

IBM WebSphere Commerce



# Customizing the WebSphere Commerce Accelerator and other Tools User Interface Centers reference

*Version 5.5*



IBM WebSphere Commerce



# Customizing the WebSphere Commerce Accelerator and other Tools User Interface Centers reference

*Version 5.5*

**Note:**

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 79.

**First Edition, March 2004**

This edition applies to IBM WebSphere Commerce Business Edition Version 5.5, IBM WebSphere Commerce Professional Edition Version 5.5, and IBM WebSphere Commerce - Express Version 5.5 (product number 5724-A18), and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality.

IBM welcomes your comments. You can send your comments by using the online IBM WebSphere Commerce documentation feedback form, available at the following URL:

[www.ibm.com/software/webservers/commerce/rcf.html](http://www.ibm.com/software/webservers/commerce/rcf.html)

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 International Business Machines Corporation 2002, 2004. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>About this book</b> . . . . .	<b>v</b>
Document description . . . . .	v
Updates to this book . . . . .	v
Conventions used in this book . . . . .	v
Path variables . . . . .	v
Knowledge requirements . . . . .	vi

---

## **Part 1. Customizing the WebSphere Commerce Accelerator and other Tools User Interface Centers reference.** . . . . . **1**

<b>Chapter 1. Wizards</b> . . . . .	<b>3</b>
Overview . . . . .	3
Detailed steps . . . . .	4
Navigation . . . . .	10
Customizations . . . . .	11
JavaScript functions . . . . .	12

<b>Chapter 2. Notebooks</b> . . . . .	<b>13</b>
Overview . . . . .	13
Detailed steps . . . . .	14
Navigation . . . . .	19
Customizations . . . . .	21
Error code handling . . . . .	21
JavaScript functions . . . . .	22

<b>Chapter 3. Dialogs.</b> . . . . .	<b>23</b>
Overview . . . . .	23
Detailed steps . . . . .	23
Navigation . . . . .	29
Customizations . . . . .	30
JavaScript functions . . . . .	30

<b>Chapter 4. Dynamic lists</b> . . . . .	<b>31</b>
Overview . . . . .	31
Detailed steps . . . . .	32
Multiple Framesets . . . . .	39
Filter enhancement . . . . .	39
JavaScript functions . . . . .	39

<b>Chapter 5. Calendars</b> . . . . .	<b>41</b>
---------------------------------------	-----------

Overview . . . . .	41
Detailed steps . . . . .	41
JavaScript functions . . . . .	42

<b>Chapter 6. Slosh buckets.</b> . . . . .	<b>45</b>
Overview . . . . .	45
Detailed steps . . . . .	45
Customizations . . . . .	46
JavaScript functions . . . . .	46

<b>Chapter 7. Tools User Interface Center</b> . . . . .	<b>49</b>
Integrating tools into a Tools User Interface Center . . . . .	49
Adding context-sensitive help . . . . .	50
JavaScript functions . . . . .	51

<b>Chapter 8. Dynamic tree</b> . . . . .	<b>55</b>
Workflow . . . . .	55
Overview . . . . .	55
Detailed steps . . . . .	55
Additional features . . . . .	61
JavaScript functions . . . . .	62

<b>Chapter 9. Search dialogs</b> . . . . .	<b>63</b>
Overview . . . . .	63
Detailed steps . . . . .	63
Navigation . . . . .	70
Customizations . . . . .	70

---

## **Part 2. Element chaining and wizard branching.** . . . . . **73**

<b>Chapter 10. Element chaining</b> . . . . .	<b>75</b>
Creating an element chain . . . . .	75

<b>Chapter 11. Wizard branching</b> . . . . .	<b>77</b>
Wizard branching example . . . . .	77

<b>Notices</b> . . . . .	<b>79</b>
Trademarks . . . . .	80



---

## About this book

---

### Document description

This document serves as an overview of the concepts involved with customizing the WebSphere® Commerce Accelerator. It addresses the high level architecture of how the user interface interacts with the business users, and the WebSphere Commerce Server. Supplementary documents, released as they become available, build upon the knowledge developed in this document, and provide the detailed information required to customize the particular components of the WebSphere Commerce Accelerator. They also act as a resource, listing the components and assets upon which the various components depend.

---

### Updates to this book

The latest version of this book, is available as a PDF file from the IBM® WebSphere Commerce technical library Web site:

[www.ibm.com/software/commerce/library/](http://www.ibm.com/software/commerce/library/)

---

### Conventions used in this book

This book uses the following highlighting conventions:

<b>Boldface type</b>	Indicates commands or graphical user interface (GUI) controls such as names of fields, icons, or menu choices.
Monospace type	Indicates examples of text you enter exactly as shown, file names, and directory paths and names.
<i>Italic type</i>	Used to emphasize words. Italics also indicate names for which you must substitute the appropriate values for your system.



This icon marks a Tip - additional information that can help you complete a task.

---

#### **Important**

These sections highlight especially important information.

#### **Attention**

These sections highlight information intended to protect your data.

---

### Path variables

This guide uses the following variables to represent directory paths:

*WC\_installdir*

This is the installation directory for WebSphere Commerce. The following are the default installation directories for WebSphere Commerce on various operating systems:

- ▶ **AIX** /usr/WebSphere/CommerceServer55
- ▶ **400** /QIBM/ProdData/CommerceServer55
- ▶ **Linux** /opt/WebSphere/CommerceServer55
- ▶ **Solaris** /opt/WebSphere/CommerceServer55
- ▶ **Windows** C:\Program Files\WebSphere\CommerceServer55

#### *WAS\_installdir*

This is the installation directory for WebSphere Application Server. The following are the default installation directories for WebSphere Application Server on various operating systems:

- ▶ **AIX** /usr/WebSphere/AppServer
- ▶ **400** /QIBM/ProdData/WebAS5
- ▶ **Linux** /opt/WebSphere/AppServer
- ▶ **Solaris** /opt/WebSphere/AppServer
- ▶ **Windows** C:\Program Files\WebSphere\AppServer

#### *WCDE\_installdir*

This is the installation directory for either the WebSphere Commerce Studio, or the WebSphere Commerce - Express development environment.

The default installation directory for WebSphere Commerce Studio is C:\WebSphere\CommerceStudio55.

The default installation directory for the WebSphere Commerce - Express development environment is C:\WebSphere\CommerceDev55.

#### *hostname*

In a non-federated, non-clustered environment, *hostname* is used as the node name in WebSphere Application Server. For example, *WAS\_installdir/installedApps/hostname/Enterprise\_app\_name.ear*. If you run in a federated or clustered environment, replace *hostname* with the node name on your system.

---

## Knowledge requirements

To customize the WebSphere Commerce Accelerator, you require knowledge of the following:

- HTML, JavaScript™, and XML
- Structured Query Language (SQL)
- Java™ Programming
- JavaServer Pages technology



- WebSphere Commerce Studio or WebSphere Commerce - Express Developer Edition

Please refer to the *WebSphere Commerce Programming Guide and Tutorials* for more information on customizing WebSphere Commerce. This book is available from the following Web site:

[www.ibm.com/software/commerce/library](http://www.ibm.com/software/commerce/library)



---

## Part 1. Customizing the WebSphere Commerce Accelerator and other Tools User Interface Centers reference

This document discusses how to customize Tools User Interface Center components, such as the WebSphere Commerce Accelerator. For simplicity, the customization steps documented use WebSphere Commerce Accelerator as an example, but the steps can be applied to any of the WebSphere Commerce user interface tools, such as the WebSphere Commerce Administration Console, and custom user interfaces. This document should only be read after you have completed reading the primary *WebSphere Commerce Accelerator Customization Guide*, as it builds upon information and concepts contained in that document.

### Attention

To protect your customized data from being overwritten during migration to a future version, or during installation of a future fix pack, or some similar event, it must be created in a safe place, separate from the WebSphere Commerce assets. See the Customization Guidelines section of the *WebSphere Commerce Accelerator Customization Guide* for more information.



---

## Chapter 1. Wizards

A wizard consists of panels in which users can enter and manipulate data. The panels are presented in a specific sequence defined when you create the wizard. This sequence displays in a table of contents frame. The user navigates using only the **Previous** and **Next** buttons and cannot select elements in the table of contents to move to a specific panel. Wizards are useful when you want the user to view and enter data into every panel. You can specify default data for the panels so that default values are pre-populated for the user. For example, you can pre-populate the year field with the current year. A **Finish** button may be placed on a panel before the last panel in the sequence. Once a user completes a wizard, if it is no longer necessary to follow the panels in the specific sequence, information entered through that wizard can be modified using notebooks or dialogs.

---

### Overview

The following is an overview of how to create a wizard. Detailed steps follow this section.

1. Create a wizard definition XML file that describes the flow among panels and data bean usage.
2. Register the wizard definition XML file in the component's specific resources.xml file.
3. Write panels using JSP and JavaScript files.
4. Write custom commands to update the database and perform custom functionality.
5. Register custom commands and JSP files in the database.
6. Create a resource bundle.
7. (Optional) Write context sensitive help files for your container element and panels, and update the *Tools User Interface Center* help map XML file to include your help files.
8. Add your new wizard to a *Tools User Interface Center* menu system (for example, WebSphere Commerce Accelerator).
9. Launch and test your wizard.

The following files are created:

- a wizard definition XML file that describes the panel flow, for example, *newWizard.xml*
- resource bundle files, for example, *MySampleResource\_locale.properties*
- HTML help files
- custom Java command files, for example, *MyCommand.java*
- JSP and JavaScript files to fill panel contents, for example, *myPanel.jsp* and *myPanel.js*

The following files are modified:

- resources.xml
- *Tools User Interface Center* menu XML file to display your new wizard
- *Tools User Interface Center* Help Map XML file to include your help files

## Detailed steps

The following steps are detailed instructions for implementing a wizard.

1. Create a wizard definition XML file to describe panel flow and data bean usage, called *newWizard.xml*, for example. Create this file in the directory */WC\_installdir/xml/tools/component*, where *component* is the name of the component to which the wizard belongs. The following tags are available for use in the wizard XML file:

XML Tag	Description
<code>&lt;wizard&gt; &lt;/wizard&gt;</code>	<p>The primary element defining a wizard. The following attributes are supported:</p> <p><b>resourceBundle</b> A required attribute that specifies which resource bundle is used. For example, <code>resourceBundle="common.userNLS"</code></p> <p><b>windowTitle</b> An optional attribute that defines the window title, this name is a key in the resource bundle file. For example, <code>windowTitle="WizardTitle"</code></p> <p><b>finishConfirmation</b> An optional attribute that names the finish confirmation, this name is a key in the resource bundle file. If null, a default message is used. For example, <code>finishConfirmation = "finishConfirmation"</code></p> <p><b>cancelConfirmation</b> An optional attribute that names the cancel confirmation, this name is a key in the resource bundle file. For example, <code>cancelConfirmation="cancelConfirmation"</code></p> <p><b>finishButtonName</b> An optional attribute that names the finish button text label. This name is a key in the resource bundle file. If null, a default message is used. For example, <code>finishButtonName = "myFinishButtonText"</code></p> <p><b>finishURL</b> An optional attribute that specifies which command executes to finish the wizard. If <code>finishURL</code> is not present, nothing happens when the user clicks the <b>OK</b> or <b>Finish</b> buttons. You can use other JavaScript functions to exit the wizard, such as <code>TOP.goBack()</code> or set this value at runtime if it is not known which controller command should be called. For example, <code>finishURL="WizardTestCmd"</code></p> <p><b>tocBackgroundImage</b> An optional attribute that specifies the URL for the background image of the table of contents frame. For example, <code>tocBackgroundImage="/wcs/images/tools/uiproperties/back.jpg"</code></p>

XML Tag	Description
<panel />	<p>Defines a panel to appear in the wizard's content frame. The following attributes are supported:</p> <p><b>name</b> A required attribute that specifies a name for the panel. This attribute is also a key in the resource bundle file, its value is used as the panel display name in the table of contents frame. For example, name="Profile3"</p> <p><b>url</b> A required attribute that sets the contents of the panel to this URL. This can link to a viewCommand or be a direct link. For example, url="/webapp/wcs/tools/servlet/myPanelView"</p> <p><b>helpKey</b> An optional attribute that defines the corresponding help key in the <i>Tools User Interface Center Help Map</i> file. For example, helpKey="MC.auction.auctionWizardPricePanel.Help"</p> <p><b>parameters</b> An optional attribute that specifies parameters to be passed into the contents panel from the parent frame (also known as outer frame, or WizardView), delimited by commas. For example, parameters="param1,param2"</p> <p><b>passAllParameters</b> An optional attribute that, when true, indicates that all of the parameters should be passed to the parent frame (also known as outer frame, or WizardView). If the parameters attribute is specified, then this attribute is ignored. For example, passAllParameters="true"</p> <p><b>hasFinish</b> An optional attribute that specifies whether the panel provides a finish button. This value can be either YES or NO. The default is NO. For example, hasFinish="YES"</p> <p><b>hasCancel</b> An optional attribute that specifies whether the panel provides a cancel button. This value can be either YES or NO. The default is YES. For example, hasCancel="NO"</p> <p><b>hasNext</b> An optional attribute that specifies whether the panel provides a next button. This value can be either YES or NO. The default is YES. For example, hasNext="NO"</p> <p><b>hasTab</b> An optional attribute that specifies whether the panel's name displays in the table of contents frame. This value can be either YES or NO. The default is YES. For example, hasTab="NO"</p> <p><b>hasBranch</b> An optional attribute that specifies whether the panel has branches. This value can be either YES or NO. The default is NO. For example, hasBranch="YES"</p>

XML Tag	Description
<databean />	<p>An optional element that specifies a data bean to hold user data and populate the fields with existing data. If defined, this bean is instantiated. If the bean is a smartDataBean, it is also activated when the wizard loads. Its properties are converted into a JavaScript object with the name defined here. The following attributes are supported:</p> <p><b>name</b> An required attribute that defines a name for the JavaScript object which is populated from the data bean. For example, name="campaign"</p> <p><b>class</b> A required attribute that specifies the class of the data bean. For example, class="com.ibm.commerce.tools.campaigns.CampaignDataBean"</p> <p><b>stoplevel</b> An optional attribute that specifies how many levels up the class hierachy tree should the bean properties be populated. By default, its value is 1. For example, stoplevel="2"</p>
<jsFile/>	<p>Specifies a JavaScript file to be included in the wizard. Files defined here are included in the parent frame. Thus, access to these functions require parent. prefixed to function calls to scope them to the parent frame. Multiple JavaScript files are allowed.The following attribute is supported:</p> <p><b>src</b> A required attribute that specifies the location of the JavaScript file. For example, src="/wcs/javascript/tools/common/DateUtil.js"</p>
<button> </button>	<p>This element defines buttons on the navigation bar. The following attributes are supported:</p> <p><b>name</b> A required attribute that names the button. The value specified here does not appear on the button, but is a key to the resource bundle. The corresponding value in the resource bundle displays on the button. For example name="sampleButtonName1"</p> <p><b>component</b> An optional attribute that sets the component, which is defined in the <i>instancename.xml</i> file.. For example, component="sampleComponent1"</p> <p><b>action</b> A required attribute that specifies the JavaScript that runs when this button is clicked. For example, action="sampleButtonAction1()"</p> <p>In this &lt;button&gt; example, a button is added with the name specified in a resource bundle which is also the key sampleButtonName1. When clicked, it calls the sampleButtonAction1() JavaScript function which is located in the included JavaScript file (The jsFile attribute in the notebook definition XML file). The component attribute determines whether the button will be displayed or not.</p>

The following is a sample wizard definition XML file:

```

<!DOCTYPE wizard SYSTEM "WizardPanels.dtd">
<?xml version="1.0" encoding="UTF-8"?>

<wizard resourceBundle="common.userNLS"
  windowTitle="WizardTitle"
  finishConfirmation="finishConfirmation"
  cancelConfirmation="cancelConfirmation"
  finishURL="WizardTestCmd"

```



```

tocBackgroundImage="/wcs/images/tools/uiproperties/wiz_back.jpg">

<panel name="Profile1"
  url="/webapp/wcs/tools/servlet/WizardTestPanel1"
  helpKey="" />
<panel name="Profile2"
  url="/webapp/wcs/tools/servlet/WizardTestPanel2"
  helpKey="" />
<panel name="Profile3"
  url="/webapp/wcs/tools/servlet/WizardTestPanel3"
  helpKey="" />
<panel name="Address"
  url="/webapp/wcs/tools/servlet/WizardTestPanel4"
  hasFinish="YES"
  helpKey="" />

<jsFile src="/wcs/javascript/tools/sample/wizardTest.js" />

<dataBean name="ItemDataBean" class="com.ibm.commerce.tools.test.PropertyDataBean" />

<button name="sampleButtonName1"
  action="sampleButtonAction1()" />
<button name="sampleButtonName2"
  action="sampleButtonAction2()" />
<button name="sampleButtonName3"
  component="sampleComponent3"
  action="sampleButtonAction3()" />

</wizard>

```

2. Register the wizard definition XML file created in step 1 in the appropriate resources.xml file. Multiple versions of this file exist for each component, in the following directory: `/WC_installdir/xml/tools/component/resources.xml`. The resources.xml files are referenced in the `instancename.xml` file, and you must register any new resources.xml files in `instancename.xml`.

Make an entry similar to the following in resources.xml

```
<XML name="sampleWizard" file="component/newWizard.xml"/>
```

The name attribute becomes a key which will be used in a later step.

The following is a sample resources.xml file:

```

<?xml version="1.0" encoding="UTF-8"?>

<resourceConfig>
  <resource nameSpace="sample">

    <!-- resource bundle file mappings -->
    <resourceBundle name="reportingString"
      bundle="com.ibm.commerce.tools.reporting.properties.ReportingString" />
    <resourceBundle name="resource"
      bundle="com.ibm.commerce.tools.reporting.properties.Reporting" />

    <!-- XML file mappings -->
    <resourceXML name="sampleWizard"
      file="reporting/newWizard.xml" />
    <resourceXML name="OfflineReportWizard"
      file="reporting/OfflineReportWizard.xml" />
    <resourceXML name="ReportContentWizard"
      file="reporting/ReportContentWizard.xml" />
    <resourceXML name="ReportStoreOverViewWizard"
      file="reporting/ReportStoreOverViewWizard.xml" />

  </resource>
</resourceConfig>

```

This sample file belongs in the `/WC_installdir/xml/tools/reporting` directory under WebSphere Commerce, and assumes that `newWizard.xml` is in the same directory.

