
This presentation explains the Managed Client Services provided by IBM Lotus Expeditor 6.1 Client for Desktop.

## Goals

- Understand the function and major components provided by Managed Client Services




The goal of this presentation is to explain the Managed Client Services provided by IBM Lotus Expeditor 6.1 Client for Desktop.

# Agenda

- Overview
- Java™
- OSGi
- Eclipse
- Other Services



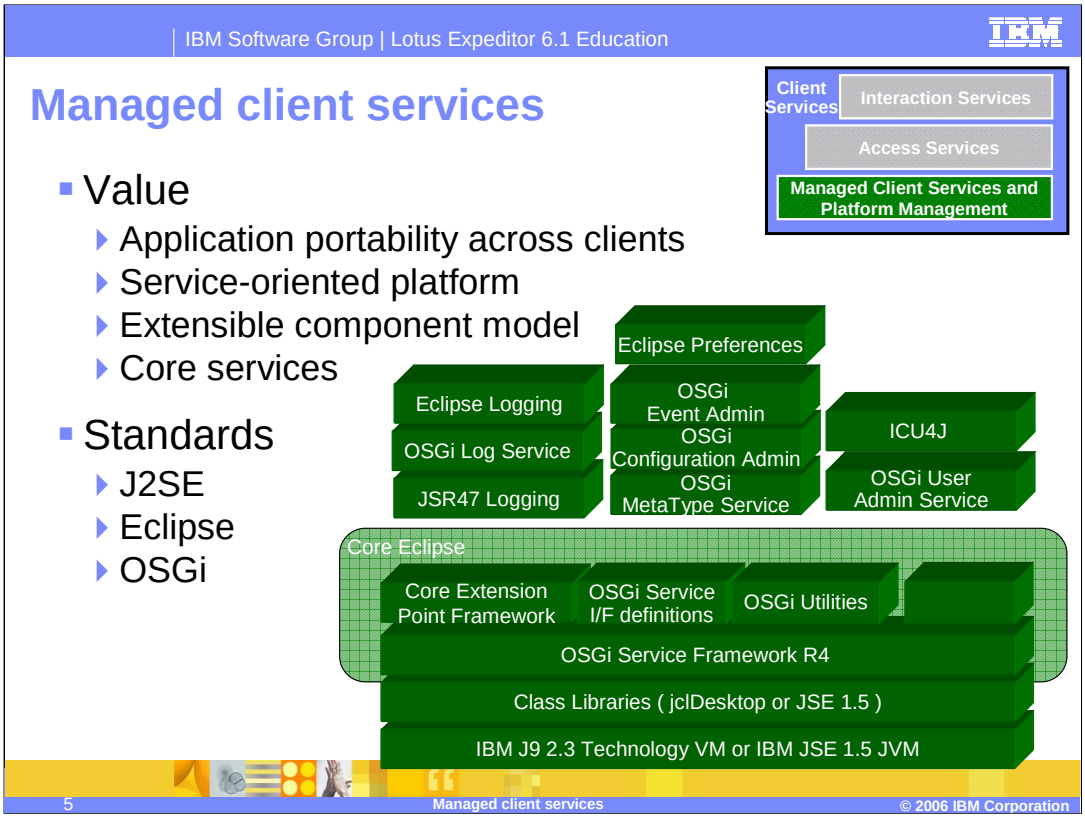

The agenda of this presentation is to provide an overview of the managed client services, describe the Java Runtime Environment, explain the OSGi framework, cover the key components of Eclipse, and describe other services.

## Section

# Overview




Let’s start with an overview of the Managed Client Services.




The client platform ships IBM's J9 2.3 JVM with JCLDesktop class libraries as the default installation. The Java Standard Edition (JSE) 1.5 JVM is optionally provided.

The client platform provides an OSGi Service Framework that implements the OSGi R4 framework specification and provides a service-oriented architecture on top of the JVM. The OSGi Service Framework is the foundation of Eclipse and the client platform. OSGi Services provide the interface definitions for standard services defined by the OSGi R4 specification and OSGi Utilities provide an implementation of the OSGi utilities interface. The Eclipse Core Extension Point Framework supports the Eclipse plug-in component model, and, by running with the OSGi Service Framework, enables plug-ins to receive the corresponding benefits of OSGi. All of these components ship in the core Eclipse framework.

The client platform provides implementations of additional services to assist you in the development of your applications including logging, preferences, globalization, and more.

## Section

# Java




Next, let's understand the Java Runtime Environment that ships with the client platform.

# Java Virtual Machine

JCLDesktop Class Libraries

IBM J9 2.3 VM

- IBM J9 2.3 Virtual Machine shipped with installation
  - JCLDesktop class libraries
  - J9 2.3 with jclDesktop is a custom configuration that provides a reduced-footprint runtime environment for the client platform.
  - Provided as a feature




J9 2.3 with JCLDesktop is a custom configuration that provides a reduced-footprint runtime environment for the client platform. jclDesktop is the default configuration for desktops.

# Java Virtual Machine

J2SE 1.5 Class Libraries

IBM J2SE 1.5 JVM

- IBM Virtual Machine for Java shipped with installation
- You must use the JVM provided.
- Provided as a feature
- Windows® XP

```
java version "1.5.0“
Java 2 Runtime Environment, Standard Edition (build pwi32dev-20060511 (SR2))
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Windows XP x86-32 j9vmwi3223-20060504 (JIT enabled)
J9VM - 20060501_06428_lHdSMR
JIT  - 20060428_1800_r8
GC   - 20060501_AA)
JCL  - 20060511a
```

- Linux®

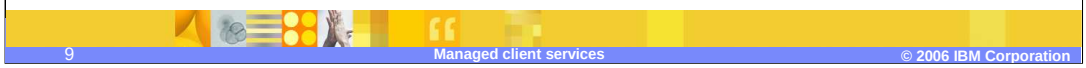
```
java version "1.5.0“
Java 2 Runtime Environment, Standard Edition (build pwi32dev-20060511 (SR2))
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9xia3223-20060504 (JIT enabled)
J9VM - 20060501_06428_lHdSMR
JIT  - 20060428_1800_r8
GC   - 20060501_AA)
JCL  - 20060511a
```




The IBM Virtual Machine for Java version 1.5 is shipped as an optional installation with the Expeditor client. This slide provides some details of the version 1.5 virtual machine.

