BEA WebLogic Workshop™ Help

Version 8.1 SP4
December 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Table of Contents

Table of Contents

Table of Contents

WebLogic Workshop Extension Development Kit

In addition to simplifying the creation of J2EE applications, WebLogic Workshop offers a rich extensibility model for developers and ISVs to integrate their products and services directly into the development environment. Using Java controls, templates, IDE extensions, and tag library extensions, developers have broad flexibility in working with Workshop. Some example WebLogic Workshop extension and application integration ideas are listed below:

- Custom application designers and windows which reside in the WebLogic Workshop IDE
- Custom controls that interface with backend resources including databases, systems, applications, and business logic
- Tool bar icons and custom menus that launch external helper applications
- New project and file types not native to WebLogic workshop
- JSP Tag Library extensions for use in web applications

The following sections provide more information about each of the major areas in which you can author extensions.

Topics Included in This Section

Extending WebLogic Workshop

Provides an overview of the kinds of extensions you can build for integrating with WebLogic Workshop.

What's New in the Extension Development Kit?

Gives an overview of recent changes.

Extension Services in WebLogic Workshop

Lists and describes the services on which extensions are built, providing links to corresponding parts of the API.

Developing Advanced Controls

Describes the model for writing complex Java controls that provide custom user interface.

Extending the WebLogic Workshop IDE

Provides an introduction to creating menus, dialogs, buttons, and other extensions to the IDE.

Developing Tag Library Extensions

Describes how you can extend custom tag libraries so that they integrate seamlessly with the WebLogic Workshop design environment.

Help Authoring Guide

Describes how you can write Java control documentation that integrates with WebLogic Workshop's existing content.

Extension Samples

Lists the samples included with this kit, along with suggestions for getting set up.

Debugging Extensions

Provides guidelines for setting up debugging properties for extension projects.

Getting Started with UI Programming

Gives a brief introduction to Java Swing, a framework for developing user interfaces, along with links to resources that give more information.

Related Topics

None.

Extending WebLogic Workshop

WebLogic Workshop provides an integrated environment for building web applications efficiently. Through the Workshop extensibility model, you can enhance this environment to provide support for application and productivity needs that are not yet addressed by WebLogic Workshop. In effect, you can use the the extensibility model to integrate new functionality whose focus is, in the spirit of WebLogic Workshop itself, to enable developers to be productive quickly. As it happens, WebLogic Workshop is itself the sum of several extensions that combine to present a unified interface.

Note: If you're new to Swing, the Java technology that you'll use to build user interface components for WebLogic Workshop extensions, you might be interested in the links available at Getting Started with UI Programming.

The model exposes four main areas for extension−building. Extensions built in these areas are typically used either as application components or as productivity enhancements. By working in these areas you can provide:

- Controls that expose complex functionality in simple ways through customizable interfaces, properties, and control−specific user interface components. Controls are used a components of applications built with WebLogic Workshop.
- Templates that set up the initial structure for new projects, adding dependencies that might otherwise have to be imported or created, and generating files that help the project's user get started quickly. Templates are productivity enhancements.
- Support for custom JSP tag libraries through tag library extensions that integrate the tags seamlessly with the Workshop design−time experience. Tag library extensions are productivity enhancements.
- Components that augment the IDE with toolbar buttons and menus, views in dockable frames, additional document and project types, editing support for additional kinds of source code, source control integration, debugging data views, and so on. IDE extensions are productivity enhancements.
- Help that is specific to your extension and fully integrated with the WebLogic Workshop help system. Help extensions are productivity enhancements.

The following sections provide more information about each of these areas, along with links to further information in this guide.

Advanced Java Controls

Templates

JSP Tag Library Extensions

IDE Extensions

Help Integration

Advanced Java Controls

Java controls are a core part of the WebLogic Workshop programming framework. Even simple controls are ideal for encapsulating application logic, particularly when that logic coordinates the work of multiple resources. One of the most compelling reasons for building Java controls is to provide simplified access to resources that may otherwise require complex code to integrate into an application.

More complex controls, such as those built with the help of this guide, build on the simpler control support described in the core WebLogic Workshop documentation. These advanced controls support customizable or dynamically−created programming interfaces, design−time user interface for inserting the control and setting its properties, and sophisticated asynchronous operation.

When you build advanced controls, you develop your sources within a control project whose result is a JAR or ZIP file that is delivered to users. WebLogic Workshop provides control installation support that automatically copies control components, documentation, and samples to appropriate locations in the user's installation. Once added to the user's WebLogic Workshop installation, new controls are available in the IDE from palettes and menus.

To get started building advanced controls, see Developing Advanced Controls. For an overview of control samples, see Advanced Control Samples.

Templates

A template gets a project's or application's user quickly to the point of writing application−specific code. For example, if you create a new web application project in WebLogic Workshop, the IDE uses a template to ensure that your new project starts out with the complement of JAR files needed to get the application running. If you're extending Workshop with a new project type, a template is a great way to help the project's users be productive quickly. A template can set up the project's initial file structure and copy into the new project any files that may be dependencies.

At a high level, the tasks involved in building a template include writing a template.xml file that specifies the names and locations of files included with the projects or applications based on the template. The template.xml file also specifies basic user interface to accompany the template, such as strings and categories displayed in the New Project dialog. You include the template.xml file and any other files that should be added to applications based on the template such as sources, JARs, and so on in a ZIP file that you deliver to the template's user.

To get started building templates, see Developing Application and Project Templates.

JSP Tag Library Extensions

Applications built with WebLogic Workshop support custom JSP tags, just as those applications support other J2EE application components. WebLogic Workshop's specific ease−of−use approach to these tags at design time is possible by integrating the tag library into the IDE through a tag library extension. This kind of extension can provide support for IDE features such as the data and design palettes, Source View error checking (those red squiggle underlines), specific tag rendering in Design View, and integrated documentation.

In other words, when building a custom tag library for use in WebLogic Workshop, you also provide a corresponding extension for seamless design−time support. The core of this extension is a TLDX file whose structure is similar to the standard TLD file. The file specifies design−time characteristics of each tag and attribute defined in the TLD file. These characteristics may require Java classes that you build and include with the extension.

For more information on building tag library extensions, see Developing Tag Library Extensions. For an overview of relevant samples, see Tag Library Extension Samples.

IDE Extensions

You might argue that extensions in each of the categories described here extend the IDE in some way, and you'd have a point. Generally speaking, though, IDE extensions are built on a framework (available through the majority of the extensibility API) whose support is more broadly applicable, rather than specific to particular component types such as controls or JSP tags. If you're planning to add menus, toolbars, dockable windows, and so on to the IDE, an IDE extension is what you're looking for.

Likewise, if you want to enhance debugging or add support for new kinds of documents, you'd do those through an IDE extension. WebLogic Workshop's productivity benefits are in part due to its special handling of certain file types, such as JWS, JSP, or JCS files. Each of these supports specific actions. Through document type IDE extensions you can implement specific support for your own file types.

You deliver an IDE extension in a JAR file. The file must include an extension.xml that describes the extension's characteristics for the IDE. To be visible to the IDE, the extension JAR must be deployed to the <WORKSHOP_HOME>/extensions folder of the user's installation.

For more information on writing IDE extensions, see Developing IDE Extensions. For a description of samples included with this development kit, see IDE Extension Samples.

Help Integration

WebLogic Workshop includes a documentation system that enables you to add new content to support your extensions. By following guidelines for delivering help, your documentation is not only integrated with the existing help table of contents and search, but is available when the user presses F1 when your API or custom JSP tag is selected in Source or Design View.

For more information on delivering integrated documentation, see Help Authoring Guide.

Related Topics

None.

What's New in the Extension Development Kit?

Release: WebLogic Workshop version 8.1 Service Pack 3

What's New for Java Controls?

Automatic Installation for Controls and Their Documentation

Workshop SP2 introduced automatic installation support for packaged 3rd−party controls. To take advantage of this support, the control must be packaged into a ZIP file with a specified format. This ZIP file may include control help files and control samples. Workshop also now supports a "control stub" file that appears to the user as an available control. If requested by the user, Workshop will initiate a download of a control deliverable from a URL specified in the stub. See the help topic Packaging Controls for details on the new control packaging and installation support.

Control help files can now be included in this ZIP file and will participate in the automatic installation process. The 3rd−party directory structure for help files has changed from 8.1.1 to accomodate this feature. Also, the toc.xml entries for 3rd party help are now under the "extensions" anchor. Please see the topic Help Authoring Guide for more details on integrating help in Workshop.

Control Deliverable Format

Control deliverables are ZIP files expected to contain three top−level directories:

- controls contains the control implementation JAR, and any dependency JARs
- help documentation and Javadoc accompanying the control
- samples samples that use the control

This control deliverable ZIP file is the same as what is produced by starting with a Control Deliverable project, adding some help and samples content files to the appropriate directories in the tree, and using the Build Control Deliverable command. By default, any JARs found in APP−INF/lib at build time are assumed to be required for the control and are bundled into the ZIP.

Note that only a *single* control implementation JAR can live in the controls folder of the ZIP file. *Multiple* controls, however, may be bundled into the control implementation JAR. For each application, when a user first tries to install the 3rd−party control into it, the control implementation JAR and all its dependency JARs will be copied to the application's Libraries folder.

The help directory *must* be organized in the following format:

```
help
  doc
    en (for English content)
      partners
        <vendor name>
          java−class
          javadoc−tag
```

```
    ja (for Japanese content)
      partners
          ...
```

The samples directory *must* be organized in the following format:

```
samples
  partners
    <vendor name>
      sample1
```

What's New for Integrating Documentation?

In order for your help files to participate fully in the WebLogic Workshop frameset, your topics must be include references to several JavaScript files and functions, as well as CSS stylesheets. To learn more about these requirements, see the Help Authoring Guide. Also, the Help Test Kit provides a way for you to test your help topics to ensure that they will integrate smoothly with WebLogic Workshop documentation. For more on the Kit, see the Extensibilty Portal.

Related Topics

None.

Extension Services in WebLogic Workshop

When you build an extension, you're actually extending one or more WebLogic Workshop *services*. These services provide the support for the various functionality areas that WebLogic Workshop provides. Each is also represented in the extensibility API. Some of these services including the frame service, action service, and preferences service might immediately call to mind types of extensions you can write: frame view, action, and preferences extensions. Other provide support for things extensions do.

The following lists the services exposed by WebLogic Workshop, along with the API representation for each.

Action service (com.bea.ide.actions.ActionSvc)

Supports action extensions, such as menus and popups. For more on building action extensions, see Adding Menus and Toolbar Buttons.

Ant service (com.bea.ide.build.AntSvc)

Supports Ant operations.

Asynchronous task service (com.bea.ide.core.asynctask.AsyncTaskSvc)

Provides a general mechanism for running tasks asynchronously.

Browser service (com.bea.ide.ui.browser.BrowserSvc)

Supports invoking a browser.

Compiler service (com.bea.ide.sourceeditor.compiler.CompilerSvc)

Provides a general mechanism for communication between the IDE and Javelin, BEA's compiler framework.

Control service (com.bea.ide.control.ControlSvc)

Supports controls and control containers.

Data palette service (com.bea.ide.ui.palette.DataPaletteSvc)

Provides support for populating the data palette for existing document types.

Data transfer service (com.bea.ide.core.DataTransferSvc)

Supports copy/paste and drag/drop operations.

Debugging service (com.bea.ide.debug.DebugSvc)

Supports debugging operations, such as setting and removing breakpoints, stepping through code, and so on.

Document service (com.bea.ide.document.DocumentSvc)

Supports obtaining and displaying documents in the IDE.

Editor service (com.bea.ide.sourceeditor.EditorSvc)

Supports actions in Source View, such as creating, obtaining, and switching among views.

File service (com.bea.ide.filesystem.FileSvc)

Provides a general service to operate on "files" represented by the IFile interface.

File system service (com.bea.ide.filesystem.FileSystemSvc)

Keeps a list of all the files of interest to the file system and will generate notifications of changes to those files to interested listeners.

Frame service (com.bea.ide.ui.frame.FrameSvc)

Provides access to the main frame of the application, and its associated docking views. For more on building a frame extension, see Adding Dockable Frames.

Help service (com.bea.ide.ui.help.HelpSvc)

Used to find and display context−sensitive help topics from the IDE.

HTTP service (com.bea.ide.core.HttpSvc)

Supports communications over HTTP.

Message service (com.bea.ide.core.MessageSvc)

Supports displaying messages to the user.

Navigation service (com.bea.ide.core.navigation.NavigationSvc)

Supports forward and backward navigation between points as set, such as from the file the user is currently viewing back to the file they were viewing previously.

Output service (com.bea.ide.ui.output.OutputSvc)

Supports sending messages to the tabbed "Output" window provide by the IDE.

Palette action service (com.bea.ide.jspdesigner.PaletteActionSvc)

Provides access to the JSP designer palette.

Preferences service (com.bea.ide.core.PreferencesSvc)

Provides access to preferences through default value lookup and quick access to related preferences. For more on building preferences extensions, see Adding Support for Preferences.

Resource service (com.bea.ide.core.ResourceSvc)

Provides support for accessing string and image resources stored in an extension JAR file.

Run service (com.bea.ide.workspace.RunSvc)

Controls the behavior and the enabled states of the Start, Start Without Debugging and Stop buttons.

Server service (com.bea.ide.workspace.ServerSvc)

Provides access to the J2EE server that the IDE is being used to develop against.

Settings service (com.bea.ide.workspace.SettingsSvc)

Supports displaying Properties dialogs.

Source control service (com.bea.ide.sourcecontrol.SourceControlSvc)

Supports interacting with a source control system.

Workspace service (com.bea.ide.workspace.WorkspaceSvc)

Manages the user application open in the IDE.

Related Topics

None.

Developing Advanced Controls

Java controls are a key part of the programming model for those writing web applications with WebLogic Workshop. Controls can be used to capture the bulk of an application's business logic, encapsulate access to resources with a simplified interface for developers, and so on.

This guide is intended for those building more sophisticated packaged controls. WebLogic Workshop provides rich run−time support and an API for building controls that support design−time features through IDE extensions, customization through JCX files, and asynchronous operation.

A Bit of Terminology

The content in this guide assumes familiarity with the following control−related terms.

- Platform and custom controls
  Controls installed with the IDE are known as *platform controls*; additional controls built by application developers, independent software vendors, and others are known as *custom controls*.
- Local and packaged controls
  Simple custom controls may be written as *local controls*. A control is local when it is used as source in the same project as its container. This is a simple way for application developers to partition business logic. However, local controls provide limited access to the features available to controls.

  In contrast, a *packaged control* is built in the context of a control project. All controls that use the features described in this control developer's kit must be built as packaged controls.
- Regular and extensible controls
  A *regular* control does not generate a JCX file; an *extensible* control does. Because a JCX file is generated, an extensible control supports design−time customization. For example, an extensible control can support the user's changing the control's interface by adding, removing, and editing methods and callbacks. It can also be customized in a way that is not editable by the user; an example is the EJB control, whose interface is defined by the EJB it will be accessing.

  For more information on creating extensible controls, see Creating Extensible Controls.
- Control stubs and control deliverables

  A *control deliverable* is a JAR or ZIP file that contains all the files −− implementation, GIFs, annotation XML files, and so on −− required to use the control. A *control stub* is a JAR file that contains merely the information needed to make the control's availability visible to users, along with a URL from which the IDE may download the control the first time the user chooses to add it to an application. Together, these are known as *available controls*. This is a just−in−time approach to control delivery and use. For more information, see Packaging Controls for Installation.

Content of this Guide

The material in this guide is intended for those building packaged custom controls that use the advanced features available for Java controls. This guide is intended to complement the material contained in the core WebLogic Workshop documentation (under "Working with Java Controls"). If you're looking for information about Java control concepts that's not covered here, it's likely you'll find it there.

In building an advanced Java control, the tasks you'll perform are likely to include:

- If you're building controls that will be distributed to multiple users, you should probably start with a control deliverable project. For more information, see Control Deliverable Projects.
- Set up debugging properties for your control project so that you can debug IDE extension components. For more information, see Setting Up for Debugging Advanced Controls.
- Develop your control's core logic in a JCS file. For more information, see the core WebLogic Workshop documentation.
- Set JCS file properties to configure features such as security roles, the name of your custom insert UI, the name of the group under which your control appears in menus, and so on. For more information, see the core WebLogic Workshop documentation.
- Handle life cycle callback events for efficient management of resource dependencies. For more information, see Handling Control Life Cycle Events.
- Define control properties in an annotations XML file, then connect the file to your JCS with the control−tags property. For more information, see Control Properties Overview.
- Build a class that validates your custom property attributes in both the Property Editor and in source code. For more information, see Creating Attribute Editors and Validators.
- Extend the IDE with a custom property editing dialog that validates the property values. For more information, see Creating Attribute Editors and Validators.
- Implement your control so that it is extensible, generating a JCX file when inserted. This enables design−time customization. For more information, see Creating Extensible Controls.
- Extend the IDE with a custom insert dialog designed to prompt users for properties specific to your control. For more information, see Creating Dialogs and Wizards for Inserting Controls.
- Write code to handle user actions (and your control's appearance) in Design View. For more information, see Controlling Appearance and Handling Actions in Design View.
- Package your control according to WebLogic Workshop specifications, making it easier for users to find and use the controls. For more information, see Packaging Controls for Installation.

Related Topics

None.

Control Deliverable Projects

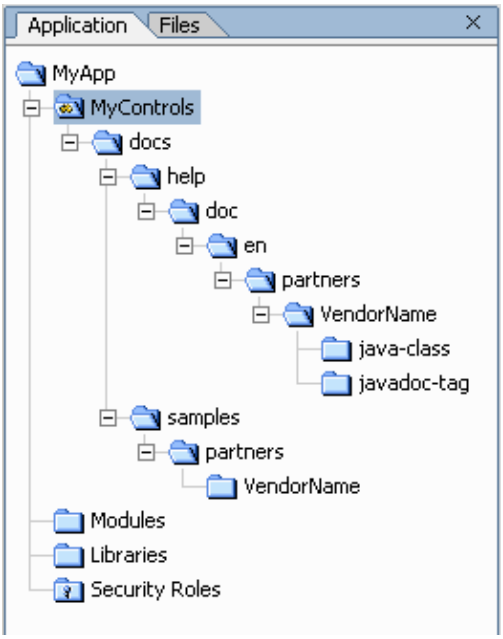
You use a control deliverable project when you want to build Java controls that will be distributed to multiple users. A control deliverable project supports automatic integration of help and samples you provide with your Java control.

The difference is that a control deliverable project includes a "docs" folder with "help" and "samples" subfolders that include folder hierarchies to support automatic integration with the user's WebLogic Workshop installation. When you are ready to test or distribute your control deliverable, you can easily build a control deliverable by right−clicking the project name, then clicking Build Control Deliverable.

To create a control deliverable project

1. Right−click the application, point to ***New***, then click ***Project***.
2. In the ***New Project*** dialog, in the left pane, click ***All***. In the right pane, click ***Control Deliverable Project***.
3. In the ***Project*** name box, enter a name for your new project, then click ***Create***.

WebLogic Workshop will create a new project with the name you gave it. If you expand the project folders to see it contents, you'll notice the docs folder and its subfolders. For example, if you created a project called "MyControls", you'll see something like this:



After creating the project, you should be sure to rename the VendorName folders to a name that clearly describes your company and product. Keep in mind that your control's users may have many controls installed from various sources.

Documentation and samples you add should go into the hierarchy provided. For more information on including documentation, see Help Authoring Guide.

When you build a control deliverable from this project, the resulting ZIP file won't contain the docs folder. Its top−level contents will include the help folder and its subfolders, the samples folder and its subfolders, along

with your control implementation JAR.

For information about how a control deliverable is used in WebLogic Workshop, see Packaging Controls for Installation.

Help in Languages Other than English

Note that the "en" folder is a <language> folder as described in Help Authoring Guide. If you translate your documentation into other languages, you must create a different <language> folder for each, along with child folders as shown here. For example, if you provide Japanese documentation, you would have a hierarchy such as docs/help/doc/ja/partners/<vendor_name>/*

Related Topics

None.

Controlling Appearance and Handling Actions in Design View

WebLogic Workshop provides several ways for you to specify your control's appearance in the IDE. These are exposed in two ways:

- An "editor support" class you provide for specifying control behavior in Design View.
- Properties of the control's JCS file, mostly to specify control characteristics in the IDE when the control is not yet part of a design.

This topic describes these two areas.

Providing Editor Support

An "editor support" class is a kind of traffic cop that directs the IDE to specific responses for events in Design View. From the IDE's perspective, control characteristics in Design View are known as *behaviors*. You write code to give specifics for behaviors by implementing an editor support class one that implements the EditorSupport interface. An easy way to provide an editor support implementation is to use or extend the DefaultEditorSupport class. You connect your editor support implementation to the IDE by specifying its fully−qualified name in control's the property annotation XML file. This name is the value of the editor−class attribute of the control−tags element.

Implementing the EditorSupport Interface

The EditorSupport interface defines constants for the behaviors supported by the IDE. It also has a single method, getBehavior, that you can implement to tell the IDE the details for each of the behaviors. You implement the getBehavior method in your editor support class, and the IDE calls this method to retrieve your specific instructions for the current behavior.

When calling the getBehavior method, the IDE passes in two arguments. The first of these is an EditorSupport interface constant representing the behavior, such as EditorSupport.BEHAVIOR_EDIT_METHOD or EditorSupport.BEHAVIOR_ATTRIBUTE_VALIDATOR.

The second argument is an instance of an interface that extends the ControlBehaviorContext interface. You use this context interface to discover specifics about the control "piece" (method, control interface, control extension, and so on) that the control's user is currently interacting with. Your getBehavior implementation tests for a combination of these, then returns a response based on the results. The following lists the EditorSupport constants passed in by the IDE, along with the possible contexts you could test for. Note that each context object provides methods for getting information specific to that context. For each behavior, a default return value is used if your code doesn't handle the specific behavior/context pair.

BEHAVIOR_ICONS

- Received when: The IDE is looking for icons to display on method arrows.
- Context: ControlMethod instance.
- Return: A String containing a path to a GIF file to display on the arrow.
- Default: No icon.

BEHAVIOR_MAINICON

- Received when: The IDE is looking for the method arrow icon that launches the main editor associated with the method. For example, double−clicking this icon might launch a mapping dialog or expression editor.
- Context: ControlMethod instance.
- Return: A String containing a path to a GIF file to display on the arrow.
- Default: No icon.

BEHAVIOR_EDIT_METHOD

- Received when: The IDE is asking whether the control's user can add, remove, or change method signatures on a control extension (JCX) file.
- Context: ControlExtensionInstance or ControlExtensionInterface instance if it appears the user is attempting to add a new method; ControlMethod instance if the user is removing a method.
- Return: Boolean.TRUE to indicate that the user may add and remove the method; otherwise, Boolean.FALSE. You can use the context object to further customize the response. Returning Boolean.FALSE indicates that the control's method set is defined outside the user's control; in other words, your control code creates the interface when the user adds the control to their design.
- Default: Boolean.TRUE

BEHAVIOR_EDIT_CALLBACK

- Received when: The IDE is asking whether the control's user can add, remove, or change callback signatures on a control extension (JCX) file.
- Context: ControlExtensionInstance or ControlExtensionInterface instance if it appears the user is attempting to add a new method; ControlMethod instance if the user is removing a method.
- Return: Boolean.TRUE to indicate that the user may add and remove the callback; otherwise, Boolean.FALSE. You can use the context object to further customize the response. Returning Boolean.FALSE indicates that the control's callback set is defined outside the user's control; in other words, your control code creates the interface when the user adds the control to their design.
- Default: Boolean.TRUE

BEHAVIOR_ATTRIBUTE_EDITOR

- Received when: The IDE is asking for an object to use to edit an attribute value. For example, the user might have clicked the ellipses (...) for the attribute in the Property Editor.
- Context: ControlAttribute instance.
- Return: An instance of an object that implements AttributeEditorSimple. You implement AttributeEditorSimple (or AttributeEditorWizard, which extends it) to provide an attribute editor user interface. In your implementation of getBehavior, you will typically create an instance of that class, passing it the current value of the attribute the user is asking to edit. When the user has finished editing the attribute, the IDE will retrieve the new value from the getNewAttributeValue method in that class.
- Default: The default editor for the attribute's type. The attribute's type is specified in the tag XML file that defines property tags and attributes for the control.

BEHAVIOR_ATTRIBUTE_VALIDATOR

- Received when: The IDE is asking for an object to use to validate an attribute value. For example, the user might have edited attribute in−place in the Property Editor.
- Context: ControlAttribute instance.

- Return: An instance of an object that implements ValidateAttribute. You implement ValidateAttribute to validate the value that the user has just attempted to set for the attribute. In particular, your implementation of the validateDuringEdit method will contain the code that executes for this behavior.
  Note that you will typically return a value for this behavior only when you haven't supplied an attribute editor (in other words, when you're not returning an editor for the BEHAVIOR_ATTRIBUTE_EDITOR behavior). That's because when you have an attribute editor, that class will have been used to edit the value, and you will probably have called your validator class from its code.

- Default: Any text is allowed.
  Note: Be aware that there are two mechanisms for validating attribute values. The value you return for the BEHAVIOR_ATTRIBUTE_VALIDATOR behavior defines the validator to use when the user has chosen to edit the attribute's value and it needs validating. In contrast, you can specify a class that should be used by the compiler for validating the value in source code, in the property annotation. You specify that class in the property tags XML file, in the class−name attribute of a custom attribute type. In other words, that's the class WebLogic Workshop uses to know when to display red squiggles for the tag's code. You can define a single class for both kinds of validation, however, as described in Creating Attribute Editors.

The following getBehavior implementation example illustrates handling for a few of these behaviors.

```
public Object getBehavior(String behavior, ControlBehaviorContext context)
{
    Object response = null;
    /*
     * If the user asks to edit a JCX method, return true.
     */
    if (behavior.equals(EditorSupport.BEHAVIOR_EDIT_METHOD))
    {
        response = Boolean.TRUE;
    }
    /*
     * If the user asks to edit a callback, return false. XQuery JCX files do
     * not support callbacks.
     */
    else if (behavior.equals(EditorSupport.BEHAVIOR_EDIT_CALLBACK))
    {
        response = Boolean.FALSE;
    }
    /*
     * If the user clicks an edit ... in the Property Editor, and if the attribute
     * corresponding to the ... is "expression", return an instance of the expression
     * edit dialog box.
     */
    else if (behavior.equals(EditorSupport.BEHAVIOR_ATTRIBUTE_EDITOR))
    {
        if (context instanceof ControlAttribute && ((ControlAttribute)context).getName().equals
        {
            // The attribute editor dialog is constructed with the attribute's current value.
            response = new QueryExpressionEditorSimple(((ControlAttribute)context).getValue());
        }
    }
    return response;
}
```

Extending the DefaultEditorSupport Class

DefaultEditorSupport is a simple implementation for when your editor support needs are very small. For example, imagine that you don't want to bother with checking for context objects, and just want to, say, set an icon for display in Design View. Your implementation might look something like this:

```
public class MyEditorSupport extends DefaultEditorSupport
{
    public MyEditorSupport()
    {
        setBehavior(BEHAVIOR_MAINICON, "/myicons/main.gif");
    }
}
```

However, there's no particular reason to extend DefaultEditorSupport if you're implementing getBehavior yourself.

Using jc−jar Properties to Specify Control Characteristics

The JCS file that's the basis for your control provides a jc−jar with several attributes that you can use to specify how you control appears to the user (or even *whether* it appears).

Properties for Palette Display

The following attributes guide the control's appearance in palettes, such as the Data Palette, and in the Insert menu.

- label The name used for the control on menus and palettes.
- icon−16 A path (relative to the JCS file) to a GIF file that should be used as the control's icon on menus, palettes, in Design View, and so on.
- icon−32 A path (relative to the JCS file) to a GIF file that should be used as the control's large icon.
- group−name The name of the submenu under which the control should appear.
- palette−priority A number that suggests a priority for this control when the IDE is ordering controls in the palette.
- display−in−palette Defines whether this control should be shown in the palette at all.

Properties for Property Editor Display

- description The text that will appear in the Description pane when the control is selected in Design View.

Note that in a JCS, methods also provide an attribute that determine whether the method should be visible to the control's users. To hide a method, click the method in Design View, locate its ide property in the Property Editor, then select true for the hide attribute.

Other Properties

- insert−wizard−class A class that extends ControlWizardSimple. If specified, WebLogic Workshop will use this class to provide a user interface for inserting a new instance of the control.
- resource−file Name of the resource to use in loading localization strings.
- requires−extension A boolean value indicating whether this control must generate a JCX file.

- version The control's version number.

Related Topics

None.

Control Properties Overview

Java control properties provide a way for a control's user to specify values that should be in effect for the duration of a control's run−time use. A user sets a control's properties at design time, and those values are compiled into the control's logic. A control author writes the control's logic so it uses the property values to make the control's behavior specific to the user's needs.

As a control author, you define properties in an annotations XML file conforming to a special annotations schema. To connect the XML file to your control, you give the XML file's name as a value of the JCS file's jc−jar property, in the file attribute.

IDE Support

WebLogic Workshop provides the following built−in support for your control's properties:

- WebLogic Workshop uses the property definitions to display the list of properties and their attributes in the Property Editor. These are displayed in predicatable styles: enumerated types are listed in a drop−down; boolean types may be set to true or false; and so on.
- Property attribute values are validated against the type you define. For example, Source View will display a red squiggle beneath the value when it is invalid. The exception is the "custom" type, for which you define your own validator, as described in Creating Attribute Editors and Validators.
- When the user changes attribute values, Source View and the Property Editor are kept in sync; a change in the Property Editor is reflected in the Source View and vice versa.

As you define control properties, you may want to validate your XML file against its schema. The schema is defined in ControlAnnotations.xsd, which is provided with the ControlDevKit sample application. You can use XML Spy, provided with WebLogic Workshop, to validate your annotations XML file.

Property Annotations

The WebLogic Workshop IDE exposes properties in two ways: A control's user can set property attribute values using the Property Editor; also, attribute values that have been edited are visible as annotations in source code, where edited values are persisted. In general, the Property Editor (or a custom dialog available from it) is the way that users will interact with properties and their attributes. However, for a control author, who defines properties and attributes and handles them in code, properties and attributes are exposed as annotations and their attributes.

You create a control annotation XML file to define the properties and attributes your control will expose. Through this file, you define:

- Properties that should be exposed at the control level or the method level. In other words, whether the property applies to the control as a whole, so that all control operations are influenced by its values, or applies to a single method or callback, so that it is operation−specific.
- Each property's attributes.
- Each attribute's type.
- Whether an attribute should be required (meaning that it must be set by the control's user), or should have a default value.
- The name of an editor support class that directs the IDE to the class to use as user interface for editing a given attribute's value.

- The name of a validator class that directs the IDE to the class to use for validating annotations for any of the properties in Source View. This class is used to show red squiggles beneath the annotation text for annotations that are not valid, and to display build−time errors. (Note that the edits to values in the Property Editor or custom user interface are validated through a class you specify in an editor support class. This is discussed later in this topic.)
- A description for each property and attribute; the IDE will display this text in the Description pane beneath the Property Editor.

Property Scope

You can specify that a property be applicable in one of three contexts. As you design your control, the decisions you make will include the properties the control must expose, and the scope for each.

When you define you properties in a control annotation XML file, you will specify how a property is applied in the following contexts:

- A control instance; that is, the control's declaration in its client, such as a JWS file.
  Values for instance properties are scoped to the control instance as a unit, rather than to specific methods or callbacks.

  For example, the Database control exposes an instance−scoped connection property whose data−source−jndi−name attribute specifies the data source that all methods of the control will use. For a control that is not extensible (does not generate a JCX file as a extension), all properties must be scoped to the instance.

  Note: Keep in mind that in WebLogic Workshop, it is possible for a user to set property values in both the container code, on the instance declaration, and in the JCX file. This is done by opening the JCX file in Design View, then setting control properties. However, annotations on an instance declaration, where they exist, will always override those in a JCX file.

- A method in a Java control extension (JCX) file.
  Values for interface method properties are scoped to the method they're applied to. For example, in the Database control the sql property is applied to a method, where its statement attribute specifies the SQL expression to use when the method is called.
- A callback in a Java control extension (JCX) file.

  Values for interface callback properties are scoped to the callback they're applied to. In the JMS control, a user can use attributes of the jms−header property to specify values for predefined JMS headers.

WebLogic Workshop provides a way for you get property values at run time. You use the ControlContext class to retrieve these values.

Attribute Groups

The annotation XML schema supports creating attribute groups. You create an attribute group when you want to define a relationship between the attributes belong to a particular property. You can specify that when a user is setting a property's attributes, the control allows at most one, exactly one, or at least one. For more information, see the Control Property Schema Reference material in the WebLogic Workshop documentation.

The following is an example of a simple annotation XML file.

```
<control−tags editor−class="corp.controls.DeptEditorSupport">
    <control−tag name="department">
    <description>Specifies information about the department making requests.</description>
        <attribute−group group−type="exactly−one">
            <attribute name="abbreviation" required="true">
                <description>Specifies the department's four−letter abbreviation.</description>
                <type>
                    <text max−length="4"/>
                </type>
                </attribute>
            <attribute name="id" required="true">
                <description>Specifies the department's ID number.</description>
                <type>
                    <integer/>
                </type>
            </attribute>
        </attribute−group>
    </control−tag>
    <method−tag name="credentials">
        <description>Specifies authentication credentials. Default values will be useful
            for calls to departments other than those above level 2.</description>
        <attribute name="username" required="false">
            <description>Specifies the name to use for authentication.</description>
            <type>
                <text/>
            </type>
            <default−value>anon</default−value>
        </attribute>
        <attribute name="password" required="false">
        <description>Specifies the password to use for authentication.</description>
            <type>
                <text/>
            </type>
            <default−value>anon</default−value>
        </attribute>
    </method−tag>
</control−tags>
```

This file defines the following:

- A control−scoped property called "department" that has two attributes: "abbreviation" and "id". The attributes are in a group whose type is "exactly−one," meaning that one and only one of the attributes may be specified. If the control's user set this property's abbreviation attribute to "ACCT", this property would be written as the following in source code:
  `@jc:department abbreviation="ACCT"`

  This annotation would be written immediately preceding the control declaration in its container's code (such as in a JWS file), or preceding the interface declaration in a JCX −− depending on which had focus when the user set the attribute's value. Note that all custom control properties receive the "jc" (Java control) prefix.
- A method−scoped property called "credentials" that has two attributes: "username" and "password". If the user set the two attributes to values other than their defaults, the result would be the following:
  `@jc:credentials username="gladyskravitz" password="snoop"`

  This annotation would be written immediately preceding the method on which the user set the

attributes and its values would apply for each call to that method.
- An "editor support" class (in the editor−class attribute) to guide specific Design View behavior related to the control. This could include directing the IDE to a custom property editor, specifying icons for display on the control's methods, and so on. For more about editor support classes, see Controlling Appearance and Handling Actions in Design View.

Property Validation

By default, WebLogic Workshop provides validation for all but one of the property attribute types defined in the annotation schema −− including boolean, decimal, enumeration, and so on. The exception is the custom type, which is provided so that you can define you own type. You do this by writing a class that the IDE can use to validate values set by the user for a custom type. You can specify that validation occurs both in user interface (such as a custom attribute editor dialog) or in Source View. For more information, see Creating Attribute Editors and Validators.

