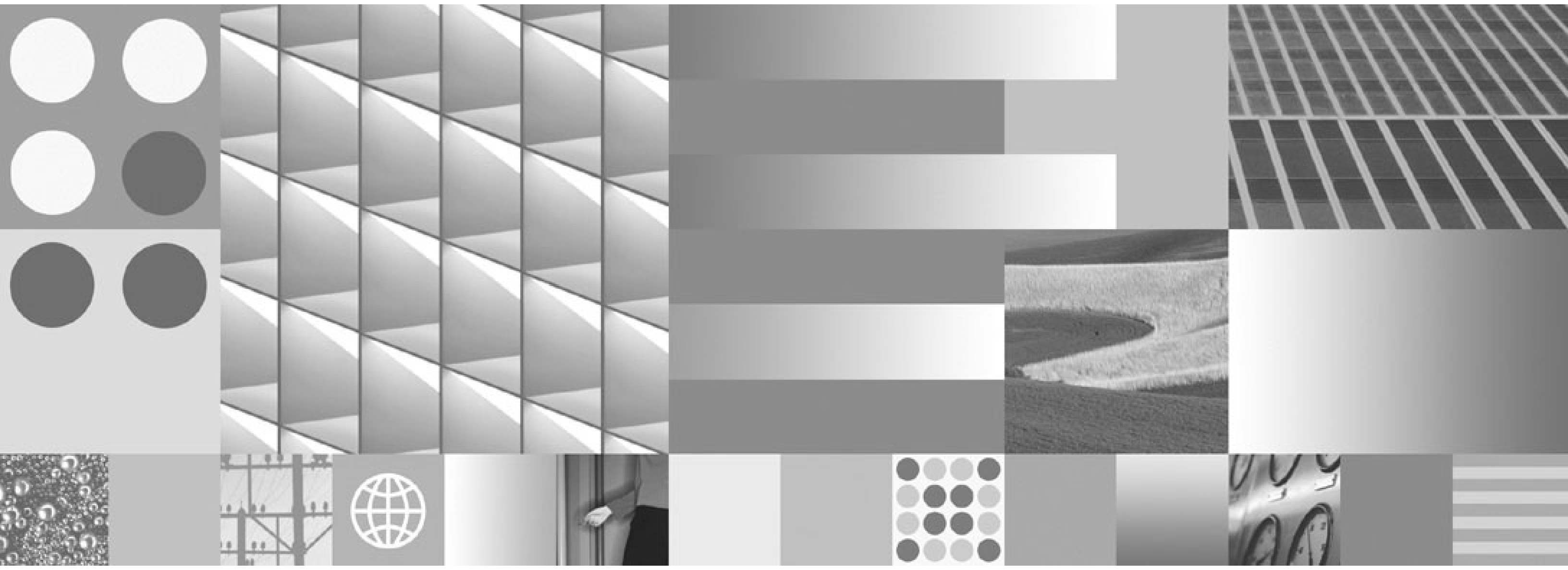
IBM WebSphere Everyplace Deployment



Developer's Guide

Note

Before using this information and the product it supports, read the information in "Notices," on page 261.

Second Edition (August 2005)

This edition applies to version _6.0_, and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Introduction . . . 1
Client platform . . . 2
  Managed client services. . . . 3
  Access services . . . 4
  Interaction services . . . 5
  Platform management . . . 6
  Packaging . . . 6
  Class loading . . . 9
Application models. . . . 11
  Rich client application model . . . 11
  Web application model . . . 12
  Composite application model . . . 12
Application design considerations . . . 12
  End-to-End applications . . . 12
  Business logic . . . 14
  Persistence . . . 14
  Messaging. . . . 14
  Management . . . 15
  Serviceability . . . 16
  Interaction. . . . 16
Client toolkit . . . 16
Getting started . . . 16
  Samples . . . 16
  Information roadmap . . . 17

Navigating and customizing the workbench . . . 21

Migrating and uninstalling . . . 23
Migrating . . . 23
  Automated migration . . . 23
  Manual migration . . . 23
Uninstalling WebSphere Everyplace Client Toolkit 24

Developing Web applications . . . 25
Client Services Web Application Concepts . . . 26
Client Services Web projects . . . 27
  Using a Client Services Web project versus a Dynamic Web project . . . 27
  Creating a Client Services Web project . . . 28
  Converting a Dynamic Web project to a Client Services Web project . . . 29
Client Services Web Application Development. . . . 29
  Unsupported tooling features . . . 29
  Accessing resources. . . . 29
  Using JSP Standard Tag Libraries . . . 30
  Java Server Faces (JSF) development . . . 30
  Struts development. . . . 31
  Securing Web Application resources . . . 31
Platform integration . . . 34
  com.ibm.eswe.workbench.WctWebApplication. . . 34
Importing and Exporting Web Application Bundles 37
  Importing a Web Application bundle . . . 37
  Exporting a Web Application bundle . . . 38
WAB Utility . . . 38
  WAB Utility installation . . . 38
  WAB Utility usage and parameters . . . 38
Web Container Logging . . . 40
  Configuring the Web Container Logging. . . . 40

Developing messaging applications . . 43
Technology overview . . . 43
  Publish and Subscribe – Topics, subscriptions and brokers . . . 43
  Point-to-point – queues and queue managers . . 44
Java Message Service . . . 45
WebSphere MQ Everyplace . . . 45
MicroBroker – Technical Preview . . . 46
MQ Telemetry Transport – Technical Preview . . . 47
WebSphere MQ Everyplace and MicroBroker comparison . . . 47
Client Services platform profile components . . . 48
Related documentation . . . 48

Developing the application user interface . . . 51
Technology overview . . . 51
  Eclipse . . . 51
  UI toolkits. . . . 52
  Visual Editor for Java . . . 52
Understanding the WebSphere Everyplace Deployment User Interface . . . 52
  User interaction in the WebSphere Everyplace Deployment 6.0 . . . 52
  User interface organization . . . 53
Creating a simple Rich Client Platform application 55
Extending the capabilities of your application . . . 57
  Adding and contributing menus . . . 57
  Creating views . . . 61
  Creating preferences . . . 61
  Applying Capitalization and punctuation guidelines . . . 63
  Creating helpful messages . . . 63
Customizing existing applications . . . 64
  Activities . . . 64
Integrating existing RCP applications into WebSphere Everyplace Deployment . . . 65

Developing data access and synchronization applications . . . 67
Databases . . . 67
  Embedded databases . . . 68
  DB2 Everyplace and Cloudscape comparison . . 68
  Creating database access application development best practices . . . 69
  Client Services platform profile components . . 70
  Deployment and synchronization . . . 70
  DB2 Everyplace and IBM Cloudscape Documentation . . . 72
SyncML . . . 73

Technology overview 74
SyncML4J common 74
SyncML4J data synchronization. 75
SyncML4J device management 75
Client Services platform profile components . . 75

Developing Embedded Transaction applications 77

Embedded Transaction concepts 78
Embedded Transaction projects 79
Using a Client Services Embedded Transaction project versus an EJB project. 79
Creating a Client Services Embedded Transaction project 79
Converting an EJB project to a Client Services Embedded Transaction project 82
Embedded Transaction specific considerations . . . 83
Implementing finder methods 83
Configuring and using data sources 85
Locating EJBs. 86
Conserving JDBC resources 86
Working with user managed transactions . . . 87
Providing custom bundle activation 87
Embedded Transaction Deployment 88
Invoking deployment 89
Automatic deployment when running or exporting a project 89
Manual deployment through the Deploy action 89
Ant script deployment. 89
Embedded Transaction Deployment Descriptor 89
Embedded Transaction Deployment Editor . . . 90
Customizing the deployment Ant script (ejb-build.xml) 91
Customizing for target data base (DB2e and Cloudscape) 91
Advanced topics. 91
Importing and Exporting Embedded Transaction Bundles 92
Importing an Embedded Transaction Bundle . . 92
Exporting an Embedded Transaction Bundle . . 93
Embedded Transaction specific debugging 93
Saving source for viewing during debugging . . 93
Enabling logging and tracing with the Embedded Transaction Container 93

Developing Mobile Web Services . . . 95

Mobile Web Services overview 95
Technologies 95
Tools 96
Creating Mobile Web Services 97
Creating Mobile Web Services providers 97
Creating Mobile Web Services clients 98
Editing Mobile Web Services 103
Custom serialization (marshalling) 103
Securing Mobile Web Services 109
Securing Mobile Web Services providers . . . 109
Securing Mobile Web services clients 110
Editing the Mobile Web Services security configuration 112
Deploying Mobile Web Services 140
Deploying Mobile Web Services providers . . . 141
Deploying Mobile Web Services clients 142
Validating Mobile Web Services using the Web Services Gateway Utility. 143
Accessing the Mobile Web Services Gateway Utility 143
Consuming Mobile Web Services 144
Exposing Mobile Web Services providers . . . 144
Listing Mobile Web Services client OSGi bundles 144
Listing Mobile Web Services provider OSGi bundles 145

Creating client runtime images 147

Getting started building platforms 147
Setting up the target platform 147
Creating a Platform Builder project 148
Updating a Platform Builder project. 149
Building a target runtime image 149
Exporting a target runtime image. 150
Running a target runtime image 150
Platform Builder page options. 150

Debugging and testing applications 157

Local debugging and testing 157
WebSphere Everyplace Deployment Launcher 157
WebSphere Everyplace Deployment Server . . 158
Remote debugging and testing 159
Remote debugging using Rational Software Development Platform 159
Starting from the command line 160

Packaging and deploying applications 161

Packaging applications for distribution 161
Understanding methods of installation 161
Understanding the types of install artifacts . . 162
Using Ant tasks to build a deployable bundle 165
Deploying projects for local testing 166
Exporting Client Services projects 166
Exporting plug-ins from the Rational Software Development Platform 167

Using the IBM WebSphere Everyplace Client Toolkit 169

Getting started with the IBM WebSphere Everyplace Client Toolkit 169
IBM WebSphere Everyplace Client Toolkit overview 169
Supported platforms 169
Roadmap of major tasks. 170
Setting up the WebSphere Everyplace Client Toolkit. 170
Creating a sample Client Services project . . . 171
Setting Manifest Editor preferences 171
Turning on build automatically 172
Concepts 172
Managing Client Services project dependencies 173
Creating and using Client Services applications . . 176
Creating a Client Services project. 176
Creating a Client Services fragment project . . 177

Converting a Java project into a Client Services project . . . 178
Setting Client Services project properties . . . 178

Using Platform Services. 179

Using the HTTP Service . . . 179
handleSecurity plug-in . . . 179
Configuring HTTP Service . . . 179
Using the Log Service and Log Reader Services 182
Configuring Log Service . . . 182
Using the Meta Type Service . . . 183
MetaData XML sample . . . 184
Using the XML parser services . . . 185
Creating help for the application . . . 186
Using logging and tracing . . . 187
Eclipse logging . . . 187
Eclipse tracing . . . 187
OSGi logging . . . 188
Enabling your plug-in for startup . . . 188
Using runtime developer tools . . . 190
Using the Platform Manager . . . 190
Using Admin Utility for OSGi . . . 192
Using the JNDI Manager . . . 194
Configuring Enterprise Definitions (JNDI) . . . 195
WebSphere Everyplace Deployment JNDI overview . . . 195
Using declarative JNDI . . . 196
Packaging . . . 199
Extending declarative JNDI. . . . 199
Globalizing your application . . . 200
Support for multiple locales . . . 200
IBM language groups. . . . 201
Supporting preferred fonts and bidirectional layouts . . . 202
Creating translatable plug-ins . . . 202

Troubleshooting 205

Logging . . . 205
Logging framework . . . 205
Manually adjusting the logging level . . . 206
Log file management . . . 207
Tracing . . . 207
Using the IBM Support Assistant . . . 208

Reference information 211

WebSphere Everyplace Deployment top level menus . . . 211
File menu . . . 211
Application menu . . . 211
View menu . . . 212
Help menu . . . 212
Extension points reference . . . 213
Applications. . . . 213
Web applications . . . 214
JNDI Binding . . . 217
JNDI generic object . . . 218
JNDI object factory . . . 219
WebSphere Everyplace Client Toolkit ANT task and types . . . 220
bde.exportJarBundle . . . 220
filedirectory . . . 221
buildpolicyinfo . . . 221
Managing client configurations . . . 222
Understanding the client file layout . . . 222
Configuring Java system properties . . . 223
Configuring Java VM arguments . . . 223
Configuring native library references . . . 223
Configuring Java Bootclasspath libraries . . . 223
Configuring workbench options . . . 224
Configuring the web container . . . 224
Configuring platform configuration files . . . 228
Specifying platform branding . . . 231
Configuring the platform launcher . . . 236
OSGi . . . 237
OSGi specification . . . 237
Working with OSGi bundles . . . 237
IBM WebSphere Everyplace Client toolkit . . . 244
Editors . . . 244
Wizards . . . 245
Dialogs . . . 253
Advanced topics . . . 255
Platform Profile and Extension Points . . . 255

Appendix. Notices 261

Trademarks . . . 263

Introduction

The WebSphere Everyplace Deployment for Windows® and Linux® Developer's Guide is your tool for developing powerful applications that run on desktop and laptop clients across Windows and Linux operating systems. If you are experienced in developing J2EE applications, Web Services, or Eclipse applications, then you are ready to develop applications for WebSphere Everyplace Deployment for Windows and Linux.