## Changing the Java Virtual Machine

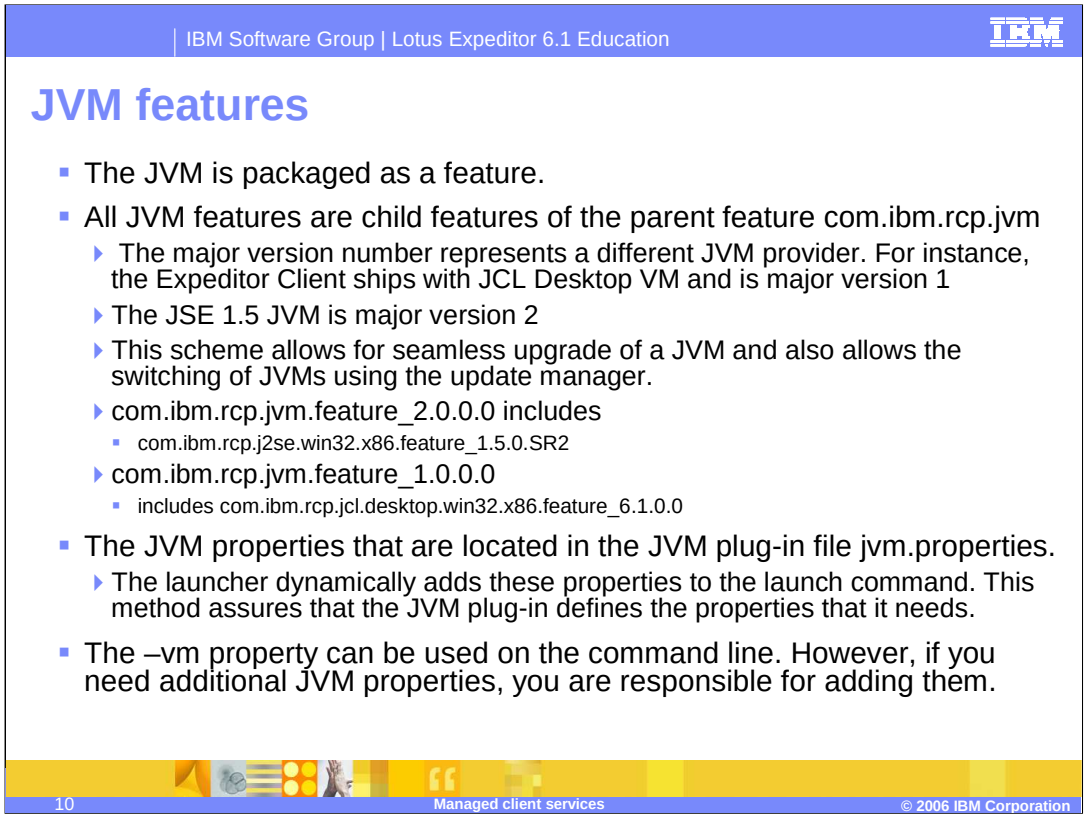
- The Expeditor Client installs by default the J9 2.3 VM with JCLDesktop class libraries
- To switch to use the J2SE 1.5 JVM
  - Install the com.ibm.rcp.jvm.feature_2.0.0.0 feature
  - Located on the update site of the Lotus Expeditor Desktop Runtime Environment CD under updates/platform






The Expeditor Desktop Client installs the J9 with jclDesktop as the default VM.

If you want to switch to use a J2SE JVM for the platform after installing Expeditor Client, you will need to install the com.ibm.rcp.jvm.feature_2.0.0.0 feature from the update site located on the Lotus Expeditor Desktop Runtime Environment. The update site can be found on the Desktop Runtime Environment CD-ROM at updates/platform/.




The JVM is packaged as a feature.

All JVM features are child features of the parent feature com.ibm.rcp.jvm. The major version number represents an entirely different JVM provider. For instance, the Expeditor Client ships with JCL Desktop JVM and is major version 1. This scheme allows for seamless upgrade of a JVM and also allows the switching of JVMs using the update manager.

The JVM properties are located in the JVM plug-in file jvm.properties. The launcher dynamically adds these properties to the launch command. This method assures that the JVM plug-in defines the properties that it needs.

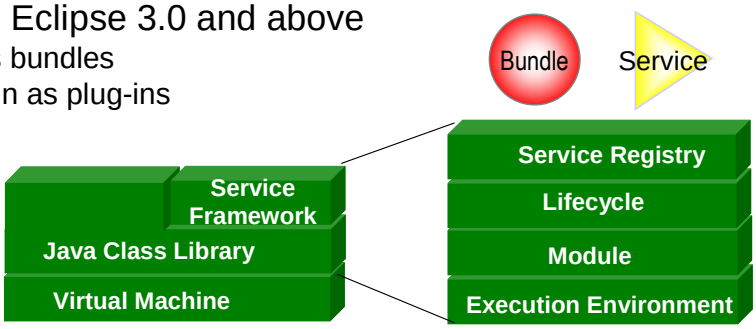
The –vm property can be used on the command line. However, if you need additional JVM properties, you are responsible for adding them.


Next, let’s explore the OSGi service framework, which is a critical component of the client platform.

# OSGi framework

OSGi Service Framework

- Value
  - Allows simultaneous execution of applications and services on “fit for purpose” class libraries on a single JVM instance
  - Enables applications to share services and packages
  - Separates service interface from service implementation
  - Supports independent life-cycle management of applications and services on a single JVM instance
- Integrated into Eclipse 3.0 and above
  - Plug-ins run as bundles
  - Bundles can run as plug-ins
- Standards
  - OSGi R3+







The OSGi framework specification is provided by the OSGi Alliance. The OSGi Alliance’s mission is to specify, create, advance, and promote wide industry adoption of an open service delivery and management platform. Incorporating the OSGi standard into the client platform provides four very important capabilities:

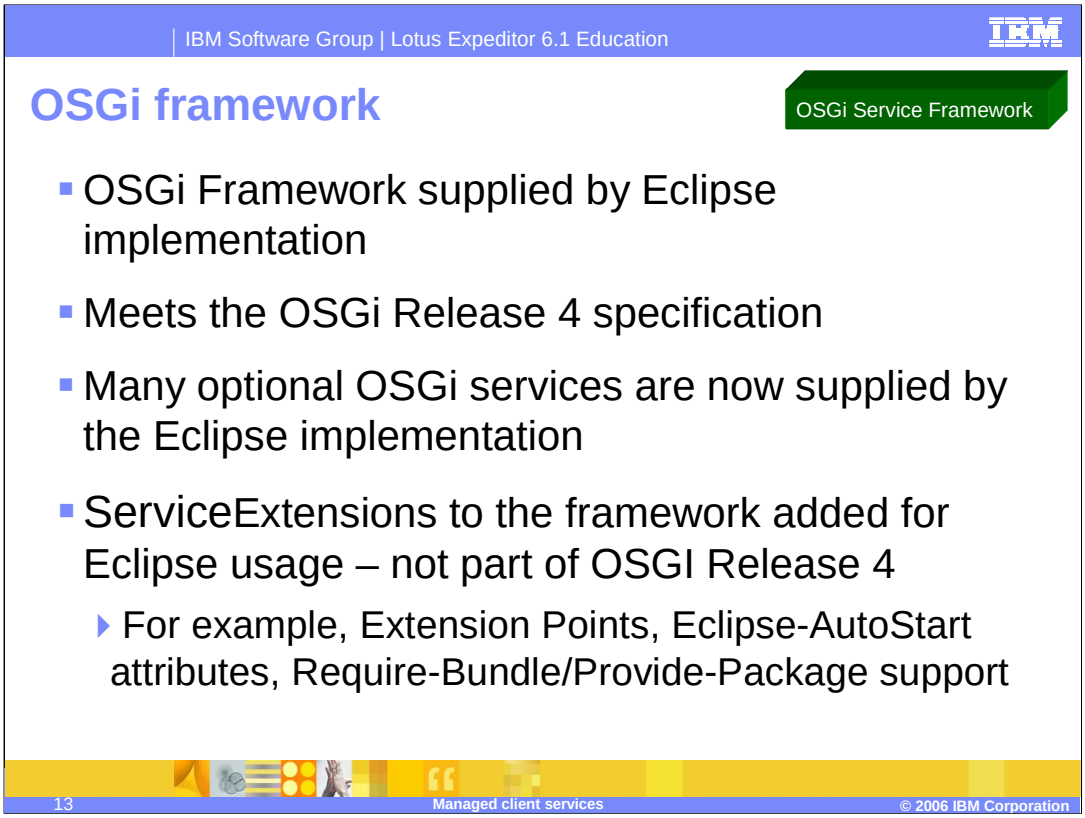
It enables multiple applications and components to share a single JVM. This saves valuable resources on the client when running multiple applications because only one instance of the JVM is launched rather than multiple instances of the JVM.