The type supported for properties are boolean, class−name, class−names, date, decimal, enumeration, file−path, integer, QNAME, text, URI, URL, URN, XML, and custom. For more information on each of the types, see the Control Property Schema Reference in the WebLogic Workshop documentation.

Retrieving Attribute Values Set By a Control's User

You can retrieve the values set by users for your control's property attributes. You do this by calling one of the ControlContext methods designed for this purpose.

- getControlAttribute(String *tagName*, String *attrName*) Returns the value for a control−scoped attribute declared on the JCX interface associated with the control.
- getControlAttributes(String *tagName*) Returns a List of Map instances, where each entry in the list is a map of the attribute values for a single occurrence of the tag with the specified name.
- getMethodAttribute(String *tagName*, String *attrName*) Returns the value for the method− or callback−scoped attribute declared on the method associated with the current invocation context.
- getMethodAttributes(String *tagName*) Returns a List of Map instances, where each entry in the list is a map of the attribute values for a single occurrence of the tag with the specified name.

Note that all of these methods are context−specific. For example, if your control is extensible your JCS file will implement the Extensible interface's invoke method to handle calls to methods defined in a JCX extending the control. The context for each of those calls to invoke is a particular JCX method in a particular JCX. Calling getMethodAttributes from within an invoke implementation will return the attributes and values for the specified property on *that method*.

Code to get the credentials property's attribute values defined in the preceding example might look like the following. This code would be executed in the context of the invoke method implementation:

```
String username = context.getMethodAttribute("jc:credentials", "username");
String password = context.getMethodAttribute("jc:credentials", "password");
```

Related Topics

None.

Creating Attribute Editors and Validators

You can write components that provide a user interface for editing your control's property attributes, and for validating new attribute values. This topic provides an introduction to building components that expose property attribute values for editing and that validate the new values. For a sample control that implements these features, see the CustomerData control. You'll find this sample in the propEditor package of the ControlFeatures project in the ControlDevKit sample application.

This topic assumes some familiarity with properties and their definitions in an annotation XML file. For an introduction to control properties, see Control Properties Overview.

Validating Properties

You can provide validation at both the property annotation level and for individual attributes for which you have defined a custom type. Providing a validator can be handy when a property requires a specific combination of attribute values. It's also useful when an attribute's value is complex, perhaps even having its own syntax (as with an SQL expression for a Database control method).

WebLogic Workshop provides validation interfaces that you can implement for validating properties and their attributes.

- You implement the ValidateAttribute interface to provide validation for a custom attribute type. Your implementation is used to validate each attribute defined to use the custom type.

  By default, WebLogic Workshop provides validation for all of its predefined attribute types. For example, a control's user may not enter "true" as the value for an attribute whose type is "decimal". For attribute types you define, you implement ValidateAttribute to provide logic for validation.
- You implement the ValidateTag interface to validate a property annotation as a whole. Having an implementation scoped to the annotation is useful when the property exposes multiple attributes and the property isn't valid unless attributes are valid together. For example, you might have a property with "user−name", "password" and "server−name" attributes. If your control's code requires all three to authenticate someone, then the property might be invalid if one of the three attributes is missing.

Implementing a Validation Interface

The two interfaces are very similar. They both define two methods whose names are the same, but whose parameters differ for obvious reasons; a ValidateTag implementation is designed to receive a complete tag, with potentially many attributes; a ValidateAttribute implementation receives a single attribute. In general, your implementation of either method should validate the incoming values, then return null if they're valid. If they're not valid, your implementation should return an array of objects that implement the Issue interface. Your implementation of Issue provides messages that the IDE can present to the user.

The two methods defined in these interfaces reflect the fact that the IDE is designed to validate at two stages: compile−time and edit−time.

- validateDuringCompile *Compile−time* validation isn't restricted to validation when the control's user chooses to build code that includes the property. Compilation for the purpose of validation actually occurs quite frequently. It's more useful to think of compile−time validation as "Source View" validation the kind of validation that results in red squiggles for invalid values.

- validateDuringEdit *Edit−time* validation is performed when the user edits a property through the Property Editor.

While it's undoubtedly more convenient to implement both compile−time and edit−time validation in the same class, it's not necessary. A class for compile−time validation and one for edit−time validation are invoked at in different ways by the IDE, as described below.

Connecting a Validator to the IDE

How you make your validator class available to the IDE differs depending on the circumstances under which validation is being done.

- You've written a custom attribute editor user interface and you want to validate the new value the user is attempting to set. In this case you call a method of your validator class yourself from within your attribute editor implementation. See later in this topic for more details.
- You want to validate the new value when the user edits the attribute's value in the Property Editor directly. You implement an editor support class so that return to the IDE your validator class for the EditorSupport.BEHAVIOR_ATTRIBUTE_VALIDATOR behavior. For more information on implementing editor support, see Controlling Appearance and Handling Actions in Design View. Note that an editor support class is only used for custom attribute types; others are validated by the IDE.
- You want to validate the property annotation code in Source View, or at build time. You specify your validator class in annotation XML file that defines the properties and their attributes. You can specify a class that implements ValidateTag for validating an entire property tag; the class name should be the value for validator−class, an attribute of the control−tag or method−tag element. You can also specify a class that implements ValidateAttribute for validating a custom attribute type; the class name should be the value for class−name, an attribute of the custom element. (Note that the IDE itself validates attributes that aren't custom types.)

Building a Custom Editor

You can build a user interface that is designed specifically to edit the value of a particular property attribute. This can be useful when an attribute's value is potentially long (requiring more space than the Property Editor can provide), or requires some "builder" style editor to compose it. In fact, there are many reasons why you might want to create your own editor.

The Java control API provides two interfaces that define the methods required by a custom editor class. Your custom editor implements one of these to provide a means for the IDE to interact with your user interface, get the value set the by user, and so on.

Choosing an interface to implement is a matter of simplicity versus customization. AttributeEditorSimple, the simpler alternative, is the one to use when your user interface will consist of a single panel. If your user interface will be made up of multiple panels or dialogs (as with a wizard), you should implement AttributeEditorWizard.

Implementing AttributeEditorSimple

The AttributeEditorSimple interface is the basic approach, and will probably suit most needs. When you implement this interface, your user interface will be fit into a dialog owned by the IDE. Your user interface need not provide an OK or Cancel button because these are in the dialog into which your user interface is

placed.

Note: The dialog created will be made as big as needed, up to the size of the IDE.

Here are the methods it exposes, roughly in the order they'll be called by the IDE.

- getFormatter The IDE calls this method to retrieve your class extending AbstractFormatter; this is used to format the attribute value in the Property Editor. You can return null to indicate that no formatting is needed. The IDE's call to this method is not connected with your custom attribute editor (although, of course, you call it yourself).
- getEditorComponent The IDE calls this to get the user interface component for the attribute editor. Extending the JPanel class is a good choice here, given that the component you return will be used in a dialog.
- onFinish Called by the IDE when the user has clicked the OK button on the dialog containing your user interface. Your implementation of this method can validate the value (by calling code in a validator class if you have created one). Return null if its all right to update the value. Otherwise, return an array of classes that implement the Issue interface; these will bear messages that the IDE will present to the user.
- getNewAttributeValue Called to retrieve the attribute's new value so that it can be written into the property annotation in source code, and so that it can be displayed in the Property Editor. The value you return from this method should be retrieved from your user interface code.

Implementing AttributeEditorWizard

Implement the AttributeEditorWizard interface when you want to manage the dialog (or dialogs) containing your attribute editor user interface. This interface defines one method, getDialog, which is called by the IDE to get the Dialog class that defines your user interface. (Note that the getEditorComponent method is not called.) You should use the Frame object received as a parameter of getDialog when constructing your Dialog. This helps to ensure that your dialog is displayed properly with respect to the IDE. When the user clicks OK in your dialog, the IDE calls your implementation of the AttributeEditorSimple.onFinish() method.

Connecting an Editor to the IDE

Once you have written the user interface for your attribute editor, you connect it to the IDE using your implementation of an editor support class. As described in Controlling Appearance and Handling Actions in Design View, an editor support class handles user actions such as double−clicking a method arrow in Design View or clicking the ellipses next to an attribute value in the Property Editor.

An editor support class provides specifics for several "behaviors" defined by the IDE, one of which corresponds to a user's request to edit an attribute value. The constant representing this behavior is EditorSupport.BEHAVIOR_ATTRIBUTE_EDITOR.

The following example describes how you might use an editor support class to connect your custom editor to the IDE.

Putting It All Together

The code in the following example is based on the CustomerData sample control available with the Control Developers Kit. For the complete working version, see the ControlDevKit sample application; there, look for the propEditor folder of the ControlFeatures project.

Validation Class Example

The basic set of tasks for a validation class is simple: receive the values to validate, validate them, and return messages to the user if validation fails (or null if it succeeds). The following code shows a simple attribute validation class.

```
// Implement the ValidateAttribute interface.
public class CustIdValidator implements ValidateAttribute
{
    // Define legal values for the attribute.
    int legalValues[] = { 987654, 987655, 987658, 987659 };

    /*
     * Implement the Issue interface as a way to send messages back
     * to the IDE when the value is invalid. The IDE will extract
     * information from this implementation and present it to the user.
     */
    static class CustIdIssue implements Issue
    {
        String _message;
        private CustIdIssue(String message) { _message = message; }
        public boolean isError() { return true; }
        public String getDescription() { return _message; }
        public String getPrescription()
        {
            return "Provide one of the following values: " +
             "987654, 987655, 987658, 987659. that's all";
        }
    }

    // Called by the attribute editor.
    public Issue[] validateId(String value)
    {
         return validateDuringCompile(null, value, null);
    }

    /*
     * Receive the attribute type name as defined in the annotation XML file. Also
     * receive the value that the user is trying to set for this attribute.
     * The context variable, unused in this example, would be used to store
     * context information that could accumulate and be inspected at validation time.
     */
    public Issue[] validateDuringCompile(String attributeType, String value, Map context)
    {
        // Initialize the Issue array for return to the IDE.
        Issue[] issues = new Issue[1];

        int val;
        try
        {
            val = Integer.parseInt(value);
        }
        catch(Exception e)
        {
            issues[0] = new CustIdIssue(e.getMessage());
            return issues;
        }

        // Find out if the value received is one of the allowed values.
        for(int i = 0; i <legalValues.length; i++)
```

```
        {
            // If the value is okay, return null: no issues here.
            if(legalValues[i] == val)
            {
                return null;
            }
        }

        /* Otherwise, the value must be invalid. Put in instance of the
         * Issue interface implementation into the array and send it to
         * the IDE.
         */
        issues[0] = new CustIdIssue("Unsupported value error");
        return issues;
    }

    public Issue[] validateDuringEdit(String attributeType, String value)
    {
        return null;
    }
}
```

Custom Editor Example

The purpose of a custom editor is to provide a user interface for editing an attribute's value, and to provide a way for the IDE to get the UI and the edited value. This example defines both the user interface and the IDE hooks in the same class, but those could easily have been in separate classes.

```
package propEditor.ide;

import java.util.StringTokenizer;
import java.awt.BorderLayout;
import java.awt.Component;
import javax.swing.JTextField;
import javax.swing.JLabel;
import javax.swing.JFormattedTextField;
import javax.swing.JPanel;
import com.bea.ide.control.EditorSupport;
import javax.swing.JTextPane;
import com.bea.ide.control.AttributeEditorSimple;
import com.bea.control.Issue;
import com.bea.control.DefaultIssue;

/*
 * Represents an attribute editing/validation dialog for the
 * customer-id attribute.
 */
public class CustomerIdEditorSimple extends JPanel implements AttributeEditorSimple
{
    private JTextField m_field;

    /*
     * Constructs a dialog using the existing value for
     * the customer-id attribute. Note that this constructor
     * is called by the getEditorComponent method in this class.
     */
    public CustomerIdEditorSimple(String origValue)
    {
        super(new BorderLayout());
```

```
        m_field = new JTextField(origValue);
        String messageText = "This sample supports the following values: \n" +
                             "987654    987655 \n" +
                             "987658    987659 \n"
                             +"that's all \n";
        JTextPane valueMessage = new JTextPane();
        valueMessage.setText(messageText);
        valueMessage.setBackground(null);
        valueMessage.setEditable(false);
        this.add(m_field, BorderLayout.SOUTH);
        this.add(valueMessage, BorderLayout.NORTH);

    }

    // No special formatting is needed for this value.
    public JFormattedTextField.AbstractFormatter getFormatter()
    {
        return null;
    }

    /*
     * Returns the component to use for the customer-id
     * editing dialog. The IDE calls this method to display
     * the dialog. This class's constructor is used to
     * contain the user interface code.
     *
     * Note that an implementation of the AttributeEditorWizard
     * (as a more customizable alternative) would implement the
     * getDialog method, rather than this one, to return user interface.
     */
    public Component getEditorComponent()
    {
        return this;
    }

    /*
     * Provides a way for the IDE to retrieve the newly
     * entered attribute value. The getText method below is
     * exposed by the JTextField class. The user puts the value in,
     * this code gets it out so the IDE can retrieve it.
     */
    public String getNewAttributeValue()
    {
        return m_field.getText();
    }

    /*
     * Called by the IDE when the user clicks OK. This
     * method calls the validator code above to
     * validate the attribute value, ensuring that it is
     * one of the allowable values.
     */
    public Issue[] onFinish()
    {
        CustIdValidator cIdV = new CustIdValidator();
        return cIdV.validateId(getNewAttributeValue());
    }
}
```

Editor Support Example

The editor support class acts as the signpost, directing the IDE to specific code for each behavior. This implementation handles two behaviors (although one of them is for illustration only): a request for a custom editor and a request for a validator.

```
package propEditor.ide;

import com.bea.ide.control.ControlBehaviorContext;
import com.bea.ide.control.EditorSupport;
import com.bea.ide.control.ControlAttribute;
import com.bea.ide.control.DefaultEditorSupport;


/*
 * Represents support for actions in the IDE. In particular, this
 * class provides code that executes when the user clicks the ... in the
 * Property Editor to edit the customer-id attribute.
 */
public class CustomerDataEditorSupport extends DefaultEditorSupport
{
    public Object getBehavior(String behavior, ControlBehaviorContext ctx)
    {
        /*
         * When the user clicks the ellipses in the Property Editor, the IDE
         * sends a request for specifics on the BEHAVIOR_ATTRIBUTE_EDITOR
         * behavior. This code handles that request, return an object that
         * implements AttributeEditorSimple and provides UI for the custom editor.
         */
        if (behavior.equals(EditorSupport.BEHAVIOR_ATTRIBUTE_EDITOR))
        {
            if (ctx instanceof ControlAttribute &
                ((ControlAttribute)ctx).getName().equals("customer-id"))
            {
                return new CustomerIdEditorSimple(((ControlAttribute)ctx).getValue());
            }
        }

        /*
         * This behavior request won't be received because validation will
         * occur in the context of the custom editor referred to above.
         * But the code is included here for illustration purposes. It would
         * be called if a custom editor were not provided and the user attempted
         * to set a new value in the Property Editor. The CustIdValidator class
         * implements ValidateAttribute.
         */
        if (behavior.equals(EditorSupport.BEHAVIOR_ATTRIBUTE_VALIDATOR))
        {
            if (ctx instanceof ControlAttribute &
                ((ControlAttribute)ctx).getName().equals("customer-id"))
            {
                return new CustIdValidator("customer-id",
                    ((ControlAttribute)ctx).getValue());
            }
        }
        // Return the default implementation for other behaviors.
        return super.getBehavior(behavior, ctx);
    }
}
```

Related Topics

None.

Handling Control Life Cycle Events

The Java control API provides three life cycle callback events that you can handle in your control code. Using these callbacks (exposed by the ControlContext.Callback interface), you can anticipate the start and end of your control's use in a container and write code that executes in response. This is useful, for example, when your control's logic includes access to a resource. You can use two of the life cycle callback events to acquire the resource when you anticipate needing it, and release the resource when the container using your control is nearly finished. The third callback event is useful for resetting state when your control is not in a conversational container. Code for handling these callbacks goes into your control's JCS file.

Keep in mind that the onAcquire and onRelease callback events are only received when the control's container is a conversational web service.

- onCreate() Received after a new instance of the control's top−level container has been created and system initialization (of context object and contained controls) has completed.
  Handle this callback to execute code that you might place in a constuctor. Note that you should never use a constructor in a JCS file.

- onAcquire() Received when a top−level container's operation is about to be called. For example, if your control is in a conversational web service, this callback will be received when one of the conversation operations is about to execute.
  Note that this callback is received when a method of a *top−level* container is about to execute; in other words, if your control is nested in another control that is itself nested in a conversational web service, then it is the call to the web service's method that will provoke this callback.

  You should handle this callback to acquire resources or state that your control will need for any of its methods. You should release the resource in the onRelease callback handler.
- onRelease() Received as a conversation operation of the control's container is finishing execution. You should handle this callback to release resources you acquired in the onAcquire callback handler.
- onReset() Received after the control's container has finished executing, but only if the container is stateless −− meaning non−conversational.
  Handle this callback to reset internal fields to a default state.
- onFinish() Received just after a top−level container's conversation has finished. For example, if your control is in a conversational web service, this callback will be received just after that service's conversation ends.

The code convention for handling these callbacks is the same as for handling other callbacks. The following illustrates how you might handle the onAcquire callback in a JCS file.

```
/*
 * onAcquire is received when a container's operation is about to start.
 */
public void context_onAcquire()
{
    // Use data retrieved from property attribute values to acquire a WebLogic MBean instance.
    try
    {
        m_localMBean = MBeanUtil.getMBean(m_userName, m_password, m_serverURL, m_serverName);
    } catch (IllegalArgumentException iae) {
        context.getLogger("ServerCheck");
        throw new ControlException("ServerCheck: Error getting the domain name.", iae);
    }
```

```
}
```

Related Topics

None.

Creating Extensible Controls

You can implement your control so that it can be customized at design time. In other words, its interface isn't defined (or isn't *completely* defined) until it's actually being used. To do this, you implement your control so that it generates a Java control extension (JCX) file when the user adds it to a design.

Extensible Controls Overview

A control is extensible when it generates or uses a JCX file. A JCX file is an extension of your control. To create a JCX at design time, when the user adds the control to an application, you write code that tells the IDE what the contents of the JCX should be. You can specify whether the generated interface is editable by the user or if, as with the EJB control (whose interface is defined by the EJB it represents), the interface may not be changed.

Here are a few other characteristics of controls extended with a JCX file:

- The JCX is an interface that extends your control's interface. The JCX can define new methods and callbacks, but has no implementation code.
- A control that generates a JCX does so by implementing a "control wizard" (extending ControlWizardSimple or ControlWizard) that is used by the IDE when the user is inserting a new instance of the control. The wizard collects the information needed to generate the JCX, then passes the actual JCX code to the IDE so that the file can be generated.
- When your control supports extension through a JCX, your JCS must implement the Extensible interface, including its invoke method. Your control can define a mixed interface that includes both predefined members (whose implementation is in the JCS file) and late−defined members. Late−defined members are those added when the user creates the control instance, or added by the user later. Calls to late−defined methods in a JCX are delegated to your implementation of the invoke method. There, you handle the specifics of the method call, including its parameter values and property attribute values.
- You can use the ControlContext interface's sendEvent method to invoke a specific callback declared in the generated JCX file.

As described in Creating Dialogs and Wizards for Inserting Controls, you can generate a JCX file by extending one of the control wizard classes. Your implementation receives the name that the user specified for the JCX, along with the package name. You then implement the getExtensionFileContent method to return a String containing the JCX file content.

Note: Before implementing an insert wizard, you may find it useful to create one or more JCX files by hand for debugging purposes, giving them interface definitions and annotations that your control will support. Once you have a stable shape for potential JCX files, and code that handles that shape, you can write code that generates the file, as well as user interface code (if needed) to prompt for required properties.

Whether the user has generated a new JCX file or is using an existing one, a kind of long−distance relationship exists between your core JCS code and the code contained in the JCX. You use the Java control API to manage communications between your core implementation code and the extension.

Validating Control Name and Method Signatures

As described in Controlling Appearance and Handling Actions in Design View, you can enable users to customize the interface represented by the JCX file. If you allow methods or callbacks to be added, removed, or edited, you may also want to validate those changes. The Java control API provides two interfaces you can implement to receive notification when the control's user is attempting an edit, and to validate their changes.

- ValidateMethod Implement this interface to have an opportunity to validate control methods as the user edits them. WebLogic Workshop calls your validateDuringCompile method implementation, passing the method's name, return type, and parameters; your implementation can validate each of these against rules you define. In addition, your implementation receives a Map object in which you can store information current at the time of validation. For example, you might want to store what you discovered about the method's parameters, get and store information about its annotations, and so on. For more information, see the next bullet point about the ValidateControl interface. Your implementation should return null if the method is valid, or return an array of objects that implement the Issue interface if it is not.
- ValidateControl Implement this interface in order to validate the control as a whole. For example, imagine that your control isn't useful unless it has three methods with particular parameters, and each method has a specific, different annotation (such as "start", "continue" and "finish"). As the user is editing the control, adding those methods and annotating them, you can use your implementation of the ValidateMethod.validateDuringCompile method to store method−specific values in the map. When ValidateControl.validateDuringCompile is called at the end of the control's validation cycle (when the method's *start* parameter is false), you can retrieve the map object (*context* parameter), extract the accumulated values, and validate based on them.

Note that the Map object WebLogic Workshop passes to the validateDuringCompile methods exposed by these interfaces is the same instance across validation calls for a given control the user is editing. You can use the object to accumulate information and validate based on it.

Note: These methods are called for each "compilation event" and the map is the same instance for that compilation event but will be empty for the next compilation. A compilation event occurs every time the compiler checks the document's validity such as when it's opened, for each edit, during a build, and so on. In other words: any time the compiler checks for errors to present squiggles in Source View, that's a compilation event.

Note: The two validator interfaces described here are very similar to the ValidateAttribute interface described in Creating Dialogs and Wizards for Inserting Controls. See that topic for more details.

Implementing the invoke Method

If your control supports customization (meaning that it creates a JCX file), it must implement the Extensible interface. Your implementation of this interface's one method, invoke, is the way your core code (in the JCS file) will receive calls to methods defined in the JCX.

Note: You should handle calls to predefined methods just as you would in a control that can not be extended. For example, for control property attributes whose values are needed by such a method, you should retrieve those attribute values in that method's code.

Handling Calls to JCX Methods

The invoke method has two parameters one for the name of the JCX method that has been called and another for its arguments. Needless to say, your invoke implementation should be prepared to handle all of the

valid method names and parameters that the user might add, and to throw an exception (ideally, ControlException) if what you receive is invalid. In other words, if your control allows the user to add multiple methods with multiple numbers of parameters in a range of types, then your invoke implementation will need to test for the possible options.

If a JCX method's logic depends on the value of a property attribute set for the method, you must get the property attribute's value within the context of the invoke method execution, as noted below.

Note: To avoid a case in which a user adds a JCX method that's inappropriate for your control, you should provide a validator class, as described above.

Getting the Value of Attributes Set on a JCX Method

You can use the ControlContext interface's getMethodAttributes method to retrieve property attribute values set on a particular JCX method (note that users will not be able to set properties on predefined methods because they won't be defined in the JCX file). You should be sure to retrieve these in the context of a call to invoke, or to a predefined method.

The ControlContext interface provides several other methods you can use to retrieve other bits of information specific to the current context, such as the current method being called. You can get a specific property attribute set on the control as a whole, a specific method argument, even a Class object representing the current control or callback interface.

Calling JCX Callbacks

Just as calls to JCX methods are delegated to your invoke method implementation, so you invoke JCX callbacks through a kind of delegation.

Note: Your control may define a callback in its core code (meaning, the JAVA file that contains your control's interface) or in a JCX that extends the control. The only circumstances under which a callback may be defined in both places is when the JCX definition extends the core callback.

To execute a JCX−defined callback from your core JCS code, you can use one of three ControlContext methods:

sendEvent This is the most direct way to invoke a JCX callback, and is something of a counterpart to the invoke method. This method has two parameters: the name of the callback you want to execute and an array containing its arguments.

scheduleEvent Use this method to specify that a callback should execute at a particular time. In addition to the callback name and its argument, you also pass as parameters a long specifying the time at which the callback should be invoked. A fourth parameter is a boolean to indicate that (if true) an exception should not be thrown if the conversation containing the callback has finished.

raiseEvent Use this method when you want to simply forward a callback from your control's code. You call raiseEvent from within a callback handler in your JCS; when you do, the callback is forwarded to a callback of the same name in the JCx. Note that the callback handlers must have the same name; if you call raiseEvent from within a myTimer_onTimeout handler, then the client must have a myTime_onTimeout handler defined in order to receive the callback.

Putting It All Together

The code in the following simple example is based on the XQuery sample control available with the Control Developers Kit. For the complete working version, see the ControlDevKit sample application; there, look for the jcxCreate folder of the ControlFeatures project.

This example shows a few pieces of a control that is extensible. Its wizard class prompts the user for the name of a JCX to create, the generates the JCX code; its editor support class tells the IDE whether the JCX may be customized by the user; and its JCS code implements the Extensible interface to handle calls to methods defined in the JCX.

Control Wizard Example

This control wizard class doesn't present a user interface because this control does not require that certain property attribute values be set up front. Instead, it main role is to receive the interface name and package name for the JCX file the user is trying to create, then use those values in constructing the content of the JCX for passing to the IDE.

```
package jcxCreate.ide;

import java.text.MessageFormat;
import com.bea.ide.control.ControlWizardSimple;
import javax.swing.JComponent;

/*
 * Represents an insert dialog for the XQuery control. While
 * this dialog doesn't provide any special user interface, it
 * does provide the text that the IDE should insert into
 * newly created JCX files.
 */
public class XQueryWizard extends ControlWizardSimple
{
    private boolean m_createExtension = false;

    String m_jcxPackage = null;
    String m_jcxName = null;

    /*
     * Tell the IDE how to configure the insert dialog. This dialog
     * should provide an option to use an existing JCX file, or to
     * create a new one.
     */
    public int getConfigurationInfo()
    {
        return CONFIG_CREATE_EXTENSION_FILE |
            CONFIG_INSERT_INSTANCE;
    }

    /*
     * Provide a place for the IDE to tell this code whether the
     * user has asked to create a new JCX or not.
     */
    public void setConfiguration(int config)
    {
        m_createExtension =
            ((config &ControlWizardSimple.CONFIG_CREATE_EXTENSION_FILE) != 0);
    }
```

```
/*
 * Tell the IDE what to use for custom insert dialog user interface.
 * None is needed here.
 */
public JComponent getComponent()
{
    return null;
}

/*
 * Provide a place for code that should execute when the user clicks the
 * Create button on the insert dialog.
 */
public boolean onFinish()
{
    if (super.onFinish() == false){
        return false;
    } else {
        return true;
    }
}

/*
 * Provides a way for the IDE to pass in the package name
 * that should be specified at the top of a new JCX file.
 */
public void setPackage(String packageName)
{
    m_jcxPackage = packageName;
}

/*
 * Provides a way for the IDE to pass in the name of the
 * interface defined in a new JCX file.
 */
public void setName(String name)
{
    m_jcxName = name;
}

/*
 * Provides a place for the IDE to retrieve the content that it
 * should insert into newly created JCX files. Here, the package name
 * and interface name passed in by the IDE are inserted into a
 * template using the MessageFormat class.
 */
public String getExtensionFileContent(){
    // did they ask for an extension?
    String jcxTemplate = this.getTemplate();

    String jcxContent =
        MessageFormat.format(jcxTemplate, new Object[] {m_jcxPackage, m_jcxName});

    return jcxContent;
}

/*
 * Returns a template for text to be inserted into a new JCX file.
 */
private String getTemplate(){
```

```
        String template =
            "package {0}; \n\n " +
            "import com.bea.control.*; \n " +
            "import com.bea.xml.XmlCursor; \n " +
            "import jcxCreate.XQuery; \n\n " +

            "public interface {1} extends XQuery, com.bea.control.ControlExtension \n " +
            "'{' \n \n" +
            "    /* \n" +
            "     * A version number for this JCX. You would increment this to ensure \n" +
            "     * that conversations for instances of earlier versions were invalid. \n" +
            "     */ \n" +
            "    static final long serialVersionUID = 1L; \n \n" +

            "    /* Replace the following method with your own. Be sure \n" +
            "     * that your method returns an XmlCursor and that its \n" +
            "     * parameter is a File object to contain the XML against \n" +
            "     * which to execute the XQuery expression. \n" +
            "     */ \n \n" +

            "    /** \n" +
            "     * @jc:query expression=\"$input/purchase-order/line-item[price <= 20.00]\" \n
            "     */ \n" +
            "     XmlCursor selectLineItem(String filePath); \n \n" +

            "'}' \n ";
        return template;
    }

}
```

Editor Support Example

For the purposes of this example, the key point of interest in this editor support class is the fact that it returns Boolean.TRUE when asked by the IDE whether the user may edit or add JCX methods and Boolean.FALSE for editing callbacks.

This editor support class also specifies what to do when the user asks to edit a method property. Be sure to see the working sample application for the simple user interface code provided.

```
package jcxCreate.ide;

import com.bea.ide.control.ControlAttribute;
import com.bea.ide.control.EditorSupport;
import com.bea.ide.control.ControlMethod;
import com.bea.ide.control.ControlBehaviorContext;
import com.bea.ide.control.DefaultEditorSupport;

/*
 * Represents IDE support for XQuery controls added to a project.
 * An "editor support" class specifies various "behaviors" within the IDE.
 * These include what to do when the user double-clicks on control methods,
 * when the user attempts to add or edit methods and callbacks, and what icon
 * to display on a method.
 */
public class XQueryEditorSupport extends DefaultEditorSupport
{
    /*
     * Provides a way for WebLogic Workshop to retrieve the behavior associated
```

```
     * with specified actions in the IDE. Each action has a corresponding
     * constant in the EditorSupport interface, which is implemented by the
     * DefaultEditorSupport class this class extends. A ControlBehaviorContext
     * argument specifies whether the context of the action is a method, attribute,
     * interface, instance, and so on.
     */
    public Object getBehavior(String behavior, ControlBehaviorContext context)
    {
        /*
         * If the user asks to edit a JCX method, return true.
         */
        if (behavior.equals(EditorSupport.BEHAVIOR_EDIT_METHOD))
        {
            return Boolean.TRUE;
        }
        /*
         * If the user asks to edit a callback, return false. XQuery JCX files do
         * not support callbacks.
         */
        else if (behavior.equals(EditorSupport.BEHAVIOR_EDIT_CALLBACK))
        {
            return Boolean.FALSE;
        }
       /*
         * If the user clicks an edit ... in the Property Editor, and if the attribute
         * corresponding to the ... is expression, return an instance of the expression
         * edit dialog box.
         */
        else if (behavior.equals(EditorSupport.BEHAVIOR_ATTRIBUTE_EDITOR))
        {
            if (context instanceof ControlAttribute &((ControlAttribute)context).getName().equa
            {
                return new QueryExpressionEditorSimple(((ControlAttribute)context).getValue());
            }
        }
        return super.getBehavior(behavior, context);
    }
}
```

JCS Example

Here's the core code for this control. Note that it implements the Extensible interface. The control is designed to be customized with methods that execute XQuery expressions (similar to the way that the Database control executes SQL expressions); the invoke method receives calls to JCX methods, grabbing the expression text as a property attribute of the method that has been called.

```
package jcxCreate;

import com.bea.control.*;
import com.bea.xml.XmlCursor;
import com.bea.xml.XmlObject;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;
import java.io.File;
import java.io.InputStream;
import java.io.FileInputStream;

/**
 * The XQuery control illustrates how to create a control
```

```
 * that generates a JCX file. A JCX file enables the control's
 * user to customize the control's interface.
 *
 * This control also includes a dialog for editing an attribute
 * containing the XQuery expression. Each method of the control's
 * extended interface is annotation with this attribute.
 *
 * @jcs:control-tags file="XQuery-tags.xml"
 * @jcs:jc-jar label="XQuery"
 *      insert-wizard-class="jcxCreate.ide.XQueryWizard"
 *      version="0.8"
 *      icon-16="/images/xml.gif"
 *      palette-priority="5"
 *      group-name="Feature Sample Controls"
 *      description="Extensible control that creates a jcx and runs an XQuery."
 *      @editor-info:code-gen control-interface="true"
  */
public class XQueryImpl implements XQuery, com.bea.control.Extensible, com.bea.control.ControlS
{
    /**
     * @common:context
     */
    com.bea.control.ControlContext context;

    /*
     * A String to hold the XQuery expression specified by the control's
     * user.
     */
    String m_expression = null;

    /*
     * Constants representing annotation and attribute names. The
     * jc:query annotation will be written into the JCX file for
     * each XQuery expression the control's user associates with a
     * control method.
     */
        public static final String TAG_XQUERY = "jc:query";
        public static final String ATTR_EXPRESSION = "expression";

    /*
     * The invoke method is implemented from the Extensible interface.
     * It is a key aspect of any control that provides a customizable
     * interface. When a method on a control's JCX file is called, that
     * call is delegated to the invoke method. The name and arguments for
     * the JCX method called are passed to invoke as arguments. Within
     * the invoke method, you extract and process the method and arguments,
     * returning a value if appropriate.
     *
     * A JCX file for the XQuery control would contain methods whose argument
     * is an object containing the XML on which to execute the query. An
     * annotation on that method would specify the XQuery expression to use.
     */
    public Object invoke(Method method, Object[] args) throws Throwable {
        XmlCursor resultCursor;
        try{
             // Create an instance of the class designed to handle the XQuery.
            XQueryUtil queryUtil = new XQueryUtil();

            // Retrieve the XQuery expression from the annotation in the JCX.
            m_expression = context.getMethodAttribute(TAG_XQUERY, ATTR_EXPRESSION);
```

```
            // Get the XML Stream named by the first parameter.
            // First look to see if it is a resource in the jar that will be created from this
            // Any non-source files in this project will be moved to the jar by default
            String filePath = new String().valueOf(args[0]);
            InputStream iXml= null;
            if (filePath.substring(0,1).equals("/"))
                iXml= this.getClass().getClassLoader().getResourceAsStream(filePath);
                // try as a file
            if (iXml==null)
            {
                File f=new File(filePath);
                if (f.exists())
                    iXml = new FileInputStream(f);
                else
                {
                    throw new ControlException("file not found at " + f.getAbsolutePath());
                }
            }
            // Load the XML using the XMLBeans API.
            XmlObject instanceDoc = XmlObject.Factory.parse(iXml);

            /*
             * Pass the value to the XQuery utility class, and return the results
             * in an XMLBeans XmlCursor object.
             */
            resultCursor = queryUtil.runXQueryExpression(instanceDoc, m_expression);
            return resultCursor;
        } catch (InvocationTargetException ite) {
            throw new ControlException("Error while executing the query: " + ite.getMessage(),
        } catch (Exception e) {
            throw new ControlException("XQueryControl: Exception while executing expression.",
        }
    }

    /*
     * Implement onException for any unhandled exceptions.
     */
    public void context_onException(Exception e, String methodName, Object [] args){
        throw new ControlException("XQuery control exception in " + methodName + ": ", e);
    }
}
```

Related Topics

None.