With WebSphere Everyplace Deployment for Windows and Linux (also called *the client platform*), you can move key components of your applications from the server to desktop and laptop clients by using standard APIs and services.

Moving application components to run on a client can have dramatic results for business. End-users benefit from improved application response time because applications perform business operations locally on the client. As a result, there is a reduction in network traffic between clients and servers, and in server workload. Furthermore, mobile end-users can continue to productively use their applications from their clients even when they are at a location that does not have network connectivity, such as a customer site. You can also utilize the local graphical user interface (GUI) capabilities of the client devices to deliver a richer user experience than can be supported by a Web browser.

The client platform supports these capabilities by providing a set of standards-based *Client Services* on which you can build your applications. These Client Services include:

- **Managed Client Services** to enable multiple applications and services to run on the same JVM, support life cycle management of these applications and services, and provide application portability across Windows and Linux operating systems.
- **Platform Management** to install, maintain, and configure the applications and services on the client.
- **Access Services**, which provide middleware so your client applications can securely access e-business on demand applications, services and data and continue to operate even when a device is offline.
- **Interaction Services**, which support interaction with end-users through a traditional Web browser and a rich graphical user interface through Eclipse technology. Eclipse is an award-winning, open source platform for the construction of powerful software development tools and rich desktop applications.

The WebSphere® Everyplace® Client Toolkit (also called *the toolkit*) provides a complete, integrated set of tools that allows you to develop, debug, test, package and deploy client applications that use the Client Services. This toolkit is built on Eclipse technology and extends the powerful Rational® suite of development tools so you can leverage your existing skills and software components. The toolkit also provides Ant tasks so you can create Ant scripts to automate the building of your applications. In addition, the toolkit provides program samples to help jump start your application development projects.

The combination of the WebSphere Everyplace Deployment for Windows and Linux client and the WebSphere Everyplace Deployment server (also called *the* *server platform*) provide the client and server middleware necessary to deliver and manage end-to-end applications (see the below figure). Administrators use the WebSphere Everyplace Deployment server to install and configure the server middleware, so client applications can securely perform assured transactions and database synchronization with Enterprise applications and data. For more information on the server platform, please refer to the WebSphere Everyplace Deployment server documentation.

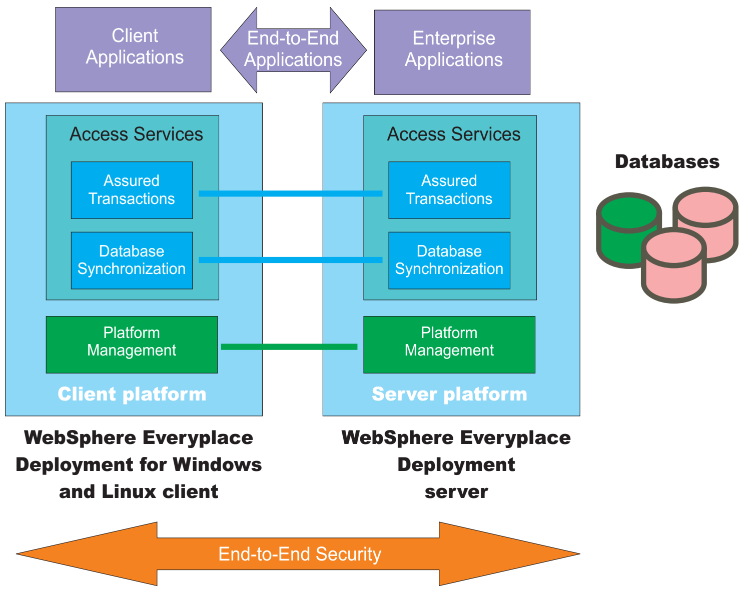


Figure 1. End-to-end solution

In summary, the powerful client platform, toolkit, and server platform enable you to develop compelling applications that run on desktops and laptops, and securely access e-business on demand applications, services, and data. You can use programming skills you have already acquired to develop these applications. This guide provides the information you need to deliver these applications to your customers.

Client platform

The client platform provides a way to install, launch and manage multiple applications within an integrated application desktop window. The client platform is built on the Eclipse 3.0.2 Rich Client Platform (RCP) framework. For application developers familiar with the Eclipse Integrated Development Environment, or Rational tools, it is the same core platform with the application development tools removed, leaving a basic container for features and components, and a window management and layout environment.

The client platform provides the following set of standards-based Client Services for the development of your client applications:

- **Managed Client Services** including a JVM, a robust component framework, and additional component services.
- **Platform Management** including the Eclipse Update Manager and an Enterprise management agent to install and update applications and services on the client platform.

- **Access Services** including relational database services and synchronization, transactional messaging, Web Services, a Web container to run local Web applications, an embedded transaction container to run local embedded Enterprise Java™ Beans (EJB's), and more.
- **Interaction Services** including integrated browser controls to launch Web applications, the Standard Widget Toolkit (SWT) and JFace Toolkit to support GUI applications, and a Workbench that enables end-users to install and launch one or more applications.

The client platform provides a standard set of APIs, such as JDBC and JMS, that you use to invoke these services.

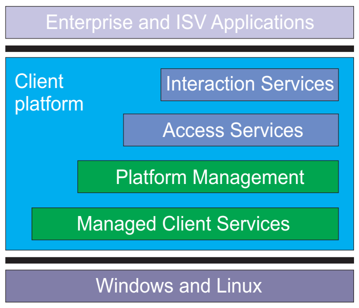


Figure 2. The client platform

Managed client services

The client platform provides IBM's Java 2 Standard Edition (J2SE) 1.4.2 JVM (service release 2) as the base Java runtime environment. The JVM runs on the client operating systems supported by the client platform: Windows XP and Red Hat Linux.

The client platform provides a Service Framework that implements the OSGi R3 framework specification and provides a service-oriented architecture on top of the JVM. The OSGi framework specification is provided by the OSGi Alliance. The OSGi Alliance's mission is to specify, create, advance, and promote wide industry adoption of an open service delivery and management platform. Incorporating the OSGi standard into the client platform provides four very important capabilities:

- It enables multiple applications and components to share a single Java Virtual Machine (JVM). This saves valuable resources on the client when running multiple applications because only one instance of the JVM is launched rather than multiple instances of the JVM.
- It enables applications to share services and packages, which further reduces resource requirements on devices.
- It separates service interface from service implementation and provides publish, find, and bind operations in support of a service-oriented architecture. This capability enables integration of business applications on the same device.
- It enables dynamic life-cycle management without a VM restart so components can be updated without impacting other unrelated components that are running at the same time.

The Eclipse RCP framework is built on the Service Framework, which provides the Eclipse RCP with powerful capabilities, such as the ability to dynamically load and unload components without restarting the Eclipse RCP framework and robust life cycle management of components.

The client platform also provides optional OSGi services, such as UserAdmin, LogService, Configuration Management, and more.

Access services

Access Services provide a familiar programming model for J2EE developers so they can reuse their skills and software components to develop applications that run on clients. Additionally, Access Services enable client applications to support offline operations. Access Services also enable you to move key components of your application to the client platform through the use of standard APIs.

The client platform provides an embedded Web container to run J2EE Web applications that support either the Servlet 2.3 and JSP 1.2 specifications, or the Servlet 2.4 and JSP 2.0 specifications. The Web container enables you to move your Web applications from the server to clients to preserve the existing browser user interface, leverage your existing Web components, and provide a richer user experience through support of local and offline operations.

The client platform also provides an embedded Transaction Container to run J2EE Enterprise Java Beans (EJB's) that conform to any of the following specifications: 1.1 and 2.0 Stateless Session Beans, Container Managed Persistence (CMP) Entity Beans, and Bean Managed Persistence (BMP) Entity Beans. This container enables you to move your business logic from the server to clients so you can leverage your existing beans to make business logic available to client applications, including Web applications, and support local and offline operations. These business logic components are referred to as *Embedded Transaction applications*.

There are two key services that support local and offline operations.

First, you can use the JDBC API with DB2® Everyplace or IBM® Cloudscape™ as a local SQL database when more advanced data manipulations are required than can be supported by placing data in a local file store. These databases can periodically synchronize with Enterprise databases to capture data on the client for use by the client application when the user is offline. These databases can also protect local data through data encryption.

DB2 Everyplace is an extremely small footprint relational database (200-300 KB). It is especially suitable for embedded devices, where large databases and sophisticated queries are not normally required, but can also be used on larger devices. DB2 Everyplace provides transaction support covering updates to multiple tables within a single transaction, encrypted tables, and zero client administration.

IBM Cloudscape is a 100% pure Java relational database, providing SQL-92, partial SQL-99, and SQLJ support, indexes, triggers, transactions, encryption, and the standard features that one expects of a relational database. Because IBM Cloudscape contains a larger number of features, it is approximately 2 MB in size. Therefore, IBM Cloudscape might not be suitable for smaller, resource-constrained devices.

Second, you can also use the Java Message Service (JMS) API with WebSphere MQ Everyplace (MQe) to send and receive messages. MQe provides once-only, assured messaging and supports offline operations with local message queues that hold messages when the device is offline and then sends these queued messages to Enterprise applications when the device is back online. Similarly, messages destined for client applications are held in server-side message queues and then sent to the client applications when the device is back online. MQe encrypts messages to protect content over the network. As a result, the client platform enables your users to conduct secure e-business on demand transactions.

For online operations, the client platform supports Web Services so client applications can consume and provide Web Services in a secure manner. As a result, your users have access to a broad range of business data and consumer information. The client platform implements Web Services similar to those defined in JSR 172 and provides support for document literal encoded streams that exchange well-typed data objects so client applications can consume Web Services. You can also develop an OSGi service and, during registration of the service, indicate that it is also available as a Web Service.

The client platform also supports a technical preview of the MicroBroker, which is suitable for applications that require messaging, notification and event services. The MicroBroker supports publish and subscribe messaging in which publishers generate messages containing information about a particular subject, subscribers express interest in messages containing information on a particular subject, and a broker receives messages from publishers and delivers messages on a particular subject to the subscribers registered for that subject.

The SyncML4J (SyncML for Java) toolkit enables you to develop data synchronization and device management client applications based on the Open Mobile Alliance (OMA) Data Synchronization (DS) and Device Management (DM) standard protocols. As a framework, SyncML4J supports user-defined data sources. Data sources can range from simple resources, such as memos and images, to complex schema-aware data types, such as relational databases or PIM databases.

Interaction services

The client platform is built on the Eclipse Rich Client Platform (RCP) so you can deliver applications that provide a rich user experience across multiple platforms. The client platform provides the Workbench, Standard Widget Toolkit (SWT), JFace, Help and Preferences interaction services.