It enables applications to share services and packages, which further reduces resource requirements on devices.

It separates service interface from service implementation and provides publish, find, and bind operations in support of a service-oriented architecture. This capability enables integration of business applications on the same device.

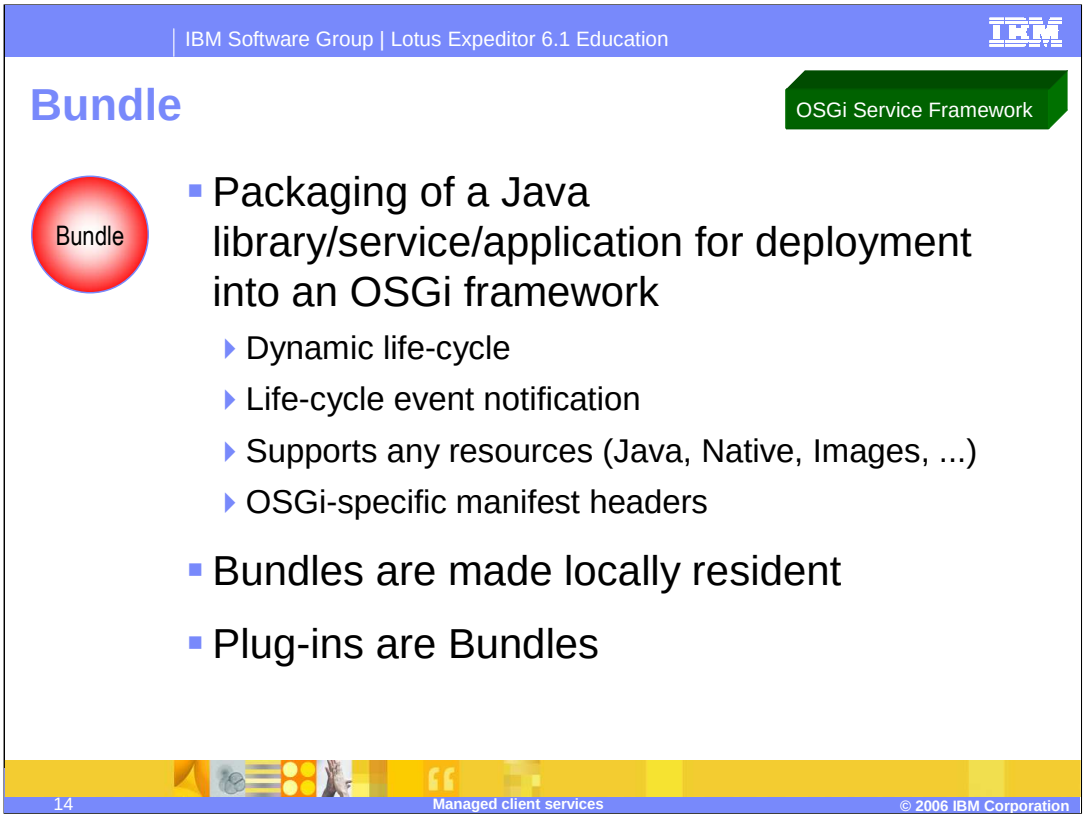
It enables dynamic life cycle management without a VM restart so components can be updated without impacting other unrelated components that are running at the same time.

The Eclipse framework is built on the OSGi Service Framework, which provides the Eclipse with powerful capabilities, such as the ability to dynamically load and unload plug-ins without restarting the Eclipse framework and robust life cycle management of plug-ins.  Therefore, you can define each component in your applications as a plug-in, a bundle, or both depending on your requirements.

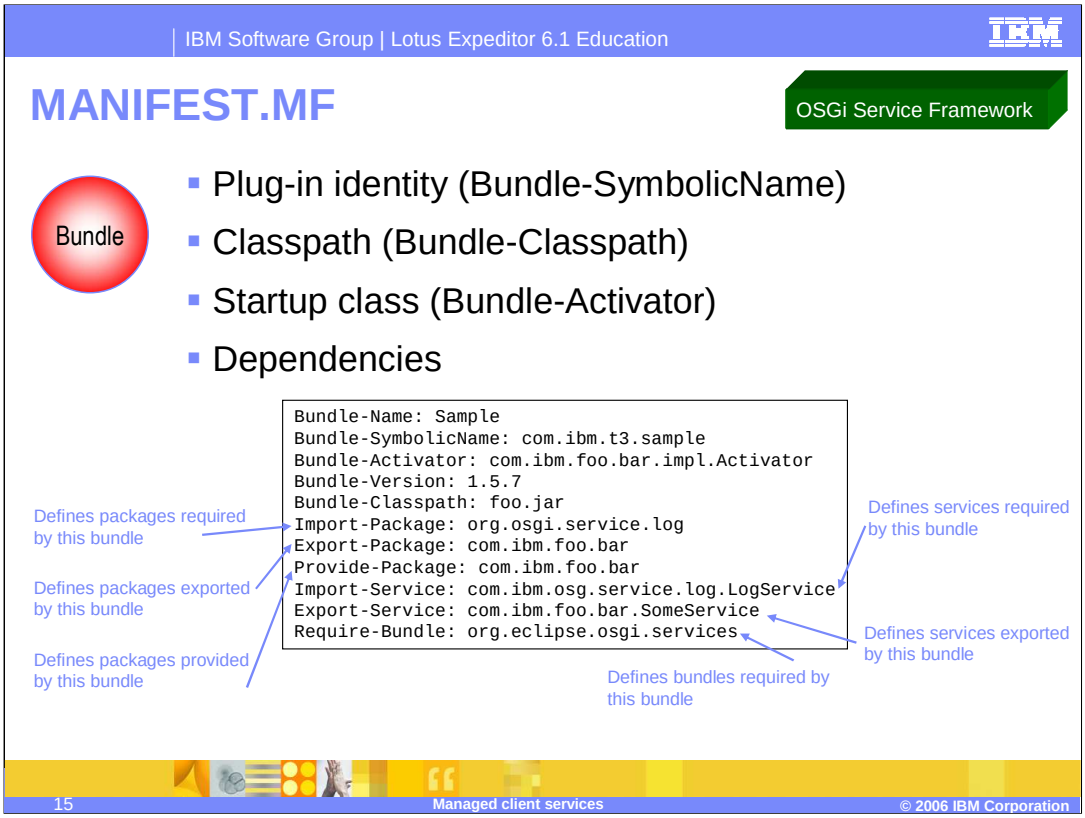



Lotus Expeditor is built on the Eclipse Rich Client Platform, which includes an OSGi framework. The framework is based upon the OSGi Service Platform Release 4 specification with additional extensions provided by the Eclipse 3.2.1 implementation of the OSGi framework.

The OSGi framework is at the R4 specification level. Many optional OSGi services are now provided by the Eclipse implementation. Service Extensions to the framework, such as extension points, eclipse auto-start attributes, and require-bundle and provide-package support, are not part of OSGi Release 4.