Creating Dialogs and Wizards for Inserting Controls

You can create a custom user interface that will be used when user adds your control to an application. You may want to provide insert UI if your control simply isn't useful until certain property attribute values are set.

The Java control API provides two classes that define the methods required by the IDE for a custom insert user interface. Your UI extends one of these to provide a means for the IDE to interact with your user interface, to get any values set by the user, and to get the text to be used as code in a JCX file if one is being generated.

- You extend the ControlWizardSimple class to provide a user interface that will be integrated with the default insert dialogs provided with WebLogic Workshop. Your UI will be added as a supplemental panel to the dialog. This is the simplest approach, and probably suits most needs.
- You extend the ControlWizard class to create a completely custom dialog.

Extending ControlWizardSimple

What your ControlWizardSimple implementation does will vary depending on your specific goals. In fact, there are reasons to extend this class that aren't about creating a user interface at all (as described later).

Note: The panel created with your user interface components is actually not always the third in a dialog. If the user adds a new extension of your control from the File menu, it will be the second panel. These are sized slightly differently by the IDE. As with all UI design, you will need to experiment a bit to ensure that you get the results you want.

For the purpose of presenting the added panel in a control insert dialog, your basic goals are captured in the following methods you can override:

- getConfigurationInfo Called by the IDE to discover whether your control supports using or generating a Java control extension (JCX) file and whether it supports being designated a control factory. Your return value should be some combination of the following four constants (or'ed together), although none of them is required.
  - ♦ Return CONFIG_CREATE_EXTENSION_FILE to indicate that your control can generate a JCX file. If you return this value, you must also override this class's getExtensionFileContent method to pass to the IDE the contents of the JCX file it generates. When you return this value, the insert dialog gives the user the option to generate a JCX and to give that file a name.

    For more information about generating and using JCX files, see Creating Extensible Controls.
  - ♦ Return CONFIG_INSERT_INSTANCE to indicate that there are properties that are scoped to the control's instance declaration that is, to the control as a whole. If you return this value, you must also override the getInstanceAnnotations method as described below.
  - ♦ Return CONFIG_NO_FACTORY to indicate that your control cannot be used as a control factory. If you return this value, the insert dialog will disable the check box that allows users to specify control factory use. (See the WebLogic Workshop control documentation for more information about control factories.)
  - ♦ Return CONFIG_USE_EXISTING to indicate that an existing JCX file may be used for this control instance. If you return this value, the insert dialog will allow the user to enter the name of an existing JCX file rather than a new one.

- getComponent Called by the IDE to retrieve the user interface components that should make up the supplemental panel. Your user interface should provide a place for the user to enter values for attributes without which the control would be less than useful. The user should end up with a control instance that immediately compiles and runs.
- onFinish Called by the IDE when the user clicks the Create button on the insert dialog. Your code here should perform any validation needed for the values the user has entered in the dialog. The method should return true if all values are valid and it's all right for the IDE to insert a new control based on them. You should return false if something's not right. (You can create a class specifically for validating property attribute values. For more information, see Creating Attribute Editors and Validators.)
  If you return false, the IDE will call the getIssues method.
- getIssues Called by the IDE to retrieve an array of classes that implement the Issue interface. You implement Issue so that you can return messages back to the user via the IDE. The Issue interface provides a place for you to describe what went wrong, what to do about it, and whether the problem is an error. The IDE will present a dialog containing your messages, after which the user will be able to edit the value they've entered and try again.

The preceding methods are those that guide the contents of the insert dialog. The ControlWizardSimple class's remaining methods don't set user interface characteristics, but are key parts of an insert dialog implementation.

- setConfiguration Called by the IDE to indicate whether the user has chosen to generate a JCX file. Your code will receive CONFIG_CREATE_EXTENSION_FILE if they have (if you specified that it is possible for them to do so, as described above).
- setName Called by the IDE to give you the name chosen by the user for the control extension interface (and, of course, the name of the JCX file). You use this value when constructing the content of the JCX file.
- setPackage Called by the IDE to give you the package name for the control extension interface if a JCX is to be generated. As with the value received via the getName method, you use this value in the JCX file's package declaration.
- getInstanceAnnotations Called by the IDE to get property annotations for the instance declaration. You should override this method if you returned CONFIG_INSERT_INSTANCE from your implementation of the getConfigurationInfo method. This method should return an array of TagAttributeList objects that the IDE can use to annotate the control's instance declaration.
- getExtensionFileContent Called by the IDE to retrieve the content of the JCX file to generate, if your control supports it. You should override this method if you returned CONFIG_CREATE_EXTENSION_FILE from the getConfigurationInfo method. The return value for this method is a String containing the contents of the JCX file. You should build this value based on values the user has entered in the insert dialog, including those you receive via the setName and setPackage methods. You should also include a serialization version ID as illustrated in Creating Extensible Controls.

Finally, the ControlWizardSimple class provides two methods you may find useful for interacting with the IDE's current context. The IDE calls setContext to pass in an instance of the EditorContext object. This object provides methods through which you can start WebLogic Server, print control−related messages to a log, get a File object representing the project to which your control is being added, and more.

Use the getContext method to retrieve this object.

Extending ControlWizard

The key differences between extending ControlWizard and extending ControlWizardSimple have to do with the fact that when you extend ControlWizard, your user interface isn't a piece of another dialog owned by the IDE. For two pieces of information, the role between your component and the IDE are swapped: you provide the control extension name (if any) and instance name to the IDE, rather than the other way around.

The ControlWizard methods you can override include the following:

- getDialog Called by the IDE to retrieve a Dialog instance representing your user interface. As with an extension of ControlWizardSimple, your UI should prompt the user for any property attribute values needed to make the new control instance compilable within its container. In addition, your UI must also prompt for an instance name (the value that goes into the control's declaration in its container code) and extension name, if a JCX is to be generated.
- getExtensionName Called by the IDE to retrieve the JCX name set by the user in your dialog.
- getInstanceName Called by the IDE to retrieve the instance variable name set by the user in your dialog.

Note that two methods overridden in ControlWizard are never called by the IDE: getComponent and getIssues. The getComponent method is rendered unused because it is replaced by getDialog, which must return a Dialog instance rather than a Component instance. The getIssues method is unused because in the context of your dialog, you are responsible for displaying messages that warn the user of invalid values.

Prompting the IDE to Validate Names

Because the IDE is "out−of−the−loop" with regard to what goes on in your custom dialog, you must do a little extra to ensure that the control instance and JCX names that the user has entered in your dialog are valid from the IDE's perspective. Your code should do this before returning from your dialog's show method.

To do this, you use the ControlWizard.NameValidator class that the IDE passes to your dialog with the setNameValidator method. Your code should hang on to that instance, then call its validateInstanceName or validateExtensionName method with the name proposed by the user. The return value for an invalid name will be an array of Issue objects you can use to present relevant messages to the user.

Connecting a Custom Insert User Interface to the IDE

To have the IDE use your control insert implementation, you simply set your JCS' insert−wizard−class attribute to the name of your implementation class. The insert−wizard−class attribute is one of the jc−jar property attributes.

For more information about the jc−jar property see Controlling Appearance and Handling Actions in Design View.

Putting It All Together

The code for the following example is based on the ServerCheck sample control in the ControlDevKit sample application. For the complete working version, see the ControlFeatures project in that application, and look in the insertWizard folder.

ControlWizardSimple Example

As discussed in this topic, an insert dialog's job is to prompt the user for information needed to create a compilable instance of the control. This example displays a supplemental pane in the default dialog to prompt the user for several values needed to connect to a WebLogic MBean. It does not generate a JCX file, instead returning the property attributes so they can be written as annotations to the control's instance declaration.

For an example that generates a JCX file, see the XQuery control in the jcxCreate folder of the ControlFeatures project.

```
package insertWizard.ide;

import com.bea.ide.control.ControlWizardSimple;
import javax.swing.JComponent;
import java.util.ArrayList;
import com.bea.ide.control.EditorContext;

/*
 * The insert-wizard-class attribute in the jc-jar file for this
 * control project specifies that this is the class WebLogic Workshop
 * should use to provide a custom insert wizard.
 *
 * This class extends ControlWizardSimple, which provides methods through
 * which WebLogic Workshop retrieves the user interface to use, the
 * annotation values to insert into the control's container, and so on.
 */
public class ServerCheckWizard extends ControlWizardSimple
{
    private ServerCheckWizardPanel m_panel;
    private boolean _createExtension = false;
    private EditorContext _context;

    /**
     * Initializes the user interface. UI components
     * are assembled in the ServerCheckWizardPanel class.
     * That class is not presented in this topic -- it's
     * typical of a Swing component. See the ControlDevKit
     * sample application for its code.
     */
    public ServerCheckWizard ()
    {
        m_panel = new ServerCheckWizardPanel();
    }

    /**
     * Tells WebLogic Workshop whether to display an option
     * to create a JCX file. The constant returned here
     * tells it that an instance is all that is needed.
     */
    public int getConfigurationInfo()
    {
        return CONFIG_INSERT_INSTANCE;
    }
    /**
     * Gives the wizard a way to interact with the IDE.
     */
    public void setContext(EditorContext ctx)
    {
        _context = ctx;
```

```
    m_panel.setContext(ctx);
}

/**
 * Provides a way for subclasses to get the context as needed.
 */
public EditorContext getContext()
{
    return _context;
}

/**
 * Returns the user interface component to use.
 */
public JComponent getComponent()
{
    return m_panel;
}

/**
 * Provides a place to execute code when the control's user
 * clicks the "Create" button on the insert dialog.
 */
public boolean onFinish()
{
    if (super.onFinish() == false)
    {
        return false;
    } else
    {
        return true;
    }
}

/**
 * Provides to WebLogic Workshop an ArrayList containing
 * the annotations it should write preceding the control
 * instance.
 */
public ArrayList getInstanceAnnotations()
{
    // An ArrayList for the annotation attributes. There are four.
    ArrayList attributeList = new ArrayList();
    // An ArrayList for this control's two annotations.
    ArrayList annotationsList = new ArrayList();

    /*
     * The attribute ArrayList is filled with TagAttributeValue
     * instances containing attribute name/value mappings. These
     * are used below for the jc:server-data annotation.
     *
     * Attribute values are retrieved from the insert dialog box UI.
     */
    attributeList.add(new TagAttributeValue("server-name", m_panel.getServerName()));
    attributeList.add(new TagAttributeValue("url", m_panel.getServerURL()));
    attributeList.add(new TagAttributeValue("user-name", m_panel.getUserName()));
    attributeList.add(new TagAttributeValue("password", m_panel.getPassword()));

    /*
     * The annotation ArrayList is filled with TagAttributeList instances
     * containing the annotation name/attribute-list mappings.
```

```
         * Note that the second annotation uses the attribute ArrayList.
         */
        annotationsList.add(new TagAttributeList("common:control", null));
        annotationsList.add(new TagAttributeList("jc:server-data", attributeList));
        return annotationsList;
    }
}
```

Related Topics

None.

Packaging Controls for Installation

You can easily package your controls for delivery to users. This topic describes how to do so, and includes the following sections:

- Available Controls: Stubs and Deliverables. Controls may be available to the user without being present in their installation.
- Using Control Stubs as Placeholders. WebLogic Workshop supports just−in−time control installation.
- Making Control Deliverables. You can easily package your control in a control deliverable.
- User−Specified Control Library Locations. Users can specify multiple locations where the IDE should look for control libraries.
- Control Appearance in the Insert Menu. You can configure how and where controls appear in the Insert menu.

Available Controls: Stubs and Deliverables

WebLogic Workshop supports the idea of available controls. An *available control* is one that is visible to the user, but whose implementation may not yet be installed or even present on the user's computer. From the users standpoint, a control is available when its name appears on the WebLogic Workshop Insert menu. An available control can take one of two forms on the user's computer:

- A *control deliverable* is a ZIP file that contains everything the control needs in order to be useful to the user. This includes the control's implementation and, if needed, annotation XML files, help files, and so on. When you're ready to test installation of your control, you can make a control deliverable from your control project. For more information on making a control deliverable, see Making Control Deliverables. Note that the control deliverable may also be digitally signed.
- A *control stub* is a ZIP file that contains information necessary to make the control visible to the user, but without including the control implementation or other files. To make a control visible as a stub, you include a jc−stub.xml file, along with any GIFs that should accompany the control's name on the IDE Insert menu. The control deliverable is downloaded and installed when the user first chooses to add it to an application. If the control is digitally signed, the user will be prompted to accept it. For information on creating stubs, see Using Control Stubs as Placeholders.

Control Deliverable Installation Process

Through the features described in this topic, WebLogic Workshop installs control deliverables as described here.

When a control deliverable or stub is found on the External Control Library path (described below) it becomes "available". WebLogic Workshop automatically adds the control title and its icon to the Insert −> Controls menu and the application context (right−click) Install menu. If the user chooses to add an available control to an application, Workshop does the control installation as follows:

1. If the item requested by the user is a control stub, the user is asked whether he or she wants to download the control implementation.
2. If the user answers "yes," WebLogic Workshop attempts to download the control deliverable from the location specifed in the jc−stub.xml (described below).
3. When the download is complete, the ZIP file is checked for a signature and the user is prompted with the results of this check. If the user agrees, the stub file is moved to the

<BEA_HOME>/ext_components/.installed_subs directory and installation now proceeds as if the control deliverable itself were found on the External Control Library path.

4. Any JAR files found under the /controls directory are copied to the current application's Libraries folder (APP−INF/lib). This may include associated JARs needed by the control.
5. If this is the first time the user has installed the control into any application, the contents of the "help" folder will be copied to <BEA_HOME>/<WEBLOGIC_HOME>/workshop/help/.
   Also, the contents of the "samples" folder will be copied to <BEA_HOME>/<WEBLOGIC_HOME>/samples/partners/
6. WebLogic Workshop starts a rebuild of the help index to incorporate the control's help content.

Using Control Stubs as Placeholders

A control stub JAR file contains a jc−stub.xml file that describes the control and provides a URL from which the implementation can be downloaded. It should also contain GIF files that will appear with the control's name in IDE menus.

The following is an example of a jc−stub.xml file.

```
<stub xmlns="urn:workshop−control/external−stub.xsd">
    <downloadURL>http://www.mycompany.com/controls/MyControl.zip</downloadURL>
    <control>
        <display−in−palette>true</display−in−palette>
        <label>My Control</label>
        <icon−16>myControl16.gif</icon−16>
        <icon−32>myControl32.gif</icon−32>
        <version>1.0</version>
        <palette−priority>3</palette−priority>
        <group−name>My Company's Controls</group−name>
        <group−priority>100</group−priority>
        <interface−class>MyControl</interface−class>
        <implementation−class>MyControlImpl</implementation−class>
        <description>Makes every application run faster and more reliably.</description>
    </control>
</stub>
```

The <downloadURL> element must be a valid URL that points to a ZIP file. Other elements have the same meanings as their counterparts in the jc−jar annotation for JCS files.

For the schema on which this XML is based, see ExternalControlStub.xsd.

Making Control Deliverables

When you are ready to package your control project for testing or delivery, you can make a control deliverable. Remember that a control deliverable is a ZIP file that contains all of the files needed for your control, including its implementation files.

Note: Developing your controls in a control deliverable project greatly simplifies the task of delivering a control that includes documentation and/or samples. For more information, see Control Deliverable Projects.

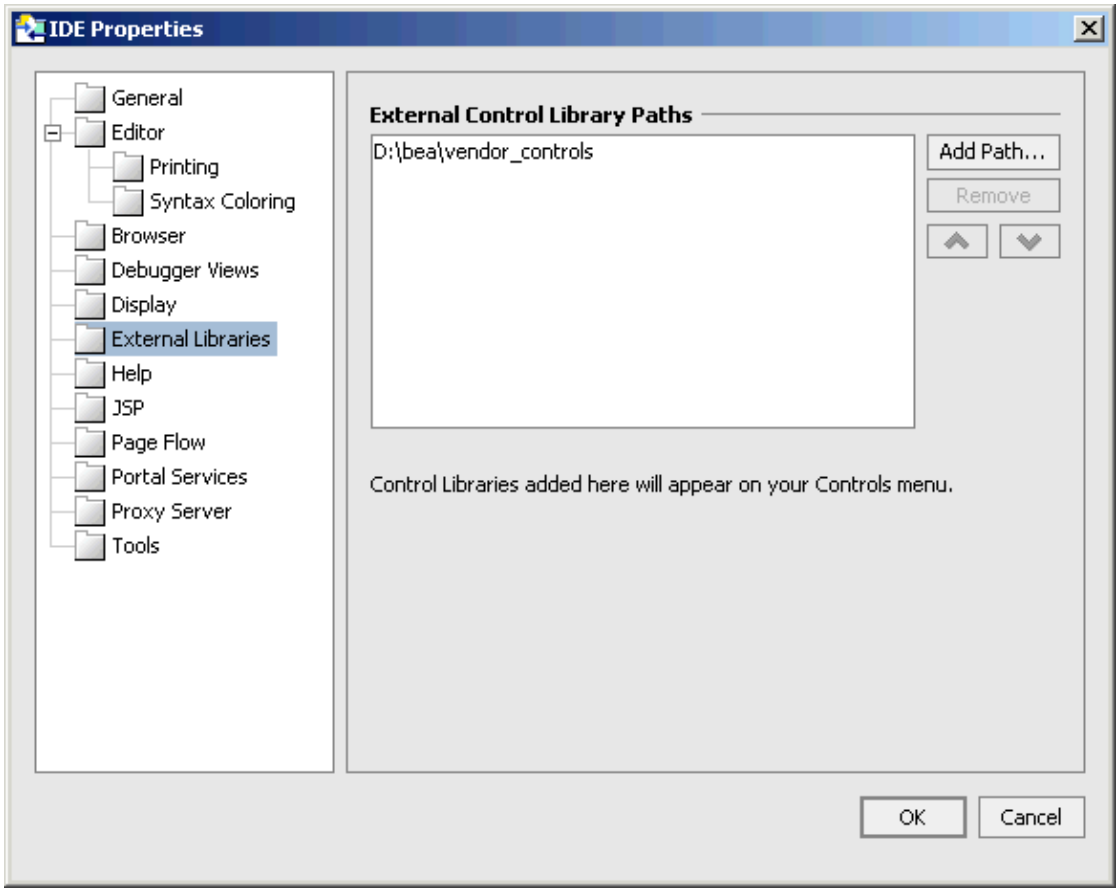
To make a deliverable, right−click the project folder, then click Build Control Deliverable. WebLogic Workshop will package all the required files into a ZIP file and put the ZIP in the folder that contains your application (the one with the application's WORK file). Note that the ZIP file won't be visible from within the

WebLogic Workshop Application view you'll need to browse the file system to get it. You can also specify the location where WebLogic Workshop should place the control deliverable; you can do this in the control project's properties.

To test the deliverable, you can add it to a folder that's specified as a control library location (see User−Specified Control Library Locations), then add the control to an application.

User−Specified Control Library Locations

Users can specify multiple locations at which the IDE should look for controls. They do this through the IDE Properties dialog (shown here) by adding external control library paths. User can add paths to ZIP files, JAR files, or merely paths to folders. Control stubs or deliverables



Control Appearance in the Insert Menu

A described previously in this topic and in the core WebLogic Workshop documentation, you can configure how and where your controls appear in WebLogic Workshop's Insert menu. You do this by setting group−name and group−priority values. For control deliverables, you can set these values as attributes of your control's JCS file. For control stubs, you set these values in the jc−stub.xml file.

group−priority

Optional. A number indicating where in the Insert menu the group named in the group−name attribute should appear. Higher numbers (relative to other groups) place the group higher in the menu. Default is 0.

group−name

Optional. The name of the submenu under which this control should appear. If the group−name atribute value ends with the string "Menu" (such as "MyGroupMenu"), then the groupname, stripped of the word Menu, always becomes a submenu name, and all controls sharing that group appear on the submenu. If the group name does not end in the word Menu, then controls in that group appear as top−level items in the Insert Controls menu, unless there are more than five controls in that group. In that case, the group name becomes the top level menu name and all controls in that group appear as sub−menu items.

The following table lists the various sources from which the IDE integrates controls, along with their group−name and group−priority values.

Control Source	group−priority Value	group−name Value	Label
Workshop built−in controls	−900	Workshop	Database, EJB, JMS, Timer, Web Service
Built−in integration controls	−600	IntegrationMenu	BEA supplied
Built−in portal controls	−300	PortalMenu	BEA supplied
Control projects in an application	default 0	<ProjectName>	User specified
Application’s Libraries folder	default 0	<ApplicationName>	User specified
User controls in current project	default 0	Local Controls	User specified
External libraries	Recommended: 100	Specified by control author	Specified by ctrl author in jc−jar.xml

For more information about JCS attributes, see the @jcs:jc−jar Annotation topic in the core WebLogic Workshop documentation. You can reach this topic quickly by selecting the attribute in the IDE, the pressing F1.

Related Topics

None.

Advanced Control Samples

The ControlDevKit application that accompanies these help topics contains a number of projects that demonstrate some of the key features and functionality for creating to−be−redistributed controls. A description of these projects is given below.

ControlFeatures Project

The samples in this project illustrate individual advanced features of Java controls. In particular, they demonstrate how to implement IDE extensions that provide customizable behavior in WebLogic Workshop. Other samples illustrate advanced event and callback related features. You can use the web services in the ControlTest project to try out these controls (see below).

The following list describes features illustrated by the samples:

- **controlEvents**

  EventRaiser — Using the JbcContext.raiseEvent method to forward callbacks to the control's client.

  EventScheduler — Using the JbcContext.scheduleEvent method to specify that a control callback should be invoked at a particular time.

  LifeCycle — Callback handlers a control can implement to handle the events in its life.

- **insertWizard**

  ServerCheck (and accompanying files) — Implementing a custom insert dialog for a control. The control wizard in this example extends ControlWizardSimple, which provides much of the user interface for inserting the control.

- **insertWizardCustom**

  CustomWiz (and accompanying files) — Implementing a fully−custom insert dialog for a control. The wizard in this example extends ControlWizard, which allows you to implement an insert dialog that does not inherit user interface components in a dialog provided by WebLogic Workshop.

- **jcxCreate**

  XQuery (and accompanying files) — Implementing support for generating a JCX file when the user adds the control to a project. This sample also includes a custom attribute editor.

- **propEditor**

  CustomerData (and accompanying files) — Implementing support for an attribute editing and validation dialog.

ControlTest Project

Contains test web services for the controls in the other two projects.

DBScripter Project

Contains a more complete example of a functional control. This control defines new JDBC data sources and their associated connection pools, and runs script files against the newly created or existing data sources. The control generates a JCX file at insert time and also supports a callback interface. This sample also illustrates how you can include documentation and sample code in such a way that it will be automatically integrated with the user's installation when they install the control from a control deliverable. For more information on packaging documentation and samples for automatic installation, see Packaging Controls for Installation.

Related Topics

None.

Describes the contents of a control jar file. The configuration of a specific control Description of the control. Will have version attribute concatenated if present. The text displayed for this control in the palette (formerly "name"). The fully qualified class name of the control interface. Fully qualified name of the control implementation class. Only needed if the control implementation does not follow the convention of InterfaceImpl in the same package as the control interface. Fully qualified classname for a user provided insert wizard. If no class is provided, the default insert wizard will be used. A string representation of the control version. The format and meaning of the version is implementation specific. Will be appended to description in tool tip. Path to an icon to be used in the control palette. If no icon is provided, only the label will be displayed in the palette. Optional group priority that can be used to organize the order in which groups of controls appear on menus. Used as a hint to the IDE for ordering of controls within the palette. Defines whether this control will be displayed in the control palette. Optional name of resource file to load when the control author has provided localization strings. Note that the only values that can be localized in jc−jar.xml are: 1. label 2. version 3. description Default group name for controls in this jar file. If a control has its own group−name attribute, that value will be used for that control. Describes the annotations for a specific control Structure of a control. Note that this structure has been re−factored in beta. A tag is defined as either a control or method tag. A tag that applies to the control instance. A tag that applies to a method within a JCX. Javelin should verify that this type of tag only exists for a customized control (i.e. the control can support a JCX). Fully qualified class name of a user provided class (e.g. property builder) Structure of a tag that configures a control the name of the tag. does not include prefix. Whether the tag can have multiple occurrences. Fully−qualified name (as string) of an optional user provided class. The class provides custom validation logic over the entire tag. This provides a mechanism for control authors to implement cross attribute validation for a specific tag instance. Range of priorities allowed for Palette Display Simple text representation Max length of the text value. An indication for the IDE as to whether the text value should be edited in place or if a text box should be opened. A list of valid values A valid value as a string true/false non−integral value The number of decimal places The minimum value inclusive The maximum value inclusive Integral value Minimum value inclusive Maximum value inclusive Date Minimum value inclusive Maximum value inclusive An XML qualified name prefix:local part Well−formed XML Fully−qualified class name Space−separated list of fully qualified class names Valid file path A custom (user defined) type The name of the type The fully qualified name of the class that implements an interface for validation of a type. Default value for the attribute. Attribute name Defines the types of methods where this tag can exist. Javelin should verify that the methods match the control implementation (e.g. the control is extensible, there is a callback interface, etc). Optional group name that can be used to organize the control palette.

Extending the WebLogic Workshop IDE

You can extend the WebLogic Workshop IDE at several levels of depth: from simple tool bar icons that launch external applications to fully integrated applications with custom windows, menus, and property sheets.

An IDE extension is made up of the following artifacts, packaged together as a JAR or directory.

- An XML descriptor extension.xml
- A set of Java classes that expose interfaces and consume various services provided by the IDE
- Resource files, including images and string property files
- Documentation

WebLogic Workshop itself is the culmination of several extensions pulled together in one common interface. wlw−ide.jar is the main WebLogic Workshop executable, containing the core class files, utility classes, and the mechanism to load extensions. But the interesting code to run the IDE is contained in the required IDE extensions. These include the debugger, the JSP designer, source control, and more.

To cause Workshop to load a specific extension at start−up, the extension s JAR file is placed in the "extensions" subdirectory below the directory containing the wlw−ide.JAR file that contains the IDE core. An extension may also be fully exploded to directories and class files in the extensions sub−directory.

Creating Workshop Extensions

A WebLogic Workshop extension is simply defined as a JAR or a directory that contains at a minimum an extension.xml file that declares an extension's participation in the IDE. For example, a JAR file with the following extension.xml file adds a new tool bar icon to the IDE.

```
<extension−definition>
    <extension−xml id="urn:com−bea−ide:actions">
        <action−ui>
            <toolbar id="sqltoolbar" path="toolbar/main" priority="2"
                label="%sqlEditor.extension.actionSQLAttributeEditor%">
                 <action−group id="sqledit" priority="10" />
            </toolbar>
         </action−ui>
         <action−set scope="com.bea.ide.lang.control.ControlDocument">
             <action class="sqlEditor.SQLAttributeEditor" label="%sqlEditor.extension.actionSQL
                 icon="images/database.gif">
                 <location priority="10" path="toolbar/sqltoolbar/sqledit"/>
         </action>
      </action−set>
    </extension−xml>
</extension−definition>
```

Note: The % signs in "%sqlEditor.extension.actionSQLAttributeEditor%" are a means of referring to a localizable string resource and that it's optional.

This extension places a new button on the toolbar with database.gif as its icon. When the new toolbar button is pressed, the SQLAttributeEditor is displayed.

For more complex extensions, the JAR file would contain the java code that is the extension's implementation,

a manifest file which defines the class path and attributes that reference dependent JARs that need to be available at run time by the extension.

At start−up, the core IDE runtime reads all the extension.xml files, batches them together and ensures that the requested services by each extension are available.

Extensions may define handlers for the <extension−xml> tag found in the extension.xml file. Handlers are associated with a particular id attribute. All extension.xml files are scanned for fragments contained within the <extension−xml> tag and those fragments are passed to handlers defined for the particular id attribute. This mechanism allows extensions to create extendable infrastructure in which other extensions can participate. The handler class is instantiated by the core IDE and is completely responsible for parsing the xml contained in the fragment.

Services Provided by the IDE

One important aspect of an extension is its ability to expose and consume the various IDE services. A service is essentially a public interface that has a single instance implementation and provides access to functionality in an extension. In addition to handlers for <extension−xml> tags, an extension may declare services that it implements. For instance, the debugger extension defines a debugger service that, among other things, provides a method for setting a breakpoint. The shell provides a document service with a method for opening a document. Services are consumed by the extensions Java classes and registered with the system using special tags in the extensions.xml file. Many services have associated <extension−xml> tag handlers which allow extensions to add functionality to the service.

For a list of services, see Extension Services in WebLogic Workshop.

The IdeDevKit sample application includes several projects that illustrate IDE extensions. For more information, see IDE Extension Samples.

Related Topics

None.

Getting Started: IDE Extension Tutorial

This brief tutorial is designed to introduce the basics of building an IDE extension. In it, you'll learn a bit about the pieces of an IDE extension, including the extension.xml file and Java classes that make up a frame view extension. You'll also learn how to set up a Java project for building and debugging extensions.

The extension you build with this tutorial is very simple, so it's not surprising that a proportionately large amount of the tutorial is about getting set up −− creating the project, setting debugging properties, and so on. Needless to say, these are things that will come fairly easily after the first time around.

The tasks in this step are:

- To create a new Java project for extension code.
- To set up for debugging.
- To create an extension.xml file.
- To write the extension's Java code.

To Create a Java Project

You build IDE extensions in a Java project type.

1. Open the IdeDevKit sample application located here:
   BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\IdeDevKit.work
2. In the ***Application*** window, right−click the top folder, ***IdeDevKit***, then click ***New*** −> ***Project***.
3. In the ***New Project*** dialog, in the left pane, click ***All***. In the right pane, click ***Java Project***.
4. In the ***Project Name*** box, type FrameViewHello.
5. Click ***Create***.

   WebLogic Workshop will display the FrameViewHello project in the IdeDevKit application. Now you will create folders for source files.
6. Right−click the ***FrameViewHello*** project folder, then click ***New*** −> ***Folder***.
7. In the ***Create New Folder*** dialog, enter src as the new folder name.

   Next you'll tell the IDE to use this folder for Java sources. By default, the source path is set to the project's root. But if you leave it that way, you'll be unable to use the IDE to create a META−INF folder in the project to hold your extension.xml file ("META−INF" is illegal as a package name). You'll also add wlw−ide.jar to your classpath. It contains the classes and interfaces that make up the extension API. You're going to need those.
8. Right−click the ***FrameViewHello*** project folder, then click ***Properties***.
9. Next to the ***Classpath*** box, click Add Jar and browse for the JAR file at:

   BEA_HOME\weblogic81\workshop\wlw−ide.jar
10. Select the JAR and click ***Select Jar***.
11. Next to the ***Source path*** box, click ***Add Path*** and browse for the ***src*** folder you added earlier, probably at:

    BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\FrameViewHello\src
12. With the ***src*** folder selected, click ***Select Directory***.
13. Click ***OK***.

Next you will create a META−INF folder for your extension.xml file.

14. Right−click the ***FrameViewHello*** project folder, then click ***New*** −> ***Folder***.
15. In the ***Create New Folder*** dialog, enter META−INF as the new folder name.
16. Click ***OK***.

To Set Up for Debugging

When debugging an IDE extension, you set up your build properties so that the extension build output is copied the WebLogic Workshop extensions folder. You set debugging properties so that a new instance of the IDE starts. This new instance examines the contents of the extensions folder, finds your extension JAR, and loads it with other extensions. This works iteratively so that each new build generates a new extension JAR which is read by the newly started IDE instance. Breakpoints you've set in the original instance are of course honored as you try out your extension in the second.

To set this up, you need to create a build.xml file in your extension project. You also need to set debugging properties so that a new instance of the IDE is started. First you'll set those debugging properties.

1. Right−click the ***FrameViewHello*** project folder, then click ***Properties***.
2. In the ***Project Properties*** dialog, on the ***Debugger*** panel, enter values as described at Setting Up Extension Debugging Properties.

Next, you'll create a build.xml file.

1. Right−click the ***FrameViewHello*** project folder, then click ***Properties***.
2. In the ***Project Properties*** dialog, on the ***Build*** panel, click ***Export to Ant file***, then click ***Cancel***.
3. In the ***Application*** window, locate the ***exported_build.xml*** file generated for you.
4. Rename ***exported_build.xml*** to simply ***build.xml***, then double−click the file to open it in ***Source View***.
5. In the ***build.xml*** file, scroll down to the "***build***" target, the add the following as the last task in the target:

   ```
   <zip destfile="${platformhome.local.directory}/workshop/extensions/${output.filename}"
           basedir="${dest.path}"
           includes="**/*.*"
           encoding="UTF8"> <!−− jar filenames are UTF8−encoded −−>
           <zipfileset dir="${project.local.directory}"
           excludes="build.xml,**/CVS/**,**/*.java,${output.filename}"
           includes="**/*.*"/>
   </zip>
   ```

   This zip task zips your extension output and copies it to the WORKSHOP_HOME/extensions folder, where WebLogic Workshop can find it when the IDE starts.

6. Save and close the file.
7. Right−click the ***FrameViewHello*** project folder, then click ***Properties***.
8. In the ***Project Properties*** dialog, on the ***Build*** panel, click ***Use Ant build***. By default, the build target should be displayed as "build", and that's as it should be.
9. Click ***OK***.

Now you're set up to build and debug an extension project. Next you'll create the source files that make up your extension.

To Create an extension.xml File

An extension.xml file connects your extension code with the IDE. You'll create an extension.xml file to tell the IDE a few simple things about your frame view, such as how the view should be listed on a menu command. But your extension.xml will also point the IDE to the Java code that it should execute for your extension's logic and user interface. You'll write that code in a moment.

1. Right−click the ***FrameViewHello/META−INF*** folder, the click ***New*** −> ***Other File Types***.
2. In the ***New File*** dialog, on the left click ***All***, on the right click ***XML File***.
3. In the ***File Name*** box, type extension.xml.
4. Click ***Create***.
5. Replace the file contents generated for you with the following:

```
<extension−definition>
    <extension−xml id="urn:com−bea−ide:frame">
        <frame−view−set>
            <frame−view id="sayhello" class="myframeview.HelloWorldFrame"
                label="Say Hello" />
        </frame−view−set>
        <application−layout id="main">
            <frame−layout id="main">
                <frame−container orientation="east" proportion="20%">
                    <frame−container orientation="tabbed">
                        <frame−view−ref class="myframeview.HelloWorldFrame" />
                    </frame−container>
                </frame−container>
            </frame−layout>
        </application−layout>
    </extension−xml>

</extension−definition
```

   The <extension−definition> element is always the root of an extension.xml file. A file can define many extensions, each specified by a separate <extension−xml> stanza. Here, the definition contains two main chunks of information. The <frame−view−set> stanza simply tells the IDE that this frame view exists, what's its menu and tab label should be, what class to use, and so on. The <application−layout> stanza tells the IDE where to display the frame view on startup. Of course, if you (or your extension's user) has been using the IDE, then preferences for the position of frames has already been stored locally. In that case, this frame will need to be selected from the View menu.
6. Save and close the file.

Now it's time to write the Java code.