The Workbench provides an integrated application desktop window so end-users can install, manage and launch one or more applications within a single window. The Workbench presents each application individually in its own perspective, only one of which is visible at any given time. When an end-user selects an application from the Workbench, the Workbench launches the perspective for that application. You specify an extension point for each of your applications so the Workbench can correctly launch the perspective for your application.

The client platform supports servlets and JSP's so users can interact with local Web Applications through a Web browser. Each Web application installed onto the Workbench runs in a browser perspective. When an end-user selects a Web application from the Workbench, the Workbench launches a browser perspective which in turn launches a local Web browser to run the Web application within the Workbench window. When you specify the extension point for Web applications, the Workbench automatically handles launching your Web applications in the browser perspective. You can also use this extension point to enable the Workbench to launch a Web application on a remote server.

The client platform also supports rich client applications, which interact with end-users through a graphical user interface (GUI). Each rich client application installed onto the Workbench runs in an application perspective. In this case, each application must contribute its own perspective to the Workbench. In each perspective, an application provides the collection of views, layout of views, and actions appropriate for the tasks that end-users will perform with the application. You use SWT and the JFace toolkit to develop the GUI for rich client applications. SWT provides a cross-platform API that tightly integrates with the native widgets of the operating system and, therefore, gives your applications a look and feel that makes them virtually indistinguishable from native applications. The JFace toolkit provides a set of components and helper utilities that simplify many of the common tasks in developing SWT user interfaces. When an end-user selects a rich client application from the Workbench, the Workbench launches the appropriate perspective to run the application within the Workbench window. When you specify the extension point for rich client applications, the Workbench automatically handles launching the perspectives for your rich client applications.

The client platform also provides services that enable you to contribute Helps and Preferences for your applications so end-users can understand and configure your applications respectively within the Workbench.

Platform management

Platform Management installs, maintains, and configures applications and services on the client. There are two platform management services.

The Update Manager enables end-users to directly install applications and components from standard Eclipse update sites onto the Workbench.

The Enterprise Management Agent works cooperatively with the Device Management Server provided by the WebSphere Everyplace Deployment server to perform management operations. The agent and server use the SyncML/DM protocol defined by the Open Mobile Alliance to communicate management requests. An administrator can schedule management jobs for devices that include software installation, update, and configuration. When installing and updating software components, the management system determines which components are already on the device and then installs only the missing components.

Packaging

Client applications and application services are packaged as features, each of which consists of one or more components. The client platform cannot directly run J2EE packaging artifacts such as EAR and WAR files.

Components

The Eclipse framework, and therefore the client platform, is organized around a plug-in and extension point model. The framework provides a core set of components. Additional components are provided in a directory or JAR file organized in a specific structure, and implement instantiations of the various extension points. The framework reads the component declarative information, and incorporates the components into the correct locations in the framework.

A *plug-in* is the level at which components are declared to the Eclipse framework. A plug-in is a JAR file with a plug-in manifest file named `plugin.xml`. The plug-in manifest describes the plug-in to the framework and enables a plug-in to consume and/or provide extensions from/to other plug-ins. For example, a plug-in can provide user interface extensions, such as perspectives, views, editors, and wizards. It can also provide business logic or core services to other plug-ins, but contribute no extensions to the user interface.

A *bundle* is the level at which components are declared to the OSGi Service Framework. A bundle is a JAR file with a bundle manifest file named `MANIFEST.MF`. The bundle manifest describes the bundle to the service framework and enables a bundle to consume and/or provide packages and services from/to other bundles. Bundles can also include a Bundle Activator class. The Bundle-Activator header in the bundle manifest file identifies the class to the framework. At startup time, the framework creates an instance of this class and calls its `start()` method. The Bundle Activator can then publish services, start its own threads, and so on. When the bundle shuts down, the framework calls the activator's `stop()` method. While the bundle shuts down, the Bundle Activator can release resources that are obtained since the start method was called and revoke any services it has published.

Recall that the Eclipse framework is built on the OSGi Service Framework. Therefore, you can define each component in your applications as a plug-in, a bundle, or both depending on your requirements.

Note: For a component to be recognized by the client platform, the toolkit and the Eclipse Plug-in Development Environment (PDE), it must have a unique name and version. If you develop a plug-in, specify a unique value for the `name` attribute and a version number for the `version` attribute in the plug-in manifest. If you develop a bundle, specify a unique value for the `Bundle-SymbolicName` attribute and a version number for the Bundle-Version attribute in the bundle manifest.

A component can generally be organized in one of three ways:

- A directory containing at least a `plugin.xml` file. The directory may also contain a `MANIFEST.MF` file located in the `META-INF` directory, additional files, as well as Java code contained within JAR files.

  A `plugin.xml` file is required if the component defines extension points for use by other plug-ins or implements extension points provided by other plug-ins.
- A directory containing at least a `MANIFEST.MF` file in the `META-INF` directory. The directory will also contain Java code contained in JAR files. The `MANIFEST.MF` will refer to the JARs by referencing them via the `Bundle-Classpath` attribute.

  Components that provide only business logic or OSGi services and do not intend to provide or implement any extension points can use this format. Components without `plugin.xml` files that need to be available when building other components or when launching the client platform by using either the toolkit or the PDE must be organized in this format.
- A single JAR file containing at least a `META-INF\MANIFEST.MF` or a `plugin.xml` file.

  Components may be provided for use in the client platform by collecting all of the component artifacts into a single JAR file. While this organization will run successfully, this organization is not compatible with the toolkit and Eclipse PDE.

Fragments

A component may not always provide a complete implementation. In some cases, *fragments* may be used to complete or extend a component.

For example, the primary component may provide an implementation that contains translatable text in a default language. Fragments are then used to provide translations for additional languages.

A second case where fragments are often used is to provide platform (processor/operating system) specific implementations.

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

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

Features

On disk, an Eclipse-based product is structured as a collection of components and fragments. Each component or fragment contains the code that provides some of the product's functionality. The code and other files for a component or fragment are installed on the local computer, and get activated automatically as required. A product's components are grouped together into *features*. A feature is the smallest unit of separately downloadable and installable functionality

The fundamentally modular nature of the Eclipse platform makes it easy to install additional features and components into an Eclipse-based product, and to update the product's existing features and components. You can do this either by using traditional native installers running separately from Eclipse, or by using the Eclipse platform's own Update Manager. The Eclipse Update Manager can be used to discover, download, and install updated features and components from special web based Eclipse update sites.

The basic underlying mechanism of the Update Manager is simple: the files for a feature or component are always stored in a sub-directory whose name includes a version identifier (e.g., "2.0.0"). Different versions of a feature or component are always given different version identifiers, thereby ensuring that multiple versions of the same feature or component can co-exist on disk. This means that installing or updating features and components requires adding more files, but never requires deleting or overwriting existing files. Once the files are installed on the local computer, the new feature and component versions are available to be configured. The same installed base of files is therefore capable of supporting many different configurations simultaneously; installing and upgrading an existing product is reduced to formulating a configuration that is incrementally newer than the current one. Important configurations can be saved and restored to active service in the event of an unsuccessful upgrade.

Large Eclipse-based products can organize their features into trees starting from the root feature that represents the entire product. This root feature then includes smaller units of functionality all the way down to leaf features that list one or more plug-ins and fragments. The capability to group features hierarchically allows products to be built on top of smaller products by including the smaller products and adding more features.

Some included features may be useful add-ons but not vital to the proper functioning of the overall product. Feature providers can elect to mark these features as optional. The Update Manager allows users to choose whether or not to install optional features. If not installed right away, optional features can be added at a later date.

Class loading

This section explains how classes are located and loaded by the client platform. A class loader is responsible for loading classes. A class is loaded by a hierarchy of cooperating class loaders as shown in the figure below.

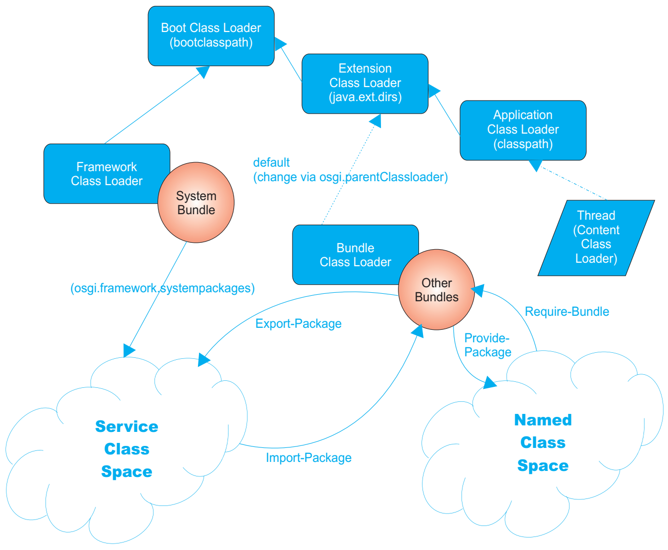

Boot Class Loader (bootclasspath)
Extension Class Loader (java.ext.dirs)
Application Class Loader (classpath)
Framework Class Loader
System Bundle
default (change via osgi.parentClassloader)
Thread (Content Class Loader)
Bundle Class Loader
Other Bundles
(osgi.framework.systempackages)
Export-Package
Require-Bundle
Provide-Package
Service Class Space
Named Class Space
Import-Package

Figure 3. Class loaders

A typical Java application has a global name space that consists of the contents of the JARs in a single, well-defined class path. A set of class loaders cooperates to locate and load classes based on this class path. These class loaders include the Application Class Loader to load application classes (normally found in the `CLASSPATH`), the Extension Class Loader to load standard extension classes (normally in the `jre/lib/ext` directory, which is specified in the `java.ext.dirs` property), and the Boot Class Loader to load system classes (normally from `rt.jar` in the `jre/lib` directory).

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

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

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

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

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

The figure below illustrates the search order used to locate classes and resources.

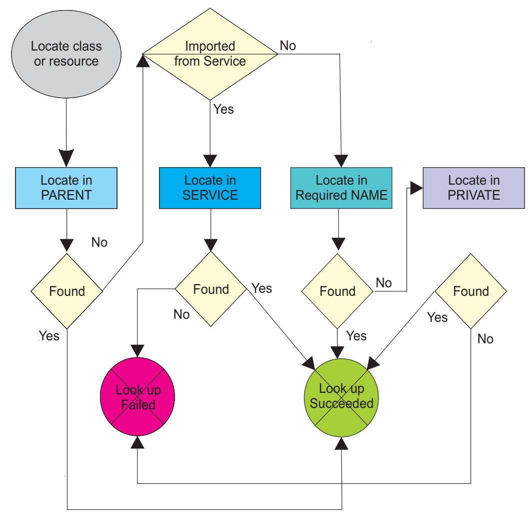


Figure 4. Search order used to locate classes and resources

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

1. The Bundle Class Loader delegates the request to its parent class loader (PARENT), which results in the Boot Class Loader and then the Extension Class Loader attempting to locate the class or resource. If the class or resource was found, then the class loader returns this result. If the class or resource was not found, then the search continues with the next step.
2. If the Bundle Class Loader determines that the requested class or resource is in a package imported from the Service Class Space (SERVICE) and it was found

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

3. The Bundle Class Loader searches the Named Class Space (NAMED) and if the class or resource was found, then the class loader returns this result. Otherwise, the search continues with the next step.
4. The Bundle Class Loader searches its own internal class path and the internal class path of any attached fragment bundles. If the class or resource was found, then the class loader returns the results. If the class or resource was not found, then the search terminates and the request fails.

Application models

There are two application user interface patterns that are recommended for use in the client platform. The first pattern is the browser user interface pattern, which is supported by the Web Application Model. Web applications present their user interface through the use of generated scripting language such as HTML, which is rendered for display by a browser.

The second pattern is to build a graphical user interface using Eclipse, to aggregate display components into views, and views into perspectives. Applications will be defined using extension points to define the actions, views, and perspective that provide the user interface. This pattern is the rich client user interface pattern, which is supported by the Rich Client Application Model.

The figure below details the client platform containing four applications. Each application is composed differently.

