BEA WebLogic Workshop™ Help

Version 8.1 SP4
December 2004

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Table of Contents

Table of Contents

The WebLogic Workshop Development Environment

WebLogic Workshop is an integrated development environment for building enterprise−class J2EE applications on the WebLogic Platform. WebLogic Workshop provides an intuitive programming model that enables you to focus on building the business logic of your application rather than on complex implementation details. Whether you are an application developer with a business problem to solve or a J2EE expert building business infrastructure, WebLogic Workshop makes it easy to design, test, and deploy enterprise−class applications.

WebLogic Workshop consists of two parts: an Integrated Development Environment (IDE) and a standards−based runtime environment. The purpose of the IDE is to remove the complexity in building applications for the entire WebLogic platform. Applications you build in the IDE are constructed from high−level components rather than low−level API calls. Best practices and productivity are built into both the IDE and runtime.

WebLogic Workshop is available in two editions:

- ***WebLogic Workshop Application Developer Edition*** includes the basic features required by application developers to build web services, web applications, custom controls and Enterprise JavaBeans (EJBs). All editions of WebLogic Workshop also include ubiquitous support for XMLBeans, BEA's technology for seamless, natural manipulation of XML in Java.
- ***WebLogic Workshop Platform Edition*** includes additional extensions to the IDE and runtime framework that let you build portal applications and business processes in conjunction with the WebLogic Portal and WebLogic Integration products, respectively.

WebLogic Workshop's intuitive user interface lets you design your application visually. Controls make it simple to encapsulate business logic and connect to enterprise resources, like databases and Enterprise JavaBeans, without writing a lot of code. Conversations handle the job of keeping track of application state. And WebLogic Workshop's support for asynchronous processes makes it easy to build highly reliable applications for the enterprise.

The following topics provide information on working with the WebLogic Workshop integrated development environment.

Developing Applications with WebLogic Workshop

Covers basic concepts you need to understand to build applications in WebLogic Workshop.

Getting Started with WebLogic Workshop

Directs you to hands−on tutorials and samples to help you get started with WebLogic Workshop.

What's New in WebLogic Workshop 8.1

WebLogic Workshop 7.0 simplified development of enterprise−class web services for J2EE experts and corporate application developers alike. With WebLogic Workshop 8.1, BEA expands that vision to make WebLogic Workshop the integrated development environment of choice for building complete J2EE applications. The following sections outline some of the new features in WebLogic Workshop.

Shared Development Environment for BEA WebLogic Platform

Now you can use the WebLogic Workshop development environment to build WebLogic Platform applications in a friendly, graphical development environment. You can build web applications and web services. You can build a portal using WebLogic Portal. You can add business processes to an application using WebLogic Integration.

Java Controls

WebLogic Workshop offers a new extensible control model for creating Java controls. Corporate developers and ISVs can create new control types and use them in Workshop applications. Controls can be simple reusable components or sophisticated components with design−time customization. Controls are the conceptual heart of WebLogic Workshop — once you encapsulate a piece of business or application logic in a control or build a control that represents an enterprise resource, that control can be used consistently from any component in the application.

Web Application Development

WebLogic Workshop provides a new JSP/HTML editor which you can use to build web applications. You can add controls to your JSP pages to easily access enterprise resources. WebLogic Workshop also includes *page flows*, based on the Apache Struts framework, for coordinating JSP pages into web applications.

Sophisticated Java Integrated Development Environment

The WebLogic Workshop IDE has been completely redesigned and now offers complete application and project management for J2EE applications, a versatile editor for handling a variety of file types in the J2EE application, an integrated debugger, and integrated Ant build scripts.

Related Topics

None.

Getting Started with WebLogic Workshop

The quickest way to get started with WebLogic Workshop is to work through the tutorials and experiment with the samples. The following sections point you in the right direction.

IDE Quick Tour

For a quick tour of the WebLogic Workshop Integrated Development Environment (IDE), see IDE Quick Tour.

The Getting Started Tutorials

The Getting Started tutorials offer a basic step−by−step introduction to the core J2EE components and how you can develop these components with WebLogic Workshop. Each tutorial is self−contained, and can be run independently of any of the other tutorials. You can do any or all of these tutorials in any order you want.

Getting Started: Web Services

Getting Started: Web Applications

Getting Started: Java Controls

Getting Started: Enterprise JavaBeans

Getting Started: Portals

Tutorials

These tutorials are designed to help the new user get up and running quickly, but provide more depth than the Getting Started Tutorials. Each tutorial helps you to build a working application. Each tutorial step is self−contained, so that after completing a step you have a working application. At each step you can continue working through the tutorial to build a progressively more sophisticated application, or stop and adapt what you've learned to your own needs.

Get started with the following tutorials:

Tutorial: Java Control

Build and test the Investigate Java control.

Tutorial: Web Service

Build and test two web service access points to the Investigate Java control.

Tutorial: Page Flow

Build and test a web−based user interface for the Investigate Java control.

Tutorial: Building Your First Business Process

Builds a basic WebLogic Integration application.

Tutorial: Building Your First Data Transformation

Build an XML−to−XML data transformation in a WebLogic Integration application.

Tutorial: Enterprise JavaBeans

Build Enterprise JavaBeans.

Samples

The samples are working applications that you can explore to learn more about building applications with WebLogic Workshop. They range from simple to complex, and are designed to demonstrate the features you'll want to know about while building your own applications.

The SamplesApp application contains sample Page Flows and Web Services. To view the SamplesApp project, start WebLogic Workshop. From the ***File*** menu, choose ***Open***−−>***Application***. Navigate to the BEA_HOME/bea/weblogic81/samples/workshop directory and select the SamplesApp.work file.

Learn more about the samples in the following topics:

Page Flow and JSP Samples

Learn about samples that demonstrate the framework for page flows and JSP pages.

Sample Web Services

Learn about samples that demonstrate how to build web services with WebLogic Workshop.

WebLogic Integration Samples

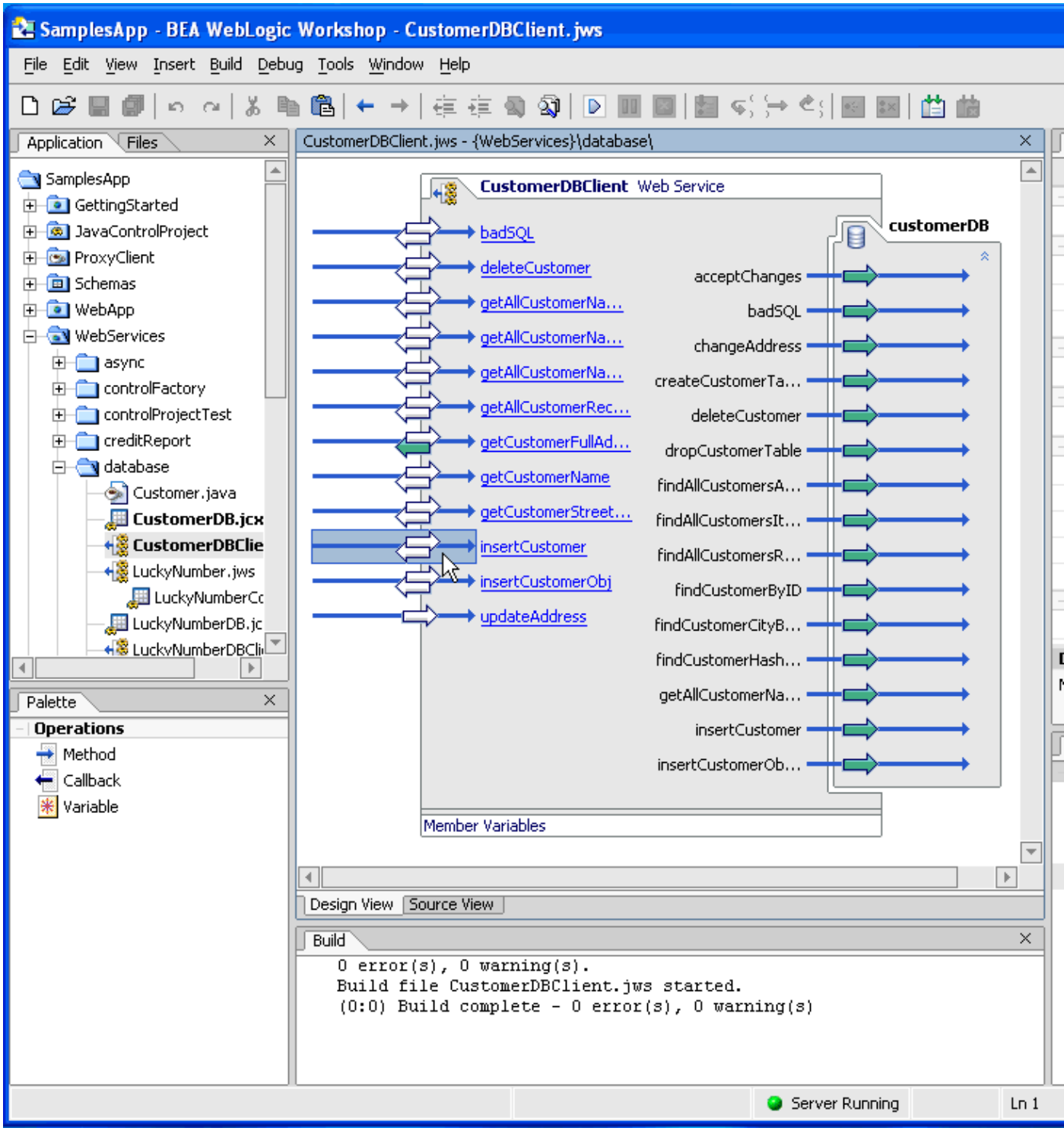
Learn about samples that demonstrate how to build integration applications with WebLogic Workshop.

WebLogic Portal Samples

Learn about samples that demonstrate how to build portlets with WebLogic Workshop.

WebLogic Workshop IDE Quick Tour

Move your cursor around the image of the WebLogic Workshop IDE (Integrated Development Environment). Tooltips will appear that explain the various area of the user interface.



The WebLogic Workshop IDE adapts itself in several ways to the type of file being edited. The file being edited in the preceding image is a Java Web Service, or JWS file. The image shows the Design View for a

web service. There are custom design views and custom source editors for all of the file types that WebLogic Workshop creates.

Filtering Files and Directories in the Application Tree View Tab

You can prevent files and directories located in your application directory from appearing in the tree view of the WebLogic Workshop Application tab. Workshop already filters certain file types and directory names by default. To make adjustments to these filters, simply open the applications .work file located in the applications root directory and change the value of the excludedirs or excludefiles attributes. These attributes are listed in red below:

```
<option name="excludedirs" value="cvs;sccs;rcs;" />
<option name="excludefiles" value="\..*|.*\.[0-9]+|wlw_[0-9]+\..*" />
```

The excludedirs attribute lists the names of directories WebLogic Workshop will not list in the Application tab's tree view. Alter the regular expression value of the excludefiles attribute if you want WebLogic Workshop to exclude certain types of files.

Related Topics

None

Developing Applications with WebLogic Workshop

The topics in this section address the basic concepts you need to understand to build entrprise−class applications with WebLogic Workshop.

Topics Included in This Section

What Can I Build with WebLogic Workshop?

Describes the building blocks of WebLogic Workshop applications.

Applications and Projects

Describes how applications are organized in WebLogic Workshop.

WebLogic Workshop File Types

Describes the file types specific to WebLogic Workshop.

Debugging Your Application

Provides an overview of debugging in WebLogic Workshop.

Java and XML Basics

Offers an introduction to Java and XML for the developer who is new to these technologies.

Related Topics

None

What Can I Build with WebLogic Workshop?

The WebLogic Workshop IDE builds enterprise−class applications that run in the WebLogic Workshop runtime. The applications you build in WebLogic Workshop typically expose systems and data within or between enterprises, typically via web applications and/or web services.

As an example, considering an express shipping company. Such a company might want to expose shipment scheduling, tracking and billing data to its business partners via web services so that partners' applications can access the data directly. The company also might want to expose tracking information via one or more web applications so that shipment originators and recipients can check the status of shipments from a web browser. WebLogic Workshop makes it easy to construct common functionality for both applications and then expose that functionality with appropriate interfaces.

In WebLogic Workshop Application Developer Edition, the core components are:

- Java controls
- Web services
- Web applications
- Enterprise Java Beans (EJBs)

In WebLogic Workshop Platform Edition you can also build:

- Portal Applications
- WebLogic Integration components: business processes, data transformations, integration−specific controls

What is an "Enterprise−class" Application?

"Enterprise−class" applications exhibit specific attributes. The J2EE foundation on which the WebLogic Platform and WebLogic Workshop is built provides reliability, availability and scalability as well as caching, security and transaction support. "Enterprise−class" also refers to additional specific attributes with regard to web services and web applications:

Enterprise−class web services are loosely−coupled and coarse−grained and optionally asynchronous. To learn more about how WebLogic Workshop enables these attributes, see Why Build Web Services with WebLogic Workshop?.

An enterprise−class web application exhibits a separation between its application logic and its presentation logic. WebLogic Workshop enables this separation with Java page flows, which are based on the well−established Model−View−Controller pattern. To learn more about Java page flows, see Developing Web Applications.

How WebLogic Workshop Simplifies Development

As you read about the various components of a WebLogic Workshop application in the sections that follow, there are two common threads you will notice: the use of Java classes with powerful custom Javadoc annotations and the presence of custom design and development tools for each component type.

Javadoc Annotations

Each of the major components of a WebLogic Workshop application is expressed as a single Java class that is annotated with special Javadoc annotations (also called Javadoc annotations). The Javadoc annotations indicate to the WebLogic Workshop runtime that the component developer wants to utilize or configure particular features of the runtime.

For example, the @common:operation Javadoc annotation on a method in the Java class defining a web service indicates that that method should be exposed as an operation in the web service's WSDL file. As another example, the @jpf:forward Javadoc annotation on a method in the Java class defining a Java page flow indicates the page to which the user should be directed when that action is invoked. Via these annotations, WebLogic Workshop hides the vast majority of the complexity of implementing sophisticated applications. The Javadoc annotations are typically managed for you; they are represented as "properties" in the IDE.

The fact that each component type is expressed in a single Java file is also significant. In WebLogic Workshop, you never have to manage generated stub files or deployment descriptors. The WebLogic Workshop runtime analyzes the annotations and performs all of the required infrastructure plumbing automatically. As a developer, you can focus completely on the business logic of your application.

While each of the core components of a WebLogic Application is expressed as a 100% pure Java class, WebLogic Workshop uses different filename extensions to denote the component types. For example, a web service is stored in a file with the .jws extension indicating *J*ava *W*eb *S*ervice.

To learn more about the filename extensions you will encounter while working with WebLogic Workshop, see WebLogic Workshop File Types.

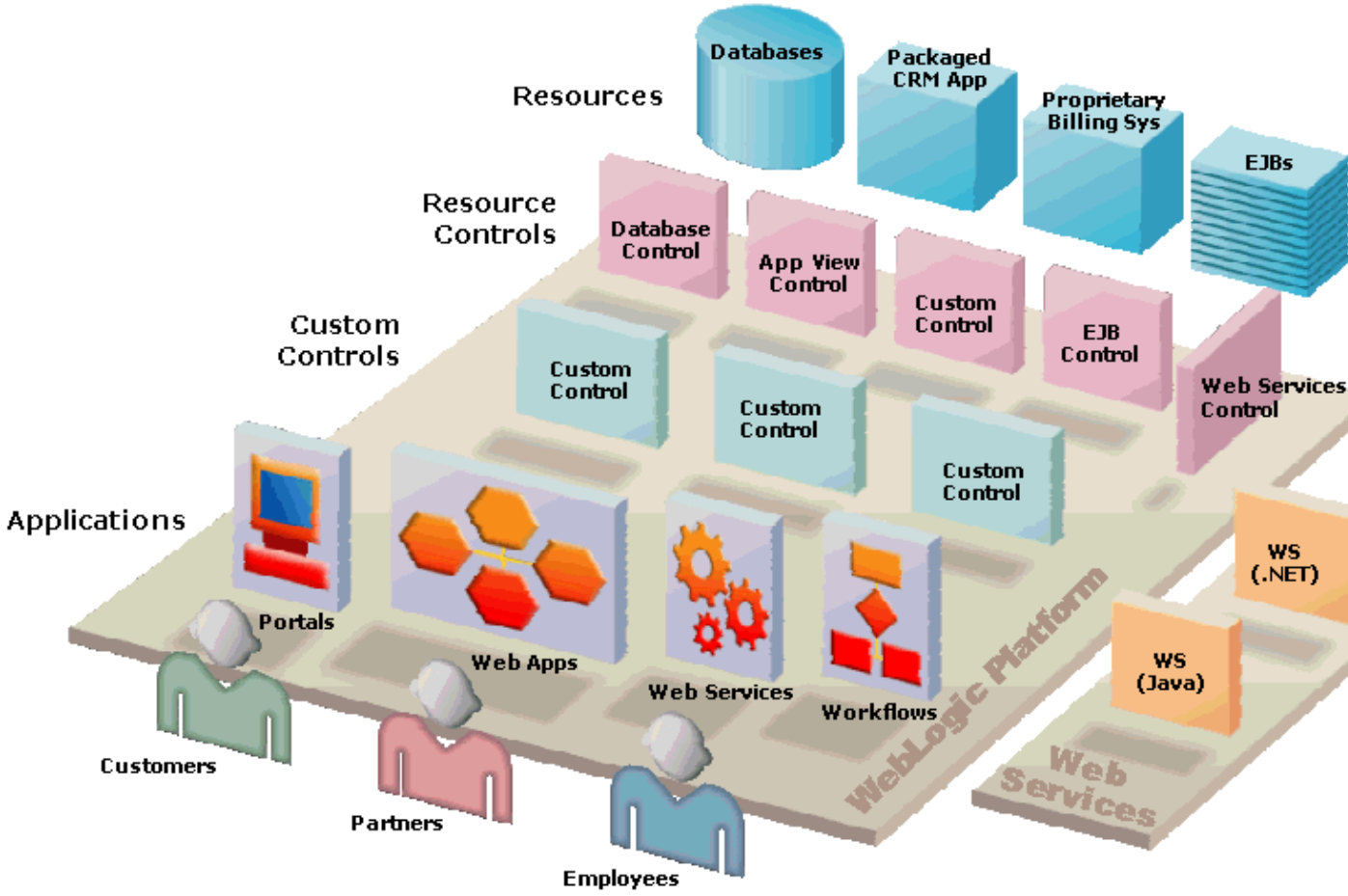
Component−Specific Development Tools

WebLogic Workshop provides customized editors for all of the component types. Most component types have a dedicated Design View, which provides an intuitive graphical view of the component under development and the other components with which it interacts. WebLogic Workshop also provides customized Source Editors with all of the features developers expect in an IDE including source code completion, syntax coloring and error and warning indications in real−time. In the cases where multiple languages appear in the same source file (for example, when fragments of XML are used in a web service source file to specify XML <−> Java mapping), the Source Editor automatically adjusts its behavior to the language of the file segment in which the cursor is currently located.