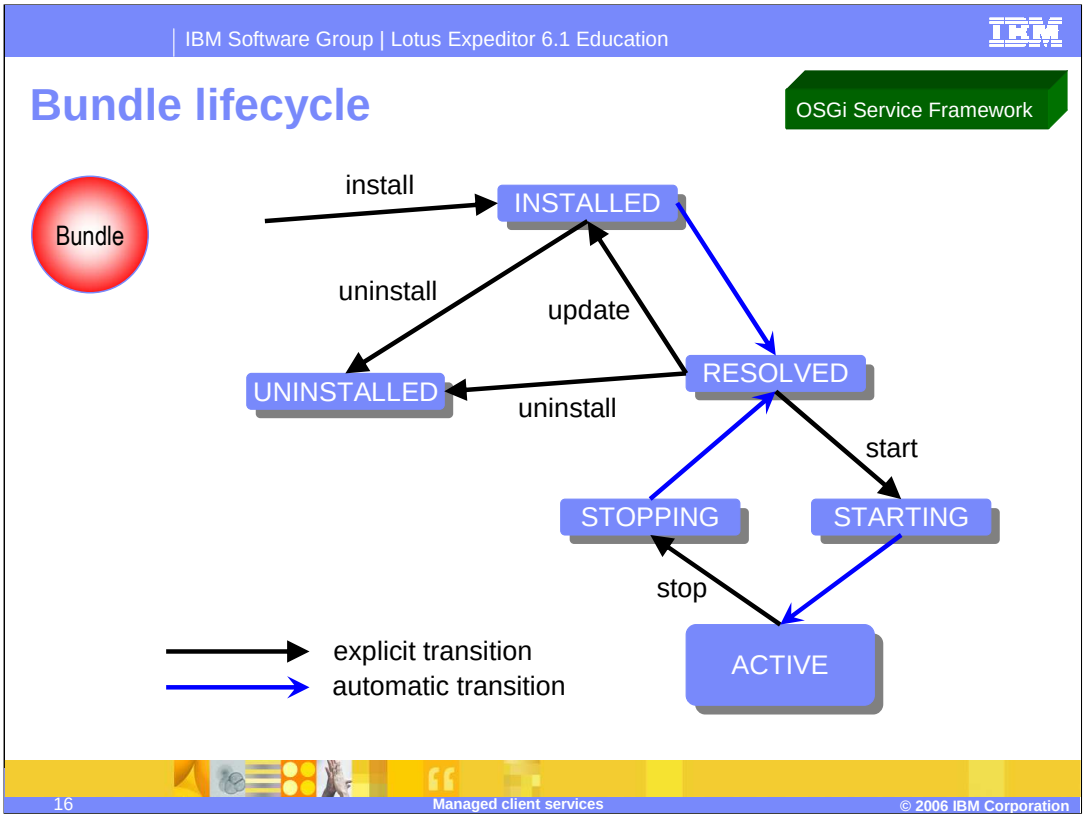

Application developers partition applications into services and other resources. Services and resources are packaged into bundles, which are files that serve as the delivery unit for applications. OSGi bundles consist of a JAR file that contains Java classes, resources, and a manifest file. Bundles can register services for other bundles to use, use services registered by other bundles, export Java packages for other bundles to use, and import Java packages from other bundles. Within the Eclipse based platforms, all plug-ins are OSGi bundles, so you can think of the terms plug-in and bundle as being interchangeable.




The bundle's manifest file contains data that the framework needs to correctly install and activate the bundle. Legacy Eclipse bundles can provide some manifest information in their plugin.xml files, but META-INF/MANIFEST.MF files are the recommended files for Manifest information. Note that a plugin.xml may contain similar information, however, a plugin.xml also contains extensions and extension points.

If a bundle contains only a plugin.xml, the Eclipse platform will generate a MANIFEST.MF equivalent when the platform starts. When you specify data in a manifest file, you must use the headers that were defined by the OSGi specification. You can use user-defined headers; however, the framework ignores any headers that it does not understand. Refer to the OSGi Service Platform Release 4 specification for more information about the OSGi Manifest file format and syntax.

The MANIFEST.MF file is located in the META-INF directory of your bundle project. The plugin.xml file, if present, should be under the root directory.




The framework manages the life cycle of bundles. As you install and run a bundle, it goes through various states. The possible states of a bundle are:

INSTALLED - the bundle has been installed, but all of the bundle's dependencies have not been met. The bundle requires packages that have not been exported by any currently installed bundle.

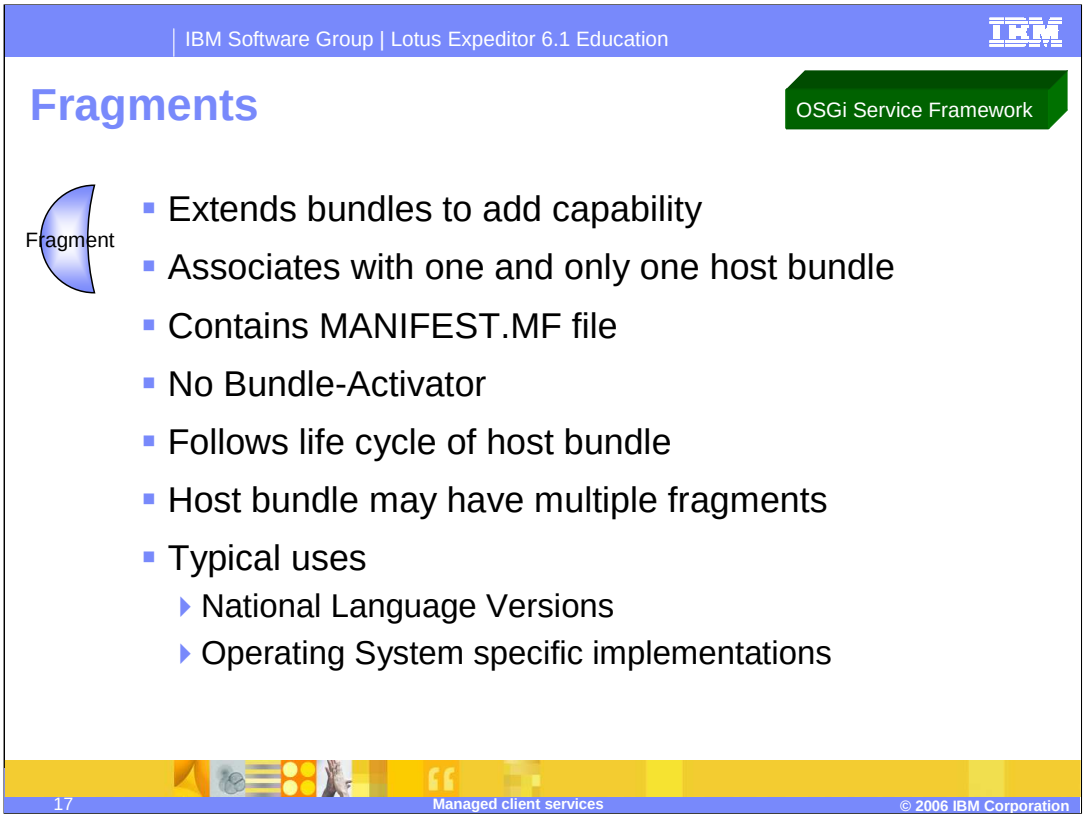
RESOLVED - the bundle is installed, and its dependencies have been met, but it is not running. If a bundle is started and all of the bundle's dependencies are met, the bundle skips this state.

STARTING - a temporary state that the bundle goes through while the bundle is starting.

ACTIVE - the bundle is running.

STOPPING - a temporary state that the bundle goes through while the bundle is stopping.

UNINSTALLED - the bundle no longer exists in the framework.