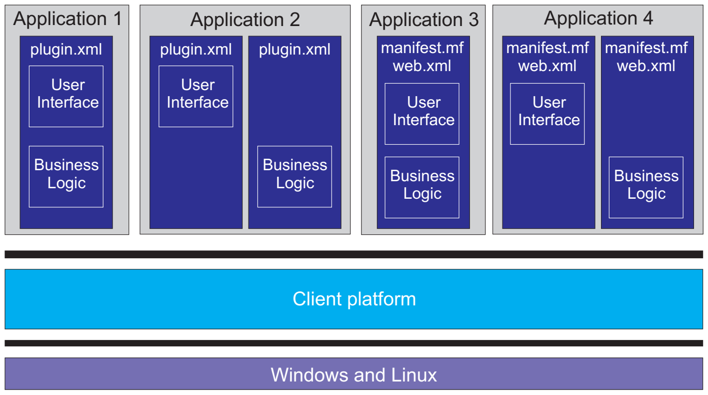


Figure 5. The client platform, with four applications

Rich client application model

Applications that compose their user interface using extension points will typically follow a plug-in model of providing components. Plug-ins will implement the defined extension points to provide functions. The plug-ins may make use of services provided by other plug-ins. In (Figure 5), Application 1 is composed of a single plug-in using extension points to create a user interface. Application 2 is composed of multiple plug-ins, one plug-in providing user interface components and a second plug-in providing business logic and packages through its `plugin.xml` file.

Web application model

Web Applications can be built to either the Servlet 2.3 and JSP 1.2 specifications, or to the Servlet 2.4 and JSP 2.0 specifications. Web Applications can make use of technologies such as Tag Libraries, Templates, and other standard Web Application features. Web Applications will typically be constructed of components that follow the standard OSGi bundle format.

In (Figure 5), Application 3 is a web application contained in a single bundle (with only the `MANIFEST.MF` file), while Application 4 separates the business logic and user interface components into two bundles (containing only `MANIFEST.MF` files).

Composite application model

Application models are not limited to just Rich Client applications or Web Applications. The term *Composite Application* refers to applications that make use of both plug-ins and bundles, such as those identified in the figure below.

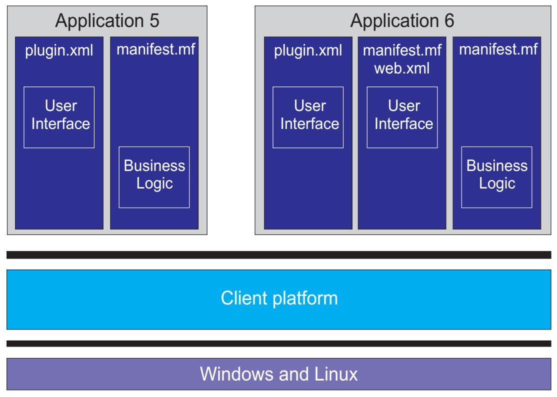


Figure 6. The client platform, with two applications

Application 5 is composed of a plug-in that provides user interface components and a bundle that provides the business logic. Application 6 provides the main user interface via a web application, but implements extension points in a plug-in to provide preferences so an end-user can customize the application.

Application design considerations

You can design client applications in much the same way that you design standard Enterprise applications. However, there are unique considerations for designing client applications. The list of considerations in this section is not necessarily comprehensive for all possible decision points; however, this list provides key considerations for developing client applications.

End-to-End applications

The client and server platforms enable the development of end-to-end applications through Access Services that connect client applications to Enterprise applications, services and data. An end-to-end application can be distributed between a client device and a server in which case there are two nodes in the application. However, an end-to-end application might be distributed across more nodes. The exact design of an end-to-end application depends on your specific requirements; however, this section discusses specific examples that illustrate how you might construct these applications (the figure below).

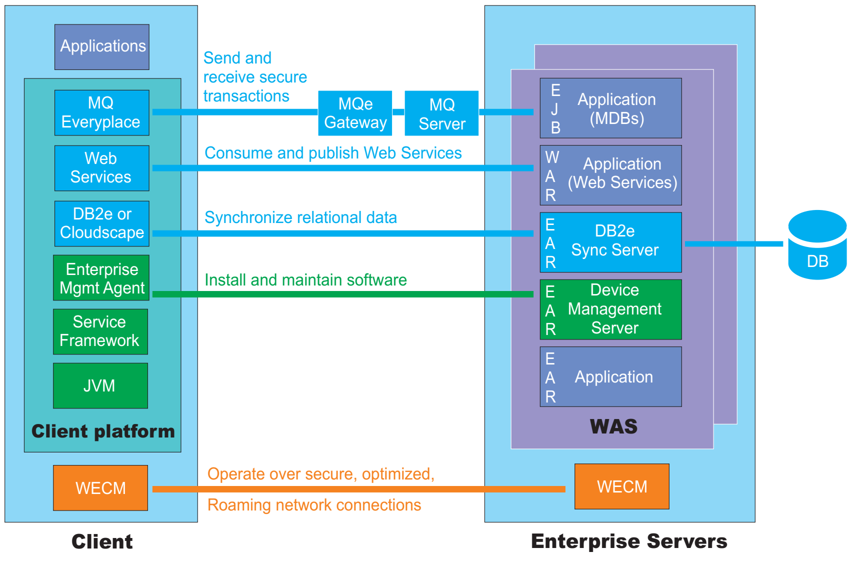

Applications
MQ Everyplace
Web Services
DB2e or Cloudscape
Enterprise Mgmt Agent
Service Framework
JVM
Client platform
WECM
Client
Send and receive secure transactions
MQe Gateway
MQ Server
Consume and publish Web Services
Synchronize relational data
Install and maintain software
Operate over secure, optimized,
Roaming network connections
EJB Application (MDBs)
WAR Application (Web Services)
EAR DB2e Sync Server
EAR Device Management Server
EAR Application
WAS
WECM
DB
Enterprise Servers

Figure 7. Example end-to-end application

Client applications can use WebSphere MQ Everyplace (MQe) to exchange secure transactional messages with Enterprise applications that support MQ messaging. MQ Everyplace operates in many topologies, from peer-to-peer, to client, to server using the MQ Everyplace gateway technology. For example, a J2EE application can implement Message Driven Beans (MDB) to exchange messages with a client application. This exchange can occur through an MQ Everyplace Gateway-MQ Server configuration. However, client applications can also use MQe to exchange messages directly with other MQe applications in the network.

DB2 Everyplace and Cloudscape are both capable of synchronizing with the DB2 Everyplace (DB2e) Sync Server, using the IBM ISync technology provided by the client platform. A System Administrator configures the DB2e Sync Server to synchronize data with Enterprise databases. The initial synchronization activity creates the local database schema, and also populates the initial set of data in the local database on a device. When a client application updates the local database, synchronization can transfer that data to Enterprise databases that are configured to receive it. When Enterprise applications update data in an Enterprise database, synchronization can transfer that data to local device databases that are configured to receive it. Database administrators set up the DB2e Sync Server with the necessary subscriptions for synchronization, and can set up filtering of data to limit the amount of data distributed between nodes. The DB2e Sync Server supports synchronizing relational data on the client with relational data on the following Enterprise databases: DB2 Universal Database™, Informix® Dynamic Server, IBM Cloudscape, Lotus® Domino® Server, Oracle, Microsoft® SQL Server, and Sybase Adaptive Server Enterprise.

Note: Administrators can use the WebSphere Everyplace Deployment server to install and configure the DB2 Everyplace Synchronization Server, WebSphere MQ Everyplace, including the function necessary for WebSphere MQ Everyplace to act as a gateway server to the WebSphere MQ products, and the Device Management Server.

Client applications can also consume and provide Web Services. This requires an active connection between the Web services consumer and provider.

An optional capability, which some customers have chosen to implement, is another IBM product called WebSphere Everyplace Connection Manager (WECM). WECM enables client applications to operate over secure, optimized, roaming network connections on wireless and wire-line networks. WECM installs below TCP/IP APIs so TCP/IP applications can continue to run without change and benefit from these capabilities.

Business logic

When you build an end-to-end application, you must decide how to distribute the business logic across the nodes that comprise the application so your mobile users are productive when they are offline. The amount of business logic in the client application must be sufficient to perform all necessary work. However, you might consider moving business logic components that require frequent updating to a server node to avoid network traffic and administration costs associated with managing these same components on clients.

In addition, client applications can provide multiple levels of capability, reserving some capability for when a reliable connection exists to an Enterprise server, and disabling that capability if the server is unavailable.

Business logic can be packaged within a client application or made available as a separate component, such as a plug-in, an embedded EJB or local Web Service.

Persistence

Most applications manipulate data. This can take the form of read-only access of databases to retrieve catalog items, or of database update for creation of orders that need to be processed. Data can take the form of files distributed on disk, or relational database capability. When dealing with databases, you can choose to use databases only as a local data repository, or as a repository that actively synchronizes with an Enterprise database.

Consider the following design possibilities:

- Use a database as a local repository when your application requires more advanced data organization and access capabilities than can be supported by a local file store – especially if a relatively large amount of data is stored on the device.
- Use a local database to protect, or encrypt, data in case a device is lost or stolen
- Use database synchronization to exchange the current state of data between local databases and an Enterprise database, when transaction boundaries or the order of state changes is not important
- Consider how much data needs to be distributed and when (once only at initialization, one-way from one node to another only on an infrequent basis, frequent exchange between nodes). Balance these considerations with the storage capabilities at each node, and the networking requirements that would permit the exchange to take place.
- Consider database organization, filtering, and conflict resolution policies if you choose to use database synchronization.

Messaging

Messaging links client applications to Enterprise applications, services, and data. Messaging can take various forms, whether a plain socket-based application, Web Services, or a more sophisticated store and forward transaction messaging capability that supports connected as well as disconnected usage. When choosing a form of messaging, you should consider the following requirements in the design of your end-to-end applications:

- Online vs. offline operation
- Security, including message confidentiality, non-repudiation, and authentication
- Synchronous vs. asynchronous messaging
- Once-only assured transactions
- Configuration of the messaging solution

Web Services support secure, online, synchronous access to information; however, an online connection must be active between the Web Services consumer and provider to access information across the network. Security features include message confidentiality, integrity, and authentication.

MQe provides transaction messaging that supports online and offline operation, security features (message confidentiality, non-repudiation, authentication), synchronous and asynchronous messaging, and once-only assured transactions. Transaction messaging provides a convenient mechanism for defining or identifying transaction boundaries when performing such actions as creating or updating orders, particularly if the transaction requires updates across multiple resources in the Enterprise such as inventory management, shipping and billing systems.

Note: Certain nodes in the end-to-end system might not be able to manage or commit transactions because these nodes might not have transaction coordination and, therefore, do not have the master copy of all of the data.

The MicroBroker implements publish and subscribe messaging, which supports online and offline operations, and synchronous and asynchronous messaging.

Management

Management covers a wide range of activities, from initial device provisioning to application management. When you design your application, you should consider the implications of your design in regards to application management, specifically in two key areas: componentization and data formats.

First, if you design a monolithic application, then the management system must distribute and install a complete new copy of the application to update nodes with the latest version of the application. Depending on the size of the application and the frequency of updates, this design might adversely affect network capacity and disrupt users. If you design and package the application as multiple installable components, then the management system distributes and installs only those components that require an update. You might also be able to reuse components in different applications that run on the same or different nodes. There is a trade-off between the granularity of the components and the complexity of administering the set of components that comprise an application.

Second, you must consider the effect on data when updating applications or components. If you design your components and data format so that local data is upwardly compatible, then users can continue to access their data after application and component updates. Otherwise, you must provide a mechanism to update the existing data to match a new or revised format and ensure that all installed components that consume this data can process this format.

Serviceability

Distributed applications pose additional issues of serviceability as compared to applications running on a single node. Logging and problem resolution might be difficult if the application is running on one node, and the node only occasionally connects to a central logging repository. In these situations, you must consider how to transfer logging information from a node to the central logging repository, how to track user usage, and help in problem resolution.

Interaction

You must first decide if your application will support user interactions and, if so, which interaction model to use. You might choose the Web Application model when moving a Web application from the server to the client to reduce development and training costs. You might choose the Rich Client Application model when you require more control over the user experience.