Finally, WebLogic Workshop provides full two−way editing: any changes you make in the source view of a component are immediately reflected in the graphical view and vice versa.

Architecture of a WebLogic Workshop Application

The following illustration shows the various components that might make up a WebLogic Workshop application.



Java Controls

All of the entities in the diagram that include the word "control" are *Java controls*. A Java control encapsulates business logic or facilitates access to an enterprise resource. Once a Java control has been created, that control can be used easily and consistently from a web service, a page flow, a portal, a business process or another Java control.

A Java control is a Java class that is annotated with special Javadoc annotations that enable features in the control when it is executed in the WebLogic Workshop runtime. A Java control may define methods and events. From the application developer's point of view, the consistent architecture of Java controls serves to greatly reduce the complexity of using the enterprise resources or business logic they encapsulate. Since all Java controls are Java classes, productivity aids such as source code−completion are available throughout the WebLogic Workshop IDE to streamline the development process.

When designing a Java control, you can see the relationship between the control, its client, and any other controls in the IDE's Design View. All of the important development activities can be accomplished by drag−and−drop or context−menu actions.

WebLogic Workshop includes several built−in controls that facilitate access to commonly used enterprise resources including databases, Enterprise Java Beans (EJBs), external web services and Java Message Service (JMS) queues and topics. You can create custom Java controls to encapsulate any business logic that is required by your application.

Business or application logic might be contained in a variety of components, including EJBs or other applications. If you have the choice, however, encapsulating business or application logic in a Java control leverages the full power of the WebLogic Workshop architecture, including asynchronous execution and conversations ("conversation" is the WebLogic Workshop model for a long−running transaction).

To learn more about Java controls, see Working with Java Controls.

Web Services

A Web Service is a piece of application logic that is exposed via standards such as Simple Object Access Protocol (SOAP; a specific dialect of XML)) and Web Services Description Language (WSDL). To use a web services, an application sends an XML message that requests an operation be performed. The application typically receives an XML response indicating or containing the results of the operation.

WebLogic Workshop makes it very easy to create and deploy web services that adhere to web service standards, to access and process the XML messages received by web services and to format XML responses. Specifically, WebLogic Workshop provides two powerful technologies for manipulating XML in web services: XML Maps using XQuery for simple inline Java <−> XML mapping specifications and XMLBeans for comprehensive Java <−> XML binding.

In WebLogic Workshop, a web service is a Java class that is annotated with special Javadoc annotations that provide simplified access to advanced web service features such as asynchrony, conversations, security and reliable messaging. Like all WebLogic Workshop components, Java web services can use Java controls to access business logic or enterprise resources. Everything you need to completely define a web service's operations, protocols, message formats and runtime behavior is contained in the web service's JWS file. There are no deployment descriptors to decipher and no external file generation to manage. WSDL generation for web services is completely automatic.

When designing a web service, the IDE's Design View shows you the relationship between the web service, its client, and any controls used by the web service. All of the important development activities can be accomplished by drag−and−drop or context−menu actions.

To learn more about Java web services and the advanced features available to web services in WebLogic Workshop, see Building Web Services.

While this topic is concerned mostly with components you build with WebLogic Workshop, it is important to note that WebLogic Workshop can easily inter−operate with web services built with other tools. One of the built−in Java controls WebLogic Workshop provides is the Web Service control. WebLogic Workshop can automatically generate a Web Service control from any valid WSDL file. Subsequently, the generated Web Service control can be used from any WebLogic Workshop component to access the remote web service as though it were a simple Java class.

Web Applications

To enable construction of dynamic, sophisticated web applications, WebLogic Workshop provides *Java page flows*. A page flow links together multiple web pages in a web application and provides a central control mechanism that coordinates the user's path through the pages and the associated flow of data.

A page flow is a Java class that is annotated with special Javadoc annotations that controls the behavior of the web application Page flows use methods, and in most cases, forms and form beans (Java classes containing

the data associated with a form) to manage navigation and state. A directory that contains a page flow class typically also includes one or more Java Server Pages (JSPs). The JSP files can reference custom WebLogic Workshop JSP annotations to raise actions, bind user interface components to data and access other application facilities. The actions referenced in a JSP correspond to action methods that are defined in the page flow class. These action methods implement code that can result in site navigation, data management, or invocation of business logic via Java controls. Significantly, the business logic in the page flow class is separate from the presentation code defined in the JSP files.

When designing a page flow, the IDE's Flow View shows the relationship between the pages in the web application and the actions that link the pages. All of the important development activities can be accomplished by drag−and−drop or context−menu actions. WebLogic Workshop also provides wizards to create specific types of page flows, generating the Java and JSP files that serve as a starting point for sophisticated applications.

Page flows are based on the Struts architecture, which is itself based in the popular Model−View−Controller user interface design pattern. Page flows add powerful, scalable support for state management and page flows and the JSPs they manage also have complete access to Java controls to access business logic or enterprise resources.

To learn more about Java page flows, see Developing Web Applications.

Enterprise Java Beans

Enterprise Java Beans (EJBs) are server−side Java software components of enterprise applications. The J2EE Specification defines the types and capabilities of EJBs as well as the environment (or container) in which EJBs are deployed and executed. From a software developer’s point of view, there are two aspects to EJBs: first, the development and deployment of EJBs; and second, the use of existing EJBs from client software.

WebLogic Workshop enables you to create new session, entity and message driven EJBs using a custom Design View. To learn how to create EJBs in WebLogic Workshop, see Developing Enterprise Java Beans.

WebLogic Workshop also provides a built−in control, called the EJB control, that makes it easy to use an existing, deployed EJB from your application. To learn more about the EJB control, see EJB Control.

Portal Applications

WebLogic Workshop Platform Edition adds the WebLogic Workshop Portal Extensions™ which allow you to build portals and portal resources using the WebLogic Workshop IDE. The portals you build are deployed using WebLogic Portal.

A portal represents a Web site that provides a single point of access to applications and information and may be one of many hosted within a single WebLogic Portal server.

Portals are becoming more and more important to companies who need to provide employees, partners, and customers with an integrated view of applications, information, and business processes. WebLogic Portal meets these needs, allowing companies to build portals that combine functionality and resources into a single interface while enforcing business policies, processes, and security requirements, and providing personalized views of information to end users.

From an end user perspective, a portal is a Web site with pages that are organized by tabs or some other form

of navigation. Each page contains a nesting of sub−pages, or portlets—individual windows that display anything from static HTML content, dynamic JSP content or complex Web services. A page can contain multiple portlets, giving users access to different information and tools in a single place. Users can also customize their view of a portal by adding their own pages, adding the portlets they want to it, and changing the look and feel of the interface.

The business problem that portals solve is illustrated in the following example. A company has the need for several types of Web presence: an Intranet for its employees, a secure site for interactions with partners, and a public Web site. WebLogic Portal's flexible portal network architecture supports multiple implementation choices which allow re−use of resources across portals.

To learn more about building portals in WebLogic Workshop, see WebLogic Portal Overview.

WebLogic Integration Components

WebLogic Workshop Platform Edition adds the capability to build WebLogic Integration components using the WebLogic Workshop IDE.

WebLogic Integration enables you to design business processes that span applications, users, enterprise networks, and trading partners.

WebLogic Integration's business process management (BPM) functionality enables the integration of diverse applications and human participants, as well as the coordinated exchange of information between trading partners outside of the enterprise. A *business process* is a graphical representation of a business process. Business processes allow you to orchestrate the execution of business logic and the exchange of business documents among back−end systems, users and trading partners (systems and users) in a loosely coupled fashion. WebLogic Workshop Platform Edition enables you to create business processes graphically, allowing you to focus on the application logic rather than on implementation details as you develop.

A business process can utilize *data transformations* using either a query or an eXtensible Stylesheet Language Transformation (XSLT). WebLogic Workshop Platform Edition includes a data mapper to create data transformations graphically. From the graphical representation of a data transformation, WebLogic Workshop generates a query. The generated query is invoked at runtime by the business process to transform data. A query is expressed in the XQuery language—a language defined by the World Wide Web Consortium (W3C) that provides a vendor independent language for the query and retrieval of XML data.

WebLogic Integration also provides a standards−based integration solution for connecting applications both within and between enterprises. WebLogic Integration provides the following tools for integrating applications: *application views*, the Adapter Development Kit (ADK), *EIS adapters* and *Application View Controls*. By using these tools, you can integrate all your enterprise information systems (EIS). Typical IT organizations use several highly specialized applications. Without a common integration platform, integration of such applications requires extensive, highly specialized development efforts.

To learn more about building WebLogic Integration components in WebLogic Workshop, see Building Integration Applications.

Related Topics

Getting Started with WebLogic Workshop

Applications and Projects

Applications and Projects

In WebLogic Workshop you create and build an *application*. A WebLogic Workshop application is a J2EE Enterprise Application and ultimately produces a J2EE Enterprise Application Archive (EAR) file. An application may contain:

- one or more *projects*
- *libraries* and *modules*
- *security roles*

This topic introduces applications and their contents.

Applications

An application in WebLogic Workshop is the collection of all resources and components that are deployed as a unit to an instance of WebLogic Server. It is also the top−level unit of work that you manipulate with WebLogic Workshop IDE. In the IDE, you may have at most one application open at a time.

When you create a new application, you specify a name for the application. By default, new applications are created in the BEA_HOME/weblogic81/samples/workshop folder of your installation, but you should create your applications elsewhere so they cannot cause conflicts when upgrading to future versions of WebLogic Platform.

An application in WebLogic Workshop contains one or more *projects*. Except in specific cases, such as accessing remote EJBs or web services, a WebLogic Workshop application is self−contained. Components in the projects of an application may reference each other, but they may not generally reference components in other WebLogic Workshop applications.

An application — more formally called an enterprise application — is different from a *web application*. Web application refers to a collection of resources, typically HTML or JSP pages, that allow users to access functionality via the Web. An application may contain zero or more web applications.

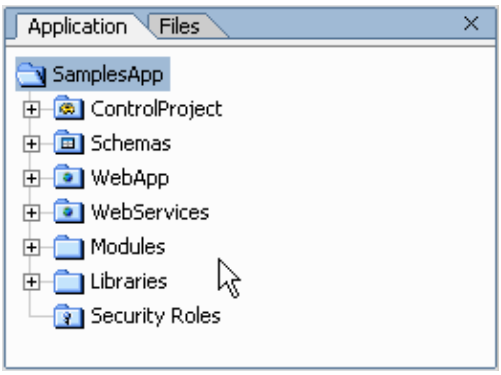
When to Create a New Application

An application should represent a related collection of business solutions. For example, if you were building an e−commerce site you might design an application that includes web applications for catalog shopping and customer account management and the components that support them (Java controls, EJBs, etc.). You might also need to deploy a human resources application for use by your employees. The human resources application is not related to the e−commerce application. Therefore, in WebLogic Workshop you would probably create two applications to separate these disparate business functions.

Applications Installed with WebLogic Workshop

WebLogic Workshop is installed with an application named SamplesApp, which contains example projects that you can explore to learn about WebLogic Workshop. The SamplesApp application is located in the file system at BEA_HOME/weblogic81/samples/workshop/SamplesApp.

The following image shows the Application pane for the SamplesApp application:



To learn more about the available samples, see WebLogic Workshop Samples.

WebLogic Workshop is also installed with an application named TutorialsApp that contains components used in the product tutorials. The TutorialsApp is located in the file system at BEA_HOME/weblogic81/samples/platform/TutorialsApp.

To learn more about the available tutorials, see WebLogic Workshop Tutorials.

Projects

A project groups related files that comprise a component of an application.

In WebLogic Workshop Application Developer Edition, there are six default project types:

- Web project
- Web Service project
- Control project
- Control Deliverable project
- EJB project
- Java project
- Schema project

WebLogic Workshop Platform Edition adds the following project types:

- Portal project
- Datasync project
- Process project

WebLogic Workshop project types are defined by *project templates*. You may create custom project templates to meet the needs of your organization.

Web Project and Web Service Project

Web projects are typically used to expose enterprise application logic via a user interface. The user interface is constructed from Java Server Pages (JSPs), which are web pages that can interact with server resources to produce dynamic content. WebLogic Workshop defines Java Page Flows that define and contain the logic required to connect multiple JSPs. Web services are typically used to expose enterprise application logic to applications (as opposed to users). Individual web service interfaces are published via Web Services Description Language (WSDL) files.

Each Web project or Web Service project ultimately produces a J2EE web application, each of which is included in the complete WebLogic Workshop application's EAR file when the application is built for deployment.

Contents of Web projects and Web Service projects are accessed via URLs. The WebLogic Workshop application is implicit in the URL for resources within that application. For example, the index.jsp sample JSP, which is located in the file system at SamplesApp/WebApp/index.jsp, is accessed from a web browser with the following URL:

```
http://localhost:7001/WebApp/index.jsp
```

Note that the name of a project becomes part of the public URL of each resource located within that project. You should choose your project names with that in mind.

You can configure a Web or Web Service project to load by default if you do not want to require users to provide a project name as part of the URL. For example, users can access http://localhost:7001/WebApp/index.jsp by simply requesting http://localhost:7001/.

To configure a project to load by default:

1. From the **Application** tab, right−click on the Web or Web Service project that you would like Web Logic Server to load by default and select **Properties**.
2. In the **Project Properties** dialog, select the **Use project name** check box..
3. In the text box labeled **Context Path**, enter / and Click **OK**.

The SamplesApp sample application contains a sample Web project named WebApp and a sample Web Service project named WebServices.

To learn more about Java page flows and building web applications, see Getting Started with Page Flows.

To learn more about Java Web Services, see Building Web Services with WebLogic Workshop.

Note: Web projects and a Web Service projects are both J2EE web applications. Web services may be created in a web application and page flows and JSPs may be created in a web service project. The only difference between the project types is the set of supporting files the project initially contains. If you add web services to a web project or page flows or JSPs to a web service project, the necessary supporting files are automatically added to the project.

Control Project

Java controls are entities that contain business logic or provide convenient access to enterprise resources. A control project may be used to construct and package controls that are intended for use by components in other projects. For example, a custom control that provides access to a proprietary Human Resources application might be used from JSPs in a web project and also from web services in a web service project.

Each Control project ultimately produces a Java Archive (JAR) file.

The SamplesApp samples application contains a sample Control Project named ControlProject.

To learn more about Java Controls and Control Projects, see Working with Java Controls.

Control Deliverable Project

You use a control deliverable project when you want to build Java controls that will be distributed to multiple users. A control deliverable projects creates directories you can use to store help and sample files relating to the control. When you build a control deliverable project, WebLogic Workshop packages the controls of that project into a Java Archive (JAR) file along with the help and sample files that you provide in the generated help and sample directories. When users install the control, WebLogic Workshop automatically integrates the help and sample files contained in the JAR with the users existing WebLogic Workshop help installation.

To learn more about Control Deliverable Projects, see Control Deliverable Projects.