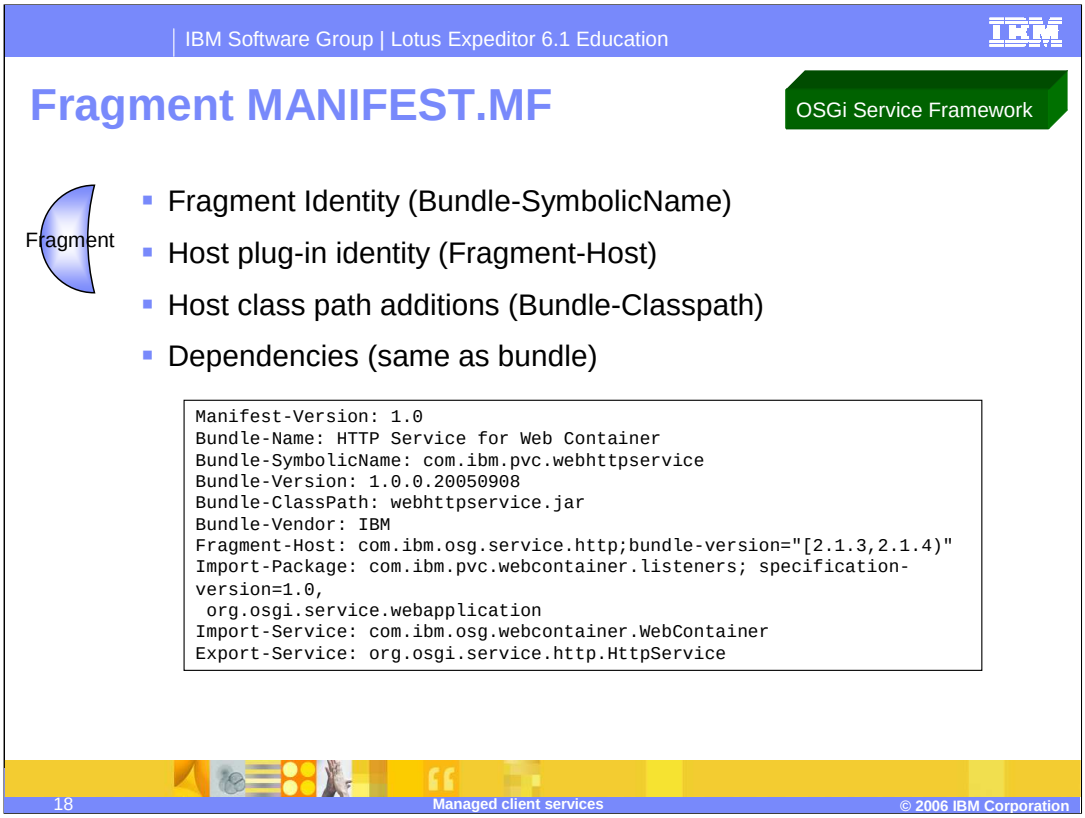



A component may not always provide a complete implementation. In some cases, *fragments* may be used to complete or extend a component. For example, the primary component may provide an implementation that contains translatable text in a default language. Fragments are then used to provide translations for additional languages.

A second case where fragments are often used is to provide platform-specific implementations, involving the processor and operating system, for example.

Fragments contain either a fragment.xml (similar to a plugin.xml), or a MANIFEST.MF, or both. A fragment is associated with, or dependent upon, a specific primary component, but still maintains a unique identity. Querying a list of components will also return fragments, so that these fragments can be individually started and stopped.

Fragments generally add classes or resources to the class path normally used by the primary component. Fragments do not contain Bundle-Activator classes. Since fragments are only extensions to a component, they cannot be required or imported by another component.




Fragments contain either a fragment.xml (similar to a plugin.xml), or a MANIFEST.MF, or both. A fragment is associated with, or dependent upon, a specific primary component, but still maintains a unique identity. Querying a list of components will also return fragments, so that these fragments can be individually started and stopped.

Fragments generally add classes or resources to the class path normally used by the primary component. Fragments do not contain Bundle-Activator classes. Since fragments are only extensions to a component, they cannot be required or imported by another component.

# Services

OSGi Service Framework


- Java object
  - Registered by a bundle
  - Used by other bundles
- Specified by a Java interface
  - Clean separation of service specification and service implementation
  - Different bundles can register different implementations of the same service




In the OSGi environment, bundles are built around a set of cooperating services that are available from a shared service registry. The service interface defines the OSGi service, which is implemented as a service object.

Services decouple the provider from the service user. The only code a service provider and a service user share is the service definition. You can use Java interfaces to define services. Any class that implements this interface can provide the service.

Bundles that use services that are not provided by the bundle can notify the framework by including an Import-Service header in the bundle manifest. However, this is not required. When code within a bundle requests a provider of the service from the framework, the bundle imports the service at runtime.

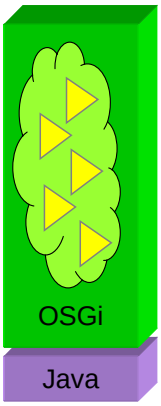
A bundle that provides services can also include an Export-Service header in its manifest. When code within a bundle makes a provider available to the framework, the bundle exports the service at runtime.

The framework passes a BundleContext object to your bundle when it invokes your BundleActivator's start method. Your bundle can use the BundleContext object to interact with the framework by calling the methods of the BundleContext object. One method that your bundle can call is registerService, which uses a service object and an interface name to register a service with the framework's service registry.

The recommended approach for using services is to provide all interface and object classes referred to in the service definition in a bundle separate from the service implementation. The service implementation bundle then imports the packages from the defining bundle, and exports no packages of its own. Therefore in a typical service usage, there are three bundles involved – a service interface bundle, the service implementation, and the service consumer.

# Service registry

OSGi Service Framework




- Dynamic registry of available services
  - Secured by permissions
- Service object registered
  - Under one or more interface names
  - With optional properties
- Lookup mechanism
- Service Factory
  - Callback to create custom service objects
- Life cycle event notification
  - Registered, Modified, Unregistering

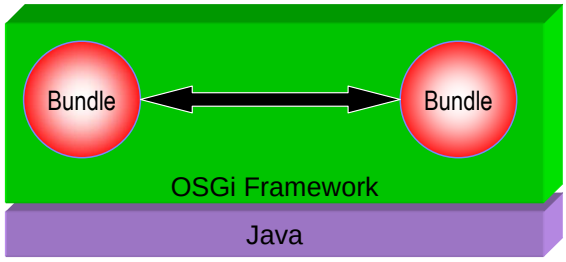



In the OSGi environment, bundles are built around a set of co-operating services that are available from a shared service registry. The service interface defines the OSGi service, which is implemented as a service object.

## Collaboration

OSGi Service Framework

- Bundles can collaborate
  - Import/Export Services
  - Import/Export Packages
- Dependencies managed







OSGi bundles consist of a JAR file that contains Java classes, resources, and a manifest file. Bundles can register services for other bundles to use, use services registered by other bundles, export Java packages for other bundles to use, and import Java packages from other bundles.