- Write JSP files for each panel, and JavaScript files. These JSP files define the panels that display in the content frame into which users enter data. Any panels you create must include the following JavaScript functions:

Function Name	Description
savePanelData()	Stores data from the HTML form into the object model in the parent frame.
validatePanelData()	Validates data entered by user.

- Write custom commands. These commands update the database with the information entered, or perform some function when the user clicks on the **Finish** button. Also, if you chose your panel URL (in your wizard XML file) to be a viewCommand, you must write a custom viewCommand.

The following is a sample controller command for a wizard:

```

package com.ibm.commerce.tools.test;
import java.util.*;
import com.ibm.commerce.ras.*;
import com.ibm.commerce.server.*;
import com.ibm.commerce.command.*;
import com.ibm.commerce.exception.*;
import com.ibm.commerce.datatype.*;
import com.ibm.commerce.tools.common.*;
import com.ibm.commerce.tools.command.*;
import com.ibm.commerce.tools.common.ui.*;
import com.ibm.commerce.exception.*;
import com.ibm.commerce.tools.resourcebundle.*;
import com.ibm.commerce.tools.util.*;

public class TestCmdImpl extends ToolsControllerCommandImpl implements TestCmd {
protected ResourceBundleProperties resourceBundle = null;
protected String viewname = null;
protected String successMsg = "Success";
protected String errorMsg = "Error";

// sample input data
protected String name = null;
protected int age;
protected float salary;
protected String department = null;
public void performExecute() {

/*
* your business logic here
*/

// exit successfully - forwarding to returning view command
responseProperties = new TypedProperty();
responseProperties.put(EConstants.EC_VIEWTASKNAME, viewname);
responseProperties.put(UIProperties.SUBMIT_FINISH_MESSAGE, successMsg);
}

public void validateParameters() throws EException {

String methodName = "validateParameters";

/* optionally, load your resource bundle file and NL messages
Locale locale = commandContext.getLocale();
resourceBundle =
(ResourceBundleProperties) ResourceDirectory.lookup("samples.samplesNLS", locale);
successMsg = (String) resourceBundle.get("successMsg");
errorMsg = (String) resourceBundle.get("errorMsg");
*/

// retrieve the data which was set on the client using parent.put
name = (String) requestProperties.getString("name", null);
age = requestProperties.getIntValue("age", 0);
salary = requestProperties.getFloatValue("salary", 0);
department = (String) requestProperties.getString("department", null);
}
}

```

```

// set returning URL view name
viewname = requestProperties.getString(ECConstants.EC_REDIRECTURL);

/* error use case - returning error code and
message to view command via EException
if (salary < 10000.00 || salary > 99999.99) {
responseProperties = new TypedProperty();
responseProperties.put(UIProperties.SUBMIT_ERROR_STATUS, "101");
responseProperties.put(UIProperties.SUBMIT_ERROR_MESSAGE, errorMsg);
throw new ECAApplicationException(
ECToolsMessage.TOOLS_TEST_USER_ERROR,
this.getClass().getName(),
methodName,
null,
viewname,
responseProperties);
}
*/
}
}

```

For further information on writing Commands, refer to the *WebSphere Commerce Programming Guide and Tutorials*.

5. Register your custom commands and JSP files. Once any required commands are created, register them in the database. Refer to the "Design Patterns" section of the *WebSphere Commerce Programming Guide and Tutorials* for details on how to register the command.

**Note:** Ensure that the URL field in the database matches the value of the finishURL attribute in your XML file.


6. Create a resource bundle for the wizard. The text in the resource bundle appears in the wizard, for example, in the table of contents frame, and displays in as the title of each panel. Resource bundles are in the

```

/WAS_installdir/installedApps/hostname/Enterprise_App_name.ear/
properties/com/ibm/commerce/tools/component_name/properties

```

directory.

 For WebSphere Commerce Developer, the resource bundles are in the *WCDE\_installdir/properties/com/ibm/commerce/tools/component\_name/properties* directory.

The following is a sample resource bundle based on the data from the wizard XML file in step 1.

```

# Panels name for TOC panel
Profile1=General
Profile2=Description
Profile3=Attributes
Address=Address
# Button Labels
sampleButtonName1=Test Button 1
sampleButtonName2=Test Button 2
sampleButtonName3=Test Button 3

```

If national languages are supported, create the national language resource bundles with the appropriate language text. The national language file names must end with the locale supported. For example, for a French-language resource bundle, the file name should be *filename\_fr\_FR.properties*.

7. Write context sensitive help files and update the *Tools User Interface Center Help Map XML* file. See the topic "Adding context sensitive help in chapter 7 for more information.

8. Add your new wizard to a *Tools User Interface Center* menu system (for example, WebSphere Commerce Accelerator.) See the topic "Integrating tools into the Tools User Interface Center" in chapter 7 for more information.
9. Stop and start your WebSphere Commerce server, then test the wizard by launching the menu registered in previous step. Although not preferred, if it is necessary to launch the wizard outside of the *tools user interface center*, the URL is :

`https://hostname:8000/webapp/wcs/tools/servlet/WizardView?XMLFile=sample.sampleWizard`

, where `sample` is a namespace defined in `resources.xml` in step 2. For more information on namespaces, see the *WebSphere Commerce Accelerator Customization Guide*.

---

## Navigation

Wizards present users with multiple navigation options. Each panel in a wizard can have buttons for next, previous, finish, or cancel. The names here are the default values but represent generic functions. These options are presented as buttons in the navigation frame at the bottom of the content window. If included, they behave according to the following guidelines:

**Next** A user clicks **Next** in the navigation panel:

1. If a panel is still loading, do nothing while it finishes loading. Otherwise, continue to the next step.
2. The `savePanelData()` function runs on the current panel to save its data into the object model. The data is saved so that if the page re-displays, for example, if some data on the page is not valid, the original data can be pre-populated onto the page. This allows the user to re-enter only the corrected data instead of all of it.
3. The `validatePanelData()` function runs on the current panel to ensure that the user has entered valid data. If there is a problem with this validation, this function returns false and remains on the current panel. You should help the user enter correct data with a meaningful error message.
4. Display the new panel. When the panel finishes loading, it must indicate this by calling the `parent.setContentFrameLoaded(true);` function. This stops the progress indicator icon in the tools UI center, and enables the **Finish** or **Next** buttons, thus allowing other panels to be selected.

**Previous**

A user clicks **Previous** in the navigation panel:

1. If a panel is still loading, do nothing while it finishes loading. Otherwise, continue to the next step.
2. The `savePanelData()` function runs on the current panel to save its data into the object model. The data is saved so that if, for example, if some data on the panel is not valid, the original data can be re-populated onto the panel. This allows the user to re-enter only the corrected data instead of all of it.
3. Display the new panel. When the panel finishes loading, it must indicate this by calling the `parent.setContentFrameLoaded(true);` function. This stops the progress indicator icon in the *Tools User Interface Center*, and enables the **Finish** or **Next** buttons, thus allowing other panels to be selected.

**Finish** A user clicks **Finish** to finalize the interaction with the wizard:

1. If a panel is still loading, do nothing while it finishes loading. Otherwise, continue to the next step.
2. Check to see if **Finish** has already been clicked. If so, return and do nothing. If not, set a flag to indicate that **Finish** has been clicked.
3. The `savePanelData()` function runs on the current panel to save its data into the object model. The data is saved so that if the page re-displays, for example, if some data on the page is not valid, the original data can be pre-populated onto the page. This allows the user to re-enter only the corrected data instead of all of it.
4. The `validatePanelData()` function runs on the current panel to ensure that the user has entered valid data. If there is a problem with this validation, this function returns false and remains on the current panel. You should help the user enter correct data with a meaningful error message.
5. The `preSubmitHandler()` runs to invoke any optional JavaScript functions that must run before the finish URL.
6. If a finish command is specified, convert the object model to XML and send it as a parameter to the finish command.
7. Set a flag to indicate that the finish command has completed.
8. If an error occurs in the finish server command (performing an update or otherwise), the command redirects back to the navigation frame passing back the following two parameters:

**SubmitErrorMessage**

The error message to display to the user. If national language-enabled, the translated message displays.

**SubmitErrorStatus**

The corresponding error status or error code.

If the `SubmitErrorStatus` variable is passed in, then `submitErrorHandler()` function is called with the above two variables as parameters. You are responsible for implementing this function in the JavaScript file so that it displays the error message to the user, and redirects to the problematic input field based on the error code. If this function is not defined, a default alert window displays the error message.

If the `SubmitErrorStatus` is not passed in, then it is assumed that the finish command succeeded.

9. Call the `submitFinishHandler()` function with the `submitFinishMessage` as a parameter, which was set in the controller command.

**Cancel**

Calls the `submitCancelHandler()` function and then the parent frame's cancel method.

---

## Customizations

To use the following JavaScript functions, first write code for the functions that implements your business logic. Once the code is written you can use these JavaScript functions in your wizard. Your JavaScript file is specified in the wizard definition XML file.

Function Name	Description
submitErrorHandler(errMessage, errorStatus)	Called when an error is received from the controller command.
submitFinishHandler(finishMessage)	Called upon successful completion by the controller command.
submitCancelHandler()	Called when the user clicks <b>Cancel</b> , then <b>OK</b> to confirm the cancel action.
preSubmitHandler()	Optionally, called after the validateAllPanels() function, but before the finish controller command.

---

## JavaScript functions

You can use the following JavaScript functions in your wizard, they are implemented by WebSphere Commerce by default. These functions are defined in the parent frame, and are called using `parent.functionName()`:

Function Name	Description
<code>get(key, defaultValue)</code>	Returns the value of the specified key from the object model. These keys are the parameters entered into the parent frame. The panel frame uses this function to get the value of these parameters from the parent frame.
<code>remove(key)</code>	Removes the specified key from the object model.
<code>put(key, value)</code>	Stores the value for the specified key in the object model. These keys are the parameters entered into the panel frame. The panel frame uses this function to put the value of these parameters into the parent frame. All JavaScript objects that are stored in the object model using this function, are submitted automatically to the finish controller command and can be retrieved their values from the requestProperties object.
<code>setContentFrameLoaded(value)</code>	Sets the contentFrameLoaded variable to either true or false. If true, then the user is allowed to switch to a different content panel. If false, the user must wait until the panel has been loaded which sets the value to true. <b>Note:</b> You must call setContentFrameLoaded(true) at the end of the panel body's onLoad() function. If this is not called, users will remain on the current content panel.
<code>getRequestProperties()</code>	Gets the Request Properties. All parameters sent back to the navigation frame (provided they are strings) are put into a request property JavaScript hashtable. These parameters can be accessed by calling this JavaScript function.
<code>addURLParameter(pname, value)</code>	Adds an additional URL parameter to finish command.
<code>removeURLParameter(pname)</code>	Removes the URL parameter, previously added using the addURLParameter() function.

---

## Chapter 2. Notebooks

The notebook element enables you to easily organize and develop groups of panels for updating and creating information in the database. Notebooks are similar to wizards because there are many panels of information, organized in a table of contents. Notebooks differ from wizards because the panels do not display in a pre-defined sequence. The user can select and view any panel from the table of contents at any time. Notebooks are used for sets of information that are not necessarily sequential. The notebook is also used to update information created in a wizard, since users may want to skip directly to the panel containing the information they wish to change. This element is not appropriate for situations that require branching in the task flow because of its parallel, non-sequential nature. For example, a notebook is not appropriate for a task that requires changing subsequent forms based on earlier choices.

---

### Overview

The following is an overview of how to create a notebook. Detailed steps follow this section.

1. Create a notebook definition XML file that describes the default flow among panels and data bean usage.
2. Register the notebook definition XML file in the component's specific resources.xml file.
3. Write the panels using JSP files and JavaScript.
4. Write custom commands to update the database and perform custom functionality.
5. Register custom commands and JSP files in the database. (See the "Design Patterns" section of the *WebSphere Commerce Programming Guide and Tutorials*.)
6. Create a resource bundle.
7. (Optional) Write context sensitive help files for your container element and panels and update the *Tools User Interface Center* Help Map XML file to include your help files.
8. Add your new notebook to a *Tools User Interface Center* menu system (for example, *WebSphere Commerce Accelerator*).
9. Launch and test your notebook.

The following files are created:

- a notebook definition XML file, *newNotebook.xml*, to describe the notebook flow
- resource bundle files, for example, *mySampleResource\_locale.properties*
- HTML help files for your users
- custom Java command files, for example, *MyCommand.java*
- JSP and JavaScript files to fill panel contents, for example, *MyPanel.jsp* and *myPanel.js*

The following files are modified:

- resources.xml
- *Tools User Interface Center* menu XML file to display your new notebook
- *Tools User Interface Center* Help Map.xml to include your help files

## Detailed steps

The following steps are detailed instructions for implementing a notebook.

1. Create a notebook definition XML file to describe the default panel flow and data bean usage called *newNotebook.xml*, for example. Create this file in */WC\_installdir/xml/tools/component*, where *component* is the name of the component to which the notebook belongs. The following tags are available for use in the notebook definition XML file:

XML Tag	Description
<notebook> </notebook>	<p>The primary element defining a notebook. The following attributes are supported:</p> <p><b>resourceBundle</b> A required attribute which specifies the resource bundle to use. For example, resourceBundle="common.userNLS"</p> <p><b>windowTitle</b> An optional attribute that defines the window title, this name is a key in the resource bundle file. For example, windowTitle="NotebookTitle"</p> <p><b>finishConfirmation</b> An optional attribute that names the finish confirmation, this name is a key in the resource bundle file. If null, a default message will be used. For example, finishConfirmation="finishConfirmation"</p> <p><b>cancelConfirmation</b> An optional attribute that names the cancel confirmation button, this name is a key in the resource bundle file. For example, cancelConfirmation="cancelConfirmation"</p> <p><b>finishButtonName</b> An optional attribute that defines the finish button text label, this name is a key in the resource bundle file. If null, a default message will be used. For example, finishButtonName="myFinishButtonText"</p> <p><b>finishURL</b> An optional attribute that specifies which command is to be executed to finish the notebook. If it is not present then nothing will happen when the user clicks <b>OK</b> or <b>Finish</b>. You can use other JavaScript functions to exit the notebook, such as TOP.goBack() or set this value at runtime if it is not known which controller command should be called. For example, finishURL="NotebookFinishCmd"</p>



XML Tag	Description
<panel />	<p>Defines a panel to appear in the content frame. The following attributes are supported:</p> <p><b>name</b> A required attribute that specifies a name for the panel. For example, name="Profile3"</p> <p><b>url</b> A required attribute that sets the contents of the panel to this URL. This can link to a viewCommand or be a direct link. For example, url="/webapp/wcs/tools/servlet/myPanelView"</p> <p><b>helpKey</b> An optional attribute that defines the corresponding help key in the <i>Tools User Interface Center</i> Help Map file. For example, helpKey="MC.auction.InitiativeGeneral.Help"</p> <p><b>parameters</b> An optional parameter that specifies parameters to be passed into the content panel from the parent frame (also called the outer frame or NotebookView), delimited by commas. For example, parameters="param1,param2"</p> <p><b>passAllParameters</b> An optional attribute that, when true, indicates that all of the parameters should be passed to this panel from the parent frame (also called the outer frame or NotebookView). If the parameters attribute is specified, then this attribute is ignored. passAllParameters="true"</p> <p><b>hasTab</b> An optional attribute that specifies whether the panel's name displays in the table of contents frame. This value can be either YES or NO. The default is YES. For example, hasTab="NO"</p> <p><b>group</b> An optional attribute which specifies a group for this panel. Groups act to organize the table of contents frame. All panels assigned to the same group display under a subheading with the group's name. This is useful if the number of total panels is large and they do not fit in the table of contents frame without scrolling. For example, group="group1"</p>

XML Tag	Description
<databean />	<p>An optional element that specifies a data bean to hold user data and populate the fields with existing data. If defined, this bean is instantiated. If the bean is a smartDataBean, it is also activated when the notebook loads. It's properties are converted into a JavaScript object with the name defined here. The following attributes are supported:</p> <p><b>name</b> A required attribute that defines a name for the JavaScript object which is populated from the data bean. For example, name="campaign"</p> <p><b>class</b> A required attribute that specifies the class of the data bean. For example, class="com.ibm.commerce.tools.campaigns.CampaignDataBean"</p> <p><b>stoplevel</b> An optional attribute that specifies how many levels up the class hierarchy tree should the bean properties be populated. By default, its value is 1. For example, stoplevel="2"</p>
<jsFile />	<p>Specifies a JavaScript file to be included in the notebook. Files defined here are included in the parent frame. Thus, access to these functions require parent. prefixed to function calls to scope them to the parent frame. Multiple JavaScript files are allowed. The following attribute is supported:</p> <p><b>src</b> A required attribute that specifies the location of the JavaScript file. For example, src="/wcs/javascript/tools/common/DateUtil.js"</p>
<button> </button>	<p>This element defines custom buttons for the navigation bar. The following attributes are supported:</p> <p><b>name</b> A required attribute that names the button. For example, name="sampleButtonName1"</p> <p><b>component</b> An optional attribute which sets the component, which is defined in the <i>instanceName.xml</i> file. For example, component="sampleComponent1"</p> <p><b>action</b> A required attribute which specifies the action to be taken when this button is clicked. For example, action="sampleButtonAction1()"</p> <p>In this &lt;button&gt; example, a button is added with the name specified in a resource bundle by key sampleButtonName1, and when clicked, it calls the sampleButtonAction1() JavaScript function which is located in the included JavaScript file (the jsFile attribute in the notebook XML). The component attribute determines whether the button is displayed or not.</p>

The following is a sample notebook definition XML file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<notebook resourceBundle="campaigns.campaignsRB"
windowTitle="initiativeNotebookTitle"
```