To Write the Extension's Java Code

Your extension's user interface and logic are defined in Java classes. In the extension.xml snippet you created, you specified the class myframeview.HelloWorldFrame, so that's the class you're going to create here.

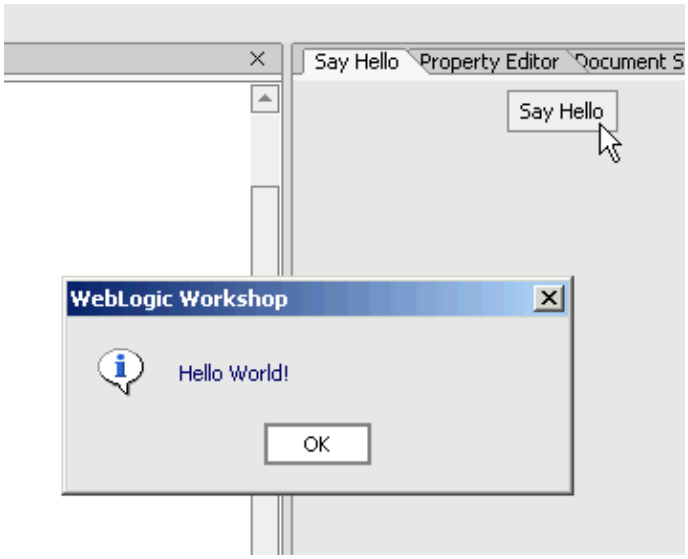
A frame view class implements the IFrameView interface, which provides a method the IDE can use to get your frame UI. It also provides a method the IDE can use to ask whether your frame should be available for display. Start by creating a new package folder and Java class.

1. Right−click the ***FrameViewHello/src*** folder you created earlier, then click ***New*** −> ***Folder***.
2. In the ***New Folder*** dialog, type myframeview for the folder name, then click ***OK***.

3. Right−click the new ***myframeview*** folder, then click ***New*** −> ***Java Class***.
4. In the ***New File*** dialog, on the left click ***Common***, on the right click ***Java Class***.
5. In the ***File Name*** box, type HelloWorldFrame.java, then click ***Create***.
6. In the JAVA file that's created, paste the following code:

```
package myframeview;

import com.bea.ide.ui.frame.IFrameView;
import com.bea.ide.util.swing.DialogUtil;
import java.awt.Component;
import javax.swing.JPanel;

public class HelloWorldFrame
    extends JPanel implements IFrameView
{
    private javax.swing.JButton btnSayHello;
    /**
     * Construct this frame by calling the method that assembles its UI.
     */
    public HelloWorldFrame()
    {
        initComponents();
    }

    /**
     * The IDE calls this to get the UI.
     */
    public Component getView(String id)
    {
        return this;
    }

    /**
     * The IDE calls this to find out if it should make this
     * frame's UI available for display.
     */
    public boolean isAvailable()
    {
        return true;
    }

    // Put together the user interface.
    private void initComponents()
    {
        // A simple button whose label is "Say Hello"
        btnSayHello = new javax.swing.JButton();
        btnSayHello.setText("Say Hello");
        // Connect event handling by adding an action listener.
        btnSayHello.addActionListener(new java.awt.event.ActionListener()
        {
            // If something happens, show a dialog with a friendly message.
            public void actionPerformed(java.awt.event.ActionEvent event)
            {
                DialogUtil.showInfoDialog(null, "Hello World!");
            }
        });
        // Add the new button to this panel.
        add(btnSayHello);
    }
}
```

That's it. Now to try it out. If you got set up for debugging as described earlier in this topic, you should be able to click the ***Start*** button to run the project. When the new IDE instance has started, if the ***Say Hello*** frame isn't visible, you can select it from the ***View*** −> ***Windows*** menu. Click the ***Say Hello*** button, and you should see the friendly dialog. With the frame docked, it might look something like this:



When you've tried out the extension, close the second IDE instance (you can also switch to the first instance and click the ***Stop*** button). If you want to try out debugging, set a breakpoint at the first line of the initComponents method, then run the extension again.

Where to Go From Here

The extension you built with this tutorial is one of several types you can build and it's one of the simplest. In addition, the extension API includes support for a great deal of functionality that can be used within multiple extension types.

For more information on setting up for building and debugging an extension, see Debugging Extensions. For more on frame view extensions, see Adding Dockable Frames.

Related Topics

Extending the WebLogic Workshop IDE

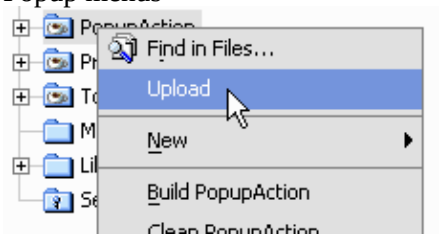
Adding Menus and Toolbar Buttons

When you want to add support for a new command through the menu or toolbar, you add an *action*. The actions in your extensions are typically ways for your extension's user to get to the extension's core functionality.

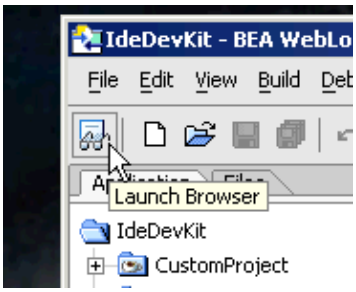
You can associate actions with:

- Menus from the main menu bar

- Popup menus



- Toolbar buttons



You define the user interface for actions by specifying buttons or menus in your extension XML file. You define the logic for actions in Java code, then specify those Java classes in the extension XML.

Be sure to see the reference for ActionSvc for information on building action extensions. Also, for action extension samples, see MenuItems Sample, PopupAction Sample, and ToolbarButton Sample.

Building Action Extensions

In your extension.xml file you sketch out your action's user interface as well as connect the action UI and logic to the IDE. The UI for your action will be a menu command (including a popup) or a toolbar button. In your extension XML, part of your user interface can be defined through the <action−ui> element, but you may want to generate user interface dynamically with Java code.

You add support for new actions by:

- Writing an extension.xml file that defines your action's user interface and connects that UI to the classes that represent your action's logic.

- Writing one or more Java classes that support the actions by implementing the required interfaces.

Extension XML for Actions

Through the extension.xml for an action extension, you define the following:

- Simple and predefined aspects of the action's user interface. These include:
  - Top−level menus or toolbars and their labels.
  - Paths to submenus.
  - The icon to display for an action.
- The fully−qualified names of Java classes that the IDE should use for action logic and additional user interface.

Note: For reference information on action extension XML, see Action Extension XML Reference.

For an example, take a look at the menu commands defined in the MenuItems action sample included with the ExtensionDevKit, in the IdeDevKit application. The following illustration shows menu the extension creates.

As you can see, the extension creates a Favorites menu in the main menu bar, along with a Links submenu. It also creates commands such as Add Favorite, Delete Favorite, and commands to open three URLs. The URL commands open a browser to the selected URL, and you can use the Add Favorite and Delete Favorite commands to add or remove custom URLs. In the following XML code from this sample's extension.xml, the <action−ui> stanza defines the structure of the user interface (although not all of the UI), while the lower <action−set> stanza points the IDE to the logic for actions.

Items grouped with <action−group> stanzas are separated by a visual cue in the IDE, such as a line between groups. These are statically−defined groups, in that they're defined here in the extension.xml and will exist when the extension is loaded.

```
<extension-xml id="urn:com-bea-ide:actions">
    <action-ui>
        <menu id="favorites" path="menu/main" priority="100" label="Favorites">
            <action-group id="linkssubmenugroup" priority="5" path="menu/favorites">
                <menu id="favoritesMenu" priority="10" label="Links">
                    <action-group id="favoritesSubMenu1" priority="10"/>
                    <action-group id="favoritesSubMenu2" priority="10"
                        generator="ideExtensions.menuItems.FavoritesGenerator"/>
                </menu>
            </action-group>
            <action-group id="addremovecommandsgroup" priority="10"/>
        </menu>
    </action-ui>

    <action-set>
```

```
        <action class="ideExtensions.menuItems.AddFavoriteAction" label="Add Favorite"
                    icon="images/workshop/debugger/icons/debug_watch.gif" show-always="true">
            <location priority="10" path="menu/favorites/addremovecommandsgroup"/>
        </action>
        <action class="ideExtensions.menuItems.DeleteFavoriteAction" label="Delete Favorite"
                    icon="images/workshop/debugger/icons/debug_watch.gif" show-always="true">
            <location priority="10" path="menu/favorites/addremovecommandsgroup" />
        </action>
        <action class="ideExtensions.menuItems.LaunchBrowserAction1" label="http://www.bea.com"
                    icon="images/workshop/debugger/icons/debug_watch.gif" show-always="true">
            <location priority="10" path="menu/favorites/favoritesMenu/favoritesSubMenu1"/>
        </action>
        <action class="ideExtensions.menuItems.LaunchBrowserAction2" label="http://www.google.c
                    icon="images/workshop/debugger/icons/debug_watch.gif" show-always="true">
            <location priority="10" path="menu/favorites/favoritesMenu/favoritesSubMenu1"/>
        </action>
        <action class="ideExtensions.menuItems.LaunchBrowserAction3" label="http://www.yahoo.co
                    icon="images/workshop/debugger/icons/debug_watch.gif" show-always="true">
            <location priority="10" path="menu/favorites/favoritesMenu/favoritesSubMenu1"/>
        </action>
    </action-set>
</extension-xml>
```

Specifying Paths

The <action−ui> stanza is largely intended to set up the user interface structure of the commands (such as the menu hierarchy), along with simple bits of user interface. The <action−ui> stanza doesn't specify any logic, so the user interface it specifies is for those items that have no logic, such as a menu command that has only a submenu. The <action−set> stanza, on the other hand, specifies the classes that are plugged into parts of this structure to support commands. It also specifies UI for those commands (such as a command's label).

The hierarchy of elements in the <action−ui> stanza establishes the structure of the menus. The id attributes are strung together in the <action−set> stanza's path attributes to tell the IDE where to hook the Java classes that define logic for the commands.

Defining Action Scope

The <action−set> element provides a scope attribute that you can use to specify when the action should be available to the user. In other words, you can specify that the action should only be available when a document of a particular type is open and visible in the IDE. For example, you might want to do this if you've created an action that is only relevant for a certain document type, such as a web service (JWS), JavaServer Pages page (JSP), or a document that you may have defined in a document extension.

Defining Action UI Mnemonics

For action user interface that's specified in the extension.xml file (such as the mnemonic for a menu command), you add an entitized ampersand character & immediately preceding the label character that will be the mnemonic. In the following example, the letter "r" is the mnemonic:

```
 <menu id="favorites" path="menu/main" priority="100"
    label="Favo&amp;rites">
```

In the extension.xml file itself, the & character must be entitized in order for the attribute value that contains it to be valid XML.

Implementing an Action's Logic

You will typically implement the IAction interface or extend the abstract DefaultAction class (which itself implements IAction). You tell the IDE which class to use for action logic by specifying that class in an <action> element of your extension XML, as in the following snippet, where the class name is in bold text:

```
<action class="ideExtensions.menuItems.LaunchBrowserAction1"
    label="http://www.bea.com"
    icon="images/workshop/debugger/icons/debug_watch.gif" show-always="true">
    <location priority="10" path="menu/favorites/favoritesMenu/favoritesSubMenu1"/>
</action>
```

The com.bea.ide.action package includes classes and interfaces that represent particular kinds of actions. These include:

- DefaultAction is a default implementation of the IAction interface. It provides support for simple scenarios. Note that the IAction.actionPerformed method is not implemented (merely inherited), so you must implement it in a class that extends DefaultAction. When called by the IDE, your implementation of actionPerformed will receive an ActionEvent instance you can use to learn more about the context of the action.
- Implement IPopupAction to support a popup menu. Use its prepare method to find out information about IDE's and the mouse pointer's context.

Other action-related parts of the API you might want to use include:

- IActionProxy, which represents the user interface component that wraps your action. You can retrieve an instance of this interface from a class that extends DefaultAction:

  ```
  IActionProxy ap = new MyActionClass().getProxy();
  ```
- ActionSvc, the class representing WebLogic Workshop's action service, on which action extensions are built. ActionSvc provides access to the static interface ActionSvc.I, whose methods give you access to actions generally. For example, you can get the IAction instance for a specific action in your extension in this way:

  ```
  IAction myAction = ActionSvc.get().getAction(MyAction.class.getName()).getIAction();
  ```

Defining Actions Dynamically

You can add actions dynamically by implementing the IGenerator interface. In particular, the IDE calls your implementation of the interface's populate method, passing to the implementation an instance of the container (such as a menu or toolbar) to which you can add actions. That container implements the IContainer interface, which provides methods through which you can discover the container's action type (whether it is a menu, toolbar, and so on), and add actions, containers, and separators.

In your extension.xml file, you assign the generator class to the action UI under which generated actions will be available. For an example, see the preceding extension.xml code. Note that the ideExtensions.menuItems.FavoritesGenerator class is assigned to the favoritesSubMenu2 branch of the action structure with the <action-set> element's generator attribute:

```
<action-group id="favoritesSubMenu2" priority="10"
    generator="ideExtensions.menuItems.FavoritesGenerator"/>
```

The generator class is responsible for creating the user interface and associated actions.

Related Topics

MenuItems Sample

PopupAction Sample

ToolbarButton Sample

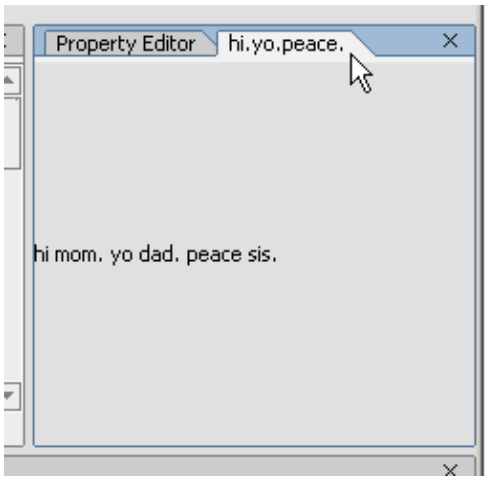
ActionSvc Class

DefaultAction Class

Adding Dockable Frames

You've probably noticed that most of the WebLogic Workshop's windows share characteristics. They have tabs displaying their names, they are dockable (you can drag a window from one part of the IDE to another and make it stick there), they feature close boxes, and so on. As extensions, these windows are known as "frames". You can use frames to provide a wide range of features, such as views on data in your extension, documents with which the extension is interacting, and so on. In particular, a frame is useful for displaying data that the user may want to get at often in the course of their work. For example, a frame could offer quick access to custom tool functionality.

The following simple frame example is created by the FrameViewSimple sample extension.



For frame extension samples, see FrameViewSimple Sample.

Building Frame Extensions

To add support for a new frame:

- Write an extension.xml file that defines one or more frames and their relationship with each other and the IDE's main window. The extension.xml file also connects your frame's logic and user interface with the IDE.
- Write one or more Java classes that support actions by implementing the required interfaces.

Extension XML for Frames

The extension XML for a frame extension specifies a few simple aspects of the frame's UI as well as specifying which of your extension classes contains the frame content and logic. The user interface characteristics specified in the extension XML are limited to customizing characteristics managed and displayed by the IDE; from a visual point of view, this means items at and beyond the frame's borders. In other words, you write Java code for the interior. The UI pieces defined in the XML include:

- The label on the frame's tab.
- The icon displayed for the frame in menus (where users can select to view the frame).
- How multiple frames should be arranged by default in the IDE.

For reference information on action extension XML, see Frame Extension XML Reference.

The following example shows the extension XML for a very simple single−frame extension.

```
<extension−definition>
    <extension−xml id="urn:com−bea−ide:frame">
        <frame−view−set>
            <frame−view class="ideExtensions.frameViewSimple.FrameView" label="hi.yo.peace."
                askavailable="true" />
            </frame−view−set>
    </extension−xml>
</extension−definition>
```

The <frame−view−set> stanza collects a set of frames; you can use an optional scope attribute to specify a circumstance under which the frames should be available for viewing. Each child <frame−view> element specifies the class that should be used for the frame's internal UI, and the label that should be used on the frame's tab. Including the askavailable attribute with a value of "true" tells WebLogic Workshop that you want it to query your implementation class's isAvailable method (as described below) to determine whether to make the frame available. Keep in mind that setting this attribute to "true" may also impact the performance of your extension (and the IDE in general) because of the frequency with which the IDE may actually do the asking.

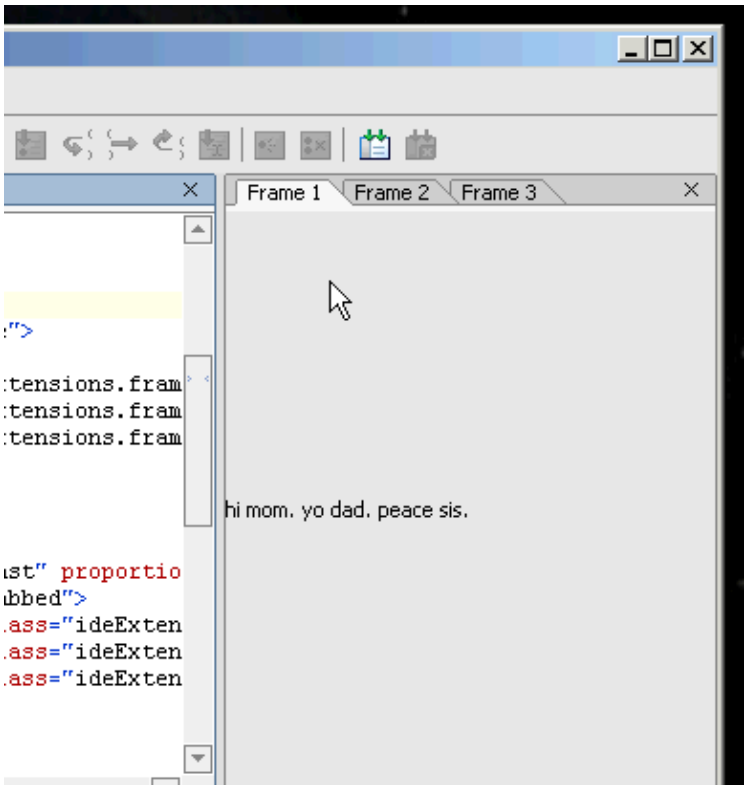
The value of the class attribute must be the fully−qualified name of a class that implements IFrameView; for more information, see Implementing Frame Logic, below.

Arranging Multiple Frames

When there are multiple frames, the <application−layout> stanza specifies their startup arrangement in the IDE.

Note: Of course, the user can rearrange the frames as they choose. Keep in mind, too, that the position of frames is stored among the IDE preferences. To reset the IDE to a default state (such as when you're debugging frame arrangements), you'll need to rename or delete the preferences file. This file is located at USER_HOME/.workshop.zpref.

At the right of the following illustration, you can see three frames grouped together. Each of these frames uses the same implementation of IFrameView, but they're referred to separately in the extension.xml.



The following XML defines startup positions for the frames in the illustration. Note that the frames themselves are specified in the <frame−view−set> stanza. Their arrangement at startup, however, is defined in the <application−layout> stanza.

```
<extension−xml id="urn:com−bea−ide:frame">

    <frame−view−set scope="main">
        <frame−view id="frame1" class="ideExtensions.frameViewSimple.FrameView"
            label="Frame 1" />
        <frame−view id="frame2" class="ideExtensions.frameViewSimple.FrameView"
            label="Frame 2" />
        <frame−view id="frame3" class="ideExtensions.frameViewSimple.FrameView"
            label="Frame 3" />
    </frame−view−set>

    <application−layout id="main">
        <frame−layout>
            <frame−container orientation="east" proportion="20%">
                <frame−view−ref id="frame1" class="ideExtensions.frameViewSimple.FrameView" />
                <frame−view−ref id="frame2" class="ideExtensions.frameViewSimple.FrameView" />
                <frame−view−ref id="frame3" class="ideExtensions.frameViewSimple.FrameView" />
            </frame−container>
        </frame−layout>
    </application−layout>

</extension−xml>
```

The <application−layout> stanza's id attribute specifies the IDE "mode" in which the group will be visible. Here, "main" means "not debugging"; a value of "urn:com−bea−ide:debug" would mean this group is available when the IDE is in debug mode.

The <frame−container> stanza specifies through its orientation attribute where in the IDE the group should be placed at startup. You can have multiple <frame−container> stanzas nested inside one another to create nested frames. Keep in mind when nesting containers that the orientation attribute is only valid on the root <frame−container> stanza. In other words, only the root can be responsible for the group's position in the IDE as a whole.

The <frame−view> elements, of course, specify what should be displayed inside the group. As you can see here, you can use the same frame implementation in multiple places as long as you disambiguate them with differing id attribute values.

Implementing Frame User Interface and Logic

The core of a frame logic is your implementation of the IFrameView interface, which has two methods: getView and isAvailable.

Implementing the getView Method

The IDE calls your implementation of the getView method to retrieve the user inteface to display inside the frame's borders. Your return value must be a class that extends java.awt.Component, a superclass for dialogs, panels, and UI you might want the IDE to display in your frame. Typically, you'll want to write and return a class that extends a Swing class such as JPanel or JScrollPane. These classes extend Component (through JComponent and others), and work well as containers for other Swing UI components such as buttons, boxes, and so on.

For more information from Sun on Swing components that act as containers, see http://java.sun.com/docs/books/tutorial/uiswing/components/components.html. For links to information on Swing, see Getting Started with UI Programming.

Implementing the isAvailable Method

You can have the IDE call your isAvailable implementation find out if your frame should be available for display. The IDE will call this method only if the corresponding <frame−view> element in your extension XML specifies an askavailable attribute whose value is "true".

Returning true from isAvailable signals to the IDE that it's okay to display your frame. So when would you want to return false? If your frame should be available only under certain circumstances, such as when a particular document type is open, or a particular project type is current, your isAvailable implementation should include code to discover if these things are true and return a value accordingly.

Keep in mind that prompting the IDE to call isAvailable (by setting the askavailable attribute to "true") can impair overalll performance because the call may occur often. You should only handle availability when your frame's usefulness relies on context.

APIs you might find useful in writing code for this method include:

- The IWorkspaceEventContext interface, with which you can get information about items selected in the application.
- The IWorkspace interface and Application class, through which you can get application context, such as a list of projects in the currently open application.

Related Topics

FrameViewSimple Sample

IFrameView Interface

FrameSvc Class

Frame Extension XML Reference

Adding New Project Types

You add support for a new project type by building a project type extension. By default, WebLogic Workshop includes support for project types such as Web Project, Control Project, Java Project, EJB Project, and others. You can add a new project type that features its own characteristics for allowable files, its own run time behavior, what the user gets when they create a new project of the type, and so on.

Strictly speaking, a project type extension itself defines only a few of the characteristics that might make up the user's sense of projects created from the type. The project type characteritics include high−level UI (such as the project type's name) and drivers for things like run−time behavior. But if projects of the type will include special debugging support for certain data types, support for new kinds of documents, a custom project properties panel, and so on, you will need to include that support in additional extensions. All of these related extensions can be packaged together, of course, sharing a single extension.xml file.

The following lists a few suggestions for other extension types you might want to include with a project type extension:

- With a project template you can have WebLogic Workshop generate a default set of project files when the user creates a new project from the type. The IDE does this, for example, with a Web Project, for which it generates libraries and starter source files through which the user can get started more quickly. For more information on developing templates, see Application and Project Templates.
- A document type extension supports a new document type. WebLogic Workshop knows how to handle files such as JCS and JSP files because corresponding document types exist for them. Through a document type, the IDE knows, for example, how to render Design View (if any) for the open file.
- A debugger extension could include debugging support for custom types your project supports. This might mean a custom view of the data in the Locals window, for example.
- You might want to include custom properties for your project. For more information, see Adding Support for Preferences.

Building Project Type Extensions

As with many other extensions, the heart of a project type extension is an extension.xml file that defines extension basics and connects the extension with the IDE.

Adding support for new type of project typically includes the following steps:

- Write an extension.xml file that specifies the extension's high−level user interface (such as Application window icons displayed for projects of the type), drivers the extension uses, and IDE attributes the project supports.
- Write Java code for the drivers, if any, that projects of this type will use. At minimum, you will need to provide a build driver in order for the project to be loaded into applications (the IDE won't load projects of the type without a build driver). Your drivers may use other Java classes as well, such as property change listeners.
- Where needed, create additional extensions to provide support for templates, new document types, and so on, as describe above.

Extension XML for Project Type Extensions

The extension.xml file for a project type extension specifies the following:

- High−level user interface, such as the icons to use in the Application window for an open and closed project.
- Attributes that the project type supports or declares for support by other components.
- Drivers that WebLogic Workshop should load for projects of the type. These might include, for example, a driver that tells the IDE how to build or run the project. You'll find information about driver implementation under Implementing Logic for Project Type Extensions.

The following example, from the CustomProject sample, illustrates a project extension XML for a "PHP" project type that specifies an attribute declaring support for PHP (see the section below on defininhg attributes) and specifies drivers that WebLogic Workshop should load with projects created from this type.

```
<extension−xml id="urn:com−bea−ide:project">
    <project−type id="urn:com−bea−ide:project.type:php" label="PHP"
        icon="images/workshop/sourceeditor/project/html.gif"
        openfoldericon="images/workshop/workspace/project/web_proj_o.gif"
        closedfoldericon="images/workshop/workspace/project/web_proj_c.gif">
        <attribute name="supportsPhp" value="true" />
            <driver type="com.bea.ide.workspace.project.IProjectDriver"
                class="ideExtensions.customProject.PhpProjectDriver"/>
            <driver type="com.bea.ide.workspace.project.IBuildDriver"
                class="ideExtensions.customProject.PhpProjectBuildDriver"/>
            <driver type="com.bea.ide.workspace.project.IRunDriver"
                class="ideExtensions.customProject.PhpProjectRunDriver"/>
    </project−type>
</extension−xml>
```

Defining and Using Attributes

In the context of a project extension, attributes are a way for the extension to declare support for some characteristic. That characteristic's meaning is defined in part by the components that query for it. For example, take a look at this attribute from the preceding example:

```
<attribute name="supportsPhp" value="true" />
```

In the sample, this attribute is used in two places. It's used in the PhpProjectListener as a way to discover if there are "projects that support PHP" in the application. Here's the code:

```
IProject[] phpProjects = workspace.getProjects("supportsPhp");
```

Theoretically, other PHP−related project types might include an attribute named supportsPhp whose value is "true"; these would also be found by the listener's code. A broader attribute, such as supportsWebApp (which is defined by the IDE), is an example of an attribute whose scope of use is fairly broad.

The supportsPhp attribute is also referred to by the document extension that accompanies this project extension. That extension's XML includes a <project−attributes> element with the following child element:

```
<requires name="supportsPhp" value="true" />
```

This tells the IDE that documents of this type should only be created within projects whose types include the supportsPhp attribute with a value of "true". In other words, you can right−click a PHP project to create a new PHP file, but you won't get that option for other project types unless their extensions declare the supportsPhp attribute.

In addition to supportsWebApp, there are other attributes defined by WebLogic Workshop. The IDE will query for this attribute's presence to discover whether a project may be added to a web app. For more information on attributes defined by WebLogic Workshop see the information about the <attribute> element in the Project Extension XML Reference.

Implementing Logic for Project Type Extensions

Within the context of a project extension alone, you'll be writing a Java class for each driver you want your project to use. A driver is loaded by the IDE for each project of the type. Drivers provide support for things like building and running the project, and for hooking in other code you want to have executing while a project of the type is present.

If you accompany the project with other extensions, of course, you'll have Java code for those, too. This section offers implementation guidelines for common drivers you might specify as part of a project type extension. For more on extensions used with project type extensions, see the list at the top of this topic.

Adding Project Build Support

Through a build driver, you collect the supporting information WebLogic Workshop needs to build your project. This includes the classpaths you want the IDE to use when building. You also provide supporting functionality, such as the property panel through which the user can set build properties and support through which the user can generate a build file by clicking "Export to Ant file" in the Build properties panel.

A build driver implements the IBuildDriver interface. WebLogic Workshop provides an implementation in the abstract DefaultBuildDriver class, which you can extend. Your driver implementation

Like IRunDriver, the IBuildDriver interface extends IProjectDriver, including the activate and deactivate methods. The IDE will call these methods when loading and unloading a project of your type. For more information, see Associating Code with a Project's Presence in an Application.

Implement IBuildDriver.getPathsPropertyPanel if your build driver supports configurable path settings (or return null if it does not). You should return an implementation of IProjectPropertyPanel. In a similar manner, you should return an IProjectPropertyPanel from IBuildDriver.getBuildPropertyPanel if your project type exposes custom build−related properties. For an IProjectPropertyPanel implementation example, see the FTPPrefsPanel class included with the PopupAction project in the IdeDevKit sample application included in the ExtensionDevKit.

Managing Build Classpath Settings

WebLogic Workshop provides APIs through which you can ensure that the project−, application−, and server−level classpaths are what they should be in order to build projects of your type. You set the project classpath settings yourself based on, for example, path properties your project type exposes to the user; application and server classpaths are managed by the IDE, but you get an opportunity to edit them. Finally, you can tell the IDE of any additions to the classpath for executing Ant tasks.

Specifying Project−Level Classpath

Your implementation of IBuildDriver.getProjectClassPath should return a similar array of strings that represent classpath additions as the project level. For example, if your project type provides a paths property panel (as described below) through which the user can add paths, you'll want to retrieve those paths and return

them in your implementation. Here's a simple example:

```
public String[] getProjectClassPath()
{
    // Retrieve the paths from preferences, where they were stored by the properties panel.
    Preferences pathsNode = _project.systemNodeForPackage(ProjectPathsPrefsPanel.class);
    String path = pathsNode.get("project.class.path", "");

    // Break the path into members of a String array.
    while (path.length() > 0 && path.endsWith(File.pathSeparator))
        path = path.substring(0, path.length() − 1);
    if (path.length() == 0)
        return new String[0];
    Pattern pathSplitter = Pattern.compile(";");
    String[] paths = pathSplitter.split(path);

    // Load the paths itno an array to return.
    for (int i = 0; i < paths.length; i++)
    {
        IFile f = m_project.getWorkspace().newFile(paths[i]);
        paths[i] = f.getAbsoluteIFile().getDisplayPath();
    }
    return paths;
}
```

Specifying Application− and Server−Level Classpaths

You can implement IBuildDriver.getApplicationClassPath to edit the application−level classpath used when building. WebLogic Workshop passes to your implementation a classpath that includes JAR files in the application's Libraries and Modules folders. Each member of the array is a path to the JAR on the path, such as: "C:/Apps/MyApp/APP−INF/lib/SomeLibrary.jar". If your application puts build output in the Libraries folder, for example, your getApplicationClassPath should remove it from the classpath and return the edited path. The following is an example:

```
public String[] getApplicationClassPath(String[] cpApplication)
{
    if (cpApplication == null)
        throw new IllegalArgumentException("cpApplication cannot be null");
    /*
     * Create a string that looks something like this:
     * "C:/bea_sp3/user_projects/applications/ExtensionTestApp/APP−INF/lib/ThisProject.jar"
     */
    String jarPath = _project.getWorkspace().getLibrariesDirectory().getDisplayPath()
        .replace(File.separatorChar, '/')
        .concat(_project.getName())
        .concat(".jar");

    /*
     * Loop through the list of paths, looking for the build output
     * JAR, removing it when it's found.
     */
    for (int pos = 0; pos < cpApplication.length; pos++)
    {
        if (cpApplication[pos].equals(jarPath))
        {
            ArrayList ncp = new ArrayList(Arrays.asList(cpApplication));
            ncp.remove(pos);
            cpApplication = (String[]) ncp.toArray(new String[ncp.size()]);
            break;
```

```
        }
    }

    return cpApplication;
}
```

In the same way, you can implement IBuildDriver.getServerClassPath if you want an opportunity to modify the server−level build classpath.

Specifying Ant Task Classpath

Lastly on the subject of classpaths, use IBuildDriver.getAntClassPath to ensure support for any custom tasks included in your project's Ant build file. Your implementation should return an array of absolute paths to items that should be on the classpath for executing Ant tasks.

Adding Project Run Support

Strictly speaking, a run driver simply provides a way for you to tell the IDE whether the Start and Stop buttons should be enabled, and what to do when the user clicks them (or uses the corresponding menu or keyboard commands). You do all this in a class that implements IRunDriver, a class that you specify in your project type's extension.xml, as shown in the extension.xml example above.

In addition to implementing IRunDriver, your run driver class must also provide a constructor that takes an IProject instance. When WebLogic Workshop opens an application in which a project of your type exists, it will construct your run driver, passing in an IProject representing the project. Multiple projects of the type mean multiple driver instances, each with a separate IProject instance. The IProject is handy for the information it contains about the project it represents, but it's also likely you'll use it when it's time to actually respond to a Start button click and run the project, as discussed below.

The IDE is likely to call your isRunnable implementation when the user navigates into your project. It calls this method to find out, not surprisingly, if the project is runnable. If your project requires certain state characteristics that the project or application has certain files, that a server is running, and so on this is a good place to ensure that conditions are suitable for running the project. Return true if they are.

In calling your doEnableChecks implementation, the IDE is prompting you to set the state of "run" controls the Start, Start Without Debugging, Pause, and Stop buttons and their corresponding menu and keyboard commands. You must explicitly use the run service to set these each time your project's run mode changes. Here's an example that enables the Start button and Start Without Debugging controls while disabling the Stop control:

```
RunSvc.get().setRunDebugEnabled(Boolean.TRUE);
RunSvc.get().setRunEnabled(Boolean.TRUE);
RunSvc.get().setStopEnabled(Boolean.FALSE);
```

WebLogic Workshop call the runProject and or runFile methods to get your project running. Your code in these should do whatever is necessary to put the project into run mode, including changing the state of the run controls. By default, the IDE will call runProject, passing to your code a boolean indicating whether the user wants to debug or not (i.e., clicked the Start button). Here's an example of a simple runProject implementation:

```
public boolean runProject(boolean debugRequested)
{
    boolean isRunnable = false;
```

```
    // Create a File object from a file name expected to be in the project.
    String indexFilePath = m_currentProject.getPath() + File.separator + "index.php";
    File indexFile = new File(indexFilePath);

    // If there's an index file, go ahead and run.
    if (indexFile.exists())
    {
        boolean isRunning = false;
        try
        {
            // Launch a browser to display the file, then set the buttons.
            Process browserProcess = BrowserSvc.get().invokeBrowser(fileUri.toURL(), true);
            RunSvc.get().setRunEnabled(Boolean.FALSE);
            RunSvc.get().setRunDebugEnabled(Boolean.FALSE);
            RunSvc.get().setStopEnabled(Boolean.TRUE);
            RunSvc.get().setRunningProject(m_currentProject);
            isRunning = true;
        } catch (MalformedURLException mue)
        {
            mue.printStackTrace();
        }
    } else
    {
        // If no index file, warn the user that they need one and don't run.
        showNoIndexAlert();
        isRunning = false;
    }
    return isRunning;
}
```

The runFile method, on the other hand, will only be called if *you* call it. This method is included in the interface to support scenarios where running the project and running an open file in the project are done in different ways. To see an example from an existing project type, look at a web project that includes a page flow, JSP files, and so on. When running a *project* of this type, you run the page flow controller, which specifies a file to use as a starting place. When running a *file*, such as a JSP file, you probably want to run the file without having to work through the page flow.