EJB Project

Enterprise Java Beans (EJBs) are Java software components of enterprise applications. The J2EE Specification defines the types and capabilities of EJBs as well as the environment (or container) in which EJBs are deployed and executed. From a software developer’s point of view, there are two aspects to EJBs: first, the development and deployment of EJBs; and second, the use of existing EJBs from client software. An EJB Project provides support for creating and deploying new EJBs.

Each EJB project ultimately produces a Java Archive (JAR) file and when an EJB project is built the EJBs in the project are automatically deployed to the development server.

The EJB Control that is a built−in control in WebLogic Workshop allows clients such as web services, JSP pages or other controls to access existing EJBs, including EJBs created in an EJB Project.

To learn more about developing EJBs with WebLogic Workshop, see Developing Enterprise Java Beans.

Java Project

A Java project is a place to develop or collect general−purpose Java code that is not directly part of special entities such as web services, controls or EJBs. The product of a Java project is a JAR file that typically holds Java classes you want to access from other parts of your application. For example, a Java Project might hold Java code for a library of string formatting functions that are used in web services and JSPs in other projects in the application.

By default, each Java project ultimately produces a Java Archive (JAR) file.

To learn more about creating a Java project, see How Do I: Create and Debug a Java Class?

Schema Project

A Schema project provides convenient, automatic access to BEA Systems' XMLBeans functionality. XMLBeans is a facility that constructs Java classes from XML Schema and allows very easy, direct and powerful manipulation of XML documents in Java. If you place an XML Schema (XSD) file into a schema project, the XMLBeans schema compiler is automatically run on the schema and the resulting JAR file is automatically placed in the Libraries folder of the application. The Java classes in the JAR file are accessible to all projects in the application. For example, a web service in a web service project in the application can reference a schema as the input type of one or more of its methods and can automatically access incoming XML documents via the XMLBeans Java APIs.

The SamplesApp samples application contains a sample Schema Project named Schemas. It contains XML Schemas used by various samples in the SamplesApp application.

Note: For XML Schemas to be available in a business process, the schemas must be imported into a Schemas project in your integration application. To learn more about using XML Schemas in integration applications, see Guide to Data Transformation.

To learn more about XMLBeans, see Getting Started with XMLBeans.

Portal Project

Portal projects are available in WebLogic Workshop Platform Edition.

Portals are used to provide a single point of access to enterprise applications and information that can be based on user, group and role attributes.

A Portal project is similar to a Web project in that it is used to expose enterprise application logic via a user interface. Portal projects usually contain Java Server Pages (JSPs), and they provide additional capabilities, in addition to those available in Web projects, that allow developers to define portal Web sites.

Note: A Portal project is a J2EE web application and supports the creation of Web services, page flows and JSPs. The difference between a Portal project and a Web or Web Service project is the set of supporting files the project initially contains. Portal projects contain additional files that support creation of portals.

To learn more about portals, see WebLogic Portal Overview.

To learn more about and building portal applications in WebLogic Workshop, see Getting Started with WebLogic Portal.

Datasync Project

Datasync projects are available in WebLogic Workshop Platform Edition.

A Datasync project is used to develop and manage general purpose portal services that are used in the development of personalized applications and portals. These portal services include user profiles, user segments, events, request properties, session properties, content selectors, placeholders, and campaigns. A Datasync project called data is generated when a Portal application is created.

Process Project

Process projects are available in WebLogic Workshop Platform Edition.

A business process orchestrates the execution of business logic and the exchange of business documents among back−end systems, users and trading partners (systems and users) in a loosely coupled fashion. A Process project typically contains business process (JPD) files, control files and transformation files.

To learn more about Business Process Projects, see Guide to Building Business Processes.

When to Create a New Project

As you develop an application, you may need to create new projects within the application for the following reasons:

- To separate unrelated functionality

  Each project should contain components that are closely related to each other. If, for example, you wanted to created one or more web services that expose order status to your customers and also one or more web services that expose inventory status to your suppliers it would make sense to organize these two sets of unrelated web services into two projects.
- To control build units

  Each project produces a particular type of file when the project is built. For example, a Java project produces a JAR file. If you want to reuse the Java classes contained in the Java project from multiple components, it would make sense to segregate the Java classes into a separate project and reference the resulting JAR file from other projects in your application.

Libraries and Modules

In addition to projects, each WebLogic Workshop application also contains two folders named Libraries and Modules. These folders can contain compiled Java code that you want to be available to the components of the application. The products of the various project types are automatically placed in the Libraries or Modules folder as appropriate. For example, the JAR file containing the XMLBeans Java classes derived form the XML Schemas in a Schema Project is automatically placed in the Libraries folder. However, you may also copy additional JAR files directly into the Libraries or Modules folders to make them available to the rest of your application.

The Libraries and Modules folders both place their contents on the application's Java class path. The main difference between the Libraries and Modules folders is that the contents of the Modules folder are automatically deployed to the appropriate instance of WebLogic Server. For example, when you build an EJB Project, the EJBs in the project are compiled and placed in a JAR file, which is then copied to the Modules folder and thereby automatically deployed to the server.

The Libraries folder is equivalent to the APP−INF/lib directory of a J2EE Enterprise Application.

Security Roles

A WebLogic Workshop application also may define security roles, which are a facet of the J2EE security framework based on *security roles*, *principals* and *groups*.

A human resources application might define the following categories of users: an ***administrator*** may perform any operation, including configuring the application itself; a ***manager*** may perform HR operations on employees (add, delete, adjust compensation, etc.); and an ***employee*** may access a subset of his or her own HR records. Each of these user categories is called a security role, an abstract grouping of users that is defined by the designers of the application. When the application is deployed, an administrator will map the roles to actual security identities in the production environment.

A security role is an application−specific grouping of users, classified by common traits such as job title. When an application is deployed, roles are mapped to security identities, such as *principals* (identities

assigned to users as a result of authentication) or *groups*, in the production environment. Based on this, a user has access rights to specific components and/or functions of an application based on the security role(s) to which the user's identify is mapped. The link is the actual name of the security role that is being referenced.

Following this model of *role−based security*, application components may be configured to allow or restrict access based on security roles. The application components do not (and *should not*) concern themselves with specific user or group identities. By abstracting security to roles, the actual configuration of an application's security settings can occur at runtime and requires no changes to the application code. In fact, J2EE allows configuration of security to a fine level of detail purely via declarative means by using files called deployment descriptors, so the application code doesn't even have to be aware of the actual security roles that are defined.

WebLogic Workshop allows you to define the security roles that will be used in your application. When your application is deployed, the security roles you have defined are deployed with it. If you have defined users and groups for test purposes in your development environment, those definitions are not deployed with your application. This eases testing of security configurations in the development environment but avoids the risk of leaving security roles in your application when it is deployed.

Cleaning Applications and Projects

Sometimes when you build an application or project, WebLogic Workshop does not update all of the appropriate build files. As a result, stale artifacts can exist between builds. This could happen for any number of reasons like moving or deleting files while WebLogic Workshop is closed. If you believe that any of your projects are exhibiting strange behavior, you can use the Clean utility to ensure that WebLogic Workshop removes all outdated build files and references. After cleaning, you can re−build the application or individual project and be sure that each build file is fresh.

To clean an application:

1. From the **Application** tab in WebLogic Workshop, right−click the application folder and select **Clean Application**.

To clean a project:

1. From the **Application** tab in WebLogic Workshop, right−click on the project folder representing the project you would like to clean and select **Clean<ProjectName>**.

Related Topics

Integrating with Source Control Systems

WebLogic Workshop File Types

This topic lists the file types you will encounter in your use of WebLogic Workshop.

WebLogic Workshop Application Developer Edition File Types

You may use a variety of files to create your application in WebLogic Workshop, some of which you may not be familiar with. The key file types you may encounter in WebLogic Workshop Application Developer Edition are:

- ***EJB*** file, or Enterprise Java Bean. An EJB file contains the Java implementation class for an EJB, with Javadoc annotations that configure the EJB.

  To learn more about building EJBs in WebLogic Workshop, see Developing Enterprise Java Beans.

- ***JCS*** file, or Java Control Source. A JCS file contains the Java implementation class for a Java control type. There is only one JCS file per Java control. If the Java control is extensible, there may be many JCX files that extend the Java control.

  To learn more about Java controls, see Working with Java Controls.
- ***JCX*** file, or Java Control eXtension. A JCX file is a local extension or customization of a Java control. For example, the Database control is defined once in a JCS file, but a local JCX file in an individual project defines the data source and operations for that particular instance of the control.
  To learn more about Java controls, see Working with Java Controls.
- ***JPF*** file, or Java Page Flow. A JPF file contains the Java implementation class for a page flow, together with Javadoc annotations that configure and control the behavior of a web application. A page flow is a controller and a collection of JSPs. The controller coordinates a user's course through the JSPs depending on changes in state as the user progresses. Page flows also enable you to bind application data to user interface components in the JSPs and to access application logic and data via Java controls.
  To learn more about Java page flows, see Developing Web Applications.
- ***JSP*** file, or Java Server Pages. The JSP file type is defined by the J2EE Specification. WebLogic Workshop defines custom JSP tag libraries that allow JSP files to reference Java controls and page flow actions. A related file type is the ***JSPF*** file, which stands for Java Server Page Fragment. JSPF files are used to hold snippets of JSP code that can be included in other JSP files. There are many sample JSP files in the WebApp project of the SamplesApp sample application installed with WebLogic Workshop.
  To learn more about JSP files, see Developing Web Applications.
- ***JSX*** file, or JavaScript with Extensions. A JSX file can contain ECMAScript (formerly called JavaScript) for manipulating XML. The functions in the JSX file are called from with XML Maps in a web service. WebLogic Workshop provides an extended ECMAScript language with support for XML as a native type, making XML processing in script very straightforward.
  To learn more about JSX files, see Getting Started with Script for XML Mapping.
- ***JWS*** file, or Java Web Service. A JWS file contains the Java implementation class for a web service, with Javadoc annotations that enable specific web service features. There are many sample JWS files in the WebServices project of the SamplesApp sample application installed with WebLogic Workshop.
  To learn more about Java web services, see Building Web Services.
- ***WSDL***, or Web Services Definition Language. WSDL files describe the interface of a web service to consumers of the web service. WebLogic Workshop can easily generate WSDL files for your web

services, and can consume WSDL files for external web services so that you may access them from your WebLogic Workshop applications.

- ***XMLMAP*** files. XML map files describe how XML should be mapped to Java, and vice versa, for a web service.
  To learn more about XML Maps, see Handling and Shaping XML Messages with XML Maps.
- ***XQ*** files, also known as XQuery maps, contain queries written in the XQuery language. These queries contain transformations that convert data between XML, non−XML, Java classes, and Java primitive data sources. You can generate these queries using the provided mapper and use these queries to create business process and web service transformations.

  To learn more about using XQuery, see Selecting XML with XQuery and XPath.
- ***XML*** files, or Extensible Markup Language files contain XML data that you can use as input to transformations.
- ***XSD*** files, or XML Schema Definition files contain a schema that describes XML data. Importing an XSD file into a WebLogic Workshop application allows you to use imported XML data types in transformations.
- ***CTRL*** files, or control files (***deprecated***). In WebLogic Workshop 7.0, control extensions were defined in CTRL files. CTRL files have been deprecated but are still supported in WebLogic Workshop 8.1. The functionality formerly provided by CTRL files is now provided by JCX files.

WebLogic Workshop Platform Edition File Types

In WebLogic Workshop Platform Edition, you may encounter the following additional file types:

WebLogic Integration File Types

- ***CHANNEL*** file. The Message Broker provides typed channels to which messages can be published and to which services can subscribe to receive messages. A message broker channel has similar properties to a Java Message Service (JMS), but is optimized for WebLogic Integration processes, controls, and event generators. Channel files define the Message Broker channels in an application. To be visible to other application components, channel files must be placed in a Schemas project in your application.
  To learn more about Message Broker channels, see Publishing and Subscribing to Channels.
- ***DTF*** file, or Data Transformation Format. A DTF file references reusable data transformation methods which convert data from one format to another. For example, XML data can be transformed from XML data valid against one XML Schema to XML data valid against a different XML Schema. Sample DTF files are available in the following applications: Tutorial: Process Application and New Process Application. (For example, if you create an application based on the Tutorial: Process Application, the TutorialJoin.dtf is available in the application.) These applications are available from File−>New−>Application in the WebLogic Workshop menu bar.
  To learn more about data transformations, see Guide to Data Transformation.

  For a tutorial on building data transformations, see Tutorial: Building Your First Data Transformation.
- ***JPD*** file, or Process Definition for Java. A JPD file contains the Java implementation class for a WebLogic Integration *business process*, with special annotations that configure the business process. Sample business processes are available in the following applications: Tutorial: Hello World Process Application, Tutorial: Process Application, New Process Application. These applications are available from File−>New−>Application in the WebLogic Workshop menu bar.
  To learn more about business processes, see Guide to Building Business Processes.

For a tutorial on building building business processes, see Tutorial: Building Your First Business Process.

- ***MFL*** file, or Message Format Language describes and constrains the content of non−XML data. An MFL file defines a schema for non−XML data. You can use the the Format Builder to create MFL files at design−time. Importing an MFL file into a WebLogic Workshop application allows you to use the imported non−XML data types (defined by the MFL) file in transformations.
  To learn more about working with MFL data, see Assigning MFL Data to XML Variables and XML Data to MFL Variables.
- ***XSL*** file. This is basically an XSLT file with an XSL extension. XSL stands for Extensible Stylesheet Language. This language is defined by the W3C that supports the use of stylesheets for the conversion of XML data. When a Transformation method of type XSLT is invoked, the XSLT processor invokes the transformations defined in the associated XSLT file.

WebLogic Portal File Types

- ***CAM*** file. Campaigns provide a container for executing scenarios to offer personalized content to users.
- ***EVT*** file. Event property sets are used to define the events available for personalization services.
- ***PLA*** file. Placeholders are used to display targeted media to users. In addition, event and behavior data can be tracked via event services.
- ***PORTAL*** file. A portal is an aggregation of applications and information in a common, coherent user interface.
- ***PORTLET*** file. A portlet provides a user interface to applications and information.
- ***REG*** file. Request property sets are used to define the attributes available in the HTTP request
- ***SEG*** file. User segments represent a business rule to classify users based upon their profile, request, session attributes as well as date and time conditions.
- ***SES*** file. Session property sets are used to define the attributes available in the HTTP session.
- ***SET*** file. Content selectors are a business rule used to retrieve content based upon user profile, request, session attributes as well as date and time conditions.
- ***USR*** file. User Profile property sets are used to define the attributes available in a user’s profile.

To learn more about portals, see WebLogic Portal Overview.

Related Topics

None.

Debugging Your Application

You can use the WebLogic Workshop integrated debugger to debug your application. The debugger allows you to set breakpoints, step through your code line−by−line, view local variables, set watches on variables, and view the call stack and exception information.

Workshop Test Browser

In order to debug an application, you must have a way to exercise the application as a real client would. WebLogic Workshop includes a Test Browser in which you may test Workshop web applications and web servcies. When you run a web application or web service, the Test Browser automatically loads Test View, a tool for exercising the application.

Testing with Debugging

To start the debugger, click the Start button on the toolbar or press Ctrl−F5. To pause debugging, click the Pause button on the toolbar. To stop debugging, click the Stop button on the toolbar.

When the debugger is started, the WebLogic Workshop Debugger command window is opened for the debugger proxy. This window must remain open in order to use the debugger. If you close this window, your breakpoints will not be hit when you test your application.

While your application is running in the debugger, it is unavailable to clients other than the Test Browser.

Breakpoints

To clear all breakpoints at once including exception breakpoints in the project, click the Clear All Breakpoints button on the toolbar, or press Ctrl−F9.

You can also disable breakpoints, which prevents the debugger from stopping at them, but doesn't delete them from, so that they're easy to re−enable. To disable all breakpoints, select Disable All Breakpoints from the Debug menu or press Ctrl−Shift−B. To enable all breakpoints, select Enable All Breakpoints from the Debug menu or press Ctrl−B.

Line Breakpoints

You can set a breakpoint in your code to halt execution on that line, immediately before it is executed. You can then examine variable values and/or the call stack or set other break points.

To set or remove a breakpoint, go to Source View and put the cursor on the line on which you want to break. Click the Toggle Breakpoint button on the toolbar. You can also toggle a breakpoint by pressing F9, by clicking in the gray edge to the left of the line numbers in Source View, or by right−clicking on the line of source code and selecting Toggle Breakpoint from the context menu that appears.

Exception Breakpoints

You can also set breakpoints that are triggered when exceptions occur. To set an exception breakpoint, select ***Debug***−−>***Create Breakpoint***, in the ***Create Breakpoint*** dialog, select the "Exception" radio button, and provide the fully qualified classname of the exception. The debugger will hit the breakpoint whenever an

exception of that class or one of its subclasses is thrown. You can determine whether the breakpoint always needs to be hit when an exception occurs, or whether the breakpoint only needs to be hit when the exception is either caught or not caught. Exception breakpoints are not set for a specific line of code, so your code in not marked with a red circle and highlighting when you create one.