```

        finishConfirmation=""
        cancelConfirmation="initiativeNotebookCancelConfirmation"
        finishURL="CampaignInitiativeSave" >
<panel name="initiativeGeneralPanel"
    url="CampaignInitiativeGeneralPanelView"
    helpKey="MC.campaigns.InitiativeGeneral.Help" />
<panel name="initiativeLocationPanel"
    url="CampaignInitiativeLocationPanelView"
    helpKey="MC.campaigns.InitiativeLocation.Help" />
<panel name="initiativeConditionsPanel"
    url="CampaignInitiativeConditionsPanelView?ActionXMLFile=campaigns.ConditionList
    &cmd=CampaignInitiativeConditionListView&selected=SELECTED&listsize=20
    &startindex=0&refnum=0" helpKey="MC.campaigns.InitiativeConditions.Help" />
<dataBean name="initiative"
    class="com.ibm.commerce.tools.campaigns.CampaignInitiativeDetailsDataBean" />
<jsFile src="/wcs/javascript/tools/common/Util.js" />
<jsFile src="/wcs/javascript/tools/common/DateUtil.js" />
<jsFile src="/wcs/javascript/tools/campaigns/Initiative.js" />
</notebook>

```

2. Register the XML file created in step 1 in the resources.xml file for the component that you are modifying. Multiple versions of this file exist, one for each component, in the following directory:

*/WC\_installdir/xml/tools/component/resources.xml*.

You need to make an entry similar to the following in resources.xml

```
<XML name="initiativeNotebook" file="component/newNotebook.xml" />
```

The name attribute becomes a key which will be used in a later step.

The resources.xml files are referenced in the *instancename.xml* file, and you must register any new resources.xml files in *instancename.xml*.

3. Write panel JSP files for each panel, and JavaScript files. These JSP files define the panels that display in the content frame into which users enters data. The following table describes the JavaScript functions for these panels:

Function Name	Description
savePanelData()	(Required) Stores HTML form data into model (parent frame). This function savePanelData() must be supplied in each content frame to save the current panel's data in the model.
validateNotebookPanel()	(Optional) Validates data entered by user. This method should return true or false.

At the end of the onLoad event, parent.setContentFrameLoaded(true) must be called. This ensures that the entire content panel loads before the user can move to another panel.

4. Write custom JSP commands. These commands update the database with the information entered, or perform some function when the user clicks **OK**. Also, if you chose your panel URL, in your notebook XML file, to be a viewCommand, you must write a custom viewCommand.

The following is a sample controller command for a Notebook:

```

package com.ibm.commerce.tools.test;
import java.util.*;
import com.ibm.commerce.ras.*;
import com.ibm.commerce.server.*;
import com.ibm.commerce.command.*;
import com.ibm.commerce.exception.*;
import com.ibm.commerce.datatype.*;
import com.ibm.commerce.tools.common.*;
import com.ibm.commerce.tools.command.*;
import com.ibm.commerce.tools.common.ui.*;
import com.ibm.commerce.exception.*;

```

```

import com.ibm.commerce.tools.resourcebundle.*;
import com.ibm.commerce.tools.util.*;

public class TestCmdImpl extends ToolsControllerCommandImpl implements TestCmd {
protected ResourceBundleProperties resourceBundle = null;
protected String viewname = null;
protected String successMsg = "Success";
protected String errorMsg = "Error";

// sample input data
protected String name = null;
protected int age;
protected float salary;
protected String department = null;
public void performExecute() {

/*
* your business logic here
*/

// exit successfully - forwarding to returning view command
responseProperties = new TypedProperty();
responseProperties.put(EConstants.EC_VIEWTASKNAME, viewname);
responseProperties.put(UIProperties.SUBMIT_FINISH_MESSAGE, successMsg);
}

public void validateParameters() throws EException {

String methodName = "validateParameters";

/* optionally, load your resource bundle file and NL messages
Locale locale = commandContext.getLocale();
resourceBundle =
(ResourceBundleProperties) ResourceDirectory.lookup
("samples.samplesNLS", locale);
successMsg = (String) resourceBundle.get("successMsg");
errorMsg = (String) resourceBundle.get("errorMsg");
*/

// retrieve the data which was set on the client using parent.put
name = (String) requestProperties.getString("name", null);
age = requestProperties.getIntValue("age", 0);
salary = requestProperties.getFloatValue("salary", 0);
department = (String) requestProperties.getString("department", null);

// set returning URL view name
viewname = requestProperties.getString(EConstants.EC_REDIRECTURL);

/* error use case - returning error code and message to
view command via EException
if (salary < 10000.00 || salary > 99999.99) {
responseProperties = new TypedProperty();
responseProperties.put(UIProperties.SUBMIT_ERROR_STATUS, "101");
responseProperties.put(UIProperties.SUBMIT_ERROR_MESSAGE, errorMsg);
throw new ECAApplicationException(
EToolsMessage.TOOLS_TEST_USER_ERROR,
this.getClass().getName(),
methodName,
null,
viewname,
responseProperties);
}
*/

}
}

```

5. Register custom commands and JSP pages. Once the command is created, register it in the database. See the "Design Patterns" section of the *WebSphere Commerce Programming Guide and Tutorials* for details on how to register the command.

**Note:** Ensure that the URL field in the database matches the value of the `finishURL` attribute in your XML file.

6. Create a resource bundle with text for the notebook. This text appears, for example, on the notebook map in the table of contents frame. Resource bundles are in the `/WAS_installdir/installedApps/hostname/Enterprise_App_name.ear/properties/com/ibm/commerce/tools/component_name/properties` directory.

**Developer** For WebSphere Commerce Developer, the resource bundles are in the `/WCDE_installdir/properties/com/ibm/commerce/tools/component_name/properties` directory. In this case, keys are based on the panel names given in your notebook definition XML file created in step 1.

The following is a sample resource bundle for the XML file defined in step 1.

```
# Panel names for TOC panel
initiativeGeneralPanel=General
initiativeLocationPanel=Location
initiativeConditionsPanel=Conditions
```

If national languages are supported, create the national language resource bundles with the appropriate language text. The national language file names must end with the locale supported. For example, for a French-language resource bundle, the file name should be `filename_fr_FR.properties`.

7. Add your new notebook to a *Tools User Interface Center* menu system. See the topic "Integrating tools into the Tools User Interface Center" in chapter 7 for more information.
8. Write context sensitive help files and update the *Tools User Interface Center Help Map XML* file. See the topic "Adding context sensitive help" in chapter 7 for more information.
9. Stop and start your WebSphere Commerce server. To test your new notebook, use this URL:

```
https://hostname:8000/webapp/commerce/tools/servlet/
NotebookView?XMLFile=namespace.myTestNotebook
```

, where `namespace` is a namespace defined in `resources.xml`

---

## Navigation

Notebooks present users with one or more navigation options: finish and cancel, which are presented as buttons in the navigation frame at the bottom of the content window. Notebooks also provide panel navigation using the table of contents frame. When included, the navigation controls behave according to the following guidelines:

### Panel Change

A user clicks on a panel in the table of contents frame other than the current one:

1. If a panel is still loading, do nothing while it finishes loading. Otherwise, continue to the next step.
2. The `savePanelData()` function runs on the current panel to save its data into the object model. The data is saved so that if, for example, if some

data on the page is not valid, the original data can be pre-populated onto the page. This allows the user to re-enter only the corrected data instead of all of it.

3. The `validateNotebookPanel()` function runs if it is defined. This ensures the that user has entered valid data. If there is a problem with this validation, this function returns false and remains on the current panel. You should help the user enter correct data with a meaningful error message.
4. Display the new panel. When the panel finishes loading, it must indicate this by calling the `parent.setContentFrameLoaded(true);` function. This stops the progress indicator in the *Tools User Interface Center*, and enables the **Finish**, **OK**, or **Next** buttons, thus allowing other panels to be selected.

**Finish** A user clicks **Finish** or **OK** to finalize the interaction with the notebook:

1. Check to see if the table of contents frame has finished loading. If not, return immediately and do nothing. Otherwise, continue to the next step.
2. Check to see if **Finish** has already been clicked. If so, return and do nothing. If not, set a flag to indicate that **Finish** has been clicked.
3. Call the `savePanelData()` function on the current panel to save its data into the object model. The data is saved so that if the page re-displays, for example, if some data on the page is not valid, the original data can be pre-populated onto the page. This allows the user to re-enter only the corrected data instead of all of it.
4. Call the `validateAllPanels()` function to perform validation on each panel, assuring that mandatory fields are completed, dates are in the correct form, and so on. If there are validation errors, you can set the content frame to the page containing errors. This function should be defined in the included JavaScript file, and because it doesn't belong to any individual panel, it is in the outer frame scope."
5. Call the `preSubmitHandler()`. This function allows you to call any Java that must run before the finish URL runs.
6. If a finish command is specified, convert the model to XML and send it as a parameter to the finish command.
7. Set a flag to indicate that the finish command is finished executing.
8. If an error occurs in the finish server command (performing an update or otherwise), the command redirects back to the navigation frame passing back the following two parameters:

**SubmitErrorMessage**

The translated error message to display to the user.

**SubmitErrorStatus**

The corresponding error status or error code.

If the `SubmitErrorStatus` variable is passed in, then `submitErrorHandler()` function is called with the above two variables as parameters. You are responsible for implementing this function in the JavaScript file so that it displays the error message to the user, and redirects to the problematic input field based on the error code. If this function is not defined, a default alert window displays the error message.

If the `SubmitErrorStatus` is not passed in, then it is assumed that the finish command succeeded.

- Call the `submitFinishHandler()` function with the `submitFinishMessage` as a parameter, which was set in the controller command.

**Cancel**

Displays a cancel confirmation dialog. If the user clicks **OK**, `submitCancelHandler()` then the parent frame's cancel method run.

## Customizations

To use the following JavaScript functions, first write code for the functions that implements your business logic. Once the code is written you can use these JavaScript functions in your notebook. Your JavaScript file is specified in the notebook's XML file.

Function Name	Description
<code>submitErrorHandler(errMessage, errorStatus)</code>	Called when an error is received from the controller command.
<code>submitFinishHandler(finishMessage)</code>	Called upon successful completion by the controller command.
<code>submitCancelHandler()</code>	Called when <b>Cancel</b> is clicked and user clicks <b>OK</b> in cancel confirmation dialog.
<code>preSubmitHandler()</code>	Optionally called after the <code>validateAllPanels()</code> function, but before the finish controller command.
<code>validateAllPanels()</code>	Called when the user clicks <b>OK</b> . Parses the entire model, validating the information inside of it. If data is found to be invalid, the <code>gotoPanel()</code> function must be used to bring the user to the panel which contains the error. An error code is passed to this function identifying the error message to display. From the panel, the <code>getErrorParams()</code> function should be used to determine which error code was passed in. More information on <code>gotoPanel()</code> and <code>getErrorParams()</code> follows in the Error code handling section.

## Error code handling

When validating for the notebook, you can display an error message from a resource bundle. The `gotoPanel()` function has an optional parameter: `errorCode`. The error code specified indicates what error has occurred. This error code is passed to your panel as an argument. Upon loading each notebook panel, check to see if any error codes have been passed in, using the `parent.getErrorParams()` function. If an error code has been passed in, display the appropriate error message for the error code.

**Note:** This means that each panel needs to be a JSP file so it can access the resource bundle.

---

## JavaScript functions

You can use the following JavaScript functions in your notebook, they are implemented by WebSphere Commerce by default. These functions are defined in the parent frame, and are called using `parent.functionName()`:

Function Name	Description
<code>get(key, defaultValue)</code>	Returns the value of the specified key from the object model.
<code>remove(key)</code>	Removes the specified key from the object model.
<code>put(key, value)</code>	Stores the value for the specified key in the object model.
<code>setContentFrameLoaded(value)</code>	Sets the <code>contentFrameLoaded</code> variable to either true or false. If true, then the user is allowed to switch to a different content panel. If false, the user must wait until the panel has been loaded which sets the value to true. <b>Note:</b> You must call <code>setContentFrameLoaded(true)</code> at the end of the panel's <code>onLoad()</code> function. If this is not called, users will remain on the current content panel.
<code>getRequestProperties()</code>	Gets the Request Properties. All parameters sent back to the navigation frame (provided they are strings) are put into a request property JavaScript hashtable. They can be accessed from the developer's code by calling the JavaScript function.
<code>addURLParameter(pname, value)</code>	Adds an additional URL parameter to the finish command.
<code>removeURLParameter(pname)</code>	Removes the URL parameter, previously added using the <code>addURLParameter()</code> function.
<code>gotoPanel(name)</code>	Saves the current panel information, then displays the selected panel.



---

## Chapter 3. Dialogs

Dialogs consist of a content frame and an action frame. Dialogs are useful for displaying summary information, confirmation information, or asking for simple input information. The content frame makes up the majority of the panel, with an action frame that contains an OK button and optionally a Cancel button. The name of the OK button can be changed to be more descriptive, for example to the word Find.

---

### Overview

The following is an overview for to creating a dialog. Detailed steps follow this section.

1. Create a dialog definition XML file that describes the panel, and data bean usage.
2. Register the XML file in resources.xml.
3. Write the panels using JSP files and JavaScript files.
4. Write custom commands to update the database and perform custom functionality.
5. Register custom commands and JSP files in the database. (See the "Design Patterns" section of the *WebSphere Commerce Programming Guide and Tutorials*).
6. Create a resource bundle.
7. (Optional) Write context sensitive help files for your dialog and panels and update the *Tools User Interface Center* help map XML file to include your help files.
8. Add your new dialog to a *Tools User Interface Center* menu system (for example, WebSphere Commerce Accelerator).
9. Launch and test your dialog.

The following files are created:

- a dialog definition XML file, *newDialog.xml* to describe the dialog panels
- resource bundle files, for example, *mySampleResource\_locale.properties*
- HTML help files for your users
- custom Java command files, for example, *MyCommand.java*
- JSP and JavaScript files to fill panel contents, for example, *myPanel.jsp* and *myPanel.js*

The following files are modified:

- resources.xml
- *Tools User Interface Center* menu XML file to display your new dialog
- *Tools User Interface Center* Help Map XML file to include your help files

---

### Detailed steps

The following steps are detailed instructions for implementing a dialog:

1. Create a dialog definition XML file to describe panel flow and data bean usage called *newDialog.xml*, for example. Create this file in */WC\_installdir/xml/tools/component/*, where *component* is the name of the

component to which the dialog belongs. The following tags are available for use in the dialog definition XML file:

XML Tag	Description
<p data-bbox="435 302 545 359">&lt;dialog&gt; &lt;/dialog&gt;</p>	<p data-bbox="641 302 1377 359">The primary element defining a dialog. The following attributes are supported:</p> <p data-bbox="641 373 1382 495"><b>resourceBundle</b> A required attribute that specifies which resource bundle is used. For example, resourceBundle="common.userNLS"</p> <p data-bbox="641 514 1406 636"><b>windowTitle</b> An optional attribute that defines the window title, this name is a key in the resource bundle file. For example, windowTitle="DialogTitle"</p> <p data-bbox="641 655 1398 804"><b>finishConfirmation</b> An optional attribute that names the finish confirmation, this name is a key in the resource bundle file. If null, a default message is used. For example, finishConfirmation = "finishConfirmation"</p> <p data-bbox="641 823 1406 945"><b>cancelConfirmation</b> An optional attribute that names the cancel confirmation, this name is a key in the resource bundle file. For example, cancelConfirmation="cancelConfirmation"</p> <p data-bbox="641 963 1419 1113"><b>finishButtonName</b> An optional attribute that names the finish button text label, this name is a key in the resource bundle file. If null, a default message is used. For example, finishButtonName = "myFinishButtonText"</p> <p data-bbox="641 1131 1398 1371"><b>finishURL</b> An optional attribute that specifies which command is to be executed to finish the dialog. If it is not present then nothing happens when the user clicks the <b>OK</b> or <b>Finish</b> buttons. You can use other JavaScript functions to exit the dialog, such as TOP.goBack() or set this value at runtime if it is not known which controller command should be called. For example, finishURL="DialogTestCmd"</p>

<panel />	<p>Defines a panel to appear in the content frame. The following attributes are supported:</p> <p><b>name</b> A required attribute that specifies a name for the panel. For example, name="Profile"</p> <p><b>url</b> A required attribute that sets the contents of the panel to this URL. This can link to a viewCommand or be a direct link. For example, url="/wcs/tools/sample/WizardTestPanel1.html"</p> <p><b>helpKey</b> An optional attribute that defines the corresponding help key in the <i>Tools User Interface Center</i> Help Map file. For example, helpKey="MC.auction.auctionDialogPricePanel.Help"</p> <p><b>parameters</b> An optional attribute that specifies parameters to be passed into the contents panel from the outer frame (DialogView), delimited by commas. For example, parameters="param1,param2"</p> <p><b>passAllParameters</b> An optional attribute that, when true, indicates that all of the parameters should be passed to this panel from the outer frame (DialogView). If the parameters attribute is specified, then this attribute is ignored. For example, passAllParameters="true"</p> <p><b>hasFinish</b> An optional attribute that specifies whether the panel provides a finish button. This value can be either YES or NO. The default is NO. For example, hasFinish="YES"</p> <p><b>hasCancel</b> An optional attribute that specifies whether the panel provides a cancel button. This value can be either YES or NO. The default is YES. For example, hasCancel="NO"</p>
-----------	--

<p>&lt;databean&gt; &lt;/databean&gt;</p>	<p>An optional element that specifies a data bean to hold user data and populate the fields with existing data. If defined, this bean is instantiated. If the bean is a smartDataBean, it is also activated when the dialog loads. Its properties are converted into a JavaScript object with the name defined here. The following attributes are supported:</p> <p><b>name</b> A required attribute that defines a name for the JavaScript object which is populated from the data bean. For example, name="campaign"</p> <p><b>class</b> A required attribute that specifies the class of the data bean. For example, class="com.ibm.commerce.tools.campaigns.CampaignDataBean"</p> <p><b>stoplevel</b> An optional attribute that specifies how many levels up the class hierarchy tree should the bean properties be populated. By default, its value is 1. For example, stoplevel="2"</p>
<p>&lt;jsFile/&gt;</p>	<p>Specifies a JavaScript file to be included in the dialog. Files defined here are included in the parent frame. Thus, access to these functions require parent. prefixed to function calls to scope them to the parent frame. Multiple JavaScript files are allowed. The following attribute is supported:</p> <p><b>src</b> A required attribute that specifies the location of the JavaScript file. For example, src="/wcs/javascript/tools/common/DateUtil.js"</p>
<p>&lt;button&gt; &lt;/button&gt;</p>	<p>This element defines custom buttons on the navigation bar. The following attributes are supported:</p> <p><b>name</b> A required attribute that names the button. The value specified here does not appear on the button, but is a key to the resource bundle. The corresponding value in the resource bundle displays on the button. For example, name="sampleButtonName1"</p> <p><b>component</b> An optional attribute that sets the component, which is defined in the <i>instancename.xml</i> file. For example, component="sampleComponent1"</p> <p><b>action</b> A required attribute that specifies the action to be taken when this button is clicked. For example, action="sampleButtonAction1()"</p> <p>In this example, a button is added with the name as specified in a resource bundle for the key sampleButtonName1, and when clicked, it calls the sampleButtonAction1() JavaScript function which is located in the included JavaScript file (The jsFile attribute in the panel's XML file). The component attribute determines whether the button will be displayed or not.</p>

The following is a sample dialog XML file:

```
<?xml version="1.0"?>
<dialog resourceBundle="catalog.ItemNLS"
windowTitle="itemFindCriteria_Title"
```

```

        finishConfirmation=""
        cancelConfirmation="itemFindCriteria_cancelConfirmation"
        finishURL="" >
<panel name="itemFindCriteria"
    url="ItemFindCriteria"
    helpKey="MC.catalogTool.productSearch.Help"
    passAllParameters="true"
    hasFinish="NO"
    hasCancel="NO" />
<jsFile src="/wcs/javascript/tools/catalog/itemFindCriteria.js" />
<button name="itemFindCriteria_button_find"
    action="button_Find();" />
<button name="itemFindCriteria_button_cancel"
    action="button_Cancel();" />
</dialog>

```

2. Register the dialog definition XML file created in step 1 in the resources.xml file for the component that you are modifying. Multiple versions of this file exist, one for each component, in the following directory:  
*WC\_installdir/xml/tools/component/resources.xml*.

Make an entry similar to the following in resources.xml

```
<XML name="sampleDialog" file="component/newDialog.xml"/>
```

The name attribute becomes a key which will be used in a later step. The resources.xml files are referenced in the *instancename.xml* file, and you must register any new resources.xml files in *instancename.xml*.

3. Write JSP files for each panel, and JavaScript files. These JSP files define the panels that display in the content frame into which users enter data. Any panels you create must include the following JavaScript functions:

Function Name	Description
savePanelData()	Stores data from the HTML form into the object model in the parent frame.
validatePanelData()	Validates data entered by user.

4. Write custom commands. These commands update the database with the information entered, or perform some function when a user clicks **OK**. If the dialog is called using a viewCommand, you must write a custom viewCommand. The following is a sample controller command for a dialog:

```

package com.ibm.commerce.tools.test;
import java.util.*;
import com.ibm.commerce.ras.*;
import com.ibm.commerce.server.*;
import com.ibm.commerce.command.*;
import com.ibm.commerce.exception.*;
import com.ibm.commerce.datatype.*;
import com.ibm.commerce.tools.common.*;
import com.ibm.commerce.tools.command.*;
import com.ibm.commerce.tools.common.ui.*;
import com.ibm.commerce.exception.*;
import com.ibm.commerce.tools.resourcebundle.*;
import com.ibm.commerce.tools.util.*;

public class TestCmdImpl extends ToolsControllerCommandImpl implements TestCmd {
protected ResourceBundleProperties resourceBundle = null;
protected String viewname = null;
protected String successMsg = "Success";
protected String errorMsg = "Error";

// sample input data
protected String name = null;
protected int age;
protected float salary;
protected String department = null;

```

```

public void performExecute() {

    /*
    * your business logic here
    */

    // exit successfully - forwarding to returning view command
    responseProperties = new TypedProperty();
    responseProperties.put(EConstants.EC_VIEWTASKNAME, viewname);
    responseProperties.put(UIProperties.SUBMIT_FINISH_MESSAGE, successMsg);
}

public void validateParameters() throws ECEException {

String methodName = "validateParameters";

    /* optionally, load your resource bundle file and NL messages
    Locale locale = commandContext.getLocale();
    resourceBundle =
    (ResourceBundleProperties) ResourceDirectory.lookup("samples.samplesNLS", locale);
    successMsg = (String) resourceBundle.get("successMsg");
    errorMsg = (String) resourceBundle.get("errorMsg");
    */

    // retrieve the data which was set on the client using parent.put
    name = (String) requestProperties.getString("name", null);
    age = requestProperties.getIntValue("age", 0);
    salary = requestProperties.getFloatValue("salary", 0);
    department = (String) requestProperties.getString("department", null);

    // set returning URL view name
    viewname = requestProperties.getString(EConstants.EC_REDIRECTURL);

    /* error use case - returning error code and
    message to view command via ECEException
    if (salary < 10000.00 || salary > 99999.99) {
    responseProperties = new TypedProperty();
    responseProperties.put(UIProperties.SUBMIT_ERROR_STATUS, "101");
    responseProperties.put(UIProperties.SUBMIT_ERROR_MESSAGE, errorMsg);
    throw new ECAApplicationException(
    ECToolsMessage.TOOLS_TEST_USER_ERROR,
    this.getClass().getName(),
    methodName,
    null,
    viewname,
    responseProperties);
    }
    */


}
}
}

```

5. Register your custom commands and JSP files. Once any required commands are created, register them in the database. Refer to the "Design Patterns" section of the *WebSphere Commerce Programming Guide and Tutorials* for details on how to register the command.

**Note:** Ensure that the URL field in the database matches the value of the finishURL attribute in your XML file.

6. Create a resource bundle with text that names each panel. This text appears as the title of each panel. Resource bundles are in the `/WAS_installdir/installedApps/hostname/Enterprise_App_name.ear/properties/com/ibm/commerce/tools/component_name/properties` directory.

 **Developer** For WebSphere Commerce Developer, the resource bundles are in the `/WCDE_installdir/properties/com/ibm/commerce/tools/component_name/properties` directory. In this case, keys are based on the panel names given in your dialog XML file created in step 1.

The following is a sample resource bundle for the XML file defined in step 1.

```
# Panels name for TOC panel
initiativeGeneralPanel=General
initiativeLocationPanel=Location
initiativeConditionsPanel=Conditions
```

If national languages are supported, create the national language resource bundles with the appropriate language text. The national language file names must end with the locale supported. For example, for a French-language resource bundle, the file name should be `filename_fr_FR.properties`.

7. Write context sensitive help files and update the *Tools User Interface Center* Help Map XML file for context sensitive help. see "Adding context sensitive help" in chapter 7 for more information.
8. Add your new dialog to a *Tools User Interface Center* menu system (for example, WebSphere Commerce Accelerator.) See "Integrating Tools into the Tools User Interface Center" in chapter 7 for more information.
9. Stop and start your WebSphere Commerce server, then test your new dialog. This is the URL of the newly created dialog:

```
https://hostname:8000/webapp/wcs/tools/servlet/DialogView?XMLFile=namespace.myTestDialog
```

, where *namespace* is a namespace defined in `resources.xml`. See the *WebSphere Commerce Accelerator Customization Guide* for more information on XML files.

---

## Navigation

Dialogs present users with one or two navigation options, either finish or cancel. These two options are presented as buttons in the navigation frame at the bottom of the content window. If included, they behave according to the following guidelines:

- Finish** A user clicks on the Finish or OK button to finish the dialog, and the following takes place:
1. Check to see if the contents panel has finished loading. If not, return immediately and do nothing.
  2. Save the panel data.
  3. Validate the current panel data.
  4. Call the `preSubmitHandler()`.
  5. If a finish command is specified, (by the `finishURL` attribute in the wizard XML file), convert the model to XML and send it as a parameter to the finish command.
  6. Set a flag to indicate that the finish command button is finished executing.
  7. If an error occurs in the finish server command (performing an update or otherwise), the command redirects back to the navigation frame passing back the following two parameters:

### **SubmitErrorMessage**

The translated error message to display to the user.

### **SubmitErrorStatus**

Error status or error code.

If the `SubmitErrorStatus` variable is passed in, then `submitErrorHandler()` is called with the above two variables as parameters. You are responsible for implementing this function in the JavaScript file to display the error message and redirect user to the

problematic input field based on the error code. If this function is not defined, a default alert window is displayed with the error message. If the `SubmitErrorStatus` is not passed in, then it is assumed that the finish command succeeded. At this point, the `submitFinishHandler()` function is called with the `submitFinishMessage` as parameter, which was set in the controller command.

### Cancel

Displays a cancel confirmation dialog. If the user clicks **OK**, `submitCancelHandler()` then the parent frame's cancel method run.

---

## Customizations

To use the following JavaScript functions, first write code for the functions that implements your business logic. Once the code is written you can use these JavaScript functions in your dialog. Your JavaScript file is specified in the dialog's XML file.

Function Name	Description
<code>submitErrorHandler(errMessage, errorStatus)</code>	Called when an error is received from the controller command.
<code>submitFinishHandler(finishMessage)</code>	Called upon successful completion by the controller command.
<code>submitCancelHandler()</code>	Called when <b>Cancel</b> is clicked and the user clicks <b>OK</b> in the cancel confirmation dialog.
<code>preSubmitHandler()</code>	Optionally, called after the <code>validateAllPanels()</code> function, but before the finish controller command.

---

## JavaScript functions

You can use these JavaScript functions in your JSP pages. These functions are defined in the parent frame, and are called using `parent.functionName()`:

Function Name	Description
<code>get(key, defaultValue)</code>	Returns the value of the specified key from the object model.
<code>remove(key)</code>	Removes the specified key from the object model.
<code>put(key, value)</code>	Stores the value for the specified key in the object model.
<code>addURLParameter(pname, value)</code>	Adds an additional URL parameter to finish command.
<code>removeURLParameter(pname)</code>	Removes the URL parameter, previously added using the <code>addURLParameter()</code> function.
<code>setContentFrameLoaded(value)</code>	Sets the <code>contentFrameLoaded</code> variable to either true or false. If true, then the user is allowed to switch to a different content panel. If false, the user must wait until the panel has been loaded which sets the value to true. <b>Note:</b> You must call <code>setContentFrameLoaded(true)</code> at the end of the panel's <code>onLoad</code> action. If this is not called, users will remain on the current panel.



---

## Chapter 4. Dynamic lists

Dynamic lists display data from the database, or some other data source whose data and elements are variables. Information describing the format in which to display this data, and actions users can perform on this data, are stored in an XML file. Dynamic lists can be used to display objects such as orders and products. Each chosen attribute of the object, for example, an order, displays in a separate column. A dynamic list is rendered as an HTML form.

Typically, dynamic lists have three frames: a scroll control frame, a base content frame, and a button frame. All of these frames can be customized using an XML file.

The scroll control frame is optional, but if present, it displays a national language enabled title for the page, and controls to navigate the list. This control panel is also optional, but if present, it provides **Next** and **Previous** buttons so that users can navigate through pages of entries in the list. The control panel also provides text fields in which users can input the number of a page they want to browse.

The base content frame displays the primary content. A dynamic list is a table that contains user data from different data sources. The button frame, which is also optional, defines buttons which perform particular actions when clicked.

---

### Overview

The following is an overview of how to create a dynamic list. Detailed steps follow this section.

1. Create a dynamic list definition XML file that describes the frameset.
2. Register the dynamic list definition XML file in the component's specific resources.xml file.
3. Write the JSP file and JavaScript file, if necessary, that define your dynamic list page.
4. Create a resource bundle.
5. (Optional) Write context sensitive help files for your dynamic list and panels and update the *Tools User Interface Center* help map XML file (for example, AcceleratorHelpMap.xml for the WebSphere Commerce Accelerator) to include your help files.
6. Optionally, add your new dynamic list to a *Tools User Interface Center* menu system (for example, WebSphere Commerce Accelerator).
7. Launch and test your dynamic list.

The following files are created:

- A dynamic list definition XML file to describe the frameset *myDynamicList.xml*,
- resource bundle files, for example, *MyDynamicListResource\_locale.properties*
- HTML help files
- JSP files to fill panel contents, for example, *myDynamicList.jsp* to

The following files are modified:

- resources.xml

- *Tools User Interface Center* Help Map XML file to include your help files

## Detailed steps

The following steps are detailed instructions for implementing a dynamic list:

1. Create a dynamic list definition XML file that describes the frameset, called *newDynamicList.xml* for example. Create this file in */WC\_installdir/xml/tools/component*, where *component* is the name of the component to which the dynamic list belongs. The following tags are available for use in the dynamic list:

<pre>&lt;action&gt; &lt;/action&gt;</pre>	<p>The primary element defining a dynamic list. The following attributes are supported:</p> <p><b>resourceBundle</b> A required attribute that specifies which resource bundle is used. For example, resourceBundle="catalog.ItemNLS"</p> <p><b>FormName</b> A required attribute that specifies the form name. A dynamic list is rendered as an HTML form. For example, FormName="ItemFindResultsFORM"</p> <p><b>beanClass</b> An optional attribute that specifies a custom bean for a simple dynamic list. For example, beanClass="com.ibm.commerce.tools.catalog.beans.simpleCatalogBean"</p> <p><b>helpKey</b> An optional attribute that defines the corresponding help key in the <i>Tools User Interface Center</i> Help Map file. For example, helpKey="WC.auction.auctionWizardPricePanel.Help"</p>
---	---

<parameter> </parameter>	<p>This element defines parameters passed to the dynamic list. The following attributes are supported:</p> <p><b>listsize</b> A required attribute that sets the maximum number of list items that display on each page. For example, listsize="15"</p> <p><b>startindex</b> A required attribute that sets index value of first list item. For example, if the first item on this list has an index of 0, type: startindex="0"</p> <p><b>resultsize</b> A required attribute that defines the length of entire list. This value is not known until run time, and is dynamically calculated. However, this property must be defined as a placeholder. It does not matter what the initial value is, as it is recalculated. For example, resultsize="0"</p> <p><b>orderby</b> An optional attribute that identifies the column by which the table is sorted. For example, orderby="name"</p> <p><b>itemName</b> An optional attribute available as a user defined parameter.</p>
<scrollcontrol> </scrollcontrol>	<p>This element defines the scroll control frame. The following attributes are supported:</p> <p><b>title</b> An optional attribute that defines the page title. For example, title="Products"</p> <p><b>display</b> A required element that determines whether the scroll control frame is displayed. For example, display="true"</p>
<controlpanel> </controlpanel>	<p>This element defines the control panel. The following attributes are supported:</p> <p><b>display</b> A required element that determines whether the control panel is displayed. For example, display="true"</p>
<button> </button>	<p>This element defines the button frame. The button element contains &lt;menu&gt; elements.</p>

<menu> </menu>	<p>This element defines a button. The following attributes are supported:</p> <p><b>name</b> A required attribute that specifies a name for the button. For example, name="New"</p> <p><b>action</b> A required attribute that defines the action performed when the button is clicked. For example, action="Action()"</p> <p><b>users</b> A required attribute that defines the access control for the button. The value must be a space delimited list of the roles that are permitted to perform the action associated with the button. For WebSphere Commerce Accelerator, the names are defined in xml/tools/common/roles.xml and match the ID column in the MBRGRP table. For example, users="makMgr merMgr merchant siteAdmin"</p> <p><b>selection</b> An optional attribute that defines how selected items in the dynamic list affect the button. Permitted values include "single", "multiple", or "none". These values specify that the button is enabled when only one item is selected, when one or more items are selected, or when no items are selected, respectively. In the case of "single" and "multiple", the action associated with the button is performed on the selected items. For example, selection="multiple"</p> <p><b>component</b> An optional attribute that sets the component of this button. For example, component="CSRComponent"</p>
<view> </view>	<p>This element defines an optional view filter contained in a drop down list in the control panel. The following attributes are supported:</p> <p><b>name</b> A required attribute that specifies a name for the list item. For example, name="newList"</p> <p><b>action</b> An optional attribute that specifies the action performed when this view filter is selected. For example, action="Action()"</p>

<pre>&lt;jsFile&gt; &lt;/jsFile&gt;</pre>	<p>Specifies a JavaScript file to be included in the dynamic list. Files defined here are included in the parent frame. Thus, access to these functions require parent. appended to function calls to scope them to the parent frame. Multiple JavaScript files are allowed. The following attribute is supported:</p> <p><b>src</b> A required attribute that specifies the location of the JavaScript file. For example,  <pre>src="/wcs/javascript/tools/common/DateUtil.js"</pre></p>
---	---

The following is a sample dynamic list definition XML file. It has buttons, view filters available for selection in the scroll control frame, a resource bundle for the national language text, a title displayed in the title area (the title is a key which maps to the resource bundle), and a help key for the base, buttons, and scroll control frames. Each view tag added is put into the view filter drop down list for filtering different types of contents.