Under the covers, the IDE handles this by associating a popup action extension with JSP documents. Right−click a JSP and you get a "Run page" popup menu command designed to run just the file. Here's an example of what an IAction.actionPerformed implementation might look like for a popup command that featured a simple "Start" button:

```
public void actionPerformed(ActionEvent e)
{
    // Get the current project.
    IProject proj = RunSvc.get().getCurrentProject();
    if (proj != null)
    {
        /*
         * Get the run driver associated with the project and call
         * runFile with its URI.
         */
        IRunDriver runner = (IRunDriver)proj.getDriver(IRunDriver.class);
        IDocument currDoc = Application.getActiveDocument();
        if (currDoc != null)
            runner.runFile(currDoc.getURI(), true);
    }
}
```

Your stopProject method should take the project out of run mode and back to design mode. This will include resetting buttons to their design mode states.

For a run driver sample, see the PhpProjectRunDriver class in the CustomProject sample project.

Associating Code with a Project's Presence in an Application

Because IBuildDriver and IRunDriver extend IProjectDriver, build, run, and project drivers provide an opportunity to include code that WebLogic Workshop should execute when it discovers a project of your type in the open application, and code that it should execute when the project is about to be removed or the app is about to be closed. In other words, in all three you can implement activate and deactivate methods that will be called by the IDE as described below for project drivers. Note that if you provide implementations in all three drivers, they will be called in this order: build, run, project.

A project driver is commonly used to load listeners that should be present whenever the project is present in the application. Through a listener, you can capture events that WebLogic Workshop raises for documents, projects, the workspace, the file system, and and so on.

Implementing IProjectDriver

The two ends of the project's lifecycle in an open app are represented in the API by the IProjectDriver.activate and IProjectDriver.deactivate methods. Aside from implementing these, your implementation class also needs to provide a constructor that takes an IProject instance.

When WebLogic Workshop discovers a project of your type in the app, it will construct an instance of the type's driver with an IProject that represents the project itself. If there are multiple projects of the type in the app, there will of course be multiple instances of the driver, each with a separate IProject instance. You can use the IProject to find out more about the project's contents. For example, you can pass the IProject instance to code that uses it to retrieve properties specific to the project.

Listening for Events

WebLogic Workshop exposes events for the various abstractions it uses to represents items in the IDE. These include the following:

- Applications, through events exposed by the Application class.
- Projects, through events exposed by the IProject interface.
- Documents, through events exposed by the IDocument interface.

Within a project driver, listeners you add in the activate method should be removed in the deactivate method. To add a listener, you call the addPropertyChangeListener method.

```
Application.get().addPropertyChangeListener(Application.PROP_InitLevel,
    myListener);
```

For a sample project driver, see the CustomProject sample project in the IdeDevKit sample application. There, PhpProjectDriver implements IProjectDriver, adding and removing PhpProjectListener for projects of the PHP project type.

Related Topics

CustomProject Sample

Project Extension XML Reference

Developing Application and Project Templates

Application and project templates provide you with a way to pre−populate the New Application and New Project dialogs and to pre−load content into new applications and projects.

The following topics explain how to use application and project templates.

Topics Included in This Section:

Application and Project Templates

Application and Project Templates Reference

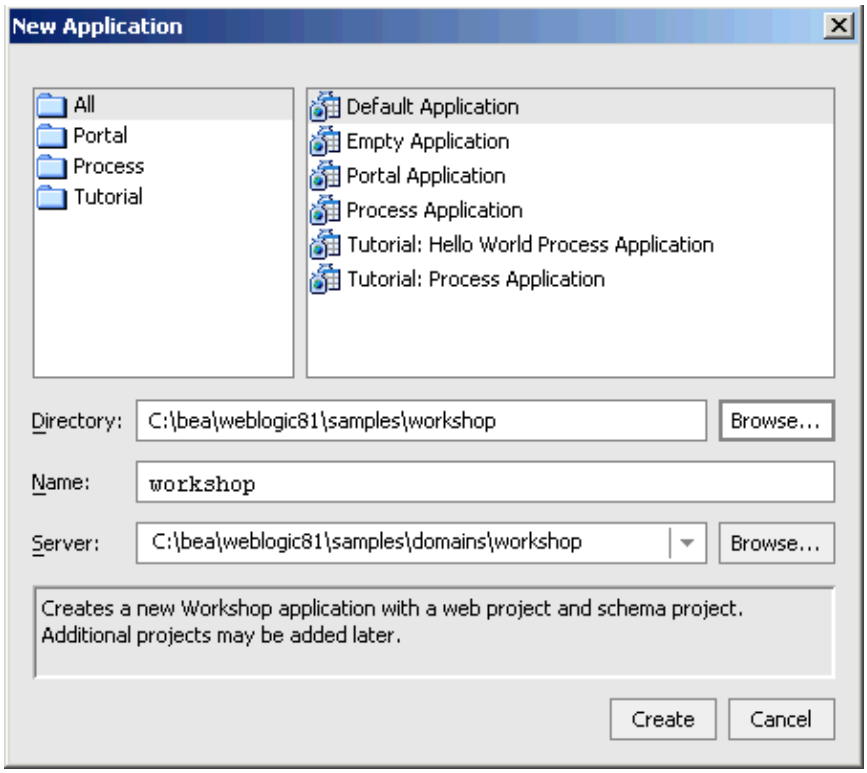
Application and Project Templates

This topic explains what application and project templates are and how to define custom templates in the WebLogic Workshop IDE.

What are Application and Project Templates?

Templates are a mechanism for pre−loading new WebLogic Workshop applications and projects into the IDE. The same mechanism is used by WebLogic Workshop whenever a user creates a new application or a new project in an existing application. A set of standard application templates are used to create web, portal and business process type applications. A set of standard projects templates are used to create Java class library, XML Schema libraries, EJB, business process, web service, web application and Java control projects.

To load an application template into the IDE, the user selects ***File−−>New−−>Application*** and is presented with the ***New Application*** dialog displayed below:



Users can select an application category on the left side of the dialog, and a particular template within that category on that right side of the dialog. A description of the template is displayed at the bottom of the dialog. User can also specify the directory where the application code will reside, the application name, and the application server.

To load a new project template into an already existing application, users select ***File−−>New−−>Project*** and complete the ***New Project*** dialog.

Defining Custom Application and Project Templates

Individual application and project templates are stored in the [BEA_HOME]/weblogic81/workshop/template directory. In order for individual templates to appear in the ***New Application*** and ***New Project*** dialogs, they must be present in the directory prior to the start of the IDE.

Templates are packaged as ZIP files containing (1) a template.xml file and (2) any other content included as part of the new application or project. Zip files in the /templates directory containing a template.xml file will be read each time a user opens a New Application, New Project, or Add Project dialog. A template.xml file may contain any number of project template (<project−template>) or application template (<application−template>) elements, each defining a entry displayed in the dialogs.

Note that the top−level template ZIP file may contain other ZIP files within it. For example, the application template ***Tutorial: Hello World Process Application*** (see the image above), is contained in the ZIP file wli−helloworld.zip, which, contains two ZIP files within it: default−project.zip and default−schemas−project.zip:

/templates/wli−helloworld/templates.xml
/templates/wli−helloworld/Schemas.jar
/templates/wli−helloworld/default−project.zip
/templates/wli−helloworld/default−schemas−project.zip

(1) The template.xml file defines which projects and other content are included in the application template.

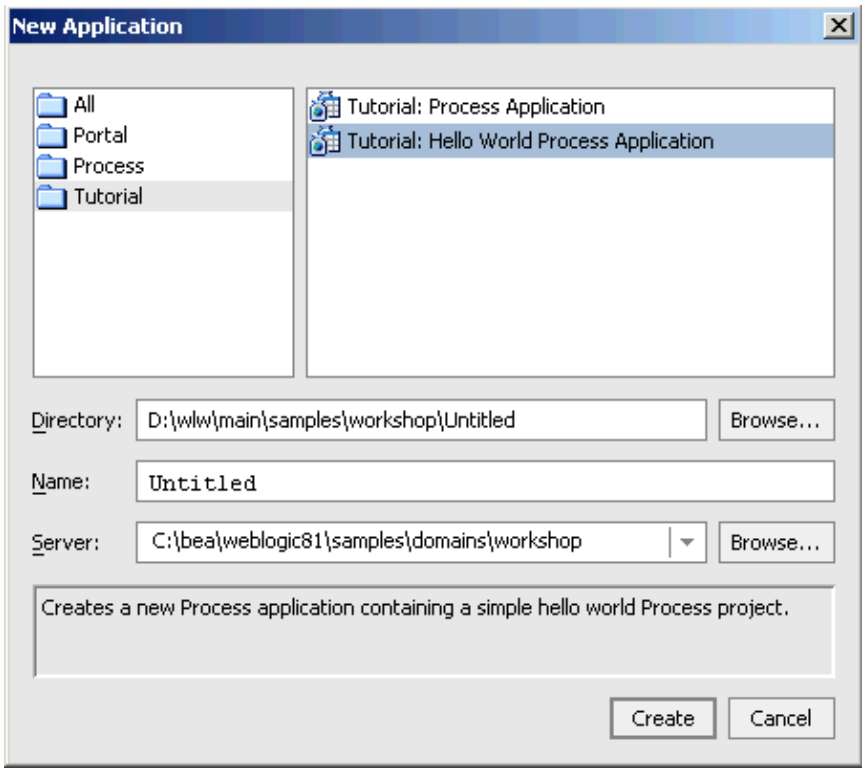
(2) The default−project.zip and default−schemas−project.zip files contain all the source code content that is included in this application template. For example, a simple process, HelloWord.jwf, is included in this template, as well as a set of schema files.

The template.xml contains the following template definition:

```
<template−definition>

    <project−template id="hello_world_process_project"
        type="urn:com−bea−ide:project.type:WebApp">
        <content type="archive" destination="project" source="default−project.zip"/>
    </project−template>

    <project−template id="hello_world_schemas"
        type="urn:com−bea−ide:project.type:Schema">
        <content type="archive" destination="project" source="default−schemas−project.zip"/>
        <content type="file" destination="libraries" source="Schemas.jar"/>
    </project−template>

    <application−template id="hello_world_process_app">
        <display
            location="newdialog"
            label="Tutorial: Hello World Process Application"
            description="Creates a new Process application containing a simple hello world Proc
            priority="20"
            categories="Tutorial"/>
        <project−template−ref type="urn:com−bea−ide:project.type:WebApp" template="hello_world_
        <project−template−ref default−name="Schemas" type="urn:com−bea−ide:project.type:Schema"
    </application−template>

</template−definition>
```

As illustrated above each <template−definition> element can contain two types of templates, application and/or project. In this case the definition contains both type but could contain one or the other.

The <application−template> above is displayed in the New Application dialog in the following way.



The template.xml file can also define any number of projects along with their source code content to pre−load in the IDE. Also, each <project−template> element has an id attribute that must uniquely identify it from all other project templates. The id allows an <application−template> to reference <project−templates> in order to pre−load a new application with specific projects and their content. Notice how the <application−template> element contains two <project−template−ref> elements that reference corresponding <project−template> elements.

Note that the referenced <project−template> elements can exist in the same template.xml definition or in a completely different template.xml definition in a different template ZIP file. In other words, a <application−template> can potentially reference all project templates contained in the WebLogic Workshop /template directory.

Extending Templates

Note that a project (or application) template can extend another project (or application) template. Display information in the extended template will not be inherited. But content from the extended template will be inherited. For example assume that you have a project template with id="baseProj".

```
<project−template id="baseProj" type="urn:com−bea−ide:project.type:WebApp">
    ...
</project−template>
```

Another project template can extend this project template by pointing to its id.

```
<project-template id="someOtherProject" type="urn:com-bea-ide:project.type:WebApp" extends=
    ....
</project-template>
```

Note that the extended and the extending project templates must be of the same type.

The extended and extending templates need not exist in the template.xml file, nor need they exist in the same ZIP file. When Workshop encounters an extending template, it will search all template.xml files within [BEA_HOME]/weblogic81/workshop/template for the extended template.

Localizing Templates

There is currently no support for automatic localization of templates: application and project templates must be localized manually. Template developers should produce a different template ZIP file for each location.

Related Topics

template.xml Reference

template.xml Reference

A Workshop template defines a set of files and/or code to be used to check the contents of or add content to a Workshop application or project.

At startup, Workshop loads its set of templates from the 'templates' directory below the Workshop root directory (BEA_HOME/weblogic81/workshop/templates). Templates are defined in zip files. Each zip file containing templates must include a template.xml file at the root level of the zip file.

A template.xml file may contain any number of project or application template definitions. A template definition may optionally include display information indicating where and how the template will be displayed to the user. All templates, regardless of whether they contain display information, may be accessed programmatically by extension writers, or be extended or referenced by other template definitions.

Templates can be displayed to the user in the following locations:

Application Templates

- New Application Dialog
- Application tab root−node context menu

Project Templates

- New Project Dialog
- Import Project Dialog
- Application tab project−node context menu

Templates displayed in application tree context menus will appear in the Install submenu.

template.xml File Elements

```
<template−definition>
   <project−template>
      <display>
      <content>
   <application−template>
      <display>
      <content>
      <project−template−ref>
```

<template−definition> Element

The top−level element for application and project templates. The <template−definition> element can contain any number of <project−template> and <application−template> elements.

```
<template−definition>
      <project−template>
      <application−template>
```

Syntax

```
<template−definition>
    <!−− Children <project−template> and <application−template> elements. −−>
</template−definition>
```

Hierarchy

Parents: None.

Children: <project−template>, <application−template>.

Related Topics

None.

<project−template> Element

A project template defines a set of content for populating a single new or existing project of a specific project type. A project template can also use a custom template processor class located in an extension jar to call the IDE extension API, examine and update configuration files, query the user for input, or control the addition of content to a source directory.

Project template definitions may be displayed in the New Project Dialog, Import Project Dialog, or in the application tree project−context menu under Install. They may also be extended by other project templates and referenced by application templates.

A project template is used by an IProjectTemplateProcessor to examine, configure or populate a project instance. IProjectTemplateProcessor has two methods: check() which determines if the project conforms to the template, and
load() which configures or adds content to the project. A custom IProjectTemplateProcessor implementation may be specified in the template definition; otherwise the default Workshop implementation will be used.

Project templates may extend other project templates. See the extends attribute below.

<template−definition>
  <project−template>
    <display>
    <content>

Syntax

```
<project−template
    type="projectType"
    id="projectID"
    [extends="extendedProjectID"]
    [processor="IProjectTemplateProcessor"]
    [default−name="defName"]
>
```

Attributes

Attribute	Description
type	Required string. Identifies the type of project that will be created. Project types are defined in extension.xml and define how the project builds, deploys and runs, as well as how this template will be processed. See the table below for possible values.
id	Required string. Uniquely identifies this template among all templates defined for its project type. Note that this id does not have to be unique among all project templates.
extends	Optional string. Points to the id of a project template of the same type that this template extends. If this attribute is set, then uninitialized, non−required attributes will be set to values from the extended template. Category lists will always be inherited. Content from the extended template will be included in this template, but display information will not be inherited.
processor	Optional string. Class name of an IProjectTemplateProcessor implementation. This class will be instantiated when extension code calls IProjectTemplate.createProcessor(). If this attribute is omitted, then a default processor implementation will be created.
default−name	Optional string. The name that will be used for the project if one is not provided by the user.

Hierarchy

Parents: <template−definition>.

Children: <display>, <content>.

Related Topics

None.

<display> Element

Specifies the display options for the New Project dialog.

<template−definition>
  <project−template>
    <display>

Syntax

```
<display
    location=" newdialog | importdialog | contextmenu "
    label="projLabel"
    [icon="imageName"]
    [description="projDescription"]
    [priority="menuLocationInteger"]
    [categories="projCategory1, projCategory2, ..."]
>
```

Attributes

Attribute	Description
location	Required string. Comma separated list of display location identifiers. Workshop−defined values: newdialog − the New Project dialog importdialog − the Import Project dialog contextmenu − the project context menu
label	Required string. Label that will be displayed in the New Project dialogs.
icon	Optional string. Icon that will be displayed for this template. All project types will specify a default icon which will be used if this attribute is omitted. The value of this attribute may be a GIF file in the template ZIP file, or it may be a resource in an extension JAR file.
description	Optional string. Description of this template that will be displayed in the Project dialogs.
priority	Optional integer. Used to order the templates displayed in the dialog template list, higher values being displayed before lower values.
categories	Optional string. Comma separated list of template category names.

Hierarchy

Parents: <project−template>.

Children: none.

Related Topics

None.

<content> Element

Specifies a menu item.

<template−definition>
  <project−template>
    <content>

Syntax

```
<content
    type=" archive | file "
    destination=" project | libraries | modules "
    source="sourceFileOrZIP"
    [overwrite=" true | false "]
>
```

Attributes

Attribute	Description
type	Required string. Indicates how the source file should be processed. Values supported by the default template processor:

	archive − a zip file file − a file
destination	Required string. Where the content will be placed. Values supported by the default template processor: project − the root of the project directory libraries − the application Libraries directory modules − the application Modules directory
source	Required string. The content source file; when opening as a stream, will first look in the template try to open the content as a resource.
overwrite	Optional boolean. Specifies the behavior if same files already exist in the application directories. is omitted, the value is false. The default template processor will immediately quit processing thi element if overwrite is false and a collision occurs.

Hierarchy

Parents: <project−template>.

Children: none.

Related Topics

None.

<application−template> Element

An application template defines a set of content for populating a new or existing Workshop application. An application template can also use a custom template processor class located in an extension jar to call the IDE extension API, examine and update configuration files, query the user for input, or control the addition of content to the application directory.

Application template definitions may be displayed in the New Application Dialog, or in the application tree root context menu under Install. They may also be extended by other application templates and referenced by application templates.

Application templates may also contain any number of content elements, however these content members may only contain application−level data: data in the Libraries, Modules folders, etc. Project content must be specified in a project template.
Project templates may be referenced, or they may be completely defined inside the application template. Project templates defined inside the application template will not appear in the New Project and Add Project dialog and may only be used
when creating this application.

```
<template−definition>
  <application−template>
    <display>
    <content>
    <project−template−ref>
```

Syntax

```
<application−template
    id="uniqueID"
    extends="extendedAppID"
    processor="processorClass"
>
```

Attributes

Attribute	Description
id	Required string. Uniquely identifies this application template among all application templates defined in this template zip file. Note that this id does not have to be unique among all application templates loaded by Workshop.
extends	Optional string. Points to the id of another application template which the current template extends. Content from extended template will be included in the extending template.
processor	Optional string. If no processor is specified, the default processor will be used: IProjectTemplateProcessor.

Hierarchy

Parents: <template−definition>.

Children: <display>, <content>, <project−template−ref>.

Related Topics

None.

<display> Element

The display element specifies display information to be used when displaying this template at a given location or locations.
An application template may contain any number of display elements, each specifying one or more display locations.

<template−definition>
  <application−template>
    <display>

Syntax

```
<display
    location=" newdialog | contextmenu "
    label="appLabel"
    [icon="imageName"]
    [description="appDescription"]
    [priority="menuLocationInteger"]
    [categories="appCategory1, appCategory2, ..."]
>
```

Attributes

Attribute	Description
location	Required string. Comma separated list of display location identifiers. Workshop−defined values: newdialog − the New Application dialog contextmenu − the project context menu
label	Required string. Label that will be displayed in the New Application dialog.
icon	Optional string. Icon that will be displayed for this template. A default application template icon is provided by Workshop if this attribute is omitted. The value of this attribute may be a GIF file in the template ZIP file, or it may be a resource in an extension JAR.
description	Optional string. Description of this template that will be displayed in the New Application dialog.
priority	Optional integer. Used to order the templates displayed in the dialog template list, higher values being displayed before lower values.
categories	Optional string. Comma separated list of template category names.

Hierarchy

Parents: <application−template>.

Children: none.

Related Topics

None.

<content> Element

Specifies a unit of content to be loaded when the the application is created.

<template−definition>
  <application−template>
    <content>

Syntax

```
<content
    type=" archive | file"
    destination=" libraries | modules "
    source="sourceFileOrZIP"
    [overwrite=" true | false "]
>
```

Attributes

Attribute	Description
type	Required string. Indicates how the source file should be processed.

	Values supported by the default template processor: archive − a zip file file − a file
destination	Required string. Specifies where the content will be placed. Values supported by the default template processor: libraries − the application Libraries directory modules − the application Modules directory
source	Required string. The content source file; when opening as a stream, will first look in the template try to open the content as a resource.
overwrite	Optional. Specifies the behavior if same files already exist in the application directories. If this at omitted, the value is false. The default template processor will immediately quit processing this c if overwrite is false and a collision occurs.

Hierarchy

Parents: <application−template>.

Children: none.

Related Topics

None.

<project−template−ref> Element

A reference to an externally defined project template. The referenced project template may be defined in the current template ZIP file, or any other template ZIP file in the /template directory. The project type id and template id are used to find the referenced project template. The referenced project template will be used to create a project with the name given and the content defined by the project template.

<template−definition>
  <application−template>
    <project−template−ref>

Syntax

```
<project-template-ref
    default-name="name"
    type="projType"
    template="projTemplateID"
>
```

Attributes

Attribute	Description
default−name	Required string. The name that will be used for the created project.
type	Required string. A project type id. See the table below for possible values.
template	Required string. A project template id.

Hierarchy

Parents: <application−template>.

Children: none.

Related Topics

None.

WebLogic Workshop Project Types

Type	Description
urn:com−bea−ide:project.type:Datasync	Datasync projects contain server data including campaigns, content, placeholders, content selectors, user segments, and property sets.
urn:com−bea−ide:project.type:Control	Control projects are used to create Java controls and packaging them as JAR files.
urn:com−bea−ide:project.type:WebApp	WebApp projects can contain web applications, web services, and business processes.
urn:com−bea−ide:project.type:EJB	EJB projects are used to create EJBs and packaging them as JAR files.
urn:com−bea−ide:project.type:PortalWebApp	A WebApp project that also enables Portals. This should only be added to an portal enabled application.
urn:com−bea−ide:project.type:Java	A project for developing Java applications.
urn:com−bea−ide:project.type:Schema	This project stores XSD files and compiles them into XMLBean

	classes.

Example

The following is an annotated example of a template.xml file.

```
<template-definition>

<!-- This a Web project template and extends the 'default' Web project template. -->
    <project-template id="_example_proj1_"
            type="urn:com-bea-ide:project.type:WebApp"
            extends="default"
            processor="workshop.workspace.project.TestTemplateProcessor">
        <!-- Defines where and how this template will be displayed to the Workshop user. -->
        <display
            location="newdialog,importdialog"
            label="_Example Project_"
            description="This project template demonstrates the Workshop template syntax, and e
            icon="exampleProject.gif"
            priority="0"
            categories="_Example_" />
        <!-- zip to be extracted at the root of the project directory -->
        <content type="archive" destination="project" source="default-project.zip"/>
        <!-- file to be put in the libraries directory -->
        <content type="file" destination="libraries" source="CreditScoreBean.jar" overwrite="tr
        <!-- file to be put in the modules directory -->
        <content type="file" destination="modules" source="CreditScoreEJB.jar" overwrite="true"
    </project-template>

    <!-- This a Web project template that will appear in the Application tree's Install context
         when the user right-clicks on a Web project folder.
         Note that this template uses a custom template processor implementation.  This class c
         query the user for input, examine and update existing configuration files, and control
         content elements are added to the project. -->
    <project-template id="_example_proj1_installmenu_"
            type="urn:com-bea-ide:project.type:WebApp"
            processor="workshop.workspace.project.InstallTemplateProcessor">
        <!-- Defines where and how this template will be displayed to the Workshop user. -->
        <display
            location="contextmenu"
            label="_Add Project Resources_"
            description="This project template demonstrates the Workshop template syntax, and e
            icon="exampleProject.gif"
            categories="_Example_" />
        <!-- zip to be extracted at the root of the project directory -->
        <content type="archive" destination="project" source="ui_resouces.zip"/>
    </project-template>

<application-template id="_example_app1_">
        <display
            location="newdialog"
            icon="exampleApp.gif"
            label="_Example Application 1_"
            description="This application template demonstrates the Workshop template syntax. S
            priority="1"
            categories="_Example_" />
        <!-- zip to be extracted in the libraries directory -->
        <content type="archive" destination="libraries" source="libraries.zip"/>
        <!-- file to be put in the modules directory -->
        <content type="file" destination="modules" source="testEJB.jar"/>
        <!-- references a project template defined in another template zip file -->
```

```
        <project-template-ref default-name="_StandardControl_" type="Control" template="default
        <!-- references a project template defined above -->
        <project-template-ref default-name="_ExampleWebApp_" type="urn:com-bea-ide:project.type
        <!-- project template only used in this application template -->
        <project-template default-name="_ExampleJava_"
                id="_example_app1_proj1_"
                type="Java">
            <content type="file" destination="project" source="Example.java"/>
        </project-template>
    </application-template>

    <!--
        This application template extends an application template defined in this
        template zip.  Currently we only support extending app templates defined
        in the same template zip.
      -->
    <application-template id="_example_app2_"
            extends="_example_app1_">
        <display
            location="contextmenu"
            label="_Example Application Content_" />
        <!-- A project template only used in this application template.
             Extends the default control project -->
        <project-template default-name="_ExampleControl_"
                id="_example_app2_proj1_"
                type="Control"
                extends="default">
            <content type="file" destination="project" source="Example.java"/>
        </project-template>
    </application-template>

</template-definition>
```

Related Topics

Application and Project Templates

IProjectTemplateProcessor

Adding Support for Drag and Drop

You can add support for drag−and−drop to your extensions. Drag−and−drop, also known as DnD, is a feature that allows the user to "drag" a portion of an application's user interface somewhere else (usually to another part of the user interface) and "drop" it there. The application's DnD support specifies what can be dragged, where it can be dropped, and what happens when dragging and dropping.

The WebLogic Workshop extensibility API provides IDE−specific wrappings for DnD functionality. These classes and interfaces make it easier to support DnD in the IDE.

Note: A full description of the mechanics of DnD is beyond the scope of this topic. For a general introduction to DnD in Java, see How to Use Drag and Drop and Data Transfer at the Sun web site.

For a sample that illustrates DnD support, see the DragDropSimple Sample.

DnD in WebLogic Workshop

The interesting DnD classes and interfaces are located in the com.bea.ide.core.datatransfer package. To add support for DnD to your extension, you'll typically do the following:

- Implement the IDragDropDriver interface to manage the DnD operation.
- Implement the IDragDropDriver.IDragSourceInfo interface to provide methods through which the IDE can retrieve the image you want displayed while dragging, the information that should be transferred on dropping, and so on.
- Register your component, along with IDragDropDriver implementation, as supporting DnD. You do this by calling the DataTransferService.I interface's registerDnDSupport method.

Implementing IDragDropDriver

In some cases, the easiest way to implement IDragDropDriver is to simply extend DefaultDragDropDriver, a default implementation provided in the API. To provide the component−specific data that will be transferred, override the getDragInfo method to return your implementation of IDragDropDriver.IDragSourceInfo. When calling getDragInfo, the IDE will pass in a Component instance representing the component dragged from the drag source. You can use this instance to retrieve the data (or other component containing the data) that should be transferred by the DnD operation. Using what you know about the drag source to create an instance of your IDragSourceInfo implementation.

For example, the DragDropSimple sample project receives a JTree through getDragInfo, then extracts the currently selected tree node to get the data that should be transferred.

Implementing IDragDropDriver.IDragSourceInfo

Like IDragDropDriver, IDragSourceInfo has a default implementation in the DefaultDragDropDriver.DragSourceInfo class. The interesting method in this interface is getTransferable, which the IDE calls to retrieve a Transferable implementation that contains the data to be transferred. If the data you're transferring is a string, you might want to skip implementing Transferable in your own class and create an instance of StringSelection, which implements the interface to support string transfers. In any case, your code should populate the Transferable implementation with the data to be transferred, then pass it back with getTransferable so that the IDE can handle the rest.

Registering the Component as DnD−able

With DnD driver in hand, you register your component with the IDE for DnD support, passing along an instance of the driver. The following example tells the IDE that a JTree created elsewhere in the code should be registered for DnD support.

```
DataTransferSvc.get().registerDnDSupport(tree,
    new SimpleTreeDragDropDriver(),
    javax.swing.TransferHandler.COPY,
    true);
```

Related Topics

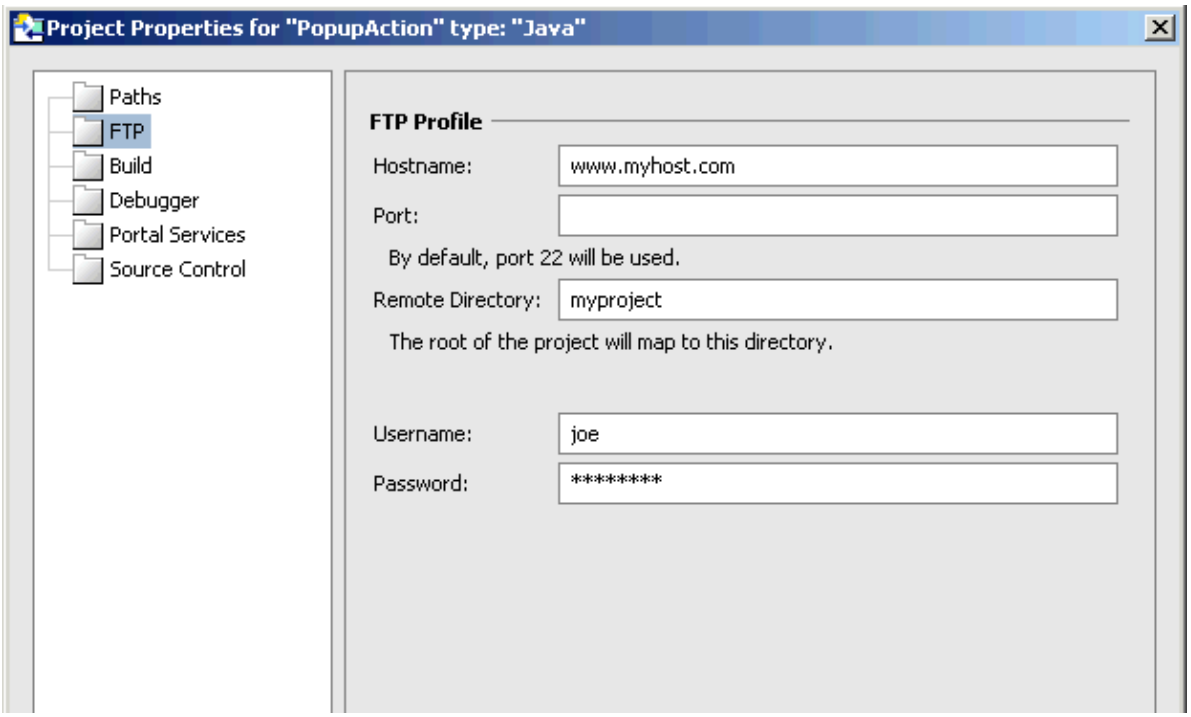
DragDropSimple Sample

Adding Support for Preferences

You can create a new properties panels that appear in a Properties dialog through a properties extension. You can also use the preferences−related API to store and access information that your extension needs but may not expose to the user as a properties panel.

For example, imagine that you've created an extension to provide support for an external tool, such as a source control manager. You'd likely want to provide a preferences extension also so that users can set the path to the source control server, user name and password, and so on.

The following illustrates a properties panel created by the PopupAction sample extension.



Note: WebLogic Workshop uses the terms "properties" and "preferences" in various contexts, and it can be confusing at first. For extensions that support application, project, and IDE properties, they are exposed as "properties" in the user interface (through panels in the Properties dialog), but defined as preferences extensions. This is reflected in the extension XML, which uses terms such as "project−preferences". In addition, you should not confuse preferences as they're referred to here the the properties exposed by the IDE and described in the PropertyListener sample project.

Building Preferences Extensions

To build a preferences extension, you:

- Write an extension.xml that specifies whether the properties panels to add are scoped to a project, application, and/or IDE. This file also specifies which classes in the extension are to be used for the panels UI and logic.
- Write Java classes that support the preference by providing the panel's user interface by implementing the required interfaces.

Scope for Properties

WebLogic Workshop provides dialogs for setting properties scoped to the applications, projects, and the IDE. The scope of the properties you're handling impacts how you define and handle them. For example, project properties should be defined as such in the extension XML and handled through an instance of the IProject interface. For more information about these differences, see the following sections.

Extension XML for Preferences

The extension XML for specifying properties panels is among the simplest you'll write. One of three elements <ide−preferences>, <workspace−preferences>, and <project−preferences> sets the scope for the properties the panel will expose. In the <panel> element, you specify the label that should be used in the left side of the Properties dialog. You also tell the IDE what extension class to use for the panel's logic. The class you specify should extend JPanel and implement IPropertyPanel, as described in Implementing Properties Handling Logic, below.

The following example shows the extension XML for a properties panel scoped to an application (also known as a workspace).

```
<extension−xml id="urn:com−bea−ide:settings">
    <workspace−preferences>
        <panel label="PHP" class="com.myco.properties.PHPPanel"/>
    </workspace−preferences>
</extension−xml>
```

For reference information on extension XML for properties extensions, see Preferences Extension XML Reference.

Implementing Properties Handling Logic

Your properties panel will implement the IPropertyPanel interface or an interface that extends it. This interface provides methods through which you can interact with the IDE to retrieved saved values, validate and store changed values, and handle the user's canceling the dialog.

Implementing IPropertyPanel

Your extension class should extend JPanel and implement IPropertyPanel (or an interface that extends it, such as IProjectPropertyPanel). The constructor is a good place to assemble the user interface. The IDE calls your implementation of the IPropertyPanel interface's loadProperties method. Here, your code retrieves saved property values and puts the values into the user interface components that will display them.

If the user clicks OK in the properties dialog to save the values, the IDE calls your implementation of the validateEntries method. Your code should use this method to retrieve values from the user interface components and determine if they're suitable as values for your properties. If they are, you should return true from the method. If they aren't, you should display an error message (a message that helps the user figure out what they need to do to correct the situation), then return false from the method. The user will not be able to dismiss the dialog by clicking OK until you return true from this method.

If your code returns true from the validateEntries method, the IDE will call your implementation of the storeProperties method. Here, your code should store values retrieved from the panel's user interface

components.

If the user clicks the Cancel button in the dialog instead of OK, then the IDE will call your implementation of the cancel method. Here, you can release any resources you may have acquired.

Accessing and Saving Preference Values

Your implementation of the loadProperties method should retrieve saved values and display them in your panel's user interface. While you don't need to use WebLogic Workshop's preferences storage mechanism, that API does provide a convenient way for you to set and get property values.

WebLogic Workshop provides a scope−specific API for storing and retrieving value sets in the form of java.util.prefs.Preferences instances. You can use this API to retrieve a Preferences instance that contains existing stored values, then use the instance to get and set the values in your panel's code. Depending on the scope of your properties panel, you can use one of the following to retrieve the preferences instance:

- IProject for project properties.
- IWorkspace for application properties.
- Application.I for IDE properties.