Method Breakpoints

Method breakpoints allow you to set a breakpoint that will be hit as soon as a particular method is called. To create a method breakpoint, select ***Debug−−>Create Breakpoint***, in the ***Create Breakpoint*** dialog, select the "Method" radio button. Enter the fully qualified class name that contains the method, and the method's name. If the method is overloaded, you may also include the argument list for the specific version on which you'd like to set a breakpoint. If the method is overloaded and you do not specify the argument list, method breakpoints will be set in all of the overloaded versions.

Taglib and JSP Breakpoints

When debugging JSP files, you may also right−click on a taglib tag in the source editor and choose ***Taglib Breakpoints*** from the context menu. This brings up a dialog that shows the implementing class for the taglib and allows you to set breakpoints on all of the taglib methods it implements.

Additionally, when setting breakpoints in a JSP file, the breakpoint icon will have a black X over it if the line is not a valid line for a breakpoint. You can still set the breakpoint, but you will not hit it when running the JSP. This feature is not available for other file types.

Debugging Properties

There are several options that can be set for debugging on a project basis. To change them, pull up the project properties by either right−clicking the project folder in the ***Application*** tab and selecting ***Properties*** from the context menu, or by going to ***Tools−−>Project Properties−−>[Project_Name]***. Then select ***Debugger*** in the tree on the left.

Smart debugging: Filters out classes that are part of the WebLogic Platform so that when you step, you stay in your own code. For example, if you step into a method that's part of WebLogic Server and that method eventually calls your own code, the debugger will step directly into your code. For web application projects, this should almost always be left on.

Build before debugging: For web applications, this property specifies that a build should be executed on the current file before running. For Java projects, the entire project is built before running.

Pause all threads after stepping: Controls whether all threads are visible after stepping in the debugger. All threads are always suspended after hitting a breakpoint, but by default only the thread in which you are stepping is visible. The default behavior provides better performance, but you cannot see the other threads in the threads window. If you choose not to suspend all threads after stepping, you can still view them on demand using the Pause command.

The following options apply only to Java and Control projects. For detailed information see How Do I: Debug a Java Project?

Create new process

Starts a Java Virtual Machine for the application when you click on Start.

Main class: The name of the class with the main method that should be run.

Parameters: Arguments that should be passed to the main method.

VM parameters: Arguments that should be passed to the virtual machine when it starts. These include −D and −X VM options.

Home directory: The directory in which to start the Java application.

Application classpath: Classpath the virtual machine should use.

Automatically append Library JARs: In addition to the classpath specified above, add all the JARs that are shown in the Libraries folder in the Application window.

Automatically append server classpath: In addition to the classpath specified above, add all the JARs that are shown "Default server classpath" list in ***Tools***−−>***WebLogic Server***−−>***Server Properties***.

Attach to process

Attaches to a Java Virtual Machine that is already running and is configured to accept a debugger when you click on Start.

Socket: Connects to the virtual machine over the TCP/IP connector. You must specify the port on which the VM is listening. If blank, the server defaults to localhost.

Shared memory: Connects to the virtual machine over the shared memory connector. You must specify the memory address on which the VM is listening.

Using the Debugging Commands

Once execution is halted on a breakpoint, you can use one of the following commands to continue executing your code with the debugger. All of these commands are available on the toolbar and on the Debug menu.

- ***Step Into:*** The Step Into command continues execution line−by−line, beginning with the next line of code. Use this command if you want to debug your code one line at a time.
- ***Step Over***: The Step Over command executes a method call without debugging that method, and halts execution on the next line. Use this command if you know that the code in a method works and you don't need to step into it.
- ***Step Out:*** The Step Out command finishes executing a method and returns execution to the procedure that called it, halting on the line immediately following the method call. Use this command if you have stepped into a method and you don't want to continue stepping all the way through it.
- ***Continue***: The Continue command resumes execution until another breakpoint is encountered or the procedure has completed.
- ***Export Threads***: The Export Threads command, found in the Debug menu, creates and opens a text file that contains the call stacks of all of the threads that are running. This is a convenient way to save or share the current threads in the event of a deadlock or other threading problem. The command is only active when the debugger has thread information.

Note that if you step into a line that contains more than one statement, all of the statements on that line will be executed when you step to the next line.

Using the Debug Windows

The debug windows provide information about values and conditions in your code while you're running it in the debugger.

To view one of the debug windows, choose Debug Windows from the View menu, and select the desired window.

The Locals Window

The Locals window automatically shows variables that are in scope as you execute your code in the debugger. You can modify the value of a primitive−type variable here while you are debugging.

You can expand entries for objects in the Locals window to view their members.

Some variable types have multiple views available in the Locals window. A small list of examples include:

- ***int***: May be viewed and edited as a decimal integer or hexidecimal integer.
- ***char***: May be viewed and edited as a character, Unicode escape, decimal integer, or hexidecimal integer.
- ***java.lang.StringBuffer***: May be viewed showing just its contents as a String, its length, and its capacity, or with the default view showing its internal data structures.
- ***javax.servlet.http.HttpServletRequest***: May be viewed showing attributes and other fields as simple name−value pairs, or with the default view showing its internal data structures.

To determine what other views are available and to switch to a different view, simply right−click on either the variable's name or value in the Locals window and look in the View As context menu. Note that the default view of internal data structures will be the only view available for some types.

The Watch Window

You can use the Watch window to observe how the value of a particular variable changes as you're debugging. To set a watch, simply type the name of the variable into the Watch window.

Static classes must be fully qualified to be viewed in the Watch window.

The Stream Window

The Stream Window is only active when doing JSP debugging and shows the output stream that the JSP is generating to send back to the Browser

The Immediate Window

The Immediate window allows you to execute arbitrary code while debugging, with the following restrictions:

1. You cannot create new classes, Interfaces, or methods, or instantiate new object instances.

2. You cannot use the instanceof operator.
3. You cannot use .class syntax, such as Class c = java.lang.Object.class.
4. The value shown after clicking ***Evaluate*** will be the result of the last statement in the text field.
5. Static classes must be fully qualified.
6. If an object has a member variable and method with the same name, the variable is accessible but the method cannot be called.

The Call Stack Window

The call stack shows the methods that have been called in order to reach the point at which execution is halted in the debugger. The methods are listed from most recently called to first called. You can also think in terms of each call being nested within the method below it on the call stack.

You can click on a method in the call stack to navigate to that method in Source View. This will also show the local variables available for that call stack, and reevaluate any watches within selected stack frame as well.

The Threads Window

The Threads window shows information about the currently executing threads. You can click on stack frames to navigate to that method and refresh the Local and Watch windows in that stack frame. If you switch the selected thread and then use any of the step commands, you will step in that thread.

The Breakpoints Window

The Breakpoints window lists the breakpoints that are currently set in your code by line number as well as the exception and method breakpoints. You can specify conditions under which each breakpoint should be hit in the Conditions column. You may also enable and disable individual breakpoints by using the checkboxes.

Remote Debugging

If you're developing against a remote server, you can also debug against that remote server. Breakpoints and other debugging information will be stored on the local machine. For more information on pointing to a remote server, see How Do I: Use a Remote WebLogic Server During Development?

You can also debug on a managed server. To do this, set the clustered servers up using a proxy or set them all pointing to one of the managed servers.

Run all of the managed servers except one, and start the admin server with the nodebug option specified. This will prevent port collisions on the debugging port.

Do all of your debugging against the managed server.

Debugging JSP Pages

Be aware that browser caching is always turned off when you are debugging. To turn caching on while you are debugging, place the following scriptlet in your JSP pages.

```
<%
    response.setHeader( "Cache-Control", null );
%>
```

Note that browser caching is the default behavior when your web application is deployed to a production server. You do not need to take special measures, such as the scriptlet above, to turn on caching on a production server.

Debugging EJBs

The debugger supports debugging EJBs regardless of whether they have been developed within WebLogic Workshop. They can be debugged locally or remotely.

Use the following procedure to attach the debugger to an EJB:

1. Open the source in Workshop
2. From the Tools menu choose Project Properties
3. In the Debugger tab, choose "Attach to Process"
4. Enter the debug port number (8453 by default) and server name or IP address.d
5. Click Play
6. Set breakpoints as needed and start debugging.

1.

Debugging on a non−Workshop enabled Server

To debug a project on a server not configured as a Workshop Server, open the Project Properties dialog box for the project you want to debug. On the Debugger tab, choose the "Non Workshop Server" radio button, and enter the debugging port number (8453 by default), the Http port number, and the name or IP address of the remote server.

Debugging Unrunnable Files

Some files cannot be directly run by WebLogic Workshop, e.g., servlets, custom Java controls, JAVA files, etc., nevertheless you can still set and hit breakpoint in these files. For example, to debug a servlet, set breakpoints in the servlet source code, add a JSP page to your project, run that JSP page (this will start the debugging process), and then enter the servlet's URL directly into the browser's address bar. You will now be able to halt at breakpoints in the servlet code.

To debug custom Java controls or JAVA files, set breakpoints in the source code of those files, and then run a web service or some other runnable file that calls into the Java control or JAVA file.

Starting Two Debugging Proxies in the Same Domain

If you start two server instances in the same domain (for example, using the start up script [BEA_HOME]/weblogic81/samples/domains/portal/startWebLogic.cmd), the second server will fail with a TransportException.

```
ERROR: Proxy already running...
weblogic.debugging.comm.TransportException
at weblogic.debugging.comm.ServerConnectInfo.createTransport()Lweblogic.
debugging.comm.CommTransport;(ServerConnectInfo.java:63)
```

The failure is caused by the debugging proxy, which attempts to open the same server socket port already used by the first server.

To start the second server running on a different port, send a port number parameter to (1) the WebLogic Workshop IDE, (2) the server, and (3) the debugger.

(1) To Set the Port Number on the WebLogic Workshop IDE

Add the following parameter to [BEA_HOME]\weblogic81\workshop\workshop.cfg.

−Dworkshop.debugProxyPort=*PORT_NUMBER*

(2) To Set the Port Number on the Server

Add the following parameter to the server startup script (for example, [BEA_HOME]\weblogic81\samples\domains\workshop\startWebLogic.cmd/.sh).

−Dworkshop.debugProxyPort=*PORT_NUMBER*

(3) To Set the Port Number on the Debugger Process

Add the following parameter to the server startup script (for example, [BEA_HOME]\weblogic81\samples\domains\workshop\startWebLogic.cmd/.sh).

−serverport=*PORT_NUMBER*

Related Topics

How Do I: Debug a Java Project?

How Do I: Debug an Application?

How Do I: Debug an Application on a Remote Production Server?

Testing Your Application with Test View

The Test View page is loaded in your web browser when you build and run a web service or page flow from WebLogic Workshop. You can use Test View to test your component.

Testing Page Flows

If you are testing a page flow, Test View functions like a web browser. You can navigate through all of the pages that make up your page flow and test the functionality of individual pages and the flow of data across pages.

Note that the first time you try to run a JSP page by clicking on the Start button, WebLogic Workshop displays a message saying that the corresponding page flow will be run instead. You can test just an individual JSP page by right−clicking on the JSP page and choosing **Run JSP Page**.

Testing Web Services

Test View has four tabs that provide information about a web service:

- Overview
- Console
- Test Form
- Test XML

By default, the Test Form tab is selected when you load Test View by clicking on the Start button.

In addition, each page lists the URL of the web service being tested. Each segment of the URL is a link to the WebLogic Workshop Directory at that level of the application. The WebLogic Workshop Directory is a page that lists the files in each level of the project directory. You may access it directly at any level via the URL http://<server>:<port>/AppName[/folder]/wlwdir.

Overview Tab

The Overview tab displays public information about the web service, including:

- A link to the WSDL that describes the public contract for the web service. Clients of the web service use the WSDL file to determine how to call the web service and what data will be returned. The WSDL describes both methods and callbacks on the web service.
- The callback WSDL for clients that cannot handle the callbacks described in the complete WSDL. Clients wishing to receive callbacks by implementing a callback listening service can communicate with the web service using this WSDL.
- The Java source code for a WebLogic Workshop service control for the web service. A developer using WebLogic Workshop who wishes to call your web service from their WebLogic Workshop application can use this source to construct a Service control.
- The Java proxy for calling the web service from a Java client. A Java client can call your web service by creating a class from the Java proxy and invoking its methods.
- The description of the web service, which lists the web service's methods and callbacks.
- Links to useful information such as specifications for WSDL and SOAP.

Note: To learn how the comments associated with each method are obtained, and how you can document your web service's methods, see Documenting Web Services.

Console Tab

The Console tab displays private information about your service, including:

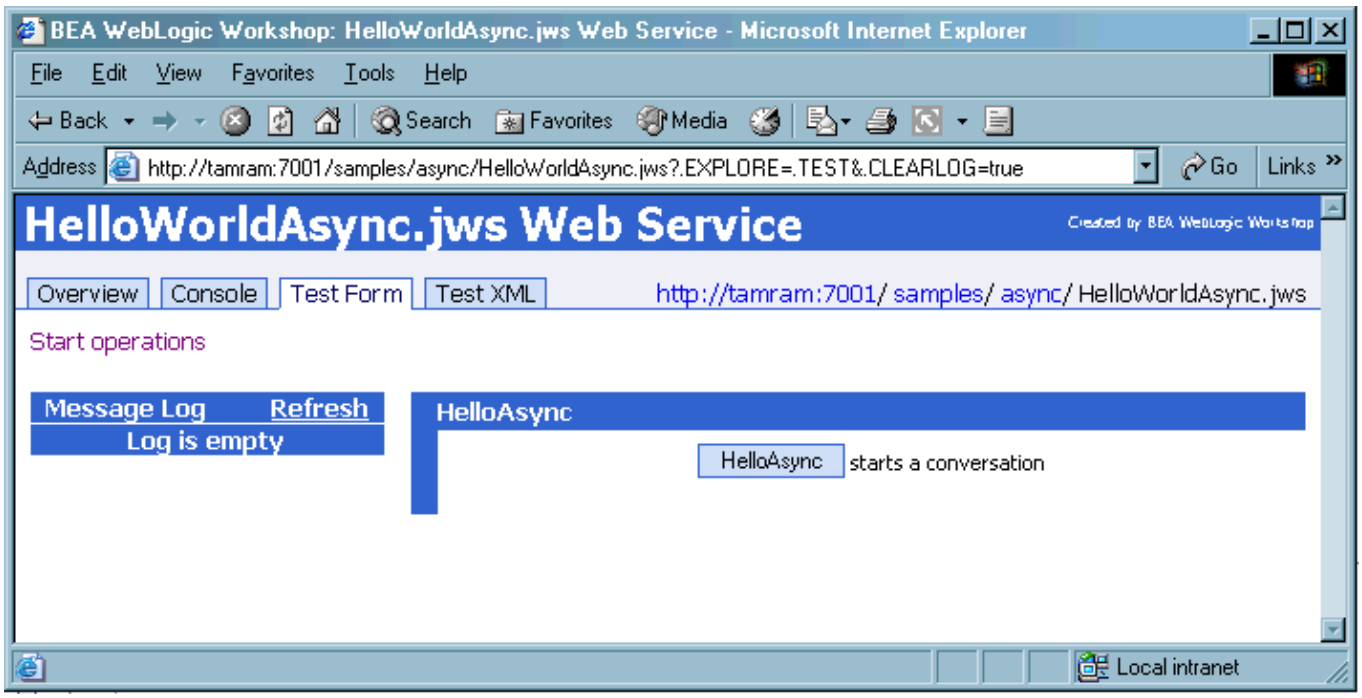
- How the service is implemented on the back end, and the version of WebLogic Workshop with which it was created. For example, web services built using WebLogic Workshop are implemented using Enterprise JavaBeans; they may also use other J2EE components.
- Links to the WebLogic Server console application.
- Settings for the message log on the Test Form tab.
- Persistence settings, for clearing the conversational state and logs for the service, and for redeploying the Enterprise JavaBeans underlying the service.

Test Form Tab

The Test Form tab provides a simple test environment for the public methods of the web service. You can provide parameters for a method and examine its return value. You can also track and test the different parts of a conversation.

Note: The Test Form tab may only be used for web services (or methods of web services) that support HTTP−GET. Web service methods that do not support GET or that receive raw XML messages must be exercises via the Test XML tab.