```
<?xml version="1.0" encoding="UTF-8" ?>
<action resourceBundle="campaigns.campaignsRB" formName="initiativeForm"
  helpKey="MC.campaigns.InitiativeList.Help">

<parameter listsize="22" startindex="0" endindex="0" orderby="name" state="AllList" />
<scrollcontrol title="initiativeListTitle" display="true" />
<controlpanel display="true" />
<button>
  <menu name="new" action="basefrm.newInitiative()" />
  <menu name="copy" action="basefrm.copyInitiative()" selection="single" />
  <menu name="delete" action="basefrm.deleteInitiative()" selection="multiple" />
  <menu name="properties" action="basefrm.initiativeProperties()" selection="single" />
  <menu name="resume" action="basefrm.resumeInitiative()" selection="multiple" />
  <menu name="suspend" action="basefrm.suspendInitiative()" selection="multiple" />
  <menu name="statistics" action="basefrm.initiativeStatistics()" selection="single" />
  <menu name="reports" component="CommerceAnalyzer"
    action="basefrm.initiativeReports()" />
</button>

<view name="AllList" action="top.setContent(basefrm.getListTitle(),
  '/webapp/commerce/tools/servlet/NewDynamicListView?state=AllList&
ActionXMLFile=campaigns.InitiativeList
&cmd=CampaignInitiativeListView', false)" />
<view name="ActiveList" action="top.setContent(basefrm.getListTitle(),
  '/webapp/commerce/tools/servlet/NewDynamicListView?state=ActiveList&
ActionXMLFile=campaigns.InitiativeList
&cmd=CampaignInitiativeListView', false)" />
<view name="SuspendList" action="top.setContent(basefrm.getListTitle(),
  '/webapp/commerce/tools/servlet/NewDynamicListView?state=SuspendList&
ActionXMLFile=campaigns.InitiativeList&
&cmd=CampaignInitiativeListView', false)" />

</action>
```

2. Register the dynamic list definition XML file created in step 1 in the appropriate resources.xml file. Multiple versions of this file exist, one for each component, in the following directory:

`/WC_installdir/xml/tools/component/resources.xml`.

Make an entry similar to the following in the resources.xml file:

```
<XML name="sampleList" file="component/sampleList.xml"/>
```

where *component* is the name of the component to which the dynamic list belongs. The name attribute becomes a key which will be used in a later step. The resources.xml files are referenced in the *instancename.xml* file, and you must register any new resources.xml files in *instancename.xml*.

3. Write the JSP and JavaScript files. Use the following Java methods (from the class `com.ibm.commerce.tools.common.ui.taglibs.comm`) to create your dynamic list table:


**Note:** `"/wcs/javascript/tools/common/dynamiclist.js"` and `"/wcs/javascript/tools/common/Util.js"` must be included in your JSP page to make these Java methods function properly.

Java Method	Description
<pre>public static void addControlPanel(String xmlfile, int totalpage, int totalitem, Locale loc) <b>Deprecated</b></pre>	<p><b>Note:</b> This method is deprecated and is listed here for reference only. See the following equivalent JavaScript function:  <code>parent.set_t_page_item(totalitem,listsiz);</code></p> <p>Adds the control panel to the scroll control frame. The following parameters are supported:</p> <p><b>xmlfile</b> A required string parameter which is an XML file that defines the page.</p> <p><b>totalpage</b> A required integer parameter corresponding to the total number of pages for the list.</p> <p><b>totalitem</b> A required integer parameter corresponding to the total number of entries in the list.</p> <p><b>loc</b> A required locale parameter corresponding to the national language in which the page is displayed.</p> <p>Throws: <code>ECSYSTEMException</code></p>
<pre>public static void startDlistTable(String tableid)</pre>	<p>This method begins the table definition. It is equivalent to the following HTML code:  <code>&lt;table style='...' id='tableid'&gt;</code></p> <p>The following parameter is supported:</p> <p><b>tableid</b> A required string parameter specifying the ID for the table.</p> <p>Throws: <code>ECSYSTEMException</code></p>
<pre>public static void endDlistTable()</pre>	<p>This method ends the table definition. It is equivalent to the following HTML code:  <code>&lt;/table&gt;</code></p>
<pre>public static void startDlistRowHeading()</pre>	<p>This method begins a heading row definition. It is equivalent to the following HTML code:  <code>&lt;tr style='... '&gt;</code></p>
<pre>public static void endDlistRowHeading()</pre>	<p>This method ends a heading row definition. It is equivalent to the following HTML code:  <code>&lt;/tr&gt;</code></p>

Java Method	Description
<pre>public static void addDlistColumnHeading(String hvalue, String svalue, boolean role, String width, Boolean wrap)</pre>	<p>This method inserts a column into a row. The following parameters are supported:</p> <p><b>hvalue</b> A required string parameter which specifies a heading name for this column. The value should be a key in a resource bundle file.</p> <p><b>svalue</b> A required string parameter that specifies the value sent back to your data bean for sorting purposes. If you do not intend to make the column sortable, set this to null.</p> <p><b>role</b> A required Boolean parameter that determines if this column requires sorting. If no sorting is required, set this value to false.</p> <p><b>width</b> An optional string parameter which specifies the width of this column, represented by a percentage. If this is not required, set this value to null.</p> <p><b>wrap (opt)</b> An optional Boolean parameter which determines whether the text in this column should wrap.</p> <p>Throws: <code>ECSystemException</code></p>
<pre>public static void addDlistCheckHeading(Boolean check, String checkfnc)</pre>	<p>This method adds a <b>Select All</b> or <b>Deselect All</b> check box. The following parameters are supported:</p> <p><b>check</b> A required Boolean parameter which determines whether the <b>Select All</b> checkbox is included.</p> <p><b>checkfnc</b> An optional string parameter which sets a user defined function. The default value is null.</p>
<pre>public static void endDlistRow()</pre>	<p>End row definition, equivalent to HTML code: &lt;/TR&gt;</p>
<pre>public static void startDlistRow(int row)</pre>	<p>This method begins a row definition. The following parameters are supported:</p> <p><b>row</b> A required integer parameter which determines the row style. This parameter accepts the following values, 1 (color 1) or 2 (color 2). These should be alternated to improve table readability. These colors are specified in a cascading style sheet specified in the panel's JSP page.</p>

Java Method	Description
<pre>public static void addDlistCheck(String name, String fnc, String value)</pre>	<p>This method adds a check box column so that list entries are selectable. The following parameters are supported:</p> <p><b>name</b> A required string parameter, which specifies a name for the check box. This name should be unique.</p> <p><b>fnc</b> A required string parameter that specifies a user defined function for this check box. If no function is defined, set this to none.</p> <p><b>value</b> An optional string parameter that specifies a value for the check box. The default value is null.</p> <p>Throws: ECSystemException</p>
<pre>public static void addDlistColumn(String name, String link)</pre>	<p>This method adds a column cell. The following parameters are supported:</p> <p><b>name</b> A required string parameter which specifies content for this column cell.</p> <p><b>link (opt)</b> An optional string parameter which specifies a URL link for this column cell. If no link is applicable, set this to none.</p> <p>Throws: ECSystemException</p>

4. Create a resource bundle with text that displays for your dynamic list. Resource bundles are in the `/WAS_installdir/installedApps/hostname/Enterprise_App_name.ear/properties/com/ibm/commerce/tools/component_name/properties` directory.

 For WebSphere Commerce Developer, the resource bundles are in the `/WCDE_installdir/properties/com/ibm/commerce/tools/component_name/properties` directory.

If national languages are supported, create the national language resource bundles with the appropriate language text. The national language file names must end with the locale supported. For example, for a French-language resource bundle, the file name should be `filename_fr_FR.properties`.

5. (Optional) Write context sensitive help files and update the *Tools User Interface Center* Help Map XML file for context sensitive help. Add an entry similar to the following:

```
<help key = "MC.component.panelname.Help" file = "filename.htm"/>
```

For more information see the Chapter 7: Tools User Interface Center.

6. Add your new dynamic list to a *Tools User Interface Center* menu system, for example, WebSphere Commerce Accelerator. This step is optional, depending on whether you want to add an item to an existing menu. This is referred to as a node in this XML file. Add the following line:

```
<node name="myList" url="/wcs/commerce/tools/servlet/
NewDynamicListView?ActionXMLFile=common.sampleList&cmd=myDynamicList" />
```

For more information see Chapter 7: Tools User Interface Center.

7. Stop and start your WebSphere Commerce server. Launch and test your new dynamic list:  
`https://hostname:8000/webapp/wcs/tools/servlet/NewDynamicListView?ActionXMLFile=sample.sampleList=SampleListView`



---

## Multiple Framesets

A set of JavaBeans™ are provided to handle the rendering of different framesets described below:

- A regular frameset which includes two frames; a list in the base content frame, and the button frame. To create this frameset, use the `getFrameset()` method. This frameset is available to create wizards, dialogs, and notebooks.
- A scroll control frameset which has three frames; the scroll control frame, the button frame, and the base frame. To create this frameset, use the `getScrollControlFrameset()` method. This frameset should only be used outside of a wizard, dialog, or notebook.
- A second scroll control frameset which does not have a button frame. To create this frameset, use the `getScrollControlButtonlessFrameset()` method.

When creating a dynamic list in a notebook, dialog, or wizard, certain JavaScript functions must be added to the parent frame of the list (for example, the `savePanelData()` function). In this case the developer must:

1. Create a parent frame JSP file which contains the frameset and includes the required JavaScript functions.
2. Make a copy of `NewDynamicList.jsp`, in your component's location in the file system.
3. Rename the `NewDynamicList.jsp` file as required.
4. If necessary, create a view command for the JSP file.

---

## Filter enhancement

Instead of specifying a `views` parameter in the URL of the scroll control command, you can specify each view in the action XML file. To specify a view filter, place a node in the action XML file as follows:

```
<action ...>
...
...
<view name="sampleList1" actionFile="common.sampleListSC"/>
</action>
```

where the `name` attribute is the key to the resource bundle for the name of the view, and the `actionFile` attribute identifies the XML file which defines the view.

---

## JavaScript functions

You must implement the following JavaScript functions in your JavaScript file and use them in your dynamic list. You must call the following JavaScript functions in your base frame JSP page.

JavaScript Functions	Description
<code>parent.loadFrames()</code>	Called when the page is initialized, this loads the other frames while the base frame is loading.
<code>parent.afterLoads()</code>	Called when the page loading completes, this finishes loading the page.
<code>parent.setResultssize(<i>size</i>)</code>	Called after the page is loaded, this specifies the list size.
<code>getUserNLSTitle()</code>	If you implement this function, the title in the scroll control frame is replaced by the translated version returned by this function.

You can use the following JavaScript functions in your dynamic list, they are implemented by WebSphere Commerce by default. These following functions are defined in the parent frame, and are called using `parent.functionName()`:

JavaScript Functions	Description
<code>parent.setTotalPage(<i>num</i>)</code>	Sets the total number of pages for the control panel navigation.
<code>parent.setNumPage(<i>num</i>)</code>	Sets the current number of the page being displayed in the control panel navigation.
<code>parent.setTotalItem(<i>num</i>)</code>	Sets the total number of items in the control panel navigation.
<code>parent.displayButton(<i>name</i>)</code>	Dynamically displays the button name.
<code>parent.hideButton(<i>name</i>)</code>	Dynamically hides the button name.
<code>parent.showControlPanel()</code>	Dynamically shows the control panel.
<code>parent.hideControlPanel()</code>	Dynamically hides the control panel.
<code>parent.removeEntry(<i>name</i>)</code>	Explicitly removes the selected element by its name (same as check box name).
<code>parent.setInstruction(<i>text</i>)</code>	Sets the instruction text in the scroll control frame.
<code>parent.setButtonPos(<i>x,y</i>)</code>	Dynamically sets the button position. For example, <code>setButtonPos('0px', '15px')</code> moves all of the buttons down by 15 pixels. Other valid units are centimeters (cm), millimeters (mm), inches (in), points (pt), and picas (pc).
<code>parent.removeEntry()</code>	Dynamic lists keep track of what was checked in previous visits to the page. If these items no longer meet the criteria for inclusion in the list, you may use the <code>parent.removeEntry</code> method to remove all of the checked items.

---

## Chapter 5. Calendars

This element displays a calendar, allowing the user to specify a date, either graphically or manually. The specified date is then returned to the parent window, and placed in the correct field. The calendar displays embedded in the HTML page, not in it's own window.

---

### Overview

The following is an overview of how to create a calendar. Detailed steps follow this section.

1. Include DateUtil.js file in the JSP file defining the panel on which you require a calendar.
2. Define the year, month, and day fields.
3. Initialize the year, month, and day fields.
4. Define the calendar window.
5. Define the calendar icon.
6. Define the setupDate() function to set parameters for calendar window.

The following files are modified:

The JSP page, to which you added the calendar

---

### Detailed steps

1. Include the JavaScript file DateUtil.js in your JSP file. Include the file by adding a line similar to the following:

```
<SCRIPT SRC="/wcs/javascript/tools/common/DateUtil.js"></SCRIPT>
```

2. Define a form that holds the Year, Month, and Day fields. These fields are where the user enters the date:

```
<FORM NAME=form1 METHOD=POST>
  <INPUT TYPE=TEXT VALUE="" NAME=YEAR1 SIZE=4>
  <INPUT TYPE=TEXT VALUE="" NAME=MONTH1 SIZE=2>
  <INPUT TYPE=TEXT VALUE="" NAME=DAY1 SIZE=2>
</FORM>
```

**Note:** You can choose any names for the form and input fields, as long as they match the names used in the following steps.

3. Initialize the Year, Month, and Day fields with current date when page loads, using the following code:

```
function init() {
  document.form1.YEAR1.value = getCurrentYear();
  document.form1.MONTH1.value = getCurrentMonth();
  document.form1.DAY1.value = getCurrentDay();
}
```

```
...
<BODY ONLOAD="init()">
```

4. Copy and paste the following lines after the <BODY> tag:

```
<SCRIPT FOR=document EVENT="onclick()">
  document.all.CalFrame.style.display="none";
</SCRIPT>
<IFRAME name="calendar" title="Calendar" STYLE="display:none;position:absolute;
```

```
width:198;height:230;z-index=100"
  ID="CalFrame" MARGINHEIGHT=0 MARGINWIDTH=0 NORESIZE
FRAMEBORDER=0 SCROLLING=NO
  SRC="/webapp/wcs/tools/servlet/Calendar">
</IFRAME>
```

This loads the calendar window into your JSP page, though it is initially invisible.

5. Add the Calendar icon to your page using the following code. This displays the calendar window when clicked.

```
<A HREF="javascript:setupDate();showCalendar(document.form1.calImg1)">
<IMG SRC="/wcs/images/tools/calendar/calendar.gif" BORDER=0 id=calImg1>
```

6. Define the setupDate() function used in step 5, using the following code:

```
function setupDate() {
  window.yearField = document.form1.YEAR1;
  window.monthField = document.form1.MONTH1;
  window.dayField = document.form1.DAY1;
}
```

This function sets the targets for data set by your user in the calendar window.

---

## JavaScript functions

You can use the following JavaScript functions in your calendar, they are implemented by WebSphere Commerce by default. These functions are defined in the parent frame, and are called using *parent.functionName()*:

Function Name	Description
validDate(String <i>inYear</i> , String <i>inMonth</i> , String <i>inDay</i> )	This function validates a selected date. It returns true if the date is valid, otherwise, it returns false.
isLeapYear(int <i>Year</i> )	This function determines if a particular year is a leap year. It returns true if it is a leap year, otherwise, it returns false.
getCurrentYear()	This function returns the current year.
getCurrentMonth()	This function returns the current month.
getCurrentDay()	This function returns the current day.

Function Name	Description
<p>validateStartEndDateTime(String <i>inStartYear</i>, String <i>inStartMonth</i>, String <i>inStartDay</i>, String <i>inEndYear</i>, String <i>inEndMonth</i>, String <i>inEndDay</i>, String <i>startTime</i>, String <i>endTime</i>)</p>	<p>This function checks that the end date and time is after the start date and time. This is useful for validating two dates to make sure that one is greater or equal to the other. Validate the dates before calling this function to ensure that they are in this format. This function expects the <i>startTime</i> and <i>endTime</i> arguments to be in HH:MM format. Validate the times first to make sure they are in this format. Input: <i>startDate</i>, <i>endDate</i>, <i>startTime</i>, <i>endTime</i>.</p> <p>Enter null for the date arguments if you only require a time comparison.. Enter null for time arguments if you only require a date comparison. Enter all arguments if you require both date and time comparisons.</p> <p>This function returns one of the following values:  Return code = true,  <i>endDate+endTime</i> &gt; <i>startDate+startTime</i>  Return code = false,  <i>endDate+endTime</i> &lt; <i>startDate+startTime</i>  Return code = -1,  <i>endDate+endTime</i> == <i>startDate+startTime</i></p>
<p>showCalendar(Object <i>callmg</i>)</p>	<p>This function shows the calendar window, positioned according to the referenced object</p>



---

## Chapter 6. Slosh buckets

The slosh bucket is a common visual metaphor containing two single column lists, side by side. Four buttons are placed between the controls: Add, Add All, Remove, Remove All. This control is generally used to pick a subset of unique entries from a larger set of data.

---

### Overview

The following is an overview of how to create a slosh bucket. Detailed steps follow this section.

1. Write your page content using JSP and JavaScript files.
2. Create a resource bundle.

The following files are modified:

- The JSP page, to which you added the slosh bucket
- Modified resource bundle files *MyPageResource\_locale.properties*

---

### Detailed steps

The following steps are detailed instructions for implementing a slosh bucket.

1. Write your page content using JSP files, JavaScript files.


Your custom JSP file, *myUpdatedPage.jsp*, must include the following JavaScript file:

```
<SCRIPT SRC="/wcs/javascript/tools/common/SwapList.js"></SCRIPT>
```

The following is sample HTML which includes a slosh bucket on the page. The values for the *buttonName*, *buttonName2*, and the *customJavaScriptFunction* variables that should be replaced by your values.