Each of these extends IPreferencesSupport. When using one of these APIs you'll typically call either the systemNodeForPackage or userNodeForPackage method to retrieve a Preferences instance through which to access and store property values. You'll pass to the method a Class representing your properties panel. You store and retrieve individual properties through get and put methods using key/value pairs, where the key is some unique value representing the property.

For example, to get a property value:

```
private IProject m_project;
Preferences prefs = m_project.systemNodeForPackage(FTPPrefsPanel.class);
String hostname = prefs.get(FTPSettings.HOSTNAME, ""));
```

Related Topics

PopupAction Sample

Preferences Extension XML Reference

Extension XML Reference

An extension.xml file contains the descriptors for WebLogic Workshop extensions. You include the extension.xml file in the META−INF folder of your extension JAR file. When then JAR file is copied to the extensions folder of a WebLogic Workshop installation (<WORKSHOP_HOME>/extensions), the IDE examines the extension.xml on startup, identifies the extensions the JAR contains, and loads them.

The root element for the XML in this file is <extension−definition>. The <extension−definition> element may have one or more <extension−xml> elements as child elements, each describing a different extension. You specify the particular type of extension described by setting the value of the <extension−xml> element's id attribute, as described in the <extension−xml> element section of this topic.

The following simple example describes an extension that adds three panels for setting properties: one for the Project Properties dialog, one for the Application (workspace) Properties dialog, and one to the IDE Properties dialog.

```
<extension−definition>
    <extension−xml id="urn:com−bea−ide:settings">
        <project−preferences>
            <panel label="%strings.workshop.debugger.extension.debuggerTab%" class="workshop.de
        </project−preferences>
        <workspace−preferences>
            <panel label="%strings.workshop.debugger.extension.debugSourcepath%" class="worksho
        </workspace−preferences>
        <ide−preferences>
            <panel label="%strings.workshop.debugger.extension.debuggerViews%" class="workshop.
        </ide−preferences>
    </extension−xml>
</extension−definition>
```

<extension−definition> Element

The root of an extension.xml file. Each <extension−xml> child element should describe a different extension type.

Hierarchy

Parents: None.

Children: <extension−xml>.

<extension−xml> Element

The root element for a WebLogic Workshop extension descriptor. Set the id attribute to designate the type of extension described. Because the descriptor elements for each extension type follows a different schema, a given id attribute value will correspond to a specific XML shape within the <extension−xml> element.

Syntax

```
<extension−xml id="urnForExtensionType">
    <!−− Extension−specific XML. −−>
</extension−xml>
```

Attributes

Attribute	Description
id	Required string. The URN for the specific kind of extension that this <extension−xml> element describes. Possible values are given in the Remarks section.

Hierarchy

Parents: <extension−definition>.

Children: <file−extension>, <create−template>, <project−attributes>.

Remarks

The <extension−xml> element may contain XML for any of the following extension types. Note that only one extension type may be defined within a given <extension−xml> element. However, multiple <extension−xml> elements may occur as children of the <extension−definition> element, which is the root element of an extension.xml file.

The following table lists the extension types that an extension.xml file may describe. Use the extension type links to see reference information about the XML shape for that type.

Extension Type	Extension Description	URN
Actions	Defines menus, popups, and toolbars, along with their associated behavior.	urn:com−bea−ide:actions
Debugger Expression	Defines support for new views of variable data while debugging.	urn:com−bea−ide:debugExpressionViews
Document	Defines support for a document type to the IDE.	urn:com−bea−ide:document
File Encoding	Defines support for a file type.	urn:com−bea−ide:encoding
Frame	Defines a frame view a dockable window in the IDE.	urn:com−bea−ide:frame
Help	Defines paths to search for help topics when context−sensitive help is requested.	urn:com−bea−ide:help
Preferences	Defines a new properties panel.	urn:com−bea−ide:settings
Project	Defines a new project type.	urn:com−bea−ide:project

Related Topics

None.

Action Extension XML Reference

Describes an action extension, with which you create menus, popups, and toolbars. For more information on build action extensions, see Adding Menus and Toolbar Buttons.

Note: The <extension−xml> element is the root for any extension descriptor. For this kind of extension, the <extension−xml> element's id attribute value must be "urn:com−bea−ide:actions".

<extension−xml>
  <action−ui>
    <action−group>
      <menu>
      <action−ref>
    <menu>
      <action−group>
    <popup>
      <action−group>
    <toolbar>
      <action−group>
  <action−set>
    <action>
      <location>

See the ActionSvc class for more information on implementing document extensions.

<action−ui> Element

Specifies user interface details for actions in the IDE.

In general, you should use this element for defining new user interface (menu and popup commands, toolbar buttons) associated with actions.

<extension−xml>
  <action−ui>

Syntax

```
<action−ui>
    <!−− Children that describe specific actions or groups of actions. −−>
</action−ui
```

Hierarchy

Parents: <extension−xml>.

Children: <action−group>, <menu>, <toolbar>, <popup>.

Related Topics

None.

<action−set> Element

Specifies actions in this extension. May occur multiple times as a child of <extension−xml>. Specific actions, such as a menu or popup command, or a toolbar button, are described by each <action> child element.

For more information about defining action sets and actions, see the ActionSvc class.

<extension−xml>
  <action−set>

Syntax

```
<action−set
    scope="classNameForView"
    extends="classNameForExtendedView"
>
```

Attributes

Attribute	Description
scope	Optional string. The fully−qualified class name for a view (a document, document view or frame view) that must be active in order for this action to be available. Default is global scope, meaning that the actions in this set may be accessible to the user regardless of the active view.
extends	Optional string. The fully−qualified class name for a view whose scope this action set extends. Extending an action−set of an existing scope adds all the actions of the base scope into the extending scope.

Hierarchy

Parents: <extension−xml>.

Children: <action>.

Related Topics

None.

<action−group> Element

Simply provides a way to group actions. In the IDE, grouped actions are visually separated from actions outside their group. If no actions are put in the action group, then it will not be visible in the UI. A separator will be shown between each visible command group.

<extension−xml>
  <action−ui>

<action−group>

Attributes

Attribute	Description
generator	String.
id	Required string. The name of this item.
path	Required string. The path in the IDE user interface to this group.
priority	A number indicating this item's priority level relative to other items rendered with with.

Hierarchy

Parents: <action−ui>.

Children: <menu>, <action−ref>.

Related Topics

None.

<menu> Element

Specifies a menu item.

```
<extension−xml>
  <action−ui>
    <action−group>
      <menu>

<extension−xml>
  <action−ui>
    <menu>
```

Syntax

```
<menu
    id="nameToUseInContextPaths"
    label="labelInIDE"
    [path="uiContextPath"]
    priority="priorityNumber"
>
```

Attributes

Attribute	Description
id	Required string. The name of this item. The full context for this action item is determined by con attribute with the path attribute.
label	Required string. The text to display for the menu in the IDE. You can also indicate the action's de key−stroke accelerator by preceding it with '@'. If the label begins and ends with the '%' escape c label is interpreted as a Java PropertyBundle path. The label (including accelerator) will be deter

	the ResourceSvc to load the property named by the path. Place '&' before the character whic menu mnemonic.
path	Optional string. A path through the UI hierarchy, followed by the final group in the last menu. Fo "menu/services/controls/add" would locate an action in the "add" group, of the "controls" sub−m "services" menu, in the main menu. Use the id attribute to specify the name of the menu item ber
priority	Required int.

Hierarchy

Parents: <action−ui>, <action−group>.

Children: <action−group>.

Related Topics

None.

<action−ref> Element

<extension−xml>
  <action−ui>
    <action−group>
      <action−ref>

Attributes

Attribute	Description
class	Required string.
priority	

Hierarchy

Parents:

Children:

Related Topics

None.

<popup> Element

Specifies a popup.

<extension−xml>
  <action−ui>
    <popup>

Syntax

```
<popup
    id="nameToUseInContextPaths"
    path="uiContextPath"
>
```

Attributes

Attribute	Description
id	Required string. The name of this item. The full context for this action item is determined by con attribute with the path attribute.
path	Required string. A path through the UI hierarchy, followed by the final group in the last menu. F "menu/services/controls/add" would locate an action in the "add" group, of the "controls" sub−m "services" menu, in the main menu. Use the id attribute to specify the name of the menu item ber

Hierarchy

Parents: <action−ui>.

Children: <action−group>.

Related Topics

None.

<toolbar> Element

Specifies a toolbar or toolbar button.

<extension−xml>
  <action−ui>
    <toolbar>

Syntax

```
<menu
    id="nameToUseInContextPaths"
    label="labelInIDE"
    [path="uiContextPath"]
    priority="priorityNumber"
>
```

Attributes

Attribute	Description
id	Required string. The name of this item. The full context for this action item is determined by con attribute with the path attribute.
label	Required string. The text to display for the menu in the IDE. You can also indicate the action's de key−stroke accelerator by preceding it with '@'. If the label begins and ends with the '%' escape c

	label is interpreted as a Java PropertyBundle path. The label (including accelerator) will be deter the ResourceSvc to load the property named by the path. Place '&' before the character whic menu mnemonic.
path	Optional string. A path through the UI hierarchy, followed by the final group in the last menu. Fo "menu/services/controls/add" would locate an action in the "add" group, of the "controls" sub−m "services" menu, in the main menu. Use the id attribute to specify the name of the menu item ber
priority	Required int.

Hierarchy

Parents: <action−ui>.

Children: <action−group>.

Related Topics

None.

<action> Element

Specifies an IDE action, such as a menu or popup command, or a toolbar button.

<extension−xml>
  <action−set>
    <action>

Syntax

```
<action
    [class="classNameForHandler"]
    [label="menuText"]
    [tooltip="tooltipText"]
    [icon="pathToGifFile"]
    [id=""]
>
```

Attributes

Attribute	Description
class	Optional string. The fully−qualified class name for the class that defines this action. The class must have a parameterless constructor and implement IAction.
label	Optional string. The text to display for this action in menus. Place '&' before the character which is to be the menu mnemonic. You can also indicate the action s default key−stroke accelerator by preceding it with '@'. If the label begins and ends with the '%' escape character, the label is interpreted as a Java PropertyBundle path. The label (including accelerator) will be determined by using the ResourceSvc to load the property named by the path.
tooltip	Optional string. The text to show as a tooltip for a toolbar, in status bar for menus, or in customization dialogs.
icon	Optional string. A path to an image resource to show on toolbars and menus.

id	Optional string.

Hierarchy

Parents: <action−set>.

Children: <location>.

Related Topics

None.

<location> Element

Specifies a location in existing IDE user interface (such as an existing menu group) where an action should be placed. This element is provided as a convenience for simple cases. In general, you should define new user interface for actions with <action−ui> elements.

<extension−xml>
  <action−set>
    <action>
      <location>

Attributes

Attribute	**Description**
path	Required string.
priority	

Hierarchy

Parents: <action>.

Children: None.

Related Topics

None.

Debugger Expression Extension XML Reference

Describes a debugger extension, with which you can add new views for variables.

Note: The <extension−xml> element is the root for any extension descriptor. For this kind of extension, the <extension−xml> element's id attribute value must be "urn:com−bea−ide:debugExpressionViews".

<extension−xml>
  <view>

The following example uses the XmlExpressionView view implementation for a view of a String variable.

```
<extension−xml id="urn:com−bea−ide:debugExpressionViews">
    <view
        priority="80"
        valueType="java.lang.String"
        description="View as XML"
        class= workshop.debugger.ui.expressionview.XmlExpressionView />
</extension−xml>
```

<view> Element

Specifies a new debugger variable view.

Syntax

```
<view
    class="viewImplementation"
    description="viewDescriptionInIDE"
    matchesNulls="true | false"
    priority="rankingAmongViews"
    valueType="typeWhoseDataThisIsAViewFor"
>
```

Attributes

Attribute	Description
class	Required string. The fully−qualified name of the class that provides the custom view. This class must implement IDebugExpressionView.
description	Required string. The description to display for the view when the user right−clicks.
matchesNulls	Required string.
priority	Required int. A number indicating this view's ranking among views for variables of this type.
valueType	Required string. The fully−qualified name of the type to provide this view for.

Hierarchy

Parents: <extension−xml>.

Children: <view>.

Document Extension XML Reference

Describes a document extension, with which you add to the IDE support for new document types. As with JWS, JAVA, JSP, and other types, new types may require specific user interface (such as an icon) and behavior. The extension.xml file for a document extension specifies the class to use for handling this type of document, the document's file extension, and so on.

Note: The <extension−xml> element is the root for any extension descriptor. For this kind of extension, the <extension−xml> element's id attribute value must be "urn:com−bea−ide:document".

```
<extension−xml>
  <document−handler>
    <file−extension>
    <create−template>
      <description>
    <project−attributes>
      <requires>
  <file−extension>
```

See the DocumentSvc class for more information on implementing document extensions.

<document−handler> Element

Specifies a document handler.

```
<extension−xml>
  <document−handler>
```

Syntax

```
<document−handler
    class="handlerClassName"
    icon="pathToIcon"
    label="descriptiveTextForDocumentType"
>
```

Attributes

Attribute	Description
class	Required string. The fully−qualified name of the handler class for this document type. The specified class must implement the IDocumentHandler interface.
icon	Required string. Path to the image that should be used to represent this file type. This will be used, for example, to allow the Application window to display an image next to the file name.
label	Required string. A plain text description of this file type. This will be used as explanatory text where appropriate.

Remarks

The <document−handler> element may have one or more <file−extension> child elements. These are used to associate a document handler with a particular file−name pattern for parsing. Currently, the text inside this tag is expected to come at the end of the file following a ".".

Hierarchy

Parents: <extension−xml>.

Children: <file−extension>, <create−template>, <project−attributes>.

<file−extension> Element

Specifies the extension of a file for which this document handler may be used.

<extension−xml>
  <document−handler>
    <file−extension>

Syntax

```
<file-extension
    priority="prioritySetting"
    [handler="handlerName"]
>
```

Attributes

Attribute	Description
priority	Required enumeration. The ranking of the handler for this file extension. Possible values are described below.
handler	Optional string.

Remarks

This priority attribute captures a relative ranking of how well the handler understands this particular extension. The document service uses this to find a default handler for a given file extension. It will choose the handler with the highest priority. In the event of a tie (this should be avoided), one of the handlers with the highest priority will be arbitrarily chosen. Valid priority values are (from lowest to highest priority):

1. info−only Indicates the handler does not posses any knowledge of the contents of the file, but merely provides an icon and description. Handlers at this level will never be asked to open a file.
2. lowest
3. low
4. medium
5. high
6. highest
7. unknown Indicates that the handler must actually inspect the contents of the file in order to determine what priority it actually has.

You will most often implement "highest" priority handlers, but you may also have "low" and "medium" priority handlers. For example, the extension .java should have a high priority handler that handles JAVA files. However, the extension .jws is also a JAVA file. In the absence of a "highest" priority handler for it (in other words, the web services extension), the JAVA file handler should be able to also handle JWS files. Therefore, the JAVA file handler may declare itself to be a "low" priority handler for files with a .jws extension.

Finally, the "unknown" priority value indicates that the handler must actually inspect the contents of the file in order to determine what priority it actually has. An example here would be a Dialog editor that understands files JAVA files, but only if the class extends the JDialog class. If no "highest" handler is available for a file at run time, "unknown" handlers will be given the opportunity to inspect the file and return one of the six more specific priority values. The highest priority among the "unknown" type handlers and any remaining non−"highest" handlers will be designated as the default handler for a file.

For more information, see IDocumentHandler.Priority.

Hierarchy

Parents: <document−handler>.

Children: None.

<create−template> Element

Specifies information used in the right−click menu and New File dialog when creating a new file of this type.

<extension−xml>
  <document−handler>
    <create−template>

Syntax

```
<create-template
    [id="id"]
    priority="priorityNumber"
    [createCategories="fileCategories"]
    [label="descriptiveText"]
>
```

Attributes

Attribute	Description
priority	Required int. A number indicating this handler's ranking for this file type.
label	Optional string. A plain text description of this file type. This will be used as explanatory text where appropriate.
createCategories	Optional string. Categories in the New File dialog under which this file type will appear. Make this value "ShortCut" to specify that the file type should appear on the right−click menu.
id	Optional string.

Hierarchy

Parents: <document−handler>.

Children: <description>.

<project−attributes> Element

<extension−xml>
  <document−handler>
    <project−attributes>

Syntax

```
<project−attributes>
    <requires name="attributeName" value="true | false"/>
</project−attributes>
```

Hierarchy

Parents: <document−handler>.

Children: <requires>.

<requires> Element

<extension−xml>
  <document−handler>
    <project−attributes>
      <requires>

Syntax

```
<requires name="attributeName" value="true | false"/>
```

Hierarchy

Parents: <project−attributes>.

Children: None.

<description> Element

Specifies the text that should appear for this file type in the New File dialog.

<extension−xml>
  <document−handler>
    <create−template>
      <description>

Syntax

```
<description>Description text that appears for file type in the New File dialog.</project−attri
```

Hierarchy

Parents: <create−template>.

Children: None.

File Encoding Extension XML Reference

Describes a file encoding extension. You use a file encoding extension to specify how a file should be handled when parsing. For example, if you wanted files with an .htm extension to be treated by the IDE in the same way as files with an .html extension, your extension XML might look like the following:

```
<extension−xml>
    <file−encoding appliesTo="htm" treatAs="html"/>
</extension−xml>
```

Note: The <extension−xml> element is the root for any extension descriptor. For this kind of extension, the <extension−xml> element's id attribute value must be "urn:com−bea−ide:encoding".

<file−encoding> Element

Specifies that files with the specified extension should be handled with the specified encoding class, or that they should be handled in the same way as other, specific kinds of files.

<extension−xml>
  <file−encoding>

Syntax

```
<file−encoding
    appliesTo="fileExtension"
    [class="pathToIcon"]
    [treatAs="descriptiveTextForDocumentType"]
>
```

Attributes

Attribute	Description
appliesTo	Required string. The extension or extensions for files this encoding extension applies to.
class	Optional string. The fully−qualified name of a class to use for handling files with this extension.
treatAs	Optional string. The extension of files whose handler should be used to handle files this encoding extension applies to.

Hierarchy

Parents: <extension−xml>.

Children: None.

Frame Extension XML Reference

Describes an extension that adds a dockable frame window to the IDE. The extension XML provides basic information about the frame, including its label in the user interface, its icon in menus, and the class that implements the user interface and behavior for the frame. In addition, you can use the <application−layout> element to specify that frames should be visible in specific positions at startup.

Note: The <extension−xml> element is the root for any extension descriptor. For this kind of extension, the <extension−xml> element's id attribute value must be "urn:com−bea−ide:frame".

For more information on creating a frame view, see the package summary for com.bea.ide.ui.frame.

<extension−xml>
  <frame−view−set>
    <frame−view>
  <application−layout>
    <frame−layout>
      <frame−container>
        <frame−view−ref>

<frame−view−set> Element

Specifies one or more frame views. Each <frame−view> child element specifies a separate frame view.

<extension−xml>
  <frame−view−set>

Syntax

```
<frame−view−set>
    <!−− frame−view elements to specify new frames −−>
</frame−view−set>
```

Attributes

Attribute	Description
scope	Optional string. The scope in which the frames defined in this set will be available for display by the user. This can be "main" or "urn:com−bea−ide:debug" to indicate that the frames should be available in the IDE main (not debugging) mode or debugging mode.

Hierarchy

Parents: <extension−xml>.

Children: <frame−view>.

<frame−view> Element

Specifies an IDE frame view (a dockable window). You use the <frame−view> element to define certain appearance and behavior properties for the view in the IDE. Most importantly, you specify the class that is

your implementation of the view its user interface and behavior.

<extension−xml>
  <frame−view−set>
    <frame−view>

Syntax

```
<frame−view
    askavailable="true | false"
    class="classNameForViewImplementation"
    [hasaction="true | false"]
    [icon="pathToGifFile"]
    [id="identifierForThisView"]
    label="labelToDisplayInUI"
>
```

Attributes

Attribute	Description
askavailable	Optional boolean. true to specify that this view implements IFrameView, and that its isAvailable() method should be called to determine whether the frame can be shown. Note that using this attribute can have significant performance impact. Consider using the <frame−view−set> scope attribute to limit how frequently the view gets asked about its availability.
class	Required string. The fully−qualified class name of the frame view implementation. A frame view implementation class must extend Component, implement IFrameView, or both.
hasaction	Optional boolean. true to specify that a menu item for this view should not be shown in any generated frame view menu.
icon	Required string. Path to a 16x16 .gif file that should be used to represent this view in menus.
id	Optional string. An identifier to distinguish between multiple instances of the implementation class.
label	Required string. The label used in the view tab, and in the View menu.

Hierarchy

Parents: <frame−view−set>.

Children: None.

<application−layout> Element

Specifies the parameters for frame layouts, or descriptors that specify startup positions for frame views. Use this element when you want one or more frame views to appear in the IDE in specific positions at startup.

<extension−xml>
  <application−layout>

Syntax

```
<application−layout
    id="layoutInWhichTheseLayoutsShouldAppear"
>
```

Attributes

Attribute	Description
id	Required string. Identifier for the application layout to which this element's children should be applied. Valid values are "main" and "urn:com−bea−ide:debug".

Hierarchy

Parents: <extension−xml>.

Children: <frame−layout>.

<frame−layout> Element

Specifies the parameters of a layout, or the specific positions of one or more frame views at IDE startup.

<extension−xml>
  <application−layout>
    <frame−layout>

Syntax

```
<frame−layout
    id="frameLayoutID"
>
```

Attributes

Attribute	Description
id	Optional string. Id of the frame layout to which the children should be applied. The valid value is "main".

Hierarchy

Parents: <application−layout>.

Children: <frame−container>.

<frame−container> Element

Specifies layout parameters for a group of frame view windows. Use this element when you have two or more frame views that should appear in specific positions relative to each other at startup. Note that <frame−container> elements can have <frame−container> elements as children, each of which define further−nested frames.

```
<extension−xml>
  <application−layout>
    <frame−layout>
      <frame−container>
```

Syntax

```
<frame−layout
    [orientation="orientationInIDE"]
    proportion="percentageOfIDESpace"
>
```

Attributes

Attribute	Description
orientation	Optional string. This container's orientation in the layout. Valid values are "tabbed", "root", "north", "south", "west", or "east" (see FrameSvc for descriptions of the positions). This attribute is valid only on the root <frame−container> in a <frame−layout> element.
proportion	Required string. The percentage of space that this container should occupy in the IDE. For example, a container oriented "south" with a proportion of "25%" would occupy the lower 25 percent of the IDE. As with HTML tables, proportions need not add up to 100 percent.

Hierarchy

Parents: <frame−layout>.

Children: <frame−view−ref>, <frame−container>.

<frame−view−ref> Element

Specifies a frame view to appear in a container. Within the <frame−container> element, which defines layout parameters, the <frame−view−ref> element specifies details about the frame itself.

```
<extension−xml>
  <application−layout>
    <frame−layout>
      <frame−container>
        <frame−view−ref>
```

Syntax

```
<frame−layout
    class="frameViewImplementationClass"
    [id="frameLayoutID"]
    [proportion="percentageOfContainerSpace"]
    [visible="true | false"]
>
```

Attributes

Attribute	Description
class	Required string. The fully−qualified class name of the frame view implementation.
id	Optional string. An identifier to distinguish between multiple instances of the implementation class.
proportion	Optional string. The percentage of space that this container should occupy in the container. For example, a container oriented "south" with a proportion of "25%" would occupy the lower 25 percent of the IDE. As with HTML tables, proportions need not add up to 100 percent.
visible	Optional boolean. false to hide this frame on startup.

Hierarchy

Parents: <frame−container>.

Children: None.

Help Extension XML Reference

Describes a help extension through which you specify paths that should be added to the locations searched by the IDE when context−sensitive (F1) help is requested. Define this extension when your help must be provided from a location other than that described in Help Authoring Guide.

When the user presses F1 in the IDE, the context−sensitive help engine searches for a topic (an HTML file) that corresponds to the user's current context (such as Source View, with the cursor positioned in a class variable name). The <help−root> element specifies paths to root directories beneath which lie HTML files that may be appropriate topics to display for the user's request.

Note that you should *not* define a help extension and help−root if you are installing your help files in the <workshop_home>/help/doc/<language>/partners folder according to the guidelines described in Help Authoring Guide. See that topic for the best practice recommendations for help provided with extensions. You should use a help extension only in those cases where your help is provided from a location outside the "partners" folder. If you define both a help−root and copy your help files as described in Help Authoring Guide, you topics will appear multiple times in the table of contents and search results.

Note: The <extension−xml> element is the root for any extension descriptor. For this kind of extension, the <extension−xml> element's id attribute value must be "urn:com−bea−ide:help".

<extension−xml>
  <help−root>

<help−root> Element

Specifies the root to help files that should be integrated with the extension.

<extension−xml>
  <help−root>

Syntax

```
<help−root
    dir="pathRelativeToParent"
    parent="parentForHelpPaths"
    url="urlToHelpRoot"
>
```

Attributes

Attribute	Description
dir	Optional string. The directory to search as a help root. This directory is relative to the value specified in the parent attribute.
parent	Optional string. A path to a directory relative to the WebLogic Workshop extensions directory. An absolute path may be used, but it is not recommended. If this attribute is omitted, the directory is assumed to be in the extension's own JAR file or directory.
url	Optional string. An absolute URL that specifies a new help root. This may be any URL, but will typically be HTTP or file based. Set this attribute's value as an alternative to setting the

	dir and parent attribute values. Note that non−file help URLs will not be indexed for search or included in the table of contents.

Hierarchy

Parents: <extension−xml>.

Children: None.

Related Topics

Help Authoring Guide

Preferences Extension XML Reference

Describes a preferences extension, which adds one or more panels to the IDE properties dialogs. These dialogs include IDE Properties, Project Properties, and Application Properties (also known as "workspace properties"). For each of these dialogs, you can specify one or more panel with the <panel> child element.

For more information on building preferences extensions, see Adding Support for Preferences.

Note: The <extension−xml> element is the root for any extension descriptor. For this kind of extension, the <extension−xml> element's id attribute value must be "urn:com−bea−ide:settings".

For more information on specifying a preferences extension, see IPropertyPanel.

<extension−xml>
  <ide−preferences>
    <panel>
  <workspace−preferences>
    <panel>
  <project−preferences>
    <panel>

<ide−preferences> Element

Specifies one or more panels that should be included in the IDE Properties dialog.

<extension−xml>
  <ide−preferences>

Syntax

```
<ide−preferences>
    <!−− panel elements that define user interface for the properties panel −−>
</ide−preferences>
```

Hierarchy

Parents: <extension−xml>.

Children: <panel>.

<workspace−preferences> Element

Specifies one or more panels that should be included in the Application Properties dialog.

<extension−xml>
  <workspace−preferences>

Syntax

```
<workspace−preferences>
    <!−− panel elements that define user interface for the properties panel −−>
</workspace−preferences>
```

Hierarchy

Parents: <extension−xml>.

Children: <panel>.

<project−preferences> Element

Specifies one or more panels that should be included in the Project Properties dialog.

<extension−xml>
  <project−preferences>

Syntax

```
<workspace−preferences>
    <!−− panel elements that define user interface for the properties panel −−>
</workspace−preferences>
```

Hierarchy

Parents: <extension−xml>.

Children: <panel>

<panel> Element

Specifies a panel to appear in a properties dialog.

<extension−xml>
  <ide−preferences>
    <panel>

or

<extension−xml>
  <workspace−preferences>
    <panel>

or

<extension−xml>
  <project−preferences>
    <panel>

Syntax

```
<panel
    class="implementationClassName"
    label="labelForThisPanelInDialog"
    [priority="numberForDisplayRanking"]
>
```

Attributes

Attribute	Description
class	Required string. The fully−qualified name of the panel's implementation class. This class must implement IPropertyPanel.
label	Required string. The name that should appear in the properties dialog, and which the user clicks to select the panel.
priority	Optional int. A number indicating the ranking for this panel in the list of panels.

Hierarchy

Parents: <ide−preferences>, <workspace−preferences>, <project−preferences>.

Children: None.

Project Extension XML Reference

Describes an extension for a new project type. By default, project types installed with WebLogic Workshop include web app, control, Java project, and the EJB project.

The XML for a project type describes visual elements associated with the project, such as folder icons, but also

Because a project incorporates many aspects of WebLogic Workshop functionality, adding support for a new project type can mean writing several different extensions. The following list describes a few of the possibilities.

- Where a new project type includes file types that WebLogic Workshop is not by default designed to handle, the extension.xml can also describe document type extensions; see Document Extension XML Reference for more information.
- You may want to consider whether your project type requires its own preferences panel; see Preferences Extension XML Reference for more information.

Note: The <extension−xml> element is the root for any extension descriptor. For this kind of extension, the <extension−xml> element's id attribute value must be "urn:com−bea−ide:project".

<extension−xml>
  <project−type>
    <attribute>
    <driver>

<project−type> Element

Specifies details, primarily related to the project's appearance in the IDE, about a project type extension.

<extension−xml>
  <project−type>

Syntax

```
<project-type
    id="identifierForProjectType"
    closedfoldericon="pathToGifFile"
    icon="pathToIcon"
    label="descriptiveTextForDocumentType"
    openfoldericon="pathToGifFile"
>
```

Attributes

Attribute	Description
id	Required string. An identifier for this project type. This should be a short name representing this project type −− for example, "php" for a PHP project type.
closedfoldericon	Required string. The path to the .gif file representing the top−level folder of a project of this type when the folder is closed.

icon	Required string. The path to the .gif file to use for representing this project type in the UI.
label	Required string. The text to display for this project type in the New Project dialog.
openfoldericon	Required string. The path to the .gif file representing the top−level folder of a project of this type when the folder is open.

Hierarchy

Parents: <extension−xml>.

Children: <attribute>, <driver>.

<attribute> Element

Specifies an IDE attribute supported by this project type.

<extension−xml>
  <project−type>
    <attribute>

Syntax

```
<attribute
    name="attributeName"
    value="attributeValue"
>
```

Attributes

Attribute	Description
name	Required string. The name of the attribute whose value is being specified. Possible values are given in the Remarks section.
value	Required string. The attribute's value. This is almost always true or false, as described in Remarks.

Hierarchy

Parents: <project−type>.

Children: None.

Remarks

Attributes correspond to predefined behavior in the IDE. In the following example, specifying "true" for the warnOnXSDAdd attribute tells the IDE that a warning message should be displayed if the user tries to add an XSD file to the project (the warning states that the XSD will not be compiled unless put into a schema project).

The following table lists attributes in use by existing project types.

IDE Attribute	Possible Values	Description
j2eeModuleType	web	If you specify workshop.workspace.project.build.WebAppBuildDriver in the class attribute of the <driver> element, the you must include this attribute, and set its value to 'web'
runnableAsWebApp	true to indicate that this	
schemaProject		
supportsJava	true to ; false to .	
supportsControl		
supportsWebApp		
supportsWLWWebApp		
warnOnXSDAdd		

<driver> Element

Specifies drivers to load when a project of this type is loaded.

You can implement drivers that should be running while an instance of your project type is running. For example, you might want to implement a driver that checks for files added to the project's file system. When the user adds a project of this type to an application, the IDE "attaches" each of the specified drivers to it, so that the driver code is running while the user is working in the application. The project itself need not have focus in order for drivers to be active.

Note: A <project−type> element must include a <driver> element that specifies an implementation of IBuildDriver. Without this, the project will not be loaded in the IDE. The <driver> element would look something like this:

```
      <driver type="com.bea.ide.workspace.project.IBuildDriver" class="myPackage.MyBuildDriver
```

<extension−xml>
  <project−type>
    <driver>

Syntax

```
<driver
    class="driverImplementationClass"
    type="driverInterface"
>
```

Attributes

Attribute	Description
class	Required string. The fully−qualified name of the driver implementation. This class must implement the interface specified in the type attribute.
type	Required string. The fully−qualified name of the driver interface.

Hierarchy

Parents: <project−type>.

Children: None.

Remarks

ICompilerDriver

IProjectDriver

IBuildDriver

ILanguageDriver

IRunDriver

IPrintDriver

ISchemaTypeSystemDriver

IDebugDriver

IHelpDriver

IPropertyViewDriver

IDataPaletteViewDriver

IDragDropDriver

ISCMDriver

ISourceViewDriver

INavigationBarDriver

IDebugControlDriver

IWebProjectDriver

ITransferDriver

Extension XML Schema Files

The XSD files in this section are provided as a convenience for those build extensions. You can use the schemas to validate your extension XML files.

Topics Included in This Section

Action Extension Schema (ActionXml.xsd)

Template Schema (ApplicationTemplate.xsd)

Control Annotations Schema (ControlAnnotations.xsd)

Debugger Expression Extension Schema (DebuggerXml.xsd)

Document Extension Schema (DocumentXml.xsd)

Extension Schema (ExtensionDefn.xsd)

Control Stub Schema (ExternalControlStub.xsd)

File Encoding Extension Schema (FileEncodingXml.xsd)

Frame Extension Schema (FrameXml.xsd)

Help Extension Schema (HelpXml.xsd)

Preferences Extension Schema (PreferencesXml.xsd)

Project Extension Schema (ProjectXml.xsd)

Related Topics

Extension XML Reference

template.xml Reference

Developing IDE Extensions

Developing Application and Project Templates

Developing Advanced Controls

IDE Extension Samples

The IdeDevKit sample application includes several samples that demonstrate a few of the ways you can extend the WebLogic Workshop IDE.

Each of the projects in ExtensionDevKit/IdeDevKit is meant to be built as a standalone extension. Each project uses a custom build script to build an extension JAR and then push it and auxiliary files to the appropriate locations in the Workshop directory structure. When you click the Start button in an extension project, the IDE builds the project, then launches a new IDE instance in which you can test the extension. For more information on building and debugging extensions, see Debugging Extensions.

For a list of the samples included, see IDE Extension Samples (IdeDevKit).

Related Topics

None.

CustomProject Sample

CustomProject demonstrates a project type extension. A project type extension adds support for a new kind of project. By default, WebLogic Workshop includes support for project types such as Web Project, Control Project, Java Project, EJB Project, and others. The CustomProject sample extension adds support for a project that includes files whose extension is PHP.

Note: PHP is a server−side web technology similar to JavaServer Pages. WebLogic Server doesn't support PHP, and you don't need PHP support in order to see how the sample works. If you're curious about PHP, see the PHP FAQ at www.php.net.

The CustomProject sample actually demonstrates a variety of extension types. For more information on these, see Adding Support for Preferences, Working with Documents, Application and Project Templates, and Adding New Project Types.

Concepts Demonstrated by this Sample

- extension.xml for a document extension and project extension. For reference information on these, see Document Extension XML Reference, Project Extension XML Reference.
- Making a project "runnable" by implementing IRunDriver.
- Adding support for exporting an Ant build file by implementing IBuildDriver.getAntScript().
- Assigning code to the beginning and end of a project's presence in the open application through a project driver that implements IProjectDriver. In this case, the code is a property change listener that implements java.beans.PropertyChangeListener.
- Adding support for a new kind of document by extending DefaultDocumentHandler, which implements IDocumentHandler. This includes rudimentary source and design view for the document.
- Illustrating how to add project type−specific debugging preferences by implementing IRunPreferences.
- Supporting a project type by including a template with which WebLogic Workshop will generate a default set of files when a new project is created from the type. For more information on templates, see Application and Project Templates.