# Package sharing

OSGi Service Framework

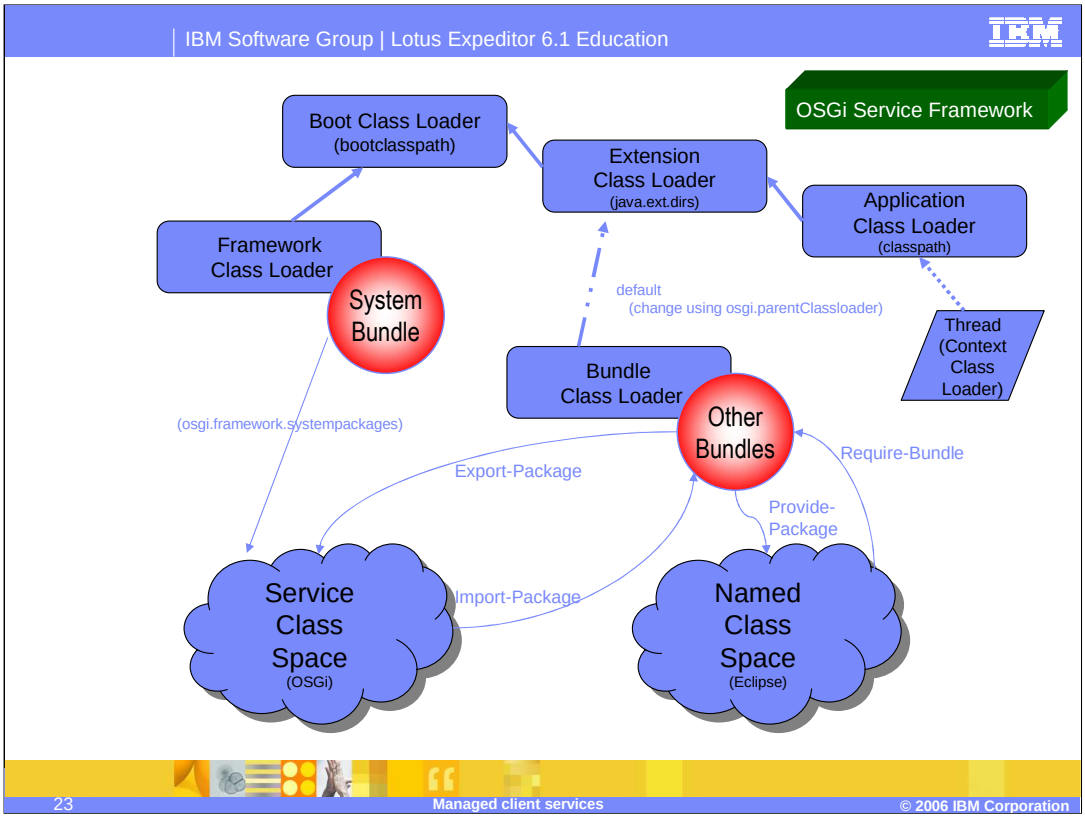
- Each bundle has its own ClassLoader
  - Bundles have separate namespaces
- Java classes are grouped into packages
  - Bundles can offer to share a package – export
  - Bundles can request to use a package – import
- Global shared package namespace
  - Class Loaders connected to allow classes to be shared
- Versioning
  - Only one version of a package may be shared at a time - defined by specification version




Bundles can use code that is defined within other bundles by declaring the packages as imported packages in the manifest file. Although you can create a bundle that does not rely on any classes other than the Java base packages, most bundles import code from other bundles or the base runtime class path.

You must import any class that you use within a bundle that is not defined in the bundle or that is not a base Java class, meaning classes within packages that begin with java.. To import another class, include an import clause for the class's package in the bundle's manifest. You can explicitly import only whole packages; individual classes cannot be explicitly imported.

A bundle can make the classes the bundle defines available to other bundles by exporting packages. To enable other bundles to access a particular package, include an export clause for the package in the manifest of the bundle that contains the package.



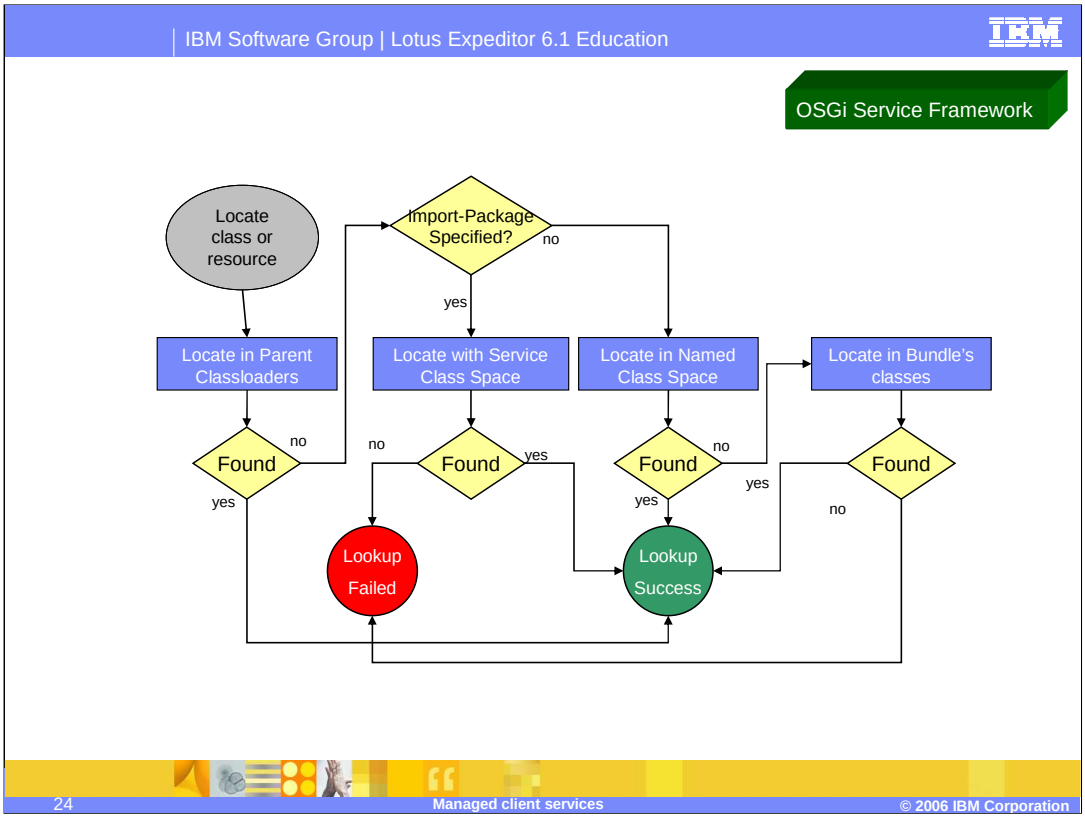

Class loading functions differently than a typical Java application because the client platform is built on the OSGi Service Framework. Since the mechanics for supporting plug-ins are implemented by using the OSGi Service Framework, a plug-in is the same as an OSGi bundle for the purpose of this explanation. The bundle and its associated classes specify and implement the process for Java class loading, prerequisite management, and the bundle's life cycle.

Each bundle installed and resolved in the OSGi Service Framework must have a class loader. This class loader, called the Bundle Class Loader, provides each bundle with its own name space to avoid name conflicts and enables package sharing with other bundles.

The Bundle Class Loader searches for classes and resources in the bundle's class path as defined by the Bundle-Classpath header in the bundle's manifest. The Bundle Class Loader has a parent class loader as specified in the osgi.parentClassloader property. By default, the parent class loader is the Extension Class Loader for the client platform. However, the Extension Class Loader also has a parent class loader - the Boot Class Loader. As a result, the parent of the Bundle Class Loader actually consists of the Boot Class Loader and the Extension Class Loader.