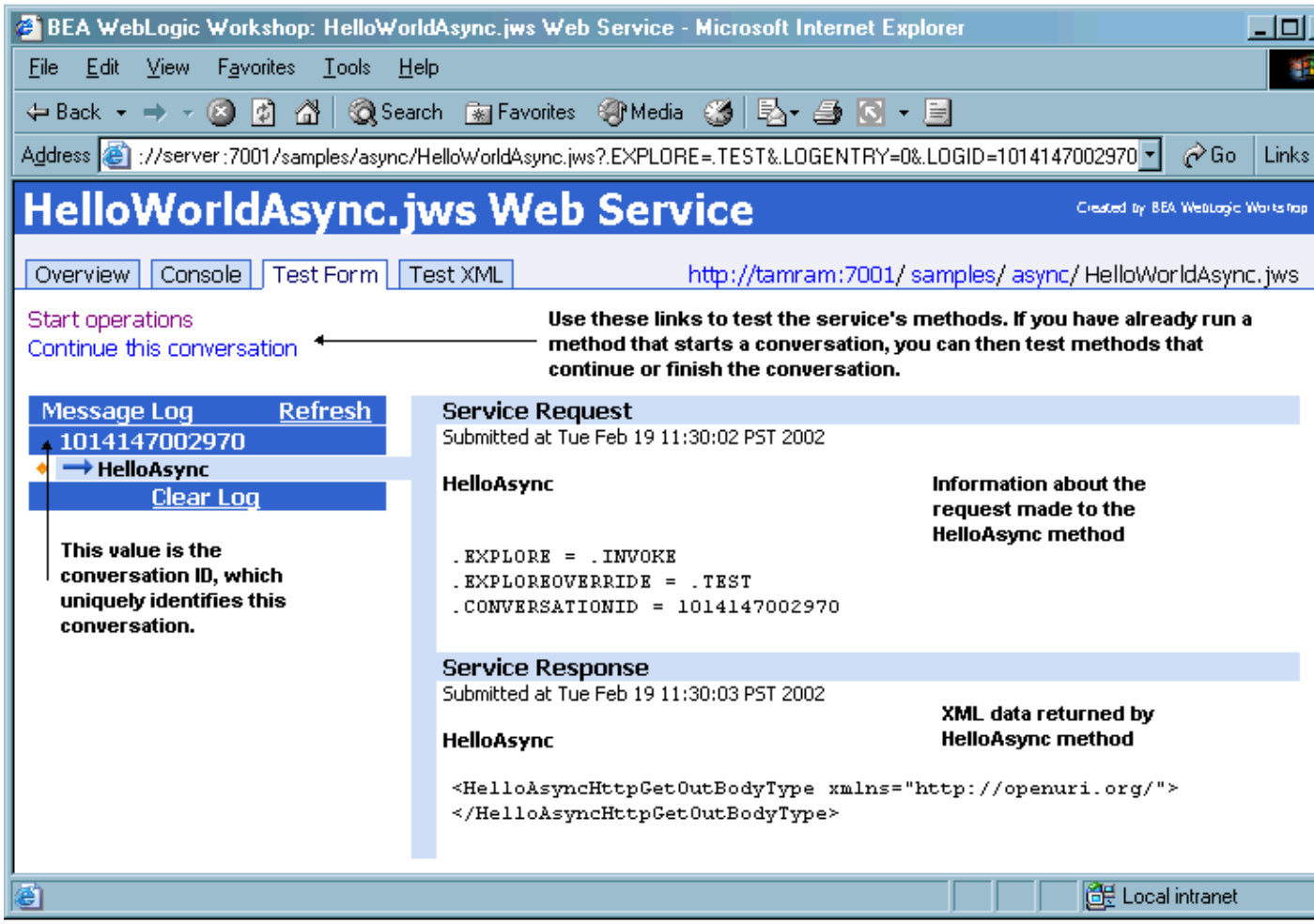
The following image shows how the Test Form tab appears for a service called HelloWorldAsync, which demonstrates a simple conversation:



To test the service, click the HelloAsync button. If this method took parameters, you would enter values for

them here.

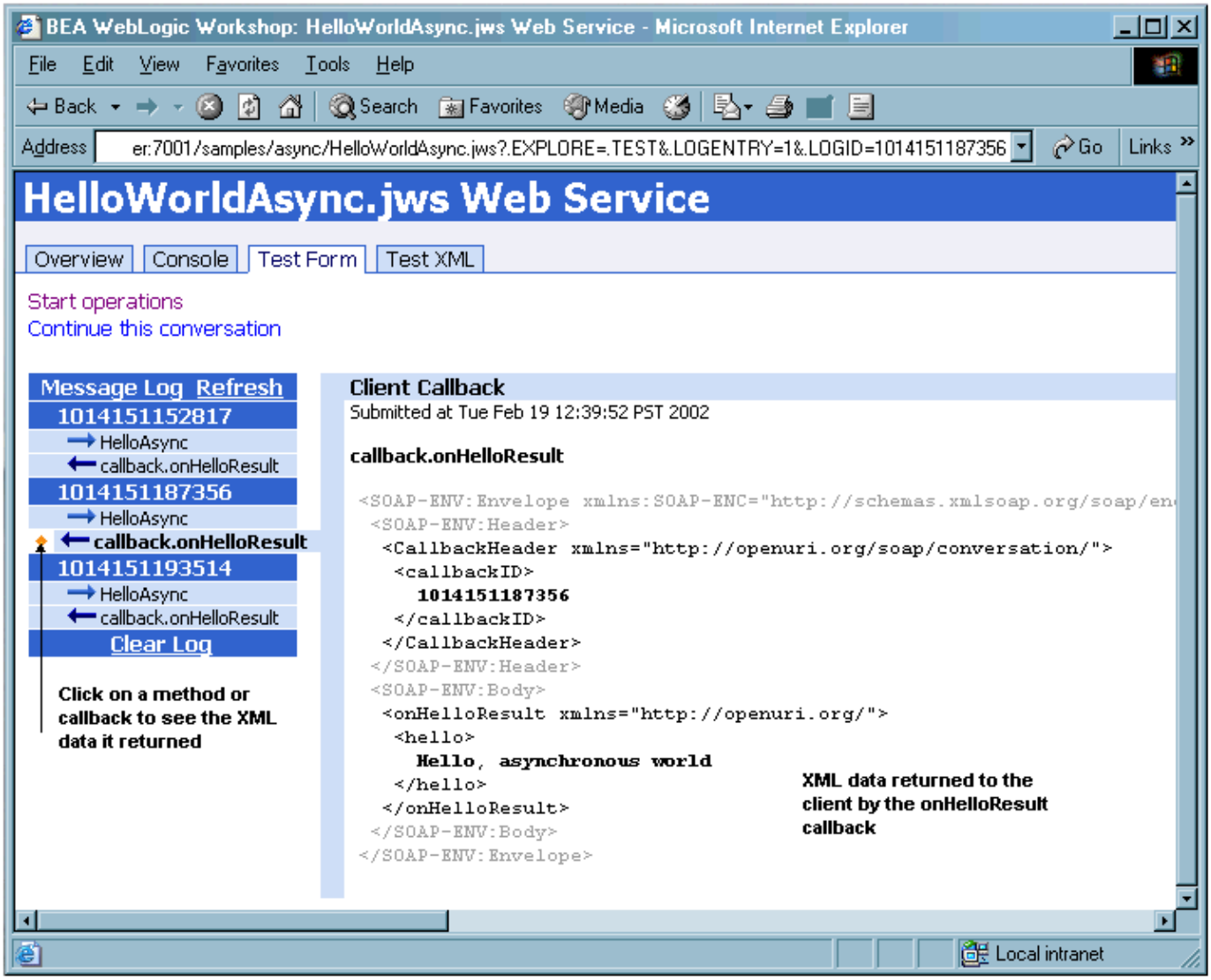
The Test Form page displays information about the service request and response, including the XML message that was returned, as shown in the following image:



If the web service implements conversations, as the HelloWorldAsync service does, you can use Test View to test the methods that start, continue, or finish a conversation and the callbacks that continue or finish a conversation. You can also test multiple conversations at once.

The conversation ID that appears in the message log uniquely identifies each conversation that is underway. Click on this value to select and work with this conversation. You can view the results for each method or callback that has participated in this conversation by clicking on its name in the list. Click the Refresh link to refresh the message log.

The following image shows Test View with multiple conversations underway:



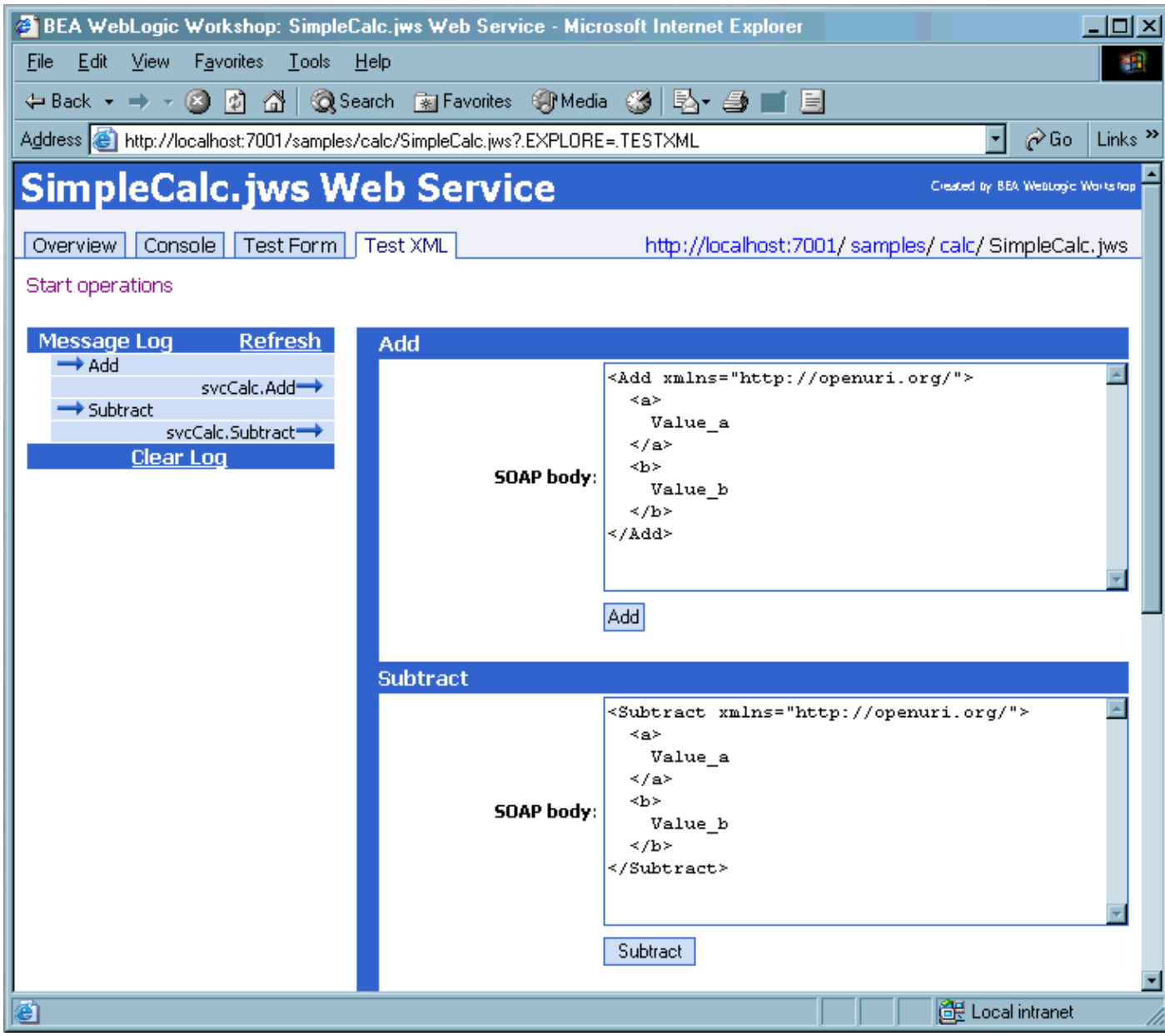
Note: To learn how the comments associated with each method are obtained, and how you can document your web service's methods, see Documenting Web Services.

Test XML Tab

The Test XML tab shows the XML data that is being sent to your service when you test its methods. You can use this page to examine and modify the XML data that is passed to a method of your service.

If your method takes parameters of a data type other than String, you must modify the parameter placeholders in the SOAP body before you click the button to call the method. For example, if your method expects an integer, you must supply a valid integer in the SOAP body.

The following image shows the Test XML tab for a service called SimpleCalc, which performs simple arithmetic calculations:



WebLogic Workshop Directory

The WebLogic Workshop Directory contains a sortable list of the files that comprise the different levels of the Project directory. You can use this page to select another service to test, test a control file, and clean up the service.

There are two ways to navigate to the service directory:

- Click on any of the links that make up the service address listed on the upper right−hand corner of the Test View pages. Each part of the path is a link to the directory listing of the files stored in this part of the project directory.
- Click on the See other services in your project link in the upper right−hand corner of the Overview

page.

You can use the WebLogic Workshop Directory to:

- See the complete list of files and folders stored at this level.
- Use filters to show files of a particular type such as JWS, Control, XML, ECMAScript, and WSDL files.
- Click on a JWS file name and test it.
- Click on a JCX file, see information about this file, and choose to generate a Test JWS for testing the JCX.
- Use the Clean All command to delete all generated .java, .class, and directories; erase cached dependency graph information, get rid of all conversation instances, undeploy all beans, and clear logs while leaving your application sources intact.

Related Topics

Web Service Development Cycle

Java and XML Basics

What if you don't know Java and XML? Fortunately, you don't need to be an expert in either to build applications in WebLogic Workshop. If you have some prior programming experience in another programming language, you should be able to learn the basics of Java quickly. And XML, although it may look complex, is not even a programming language—it's just a structured way to describe data.

This section provides a brief overview of Java and XML to get you started. If you need more, there are a number of excellent books available on both Java and XML.

Topics Included in this Section

Introduction to Java

Learn the basics of programming in Java.

Introduction to XML

Explore the parts of an XML document.

Introduction to Java

This section outlines the basic rules of Java syntax and gives a brief introduction to Java concepts useful for the WebLogic Workshop developer. If you are familiar with another high−level programming language, such as C, C++, Visual Basic, or even JavaScript, you should have no trouble learning Java quickly. If you want a more detailed and thorough introduction to Java, there are a number of excellent books and courses available to you.

Basic Java Syntax

The following sections cover some some of the basics of Java syntax.

Primitive Types and Wrapper Classes

In Java, you must declare a variable before you use it, providing both its name and type. While this may seem like more work up front than simply using the variable when you need it, the extra bit of effort pays off, because the compiler can check and enforce type restrictions. This can save you debugging time later on.

For example, you can declare a new variable to store an integer as follows:

```
int x;
```

The following declares a new variable to hold a text string:

```
String strName;
```

After you've declared the variable, you can assign a value to it:

```
x = 5;
```

or

```
strName = "Carl";
```

The prefix *str* in the variable above makes it easy for the developer to remember that it is a variable of type String.

Java provides several primitive types that handle most kinds of basic data. The primitive types are:

- boolean
- byte
- char
- double
- float
- int
- long
- short

In Java, you use primitive types to handle basic values and Java objects to store more complex data. Every Java object is derived from a basic ***Object*** class. A class is a template for creating an object; you can think of a

class as a cookie cutter, and the objects created from it as the cookies.

It is important to keep in mind the difference between a primitive type and an object. An ***Integer*** object is different from an ***int***, even though both store an integer value. A method that accepts one as an argument will not accept the other unless it has been specifically defined to do so. Also, Java objects always have methods which you use to work with them; a primitive type cannot have a method.

Note that in Java, the memory size of the primitive types is well−defined. An ***int***, for example, is always a 32−bit value.

Java Tip: Many useful classes in Java that store and manage collections of objects (lists, trees, arrays, etc.) cannot operate on primitive types, because the primitive types are not derived from the basic Java ***Object*** class. However, Java includes wrapper classes for the primitive types, which you can use to temporarily create an object. You can then use that object in the desired operation. The wrapper classes have similar names to the primitive types, but with an uppercase first letter:

- Boolean
- Byte
- Char
- Double
- Float
- Integer
- Long
- Short

It is easy to create a wrapper class object from a primitive value, as shown below for Integer and int:

```
int myInt;
Integer myInteger = new Integer(myInt);
```

The primitive value can then be retrieved from the wrapper object as follows:

```
myInt = myInteger.intValue();
```

Statements

A statement is a logical line of code. A statement may assign a value to a variable or call a function. A statement is the basic building block of your Java program. For example, the following statement assigns the value *5* to the variable *x*:

```
x = 5;
```

This statement calls a function:

```
calculateIT(x);
```

As you may notice above, statements must end in a semicolon. Exceptions to this rule are statements that begin new blocks of code−–that is, sections of code enclosed in curly braces ({}).

Blocks

Curly braces ({}) are used to enclose blocks, as shown below:

```
for( i=0; i< 100; i++)
{
    <one or more statements>
}
```

The position of lines breaks is not important. The following example is logically equivalent to that above:

```
for(i=0; i<100; i++) { <one or more statements> }
```

However, most programmers would consider the latter example to be poor style since it hampers readability.

Blocks group a set of statements together. In the above example, using a block here indicates that all the statements in the block should be executed in turn as part of the for loop.

Special Operators

The following section discusses some of the operators you need to understand to write Java code.

The new Operator: Object Creation

To use an object in Java, you must follow a two−step process. First, you must declare an object variable, providing its name and type; second, you must create a new instance of the object's class. The second step creates a new object for you in memory and assigns it to the variable you've declared. You use the new keyword to create a new instance of a class, together with a class constructor.

A Java class always has one or more constructors, which are methods that return a new instance of the class. Every Java class provides a basic constructor method that takes no arguments. Consider the following example:

```
class MyClass
{
   int firstElement;
   float secondElement;
}
MyClass myClassInstance;         // at this point, myClassInstance is null
myClassInstance = new MyClass(); // now myClassInstance refers to an object
```

A Java class may also have constructor methods that take several arguments; these constructors are generally provided as a convenience for the programmer using the object. You decide which one to use based on how you want to store data in the object.