```
<TR>
<TD VALIGN="BOTTOM" CLASS="selectWidth">
  <SELECT NAME="collateralSelected" CLASS='selectWidth' SIZE='5' MULTIPLE
    onChange="customJavaScriptFunction"></SELECT>
</TD>
<TD WIDTH=150px ALIGN=CENTER><br>
  <INPUT TYPE="button" NAME="addToSloshBucketButton" VALUE="buttonName"
    style="width: 120px" ONCLICK="addToSelectedCollateral();" ><br>
  <INPUT TYPE="button" NAME="removeFromSloshBucketButton" VALUE="buttonName2"
    style="width: 120px" ONCLICK="removeFromSelectedCollateral();" ><br>
</TD>
<TD VALIGN="BOTTOM" CLASS="selectWidth">
  <SELECT NAME="collateralAvailable" CLASS='selectWidth' SIZE='5' MULTIPLE
    onChange="customJavaScriptFunction"></SELECT>
</TD>
</TR>
```

2. Create a resource bundle with text for the slosh bucket. Resource bundles are in the */WAS\_installdir/installedApps/hostname/Enterprise\_App\_name.ear/properties/com/ibm/commerce/tools/component\_name/properties* directory.

 For WebSphere Commerce Developer, the resource bundles are in the */WCDE\_installdir/properties/com/ibm/commerce/tools/componentname/properties* directory.

If national languages are supported, create the national language resource bundles with the appropriate language text. The national language file names

must end with the locale supported. For example, for a French-language resource bundle, the file name should be *filename\_fr\_FR.properties*.

---

## Customizations

To increase the width of a slosh bucket list, in your output JSP file, set the following variable as follows:

```
<style type='text/css'>
    .selectWidth {width: 235px;}
</style>
```

Use code similar to the following in your JSP file when defining your list:

```
<SELECT NAME='definedShopperGroup'
    CLASS='selectWidth'
    MULTIPLE
    SIZE='<%=numOfVisibleItemsInList%>'
    onChange="updateSloshBuckets(this, document.f1.removeButton,
        document.f1.allShopperGroup, document.f1.addButton);">
</SELECT>
```

---

## JavaScript functions

You can use the following JavaScript functions in your slosh bucket, they are implemented by WebSphere Commerce by default. These functions are defined in the parent frame, and are called using *parent.functionName()*:

Function Name	Description
<i>move(fromList, toList)</i>	Moves one or more items from one list box to the other.
<i>allItemsSelected(aComponent)</i>	Determines whether the user has selected all of the items in a list box. Returns true if all of the items have been selected, otherwise returns false.
<i>setItemsUnselected(aComponent)</i>	Sets all items in a particular list box as unselected.
<i>setItemsSelected(aComponent)</i>	Sets all items in a particular list box as selected.
<i>setAnItemSelected(aComponent, value)</i>	Sets a single item in a particular list box as selected.
<i>isItemSelected(aComponent, value)</i>	Determines whether a single item in a particular list box has been selected. Returns either true or false.
<i>hasItem(aComponent, item)</i>	Determines whether a single item is listed in a particular list box. Returns true if the item is in the list, otherwise returns false.
<i>isListBoxEmpty(aComponent)</i>	Determines whether a particular list box is empty. Returns true if the list box is empty, otherwise returns false.
<i>countSelected(aComponent)</i>	Determines the number of items the user has selected. It returns 0 if no items are selected, or an integer matching the number of selected items.
<i>setButtonContext(aComponent, aButton)</i>	Enables or disables a button associated with a slosh bucket list box. Input arguments include a list box and a button. If any items are selected in the list box, the button is enabled, otherwise, it is disabled. This behavior disables the remove button, for example, when no items are in the list, and therefore cannot be removed. Use this function in the onChange block of the SELECT HTML form.



Function Name	Description
updateSloshBuckets( <i>aComponent1</i> , <i>aButton1</i> , <i>aComponent2</i> , <i>aButton2</i> )	This function updates which buttons are enabled and disabled depending on where the user last clicked. For example, if the slosh bucket consists of two side by side lists and the user clicks on an item in the left list, this function enables the button to move the item to the right list and disables the button to move items from the right to the left list.
initializeSloshBuckets ( <i>aComponent1</i> , <i>aButton1</i> , <i>aComponent2</i> , <i>aButton2</i> )	This function is similar to "updateSloshBuckets()" except that it is designed to be called when the page is first loaded, as part of the "onLoad" event.
whichItemIsSelected( <i>aComponent</i> )	Determines which item the user has selected in an option box. It returns the value of the first selected item found. If no items are selected, it returns an empty string.



---

## Chapter 7. Tools User Interface Center

The Tools User Interface Center provides the structural framework in which tools are presented to the user. The Tools User Interface Center consists of a banner frame, which contains a progress indicator and a page history, a menu frame, and a content frame. The WebSphere Commerce Accelerator is an instance of the Tools User Interface Center. The Organization Administration Console is an example of another instance of the Tools User Interface Center.

---

### Integrating tools into a Tools User Interface Center

The steps below describe how to integrate a tool into the WebSphere Commerce Accelerator. For other Tools User Interface Centers, follow similar steps using that tool's XML and resource bundle files.

1. Open the Tools User Interface Center instance's definition XML file. For WebSphere Commerce Accelerator, open `/WC_installdir/xml/tools/common/CommerceAccelerator.xml`. Add menu items, nodes, or both to the above definition XML file as shown below:


```
<menuitem name="customerService" enabled="true" component="CommerceAnalyzer"
  users="cusRep merchant siteAdmin">
  <node name="customers" component="CampaignManagement"
    url="/webapp/wcs/tools/servlet/Calendar" users="merchant" />
  .
  .
  .
</menuitem>
```

Both the `<menuitem>` and the `<node>` elements support the following attributes, as used in the above code sample:

Attribute	Description
name	An required attribute that names the menu. This name is a key in the resource bundle file. If null, a default message is used. For example, <code>menuName = "myMenuText"</code>
enabled	A required attribute for the <code>&lt;menuitem&gt;</code> element, which determines whether the menu is available for selection. The value can be either <code>true</code> or <code>false</code> .
component	An optional element which specifies the component with which the menu item is associated. Components can be turned on or off using Configuration Manager. A menu item associated with a component which is disabled does not display.
url	A required element which specifies the target URL for the menuitem. When this URL is called, the tool displays in the content frame.

Attribute	Description
users	A required attribute that defines the access control for the menu. Unauthorized users will not see the menu or node when they log on. The value must be a space delimited list of the roles that are permitted to perform the action associated with the menuitem. These names are defined in the file roles.xml, and match the ID column in the MBRGRP table. For example, users="makMgr merMgr merchant siteAdmin"  The roles are predefined in ROLE table and roles.xml file.
display - optional	An optional attribute that specifies whether this menuitem is visible or not. The default value is "true".
type	An optional attribute that specifies the type of the menuitem, help is the only value supported.

Resource bundles are in the file  
`/WAS_installdir/installedApps/hostname/Enterprise_App_name.ear/  
properties/com/ibm/commerce/tools/  
properties/mccNLS_locale.properties.`

 For WebSphere Commerce Developer, the resource bundles are in the file `WCDE_installdir/properties/com/ibm/commerce/tools/properties/mccNLS_locale.properties.`

. Update this file with any text that displays in the menu item for your new element. The name of the text is defined in the file resources.xml. For example, for a dynamic list, add a line similar to `sampleList =Sample List`, where `sampleList` is the node name specified in resources.xml, and `Sample List` is the text that displays in the menu.

Note that this file is for WebSphere Commerce Accelerator. If you are customizing a different *Tools User Interface Center*, find the reference to the resource bundle file name at the top of the XML file, in a line similar to the following:

```
<menu resourceBundle="common.mccNLS">
```

---

## Adding context-sensitive help

No matter how intuitive you might think that your custom interface is, it is still possible that your users will have questions, about either usage or input. When a user clicks **Help**, the current page in the content frame is polled to obtain the help key for the page. This key is then included as an argument in a call to the `getHelp()` function, which launches a separate browser containing the HTML file associated with the specified key. To provide context-sensitive help for your custom interface elements, you need to create or modify a HTML help file, you need to specify a key in your element's defining XML file, and you must update the help mapping file for the *Tools User Interface Center*.

### Create or modify an existing HTML file

Writing the online help file for your custom page is beyond the scope of this document, but if your aim is to have the file launched in the product's help window, with a similar look and feel, you should base your help file on any of the files located in the `/WC_installdir/web/doc/locale/f1` directory. Taking one of these files, and updating it to reflect your interface, and then saving it with a

unique file name should be sufficient, so long as you also make a copy of the corresponding file in the `/WC_installdir/web/doc/locale/f1_fs` directory, and match the file name to the file you saved. The `f1_fs` directory contains files which define the frameset in which the help files display.

### Define a help key by adding an entry to the help map XML file

For example, for WebSphere Commerce Accelerator, the help map file is `/WC_installdir/xml/tools/common/AcceleratorHelpMap.xml`. Add a line similar to the following:

```
<help key="MC.notebook.panel1.Help" file ="sample_help1.html"/>
```

where `MC.notebook.panel1.Help` is the help key, and `sample_help1.html` is the help file name.

If you are working on a different center, you need to find out the file name by looking at the top of the tools center XML file in a line similar to the following:

```
<menu helpMap="common.MerchantCenterHelpMap">
```

### Reference the help key in your element's defining XML file.

In the XML file that defines the User Interface element, you must define a help key. The method by which you define the key depends on the user interface element you are creating.

- If you are writing a panel inside of a notebook, a wizard, or a dialog, the help key is defined in the XML file as shown below:

```
<panel name="Profile" url="/tools/sample/notebookPanel1.html"
      helpKey="MC.notebook.panel1.Help" />
```

- If you are writing a dynamic list page, the help key is defined in the XML file as shown below:

```
<action 1 ... HelpKey="MC.order.orderList.Help" ... />
```

- If your page is neither of the above types, you must define the following JavaScript function in your page:

```
function getHelp() {
    return "MC.myPage.Help"; // change this return value according to your actual helpKey
}
```

---

## JavaScript functions

The following JavaScript functions are available to your JSP page if it is inside of a *Tools User Interface Center*, from the top frame, accessible by `top.functionName()`.

JavaScript Function	Description
<code>put(String key, String value)</code>	Stores the given value in the top level JavaScript object.
<code>get(String key, String defaultValue)</code>	Returns the value of the given key from top level JavaScript object.
<code>remove(String key)</code>	Removes the given key from the top level JavaScript object.
<code>openHelp()</code>	Opens a context sensitive help window. This is functionally equivalent to when the user clicks <b>Help</b> .

JavaScript Function	Description
setContent(String <i>text</i> , String <i>link</i> , Boolean <i>newtrail</i> , Object <i>parameters</i> )	<p>Sets the content frame URL and updates the breadcrumb trail. Supported arguments are:</p> <p><b>text</b> This value specifies that text that displays in the breadcrumb trail.</p> <p><b>link</b> This value defines the URL to load in the content frame.</p> <p><b>newtrail</b> This value specifies whether a new item should be added to the breadcrumb trail. If true, an additional item is added to the end of the breadcrumb trail; if false, this item replaces the last item in the breadcrumb trail.</p> <p><b>parameters (optional)</b> Use this only if your link contains locale dependent characters (which may get corrupted using direct URL location replacement). If this parameter is used, the Tools Framework dynamically generates a form based on this parameter object and submits the URL parameters as name-value pairs.</p>
showContent(String <i>link</i> , Object <i>parameters</i> )	<p>Sets content frame URL without updating the breadcrumb trail. Supported arguments are:</p> <p><b>link</b> This value defines the URL to load in the content frame.</p> <p><b>parameters (optional)</b> Use this only if your link contains locale dependent characters (which may get corrupted using direct URL location replacement). If this parameter is used, the Tools Framework dynamically generates a form based on this parameter object and submits the URL parameters as name-value pairs.</p>
setHome()	Sets the content frame to the default homepage, and resets the breadcrumb trail to the initial state. For example, sets the breadcrumb trail to "logout - home".
goBack()	Goes back one item in the breadcrumb trail and removes the last item from the breadcrumb trail.
resetBCT()	Resets the breadcrumb trail to the initial state. For example, sets the breadcrumb trail to "logout - home".
refreshBCT()	Refreshes the breadcrumb trail to reflect the current state.
showProgressIndicator(Boolean <i>flag</i> )	Manually turns on or off the progress indicator.
SaveData(Object <i>model</i> , String <i>slotName</i> )	<p>Saves data so that it can be retrieved later, either in the same page or in another page or element. Supported arguments are:</p> <p><b>model</b> Specifies a data object, which requires saving.</p> <p><b>slotName</b> A handle to be used later to retrieve the data.</p>

JavaScript Function	Description
getData(String <i>slotName</i> , int <i>stepsBack</i> )	Gets data saved before using the SaveData function. Supported arguments are:  <b>slotName</b> A handle pointing at the location where the data was previously saved.  <b>stepsBack</b> An optional value, which specifies how many items back, with respect to the breadcrumb trail, this data object was saved. The default value is '0'.
sendBackData(Object <i>data</i> , String <i>slotName</i> )	Sends a data object back to the calling wizard. This is used in a Wizard Chaining. Supported arguments are:  <b>Data</b> The data object you want to send back to the previous item in the breadcrumb trail.  <b>SlotName</b> A handle to be used later to retrieve the data.
saveModel(Object <i>model</i> )	A function provided for convenience which saves the "model" object, which is used often within notebooks, wizards, and dialogs. Supported arguments are:  <b>model</b> Specifies the model object.
getModel(stepsBack)	A convenience function to get back the "model" object previously saved. Supported arguments are:  <b>stepsBack</b> An optional value that specifies how many items back, with respect to the breadcrumb trail, this model object was saved. The default value is '0'.
setReturningPanel(String <i>panelName</i> )	Sets the returning panel name. Usually used in a Wizard Chaining scenario. Supported arguments are:  <b>panelName</b> Specifies the name of the panel in a notebook or wizard.
setRemoteHelp(String <i>key</i> )	Sets the remote helpkey. Use this function only if your next content page is served from a remote site. Due to a browser restriction, the tools user interface top frame cannot call remote content frame's getHelp() function. Instead, a help key must be set before the remote page is loaded.
getRemoteHelp()	Gets the remote helpkey set by the setRemoteHelp() function.
getCSSFile()	Gets the locale dependent cascading stylesheet file name. For example, centre_zh_TW.css.
menuVisible(int <i>index</i> , boolean <i>flag</i> )	Sets menu visible or hidden dynamically. Supported arguments are:  <b>index</b> Specifies the number of the menu.  <b>flag</b> A boolean, which determines whether the menu is visible. If true, the menu is visible.





---

## Chapter 8. Dynamic tree

A dynamic tree organizes information in a tree layout, allowing the user to navigate the tree, expand and collapse tree branches. The tree can be pre-cached to any number of levels, while leaves under uncached nodes are retrieved dynamically as they are expanded. When a user right clicks on a selected item, a context sensitive menu appears, allowing the user to open the object, or to perform other specified actions. The dynamic tree can contain any number of items, and can contain any number levels. A custom data bean provides the context menu items and actions as well as the tree structure, one node at a time. The dynamic tree also allows nodes of different types to be identified using custom icons and menu groups. Developers can also programmatically open a specific node, to which the tree automatically expands and highlights the specified node.

---

### Workflow

A description of the flow of data in the dynamic tree follows:

1. A tree frame is called by a client browser.
2. The tree frame initiates a hidden Data Frame URL.
3. The data frame calls the tools framework's DynamicTreeBean.
4. The DynamicTreeBean calls a custom DynamicTreeUserDataBean, which is specified in an XML file.
5. The custom DataBean returns the first level (or multiple levels if pre-caching is used) tree nodes.
6. The DynamicTreeBean returns the tree nodes (formatted into a JavaScript array) to the data frame.
7. The data frame calls tree frame's setNode() function which notifies it that the data fetch is complete, and passes the tree array. The tree frame expands and displays tree.
8. When a user clicks to expand a node that is not cached, the tree frame resets the location of the data frame to a new URL to populate the children of selected node. Steps 3-7 repeat for each node expansion.

---

### Overview

The following are the quick steps to create a dynamic tree, to be used by developers who are familiar with the details involved in each step. Detailed steps follow this section.

1. Create a dynamic tree definition XML file that defines the dynamic tree, based on DynamicTree.dtd.
2. Create a frameset definition file.
3. Write a data bean implementing DynamicTreeUserDataBean.
4. Write any required JavaScript or JSP files to perform actions on the dynamic tree.

---

### Detailed steps

The following steps are detailed instructions for implementing a dynamic tree.

1. Write a dynamic tree definition XML file specifying your data bean and required parameters. This dynamic tree definition XML file is used by a Tools Framework JSP which converts your data bean data into JavaScript. Thus, it must specify your data bean. Other features and configurations are listed below.

**Note:** Parameters with default values (specified in DynamicTree.dtd) do not need to be specified unless you want to change the default.

XML Tag	Description
<tree> </tree>	<p>The primary element defining a dynamic tree. The following attributes are supported:</p> <p><b>dataBean</b> A required attribute that specifies the fully qualified name of the data bean that populates the tree.</p> <p><b>initDataURLParam</b> An optional attribute used to pass any arguments you want to pass to the data bean (such as preLoad). By default, this string is empty. For example, initDataURLParam="preLoad=3&amp;myParam=something"</p> <p><b>targetFrame</b> An optional attribute that specifies the name of the content frame, as specified in your frameset definition page. The default value is "content". For example, targetFrame="content"</p> <p><b>expandInContextMenu</b> An optional boolean value that specifies whether the <b>Expand</b> or <b>Collapse</b> menu items appear in the context sensitive menu for all tree nodes with children. The default value is "true".</p> <p><b>folderIcon</b> An optional boolean value that specifies whether the folder icon appears in the tree. The default value is "true".</p> <p><b>contextMenu</b> An optional boolean value that specifies whether the contextMenu is enabled. This determines whether users can right-click on entries in the tree, and see the context menu. The default value is "true".</p> <p><b>expandLevel</b> An optional numerical value specifying how far to expand the tree when it first opens. <b>Note:</b> The tree can only open to levels which have been pre-loaded. The default value is "0".</p> <p><b>treeSearchFailedResourceBundle</b> An optional attribute that specifies the resource bundle that contains the error message to be displayed when the tree cannot find a specified node when performing a search. The default value is "common.uiNLS".</p> <p><b>treeTitle</b> An optional attribute that specifies that a title should be inserted above the tree using the &lt;h1&gt; HTML tag. The default value is an empty string.</p>

XML Tag	Description
<jsFile/>	<p>An optional element that must be inside the &lt;tree&gt; element if specified. This attribute specifies, a JavaScript file to be included in the tree JSP file. Files defined here are included in the parent frame. Thus, access to these functions require parent. prefixed to function calls to scope them to the parent frame. Multiple JavaScript files are allowed. The following attribute is supported:</p> <p><b>src</b> A required attribute that specifies the location of the JavaScript file. For example,  src="/wcs/javascript/tools/common/TreeUtil.js"</p>

The following is a sample dynamic tree definition XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DynamicTree SYSTEM "dynamicTree.dtd">

<tree databean="com.ibm.commerce.tools.test.DynamicTreeTestBean2"
  initDataURLParam="preLoad=2&uid=Item 1"
  targetFrame="content"
  expandInContextMenu="true"
  folderIcon="true"
  contextMenu="true"
  expandLevel="0"
  treeSearchFailedResourceBundle="common.uiNLS">

</tree>
```

2. Create a frameset definition file, if necessary. If your frameset definition file does not already exist, you must create one. Typically, a dynamic tree is displayed in a frameset where the dynamic tree is on the left and the content frame is on the right. This is a typical use, and variations require a custom implementation. The following is a sample frameset definition file:

```
<html>
  <frameset framespacing="0" border="0" frameborder="0" cols="60%,*">
    <frame src="/webapp/wcs/tools/servlet/DynamicTreeView?XMLFile=samples.testTree"
      name="tree">
    <frame src="/webapp/wcs/tools/samples/dtreebuttons.html" name="content">
  </frameset>
</html>
```

3. Write a data bean Java class, implementing the following interface:

```
public interface DynamicTreeUserDataBean extends
  com.ibm.commerce.beans.SmartDataBean {
  public Vector getNodeInfo() throws ECSystemException;
  public Vector getIconInfo() throws ECSystemException;
  public Vector getMenuInfo() throws ECSystemException;
}
```

Notice that `DynamicTreeUserDataBean` inherits from `SmartDataBean`, which means you need to implement all the regular `SmartDataBean` methods, plus the `getNodeInfo()`, `getIconInfo()`, and `getMenuInfo()` methods. Refer to the "Creating a new Entity Bean" section of *WebSphere Commerce Programming Guide and Tutorials* for information on how to write and use a `SmartDataBean`.

