**Explanation:** ManagementGateway service failed to connect to server.
**User response:** Ensure server is running.

**CWOBJ1603E**: Management service failed to respond to ({0}) remote request.
**Explanation:** CMSG0001
**User response:** CMSG0002

**CWOBJ2400E**: Invalid Configuration: backing map {0} is a member of more than one map-set.
**Explanation:** A backingMap can belong to only one map-set.
**User response:** Edit the cluster XML file so that each backing map belongs to only one map-set.

**CWOBJ2401E**: Invalid Configuration: backing map {0} in distributed ObjectGrid {1} is not in a map-set.
**Explanation:** Each backing map of a distributed ObjectGrid must be placed in a map-set.
**User response:** Edit the cluster XML file so that each backing map in a distributed ObjectGrid belongs to a map-set.

**CWOBJ2402E**: Invalid Configuration: map-set has a reference to a {0} map. This backing map does not exist in the ObjectGrid XML file.
**Explanation:** Each map within a map-set must reference a backing map from the ObjectGrid XML file.
**User response:** Edit the XML file(s) so that each map within the map-set references a backing map from the ObjectGrid XML file.

**CWOBJ2403E**: The XML file is invalid. A problem has been detected with {0} at line {1}. The error message is {2}.
**Explanation:** The XML file does not conform to the schema.
**User response:** Edit the XML file so that it is conforms to the schema.

**CWOBJ2404W**: The value specified for {0} is {1}. This is an invalid value. {0} will not be set.
**Explanation:** The value for this configuration attribute is not valid.
**User response:** Set the configuration attribute to a proper value in the XML file.

**CWOBJ2405E**: The objectgrid-binding ref {0} in the Cluster XML file does not reference a valid ObjectGrid from the ObjectGrid XML file.
**Explanation:** Each of the objectgrid-bindings must reference an ObjectGrid from the ObjectGrid XML file.
**User response:** Edit the XML files so that the objectgrid-binding in the Cluster XML references a valid ObjectGrid in the ObjectGrid XML.

**CWOBJ2500E**: Failed to start ObjectGrid server {0}.
**Explanation:** The ObjectGrid server failed to start properly.
**User response:** Check the log for exceptions.

**CWOBJ2501I**: Launching ObjectGrid server {0}.
**Explanation:** An ObjectGrid server is starting up.
**User response:** none

**CWOBJ2502I**: Starting ObjectGrid server using ObjectGrid XML file URL "{0}" and Cluster XML file URL "{1}".
**Explanation:** An ObjectGrid server is starting using a cluster XML file and an ObjectGrid xml file.
**User response:** none

**CWOBJ2503I**: Bootstrapping to a peer Objectgrid server on host {0} and port {1}.
**Explanation:** This ObjectGrid server will bootstrap to a peer server to retrieve information required to start.
**User response:** none

**COBJ2504I**: Attempting to bootstrap to a peer ObjectGrid server using the following host(s) and port(s) "{0}".
**Explanation:** This ObjectGrid server will use the list of hosts and ports provided in an attempt to connect to a peer ObjectGrid server.
**User response:** none

**CWOBJ2505I:** Attempting to bootstrap to a peer ObjectGrid server using the Cluster XML file URL "{0}".
**Explanation:** This ObjectGrid server will use the list of servers in the Cluster XML file in an attempt to connect to a peer ObjectGrid server.
**User response:** none

**CWOBJ2506I:** Trace is being logged to {0}.
**Explanation:** The trace file has been set on the command line.
**User response:** See the specified trace file for ObjectGrid server start-up trace.

**CWOBJ2507I:** Trace specification is set to {0}.
C**Explanation:** The trace specification has been set on the command line.
**User response:** none

**CWOBJ2508I:** A security properties file "{0}" has been specified and will be used to start the server.
**Explanation:** A security properties file has been provided to start a secure server.
**User response:** none

**CWOBJ2509E:** Timed out after waiting {0} seconds for the server to start.
**Explanation:** The ObjectGrid server did not start within the timeout interval.
**User response:** Check the log for exceptions.

**CWOBJ2510I:** Stopping ObjectGrid server {0}.
**Explanation:** Stopping ObjectGrid server.
**User response:**

**CWOBJ2511I:** Waiting for the server to stop.
**Explanation:** Waiting for the ObjectGrid server to stop.
**User response:** none

**CWOBJ2512I:** ObjectGrid server {0} stopped.
**Explanation:** ObjectGrid server stopped.
**User response:** none

**CWOBJ2513E:** Timed out after waiting {0} seconds for the server to stop.
**Explanation:** The ObjectGrid server did not stop within the timeout interval.
**User response:** Check the log for exceptions.

**CWOBJ2514I:** Waiting for ObjectGrid server activation to complete.
**Explanation:** The ObjectGrid server has launched. Waiting for the server to complete activation.
**User response:** none.

**CWOBJ2515E:** The arguments provided are invalid. Here are the valid arguments.{0}{1}
**Explanation:** The arguments provided to this script are invalid.
**User response:** Enter valid arguments.

**CWOBJ2516I:** ObjectGrid server has completed activation.
**Explanation:** The ObjectGrid server is active and ready to process requests.
**User response:** none.

**CWOBJ2517I:** Successfully bootstrapped to peer Objectgrid server on host {0} and port {1}.
**Explanation:** This ObjectGrid server has successfully bootstrapped to a peer server to retrieve information required to start this server.
**User response:** none

**CWOBJ2407W:** The {0} property on the {1} plug-in class could not be set. The exception is {2}.
**Explanation:** The property for this plug-in could not be set.
**User response:** See the exception for more information.

# Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, New York 10594 USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
522 South Road
Poughkeepsie, NY 12601-5400
USA
Attention: Information Requests

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

© Copyright IBM Corp. 2004, 2005

# Trademarks and service marks

The following terms are trademarks of IBM Corporation in the United States, other countries, or both:

- AIX
- CICS
- Cloudscape
- DB2
- Domino
- IBM
- Lotus
- RACF
- Redbooks
- Tivoli
- WebSphere
- z/OS

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

LINUX is a trademark of Linus Torvalds in the U.S., other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

© Copyright IBM Corp. 2004, 2005