Client toolkit

The WebSphere Everyplace Client Toolkit provides a complete, integrated set of tools that allows you to develop, debug, test, package and deploy client applications that use Client Services. You can use the toolkit to develop the following types of client applications:

- Eclipse Rich Client Platform applications
- Web applications
- Embedded Transaction applications
- Database applications
- Messaging applications
- Web Services applications

The toolkit provides wizards that enable you to create Client Services projects to develop client applications. The toolkit uses Platform Profiles to provide a convenient method for you to specify the runtime environment, the build-time environment, and the set of components that can run on the platform. For example, when you create a Client Services project, you select a Platform Profile from a list of available profiles and the toolkit automatically sets up the Java Build Path and runtime for your project. You can then edit, compile, and debug your project. The toolkit provides a default list of Platform Profiles; however, you can create your own profiles.

You can also use the toolkit to build custom client platforms for your devices. However, custom platforms require an OEM license from IBM.

The toolkit is built on Eclipse 3.0.2 and extends the Rational suite of development tools so you can leverage your existing skills and software components.

Getting started

Samples

To get started using the client platform or specific features of the client platform, review the collection of client platform samples in the Samples Gallery. The Sample Gallery is a facility that is available in the Rational product set. It acts as a centralized location or repository for samples. The gallery is accessible from both the Welcome panel and from the Help menu. The samples in the gallery are split into three categories: Showcase samples, Application samples and Technology samples. Showcase samples are end-to-end applications that follow best practices for application development. Application samples are samples that demonstrate more than one tool or API while Technology samples are samples that demonstrate a single tool or API. When you open a sample, you see a short explanation of the sample and a link for importing the sample into your tooling workspace. Samples for the client are located in the WebSphere Everyplace Deployment section of each category.

Information roadmap

The following information roadmap provides a list of appropriate documentation for a variety of different user roles supported by the client. If the listed documentation is part of the Developer's Guide, then a link to the appropriate section is provided.

Table 1. Information roadmap

User Role	Appropriate Documentation
Web Developer • Creates Web components • Converts J2EE Web components • Migrates Workplace Client Technology™, Micro Edition 5.7.1 Web components • Selects Platform Profile for target • Specifies deployment information • Writes servlet source code • Writes JSP and HTML files • Optionally uses Struts or JSF • Compiles Web components • Packages the components in a Web Application Bundle (WAB) file • Unit tests and debugs components • Imports/exports WABs	WebSphere Everyplace Deployment Developers Guide: "Developing Web applications" on page 25 In addition, you can use the following services to develop your applications: • Developing messaging applications "Developing messaging applications" on page 43 • Developing data access and synchronization applications "Developing data access and synchronization applications" on page 67 • Developing Mobile Web Services "Developing Mobile Web Services" on page 95
Enterprise Bean Developer • Creates Embedded Transaction applications • Converts J2EE EJBs • Selects Platform Profile for target • Specifies deployment information • Writes and compiles the source code • Packages Embedded Transaction applications in Embedded Transaction bundles • Unit tests and debugs components • Imports/exports Embedded Transaction applications	WebSphere Everyplace Deployment Developers Guide: "Developing Embedded Transaction applications" on page 77 In addition, you can use the following services to develop your applications: • Developing messaging applications "Developing messaging applications" on page 43 • Developing data access and synchronization applications "Developing data access and synchronization applications" on page 67 • Developing Mobile Web Services "Developing Mobile Web Services" on page 95

Table 1. Information roadmap (continued)

User Role	Appropriate Documentation
RCP User Interface Developer • Creates RCP User Interface components • Selects Platform Profile for target • Writes and compiles the source code • Uses the required extension points to register applications with the client platform and integrate with the workbench, helps, and preferences • Specifies the `plugin.xml` file • Packages the .class files and `plugin.xml` file in a plug-in JAR file • Unit tests and debugs applications	WebSphere Everyplace Deployment Developers Guide: "Developing the application user interface" on page 51 In addition, you can use the following services to develop your applications: • Developing messaging applications "Developing messaging applications" on page 43 • Developing data access and synchronization applications "Developing data access and synchronization applications" on page 67 • Developing Mobile Web Services "Developing Mobile Web Services" on page 95
Platform Provider • Defines one or more Platform Profiles, which specify the components and applications for each platform • Builds platform-specific files that can be installed into devices with components and applications • Provides an Update site for optional components and/or updates	WebSphere Everyplace Deployment Developers Guide: "Creating client runtime images" on page 147
Application Integrator • Assembles components into Eclipse Features • Creates licenses for assembled Features • Executes a simple path through the application / solution to validate it • Verifies that the Feature can run on the Platform Profiles supported by the target devices	System Administrator's Guide
Solution Deployer • Checks for and installs dependencies • Configures applications for operational environment (e.g. a port) • Deploys applications and components via Update Manager • Deploys applications and components via the Device Management Server • Executes a simple install from the Update Site or the Device Management Server to verify Features can be installed correctly • Where necessary, manages the rollback to previous versions • Manages deployment on a day-to-day basis.	System Administrator's Guide

Table 1. Information roadmap (continued)

User Role	Appropriate Documentation
End User • Installs the client platform onto their desktop or laptop • Installs optional components • Launches the client platform • Installs one or more applications • Launches applications from the client platform • Updates applications • Migrates applications from WCTME-EO 5.8.1 to the client • Uninstalls applications • Uninstalls the client platform • Manages bundles/plug-ins • Changes/configures runtime environment settings • Submits problem log for support	WebSphere Everyplace Deployment for Window's and Linux User's Guide

Navigating and customizing the workbench

The IBM WebSphere Everyplace Client Toolkit provides a number of additional features to help developers build Client Services applications for the WebSphere Everyplace Deployment platform.

Enhanced validation of the bundle manifest can be configured via the preferences page. Access this page by navigating to **Window > Preferences > WebSphere Everyplace Client Toolkit**. On this screen you can configure the notification level you would like for the different possible problems detected in the manifest.

From the export panel (**Window > Preferences > WebSphere Everyplace Client Toolkit > Export**), you can configure the naming format of the JAR files exported via the OSGi bundle wizard. The default is the Eclipse naming and version convention.

The WebSphere Everyplace Client Toolkit uses the Eclipse target platform to build and run applications for WebSphere Everyplace Deployment. Upon installation or new workspace creation, you will be prompted to have the target platform and workspace JRE updated to the proper environment. If you wish to modify these settings by hand, please see “Setting up the target platform” on page 147.

Additional customizations to the WebSphere Everyplace Client Toolkit can be set via the PDE. Please refer to the *PDE Guide* available at **Extending Rational <*development platform*> functionality > Extending the workbench > PDE Guide**

Where <*development platform*> is one of the following, depending on your development platform:

- Software Architect
- Web Developer
- Application Developer

WebSphere Everyplace Client Toolkit project specific properties can be updated from the Project properties menu. To access these settings, select a Client Services project and click **Project > Properties > Client Services**.

Migrating and uninstalling

WebSphere Everyplace Client Toolkit provides two methods to migrate Extension Service 5.7 projects. Both methods convert Extension Service projects in the same way. The toolkit sets the JRE and platform profile for all migrated projects to default values.

The toolkit tries to migrate the application service from the platform profiles in SMF Bundle Developer, to the latest platform profiles in the WebSphere Everyplace Client Toolkit. After migrating a project, you may edit/add application services and change platform profiles. To do this, right click on the project and select **Properties**. Choose **Client Services**, then select the **Application Profile** tab. For more information on this screen, refer to "Setting Client Services project properties" on page 178.

Migrating

Automated migration

WebSphere Everyplace Client Toolkit allows you to migrate your Extension Services projects from WebSphere Studio Device Developer 5.7 workspaces. Upon startup on an existing WebSphere Studio Device Developer 5.7 workspace, the toolkit detects and automatically converts all workspace projects.

This modifies your projects without any user interaction. If you would like to preserve your 5.7 project format and data, you should back up the workspace prior to opening the WebSphere Everyplace Client Toolkit on the folder.

Note: During migration, the class path is updated to match current platform profiles, and the manifest is updated to match current runtime requirements. The toolkit will update the `MANIFEST.MF` file if one exists, creating a backup called `MANIFEST.MF.BAK`. It will also set the Java Build Path, the JRE, and platform profile to default values.

Manual migration

You may choose to import an existing 5.7 Extension Service project into a Rational Application Developer 6.0 workspace. If you would like to preserve your 5.7 project format and data, back up the project directory prior to using the Migration wizard. Use the Extension Services Migration Wizard to migrate your project by performing the following procedure:

1. Open the Wizard dialog by selecting **File > New > Other**.
2. Select **Client Services > Migrate Extension Services Project**.

   If there is no **Client Services** category, select **Show All Wizards** to display the **Client Services** category.
3. Select **Next**. The Migrate Extension Services Wizard opens.
4. Choose the projects you wish to migrate. Use the **Select All** button to migrate all Extension Service projects in the workspace, if necessary.

   Note: If the project(s) reference class path variables defined in other workspaces, they will not resolve. To redefine/copy them into the current workspace, you must include the location of the old workspace in the **Directory** text box at the bottom of the page. Select the **Browse** button to choose the folder.

5. Select **Finish** to migrate the selected project(s).

Note: During migration, the class path is updated to match current platform profiles, and the manifest is updated to match current runtime requirements. The toolkit will update the `MANIFEST.MF` file if one exists, creating a backup called `MANIFEST.MF.BAK`. It will also set the Java Build Path, the JRE, and platform profile to default values.

Uninstalling WebSphere Everyplace Client Toolkit

To uninstall the WebSphere Everyplace Client Toolkit, perform the following procedure:

1. Select **Windows > Preferences > Java > Installed JREs**. The Installed Java Runtime Environments panel displays, showing all installed JREs. Unselect **WebSphere Everyplace Deployment** (to remove it as the default JRE), and check the JRE originally provided with the Rational Software Development Platform.
2. Select **Windows > Preferences > Plug-in Development > Target Platform** and change the **Location** field to the Rational Software Development Platform, by selecting it from the drop down list.
3. Select **Help > Software Updates > Manage Configuration**. The Product Configuration menu displays. From the Navigator window, open the **sdpisv** Extension Location, and select IBM WebSphere Everyplace Client Toolkit. The information for the feature displays in the window to the right. Select **Disable**.[1]
4. Restart the platform when prompted.[1]
5. Select **Help > Software Updates > Manage Configuration**. The Product Configuration menu displays. From the Navigator window, open the **sdpisv** Extension Location and select **IBM WebSphere Everyplace Client Toolkit**. The information for the feature displays in the window to the right. Select **Uninstall**.
6. Select **OK** in the next dialog. Restart the platform when prompted.

1. Steps 3 and 4 are only required on Windows. Linux users can step directly from step 3 to step 5.

Developing Web applications

The WebSphere Everyplace Deployment platform supports Servlet 2.4 and JSP 2.0 web applications as well as Servlet 2.3 and JSP 1.2 web applications. Web applications targeting the WebSphere Everyplace Deployment platform are called Client Services web applications. Since components in the WebSphere Everyplace Deployment platform are called bundles, a web application targeting this platform is also referred to as a Web Application Bundle or WAB.

A WAB can be developed using many of the same web development tools provided by the Rational Software Development platform. You should therefore refer to the Rational online help section “Developing Web applications” as your initial web development tools reference. The following topics discuss the additional development considerations and tool usage required when targeting a web application for the WebSphere Everyplace Deployment platform.

The following table provides pointers to information on web development activities and information on tasks that are unique to, or require special consideration when developing web applications for the WebSphere Everyplace Deployment platform.