Location of Sample Files

To view the sample code in WebLogic Workshop:

1. Start WebLogic Workshop.
2. Open the following application installed to your file system:

   BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\IdeDevKit.work

3. In the Application window, expand the folder at ***IdeDevKit*** −> ***CustomProject***.

How to Run the Sample

To run this sample, open one of its files in WebLogic Workshop and click the Start button. The sample will build, copying the resulting extension JAR file to the WORKSHOP_HOME/extensions folder, then launch a new instance of WebLogic Workshop. If you've set breakpoints in the first IDE instance, these should be hit as you exercise the sample code in the new instance.

Note that the CustomProject folder hierarchy includes a php_template folder. When you build the sample (WebLogic Workshop builds it when you run it), the Ant build target zips the contents of this folder into php.zip, which is copied to the WORKSHOP_HOME/templates directory. With the ZIP file in this directory, WebLogic Workshop can find it to populate a PHP project with needed files when a new one is created.

For information on debugging extension samples, see Debugging Extensions.

Related Topics

Adding Support for Preferences

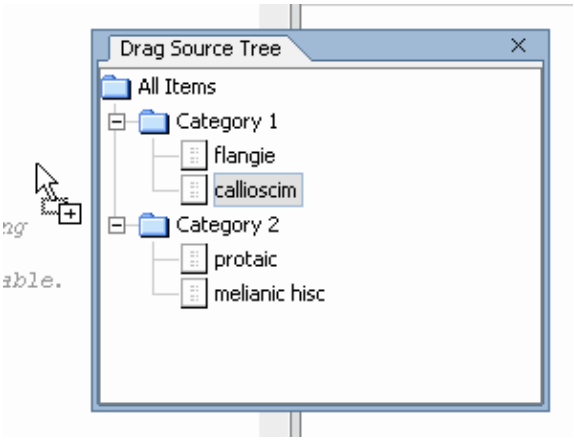
Working with Documents

Application and Project Templates

Adding New Project Types

DragDropSimple Sample

DragDropSimple demonstrates the IDE's support for drag/drop functionality between IDE windows. Through this functionality, you can enable users to copy data from one window to another by dragging items within the IDE. The DragDropSimple sample illustrates this with a tree view of static data; you can drag nodes from the tree to a document open in Source View to copy information from the tree to the source.



For more information on implementing drag−and−drop support in your extension, see Adding Support for Drag and Drop.

Note: Drag/drop support in WebLogic Workshop is in some ways a wrapper for the more general API provided by Java. For more information on drag/drop (also known as DnD) and data transfer in Java, see Drag and Drop with Swing at Sun's Java web site.

Concepts Demonstrated by this Sample

- extension.xml for a frame extension. You'll find reference information for this XML at Frame Extension XML Reference. You'll find more in building frames at Adding Dockable Frames.
- Creating a simple tree view in Java using the JTree and DefaultMutableTreeNode classes.
- Extending DefaultDragDropDriver to create a drag/drop driver that the IDE will use to handle drag operations from a tree view.
- Specifying the data that should be copied from a drag source (here, the tree) to a drop target (a file in Source View).
- Registering a drag/drop driver using the DataTransferSvc class.

Location of Sample Files

To view the sample code in WebLogic Workshop:

1. Start WebLogic Workshop.
2. Open the following application installed to your file system:

   BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\IdeDevKit.work

3. In the ***Application*** window, expand the folder at ***IdeDevKit*** −> ***DragDropSimple***.

How to Run the Sample

To run this sample, open one of its files in WebLogic Workshop and click the Start button. The sample will build, copying the resulting extension JAR file to the WORKSHOP_HOME/extensions folder, then launch a new instance of WebLogic Workshop. If you've set breakpoints in the first IDE instance, these should be hit as you exercise the sample code in the new instance.

To use the sample extension, view the Drag Source Tree frame (if it's not visible, display it by clicking View −> Windows −> Drag Source Tree). Expand the tree in the frame and drag one of the nodes onto a source file in Source View. Text corresponding to the node should be copied into the source file.

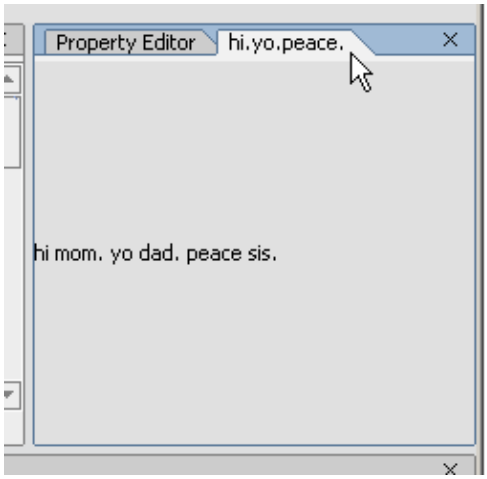
For information on debugging extension samples, see Debugging Extensions.

Related Topics

Adding Support for Drag and Drop

FrameViewSimple Sample

FrameViewSimple demonstrates a frame extension that displays a dockable window with a simple one−line message.



For more information on creating frame extensions, see Adding Dockable Frames.

Concepts Demonstrated by this Sample

- extension.xml for a frame extension. You'll find reference information for this XML at Frame Extension XML Reference.
- Organizing multiple frames for display together at startup.

Location of Sample Files

To view the sample code in WebLogic Workshop:

1. Start WebLogic Workshop.
2. Open the following application installed to your file system:

   BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\IdeDevKit.work

3. In the Application window, expand the folder at ***IdeDevKit*** −> ***MenuItems***.

How to Run the Sample

To run this sample, open one of its files in WebLogic Workshop and click the Start button. The sample will build, copying the resulting extension JAR file to the WORKSHOP_HOME/extensions folder, then launch a new instance of WebLogic Workshop. If you've set breakpoints in the first IDE instance, these should be hit as you exercise the sample code in the new instance.

With the second IDE instance running, you should have access to the frame pictured at the top of this topic. If it isn't visible at first, you can display it by clicking View −> Windows −> hi.yo.peace.

Note: You may not see the "multiple frames" aspect of this sample unless you start WebLogic Workshop with your user preferences reverted to default settings. To do this, quit the IDE, rename or delete the file at USER_HOME/.workshop.zpref, then start the IDE again.

For information on debugging extension samples, see Debugging Extensions.

Related Topics

Adding Dockable Frames

MenuItems Sample

MenuItems demonstrates a simple menu action extension. An action is a menu, popup menu, or toolbar command in the WebLogic Workshop IDE. The MenuItems extension adds a Favorites menu whose submenus allow the user to add and remove URLs as favorites; clicking a URL in the menu opens a browser to that URL's location.

For more information on creating action extensions, see Adding Menus and Toolbar Buttons.

Concepts Demonstrated by this Sample

- extension.xml for an action extension. You'll find reference for this XML at Action Extension XML Reference.
- Executing code when a menu command is clicked.
- Generating actions (menu commands) dynamically with the ActionSvc class, the IActionProxy interface, and an implementation of the IGenerator interface.
- Specifying menu mnemonics.
- Accessing preferences stored for the user through the PreferencesSvc class.
- Logging debugging messages with the MessageSvc class.
- Simple user interface with Java Swing components. For Swing−related links, see Getting Started with UI Programming.

Location of Sample Files

To view the sample code in WebLogic Workshop:

1. Start WebLogic Workshop.
2. Open the following application installed to your file system:

   BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\IdeDevKit.work

3. In the Application window, expand the folder at ***IdeDevKit*** −> ***MenuItems***.

How to Run the Sample

To run this sample, open one of its files in WebLogic Workshop and click the Start button. The sample will build, copying the resulting extension JAR file to the WORKSHOP_HOME/extensions folder, then launch a new instance of WebLogic Workshop. If you've set breakpoints in the first IDE instance, these should be hit as you exercise the sample code in the new instance.

With the second instance of the IDE running, you should have access to the Favorites menu pictured at the top of this topic. To try out the sample, click the menu and select items from it. You can also add and remove

items using the commands provided.

For information on debugging extension samples, see Debugging Extensions.

Related Topics

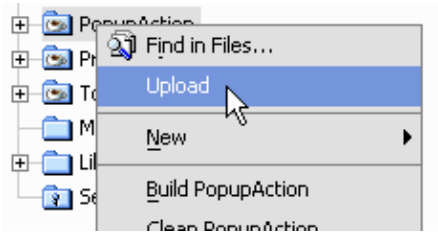
Adding Menus and Toolbar Buttons

PopupAction Sample

ToolbarButton Sample

PopupAction Sample

PopupAction demonstrates a simple popup action extension. An action is a menu, popup menu, or toolbar command in the WebLogic Workshop IDE. The PopupAction extension adds an Upload popup menu through which you can use FTP to upload a file or project to a site specified in project preferences.



For more information on creating action extensions, see Adding Menus and Toolbar Buttons.

Concepts Demonstrated by this Sample

- extension.xml for an action extension. You'll find reference for this XML at Action Extension XML Reference.
- Executing code when a popup menu command is clicked.
- Generating actions (menu commands) dynamically with the ActionSvc class and the IActionProxy interface.
- Storing and retrieving project−specific properties using the IPreferencesSupport interface through an IProject instance. Displaying these preferences in a properties dialog that implements the IProjectPropertyPanel interface.
- Storing user interface strings in a separate file for localizing, and accessing the strings using the ResourceSvc class.
- Getting application context, such as a list of projects in the currently open application, through the IWorkspace interface and Application class.
- Getting information about items selected in the application through the IWorkspaceEventContext interface.
- Logging debugging messages with the MessageSvc class.
- Using the OutputSvc class to print messages to a dockable window.
- Adding support for an asynchronous IDE task by implementing the IAsyncTask interface and using the AsyncTaskSvc class.
- Simple user interface with Java Swing components. For Swing−related links, see Getting Started with UI Programming.

Location of Sample Files

To view the sample code in WebLogic Workshop:

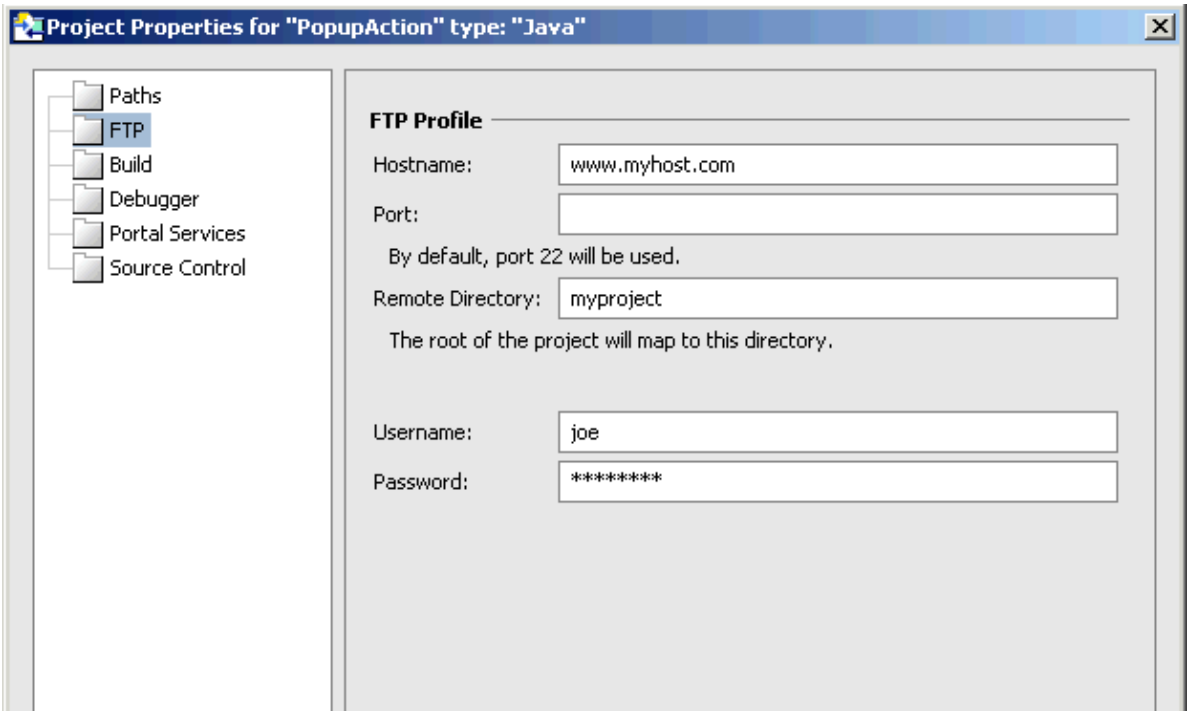
1. Start WebLogic Workshop.
2. Open the following application installed to your file system:

   BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\IdeDevKit.work
3. In the Application window, expand the folder at IdeDevKit −> PopupAction.

How to Run the Sample

To run this sample, open one of its files in WebLogic Workshop and click the Start button. The sample will build, copying the resulting extension JAR file to the WORKSHOP_HOME/extensions folder, then launch a new instance of WebLogic Workshop. If you've set breakpoints in the first IDE instance, these should be hit as you exercise the sample code in the new instance.

With the second instance of the IDE running, you should be able to right−click a folder in the Application window to have access to the popup menu pictured at the top of this topic. This extension also provides a properties panel you can try out.



To reach these properties, right−click a project in the Application window, then click Properties. In the Properties dialog, click FTP. Keep in mind that in order to actually try out the FTP functionality, you'll need to specify FTP properties for the project whose contents you want to upload.

For information on debugging extension samples, see DebuggingExtensions.

Related Topics

Adding Menus and Toolbar Buttons

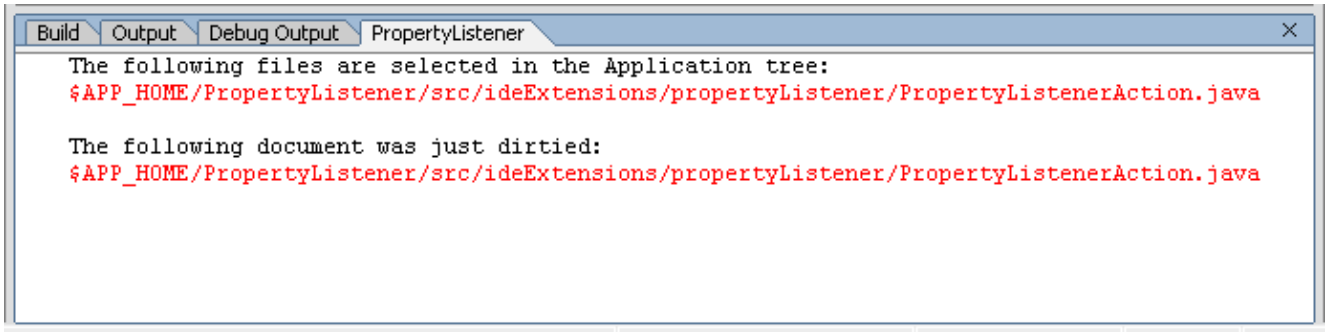
MenuItems Sample

ToolbarButton Sample

PropertyListener Sample

PropertyListener demonstrates an extension can keep informed of changes in the IDE's state, including the files it contains, what has focus, and so on. registers to listen for IDE events. These events include com.bea.ide.Application.EVENT_DocumentDirtyChange and com.bea.ide.Application.PROP_FocusedURIs.

EVENT_DocumentDirtyChange is delivered when a document is changed (made "dirty") or saved. The PROP_FocusedURIs event is sent any time the set of files highlighted in the Application tree changes. When either of these events is fired, the handlers in the sample's PropertyListenerAction class propagate diagnostic info to an instance of IOutputWindow.



Concepts Demonstrated by this Sample

- extension.xml for an action extension. You'll find reference for this XML at Action Extension XML Reference. For information about building action extensions, see Adding Menus and Toolbar Buttons.
- Listening for changes to IDE properties by registering a PropertyChangeListener as a listener. The registration is done through Application.I.addPropertyChangeListener.
- Listening for changes to the Application.EVENT_DocumentDirtyChange and Application.PROP_FocusedURIs properties.
- Displaying a window for messages in the IDE using OutputSvc.IOutputWindow.
- Getting a URI for the application home directory through the Application class, IWorkspace interface, and IFile.getAbsoluteURI.

Location of Sample Files

To view the sample code in WebLogic Workshop:

1. Start WebLogic Workshop.
2. Open the following application installed to your file system:

   BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\IdeDevKit.work

3. In the Application window, expand the folder at ***IdeDevKit*** −> ***PropertyListener***.

How to Run the Sample

To run this sample, open one of its files in WebLogic Workshop and click the Start button. The sample will build, copying the resulting extension JAR file to the WORKSHOP_HOME/extensions folder, then launch a

new instance of WebLogic Workshop. If you've set breakpoints in the first IDE instance, these should be hit as you exercise the sample code in the new instance.

With the second IDE instance running, you should have access to Properties menu on the main menu bar. Click Properties −> Property Listener to display a message window like the one at the top of this topic. Click items in the Application window, or edit an open document to see messages generated by the event listeners.

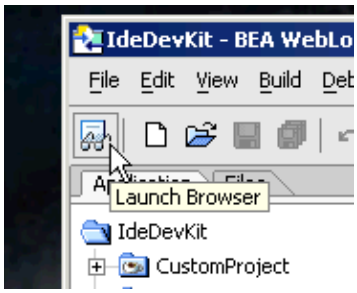
For information on debugging extension samples, see Debugging Extensions.

Related Topics

Adding Menus and Toolbar Buttons

ToolbarButton Sample

ToolbarButton demonstrates a simple toolbar button action extension. An action is a menu, popup menu, or toolbar command in the WebLogic Workshop IDE. The ToolbarButton extension adds a toolbar button that launches a browser to http://dev2dev.bea.com.



For more information on creating action extensions, see Adding Menus and Toolbar Buttons.

Concepts Demonstrated by this Sample

- extension.xml for an action extension. You'll find reference for this XML at Action Extension XML Reference.
- Executing code when a toolbar button is clicked.
- Logging debugging messages with the MessageSvc class.

Location of Sample Files

To view the sample code in WebLogic Workshop:

Start WebLogic Workshop.

Open the following application installed to your file system:

BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\IdeDevKit.work

In the Application window, expand the folder at IdeDevKit −> ToolbarButton.

How to Run the Sample

To run this sample, open one of its files in WebLogic Workshop and click the Start button. The sample will build, copying the resulting extension JAR file to the WORKSHOP_HOME/extensions folder, then launch a new instance of WebLogic Workshop. If you've set breakpoints in the first IDE instance, these should be hit as you exercise the sample code in the new instance.

With the second instance of the IDE running, you should see the toolbar button pictured at the top of this topic. To try out the extension, click the button. It should launch a web browser to display the dev2dev web site.

For information on debugging extension samples, see Debugging Extensions.

Related Topics

Adding Menus and Toolbar Buttons

MenuItems Sample

PopupAction Sample

Developing Tag Library Extensions

A tag library extension integrates a custom JSP tag library with the WebLogic Workshop development environment. JSP developers use tags in your library for declarative access to Java−based logic; your corresponding library extension connects the tags in the library to IDE support that makes using them easier. This support includes:

- Visibility on the Palette and Data Palette.
- Error checking in Source View.
- Custom tag rendering in the JSP designer.
- A custom property builder accessible through the Property Editor.

TLDX File and Extension Classes

A tag library includes a Tag Library Descriptor (TLD) file to describe the library's tags and their handlers. When you extend the tag library to include IDE support, you add a Tag Library Descriptor Extension (TLDX) file to describe the IDE functionality supported and to point to the Java classes that contain the extension logic. You implement the TLD and tag handlers just as you ordinarily would; the TLDX and supporting classes simply provide IDE support.

To compare a TLD and TLDX file, see the examples in the ExtensionDevKit set of sample applications. Open the TaglibExtDevKit sample application. There, under TaglibWebProject/WEB−INF, compare the TLD and TLDX files, including tdk−tags.tldx and tdk−tags.tld, which accompany sample extensions. You'll see that the TLDX format mirrors and extends the corresponding TLD file with extra properties. For more information about the specific properties available in a TLDX file, see TLDX File Contents.

See Tag Library Extension Samples for more information on building and running the sample tag library extension included in this kit.

Integrating Custom Tag Library Documentation

You can integrate documentation for your custom tags so that those topics are visible in the WebLogick Workshop help browser. The file at TaglibExtDevKit/TldxHandlers/resources/META−INF/extension.xml specifies a directory that the IDE should look for help files. As noted in the sample build instructions in Tag Library Extension Samples, the help index must be recreated for the extension help to be recognized. Once you have done this, select Help Topics from the Help menu. The TDK help files will appear under the "Custom Controls" topic

Note: In the GA release of WebLogic Platform 8.1, custom tag libraries will not have their own topic heading in the table of contents.

Related Topics

None.

TLDX File Contents

You create a TLDX file to specify characteristics of a tag library extension. For example, a TLDX fie specifies what should happen when the contents of a tag are deleted, it names the class that provides logic for rendering the tag at design time, and so on.

The structure of a TLDX file is similar to that of a TLD file. They have a few of the same elements, and both list the same tags and attributes. However, for each tag and attribute, a TLDX file describes IDE−specific functionality.

Click the following elements to view a description of each.

<taglib>
  <uri>
  <palettegenerator>
  <link>
    <prefix>
    <uri>
  <tag>
    <name>
    <deletewhenempty>
    <requiredparent>
    <requiredchild>
    <illegalancestor>
    <bodycontent−pref>
    <output>
    <renderer>
    <whitespace>
    <data−palette−driver>
    <attribute>
      <name>
      <category>
      <propertyclass>
      <deprecated>
      <validationrule>
      <validationmessage>
      <extype>
      <reftype>

Here is an overview of the syntax for a TLDX file:

```
<taglib>
    <uri>uriAssociatedWithATaglib</uri>
    <palettegenerator>package.PaletteGeneratorClassName</palettegenerator>
    <link>
        <prefix>prefixUsedWhenLinkingATaglib</prefix>
        <uri>uriAssociatedWithLinkedTagLib</uri>
    </link>
    <tag>
        <name>nameOfextendedTag</name>
        <deletewhenempty>true|false</deletewhenempty>
        <requiredparent>Space separated list of tag names</requiredparent>
```

```
        <requiredchild>Space separated list of tag names</requiredchild>
        <illegalancestor>Space separated list of tag names</illegalancestor>
        <bodycontent-pref>empty|JSP</bodycontent-pref>
        <output><![CDATA[<b>Text that should be displayed in Design View.</b>]]></output>
        <renderer>package.RendererClassName</renderer>
        <whitespace>preserve|indent|inline|block</whitespace>
        <data-palette-driver>package.DataPaletteDriverClass</data-palette-driver>
        <attribute>
            <name>nameOfExtendedAttribute</name>
            <category>nameOfPropertyEditorGroup</category>
            <propertyclass>package.PropertyBuilderClass</propertyclass>
            <deprecated>true</deprecated>
            <validationrule>regularExpressionOrExpressions</validationrule>
            <validationmessage>Message to user for invalid value</validationmessage>
            <extype>java-name|java-id</extype>
            <reftype>reftype</reftype>
        </attribute>
    </tag>
</taglib>
```

<taglib> Element

Required. The root of this extension descriptor. The <taglib> element describes extensions to the tags in a custom tag library. Note that the <uri> child element must match the tag library you are extending. <tag> child elements correspond to tags defined in the library.

<taglib>
  <uri>
  <palettegenerator>
  <link>
  <tag>

Syntax

```
<taglib>
    Child elements for defining specifics of the tag library extension.
</taglib>
```

<uri> Element

Required. The URI from the taglib with which this TLDX is associated. This is the value of the corresponding TLD file's <uri> element.

<taglib>
  <uri>

Syntax

```
<uri>uriAssociatedWithATaglib</uri>
```

<palettegenerator> Element

Specifies a class that extends com.bea.ide.jspdesigner.PaletteGenerator, and which defines how a library's tags appear in the palette.
The tag group will also appear in the main Insert drop−down menu. manifestation in TDK: open TaglibExtDevKit/TaglibWebProject/demo.jsp in design view, and notice the "TDK tags" group in the palette. you will find the code that generates this group at TaglibExtDevKit/TldxHandlers/TDK/PaletteGenerator.java

<taglib>
  <palettegenerator>

Syntax

```
<palettegenerator>package.PaletteGeneratorClassName</palettegenerator>
```

<link> Element

Specifies a reference to another tag library that may be used in the <requiredparent> or <requiredchild> elements. May occur multiple times in a TLDX.

<taglib>
  <link>
    <prefix>
    <uri>

Remarks

You can use this element, along with <requiredparent> or <requiredchild>, to enforce that a tag in the current library requires a tag in another library as its parent or child. For an example, see the netui−tags−html.tldx file installed with WebLogic Workshop. In that file, the following <link> element specifies the prefix and URI for the other tag library:

```
<link>
    <prefix>data</prefix>
    <uri>http://www.bea.com/workshop/netui−tags−databinding−1.0</uri>
</link>
```

With this prefix specified in the <link> element, it is used in the <requiredparent> element. The following TLDX code lists several tags which must be parents, including three from the databinding library specified above.

```
 <requiredparent>label select textArea textBox data:anchorColumn data:basicColumn data:expressi
```

<prefix> Element

Prefix that will be used in the list of tag names for <requiredparent> or <requiredchild> elements.

<taglib>
  <link>
    <prefix>

Syntax

```
<prefix>prefixUsedWhenLinkingATaglib</prefix>
```

<uri> Element

Specifies the URI from the other tag library.

<taglib>
  <link>
    <uri>

Syntax

```
<uri>uriAssociatedWithLinkedTagLib</uri>
```

<tag> Element

Describes extensions for a tag defined in the corresponding TLD file. There must be a <tag> element in the TLDX for each <tag> element in the TLD.

<taglib>
  <tag>
    <name>
    <deletewhenempty>
    <requiredparent>
    <requiredchild>
    <illegalancestor>
    <bodycontent−pref>
    <output>
    <renderer>
    <whitespace>
    <data−palette−driver>
    <attribute>

Syntax

```
<tag>
    Child elements describing extensions for a specific tag.
</tag>
```

<name> Element

Required. The name of the tag being extended. This value must match the name of the tag defined in the corresponding TLD.

<taglib>
  <tag>
    <name>

Syntax

```
<name>nameOfextendedTag</name>
```

<deletewhenempty> Element

Optional. true if the extended tag should be deleted from the JSP page at design time when one of the following occurs:

- The tag's last child is deleted
- All of the text inside the tag has been removed.

```
<taglib>
  <tag>
    <deletewhenempty>
```

Syntax

```
<deletewhenempty>true | false</deletewhenempty>
```

<requiredchild> Element

Optional. A space−separated list of tags which can be a child of this tag; only tags listed here may be children of the tag specified by the <name> element. In addition, there are two special values for this element:

- #nothing Use this value only to indicate that this tag can not have children.
- #text Add this value to the list of tags to indicate that this tag is allowed to contain non−whitespace text. This causes the "innerText" property to appear in the Property Editor.

```
<taglib>
  <tag>
    <requiredchild>
```

Syntax

```
<requiredchild>Space separated list of tag names</requiredchild>
```

<requiredparent> Element

Optional. A space−separated list of names of tags which can be a parent of this tag. This element allows you to enforce a list of tags that are valid tags for a given tag. An invalid parent tag will give a warning in the source editor, such as "WARNING: Invalid parent tag."
If these tags are from this tag library, then the names need not have a prefix. If they are from another tag library, then the name must include a prefix specified by one of the <link> elements in this TLDX file. If this element is not present, then the tag may have any tag as its parent.

```
<taglib>
  <tag>
    <requiredparent>
```

Syntax

```
<requiredparent>Space separated list of tag names</requiredparent>
```

<illegalancestor> Element

Optional. A space−separated list of tags that are not allowed to be an ancestor of this tag. For example, <form> tags cannot be nested within each other.
This affects drag/drop behavior, but does not result in warnings in the source code.

<taglib>
  <tag>
    <illegalancestor>

Syntax

```
<illegalancestor>Space separated list of tag names</illegalancestor>
```

<bodycontent−pref> Element

Optional. Indicates whether this tag should be closed with /> or </tagname> when the tag is inserted in Source view. Possible values:

- empty Use an end tag, such as </tagname>, to close this tag.
- JSP Use /> to close this tag.

<taglib>
  <tag>
    <bodycontent−pref>

Syntax

```
<bodycontent−pref>empty | JSP</bodycontent−pref>
```

<output> Element

Optional. Specifies the HTML that represents this tag in the designer. Enclose the HTML value in a CDATA tag to ensure that this TLDX file remains a valid XML document.

<taglib>
  <tag>
    <output>

Syntax

```
<output><![CDATA[<b>Text that should be displayed in Design View.</b>]]></output>
```

<renderer> Element

Specifies the class that should be used to provide a customized display for this tag in Design View. This class must extend com.bea.ide.jspdesigner.Renderer.

To see an example, open the TaglibExtDevKit sample application. There, under TaglibWebProject, open demo.jsp in design view. Notice the custom view of the <tdk:barcode> tag. Also, if you select the tag and modify the value attribute in the Property Editor, you'll see that Design View re−renders the tag. The code that renders this tag is contained in the same application, in TldxHandlers/TDK/Renderer.java.

<taglib>
  <tag>
    <renderer>

Syntax

```
<renderer>package.RendererClassName</renderer>
```

<whitespace> Element

Optional. Specifies how whitespace should be treated when reformatting the source code of the document. Possible values are:

- preserve Preserve white space.
- indent
- inline (default)
- block

<taglib>
  <tag>
    <whitespace>

Syntax

```
<whitespace>preserve | indent | inline | block</whitespace>
```

<data−palette−driver> Element

Optional. Specifies the class that should be used to provide custom population of the Data Palette on a tag−by−tag basis. This class must extend com.bea.ide.jspdesigner.DataPaletteTagDriver.

To see an example, open the TaglibExtDevKit sample application. There, in the TaglibWebProject project, open a JSP page and insert a NetUI Form. When the Form Wizard appears, select Create New, then click Create. The data−palette−driver places the new action under the Actions category in the Data Palette.
<taglib>
  <tag>
    <data−palette−driver>

Syntax

```
<data-palette-driver>package.DataPaletteDriverClass</data-palette-driver>
```

<attribute> Element

Describes extensions for a tag attribute defined in the corresponding TLD file. There must be an <attribute> element in the TLDX for each <attribute> element in the TLD.

```
<taglib>
  <tag>
    <attribute>
      <name>
      <category>
      <propertyclass>
      <deprecated>
      <validationrule>
      <validationmessage>
      <extype>
      <reftype>
```

Syntax

```
<attribute>
    Child elements describing extensions for a specific attribute.
</attribute>
```

<name> Element

Required. The name of the attribute being extended. This value must match the name of the attribute as defined in the corresponding TLD.

```
<taglib>
  <tag>
    <attribute>
      <name>
```

Syntax

```
<name>nameOfExtendedAttribute</name>
```

<category> Element

The name of the group that this attribute should appear under in the Property Editor.

```
<taglib>
  <tag>
    <attribute>
      <category>
```

Syntax

```
<category>nameOfPropertyEditorGroup</category>
```

<propertyclass> Element

Specifies the class that defines an attribute's custom builder available from the Property Editor; this class must extend com.bea.ide.jspdesigner.PropertyClass.

To see an example, open the TaglibExtDevKit sample application. There, in the TaglibWebProject project, open a JSP page and insert a NetUI Anchor. In the Property Editor, under General, next to the href attribute, click the ... button. The Open dialog that appears is the builder for this attribute.

```
<taglib>
  <tag>
    <attribute>
      <propertyclass>
```

Syntax

```
<propertyclass>package.PropertyBuilderClass</propertyclass>
```

<deprecated> Element

true if this attribute has been deprecated.

```
<taglib>
  <tag>
    <attribute>
      <deprecated>
```

Syntax

```
<propertyclass>true | false</propertyclass>
```

<validationrule> Element

The regular expression to use to validate this attribute's value. If the <validationrule> element's value is of the form expression1|expression2|expression3, then the IDE will display a dropdown list in Source View and the property sheet.

```
<taglib>
  <tag>
    <attribute>
      <validationrule>
```

Syntax

```
<validationrule>regularExpressionOrExpressions</validationrule>
```

<validationmessage> Element

Specifies the message to be displayed when this attribute's value is invalid.

<taglib>
  <tag>
    <attribute>
      <validationmessage>

Syntax

```
<validationmessage>Message to user for invalid value</validationmessage>
```

<extype> Element

Specifies the extended type information for validation. Currently supports:

- java−name Represents the type of valid Java identifiers.
- java−id Represents the type of (optionally) qualified Java names.

<taglib>
  <tag>
    <attribute>
      <extype>

Syntax

```
<extype>java−name | java−id</extype>
```

<reftype> Element

Specifies the name passed to com.bea.language.jsp.IFileReferences.getTransform() for use in transforming this attribute into a file reference.

<taglib>
  <tag>
    <attribute>
      <reftype>

Syntax

```
<reftype>reftype</reftype>
```

Related Topics

None.

Tag Library Extension Samples

The TaglibExtDevKit sample application includes an extended custom JSP tag, along with a web application in which to test it. Also included in the application is a project that illustrates how to make your custom tag library available to WebLogic Workshop web applications.

The following projects are included in the TaglibExtDevKit application:

- TagHandlers Handler code for the tag library of which the <tdk:barcode> custom JSP tag is a part.
- TldxHandlers Code that provides logic for extensions to the tag library.
- TaglibWebProject A web project to use for trying out the sample extensions.
- TaglibInstall A project that illustrates how to make a tag library available to web applications.

The first three of these projects are described in more detail under <tdk:barcode>. The last is described under TaglibInstall Sample.

<tdk:barcode> Sample

The <tdk:barcode> tag sample provided with this release illustrates the basics of extending a custom JSP tag library. As described in Developing Tag Library Extensions, you extend a tag library in order to integrate the tag with the WebLogic Workshop IDE.

Note that if you haven't already, you will need to build the sample in order to see it working in the IDE. For more information, see Building and Running the Sample.

The code for this sample (and other extended JSP tags) can be divided into two categories:

- ***Standard JSP tag code***. This includes the tag handler in the TaglibExtDevKit sample application, at TagHandlers/TDK/Barcode.java. It also includes the tag library's TLD file, located in the TaglibWebProject application, at WEB−INF/tdk−tags.tld. Keep in mind that this handler conforms to the standard for JSP tags (for example, it extends the BodyTagSupport class).
- ***Tag library extension code***. This includes handler classes that provide user interface integration with WebLogic Workshop. It also includes the extension's TLDX file, located in the TaglibWebProject application, at WEB−INF/tdk−tags.tldx. This file describes the extension's characteristics, pointing to the extension handler classes where needed. For more information on TLDX files, see TLDX File Contents.

Features Illustrated by this Sample

The <tdk:barcode> sample illustrates a few basic areas where tag library extensions provide tag integration with WebLogic Workshop. In Design View, the tag displays a barcode representing whatever value you type into the tag's value property attribute.

- Using a renderer class to customize tag display in Design View. The Renderer class displays a barcode symbol (created by the BarcodeEncodings class) based on the value attribute.
- Using a palette generator class to customize how the tag is listed in the Palette. The PaletteGenerator class defines where and how the custom tag will appear in the WebLogic Workshop Palette.
- Using several TLDX elements to define other characteristics. These include how the <tdk:barcode> tag should be formatted in source code.