The `getNodeInfo()` method returns a vector of `com.ibm.commerce.tools.common.ui.DynamicTreeNode` objects, which describe each node on the tree individually. This vector is populated in the `populate()` method of your data bean. Below is a sample implementation of a `getNodeInfo()` method:

```
public java.util.Vector getNodeInfo() throws
  com.ibm.commerce.exception.ECSystemException {
  return nodeInfo; // populate nodeInfo before this method is called
}
```

The `getIconInfo()` method returns a vector of `DynamicTreeIconType` objects, which define a number of icons belonging to specific node types. This vector can be populated in the `populate()` method of your data bean. If you are not using icon types, this method may return "null". Following is a sample implementation of a `getIconInfo()` method:

```
public java.util.Vector getIconInfo()
    throws com.ibm.commerce.exception.ECSystemException {
    return iconInfo;
}
```

The `getMenuInfo()` method returns a `Vector` of `DynamicTreeMenuType` objects, which define a number of menus belonging to specific node types. This `Vector` is populated in the `populate()` method of your data bean. If you are not using menu grouping, this method may return "null". Following is a sample implementation of a `getMenuInfo()` method:

```
public java.util.Vector getMenuInfo()
    throws com.ibm.commerce.exception.ECSystemException {
    return menuInfo;
}
```

Use the `setRequestProperties()` method, which is required by the `SmartDataBean` interface, to set and retrieve request properties. Ensure that you capture the `gotoNode` parameter. If `gotoNode` is specified (which is a path separated by '/'), your data bean should return a `Vector` of `DynamicTreeNode`s, with children ending at that node. The following is an example of how to capture the `gotoNode` parameter from the `setRequestProperties()` method:

```
public void setRequestProperties(com.ibm.commerce.datatype.TypedProperty param)
    throws Exception {
    // get URL parameters from here... below is just an example
    requestProperties = param;          try {
        gotoNode = requestProperties.getString("gotoNode");
    } catch (ParameterNotFoundException e) { }
}
```

Use the `populate()` method to populate the `nodeInfo`, `menuInfo`, and `iconInfo` vectors that your bean must supply to the Dynamic Tree web page. This method is called when your `SmartDataBean`, (implementation of `DynamicTreeUserDataBean`), is activated, after the `setRequestProperties()` and `setCommandContext()` methods are called.

In the `populate()` method, you create the children nodes or leaves under the node that the user clicks on, or the root nodes, or the leaves if the tree is loading for the first time. In our samples, we use a vector called `nodeInfo` to hold `DynamicTreeNode` objects. See the following section for details about the `DynamicTreeNode` class.

You can also choose to preload grandchildren, or any other deeper level. The `DynamicTreeNode` class has a property called `children` that is a vector. This children vector can contain children of the current node. This nesting feature is useful when you want to preload the nodes for several levels at startup time. You can define the initial URL parameters in the XML file. Ensure that your data bean recognizes a special (your choice) URL parameter indicating when to preload. The tools framework suggests you use the `initDataURLParam` parameter from your XML file to include a `preLoad` name-value pair, to indicate the depth of tree nodes that need to be preloaded.

The following is an example of a `populate()` method. Note the technique for preloading children in the `looper()` method, which loads all of the children in a specified search path (for example, `Item 1.1/Item 1.1.5/Item 1.1.5.3`).

```
public void populate() throws Exception {
    // if searching for a specific node from URL param gotoNode, method looper() will take
    // care of it for us. Otherwise, just populate the tree with the node clicked on.
    if (!gotoNode.equals("")) {
        StringTokenizer st = new StringTokenizer(gotoNode, "/");
    }
}
```

```

if (st.hasMoreTokens()) {
    for (Enumeration e = looper(st, "Item 1").elements(); e.hasMoreElements();) {
        nodeInfo.addElement(e.nextElement());
    }
}
} else {
    // create 5 nodes in this sample
    for (int i = 1; i <= 5; i++) {
        // construct the children URL parameters,
        // the following is just an example (every other node has children)
        String param = "";
        if (i % 2 == 1)
            param = "uid=" + uid + "." + String.valueOf(i) + "&para2=somevalue";
        // set sample value of all node to 13
        String testValue = new String("13");
        // create a new node. "Catalog" is the menu type.
        DynamicTreeNode node = new DynamicTreeNode(uid + "." + String.valueOf(i),
testValue, param, "", "Catalog");
        // make every other node different
        if (i % 2 == 0) {
            node.setIconType(new String[] {"Arrow"});
            node.setMenuType("Speed");
        }
        // nodeInfo is a class field of type Vector
        nodeInfo.addElement(node);
    }
}

// create some menu groups
String[][] groupMenu = {"Test", "http://www.ibm.com"},
{"", ""},
{"Erase", "http://www.ibm.com"},
{"Shutdown", "http://www.ibm.com"};
DynamicTreeMenuType mnu = new DynamicTreeMenuType(groupMenu, "Catalog");

menuInfo.addElement(mnu);

String[][] groupMenu2 = {"Fast", "http://www.ibm.com"},
{"Slow", "http://www.ibm.com"},
{"Stop", "http://www.ibm.com"};
DynamicTreeMenuType mnu2 = new DynamicTreeMenuType(groupMenu2, "Speed");

// menuInfo is a class field of type Vector
menuInfo.addElement(mnu2);

// create some icon types
DynamicTreeIconType iconType = new DynamicTreeIconType("Calendar",
"calendar/calendar.gif");

// iconInfo is a class field of type Vector
iconInfo.addElement(iconType);

DynamicTreeIconType iconType2 = new DynamicTreeIconType("Arrow", "list/arrow.gif");

iconInfo.addElement(iconType2);
}

// looper method used when trying to locate a specific node
// this method rerecursively calls itself until the search path is exhausted
public java.util.Vector looper(StringTokenizer st, String sName) {
// create a default menu
String[][] menu = {"Open", "http://www.ibm.com"},
{"", ""},
{"Copy", "http://www.ibm.com"},
{"Delete", "http://www.ibm.com"},
{"Modify", "http://www.ibm.com"};
Vector newVec = new Vector();
// get the first part of the search path
String match = (String)st.nextElement();
// create 5 nodes on each level
for (int i = 1; i <= 5; i++) {
    String param = "";
    String testValue = new String("13");
    String cMenuParam = new String("myValue=Test&otherValue=3");
    String objType = new String("Speed");

```

```

        String name = sName + "." + String.valueOf(i);
        DynamicTreeNode node;
        // every other node will be a leaf by setting the childrenURLParam="" (param)
        if (i % 2 == 1) {
            param = "uid=" + name + "para2=somevalue";
            node = new DynamicTreeNode(name, testValue, param, menu);
        } else {
            node = new DynamicTreeNode(name, testValue, param, cMenuParam, objType);
            node.setIconType(new String[] {"Calendar", "Arrow"});
        }
        // if the newly created node/leaf matches the search string and there are still
        // more parts of the search path, call looper() again setting this nodes children
        // to contain the next level of the search path
        if (match.equals(name) && i % 2 == 1) {
            if (st.hasMoreElements()) {
                node.setChildren(looper(st, name));
            }
        }
        // finally add the DynamicTreeNode to the newVec
        newVec.addElement(node);
    }
    return newVec;
}

```

Each DynamicTreeNode object, stored in the nodeInfo vector, holds the properties of a tree node or leaf. The following are the possible properties of a DynamicTreeNode object:

Property	Description
protected String name	A required value that specifies a text label for the node. The locale specific label is obtained by your data bean.
protected String childrenUrlParam	A required value that specifies the URL parameters used to construct the full URL for expanding the node. For example, if the childrenUrlParam is p1=a&p2=99, a full URL will be constructed by JavaScript to something like:  <code>DynamicTreeData?XMLFile=common.testTree&amp;p1=a&amp;p2=99</code>  This full URL serves as the node-expanding-URL, which your data bean must understand when it is called. If this value is null or empty, and no children are preloaded, the node becomes a leaf and cannot be expanded from the tree interface.
protected String[][] contextMenu	A required array of string arrays which holds the name-value pairs for the context menu. The first item in the array is the locale dependent menu title. The second item in the array is the URL that launches the node's action. For example:  <code>contextMenu[0][0] = "Open";</code> <code>contextMenu[0][1] = "/webapp/wcs/tools/servlet/DialogView?XMLFile=csr.shopperSearch";</code> <code>contextMenu[0][0] = "";</code> <code>contextMenu[0][1] = "";</code>  <b>Note:</b> A blank name-value pair yields a divider line in the context menu.
protected Vector children	An optional vector of the same DynamicTreeNode class to hold this node's children info. It only needs to be populated if you want to preload more than one level's nodes. If not, leave it as null. If children are included in a node, the childrenURLParam is ignored.
protected String value	An optional value of this node or leaf. (Product ID, for example): 13323
protected String contextMenuParams	An optional value that specifies the parameters that are passed in all instances in the tree when an action is launched from the context menu.

Property	Description
protected String[] iconType	An optional value that specifies the icon type applied to this node or leaf. If this is specified, you must also have a matching DynamicTreeIconType object which specifies the actual icon graphic files included. A node or leaf may be of more than one icon type.
protected String menuType	An optional value that specifies the menu type for the current node or leaf. If this is specified, a matching DynamicTreeMenuType object must be created.

There are many constructors for the DynamicTreeNode class, but if the one you need is not there, use the setter methods to set individual properties. IconType objects have the following values:

Property	Description
protected String iconType	This value specifies the name of this icon type.
protected String[] icons	This value specifies the icon graphic files included for this icon type. The image path is relative from \wcs\web\images\tools\. For example: { "Arrow", "list/arrow.gif" }

MenuType objects have the following properties:

Property	Description
protected String menuType	This value specifies the name of this menu type.
protected String[][] menu	This value is an array of string arrays which hold the name-value pairs for the context menu. The first item in the array is the locale sensitive menu title. The second item in the array is the URL which launches the node's action. For example: { {"Open", "/webapp/wcs/tools/servlet/DialogView?XMLFile=csr.shopperSearch"}, {"", ""} }  <b>Note:</b> A blank name-value pair yields a divider line in the context menu.

- Write any required JavaScript and JSP files to perform actions on the tree. These files may add additional processing on selected tree nodes.

---

## Additional features

Searches in the dynamic tree are performed using a search path. A search path is the entire path, starting at the tree root, separated by forward slashes. For example, your JavaScript may call the dynamic tree function gotoAndHighlight("Item 1.1/Item 1.1.3/Item 1.1.3.5/Item 1.1.3.5.4") to search for Item 1.1.3.5.4. Your databean must accept the gotoNode parameter, and return a vector which contains all of the nodes from the root through to the level that contains the specified node or leaf. See the looper() method above for an example.

The developer may pass any additional URL parameters to the dynamic tree frame and these parameters can be picked up in the custom data bean to provide further control and processing logic. This should be done in a manner similar to

```
<frame src="/webapp/wcs/tools/servlet/DynamicTreeView?XMLFile=samples.testTree&myParam=myValue" name="tree">
```

where MyParam is the parameter you wish to pass and MyValue is the value of said parameter. Any additional parameters should be picked up by the setRequestProperties() method in your data bean.

Developers may open the tree to a specific node when the tree first starts by specifying the URL parameter gotoNode=xxxx (where xxxx is the search path) in the original call to the Dynamic Tree frame. This should be a the form similar to

```
<frame src="/webapp/wcs/tools/servlet/
DTree?XMLFile=samples.testTree&gotoNode=Item 1.1/Item 1.1.4"
name="tree">
```

**Note:** Your databean must return the specified node or leaf or must display "search failed" as soon as the tree is loaded. See the looper() method above for a sample of returning a specified node.

Font sizes and style are configurable in the centre.css files found in /WC\_installdir/web/tools/common directory.

The getHighlightedNode() function returns the currently selected node. For example:

```
var node = parent.tree.getHighlightedNode();
```

Once the node is retrieved, the developer has public access to all of the node data, which is the same as the DynamicTreeNode fields. Thus, the developer can programmatically modify this data. However, changes are not reflected outside of the tree unless the changes are also programmatically made by the developer. For example, the following JavaScript code shows the name and value of the currently selected node or leaf:

```
alert("Node name: " + node.name + "\nNode value: " + node.value);
```

The implementer may obtain the namePath or valuePath of any node by using getNamePath(node) and getValuePath(node) respectively.

---

## JavaScript functions

You can use the following JavaScript functions in your dynamic tree, they are implemented by WebSphere Commerce by default. These following functions are defined in the parent frame, and are called using parent.functionName():

Function Name	Description
gotoAndHighlight (namePath)	Searches the tree for the specified namePath. The namePath is the text displayed in the user interface. If found the tree expands to the specified path and the node or leaf is highlighted. If not found an error message is displayed.
gotoAndHighlight (valuePath)	Searches the tree for the specified valuePath. The valuePath is the value of the node in the tree. If found the tree expands to the specified path and the node is highlighted. If not found an error message is displayed.
getNamePath(node)	This function returns the path for the specified node, from the tree root. The path is specified by name, that is the names that display in the user interface.
getValuePath(node)	This function returns the path for the specified node, from the tree root. The path is specified by value, that is the value of the nodes that display in the user interface.
getHighlightedNode()	This function returns the highlighted or selected node.



---

## Chapter 9. Search dialogs

Search Dialog is an user interface fragment that provides a common infrastructure for developing search application. With a simple configuration file and a criteria data bean, it generates a common look-and-feel search criteria page for the user. It also allows the user to navigate between the search criteria page and the result page within the dialog, hence able to persist user's inputs automatically.

When implementing a search dialog, you need to validate the criteria inputs, and create a result page to display the search results. A Dynamic List is frequently used to display the search results.

---

### Overview

The following is an overview of how to create a Search Dialog. Detailed steps follow this section.

1. Create a search dialog definition XML file to configure the contents of the search criteria panel and an URL of the result panel.
2. Register the search dialog definition XML file(s) in the component's specific resources.xml file.
3. Create or update resource bundle for national language support.
4. Write a criteria data bean implementing SearchDialogCriteriaBean, if attribute "beanMethod" is defined in the XML file.
5. Write JavaScript functions to validate user inputs, and save additional data that may be required.
6. Write context sensitive help files.
7. Write a result page to process user inputs and display the result list. (Example: Dynamic List)
8. Add a node to Tools User Interface menu.(Optional)

---

### Detailed steps

The following steps are detailed instructions for implementing a Search Dialog.

1. Create a search dialog definition XML file based on `/WC_installdirxml/tools/common/SearchDialog.dtd`, to configure the contents of the criteria panel, and to specify an URL to display the result panel. This file must be in the `/WC_installdirxml/tools/componentname\` directory, where *componentname* is the name of the component to which the search dialog belongs. The following tags are used to configure a Search Dialog.

XML Tag	Description
<code>&lt;searchDialog&gt; &lt;/searchDialog&gt;</code>	The primary element defining a search dialog. The following attributes are supported: <b>resourceBundle</b> (Required) Resource bundle to be used throughout the entire Search Dialog. For example, <code>samples.samplesNLS</code>

XML Tag	Description
<criteriaPanel> </criteriaPanel>	<p>Panel to define the criteria fields and its dictionary.</p> <p><b>databean</b>            (Optional) A fully qualified name of user criteria data bean to provide data definitions on the criteria panel. For example,  <code>com.ibm.commerce.tools.test.SampleSearchCriteriaDataBean</code></p> <p><b>title</b> (Required) Title to be displayed on criteria panel. For example,  <code>searchTitle</code></p> <p><b>description</b>            (Optional) Description to explain this Search Dialog's function. For example,  <code>searchDesc</code></p>
<jsFile/>	<p>External JavaScript file to be included in the Search Dialog. Files defined here are included in the parent frame. Thus, access to these functions require parent. prefixed to function calls to scope them to the parent frame. Multiple JavaScript files are allowed.</p> <p><b>src</b> (Required) Location of the JavaScript file. For example,  <code>/wcs/javascript/tools/samples/sampleSearchDialog.js</code></p>
<jsMessage/>	<p>NL enabled message to be used in JavaScript.</p> <p><b>name</b> (Required) JavaScript variable name to hold the message. For example,  <code>missingFieldMsg</code></p> <p><b>resourceKey</b>            (Required) The resource key name used to retrieve the message from the resource bundle. For example,  <code>missingFieldMsg</code></p>

XML Tag	Description
<field></field>	<p>Criteria field to be displayed on the criteria panel.</p> <p><b>type</b> (Required) The criteria field type. Supported values are hidden, text, select-one, select-multiple, checkbox, radio, and calendar.</p> <p><b>name</b> (Optional) The HTML form input name to represent this criteria field. For example, productNumber</p> <p><b>value</b> (Optional) The HTML form input value for this criteria field. For example, sports0001</p> <p><b>size</b> The HTML form input size for this criteria field. This field is valid only when the type specified is text. For example, 50.</p> <p><b>maxlength</b> (Optional) The HTML form input maxlength for this criteria field. This field is valid only when the type specified is text. For example, 50</p> <p><b>resourceKey</b> (Optional) The resource key name used to display the field name that describes the criteria field. For example, productNumber</p> <p><b>beanMethod</b> (Optional) A Java method name in the databean defined in a &lt;criteriaPanel&gt; element. Teh Tools Framework invokes this method, and expects it to return a String for hidden field type, and Hashtable for field types select-one, select-multi and radio. For example, getMessage</p>
<operatorBox></operatorBox>	<p>A drop-down selection box to represent matching operator. This applies only to a &lt;field&gt; element.</p> <p><b>name</b> (Required) The HTML form selection box name to represent this operator. For example, productNumberOp</p>