Table 2. Web development activities

Task	**Reference**
Understanding Client Services web application concepts.	“Client Services Web Application Concepts” on page 26
Working with Client Services Web projects versus Dynamic Web projects, and when to use one versus the other.	“Client Services Web projects” on page 27
Developing Client Services web application logic. This encompasses any special development considerations when coding and constructing the web application logic.	“Unsupported tooling features” on page 29 “Accessing resources” on page 29 “Using JSP Standard Tag Libraries” on page 30 “Java Server Faces (JSF) development” on page 30 “Struts development” on page 31
Integrating the web application into the WebSphere Everyplace Deployment platform. This includes registering the web application as a platform application, along with specifying browser behavior.	“Platform integration” on page 34
Importing and exporting web application bundles.	“Importing a Web Application bundle” on page 37 “Exporting a Web Application bundle” on page 38
Securing the web application through user authentication and authorization.	“Securing Web Application resources” on page 31
Debugging and testing the web application.	“Debugging and testing applications” on page 157

Table 2. Web development activities (continued)

Task	Reference
Deploying the web application to a runtime.	"Deploying projects for local testing" on page 166
Using the command line WAB tool to convert a WAR to a WAB.	"WAB Utility" on page 38
Configuring the runtime web container.	Refer to the Web Container Configuration information in the *WebSphere Everyplace Deployment for Windows and Linux Administrator's Guide*
Web container logging.	"Web Container Logging" on page 40

Client Services Web Application Concepts

Client Services web applications run on the WebSphere Everyplace Deployment platform. A primary difference between a Client Services web application and one that is deployed to run on a WAS or Tomcat runtime is that the Client Services web application must also be a valid OSGi bundle. Refer to "Working with OSGi bundles" on page 237 for more information on bundles and the WebSphere Everyplace Deployment platform. The WebSphere Everyplace Client toolkit automatically handles many of these bundle specific details, which is why developing the web application through a Client Services web project is the recommended development path for web applications that are to be run on the WebSphere Everyplace Deployment platform. Nevertheless, it is also possible to develop the web application through a Dynamic Web project, and subsequently test run it on the WebSphere Everyplace Deployment platform. Refer to "Using a Client Services Web project versus a Dynamic Web project" on page 27 for more details. It is also possible to transform an existing Web Application Archive (WAR) file into a Web Application Bundle (WAB) suitable for running on the WebSphere Everyplace Deployment platform through the use of the WAB Utility.

The following lists aspects of a Client Services web application that differ from a standard web application.

- The WebSphere Everyplace Deployment platform does not support deploying Enterprise Applications through an EAR. The web application is directly deployed to the runtime.
- A Client Services web application has a manifest file, located in `META-INF/MANIFEST.MF`, that contains bundle information including package and bundle dependencies. This is associated with the bundle, and is separate from the manifest file found under the web application's content folder.
- A Client Services web application contains additional deployment information in `wab.properties`. This is located in the web content `WEB-INF` folder.
- JSP files are translated into their respective servlet classes before the web application is deployed to the runtime as a WAB.

In most cases, these artifacts and differences are handled transparently by the WebSphere Everyplace Client tools. These differences do not affect the functionality of the web application. There are, however, some development considerations you should take into account depending on the web technologies you will be using. These are described in "Client Services Web Application Development" on page 29.

A Client Services web application can be developed using many of the same web development tools provided by the Rational Software Development platform. The primary differences are:

- Use the Client Services Web project wizard to create a Client Services web project, as described in "Creating a Client Services Web project" on page 28.
- Since the WebSphere Everyplace Deployment platform does not support EARs, EAR projects are ignored.
- When testing the project, target the WebSphere Everyplace Deployment runtime when using the Run / Debug on Server action. Or, use the WebSphere Everyplace Deployment launch configuration when using the Eclipse Run / Debug launch feature. This is explained in "Debugging and testing applications" on page 157.
- When exporting the web application, use the export OSGi bundle file wizard, as described in "Exporting a Web Application bundle" on page 38.
- Use the Import WAB wizard to import a Web Application Bundle.

Client Services Web projects

Using a Client Services Web project versus a Dynamic Web project

Web applications can be developed using either a Client Services web project or a Dynamic Web Project. The choice of which to use depends on the application content and its primary usage. In general, web applications that primarily target the WebSphere Everyplace Deployment platform or depend on other OSGi services besides core servlet and JSP support should be developed using a Client Services web project.

A Client Services web project is an extension of the dynamic web project. Because of this, both types of projects make use of the Rational Software Development web tools. In addition to this, a Client Services web project provides the following support for developing a web application that is targeting the WebSphere Everyplace Deployment platform.

- The OSGi manifest file required by WebSphere Everyplace Deployment applications can be automatically managed by the tools.
- The project's class path is maintained to match the class path environment that will exist in the WebSphere Everyplace Deployment runtime. This is useful for detecting class visibility problems at development time rather than runtime.

A Dynamic web project will not have the WebSphere Everyplace Deployment specific tooling aids listed above, but can still be tested and run on the WebSphere Everyplace Deployment platform. This is accomplished by targeting the project's server to the WebSphere Everyplace Deployment runtime through the project's server properties. The tooling will automatically add the proper OSGi manifest entries for Servlet and JSP support. However, if the application references other OSGi services or bundles, the developer will have to manually add these dependencies to the manifest file.

A Client Services web project can also be tested and run on a platform other than WebSphere Everyplace Deployment by reassigning its targeted runtime through the project server properties. Refer to "Debugging and testing applications" on page 157 for further information.

Creating a Client Services Web project

Perform the following procedure to create a new Client Services Web project:

1. Select **File > New > Project**. The new project wizard displays.
2. Expand the Client Services folder. This lists the Client Services project wizards. Choose **Client Services Web Project**, then select **Next**. The Client Services Web Project panel displays.
3. Specify a project name in the Project name field. This is the only field you are required to fill in. Select **Finish** to create a project with default settings.

The additional settings that can be configured through this wizard are described in the following tables, along with their default values. Access the additional wizard panels through the **Next** and **Back** buttons. Selecting **Finish** on any of the wizard pages will create the project with the settings you have specified up to that point.

Client Services Web Project panel

Table 3.

Option	Description	Default value
Project name	Enter a name for the new Client Services Web Project.	None
Project location	You may click **Browse** to select a file system location for the new project.	The default location creates the project in your current workspace.
Servlet version	The Servlet version that the project is intended to use.	2.4
Context root	The web application context root.	Project name

Platform Profile panel

Allows the platform profile and associated application services to be selected for the project. By default, the necessary services for supporting web projects will be selected. Refer to “Platform Profile” on page 174 for a discussion of platform profiles, and “Application Services” on page 174 for a discussion of application services.

Table 4.

Option	Description	Default value
Platform Profile	Select from the list the Platform Profile this Client Services project will target. You can change your selection later in the Client Services property page.	WebSphere Everyplace Deployment (6.0.0) Default
Application Services	Check the Application Services that your Client Services project will require. You can change your selection later in the Client Services property page. Grey entries are required by the Platform Profile and cannot be deselected.	The ″Core OSGi Interfaces″ Application Service is required by all Platform Profiles.

Converting a Dynamic Web project to a Client Services Web project

You can convert an existing Dynamic Web project into a Client Services Web project by using the Convert Project to Client Services project wizard. Refer to "Convert Project to Client Services Project Wizard" on page 250 for information on how to use this wizard.

This will retain the existing web application logic of the project, and will add Client Services tooling support.

Note: There is no wizard to convert a Client Services Web project back to a Dynamic Web Project. If you wish to retain the original Dynamic Web Project, you should copy the project before converting it. This can be done as follows:

1. In the Package Explorer or Project Explorer view, right click the project to be copied to display its context menu.
2. Select **Copy**.
3. Right click on an empty space in the project view.
4. Select **Paste**. This displays the Project Copy dialog.
5. Enter a project name for the project copy, and select **OK**.

Client Services Web Application Development

Unsupported tooling features

Various web application features can be selected through the project's Web Project Features properties page. Not all of these features are supported by the WebSphere Everyplace Deployment platform. The following features are not supported:

- EGL Support
- Faces Base Components and Faces Client Framework

  Java Server Faces is supported, as described in "Java Server Faces (JSF) development" on page 30, but these particular features are unsupported.
- Domino SDO Mediator
- WDO Relational database runtime
- Crystal Reports

Accessing resources

The Servlet and JSP specifications do not guarantee that a web application's resources will be represented as files in the host machine's file system. Many web container implementations do expand web applications into the file system, and some existing web applications take advantage of this implementation detail to reference resources as Java Files. However, web applications that target a WebSphere Everyplace Deployment runtime are represented as jar bundles that do not have to be expanded in the file system to run. Because of this, your web application should use the `ServletContext.getResourceAsStream()` API when accessing web application resources. It should not assume these resources will be available as files. The API `ServletContext.getRealPath()` should also not be used as it is implementation dependant. For the WebSphere Everyplace Deployment runtime, it will return null since the web application resources are not expanded in the file system. Again, such resources can be accessed as IO streams through `ServletContext.getResourceAsStream()`.

Using JSP Standard Tag Libraries

The WebSphere Everyplace Deployment platform includes the JSP Standard Tag Libraries (JSTL) version 1.1 as part of the runtime. These libraries will support both Servlet 2.4 / JSP 2.0 compliant applications as well as Servlet 2.3 / JSP 1.2 applications. If your application makes use of JSTL tags, you do not need to include copies of the JSTL libraries in your web application's `WEB-INF/lib` directory.

You should not use the project's Web Project Features properties page to select JSP Tag Libraries. This will incorrectly copy JSTL 1.0 libraries into your project's `WEB-INF/lib` directory. Remove these libraries if this option was selected

Java Server Faces (JSF) development

JavaServer Faces is a technology that helps you build user interfaces for dynamic Web applications that run on the server. The JavaServer Faces framework manages UI states across server requests and offers a simple model for the development of server-side events that are activated by the client. JavaServer Faces is consistent and easy to use. For additional information on JavaServer Faces, refer to the **Developing applications and Web Sites > Developing Web Applications > Web Application Overview > Web Tools Features** section in the Help Contents of the Rational Software Development Platform.

Please refer to the product release notes for any updates to the Java Server Faces development process.

The client platform supports the use of JavaServer Faces based web applications, provided that the required JAR files are included within each web application. The client platform does not include JSF jars as part of the client platform runtime as does WebSphere Application Server v6.0.

To enable the creation of new web applications that use JavaServer Faces, you must use the following project types and options when creating a new web application:

- Client Services Web Project

  You may select either Servlet Version 2.3 or 2.4
- Dynamic Web Project

  You must choose from one of the following combinations:
  - Servlet Version: 2.3 and Target Server: WebSphere Application Server v5.1
  - Servlet Version: 2.3 and Target Server: WebSphere Everyplace Deployment Client Runtime v6.0
  - Servlet Version: 2.4 and Target Server: WebSphere Everyplace Deployment Client Runtime v6.0

  Only these combinations will correctly add the required libraries to your Dynamic Web Project upon the creation of a Faces JSP file.

  Note: The version of the JSP Standard Tag Library (JSTL) that will be added to your project will be based on the 1.0 specification. Refer to "Using JSP Standard Tag Libraries" for more information.

If you have an existing web application that was created with a Target Server: WebSphere Application Server v6.0, and you intend to run this application on a WebSphere Everyplace Deployment Client Runtime, then you will need to perform the following procedures:

1. If you want to preserve your original project environment, make a copy of the project in your workspace.
2. Change the Target Runtime for your project:
   a. Select your project, right click, then select **Properties**.
   b. Select **Server**.
   c. Change the Target Runtime to **WebSphere Everyplace Deployment Client Runtime v6.0**.
3. Create a new web application project following the guidelines above.
4. Copy the set of libraries from the `WebContent\WEB-INF\lib` directory of the new project into the `WebContent\WEB-INF\lib` directory of the existing project. If you are prompted to overwrite the `jsf-api.jar`, `jsf-impl.jar`, or the `jsf-ibm.jar`, select **Yes**.

You should now be able to continue development of your JSF based web application.

Struts development

Struts is a framework of open-source software that can help you build Web applications quickly and easily. It relies on standard technologies such as Java beans, Java servlets, JavaServer Pages (JSP), and XML. Struts encourages application architectures based on the Model 2 approach, which is basically the same as the model-view-controller (MVC) design pattern. For additional information on Struts, refer to the Developing applications and **Web Sites > Developing Web Applications > Web Application Overview > Web Tools Features** section in the Help Contents of the Rational Software Development Platform.