A bundle can export the classes and resources in one or more of its packages by specifying each such package name in the Export-Package header in its manifest. The classes and resources in each exported package become part of the Service Class Space and are made available to other bundles with permission to use the package. A bundle can import one or more packages by specifying each package name in the Import-Package header in its manifest. If the bundle has permission to import these packages, then the bundle can use the classes and resources in these packages as defined in the Service Class Space. A package can be shared based on its name and, optionally, its version. However, if multiple bundles share a package with the same name, then the OSGi Service Framework determines the bundle that shares that package with other bundles based on the highest version of the declared package. As a result, a bundle that imports a package must know the name of the package it needs to import but cannot explicitly control which bundle provides the package it actually uses.

A bundle can also provide the classes and resources in one or more of its packages by specifying each such package name in the Provide-Package header in its manifest. The classes and resources in each provided package become part of the Named Class Space and are made available to other bundles with the appropriate permissions. A bundle can explicitly use packages provided by a bundle by specifying each required bundle in the Require-Bundle manifest in its header. The Require-Bundle manifest header contains a list of bundle symbolic names that need to be searched after the imports are searched but before the bundle's class path is searched. However, only packages that are marked as provided by the required bundles are visible to the requiring bundle.




To locate a class or resource, the search order is as follows:

The Bundle Class Loader delegates the request to its parent class loader, which results in the Boot Class Loader and then the Extension Class Loader attempting to locate the class or resource. If the class or resource was found, then the class loader returns this result. If the class or resource was not found, then the search continues with the next step.

If the Bundle Class Loader determines that the requested class or resource is in a package imported from the Service Class Space and it was found in the Service Class Space, then the class loader returns this result. If the class or resource is not found, then the request fails. If the Bundle Class Loader determines that the requested class or resource was not in the Service Class Space, then the search continues with the next step.

The Bundle Class Loader searches the Named Class Space and if the class or resource was found, then the class loader returns this result. Otherwise, the search continues with the next step.

The Bundle Class Loader searches its own internal class path and the internal class path of any attached fragment bundles. If the class or resource was found, then the class loader returns the results. If the class or resource was not found, then the search terminates and the request fails.

## Section

# Eclipse




Next, let’s review the Eclipse components in the client platform.

## Eclipse

- IBM Lotus Expeditor 6.1 Client for Desktop includes the following Eclipse components:
  - Eclipse 3.2.1
  - Updates/patches to some Eclipse plug-ins
  - The Eclipse RCP (Rich Client Platform)
  - Additional plug-ins from the SDK
- Customers can add other Eclipse plug-ins not included by IBM
  - IBM does not support these plug-ins
  - Versions should be the Eclipse 3.2.1 version

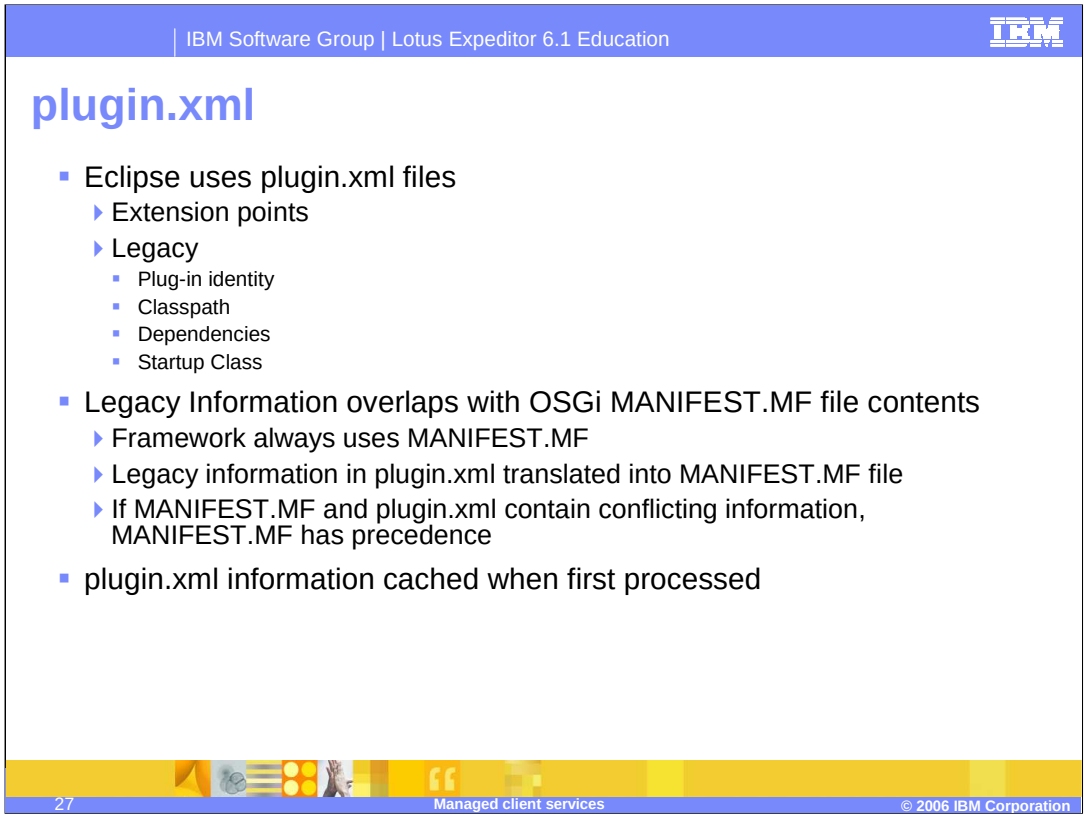



Lotus Expeditor is based upon the Eclipse Rich Client Platform, but has added a set of Eclipse plug-ins above and beyond the set normally part of RCP. In addition, Lotus Expeditor has provided a workbench advisor and an application that can be used for many situations.

Application developers may have constructed applications based solely upon the Eclipse Rich Client Platform (or other Eclipse-based platforms), without necessarily targeting Lotus Expeditor.

Generally, you can run applications which are written for generic RCP on Lotus Expeditor. However, there may be some differences between the generic Eclipse RCP or SDK and Lotus Expeditor.

For more information, see the section “Integrating existing RCP applications into Lotus Expeditor” in the Lotus Expeditor Developer’s Guide.




Each bundle must contain either a manifest file or a plugin.xml file. The bundle's manifest file contains data that the framework needs to correctly install and activate the bundle. Legacy Eclipse bundles can provide some manifest information in their plugin.xml files, but META-INF/MANIFEST.MF files are the recommended files for Manifest information. Note that a plugin.xml may contain similar information, however, a plugin.xml also contains extensions and extension points.

If a bundle contains only a plugin.xml, the Eclipse platform will generate a MANIFEST.MF equivalent when the platform starts.

# fragment.xml

- Eclipse uses fragment.xml files
  - Extension points
  - Legacy
    - Fragment identity
    - Host identify
    - Classpath
    - Dependencies
- Legacy Information overlaps with OSGi MANIFEST.MF file contents
  - Framework always uses manifests
  - Legacy information in fragment.xml translated into MANIFEST.MF file
  - If MANIFEST.MF and fragment.xml contain conflicting information, MANIFEST.MF has precedence




Fragments contain either a fragment.xml (similar to a plugin.xml), or a MANIFEST.MF, or both. A fragment is associated with, or dependent upon, a specific primary component, but still maintains a unique identity. Querying a list of components will also return fragments, so that these fragments can be individually started and stopped.

## Section

# Other services




Finally, let's review the other managed client services available to you in the client platform.