XML Tag	Description
<operator/>	<p>An operator entry in the operator drop-down selection box. This applies only to an &lt;operatorBox&gt; element.</p> <p><b>value</b> (Required) The HTML form select option value for this operator. For example, EQ</p> <p><b>resourceKey</b> (Required) The resource key name used to display the text for this operator. For example, exactMatch</p>
<checkbox/>	<p>A checkbox entry field. Only applies to a&lt;field&gt; element if its field type is checkbox</p> <p><b>name</b> (Required) The HTML form checkbox name for this field. For example, displayNum</p> <p><b>value</b> (Required) The HTML form checkbox value for this field. For example, 20</p> <p><b>resourceKey</b> (Required) The resource key name used to display the text for this checkbox. For example, numToDisplay</p>
<yearField/>	<p>4-digit year entry field. Only applies to &lt;field&gt; element if its field type is calendar</p> <p><b>name</b> (Required) The HTML form input name for this year field. For example, startDateYear</p>
<monthField/>	<p>2-digit month entry field. Only applies to &lt;field&gt; element if its field type is calendar</p> <p><b>name</b> (Required) The HTML form input name for this month field. For example, startDateMonth</p>

XML Tag	Description
<dayField/>	<p>2-digit day entry field. Only applies to &lt;field&gt; element if its field type is calendar</p> <p><b>name</b> (Required) The HTML form input name for this day field. For example, startDateDay</p>
<resultPanel/>	<p>Panel to define the search result page.</p> <p><b>url</b> (Required) The URL to display the search result page. For example,  <pre> /webapp/wcs/tools/servlet/ NewDynamicListView?ActionXMLFile=samples. sampleSearchResult&amp;cmd=ResultList </pre> </p> <p><b>target</b> (Optional) The target frame that the URL should be displayed on. By default, it is the dialog's CONTENTS frame. For example, mccccontent</p> <p><b>navigationPanelXML</b> (Optional) A customized XML file for Search Dialog's navigation panel. You may customize buttons and actions according to a dialog's XML file. This attribute will only be effective if the default target frame is not changed. For example, samples.sampleSearchDialogRefine</p>

The following is a sample Search Dialog search dialog definition XML file:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE searchDialog SYSTEM "../common/SearchDialog.dtd">
<searchDialog resourceBundle="samples.samplesNLS">
  <criteriaPanel databean="com.ibm.commerce.tools.test.SampleSearchCriteriaDataBean"
    title="searchTitle" description="searchDesc">

    <jsFile src="/wcs/javascript/tools/samples/sampleSearchDialog.js"/>

    <jsMessage name="invalidCharMsg" resourceKey="invalidCharMsg"/>
    <jsMessage name="missingFieldMsg" resourceKey="missingFieldMsg"/>

    <field type="hidden" name="secret" value="password"/>
    <field type="hidden" name="messageFromServer" beanMethod="getMessage"/>

    <field type="text" name="productNumber" resourceKey="productNumber"
      size="50" maxlength="50">
      <operatorBox name="productNumberFilter">
        <operator resourceKey="equals" value="EQ"/>
        <operator resourceKey="greater" value="GT"/>
        <operator resourceKey="less" value="LT"/>
      </operatorBox>
    </field>

    <field type="select-multiple" name="manufacturer" resourceKey="manufacturer"
      beanMethod="getManufacturer"/>

    <field type="text" name="shortDesc" resourceKey="shortDesc" size="50">
      <operatorBox name="shortDescOperator">
        <operator resourceKey="exactMatch" value="EQ"/>
        <operator resourceKey="containsPhrase" value="LIKE"/>
      </operatorBox>
    </field>
  </criteriaPanel>
</searchDialog>

```

```

        <field type="select-one" name="store" resourceKey="store" beanMethod="getStore"/>

        <field type="radio" name="displayNum" resourceKey="displayNum"
beanMethod="getDisplayNum"/>

        <field type="checkbox">
            <checkbox name="includeBundle" value="true" resourceKey="includeBundle"/>
            <checkbox name="includePackage" value="true" resourceKey="includePackage"/>
        </field>

        <field type="calendar" resourceKey="startDate">
            <yearField name="startDateYear"/>
            <monthField name="startDateMonth"/>
            <dayField name="startDateDay"/>
        </field>

        <field type="calendar" resourceKey="endDate">
            <yearField name="endDateYear"/>
            <monthField name="endDateMonth"/>
            <dayField name="endDateDay"/>
        </field>
    </criteriaPanel>
    <resultPanel url="/webapp/wcs/tools/servlet/tools/samples/DumpRequest.jsp"
        target="mcccontent" navigationPanelXMLFile="samples.sampleSearchDialogRefine"/>
</searchDialog>

```

2. Register the search dialog definition XML file in `/WC_installdir/xml/tools/componentname/resources.xml`, where `componentname` is the component to which the search dialog belongs. This XML file created in step 1 must be registered here in order to function. Create an entry similar as below:

```
<XML name="sampleSearchDialog" file="<subdirectory>/sampleSearchDialog.xml" />
```

The file `resources.xml` is referenced in `instancename.xml`, so you must also update `instancename.xml` for new `resource.xml` files..

3. Create or update any resource bundle (properties) files that might have defined in the XML file, and place this resource bundle file in `/WC_installdir/properties/com/ibm/commerce/tools/componentname/properties`, where `componentname` is the component to which the search dialog belongs. Register this resource bundle file in `/WC_installdir/xml/tools/componentname/resources.xml`, if it is not already registered.

Create an entry similar as below:

```
<resourceBundle name="samplesNLS"
    bundle="com.ibm.commerce.tools.samples.properties.samplesNLS"/>
```

4. If attribute `"beanMethod"` is defined in the XML file, then write a criteria data bean implementing `com.ibm.commerce.tools.common.ui.SearchDialogCriteriaBean`. This data bean is used to populate the displaying text for the criteria panel. For `<jsMessage>`, String type is expected to be returned from the `beanMethod`. For field type of `"select-one"`, `"select-multi"` and `"radio"`, Hashtable type is expected to be returned.

**Note:** `SearchDialogCriteriaBean` is extended from `SmartDataBean`, that means at least `getCommandContext()`, `setCommandContext()`, `getRequestProperties()`, `setRequestProperties()`, and `populate()` methods need to be implemented.

Below shows an example of an implementation of `SearchDialogCriteriaBean`:

```
package com.ibm.commerce.tools.test;

import java.util.Hashtable;
import com.ibm.commerce.command.CommandContext;
```

```

import com.ibm.commerce.datatype.TypedProperty;
import com.ibm.commerce.tools.common.ui.SearchDialogCriteriaBean;

public class SampleSearchCriteriaDataBean implements SearchDialogCriteriaBean {
    protected CommandContext commandContext = null;
    protected TypedProperty requestProperties = null;

    public CommandContext getCommandContext() {
        return commandContext;
    }

    public void setCommandContext(CommandContext cc) {
        commandContext = cc;
    }

    public TypedProperty getRequestProperties() {
        return requestProperties;
    }

    public void setRequestProperties(TypedProperty reqProp) {
        requestProperties = reqProp;
    }

    public void populate() {
    }

    public Hashtable getManufacturer() {
        Hashtable hash = new Hashtable();
        hash.put("IBM", "IBM");
        return hash;
    }

    public Hashtable getDisplayNum() {
        Hashtable hash = new Hashtable();
        hash.put("10", "10");
        hash.put("20", "20");
        hash.put("30", "30");
        return hash;
    }

    public Hashtable getStore() {
        Hashtable hash = new Hashtable();
        hash.put("demoStore", "Demo Store");
        hash.put("protomartB2B", "protomart B2B");
        hash.put("protomartB2C", "protomart B2C");
        return hash;
    }

    public String getMessage() {
        return "message from the server";
    }
}

```

5. Write a JavaScript file for the criteria panel. In this JavaScript file (as specified in the XML file from step 1), the following functions should be implemented:

Function Name	Return Type	Description
userSavePanelData()	N/A	All the criteria input fields will be saved automatically. If additional data are required, programmers may save their data in this function. Note that this function is not mandatory to implement.
userValidatePanelData()	boolean	Programmers may validate all the required criteria fields in this function. If it is valid, return a true value, otherwise return a false value.

6. Write context sensitive help files. For more information see "Adding context-sensitive help" in chapter 7.

7. Write a JSP (normally, it would be a Dynamic List) and necessary data beans to handle the criteria fields passed by Tools Framework. There are two ways the programmers can retrieve the criteria fields. One is from the request parameters, and the other way is through the request properties object.
8. Optionally add a node to Tools User Interface Center menu XML file (for example, `/WC_installdir/xml/tools/common/CommerceAccelerator.xml`) for the newly created search dialog. Below is an example of how to configure to launch a search dialog from tools user interface menu:

```
<node name="sampleSearchDialog"
url="/webapp/wcs/tools/servlet/SearchDialogView?ActionXMLFile=
samples.sampleSearchDialog"/>
```

---

## Navigation

Search dialogs present users with one or more navigation options, next, previous, finish, or cancel. These options are presented as buttons in the navigation frame at the bottom of the content window. If included, they behave according to the following guidelines:

### Search

User clicks on Search button to perform a search request.

1. Check if function `userSavePanelData()` exists, and execute it if it does exist.
2. Execute function `userValidatePanelData()` if it exists. If invalid input is determined, programmers may pop up an alert message to notify the user in this function, and return a false value. Otherwise, simply return a true value at the end of the function.
3. If false value is returned, criteria panel will still be displayed with all the entered input fields pre-populated. If true value is returned, user will be redirected to the result panel.

### Refine

User clicks on Refine button to go back to the criteria panel. When the user is brought to the result panel, by default, the user has an option to click on the Refine button to go back to the criteria panel with all the entered criteria fields pre-populated.

### Cancel

Displays a cancel confirmation dialog. If the user clicks **OK**, `submitCancelHandler()` then the parent frame's cancel method run.

---

## Customizations

### Result Navigation Buttons

By default, there are two buttons, Refine and Cancel, on the result navigation panel. These buttons and their actions can be customized by creating an partial dialog XML file. Below shows an example of a customized XML file for the result navigation panel:

```
<?xml version="1.0"?>
<dialog resourceBundle="samples.samplesNLS"
        windowTitle=""
        finishConfirmation=""
        cancelConfirmation="cancelConfirmation"
        finishURL="DialogNavigation">
    <button name="custom" action="customAction()"/>
    <button name="refine" action="refineAction()"/>
</dialog>
```



### Result Navigation Panel

By default, there is a navigation panel at the bottom of the result panel. If this navigation is not required, it can be turned off by simply assigning value "mcccontent" to attribute target of <resultPanel> in the XML file as below:

```
<resultPanel url="/webapp/wcs/tools/servlet/tools/samples/DumpRequest.jsp"
  target="mcccontent"
  navigationPanelXMLFile="samples.sampleSearchDialogRefine"/>
```



---

## **Part 2. Element chaining and wizard branching**

This section describes element chaining and wizard branching, two methods you can use to further customize your Tools User Interface elements.



---

## Chapter 10. Element chaining

Element chaining refers to how wizard, notebook, and dialog container elements can be linked together to accomplish one task. Information about element chaining follows:

- There is no limit on the number of container elements that can be chained together.
- Users can go back to any previous container element by clicking on that item in the page history.
- To return to the previous container element, call the `top.goBack()` function from the WebSphere Commerce Accelerator's top frame. This normally happens when a user clicks the OK or Finish button in the chained container element.
- After returning from the chained container element, the originating container element displays the same panel from which it launched the chained container element. For example, if a user is in the second panel of a wizard and chained container element launches, the second panel of the originating wizard displays after the user completes the chained container element.
- Optionally, a container element can save its current state, without going back to the server, before it calls another one. For example, a user entering information in the second panel of an unfinished wizard can go to another wizard, come back, and find all of their information still there.
- Optionally, the chained container element can return a variable to the originating container element. This is frequently the hashtable value, `model`. This name is used across all Tools User Interface elements.

---

### Creating an element chain

This example uses wizards, however, it is possible to chain any of wizards, dialogs, and notebooks together. To chain two wizards together, there are three stages for which you must add code:

1. In the originating wizard, immediately before launching the chained wizard
2. In the chained wizard, immediately before returning to the originating wizard
3. Immediately after returning to the originating wizard

#### **In the originating wizard, immediately before launching the chained wizard**

1. If necessary, save data from the first wizard:
  - a. If the first container element needs to save its current state information, call the following JavaScript function: `top.saveModel(parent.model);`
  - b. To save the current panel name, call the following JavaScript function:  
`top.setReturningPanel(String panelName);`

If no panel is specified, the first panel is used as the default. This function sets a URL parameter called `startingPage`, used to load the wizard starting from a user specified panel. This parameter is used to construct a link, which when clicked from the page history, leads to the correct panel in a wizard.

- c. If you have other data to save, use the following JavaScript function:  
`top.saveData(Object data, String slotName);`

Where *data* is the JavaScript variable you want to save, *slotName* is a unique name needed later as a key to retrieve the data. `slotName` works as a hash

key and its value is overwritten without warning. Caution: The slot name `model` is a reserved keyword and should not be used. If it is used, the information saved using `top.saveModel()` is overwritten.

2. Launch the second wizard:

To launch the second wizard, call the following JavaScript function: `top.setContent(String text, String link, Boolean value, Object parameter)`; where *text* is the text displayed in the page history, for example, the title of the second wizard. The value of *link* is the second wizard's launching URL. If *value* is true, this adds a new item to the page history. *parameter* is used only if your link contains national-language characters, which may become corrupt using direct URL location replacement. If this parameter is used, the tools framework dynamically generates a form based on this parameter object and submits the URL parameters in name-value-pair format.

**In the chained wizard, immediately before returning to the originating wizard**

1. Optionally, call the following JavaScript function to return a JavaScript value (normally the model of the wizard) to the first wizard. This data is passed under the first wizard's model object. `top.sendBackData(Object data, String slotName)`; where *slotName* is a unique name used to retrieve the data from the first wizard.

To retrieve data from previous wizards in the chain, use the following functions: `top.getModel(int stepsBack)`; `top.getData(String slotName, int stepsBack)`; Where *stepsBack* is the number of wizards previous to the current one which contain the data being requested (0 for the current wizard, 1 for the previous wizard.) This parameter is optional, and the default value is 0.

2. To return to the first wizard, call the following JavaScript function: `top.goBack(int stepsBack)`; The *stepsBack* parameter is optional, and the default value is 1. This function is equivalent to clicking the previous item on the page history.

**Immediately after returning to the originating wizard**

The tools framework automatically refreshes the model variable to your saved state, set in stage 1, part a and go to the returning panel, set in step 1 part b. Optionally, you can call the following JavaScript function to get the "model" data sent back from the 2nd wizard, set in stage 2, part a. `top.getData(String slotName)`; where *slotName* is the unique name where you send back the data in stage 2, part a.

---

## Chapter 11. Wizard branching

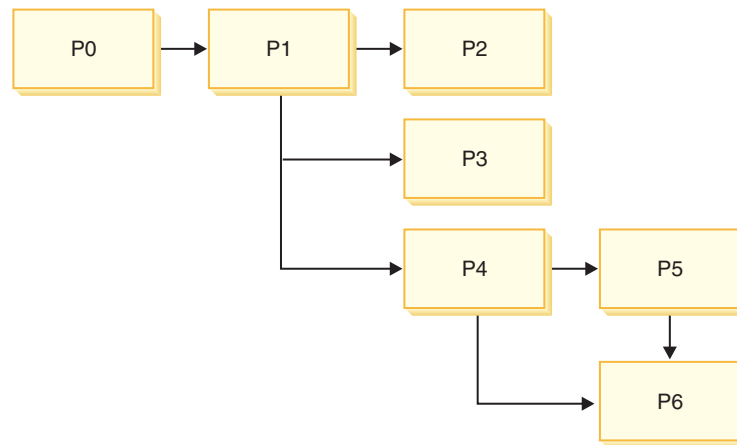
Wizard branching refers to how wizard panel flow can be altered depending on data entered in previous panels.

Branching is controlled through the wizard's XML file and JavaScript in the panel JSP pages. Each <panel> tag in the XML file has two attributes: `hasBranch` and `hasNext`. To indicate that a panel is a branching point, set `hasBranch=YES`. Within that panel's JSP page, indicate which branch to follow by using the command `parent.setNextBranch(String branch_panel_name)`, where `branch_panel_name` is the name of the next panel to display. To indicate that a panel is the end of a branch, set `hasNext=NO`.

---

### Wizard branching example

The following example describes the creation of branching wizards. The diagram describes the panel flow. Depending on data from the user, panel P1 may be followed by P2, P3, or P4. Similarly, panel P4 may be followed by P5 or P6. The table of contents displays the panels as the user traverses the branches. In the above example, only P0 and P1 display initially. After the P4 branch is selected P0, P1, and P4 appear. Later, if no `setNextBranch()` is called in P4, P5 is selected as the default next branch path. Notice that P5 does not have the "hasNext=NO" tag, so the wizard does not stop there, it goes on and stops at P6, the last panel.



The following code sample would appear in the wizard XML file:

```
<panel name="P0" url="BranchDiscountWelcomeView"
helpKey="MC.discount.type.Help">

<panel name="P1" url="BranchDiscountWizTypeView"
helpKey="MC.discount.type.Help"hasBranch="YES"> //this is a branching
panel

<panel name="P2"
url="BranchDiscountWizOrderView"helpKey="MC.discount.order.Help"
hasTab="NO" hasNext="NO">
```

```
<panel name="P3" url="BranchDiscountWizPrdView"
helpKey="MC.discount.product.Help" hasTab="NO" hasNext="NO">

<panel name="P4" url="BranchDiscountWizCusTypeView"
helpKey="MC.discount.customtype.Help" hasTab="NO" hasBranch="YES"> //another
branching panel

<panel name="P5" url="BranchDiscountWizCusView"
helpKey="MC.discount.custom.Help" hasTab="NO">

<panel name="P6" url="BranchDiscountWizRangesView"
helpKey="MC.discount.range.Help" hasTab="NO" hasFinish="YES">
```



---

## Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described 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  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.*

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

*IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan*

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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 Canada Ltd.  
Office of the Lab Director  
8200 Warden Avenue  
Markham, Ontario  
L6G 1C7  
Canada*

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

---

## Trademarks

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

400	AIX	IBM
iSeries	OS/400	WebSphere

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

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

Other company, product and service names may be trademarks or service marks of others.





Printed in USA