The . Operator: Object Member Access

The . (dot) operator is used to access information or to call an object's methods. For example, given the class definition and variable declaration above, the members of myClassInstance object can be accessed as follows:

```
myClassInstance.firstElement = 1;
myClassInstance.secondElement = 5.0f;
```

The [] Operator: Array Access

In Java, arrays are accessed by integer array indices. Indices for arrays begin with zero (0). Each element of an array can be accessed by its index using the [] (array) operator.

For example, to store an array of six names, you can declare an array as follows:

```
String strNames[5];
```

You can then set the value of first element in the array by assigning an appropriate value:

```
strNames[0] = "Carl";
```

Collections and Iterators

The following section explains collections and iterators.

Collections

Java defines a set of classes known as collection classes. These classes are defined in the java.util package (see more on packages below). The most commonly used collection classes are:

- ArrayList
- HashMap
- LinkedList

Objects created from these classes provide convenient management of sets of other objects. An advantage of a collection over an array is that you don’t need to know the eventual size of the collection in order to add objects to it. The disadvantage of a collection is that it is generally larger than an array of the same size. For example, you can create a LinkedList object to manage a set of objects of unknown size by saying:

```
LinkedList l = new LinkedList();

l.Add("Bob");

l.Add("Mary");

l.Add("Jane");

…
```

Note that in versions of Java before 1.2, the Vector and Hashtable classes were often used to manage collections of objects. These classes, however, are *synchronized*, meaning they are safe for multi−threaded use. Synchronization has performance implications. If you do not require synchronization behavior, you will achieve better performance by using the newer ArrayList and HashMap classes.

Iterators

Some collection classes provide built−in iterators to make it easy to traverse their contents. The built−in iterator is derived from the java.util.Iterator class. This class enables you to walk a collection of objects, operating on each object in turn. Remember when using an iterators that it contains a snapshot of the collection at the time the iterator was obtained. It’s best not to modify the contents of the collection while you

are iterating through it.

String and StringBuffer Classes

Java provides convenient string manipulation capabilities via the java.lang.String and java.lang.StringBuffer classes. One of the most common performance−impacting errors new Java programmers make is performing string manipulation operations on String objects instead of StringBuffer objects.

String objects are immutable, meaning their value cannot be changed once they are created. So operations like concatenation that appear to modify the String object actually create a new String object with the modified contents of the original String object. Performing many operations on String objects can become computationally expensive.

The StringBuffer class provides similar string manipulation methods to those offered by String, but the StringBuffer objects are mutable, meaning they can be modified in place.

Javadoc Comments

Java defines a mechanism for attaching information to program elements and allowing automatic extraction of that information. The mechanism, called Javadoc, was originally intended to attach specially formatted comments to classes, methods and variables so that documentation for the classes could be generated automatically. This is how the standard Java API documentation is produced.

Javadoc requires a special opening to Javadoc comments: they must start with /** instead of /*.

Java Tip: While the comment may contain arbitrary text before the first Javadoc annotation in the comment, all text after the first Javadoc annotation must be part of an annotation value. You cannot include free comment text after Javadoc annotations; doing so will cause compilation errors.

In addition to the special opening of the comment, Javadoc defines several Javadoc annotations that are used to annotate parts of the code. Javadoc annotations always start with the @ character. Examples of standard Javadoc annotations are:

- @param: The value is a description of a parameter to a method.
- @see: The value is a reference to another class, which will become a hyperlink in the documentation.
- @since: The value is the version number in which this class first appeared.

Although originally intended to produce documentation, Javadoc annotations have now found other uses, as they are a convenient and systematic way to attach information to program elements that may be automatically extracted.

WebLogic Workshop uses Javadoc annotations to associate special meaning with program elements. For example, the @common:control annotation on a member variable of the class in a JWS file tells WebLogic Workshop to treat the annotated member variable as a WebLogic Workshop control.

Exceptions

Java defines a common strategy for dealing with unexpected program conditions. An exception is a signal that something unexpected has occurred in the code. A method throws an exception when it encounters the unexpected condition. When you call a method that throws an exception, the Java compiler will force you to

handle the exception by placing the method call within a *try−catch* block (see below).

An exception is a Java object, which makes it easy to get information about the exception by calling its methods. Most Java exception objects inherit from the basic java.lang.Exception class. Specific exception classes contain information pertinent to the particular condition encountered. For example, a SQLException object provides information about a SQL error that occurred.

Try−Catch−Finally Blocks

You use *try−catch−finally* blocks to handle exceptions. The *try−catch−finally* block allows you to group error handling code in a single place, near to but not intermingled with the program logic.

The following example demonstrates a *try−catch−finally* block. It assumes that method doSomething() declares that it throws the BadThingHappenedException:

```
public void callingMethod()
{
   try
   {
      doSomething();
   }
   catch (BadThingHappenedException ex)
   {
      <examine exception and report or attempt recovery>
   }
   finally
   {
      <clean up any work that may have been accomplished in the try block>
   }
   return;
}
```

If doSomething() completes normally, program execution continues at the *finally* block.

If doSomething()throws the BadThingHappenedException, it will be caught by the *catch* block. Within the *catch* block, you can perform whatever action is necessary to deal with the unexpected condition. After the code in the catch block executes, program execution continues at the *finally* block.

Notice that the *finally* clause is always executed, whether or not an exception occurred in the *try* block. You can use the *finally* clause to clean up any partial work that needs to be performed, regardless of whether an error occurred. For example, if a file is opened in the *try* block, the *finally* block should include code to close the file, such that the file is closed regardless of whether or not an exception occurred.

Garbage Collection

Unlike its predecessor compiled languages like C and C++, Java is never compiled all the way to machine code that is specific to the processor architecture on which it is running. Java code is instead compiled into Java byte code, which is machine code for the Java Virtual Machine (JVM). This is the feature that gives Java its inherent portability across different operating systems.

Since all Java code is run in the JVM, the JVM can include capabilities that would be difficult to implement in a traditional programming language. One of the features the JVM provides is garbage collection.

The JVM keeps track of all references to each object that exists. When the last reference to an object is removed, the object is no longer of any use (since no one can reference it, no one can use it). Periodically, or when memory resources are running low, the JVM runs the garbage collector, which destroys all unreferenced objects and reclaims their memory.

For the programmer, this means that you don't have to keep track of memory allocations or remember when to free objects; the JVM does it for you. While Java includes the new keyword, which is analogous to the malloc function in C/C++ and the new operator in C++, Java does not include a free() function or a delete keyword. It is still possible to leak memory in Java, but it is less likely than in other high−level languages.

Packages

The following sections explain Java packages and their relationship to Java classes.

Declaring Packages

All Java code exists within what is know as the *package namespace*. The package namespace exists to alleviate conflicts that might occur when multiple entities are given the same name. If the entities exist in different portions of the namespace, they are unique and do not conflict.

A package statement may optionally be included at the top of every Java source file. If a file does not specifically declare a package, the contents of the file are assigned to the *default package*.

Every class and object has both a simple name and a fully qualified name. For example, the Java class String has the simple name String and fully qualified name java.lang.String, because the String class is declared in the java.lang package.

There is a relationship between the package names and the directory hierarchy within which the files exist. A file containing the statement

```
package security.application;
```

must reside in the directory security/application or the Java compiler will emit an error.

Note that packages allow the same class or object name to exist in multiple locations in the package hierarchy, as long as the fully qualified names are unique. This is how the standard Java packages can contain a List class in both the java.util and java.awt packages.

If a file includes a package statement, the package statement must be the first non−comment item in the file.

Importing Packages

Before you can refer to classes or objects in other packages, you must *import* the package or the specific entities within the package to which you want to refer. To import all classes and objects in the package from the previous example, include the following statement in your file:

```
import security.application.*;
```

Note that the following statement imports only the classes and objects that are declared to be in the security package; it does not import the security.application package or its contents.

```
import security.*;
```

If ***UserCredentials*** is a class within the security.application package and it is the only class or object in the package to which you want to refer, you may refer to it specifically with the following statement:

```
import security.application.UserCredentials;
```

If you are only importing a few classes from a large package, you should use the specific form instead of importing the entire package.

Import statements must be the first non−comment items in the file after the optional package statement.

Related Topics

Developing Applications with WebLogic Workshop

Introduction to XML

Extensible Markup Language (XML) is a way to describe structured data in text. Web services exchange messages in XML. Of the many roles XML plays in web services, perhaps the most visible to you as a developer is providing the format for the messages a web service sends and receives. A web service receives method calls, returns values, and sends notifications as XML messages. As you test and debug your service, for example, you might construct XML documents that represent sample messages your service could receive, then pass these documents to your service and view the results.

For more information about the XML standard, see Extensible Markup Language (XML) 1.0 (Second Edition) at the web site of the W3C. The W3C also provides a page with useful links to more information about XML at Extensible Markup Language (XML).

Note: WebLogic Workshop provides easy programmatic access to XML through the XMLBeans API. For more information, see Getting Started with XMLBeans.

Anatomy of an XML Document

Let's start with a sample XML document. Imagine that you are writing a web service to return information about employees from a database. Based on first and last name values sent to your service, your code searches the employee database and returns contact information for any matches. The following example is XML that might be returned by your service. Even if you have no experience with XML, you can probably see structure in the example.

```
<!-- Information about the employee record(s) returned from search. -->
<employees>
    <employee>
        <id ssn="123-45-6789"/>
        <name>
            <first_name>Gladys</first_name>
            <last_name>Cravits</last_name>
        </name>
        <title>Busybody</title>
        <address location="home">
            <street>1313 Mockingbird Lane</street>
            <city>Anytown</city>
            <state>IL</state>
            <zip>12345</zip>
        </address>
    </employee>
</employees>
```

Elements

In XML, text bracketed by < and > symbols is known as an element. Elements in this example are delimited by tags such as <employees> and </employees>, <name> and </name>, and so on. Note that most elements in this example have a start tag (beginning with <) and an end tag (beginning with </). Because it has no content the <id> element ends with a /> symbol. XML rules also allow empty elements to be expressed with start and end tags, like this: <id ssn="123-45-6789"></id>.

Root Elements

Every XML document has a root element that has no parent and contains the other elements. In this example, the <employees> element is the root. The name of the root element is generally based on the context of the document. For example, if your service is designed to return merely an address—an <address> element and its contents—then the root element is likely to be <address>.

Parent/Child Relationships

Elements that contain other elements are said to be parent elements; the elements that parent elements contain are known as child elements. In this example, <employees>, <employee>, <name>, and <address> are parent elements. Elements they contain—including <employee>, <id>, <name>, and <city>—are children. (Note that an element can be both a parent and a child.)

Note: This example uses indentation to accentuate the hierarchical relationship, but indentation is not necessary.

Attributes

Attributes are name/value pairs attached to an element (and appearing in its start tag), and intended to describe the element. The <id> element in this example has an ssn attribute whose value is 123−45−6789.

Content

The content in XML is the text between element tags. Content and attribute values represent the data described by an XML document. An element can also be empty. In the example above, "Gladys" is the content (or *value*) of the <first_name> element.

Comments

You can add comments to XML just as you would with HTML, Java, or other languages. In XML, comments are bracketed with <!−− and −−> symbols (the same symbols used in HTML). The first line of the example above is a comment.

Namespaces

A namespace in XML serves to define the scope in which tag names should be unique. This is important because XML's wide use and textual nature make it likely that you will see occasions where element names with different meanings occur in the same document. For example, namespaces and prefixes are used in XML maps, where a prefix differentiates the tags that are needed for mapping from those associated with the mapped method's XML message.

Related Topics

Getting Started with XMLBeans

Introduction to XQuery Maps

Managing the Build Process

The topics in this section discuss the WebLogic Workshop build environment and provides tips on how to optimize your build process.

Topics Included in This Section

Using the Build Process More Efficiently

Provides tips on how to build applications and projects efficiently. You can significantly reduce build times by making a few small configurations to the build environment and following a few of the best practice guidelines outlined in this document.

Customizing the Build Process

Introduces the scripts that WebLogic Workshop uses to perform application and project level builds. This topic describes how you can export, edit and deploy these scripts to create a custom build process.

How Do I ...?

Contains topics that guide you through the various tasks associated with managing the WebLogic Workshop build process.

Related Topics

Using Source Control with WebLogic Workshop

Deploying an Application to a Production Server

Using the Build Process More Efficiently

This topic presents best practice recommendations for using WebLogic Workshop build tools efficiently. These recommendations help you optimize the build process and save valuable time during iterative development cycles. This topic divides these recommendations into the following sections:

- Building Projects Instead of Applications
- Running Project Files Without Building
- Disabling JSP Pre−compilation
- Manually Compiling Schema Projects
- Managing WebLogic Workshop Memory

Building Projects Instead of Applications

If your application contains multiple projects, you can choose to build only the project you are working on to save time. To build a single project, simply right−click on the project folder and then click ***Build <ProjectName>***. Note that the Build button located in the WebLogic Workshop IDE toolbar always builds the entire application. You should build the entire application if you are changing files across multiple projects, especially if there may be dependencies between them. And you should always build the application prior to deploying it to a production environment.

Running Project Files Without Building

You can modify and test changes to JSP, Java Page Flow (JPF), and web service (JWS) files without having to compile an entire project. To do this:

1. Ensure that the project file you would like to test is displayed in the main area of the WebLogic Workshop IDE. You can ensure this by double−clicking the file displayed in the ***Application*** tab.
2. Click the start button located in the toolbar of the WebLogic Workshop IDE. The start button appears as follows:

You cannot test all file types using this approach. For example, to modify and test changes to Java control file, you must first build the entire control project.

Disabling JSP Pre−compilation

WebLogic Workshop automatically pre−compiles JSP pages in a web project. This enhances server−side execution of those pages when users request them in a production environment. However, this enhanced run−time execution can translate into slower performance during development. When you change any file in a web project, and then build, WebLogic Workshop compiles every JSP file in that project. If you want to improve performance at development time, you can disable this behavior. The server will then build a JSP file at run−time only when a client requests that page.

To disable pre−compilation of JSP files for a Web Project:

1. Click ***Tools/Project Properties/<ProjectName>***

2. From the ***Project Properties*** dialog, click ***Build*** in the left panel.
3. Under the heading ***Web Project***, check the box labeled ***Pre−compile JSP's*** and click ***OK***.

If you choose to pre−compile JSP pages, you can still avoid compiling all of the JSP files in a project by running the JSP file without explicitly building the project.

Manually Compiling Schema Projects

WebLogic Workshop automatically builds a schema project any time you import a new schema to the project or modify the contents of an existing schema within the project. To disable automatic building of schema projects:

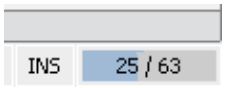
1. Right−click the appropriate schema project and click ***Properties***.
2. In the left pane of the ***Project Properties*** dialog, click ***Build***.
3. In the right pane, under ***Schema Project***, clear the ***Auto−build when file changes occur*** check box.

Managing WebLogic Workshop Memory

Increasing the amount of memory available to WebLogic Workshop and ensuring that all unused memory is available for re−allocation are two ways to ensure that your development environment performs optimally. The following sections describe how to manage memory for maximum performance.

Allocating More Memory to WebLogic Workshop

You can monitor the amount of memory used by WebLogic Workshop using the memory monitor located in the lower−right corner of the WebLogic Workshop IDE. The following figure illustrates the memory monitor:



The number on the left side of this monitor indicates memory being used by WebLogic Workshop. The number on the right side is the amount of memory currently available to WebLogic Workshop. Note that WebLogic Workshop is automatically configured to use a maximum of 256 Megabytes of memory by default. If you have additional memory to spare, you can increase this amount by opening the <BEA_HOME>/<WEBLOGIC_HOME>/workshop.cfg file (workshop.sh for Linux machines) and editing the portion of that file represented as red text below:

```
C:\bea\weblogic81\workshop C:\bea\jdk142_04\jre\bin\javaw.exe −XX:−UseThreadPriorities
    −Xmx256m −Xms64m −Xss256k −client −Dprerelease=true −Dsun.io.useCanonCaches=false
    −Dsun.java2d.noddraw=true −Dsun.java2d.d3d=false −Djava.system.class.loader="workshop.core.
    −cp "C:\bea\weblogic81\workshop\wlw−ide.jar" workshop.core.Workshop
```

Collecting the Garbage

As you build and run applications, WebLogic Workshop allocates memory that it later reclaims during a process called garbage collection. You can force WebLogic Workshop to reclaim that memory immediately by clicking on the memory monitor in the lower−right corner of the WebLogic Workshop IDE. When you click this monitor, all unused memory becomes available for building and running your applications.

Related Topics

Customizing the Build Process

Customizing the Build Process

WebLogic Workshop provides you with full control over the way you build JAR and EAR files from WebLogic applications and their projects. WebLogic Workshop accomplishes this by giving you complete access to the Another Neat Tool (ANT) XML build files that it uses to create these archives. You can customize the contents of these build files to specify the order in which WebLogic Workshop carries out build tasks, introduce new logic in between build steps, or integrate multiple build processes. WebLogic Workshop gives you access to both application−level or project−level ANT build files. The following sections discuss how to edit and use both files to customize your WebLogic Workshop build process.