The client platform supports the use of Struts-based web applications. All web applications that intend to use Struts must include the Struts jars within the web application.

The Web tools in the Rational Software Development Platform enable development of Struts-based applications by adding libraries to the `WebContent\WEB-INF\lib` directory of each web application. Perform the following procedure to add the Struts libraries to your project:

1. Right click on your project in the Project Explorer view, and select **Properties**.
2. Select the **Web Project Features** page.
3. Select **Struts**, and then select **OK**.

Securing Web Application resources

Configuring a Web Application

The Web Container supports the declarative J2EE security model. In declarative security the application's web descriptor specifies the application's security policy (roles, access control etc.) without changing the applications code. The following is an example code snippet from a web descriptor that shows the declarative security syntax. This example secures web application resources with `url-pattern=/secure/*` -

```
<security-constraint>
<display-name>myLoginTest</display-name>
 <web-resource-collection>
  <web-resource-name>LoginTest</web-resource-name>
  <url-pattern>/secure/*</url-pattern>
  <http-method>GET</http-method>
  <http-method>PUT</http-method>
  <http-method>HEAD</http-method>
  <http-method>TRACE</http-method>
  <http-method>POST</http-method>
  <http-method>DELETE</http-method>
  <http-method>OPTIONS</http-method>
 </web-resource-collection>
 <auth-constraint>
  <description>Any user</description>
  <role-name>user.anyone</role-name>
 </auth-constraint>
</security-constraint>
<login-config>
 <auth-method>FORM</auth-method>
 <form-login-config>
  <form-login-page>/login.jsp</form-login-page>
  <form-error-page>/error.jsp</form-error-page>
 </form-login-config>
</login-config>
<security-role>
 <role-name>user.anyone</role-name>
</security-role>
```

To configure a web application to use declarative security on the Web Container, the web descriptor must define a list of valid User Admin roles in the `<role-name>` tag. This list of roles can include user and group roles. The above example uses the default User Admin role of `user.anyone`. This means any valid user can be used to log into this web application. The Web Container assumes that all User Admin users store their passwords as a credential with the key ″password″. If no valid users are created with User Admin then the Web Container will not let anyone access the web application resources that have been secured.

Note: Developers may also use programmatic security to control access to a web application. For more information on the Web descriptor, declarative and programmatic security models refer to the Servlet 2.3 specification.

Using the User Admin Service to create users and roles

The Web Container uses the User Admin service to authenticate and authorize requests for secured web application resources. You can use the User Admin API to add, modify, or delete properties and credentials for existing users. The following snippets of code show how you can add a user and delete a user:

Add a user

```
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.osgi.service.useradmin.UserAdmin;

public class MyWebApplication {

/**
 * Used to store reference to UserAdmin service
 */
private static UserAdmin userAdmin = null;

/**
 * Plug-in bundle context
 */
private BundleContext context;

/**
 * Plug-in start method
 */
public void start(BundleContext bc) throws Exception {
 String username = "Joe";
this.context = bc;

ServiceReference ref = bc.getServiceReference("org.osgi.service.useradmin");
userAdmin = (UserAdmin) bc.getService(ref);
useradmin.createRole(username, Role.User);
}

/**
 * Plug-in stop method
 */
public void stop(BundleContext bc) throws Exception {
}
}
```

Delete a user

```
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceReference;
import org.osgi.service.useradmin.UserAdmin;

public class MyWebApplication {

/**
 * Used to store reference to UserAdmin service
 */
private static UserAdmin userAdmin = null;

/**
 * Plug-in bundle context
 */
private BundleContext context;

/**
 * Plug-in start method
 */
public void start(BundleContext bc) throws Exception {

 String username = "Joe";
this.context = bc;

ServiceReference ref = bc.getServiceReference("org.osgi.service.useradmin");
userAdmin = (UserAdmin) bc.getService(ref);
useradmin.removeRole(username);
}

/**
 * Plug-in stop method
 */
public void stop(BundleContext bc) throws Exception {
}
}
```

For more information on the User Admin service, please refer to the OSGi Release 3 specification and the OSGi Javadoc.

Using the Admin Utility for OSGi to create users and roles

With the Admin Utility for OSGi, you can use the User Admin Service to manipulate user definitions. You can add, modify, or delete properties and credentials for existing users.

To add a user:

1. Launch the WebSphere Everyplace Deployment platform.

2. Install the Admin Utility for OSGi feature.
3. Select **Application Open > Admin Utility for OSGi** to run the application
4. Once the application is started, select the **User Admin** label in the left-side frame.
5. Select **Create New User**. The Create New User input fields display.
6. Enter a user name and select **Create User**.

To delete a user:
1. Launch the WebSphere Everyplace Deployment platform.
2. Install the Admin Utility for OSGi feature.
3. Select **Application Open > Admin Utility for OSGi** to run the application
4. Once the application is started, select the **User Admin** label in the left-side frame.
5. Check the check box for the user to be deleted and select **Delete**. You will need to refresh the data before removing the user from the form

To create a group:
1. Launch the WebSphere Everyplace Deployment platform.
2. Install the Admin Utility for OSGi feature.
3. Select **Application Open > Admin Utility for OSGi** to run the application
4. Once the application is started, select the **User Admin** label in the left-side frame.
5. Select **Create New Group**. The Create New Group form displays.
6. Add members to the group by selecting a member in the Available Roles box and choosing either **Basic** or **Required** to move the member to the New Member Roles box.
7. Select **Create Group** to finish creating the group.

Platform integration

com.ibm.eswe.workbench.WctWebApplication

This extension point provides the definition of a web application to be launched.

Since: Enterprise Offering 5.8.0

Configuration markup:

```
<!ELEMENT extension EMPTY>
      <!ATTLIST extension
      point CDATA #REQUIRED
      id    CDATA #IMPLIED
      name  CDATA #IMPLIED>
```

- **point** - Fully qualified identifier of the target extension point
- **id** - ID identifying this instance of the extension point. If the web application is using an `IUrlProvider` implementation to generate the URL, then the id should not contain any decimals ('.')..
- **name** - Name associated with the extension point

```
<ELEMENT DisplayName (#CDATA)>
```

DisplayName - Display Name to use for the application in the **Application > Open** and **Application Switcher** menus. Required.

```
<!ELEMENT Url (#CDATA)>

 <!ATTLIST Url
       provider CDATA #IMPLIED
       local    CDATA #IMPLIED
       secured  CDATA #IMPLIED>
```

Url - The Url text portion of the element specifies either the context root and application specific path for a Local application, or the entire URL for a remote application. Required.

The Url element will contain the following attributes:

- A provider attribute that specifies the name of a class that will return the Url to be displayed. If the provider attribute is present, the text value of the Url element is not required. Any provider must implement the IUrlProvider interface and use the IPageDescriptor API to retrieve application information required to construct the Url.
- A local attribute that indicates whether the content of the Url text portion is intended to be run against the local web container, or is a full URL. Values are true or false. This replaces the Local element that previously existed. The default is true. If the provider attribute is specified, the workbench will do nothing with this value (except for setting it up within the IPageDescriptor implementation)
- A secured attribute used only if local="true" that indicates that HTTPS should be used by the browser to connect to the web application. Values are true or false. The default is false. If the provider attribute is specified, the workbench will do nothing with this value (except for setting it up within the IPageDescriptor implementation)
- The id attribute defined as part of the extension element is required if the provider attribute of the Url is used.

Note: The value of the id attribute must be a simple string with no special characters (e.g. '.').

```
<!ELEMENT Icon (#CDATA)>
```

Icon - Relative path of icon to be used in the **Application > Open** and **Application > Switcher** menus. Optional.

If specified, web application developers should provide both a 16x16 and a 32x32 pixel color image. If the image size is not 16x16 or 32x32 or if the <Icon> element is not specified, then the default web application images will be used. Image name should contain either '16x16' or '32x32' to denote the size of the image.

```
<!ELEMENT BrowserOptions>
 <!ATTLIST BrowserOptions
       browser CDATA #IMPLIED
       showAddressbar    CDATA #IMPLIED
       showToolbar  CDATA #IMPLIED
       showHistory  CDATA #IMPLIED
       showHome  CDATA #IMPLIED
       showPageCtrl  CDATA #IMPLIED
       showPrint  CDATA #IMPLIED
       showBookmark  CDATA #IMPLIED
       userid  CDATA #IMPLIED
      password  CDATA #IMPLIED >
```

BrowserOptions- Configures browser options. Optional

The BrowserOptions element contains the following attributes:

- A browser attribute that specifies the type of browser to use. The supported values are “platform”, “MSIE” and “Mozilla”. The default for Windows is MSIE, the default for Linux is Mozilla.

  Note: Mozilla on Windows is not supported for this release.
- A `showAddressbar` attribute that specifies whether or not the browser address bar displays. The supported values are “true” or “false”. The default is “true”.
- A `showToolbar` attribute that specifies whether or not the browser tool bar displays. The supported values are ″true″ or ″false″. The default is ″true″. If `showToolbar` is set to ″false″, then none of the toolbar buttons (Print, Stop, etc...) will display.
- A `showHistory` attribute that specifies whether or not the browser Back and Forward buttons display. The supported values are ″true″ or ″false″. The default is ″true″.
- A `showHome` attribute that specifies whether or not the browser Home button bar displays. The supported values are “true” or “false”. The default is “true”.
- A `showPageCtrl` attribute that specifies whether or not the browser Stop and Refresh buttons display. The supported values are ″true″ or ″false″. The default is ″true″.
- A `showPrint` attribute that specifies whether or not the browser Print button displays. The supported values are “true” or “false”. The default is “true”.
- A `showBookmark` attribute that specifies whether or not the browser Bookmark button displays. The supported values are “true” or “false”. The default is “true”.
- A `userid` attribute that specifies the `username` to use to replace the `%USERID%` tag in the web application URL.
- A `password` attribute that specifies the `password` to use to replace the `%PASSWORD%` tag in the web application URL.

Examples

1. The following example uses the `Url` element to specify the web application URL, and shows the browser address bar while hiding the Home button:

```
<?eclipse version="3.0"?>
<plugin>
   <extension
     point="com.ibm.eswe.workbench.WctWebApplication">
    <DisplayName>%webapp.name</DisplayName>
    <Url local="true" secured="false">/OrderEntry</Url>
    <BrowserOptions browser="platform"
         showAddressBar="true"
         showHome="false"/>
     <Icon>icons/OEwctwebapp_32x32.gif</Icon>
  </extension>
</plugin>
```

2. The following example specifies a secured web application URL, and removes the browser address bar and toolbar:

```
<?eclipse version="3.0"?>
<plugin>
   <extension

   point="com.ibm.eswe.workbench.WctWebApplication">
     <DisplayName>%webapp.name</DisplayName>
     <Url local="true" secured="true">/OrderEntry</Url>

     <BrowserOptions browser="platform"
          showAddressBar="false"
          showToolbar="false"/>
    <Icon>icons/OEwctwebapp_32x32.gif</Icon>
   </extension>
</plugin>
```

3. The following example uses the Mozilla browser, and removes Print button from toolbar:

```
<?eclipse version="3.0"?>
<plugin>
   <extension      point="com.ibm.eswe.workbench.WctWebApplication">
    <DisplayName>%webapp.name</DisplayName>
    <Url local="true">/OrderEntry</Url>
    <BrowserOptions browser="Mozilla"
    showPrint="false"/>
    <Icon>icons/OEwctwebapp_32x32.gif</Icon>
   </extension>
</plugin>
```

4. The following example shows how to use the `UrlProvider` capability:

```
<?eclipse version="3.0"?>
<plugin>
   <extension
   id=MyApplication   point="com.ibm.eswe.workbench.WctWebApplication">
    <DisplayName>%webapp.name</DisplayName>
    <Url provider="myApplication.myProvider"/>
    <BrowserOptions browser="platform"
        showAddressBar="true"
        showToolbar="true"
        showHistory="true"
        showPageCtrl="true"
        showHome="false"
        showPrint="true"/>
<Icon>icons/OEwctwebapp_32x32.gif</Icon>
   </extension>
</plugin>
```

This extension point will be used to display the menu items that appear in the **Application > Open** menu, and to display the web applications within the Desktop view.