Support for Help

To illustrate help integration, the <tdk:barcode> sample provides a few placeholder topics that are displayed when the user presses F1. This is typical for custom tags and controls used in WebLogic Workshop.

Note: As installed, these topics don't open in the frameset used by other topics. To enable this, you will need to include several templates that all workshop help files include. For the GA release of WebLogic Platform 8.1, we do not provide these templates, but you are encouraged to look at the source of any of our help files if you would like to reproduce this functionality.

Building and Running the Sample

This section describes how to build components of the TaglibExtDevKit sample application so that you can run the TaglibWebProject sample and see the result. You needn't perform these steps in order to merely browse the sample code.

1. In WebLogic Workshop, open the TaglibExtDevKit application.
   In the Extension Development Kit, the WORK file is located at ExtensionDevKit/TaglibExtDevKit/TaglibExtDevKit.work.
2. Build the TagHandlers Java project: In the Application pane, right−click the ***TagHandlers*** folder, then select ***Build TagHandlers***.
3. Build the TldxHandlers Java project: In the Application pane, right−click the ***TldxHandlers*** folder, then select ***Build TldxHandlers***.

   Building the project will compile and package the sources. It will also copy resulting JAR to the workshop/extensions folder where the tab library extensions can be found the IDE, and it will copy the documentation in the project to the correct location in the help hierarchy (workshop/help/doc/en/tldx).
4. Restart the IDE to pick up changes.
5. Finish integrating the tag library documentation by rebuilding the search index. From the ***Help*** menu, select ***Rebuild Search Index***. This may take a few minutes.
   Note that after you rebuild the index, you will only need to rebuild it again if you have either added new help files or modified the toc.xml file.
6. Update the test client web application to include the newly built tag library extensions. In the Application pane, right−click the ***TaglibWebProject*** folder, then select ***Install*** −> ***Web Project Libraries***. In the Web Project Libraries dialog, select the All Web Project Files check box, then click ***Install***.
   The Web Project libraries need to be in sync with the version of WebLogic Workshop that is currently running.
7. Try out the extensions that support Design View. Open TaglibExtDevKit/TaglibWebProject/demo.jsp. With demo.jsp in Design View, click the tag in the design. In the ***Property Editor***, edit the value attribute. Notice that as you change the value, the barcode updates to represent the new value. Also, note that there is a ***.:TDK tags:.*** category in the ***Palette***, along with a ***.:barcode:.*** tag so that you can add new instances of the tag to the page design.
8. Run demo.jsp to test the tag handler. Right−click the JSP design, then select ***Run JSP***. If you are prompted to start WebLogic Server, click ***OK***.
   demo.jsp should open in a Test Browser. If the test is successful, the browser will display the following text:

   SUCCESS: value to encode=*value_of_value_attribute*

TaglibInstall Sample

This sample demonstrates how you can write an extension that enables your tag libraries for use in web projects.

To use this sample, use the following steps:

1. Right−click on the TaglibInstall project and select Build TaglibInstall.
   The build target copies an extension jar into workshop's extension folder and a template archive into workshop's template folder.
2. Restart the IDE and create a new web project in your application.
3. Right−click on the new webapp project in the application pane, then click ***Install*** −> ***TDK Taglibs***.
   This copies the files from the template archive into the webapp directory structure.

   Additionally, a new entry for the taglib is created in the web project's web.xml. Together, these steps make the TDK taglibs available to the user for this web project.

The logic for this process is captured in TaglibInstall/TDK/TemplateProcessor.java. The processor class must implement com.bea.ide.workspace.project.IProjectTemplateProcessor and be referenced in the template archive's template.xml file (for an example, see TaglibInstall/template/template.xml).

Related Topics

None.

Help Authoring Guide

The WebLogic Workshop help system contains a collection of HTML pages that are organized in a table of contents. In addition, a user can search the help pages and receive context−sensitive help by selecting an element in Source View or Design View and pressing F1.

When you are creating help for WebLogic Workshop extensions and controls that are intended for redistribution, you can take full advantage of the features of the WebLogic Workshop help system. That is, you can create help pages that are visible in the table of contents, are searchable, and can be accessed through context−sensitive help. This topic describes what you need to know to author help topics and add these to the WebLogic Workshop help system.

This topic contains the following sections:

- Help Integration Support in the IDE. Describes the difference in support for help integration between Java controls and other extensions.
- Organizing Your Help System. Describes how the various help files you are creating should be organized.
- Creating Help Pages. Discusses how to create help content that will look and feel like the help provided by WebLogic Workshop.
- Making Search Results Useful. Discusses how to make your help content searchable.
- Creating a TOC. Describes how to create a table of contents for the help topics that can be merged into WebLogic Workshop help.

Help Integration Support in the IDE

The guidelines provided below describe how to create and structure your help files so that they integrate smoothly with installed WebLogic Workshop documentation. Note that how these files are handled when the extension is installed differs between controls and other kinds of extensions.

For extension help that follows the guidelines in this topic, WebLogic Workshop provides support for the following:

- Visibility of topics to full−text search.
- Visibility of topics in the table of contents.
- Visibility of class, annotation, and JSP tag reference topics when context−sensitive help is requested (such as when the user presses F1 in Source View or Design View).

Help Integration for Java Controls

In version 8.1 SP2, WebLogic Workshop supports automatic integration for documentation provided with Java controls. This means that if you follow the guidelines in this topic and include your help in a control deliverable ZIP file, your help files will be automatically integrated. See Packaging Controls for Installation for more information on creating a control deliverable.

When the user installs a control deliverable for the first time, the IDE does the following to integrate the help provided with the control:

- Explodes the top−level help folder and copies its contents into <workshop_home>/help.

- Merges the provided table of contents file (toc.xml) with the installed table of contents. The result is that the control help is available beneath the Extensions −> <vendor_name> node in the table of contents.
- Rebuilds the search index so that the new topics are available for full−text search.

Help Integration for Other Kinds of Extensions

For other kinds of extensions, you must copy your documentation to the correct location for example, you can do this with an installer program. In addition, after installation, you must rebuild the table of contents and search index to integrate your content with the Workshop documentation.

For seamless integration that does not require the user's help, your installation process should do the following:

- Copy your top−level help folder and its contents to <workshop_home>/help. Note that your help hierarchy should not include a copy of workshop.css in any <language> folders. (See Organizing Your Help System for more information.)
- Rebuild the search index and table of contents by invoking the workshop.core.IndexIdeHelp class. You can do this with the following command:
  ```
  java −Djava.system.class.loader="workshop.core.AppClassLoader" −cp wlw−ide.jar;wlw−rebui
  ```

  You can find the wlw−ide.jar and wlw−rebuild−index.jar files on the user's machine in the $WL_HOME/workshop directory.

If your installation process is unable to invoke IndexIdeHelp class, you should prompt the user to index help by clicking Help −> Rebuild Search Index. Your help will not appear in the table of contents or be visible to full−text search until one of these procedures has occurred.

Note: For material you can use to build and test your help installation, see the Help Test Kit at the WebLogic Workshop Extensibility Portal page at dev2dev.

Organizing Your Help System

Your help topics should be organized as described in the following list. Note the presence of <language> and <vendor_name> placeholders for folder names. The <language> placeholder represents the language for the content. This would be "en" for English, "ja" for Japanese, and so on. The <vendor_name> placeholder represents your company name, such as "Acme".

Note: You should choose the name of the <vendor−name> directory carefully. A user may have a large number of extensions installed on his or her system. The name of your vendor directory should clearly and uniquely identify your company and product name.

- ***Help root*** The root directory for your help files should be help/doc/<language>/partners/<vendor_name>.
- ***Table of Contents file*** The toc.xml file, which is used to add your help pages to the table of contents, must be located at the root of your help structure, at help/doc/<language>/partners/<vendor_name>/toc.xml
- ***CSS file*** The workshop.css file, the CSS style sheet for help topics, should be located in the <language> folder. Using the style rules defined in this file helps to ensure that your content has a look and feel consistent with the rest of the Workshop documentation.

Note: You *should not* include the workshop.css file in your <language> folder with your packaged control. If the file is present in your system, your installer may overwrite the installed version. In other words, the file is useful for authoring topics and applying styles, but should be ignored during packaging.

- ***Context−sensitive topics*** Be sure to place all help files that *are* used for context−sensitive help in the folder structure described below. The context−sensitive help mechanism (used when the user presses F1 while an API or tag is selected) is designed to identify the item selected, then look for a corresponding help topic according to the following rules.
  - <vendor_name>/java−class for Javadoc or similar references for Java APIs. Subfolders should reflect the API package structure, and HTML files should be named after the classes. For instance, the context−sensitive help for the ControlException class, which is part of the package com.bea.control, is located in java−class/com/bea/control/ControlException.html.

    All segments of the path are case−sensitive; the HTML file name must have exactly the same capitalization as the class name. Note that while the directory structure is the same as produced by Javadoc, the content of the files does not have to be Javadoc output. Javadoc is a convenient tool for producing reference documentation for Java classes, but you could place a manually constructed (non−Javadoc) conceptual topic at the expected location if you desire.
  - <vendor_name>/javadoc−tag for annotation references. Subfolders should reflect the prefix of the custom annotation, and HTML files should be named after the tags. For instance, the context−sensitive help for the @jc:sql annotation is located in javadoc−tag/jc/sql.html. All segments of the path are case−sensitive.
  - <vendor_name>/taglib for custom JSP tag library references. Subfolders should be structured based on the tag library URI. (The tag library URI is the value of the <uri> element in the tag library's TLD file.) For example, the context−sensitive help for the callControl tag, whose URI is http://www.bea.com/workshop/netui−tags−databinding−1.0, is located in <vendor_name>/taglib/www.bea.com/workshop/netui−tags−databinding−1.0/callControl.html. All segments of the path are case−sensitive.

- ***Other topics*** In the <vendor_name> folder, in any number and arrangement of subfolders, you can put any number of HTML help files that are *not* used for context−sensitive help. For instance, the WebLogic Workshop convention is to locate all help files that are not used for context−sensitive help in a guide folder. In your structure, this would be <vendor_name>/guide.

Creating Help Pages

WebLogic Workshop requires that help topics are HTML files. You can use the HTML editor of your choice to write them. To make these pages look and feel like the other help pages in WebLogic Workshop help, your topics should reference the CSS file workshop.css, which is included with the ControlDevKit sample at ExtensionDevKit/ControlDevKit/DBScripter/help/doc/en.

The workshop.css file contains detailed comments for each of the styles used. The styles you are most likely to use are:

- ***Title***. Use this style for the title of a help topic.
- ***h1***, ***h2***, and ***h3***. Use these styles for first, second, and third−level headings within the body of a topic.
- ***relatedTopics***. Use this style for the section *Related Topics*.
- ***procTitle***. Within the body of How Do I..? topics, use this style for headings that are followed by bulleted procedures. These headings generally start with *To do 'X'*.
- ***pre***. If you want to include a section with sample code, use the preformatted paragraph, and use four

spaces for indentation.
- ***langinline***. For code fragments, file names, class names, etc., within a (non−code) regular section, you should apply this style.
- ***notepara***. To add a note section, apply this style. Start a note with "***Note***:" (where "Note" is in bold and the rest of the note text is not).

To insert a link in your document, you can add the standard HTML tag <a href>. You can use relative links to refer to other help topics you provide. To refer to help topics in WebLogic Workshop help, you can either link to the online documentation at http://edocs.bea.com, or you can use relative links. Note that for relative links the path will include the folder weblogic81, which is the default version−specific product installation directory for WebLogic Workshop. During installation, users have the option of using a different folder name instead, and if they do so, relative links from your help topics to WebLogic Workshop help will no longer work.

Required Functions

Through JavaScript functions provided with WebLogic Workshop, your help topics can support three features that are available with the Workshop help system:

- Automatically syncing the table of contents to the topic displayed.
- Writing the topic's URL into the topic so that users can refer to it specifically.
- Specifying an email address to which topic feedback should be sent.

The JavaScript files and functions described here are included in the Help Test Kit, which you can find at the WebLogic Workshop Extensibility Portal page at dev2dev. You are strongly encouraged to use the Help Test Kit to validate your topics before packaging your extension for distribution. Also, use the included files for testing only; they are not needed in your deployed help because the files are present with WebLogic Workshop installed documentation.

Including the Required Script Files

To ensure that your topics have access to all of the functionality exposed by the help system, you will need to include three script files in the <head> section of each topic's HTML. These files are topicInfo.js, CookieClass.js, and displayContent.js. The src attribute of each <script> tag must resolve to a file in the "core" folder of the WebLogic Workshop help installation. By default, this folder is located at <workshop_home>/help/doc/<language>/core. In other words, the src attribute's value must be a path that is relative from the installed location of the current help topic to the core folder.

As described in Organizing Your Help System, your help topics are installed in <workshop_home>/help/doc/<language>/partners/<vendor_name>. Consider a vendor named "MyCompany" whose help has a topic at guide/introduction.html. Once installed, the English version of that topic would be located at:

<workshop_home>/help/doc/en/partners/MyCompany/guide/introduction.html

<script> tags in that topic's <head> section, with URLs to the required JavaScript files, would look like this:

```
<script language="JavaScript" src="../../../../core/topicInfo.js"></script>
<script language="JavaScript" src="../../../../core/CookieClass.js"></script>
<script language="JavaScript" src="../../../../core/displayContent.js"></script>
```

Supporting Automatic Syncing

Assuming you have created a toc.xml file as described in Creating a toc.xml File, you can support the help system's ability to automatically sync the table of contents to your topic when the topic is displayed. In other words, when the topic is displayed, the table of contents expands to that topic's place in the hierarchy so that users can see the topic's location in context.

To support this feature, each topic to which the table of contents should automatically sync must:

- Be represented by a corresponding element in the toc.xml file.
- Include a call to the displayInFrames() function provided by the help system.
- Include an <a> tag whose href attribute value resolves to the index.html file in the <workshop_home>/help/doc/<language>/core directory, and whose id attribute value is "index".

Note that the first two requirements are interdependent. If the function is not called in the topic's HTML the table of contents will not sync, whether or not the toc.xml includes the topic; if the topic was requested via context−sensitive help (by pressing F1), then the table of contents will not display at all. If the function *is* called, but the topic is not represented in the toc.xml, then a WebLogic Workshop default topic will be displayed in place of the requested topic.

Note: As you might have guessed, this represents a design decision. Do you want your help to participate in the WebLogic Workshop frameset and table of contents or not? If you do, you must provide a toc.xml that represents, and call displayInFrames() from, every topic. If you don't want to participate, you don't need to provide a toc.xml and you *should not* call displayInFrames().

For more information on creating a toc.xml file, see Creating a toc.xml File. To call the function, place the following code just after each topic's <body> tag, before other body content:

```
<script language="JavaScript">
    displayInFrames();
</script>
```

Finally, the third requirement mentioned above ensures that the topic will be able to find the frameset in order to display itself within the frameset. Due to the way the script works, the <a> tag must be the first anchor with an href attribute in the topic. Putting the following line of code either in the <head> section (along with the <script> tags) or immediately following the <body> tag should do the trick.

In keeping with the "introduction.html" example above, the line of HTML would look something like this:

```
<a href="../../../../core/index.html" id="index"></a>
```

Supporting Feedback Links and Current Topic URLs

WebLogic Workshop documentation supports a feedback mechanism that sends email to the documentation team. You can specify a different email address through the writeCustomTopicInfo JavaScript function. The same function also provides a way for you to specify the current topic's location, perhaps that topic's location at your company's web site; this URL will be printed at the bottom of the topic.

The function is described as follows:

writeCustomTopicInfo(*customHelpURL*, *customFeedbackAddress*)

The *customHelpURL* argument specifies a base URL at which the current topic may be found. The Workshop help system will append the current topic's relative URL to *customHelpURL*. For example, if you specify "http://mycompany.com/docs/" as *customHelpURL*, and the topic is located under the partners help directory at MyCompany/guide/introduction.html, the resulting URL will be "http://mycompany.com/docs/en/partners/MyCompany/guide/introduction.html". If you specify null for this argument, the help system will insert a path to the topic on the user's computer.

The *customFeedbackAddress* argument specifies the email address to which feedback should be sent. The Workshop help system will provide a form in which users can enter feedback details. The resulting information will be sent to the email address you specify in the *customFeedbackAddress* argument.

To support the feedback and URL features, place the following code just before each topic's </body> tag:

```
<script language="JavaScript">
    writeCustomTopicInfo("http://mycompany.com/docs/", "feedback@mycompany.com");
</script>
```

Making Search Results Useful

When a user searches the help topics, the search algorithm returns a list of the topic titles that match the search parameters. Specifically, the list will contain the topic title as given in the <title> tag in the <head> section of each help topic's HTML file, for example:

```
<title>WebLogic Workshop Help Authoring Guide</title>
```

In order for search to add your topic to the search list, you must include a <title> tag. If a help topic does not contain this tag, the search algorithm might return this help topic as matching, but it cannot add its topic title to the list of matching topics and users will not be able to select the topic.

When selecting a title for your topics, you should be careful to provide a title precise enough to identify the subject among many potential search results. Help for controls should include some form of the vendor name in every topic title. However, be aware that if you use the same long prefix in every topic title, the search panel will appear to return multiple identical results unless the search results panel is made wider. Be concise but informative.

For more information on the search algorithm, see the WebLogic Workshop help topic *Search Tips*.

Creating a toc.xml File

To add your help topics to the table of contents of WebLogic Workshop help, you need to provide a toc.xml file in help/doc/<language>/partners/<vendor_name>. When users choose to add your documentation to WebLogic Workshop when they add your control JAR file to the application, the help topics are added underneath the table of contents book *Extensions*.

The toc.xml file must comply with the XML schema defined in the toc.xsd file. A copy of toc.xsd can be found in the Help Test Kit at the WebLogic Workshop Extensibility Portal. The toc.xml file that you create should follow this template:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- 'My Company' TOC -->
<toc-root component="other" xmlns="http://www.bea.com/help/toc.xsd">
```

```
    <toc-reference anchor="extensions">
        <toc-node label="My Company"  url="somePageA.html">
            <toc-node label="My Control 1" url="somePageB.html">
                <toc-node label="My Control 1, Topic 1" url="somePageC.html"/>
                <toc-node label="My Control 1, Topic 2" url="somePageD.html"/>
                ...
            </toc-node>
            <toc-node label="My Control 2" url="somePage.html">
                ...
            </toc-node>
            ...
        </toc-node>
    </toc-reference>
</toc-root>
```

In the example, the book *My Company* has its own help topic *somePageA.thml* and contains a nested book *My Control 1*. This nested book in turn has its own help topic *somePageB.html*, a nested topic *My Control 1*, Topic 1 with a help topic *somePageC.html*, etcetera.

Your toc.xml file must contain a <toc−root> element exactly as shown in the example, followed by a <toc−reference> element, again exactly as shown in the example. The <toc−reference> element should hold exactly one <toc−node> element, with a label name that clearly reflects the name of your company and product. This element will create the top level book for your help topics, and will be placed beneath the table of contents book *Extensions.* Within this element you should nest all your help topics.

You should choose the label of the top−level <toc−node> directory carefully. A user may have a large number of extensions installed on his or her system. The name you choose must clearly and uniquely identify your company and product name.

Every <toc−node> element must have label and url attributes. The label contains the title of the topic in the table of contents, while the url contains a relative link to the help page. When the <toc−node> element represents a single help topic, the tag is closed at the end, as is shown in the following example:

```
<toc-node label="My Control 1, Topic 1" url="somePageC.html"/>
```

Note that the table of contents will automatically display as a book any <toc−node> entry that contains other entries. When the toc−node element represents a book, it will have a begin tag and an end tag, and its nested topics (or books) are contained between these two tags, as is shown in the following example:

```
<toc-node label="My Control 1" url="somePageB.html">
    <toc-node label="My Control 1, Topic 1" url="somePageC.html"/>
    <toc-node label="My Control 1, Topic 2" url="somePageD.html"/>
</toc-node>
```

Related Topics

workshop.css

Extension Samples

The ExtensionDevKit provides several samples you can browse and run to understand how the extensibility API works. These are divided among three separate WebLogic Workshop applications, each with multiple projects. Because build and deployment for extension types differs, each of the three applications has its own build and run characteristics. These differences are described below.

- Control Samples (ControlDevKit)
- IDE Extension Samples (IdeDevKit)
- Tag Library Extension Sample (TaglibDevKit)

Control Samples (ControlDevKit)

This application illustrates advanced controls, such as controls that have a custom insert wizard, JCX−generation capability, and so on. By default, the application is located here:

BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\ControlDevKit\ControlDevKit.work

Samples Included

This application has three projects.

ControlFeatures project

This project contains several controls, each of which focuses on a different feature of advanced control building.

- ***EventRaiser*** illustrates how you can pass a callback from a nested control through to a client.
- ***EventScheduler*** illustrates how you can schedule a control's callbacks to occur at specified times.
- ***ServerCheck*** illustrates how to connect a custom dialog box that will be used to collect attribute values when the control is added to a project.
- ***CustomWiz*** features ***CustomInsertDialog***, a control wizard implementation that returns a full dialog to prompt the control's user for information when they are inserting or creating the control.
- ***XQuery*** illustrates how to create a control that generates a JCX file.
- ***CustomerData*** illustrates how to connect a custom attribute editing/validation dialog.

DBScripter project

This project illustrates how you can structure a control project to support automatic deployment of your control's documentation and samples. It contains a reasonably full control treatment, including control sources, documentation, and samples.

ControlTest project

This project contains several web services, each designed to test one of the sample controls in use.

Getting Started

Due to the way control projects work, this application requires a little setup. First, you'll need to add wlw−ide.jar to the application's classpath to get the control projects to build. Also, because the ControlTest project in this application requires that the controls have been built, you won't be able to run the tests until after you've built the controls.

When you first open this application, you'll be unable to build its sources. This is because the control samples here require that wlw−ide.jar be on their classpath. This path isn't specified when the application is installed because the path itself is stored in the control author's (i.e., your) local preferences. For more information, see Adding wlw−ide.jar to the Classpath.

IDE Extension Samples (IdeDevKit)

This application illustrates IDE extensions, including menus, frames, custom project support, and the like. By default, the application is located here:

BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\IdeDevKit\IdeDevKit.work

Samples Included

This application includes the following seven projects.

- ***CustomProject*** illustrates how to add support for a new project type, including a new document type and support for running the project.
- ***DragDropSimple*** illustrates how to use WebLogic Workshop's support for DnD.
- ***FrameViewSimple*** is a simple dockable frame.
- ***MenuItems*** is a menu that's customizable at run time.
- ***PopupAction*** illustrates how to add a popup menu that's available by right−clicking an item in the Application window.
- ***PropertyListener*** listens for changes to IDE properties, such as when a document is edited or saved.
- ***ToolbarButton*** illustrates how to add a toolbar button.

Getting Started

These samples should run as installed. In general, to run a sample, open a file in its project and click the Start button. For specific information about what to look for in each, see their topics.

Tag Library Extension Sample (TaglibDevKit)

This application illustrates how to provide design−time support for a custom tag library. By default, the application is located here:

BEA_HOME\weblogic81\samples\workshop\ExtensionDevKit\TaglibDevKit\TaglibDevKit.work

Samples Included

This application includes a sample that illustrates how you can add design−time support for a custom tag library. This support includes, for example, having tags in the library available in WebLogic Workshop's

palette. For more information, see Tag Library Extension Samples.

Getting Started

You'll need to build the tag library and extension before trying them out. For more information, see Tag Library Extension Samples.

Related Topics

None.

Debugging Extensions

When debugging extensions, your setup needs to take into account the fact that the IDE only becomes aware of your extension at startup. With that in mind, these debugging recommendations describe a process in which WebLogic Workshop builds your extension and copies it to a place where the IDE can find it, then starts a new IDE instance that is aware of the newly built extension. Breakpoints you set in the first IDE instance will be recognized as you work with your extension in the second.

This topic includes:

Setting Up Extension Debugging Properties

Debugging IDE Extensions

Debugging Java Controls

Setting Up Extension Debugging Properties

When setting up for debugging, begin by ensuring that the properties for the project you're debugging specify the following values on the Debugger pane. If you want to load a separate application when the second IDE instance starts, see the section below.

To Specify IDE Extension Debugger Properties

1. In WebLogic Workshop, in the ***Application*** window, right−click the folder for your control project, then click ***Properties***.
2. In the ***Project Properties*** dialog, in the left pane, click ***Debugger***.
3. Specify the following values for debugger properties:

Option	***Value***
Build before debugging	Selected
Pause all threads after stepping	Cleared
Create new process	Selected
Attach to process	Cleared
Main class	workshop.core.Workshop If you don't want the IDE to load a specific application when it starts the second IDE instance, you can leave this empty.
Parameters	To load a specific application, enter here a path to the WORK file of the application to load. Note that this path must be relative to the path specified in the ***Home directory*** box.
VM parameters	*When debugging with 8.1 GA:* −ea −Xmx512m −Xms128m −Dsun.java2d.d3d=false −Dworkshop.home=../workshop

	When debugging with 8.1 SP2 and later: −ea −Xmx512m −Xms128m −Dsun.java2d.d3d=false −Djava.system.class.loader=workshop.core.AppClassLoader If you don't want the IDE to load a specific application when it starts the second IDE instance, leave this as the WebLogic Workshop home directory, which is automatically set by the installer.
Home directory	To load a specific application, it's a good idea to have something here. The path you give in Parameters to the WORK file of the application to load must be relative to what is given here. Having a path here will make this connection easier to keep track of.
Application classpath	Must include WORKSHOP_HOME/wlw−ide.jar.
Automatically append Library JARs	Selected
Automatically append server classpath	Selected
Smart debugging	Selected

Specifying an Application to Load for Testing

When you launch a new instance of the IDE for debugging an extension, you can specify that the new instance load a particular application. This can be useful if you've created a test application separate from the one you're creating your extension in. If you don't specify a particular application, WebLogic Workshop will load whatever it would have loaded on startup anyway typically the application that was open the last time you shut down the IDE.

You specify an app to load through the values you give for the ***Home directory*** and ***Parameters*** debugging properties.

For example, imagine you want to load the SamplesApp that is installed with WebLogic Workshop. SamplesApp might be located here:

C:\bea\weblogic81\samples\workshop\SamplesApp.work

You could make your Home directory the directory containing your extension application:

C:\bea\user_projects\applications\MyApp

In the Parameters box you'd enter the path to the SamplesApp application's WORK file, *relative to your Home directory*. In Parameters, you'd enter this:

../../../weblogic81/samples/workshop/SamplesApp/SamplesApp.work

Debugging IDE Extensions

When you debug an IDE extension, your build process needs to copy the build output (the extension ZIP file) to the WORKSHOP_HOME/extensions folder. That's where WebLogic Workshop looks for extensions to

load on startup. In other words, setting up for debugging IDE extensions includes:

- Creating an Ant build file that copies build output to the correct folder in addition to building your sources.
- Setting up extension debugging properties so that a new instance of the IDE is started.

Creating an Ant Build File for Use When Debugging

When you build, the resulting extension ZIP file goes into the WORKSHOP_HOME/extensions folder, where it can be found by the IDE on startup. This is a simple matter of creating an Ant build file and adding a zip task to the file generated by WebLogic Workshop.

The following steps assume you don't yet have a build.xml file you're building with.

1. Right−click the project folder, then click ***Properties***.
2. In the ***Project Properties*** dialog, on the ***Build*** panel, click ***Export to Ant file***, then click ***Cancel***.
3. In the ***Application*** window, locate the ***exported_build.xml*** file generated for you.
4. Rename ***exported_build.xml*** to simply ***build.xml***, then double−click the file to open it in ***Source View***.
5. In the ***build.xml*** file, scroll down to the "***build***" target, the add the following as the last task in the target:

   ```
   <zip destfile="${platformhome.local.directory}/workshop/extensions/${output.filename}"
           basedir="${dest.path}"
           includes="**/*.*"
           encoding="UTF8"> <!−− jar filenames are UTF8−encoded −−>
           <zipfileset dir="${project.local.directory}"
           excludes="build.xml,**/CVS/**,**/*.java,${output.filename}"
           includes="**/*.*"/>
   </zip>
   ```

   This zip task zips your extension output and copies it to the WORKSHOP_HOME/extensions folder, where WebLogic Workshop can find it when the IDE starts.

6. Save and close the file.
7. Right−click the project folder, then click ***Properties***.
8. In the ***Project Properties*** dialog, on the ***Build*** panel, click ***Use Ant build***. By default, the build target should be displayed as "build", and that's as it should be.
9. Click ***OK***.

Debugging Java Controls

When debugging Java controls, you'll need to specify debugging properties. You'll also need to ensure that wlw−ide.jar is visible to the IDE on the classpath.

When you build the control project, WebLogic Workshop will by default copy the resulting JAR file to the Libraries folder. That's where the IDE looks for controls.

Adding wlw−ide.jar to the Classpath

The IDE requires that the JAR be on *its* view of the classpath (the classpath used by the IDE for resolving dependencies, and so avoiding errors in Source View). This JAR contains the IDE extension API that's needed by controls that use an IDE extension, such as a custom insert wizard. Because a control project doesn't provide a project−specific a way to specify external JARs that are visible to the IDE's code checker, you'll need to add the JAR at the application level.

Note: Once the controls are built and deployed, they can locate wlw−ide.jar from within the Libraries folder where the control JAR ends up. You just need to do this to get them built.

To add wlw−ide.jar to an application's classpath where the IDE can see it:

1. Right−click the application folder (the top folder), then click ***Properties***.
2. On the right, scroll down to the bottom of the panel.
3. Under ***Server classpath additions***, click ***Add Jar*** and browse for the following JAR:

   BEA_HOME\weblogic81\workshop\wlw−ide.jar
4. Select the JAR, then click ***Select Jar***.
5. Click ***OK***.

Related Topics

None.

Getting Started with Extension UI Programming

Many WebLogic Workshop extensions you'll write will incorporate user interface components. Some advanced controls, for example, will display "insert wizard" custom dialog boxes to prompt the user for values required when creating a new JCX for the control. IDE extensions such as frame views can provide dockable user interface exposing a tool or displaying data to help increase a developer's productivity.

When building user interface for extensions, you use Java APIs specifically designed for UI. While WebLogic Workshop supports the use of both Abstract Window Toolkit (AWT) and Swing, it is recommended that you use Swing. WebLogic Workshop is itself built from Swing, and the extensibility API provides functions that are specifically designed for use with Swing components.

While a full introduction to Swing is beyond the scope of this documentation, this topic provides a few links that you might find helpful if you're new to that technology.

Swing, Briefly

First released in 1998, the Swing API is built on the Abstract Window Toolkit (AWT) API. AWT, Swing, and a few other APIs make up the Java Foundation Classes (JFC) suite of libraries. In general, Swing−based user interfaces are made up of *components* (such as panels, labels, and buttons) arranged together with a *layout manager*. A component in the UI responds to the user through *actions* that are connected to the component via a *listener*.

Here's a brief example from the tutorial in Getting Started: IDE Extension Tutorial. This example uses the default layout managers, so none is set here.

```
// Create a button with a label.
private javax.swing.JButton btnSayHello;
btnSayHello = new javax.swing.JButton();

btnSayHello.setText("Say Hello");

// Connect a listener that will handle button clicks.
btnSayHello.addActionListener(new java.awt.event.ActionListener()
{
    public void actionPerformed(java.awt.event.ActionEvent event)
    {
        // Use a Workshop API class to show a message dialog when the button is clicked.
        DialogUtil.showInfoDialog(null, "Hello World!");
    }
});
// Add the button to this component −− perhaps a JPanel.
add(btnSayHello);
```

You'll find UI examples in many of the extension samples included in the IdeDevKit sample application. For more information, see Samples That Use Swing. The Sun web site also provides general Swing tutorials here. In particular, Creating a GUI with JFC/Swing provides an introductory tutorial to Swing basics.

For a few useful Swing−related utility classes included for use with WebLogic Workshop, you might explore the com.bea.ide.util.swing package.

Components, Layout, and Events

Swing provides many components that you can draw from to build user interfaces. For an introduction to these, see this useful visual index to the Swing components. Swing also offers several ways for you to manage visual component arrangement. This visual guide to layout managers should help to get you started. (Remember, too, that if you're using a graphical design tool for building your UI, that tool will list available components and layout options.)

For an introduction to connecting event handlers, you might be interested in reading through the Writing Event Listeners portion of the Java tutorial.

Drag and Drop (DnD)

While it's not technically a part of Swing, the drag−and−drop (DnD) API is a part of the JFC, and it's often a key part of applications that have a user interface. It's certainly a key part of the WebLogic Workshop IDE. DnD support simply provides a way for a user to drag one part of the user interface to another. Dropping the dragged part prompts something to happen, such as transferring data from the drag source to the drop target. In a WebLogic Workshop web application, for example, users can drag an action from the Data Palette and drop it into Source View or Design View to add that action to the open document.

The Sun web site offers How to Use Drag and Drop and Data Transfer, a useful introduction and tutorial on DnD. Due to its number of moving parts, DnD is sometimes considered one of the more complex parts of programming user interfaces with Java. The WebLogic Workshop extensibility API provides IDE−specific wrappings for DnD functionality that make doing DnD a little easier. You'll find the DnD−related APIs in the com.bea.ide.core.datatransfer package. For information on a DnD−related sample, see DragDropSimple Sample. For an introduction to DnD in WebLogic Workshop extensions, see Adding Support for Drag and Drop.

Graphical Design for Java UI

As you've no doubt noticed, WebLogic Workshop doesn't provide much in the way of support for designing Swing user interface. To make the initial UI design process a little easier, you might try using a tool such as NetBeans. With a Swing−focused development tool, you can typically drag components onto a design area and arrange them in the way you like. As you edit the design graphically, the tool generates corresponding code behind the scenes. When you've got things looking the way you want them, simply open the generated code, copy the portion you need for UI, and paste the copied code into your class source in WebLogic Workshop.

UI Notes and Recommendations

- Avoid setting characteristics such as colors, fonts, and so on. Swing visual components typically expose accessors through which you can set, say, the component's back color or the font its text is displayed in. But if you resist setting these explicitly, you'll find that in most cases your results blend very nicely with the rest of the IDE.
- Where your visual results aren't what you thought they'd be, take a look at the LookAndFeelConstants class. You may find a match for your needs among the values represented by the constants there. To use one of these,

```
JProgressBar scoreBar = new JProgressBar(0,100);
```

```
scoreBar.setForeground(UIManager.getColor(LookAndFeelConstants.MINI_FRAME_INACTIVE_BORDE
```

- Scan the com.bea.ide.util.swing package, which provides useful shortcuts and workarounds for Swing programming.

Samples That Use Swing

All of the samples in the IdeDevKit application include some sort of user interface. But a couple of them are a bit more involved:

- In the DragDropSimple project, the SimpleTree class is an extension of JTree. It uses DefaultMutableTreeNode to represents nodes in the tree.
- In the PopupAction project, the FTPPrefsPanel class extends JPanel and arranges several components, including JPanel, JLabel, and JTextField. It also uses the GridBagLayout layout manager to arrange the pieces.

Related Topics

None.