Customizing Project−Level Builds

If you want to customize the way WebLogic Workshop builds an individual project, you can use the WebLogic Workshop IDE to export the ANT build file that WebLogic Workshop uses to build that project. WebLogic Workshop exports an ANT XML build file entitled exported_build to the home directory of that project. Change this file using syntax that conforms to the Apache ANT Specification. When you are through making your changes, simply reference that file in the project settings of the WebLogic Workshop IDE. When you build the project, WebLogic Workshop will use that custom build file to create the project JAR. For details on customizing a WebLogic Workshop project build, see How Do I: Use a Custom Ant Build for a Project?

If you want to automate the build process, simply use the BEA_HOME/weblogic81/workshop/wlwBuild.cmd and use the appropriate command line switch to specify that you only want to build that individual project. For more information on this command, see wlwBuild command.

Application−Level Builds

If you want to customize the way WebLogic Workshop builds an entire application, you can also use the WebLogic Workshop IDE to export the ANT XML build file that WebLogic Workshop uses to build the application. Like the project−level ANT file, WebLogic Workshop exports an XML file entitled exported_build to the home directory of the application. Use syntax that conforms to the Apache ANT Specification to execute logic from other archives, alter the order of project builds based on some condition, or make any other custom adjustment. Unlike project builds, you cannot execute this build file from within the WebLogic Workshop IDE. Instead you must execute the file using an ANT script. For more information on customizing an application build, see How Do I: Call wlwBuild.cmd from an ANT build.XML file?

Excluding Files From an Application−Level Build

When you build a WebLogic Workshop application, the Workshop compiler creates an Enterprise Archive (EAR) file. You can control what files WebLogic Workshop includes in that EAR by editing the excludefilesFromEar attribute in the applications .work file. The excludefilesFromEar attribute is shown below in red:

```
<option name="excludefilesFromEar" value="[default]" />
```

Note that WebLogic Workshop automatically sets the value of this attribute to [default]. This excludes file types with the extensions app, ctrl, dtf, ejbbean, java, jcs, jcx, jpd, jpf, jsx, jwf, jws, and wlbean by default.

Related Topics

How Do I: Use a Custom Ant Build for a Project?

How Do I: Call wlwBuild.cmd from an ANT build.XML file?

wlwBuild Command

Apache Ant 1.5.4 Manual

How Do I ...?

This section includes topics that explain how to manage the WebLogic Workshop build process.

Topics Included in This Section

How Do I: Compile a Single Java File?

How Do I: Use a Custom Ant Build for an Application?

How Do I: Call wlwBuild.cmd from an Ant build.xml file?

How Do I: Compile a Single Java File?

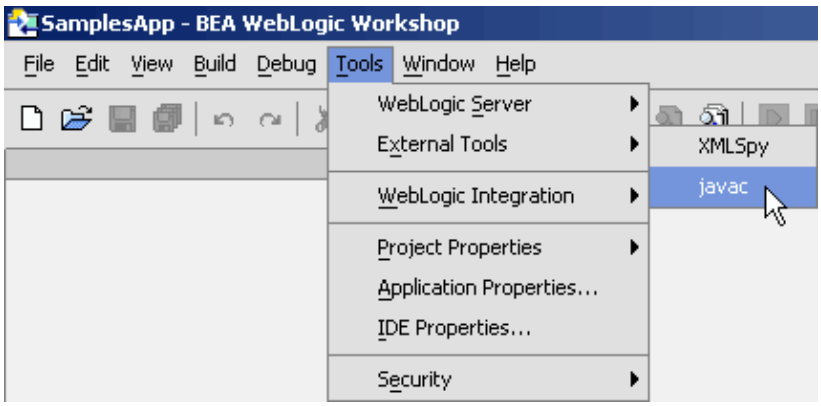
To selectively compile a single JAVA file, you can integrate a custom compilation script into the Workshop IDE by declaring the custom script as an "external tool". You can then pass single JAVA files from the IDE to the custom script.

Declaring a Custom Compilation Script as an "External Tool"

1. On Windows operating systems, open the the Workshop Preferences file located at C:\Documents and Settings\[Windows_user_name]\.workshop.pref.
2. Edit the element <component name="workshop.workspace.tools.CustomToolAction"> to include your custom compilation script. This element determines the options that appear in the menu bar Tools−−>External Tools. (By default XMLSpy is already included as a menu option.)

```
<component name="workshop.workspace.tools.CustomToolAction">
  <node name="XMLSpy">
    <option name="capture" value="false" />
    <option name="command" value=""C:\Program Files\Altova\xmlspy\XMLSpy.exe" "${fil
  </node>
  <node name="javac">
    <option name="capture" value="true" />
    <option name="command" value="compile.cmd  "${file.path}"" />
    <option name="directory" value="C:\build_scripts" />
    <option name="interactive" value="false" />
  </node>
</component>
```

The element <node name="javac"> causes a menu option named "javac" to appear on the menu bar Tools−−>External Tools.



The element <option name="directory" value="C:\build_scripts" /> tells Workshop where to look for your build script (you can place the build script in any local directory).

The element <option name="command" value="compile.cmd" "${file.path}" /> tells Workshop to pass the currently open file to the build script.

Place the Custom Compilation Script in the Appropriate Directory

- Save your build script in the directory specified in the element <option name="directory" value="C:\build_scripts" />. Below is a build script that you can use as a template.

compile.cmd

```
set PATH=D:\jdk1.4.2_04\bin;

set CLASSPATH=MyClasses

set CLASS_DIR=C:\bea\weblogic81\samples\workshop\SamplesApp\WebApp\WEB-INF\classes

javac -d %CLASS_DIR% -classpath %CLASSPATH% %1

echo compiled %1

popd
```

To compile a JAVA file, open the file in the Workshop IDE, and select Tools-->External Tools-->javac. The compile results will be displayed on the javac tab.

Related Topics

How Do I: Use a Custom Ant Build for a Project?

How Do I: Call wlwBuild.cmd from an ANT build.xml file?

wlwBuild Command

Apache Ant 1.5.4 Manual

How Do I: Use a Custom Ant Build for an Application?

WebLogic Workshop uses a standard Ant build process for all applications. You can create a custom Ant build file for an application that extends the standard build process or uses different build targets altogether. If you want to extend or modify the standard build process, you should first export the standard application build file and begin by modifying that file.

To Export the Standard Application Build File

1. Choose ***Tools***−−>***Application Properties*** to display the ***Application Properties*** dialog.
2. In the left−hand pane, select ***Build***.
3. In the ***Build Type*** pane, click ***Export to Ant File***. A dialog appears informing that the exported_build.xml file has been stored in the root of the application folder. Click ***OK***.

The exported Ant file contains a set of standard tasks and targets for managing the build for your application.

Note the class workshop.core.WlwBuildTask called by the "wlwBuild" task is the same class called by the command line tool wlwBuild.cmd.

```
    <taskdef name="wlwBuild" classname="workshop.core.WlwBuildTask" classpath="${weblogic.home}
```

The most important target in the Ant file is the "build" target.

```
  <target name="build">
    <!−− Builds the full application creating an ear at the specified location.
         The server classpath parameter is used, but may be omitted if the server home director
         configured in Workshop is accessible. −−>
    <wlwBuild work="${app.dir}/${work.file}" serverclasspath="${server.classpath}">
       <build outputdir="${output.dir}" outputfilename="${output.file}"/>
    </wlwBuild>
  </target>
```

Note that the following parameters are available for the "build" target:

project: String. Name of a specific project within the application to build. If omitted, the entire application will be built.

outputdir: String file path. Directory where build output file(s) will be placed. Defaults to application directory. May not be applicable for some types of project build.

outputfilename: String file path. Filename to use for output archive. Defaults to *appname*.ear. May not be applicable for some types of project build.

nodebug: Boolean value. Turns debugging information on or off. If no value is specified, the value is set to false.

noear: Boolean value. Specifies whether an EAR file should be generated for the application build. If no value is specified, the value is set to false. Not applicable for a project build.

To Create a Custom Ant Build File

You can use the standard exported build task and add additional build (sub)targets, or you can create a new ant file. To add a new task, you must use the <taskdef> element:

<taskdef name="myTask" classname="myTask" classpath="AllMyClass.jar"/>

Use the classpath attribute to specify the physical location of the JAR file as shown above, or you can specify the location when you change the project properties to use the custom build file as is described next.

You can also create a custom build process for an individual project within your application. Again, you should begin by exporting the standard build file and modifying that file.

To Export the Standard Project Build File

1. Open the target application in WebLogic Workshop.
2. From the ***Tools*** menu, select ***Project properties***, then the target project name.
3. In the ***Project Properties*** dialog, from the left−side navigation panel, select the ***Build*** node.
4. In the section labeled ***Build Type***, click the ***Export to Ant file*** button.
5. An Ant file named ***exported_build.xml*** is generated and placed in the project's root directory, e.g., BEA_HOME/weblogic81/samples/workshop/SamplesApp/WebServicesexported_build.xml.

To Use the Project Build File

1. Choose ***Tools***−−>***Project Properties***−−>***<projectname>*** to display the ***Project Properties*** dialog.
2. In the left−hand pane, select ***Build***.
3. In the ***Ant Settings*** area, click ***Browse***, navigate to the new build file and click ***Open***.
4. If your build file contains a new task definition without a classpath attribute, click ***Add JAR*** to specify the location of the Ant task. Navigate and select the file, then click ***Select JAR***.
5. In the ***Build Type*** area, select ***Use Ant Build***.
6. From the ***Build Target*** drop−down list, select the build target.
7. Click ***OK***.

The ***Ant*** window will now appear in WebLogic Workshop, listing the various targets. From this window you can run a target by double−clicking it.

Related Topics

How Do I: Compile a Single Java File?

How Do I: Call wlwBuild.cmd from an Ant build.xml file?

wlwBuild Command

Apache Ant 1.5.4 Manual

How Do I: Call wlwBuild.cmd from an Ant build.xml file?

The following topic explains how to create an Ant task that calls the command line tool wlwBuild.cmd. wlwBuild.cmd is used to create application−level EAR files or project−level JAR files for deployment to a production server.

To create an Ant build.xml file, you can either auto−generate a build.xml file or you can write the file by hand.

To auto−generate an Ant build.xml file for the application

1. Open the target application in WebLogic Workshop.
2. From the ***Tools*** menu, select ***Application properties***.
3. In the ***Application Properties*** dialog, from the left−side navigation panel, select the ***Build*** node.
4. In the section labeled ***Export***, click the ***Export to Ant file*** button.
5. An Ant file named ***exported_build.xml*** is generated and placed in the application's root directory, e.g., BEA_HOME/weblogic81/samples/workshop/SamplesApp/exported_build.xml.

The generated Ant file contains two tasks by default: "build" and "clean". When calling the build file from the command line, use the −f flag to name the file exported_build.xml:

```
C:\bea\weblogic81\samples\workshop\SamplesApp>ant −f exported_build.xml build
```

To build selected projects with the application, modify the <build> element within the exported_build.xml file. By adding the attribute project="Schemas", only the Schemas project will be build:

```
<target name="build">
  <!−− Builds the full application creating an ear at the specified location.
       The server classpath parameter is used, but may be omitted if the server home director
       configured in Workshop is accessible. −−>
  <wlwBuild work="${app.dir}/${work.file}" serverclasspath="${server.classpath}">
     <build outputdir="${output.dir}" outputfilename="${output.file}" project="Schemas"/>
  </wlwBuild>
</target>
```

See the documentation within the exported_build.xml file for more information about specifying output directories, output file names, etc.

You can also export an Ant build.xml file for building an individual project in much the same way as you do for the application.

To auto−generate an Ant build.xml file for a project

1. Open the target application in WebLogic Workshop.
2. From the ***Tools*** menu, select ***Project properties***, then the target project name.
3. In the ***Project Properties*** dialog, from the left−side navigation panel, select the ***Build*** node.
4. In the section labeled ***Build Type***, click the ***Export to Ant file*** button.
5. An Ant file named ***exported_build.xml*** is generated and placed in the project's root directory, e.g., BEA_HOME/weblogic81/samples/workshop/SamplesApp/WebServicesexported_build.xml.

To Write an Ant Script Manually

You can all the wlwBuild.cmd command line tool from an Ant script by using the <EXEC> task. For more information see the the Apache Ant 1.5.4 Manual and the reference documentation for the wlwBuild Command.

Related Topics

wlwBuild Command

Apache Ant 1.5.4 Manual

How Do I: Use a Custom Ant Build for a Project?

How Do I: Compile a Single Java File?

Using Source Control with WebLogic Workshop

WebLogic Workshop integrates directly with these source control systems: IBM Rational ClearCase, CVS, and Perforce. The topics in this section describe how to set up your source control system and your WebLogic Workshop application to most effectively put your application under source control.

Topics Included in This Section

Integrating with Source Control Systems

Provides general information on how to integrate your application with source control, including which files in a Workshop application to add to the repository.

Source Control Integration with ClearCase

Describes how to integrate your Workshop application with ClearCase.

Source Control Integration with CVS

Describes how to integrate your Workshop application with CVS.

Source Control Integration with Perforce

Describes how to integrate your Workshop application with Perforce.

Related Topics

Applications and Projects

Integrating with Source Control Systems

WebLogic Workshop integrates directly with the following source control systems: CVS, Perforce, and IBM Rational ClearCase. Once you have added the files in your WebLogic Workshop application to a respository managed by one of these source control products, you can check files in and out using commands available in WebLogic Workshop.

Enabling Source Control Integration in WebLogic Workshop

If you are part of a team development project and the application you are working on is already in a source control repository, you can simply point at that repository from WebLogic Workshop to enable source control integration from within the IDE. After you enable source control integration, the source control module's commands are available to you in WebLogic Workshop. For example, when you right−click on a file you will see a menu item for the name of the source control module (***CVS***, **Perforce**, or **ClearCase**), and beneath that menu item, the commands available for working with the file.

For more information on enabling source control integration with one of these source control systems, see Source Control Integration with CVS, Source Control Integration with Perforce, or Source Control Integration with ClearCase.

To Configure WebLogic Workshop for Source Control Integration

1. Configure your source control system as advised by your system administrator. In most cases you will need at least a client configuration that specifies how your computer interfaces with the source control system.
2. From the **Tools** menu, select **Application Properties**.
3. Click the **Source Control** tab.
4. From the **Source control module** drop−down, select Perforce, CVS, or ClearCase.
5. Set the properties for the source control module to map to the configuration on your computer.

If you want to enable source control integration only for a project within your application, or if you have different projects within different source control repositories, you can specify source control settings at the project level. From the ***Tools*** menu, choose ***Project Properties***−−><***projectname***>, click the ***Source Control*** tab, and clear the ***Use application's source control settings*** option.

Using Source Control Commands from WebLogic Workshop

Once you've configured Workshop for source control integration, you can perform commands against individual files in your application or project. There are two ways to execute source control commands:

- Right−click the file and choose the name of the source control system you're using −− ***CVS***, ***Perforce***, or ***ClearCase***. The commands available for that file are listed on the submenu.
- Open the file in WebLogic Workshop. From the ***Tools*** menu, choose your source control system; the available commands are listed on the submenu.

Which Files Should You Add to Source Control?

There are a lot of files in a WebLogic Workshop application, and not all of them need to be checked in. The files which should be checked in are those that are not modified by the build process. These files include

source files and some other files that are created with the project but never modified. The project can be built while any of these are read−only (for source control systems that govern read/write access), so that you only have to check out the files that you wish to modify.

The files to add to source control are:

- The .work file that represents the application. It appears at the root of the application directory.
- All source files that you have added or modified −− JWS, JCX, JSX, JSP, and so on.
- Any XML schema files that you have added to the Schemas project.
- The files in the Resources folder in a web project.
- Any JAR files that you have added to the **Modules** folder. These files are stored at the root of your application in the file system.
- Any JAR files that you have added to the **Libraries** folder. These files are stored in the APP−INF/lib folder in the file system.
- The following files in the WEB−INF folder in a web project: web.xml, weblogic.xml, wlw−config.xml.
- The global.app file in the WEB−INF/src/global folder.
- The tag libraries that appear in the WEB−INF folder in a web project, if you have page flows and JSP files in your project. These files end with the .tld and .tldx extensions.
- The JAR files that appear in the WEB−INF/lib folder in a web project.

The files that are modified by the build process −− compiled classes, for example −− do not need to be added to source control. You can add them, but you'll have to check all of them out each time you build your application. And since many of them are binary files and can't be modified directly, you don't gain anything by keeping them under source control; if you're developing your application as part of a team, putting these files under source control is likely to generate confusion for team members.