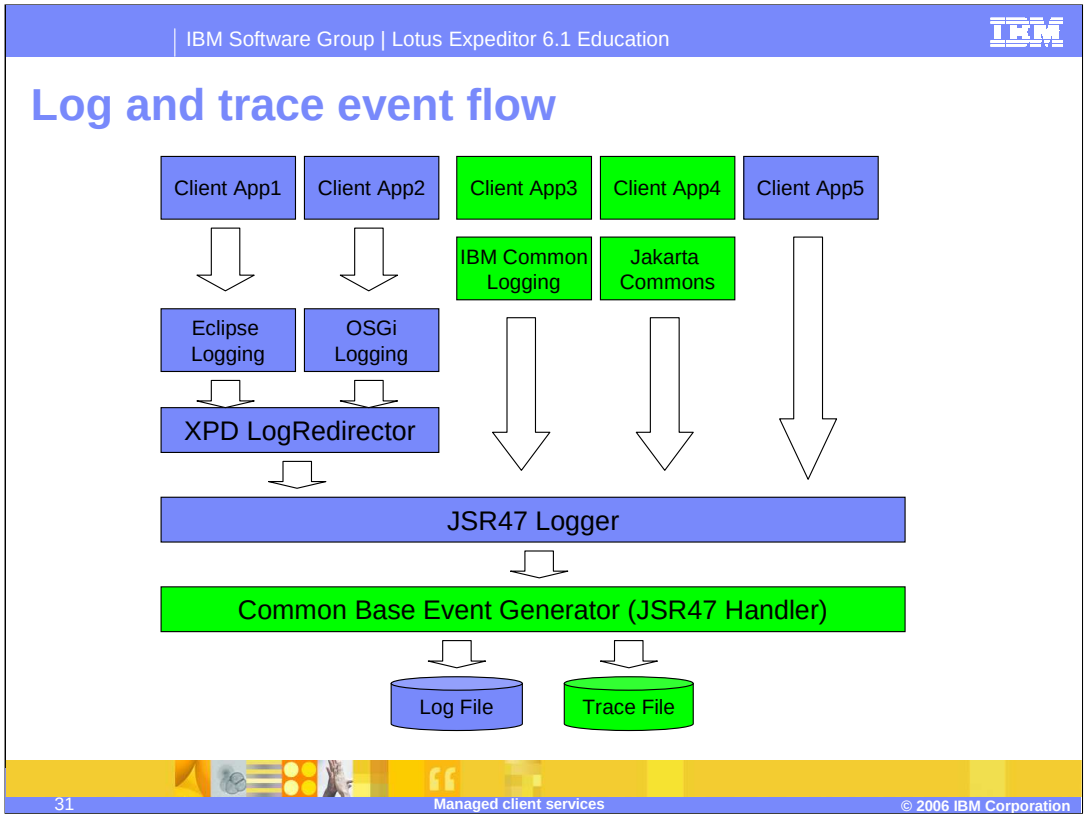
## Logging

- Applications can use the following methods for logging messages
  - Eclipse logging interface
  - OSGi LogService interface
  - Apache Commons Logging API
  - Standard error and standard out
  - java.util.logging interface (JSR47)
- Expeditor collects log messages in one file
  - Captures messages logged by all logging methods.
  - Maps logging levels to JSR47 logging levels
  - Redirects messages to JSR47 java.util.logging



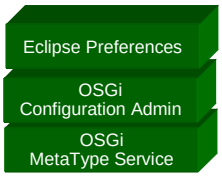

You use logs to gather information about problems that might happen while using the client platform. The OSGi framework, Eclipse framework and the java.util.logging all provide different systems for logging messages. Applications in some cases also leverage standard error and standard out for message delivery, and these messages must also be captured to ensure all data is available during the problem determination stage. The client platform provides a plug-in, called the logRedirector, that collects all messages logged in the client platform into one persistent log file. The logRedirector captures messages logged from the OSGi logService, the Eclipse logging APIs, and standard error and standard out and redirects them to the JSR47 java.util.logging. A java.utils.logging log file manager is also provided by the client platform, which supports configuration of the JDK logging and manages the persistent log file.

Each of the logging systems available in the client platform has its own definition of logging levels, which provide information on the severity and type of logging information. To bring all of the messages from these disparate systems together, the logging system maps logging levels from the Eclipse logging service and the OSGi LogService to the logging levels defined by JDK 1.4 (for example, Severe, Warning, Info, Finest).




This diagram gives a high-level overview of the logging system in Expeditor 6.1. Applications may choose to log using many different mechanisms, including OSGi logging, Eclipse logging, IBM Common logging, and Apache logging, however they all consolidated at the JSR47 logger level so that all log messages are logged to one file.

## Preferences




- Preferences are key/value pairs
  - key describes the name of the preference
  - value is one of several different types, including Boolean, double, float, int, long, and string
- Eclipse platform stores plug-in preferences and supports showing these preferences in Preferences dialog box
- Client platform adds OSGi Configuration Administrative Service
  - Persistently stores preference information
  - Enables system administrators to query and update configuration values with Enterprise Management Agent
  - Enables use of the OSGi Metatype Service to provide a metadata description of the parameter types, default values, and validation logic to be applied for each parameter




Preferences are key/value pairs, where the key describes the name of the preference, and the value is one of several different types, including Boolean, double, float, int, long, and string.

The Eclipse platform provides support for storing plug-in preferences and showing them to the user on pages in the workbench Preferences dialog box.

The client platform extends the Eclipse capabilities by including the OSGi Configuration Administrative service. Configuration Administration provides capabilities to store preference or configuration information based on key/value pairs. Applications that use Configuration Administration will be notified when configuration information changes. Applications that use Configuration Administration to store configuration and preference information can also use the Metatype service to provide a metadata description of the information. The metadata can describe the parameter types, default values, and validation logic to be applied for each parameter. If Configuration Administration is used to store configuration information, system administrators can query and update configuration values using the Enterprise Management Agent.

## Other OSGi services


- OSGi MetaType Service – Enables an administrative bundle to dynamically discover what another bundle's configuration looks like and make changes to it.
- OSGi Event Administration - provides an inter-bundle communication mechanism. It is based on an event publish and subscribe model.




Additional OSGi Services include:

OSGi MetaType Service, which enable an administrative bundle to dynamically discover what another bundle's configuration looks like and make changes to it.

OSGi Event Admin, which provides an inter-bundle communication mechanism. It is based on an event publish and subscribe model, popular in many message based systems.

## Other OSGi services


- OSGi Utilities – Provides a consistent way to handle a diverse range of measurements and geographic positions
- OSGi User Administration Service - Supports authentication and authorization of the users of an OSGi platform




Additional OSGi Services include:

OSGi Utilities, which provide a consistent way to handle a diverse range of measurements and geographic positions

OSGi User Administration Service, which supports authentication and authorization of the users of an OSGi platform

# Additional services


- International Components for Unicode for Java (ICU4J) – Supports globalization of applications
  - Support multiple locales
  - Support bidirectional text layouts
  - Create translatable components
- XML Parsers
  - Use standard Java API for XML Parsing (JAXP)
  - Use XML Parsing Service I/F to dynamically select parser at runtime – requires additional code




You can use International Components for Unicode for Java (ICU4J) to globalize your applications. ICU4J enables you to:

Support multiple locales

Support bidirectional text layouts

Create translatable components

You can also use XML Parsers to process XML data.

# Trademarks, copyrights, and disclaimers

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM Lotus

Windows, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

J2SE, Java, JVM, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.



Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.




This concludes the presentation.