The files that you do not need to add to source control are:

- Files in the .workshop directory beneath the base application directory
- Files in the META−INF directory
- Files in the WEB−INF/.pageflow−struts−generated directory in a web application project
- Files in the WEB−INF/classes directory in a web application project
- Files in the WEB−INF/lib directory in a web application project

Modifying the .work File Under Source Control

There is currently no way to check out the .work file manually from within WebLogic Workshop in the same way that you can check out other files in your application. However, if the .work file is read−only, and you make a change that requires modifying the .work file, WebLogic Workshop will prompt you to check it out.

Changes to your application which modify the .work file include:

- Adding, importing, or deleting a project
- Changing settings in the ***Application Properties*** dialog.
- Changing settings in the ***Project Properties*** dialog.

Note: You should be careful about checking in the modified .work file when you are working in a team development environment. When you save the file, WebLogic Workshop may change the server.path variable within the file from a relative path to an absolute path to the server directory on your computer. Since it's

unlikely that other users have the exact same absolute path on their computers, the application may not function properly after they sync to your change. Unless you specifically want to check in a change to the .work file, you generally want to revert your changes rather than checking them in. A good way to know is to perform a diff operation between the file that's in the source control repository and the modified file on your local computer. If you do want to check in changes to the .work file, you may need to edit the .work file manually using a text editor to change the server.path variable back to a relative path value.

Related Topics

Source Control Integration with ClearCase

Source Control Integration with CVS

Source Control Integration with Perforce

WebLogic Portal Team Development Guide

ClearCase Source Control Integration

WebLogic Workshop integrates with the IBM Rational ClearCase. Information about ClearCase is available at http://www−306.ibm.com/software/rational/offerings/scm.html. This topic describes how to put your Workshop application under source control with ClearCase.

WebLogic workshop integrates with these ClearCase products: ClearCase, ClearCase MultiSite and ClearCase LT, versions V2003, V2002 and V4.2.

Setting Up Source Control Integration with ClearCase

WebLogic Workshop supports integration with ClearCase for a Workshop application and all of its projects, or for an individual project with an application. In either case, all of the files of the Workshop application or project must be associated with a single ClearCase Version Object Base (VOB). This restriction also means that:

- The mapping between the files in a Workshop application or project and the ClearCase VOB must be defined by a single dynamic or snapshot view.
- The root directory for the application or project must reside beneath the root directory for the ClearCase VOB. If the root directory for the application or project is in a subdirectory of the VOB root directory, then all of the parent directories must also exist in the VOB.

Note that it's possible for a single Workshop application to contain multiple projects that are mapped to different VOBs and have separate associated views.

To Add Your Workshop Application or Project to ClearCase

1. Make sure that you have a dynamic or snapshot view that provides access to the VOB where your application or project files will reside. If you need help creating this view, see the ClearCase documentation or ask your system administrator.
2. Create a new Workshop application or project in a directory that is beneath the root directory for the VOB. If you are copying or moving an existing application or project, you should clean it before you add it to source control, so that build artifacts are not added along with source files. To clean an application, select the application name in the ***Application*** pane, right−click, and select ***Clean Application***. To clean a project, select the project name in the ***Application*** pane, right−click, and select ***Clean <projectname>***.
3. If you're adding an application to source control, select ***Tools−−>Application Properties***, then select the ***Source Control*** tab. If you're adding a project, select ***Tools−−>Project Properties−−><projectname>***, then select the ***Source Control*** tab and clear the ***Use application's source control settings*** option.
4. Set the ***Source control module option*** to ***ClearCase***.
5. Set the ***cleartool directory*** option to point to the location of the ClearCase cleartool utility. If you accepted the defaults on installation, the directory containing the cleartool utility should look something like C:\Program Files\Rational\ClearCase\bin. Note that you should include the path only, not the file name. The cleartool utility is the command−line utility that WebLogic Workshop uses to integrate with ClearCase.
6. Set the ***ClearCase Version*** option to your server version. Be sure to verify that you have specified the right version, as you may experience problems with ClearCase integration if the version is incorrect.
7. Set the ***ClearCase view type*** setting to snapshot or dynamic, depending on your view type.
8. Set other options in the properties dialog as desired.

9. When you click OK, WebLogic Workshop will verify the location of the cleartool utility and will verify that the application or project's root directory is beneath the ClearCase VOB. If your view is a snapshot view, WebLogic Workshop will also prompt you to perform an update against the parent directory of the application or project root directory.

Adding Files to ClearCase

After you've configured WebLogic Workshop to integrate with ClearCase, you can add the files in your application or project to ClearCase through the IDE. To add a file, select the file in the Application pane, right−click, and choose ***ClearCase−−>Add*** or ***Add and Checkin***.

There are some differences between versions of ClearCase in terms of how files and directories are added to source control. These differences are outlined in the following sections:

ClearCase V2003

With ClearCase version V2003, when you add a file to source control, its parent directories are automatically added as elements and checked out. If you add the application or project root directory, the parent of this directory, which is not visible in the Workshop IDE, is automatically checked out. You must use an external ClearCase tool to check this directory back in.

If you execute the ***Add and Checkin*** command on a file, the parent directories of the file are automatically checked in.

If you execute the ***Add and Checkin*** command on a directory, you must check in the parent directories of that directory manually in order to commit the addition.

ClearCase V2002 or Earlier

With ClearCase version V2002 or earlier, all parent directories must already exist in the VOB and must be checked out before you can add a file that resides in a subdirectory. You must check in all checked out parent directories to commit the operation.

Using External ClearCase Tools

In some situations you will need to use external ClearCase tools, such as ClearCase Explorer or the cleartool utility, to perform certain operations on files in your application or project. These operations include:

- Adding the .work file to ClearCase or checking it in. However, once you've added the .work file to source control, WebLogic Workshop will prompt you to check it out if you make a change that affects the .work file.
- Adding, checking out, or checking in the parent directory of the root directory of the application or project. In version V2003 only, this directory will be automatically checked out when the root directory is added to ClearCase, but an external tool is required to check it back in.
- Checking in a file that is not the most recent version on the branch.

The ClearCase Find Checkouts utility may be useful in conjunction with WebLogic Workshop. This utility shows all checkouts in the view, including those that are not visible in WebLogic Workshop, like the parent directory of the root directory of the application or project.

Checking Out Files

To check out a file from within WebLogic Workshop, right−click on the file in the ***Application*** pane and choose ***ClearCase−−>Checkout***.

If the file you are checking out is not the latest version in the VOB, you'll see a warning in the checkout dialog. At this point it's recommended that you dismiss the dialog and update the file before continuing. You can also choose to check the file out and merge your changes in with the head version in the VOB when you submit the file.

Note that you cannot check out a file that is writeable.

Stopping ClearCase Commands

Once in awhile a ClearCase command may fail to finish executing. If this happens, you can halt the command by right−clicking in the ***ClearCase*** window and choosing ***Stop***. Once you've enabled ClearCase integration, the ClearCase window is available by choosing ***View−−>Windows−−>ClearCase***.

Warning: Use caution when halting executing ClearCase commands, as doing so can have unpredictable or undesirable results.

Related Topics

Integrating with Source Control Systems

CVS Source Control Integration

WebLogic Workshop integrates with Concurrent Versions System (CVS), an open−source version control system. The CVS source, binaries, and documentation are available at http://www.cvshome.org. This topic describes how to set up and use CVS with a WebLogic Workshop application.

For more thorough information on CVS, you should consult the CVS documentation. A good resource for information on setting up a CVS repository and using CVS is *The CVS Book*, by Karl Fogel and Moshe Bar. It's available in various formats on the Internet under the GNU General Public License.

Setting Up Source Control Integration with CVS

The first thing you need to do to put your WebLogic Workshop application under source control in CVS is to install CVS and set up the CVS repository. Here's a brief outline of the steps you need to follow to set up a repository on your local machine.

To Create a CVS Repository on Your Local Computer

1. Make sure that the directory in which you've installed CVS is listed on your system PATH.
2. Initialize the repository from the command line. The CVS init command creates the repository directory and the CVSROOT subdirectory and copies in the necessary CVS config files. For example, cvs −d /myrepos init creates a new repository myrepos at the root level.
3. Set the CVSROOT environment variable on your machine so that you don't need to specify the repository every time you refer to it from the command line. For example, SET CVSROOT=/myrepos.
4. Modify the cvswrappers configuration file to specify that CVS should automatically detect and add JAR files as binary files. Otherwise you will later need to modify each of these files individually so that they are flagged as binary.

Once your repository is set up, you can add a WebLogic Workshop application to it as described below.

The following instructions outline how to connect to an existing CVS repository on a remote server.

To Connect to a CVS Repository on a Remote Server

1. If you are using a remote CVS repository, you'll need a user name and password for that CVS server, and you'll need to know the server host name and the path to the CVSROOT. Get these from your system administrator.
2. Set the CVSROOT environment variable to the remote repository. In this case, you use the :pserver: method to specify a remote connection, and you specify your user name, the server host name, and the CVSROOT directory. For example, the command should look something like the following: SET CVSROOT=:pserver:guest@cvsserver.bea.com:/remoterepos.
3. Use the CVS login command to login to the remote server, and specify your password when prompted.

To Add a WebLogic Workshop Application to CVS

1. Create a new WebLogic Workshop application to add to source control. If you have an existing application, you should clean it before you add it to source control, so that build artifacts are not added along with source files. To clean your application, select the application name in the

Application pane, right−click, and select ***Clean Application***.

2. Exclude certain files in the application from the repository by adding .cvsignore configuration files to the appropriate directories:
    1. Create a .cvsignore file containing the line ".workshop" in the base directory for the application (that is, the directory containing the application .work file).
    2. Create a .cvsignore file in the META−INF directory containing the lines ".wlwLock" and "*.xml".
    3. Create a .cvsignore file in APP−INF/lib containing the line "*".
    4. Create a .cvsignore file in the WEB−INF directory for each web application containing the lines ".pageflow−struts−generated", "class", "lib", "*.tld", "*.tldx". If you have added your own tag libraries or JAR files, you won't want to exclude those; instead list the files to be excluded by name.
3. Import your Workshop application into the repository. From the command line, navigate to the directory where your application resides. The directory from which you want to perform the import is the base directory for the application, the directory containing the application's .work file. From that directory, call the CVS import command, specifying a name for the directory beneath the CVSROOT directory into which the files will be imported. For example, cvs import −m "import samples" SamplesApp vtag rtag imports everything in the current directory into a project directory named SamplesApp beneath the CVSROOT. The −m flag specifies a log message for the operation, and the vtag and rtag parameters provide placeholders for additional information. You can specify any string for these values.
4. Create a working directory for your CVS project. Create a new directory in the file system, then navigate to that directory and perform a cvs checkout to add the project files to the working directory. Specify the project directory that you named in the import command. For example, to populate the working directory using the SamplesApp project that was imported into the repository in the previous step, call cvs checkout SamplesApp from the command line within your new working directory.
5. You may want to rename your original project directory (the one from which you imported your application files) so that you don't confuse it with the working directory. In the working directory, CVS keeps track of which files have been modified, so that it can submit those to the respository. You can also delete the original project directory once you're sure that the import has been successful.
6. Open your Workshop application from the working directory. You may be prompted to update the libraries for any web application projects. These libraries don't need to be added to source control, but are necessary for your application to build properly.
7. The CVS import process may not import empty directories, so you may have to re−create them. For example, if the Schemas project was empty at the time of import, you may need to delete and re−create this project within your application.
8. In WebLogic Workshop, select ***Tools−−>Application Properties***, ***Source Control*** tab.
9. Specify CVS for the ***Source control module*** setting.
10. Specify the location of your repository for the ***CVSROOT*** setting. For example, you might specify /myrepos if you are using a local repository. If you have already set the CVSROOT environment variable, this setting shouldn't be necessary, but it doesn't hurt to include it.
11. The CVS file commands should now be available on each file in your application (excepting those you have excluded). Right−click the file name in the ***Application*** pane and choose ***CVS*** from the menu to see the available commands.

Related Topics

Integrating with Source Control Systems

Perforce Source Control Integration

WebLogic Workshop integrates with the Perforce Source Control Management System. Information about Perforce is available at http://www.perforce.com. This topic describes how to put your Workshop application under source control with Perforce.

Setting Up Source Control Integration with Perforce

Before you put your application under source control with Perforce, you need a Perforce account and password and port information for the Perforce depot. You also need a client spec that includes the depot view where your application will reside. See your system administrator for assistance with these requirements.

To Add Your Workshop Application to Perforce

1. Create a new WebLogic Workshop application to add to source control, in the directory where you want it to reside in the depot. If you have an existing application, you should clean it before you add it to source control, so that build artifacts are not added along with source files. To clean your application, select the application name in the ***Application*** pane, right−click, and select ***Clean Application***.
2. Add the files in your application to Perforce. For a list of the files that you should add and those that you should exclude, see Integrating with Source Control Systems. Because you are adding some files and excluding others, you don't want to add the entire application directory recursively; on the other hand, you probably don't want to add files one at a time either. One way to add individual directories recursively is to arrange the file system and Perforce windows side by side on your screen, and drag and drop each directory onto the Perforce changelist.
3. If you want to ensure that an empty directory is created in Perforce, create a .ignore file within that directory and add it to Perforce.
4. To test that you have added the appropriate files, you can submit your file additions to the Perforce depot, rename the application directory on your local file system, and re−sync to the same revision in Perforce. Then open the application in Workshop. You will be prompted to update missing or outdated libraries; after you do this, your application should build successfully, barring any build errors in your code.
5. In WebLogic Workshop, select ***Tools−−>Application Properties***, then select the ***Source Control*** tab.
6. Set the ***Source control module*** option to ***Perforce***.
7. WebLogic Workshop will try to fill in the ***Port***, ***ClientSpec***, ***User name***, and ***Password*** fields for you; you can modify these if you wish to use something other than the default settings.

Related Topics

Integrating with Source Control Systems

Message Logging

WebLogic Workshop reports diagnostic information that you can use to track the run−time activities of a WebLogic Workshop application. You can use this information to diagnose unexpected behaviors or monitor the general progress of any WebLogic Workshop application.

This topic describes how WebLogic Workshop reports diagnostic information and shows you how you can configure WebLogic Workshop to report information in a way that best suites your needs.

Log Files

Customizing the Way WebLogic Workshop Logs Messages

IDE Log Messages

Log Files

WebLogic Workshop automatically writes all internal log messages to a file named workshop.log. If you use Log4j to add log messages to your application code, WebLogic Workshop will append those messages to workshop.log as well. For information about adding logging messages to your application code, see workshopLogCfg.xml Configuration File.

You can find workshop.log in the root directory of the domain under which your WebLogic Workshop application runs. For example, if you are running the SamplesApp application that ships with WebLogic Workshop, you can find this file in the <BEA_HOME>/<WEBLOGIC_HOME>/samples/domains/workshop/directory. This is because the SamplesApp application runs in the workshop domain.

In addition to workshop.log, WebLogic Workshop writes all warning and error messages to a file named workshop_errors.log. This file is located at <BEA_HOME>/<WEBLOGIC_HOME>/workshop. You can ignore the file labeled workshop_debug.log located in that same directory. The workshop_debug.log file contains messages that are only useful to internal BEA developers, support representatives, and quality assurance representatives.

The workshop.log and workshop_error.log files contain every piece of information that WebLogic Workshop reports about its internal libraries as well as the any Log4j log messages that you add to your WebLogic Workshop application code. Note that WebLogic Workshop only reports run−time messages to these files. If you are interested in reviewing compile−time messages, you can view those messages in the ***Build*** window of the WebLogic Workshop Integrated Development Environment (IDE).

Customizing the Way WebLogic Workshop Logs Messages

You might want to customize the way WebLogic Workshop reports diagnostic information. For instance, you might want to log information to a console or database instead of a file. Perhaps you would like to adjust the way these messages appear in the log file. If you want to have better control over these details, you can do so by editing the workshopLogCfg.xml file. This is WebLogic Workshop's default log configuration file. You can locate workshopLogCfg.xml at <BEA_HOME>/<WEBLOGIC_HOME>/common/lib/.

The workshopLogCfg.xml file defines which run−time libraries report information, the type of information they report, the display format of that information, and the log files to which that information is written. You can customize any of these details by editing workshopLogCfg.xml. Alternatively, you can override the settings in workshopLogCfg.xml by providing your own custom configuration file. For guidance on both approaches, see workshopLogCfg.xml Configuration File.

IDE Log Messages

WebLogic Workshop also logs messages related to the WebLogic Workshop IDE. These messages can help BEA technical support troubleshoot issues you may encounter when developing applications with WebLogic Workshop.
WebLogic Workshop appends IDE related log messages to file named ide.log. You can find this file at <BEA_HOME><WEBLOGIC_HOME>/workshop/.

Because this information is of minimal use to most WebLogic Workshop developers, WebLogic Workshop disables IDE logging by default. If you contact BEA technical support, representatives might ask you to enable logging in order to better diagnose IDE related issues. To configure WebLogic Workshop to log IDE related messages to the ide.log file:

1. Open <BEA_HOME><WEBLOGIC_HOME>/workshop/Workshop.cfg (workshop.sh on Linux or Unix machines) in a text editor.
2. Add −Dworkshop.enablelogging=true to the javaw.exe command.

Related Topics

workshopLogCfg.xml Configuration File