Importing and Exporting Web Application Bundles

Importing a Web Application bundle

You can import a Web Application bundle (WAB) that was exported with source. Refer to “Exporting Client Services projects” on page 166 for information on exporting a bundle.

If the bundle was not exported with source, it will not be eligible for import. Perform the following procedure to import a WAB:

1. Open the Import wizard by selecting **File > Import...** The Import window displays.
2. Choose **WAB file**, and select **Next**. The Import WAB window displays.
3. Choose the WAB file you want to import by selecting **Browse**, or by typing the file name into the WAB file field.
4. Select **Finish**. A project will be created for the imported bundle. The project name will be the bundle’s symbolic name.

Exporting a Web Application bundle

A Client Services Web project can be exported as a bundle by using the OSGi bundle export wizard. Refer to "Exporting Client Services projects" on page 166 for information on how to export a bundle.

WAB Utility

The WAB utility is a command line utility for transforming Web Application Archive (WAR) files into Web Application Bundle (WAB) files that are suitable for running in the WebSphere Everyplace Deployment platform. Note that the web development tools already provide a wizard as well as an ant task for exporting both Client Services web projects and Dynamic web projects as WABs. See "Exporting a Web Application bundle" and "Using Ant tasks to build a deployable bundle" on page 165. These tools should be used when you are dealing with web projects under the WebSphere Everyplace Client toolkit. The WAB utility is a standalone utility that can be used to transform existing WAR files independent of any projects managed by the WebSphere Everyplace Client toolkit.

WAB Utility installation

Perform the following procedure to install the WAB utility:

1. Install a J2SE 1.4.2 JDK on your development machine, and set the environment variable `JAVA_HOME` to the location of this JDK.
2. The WAB utility is delivered as one of the plug-ins installed with the WebSphere Everyplace Client toolkit. It is located in the following directory (where `RATIONAL_HOME` is the directory that your Rational Software Development platform has been installed to):

   ```
   RATIONAL_HOME/sdpisv/eclipse/plugins/com.ibm.pvc.tools.web.translator_6.0.0
   ```

   You can copy this directory elsewhere on your development machine, if desired, or use it in place. For convenience, you should add the WAB utility directory to your system's `PATH` environment variable. This will enable you to invoke the WAB utility scripts without specifying their full path.
3. The invocation scripts for the WAB utility are within the WAB utility directory described in step 2.
   - **wab.bat** - Windows script for invoking the WAB utility
   - **wabc** - Linux script for invoking the WAB utility

   Refer to "WAB Utility usage and parameters" for details on how to use the WAB utility.

WAB Utility usage and parameters

The WAB utility is invoked through a script. On Windows systems, this script is `wab.bat`. On Linux systems this script is wabc. The following will use the Windows `wab.bat` script in examples, and assumes that the WAB utility directory has been added to the system's PATH environment.

The WAB utility performs the following transformations on your web application:

- All JSP files are translated to their underlying servlet classes. In addition to the standard J2SE libraries, the utility automatically adds required`javax.servlet*` class libraries to the class path during translation. Any other classes referenced by the application's JSPs, that are not part of the application itself (through `WEB-INF/classes` or `WEB-INF/lib`), must be specified through the utility's `–classpath` parameter.

- If an OSGi compliant manifest file does not exist in the web application, one will be added. It will have the necessary package dependency statements for supporting web applications. If your application is referencing additional external packages, you will need to include your own custom `META-INF/MANIFEST.MF` manifest file that includes these package dependencies through either the Import-Package or `Require-Bundle` fields. Note that the utility will augment an existing manifest file to contain any missing dependencies, and will not overwrite any preexisting entries.

WAB Utility examples

The simplest use of the WAB script only specifies the war file to be translated:

```
wab myweb.war
```

The above will create a `myweb.jar` file in the directory from which the tool was invoked.

You can specify the target name and location for the WAB JAR using the `-o` parameter:

```
wab myweb.war -o /myruntime/eclipse/plugins/myweb.jar
```

You can add additional libraries to the translation class path using the `-classpath` parameter:

```
wab myweb.war -classpath myLib1.jar;myLib2.jar
```

You can use the `-g` option to specify compilation with debug information.

WAB Utility parameters

WAB utility invocation has the following form:

```
wab  <war file>  [ Options ]
```

The following table describes the options parameters available to the WAB utility.

Table 5. WAB Utility options parameters

Option	Description
<war file>	File name of the war file to be transformed into a WAB file. This must be a Servlet 2.3 or 2.4 compliant war file.
-contextpath <path>	Specify the context path for the web application. By default, the base name of the output file is used for the context path. For example, the default context path for `inventory.jar` is `/inventory`.
-classpath <classpath>	Augment the class path to be used for the JSP file compilation. The WAB utility automatically includes the `javax.servlet.*` packages on the class path.
-o <output file>	File name of the resulting WAB file. The default name is the base <war file> name with .JAR file extension, placed in the same directory as <war file>.
-includesource	When specified, the WAB file will include the original JSP source files. By default, JSP files are removed from the WAB, since they are translated to their underlying servlet classes.

Table 5. WAB Utility options parameters (continued)

Option	Description
-g	When specified, JSP files are compiled with debug information.
-id <name>	Specify the bundle symbolic name. By default, this is the base name of the output file.

Web Container Logging

The Web Container will log all messages using the OSGi Log service. The WebSphere Everyplace Deployment logger plug-in will redirect these messages using JDK 1.4 logging to `<USER_HOME>\IBM\RCP\<INSTALL_ID>\<USER_NAME>\logs\rcp.log.*`.

The following table shows the mapping between the Web Container log levels, OSGi log levels and the levels used by JDK 1.4 logging:

Table 6. Web Container log level mapping

Web Container Log Level	OSGi Log Level	java_util.Level
ERROR	ERROR	SEVERE
WARNING	WARNING	WARNING
INFO	INFO	INFO
DEBUG	DEBUG	FINEST
EVENT	DEBUG	FINEST
ENTRY	DEBUG	FINEST
EXIT	DEBUG	FINEST

Configuring the Web Container Logging

Web application developers can configure the Web Container logging mechanism using the `trace.properties` file located in the RAS (`com.ibm.pvc.ras`) plug-in directory. The `trace.properties` file allows developers to debug the Web Container at three levels (in order of precedence) – Web Container level, Web Container component level, and Web Container class level. Please note that whether or not anything is logged depends on whether or not the classes you are debugging have the requisite hooks.

Web Container level properties

- Debug.ALL
- Entry.ALL
- Event.ALL

Note: If these are set to `true`, they will override the rest of the settings in this file.

Note: Error, Warning and Info level messages are always logged.

Examples:

```
Debug.ALL=true
Entry.ALL=true
```

Web Container component level properties

Component-level syntax is `<Level>.<component>`. The allowed components are `Webcontainer`, `HttpSession`, and `HTTP_Transport`.

Examples:

```
Debug.Webcontainer = false
Debug.HttpSession = true
Debug.HTTP_Transport = true
```

Developing messaging applications

IBM WebSphere Everyplace Deployment for Windows and Linux provides both enterprise class messaging through the Java Message Service (JMS), and embedded messaging using WebSphere MQ Everyplace (MQe) and the MicroBroker technical preview with the MQ Telemetry Transport (MQTT) Java client APIs. MQe provides a point-to-point JMS provider, which enables Java developers to leverage the JMS APIs to send and receive messages from the WebSphere Everyplace Deployment client runtime.

Technology overview

Messaging is intended to enable a wide variety of computers to exchange information. One of the main benefits of messaging is to 'decouple' a sending application from a receiving application. This decoupling provides a very powerful abstraction, enabling the exchange of information to be independent of manufacturer, application, operating system or connectivity reliability.

Traditionally, messaging is between large computers and a server. However, with the advent of Java messaging implementations, now a new class of device interoperates with the messaging infrastructure. This provides opportunities for an enterprise to broaden the reach of its networks.

Along with the transmission of simple messages, messaging is useful for transactional updates, or where intermediate data updates or data ordering is required. Messages containing the complete update can be sent to a server, where transaction managers can coordinate the update of multiple resources. Messaging can also be paired with synchronization technology, such that transactions are sent by messages, and the resulting database updates distributed back to the client through synchronization.

Messaging also can be used effectively in a disconnected environment (areas where connectivity is not reliable), since a local queue manager is available to contain messages until the connection to the server infrastructure is reestablished. After the connection is available, the queued messages are transferred to the messaging server for further action.

Messaging, irrespective of the particular product or product group, is separated into two main categories:

- Point-to-point messaging
- Publish and Subscribe messaging

To take full advantage of the messaging capabilities of WebSphere Everyplace Deployment, it is crucial to understand the differences in these two messaging types.

Publish and Subscribe – Topics, subscriptions and brokers

The application programming model for a publish/subscribe messaging paradigm consists of the following:

- **Subscribers**: Express an interest in messages containing information on a particular subject.

- **Subscriptions**: Contain records of registered interest with the Broker from a subscriber.
- **Brokers**: Act as go-betweens, receiving messages from publishers and comparing them to the needs of subscribers. A message is delivered to all subscribers that have expressed an interest in the subject of the message.
- **Publishers**: Generate messages containing information about a particular subject. Messages are sent to a broker.

The publish and subscribe messaging paradigm provides one-to-many messaging.

The following graphic shows the topology for a publish and subscribe messaging paradigm:

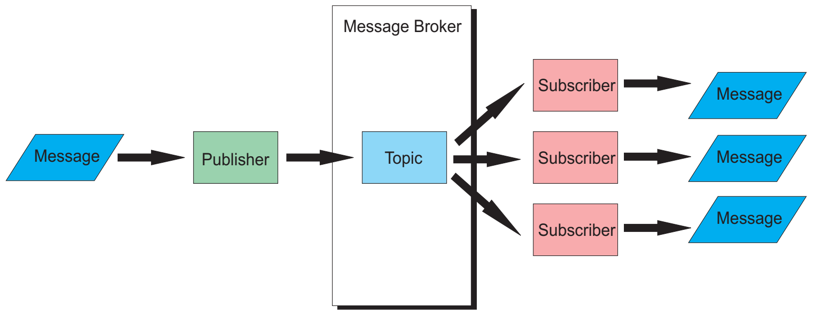


Figure 8. Publish and subscribe messaging

Point-to-point – queues and queue managers

Queue managers handle queues that store messages. Applications communicate with a local queue manager, and get or put messages to queues. If a message is put to a remote queue (a queue owned by another queue manager), the message is transmitted over connections to the remote queue manager. In this way, messages can hop through one or more intermediate queue managers before reaching their final destination. You can configure queue managers with or without local queuing. All queue managers support synchronous messaging operations. A queue manager with local queuing also supports asynchronous message delivery.

The point-to-point messaging paradigm provides one-to-one messaging. In other words, messages are consumed by only one receiver, unlike publish-and-subscribe where messages are consumed by multiple receivers.

The following graphic shows the topology for point-to-point messaging:

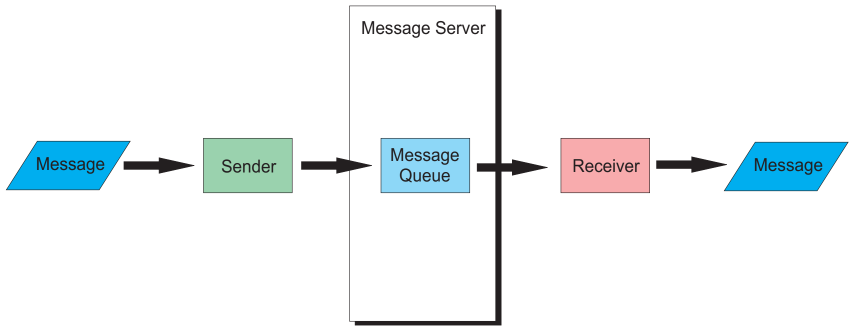


Figure 9. Point-to-point messaging

Java Message Service

Java Message Service (JMS) is the standard Java API for messaging. It supports the two messaging categories: point-to-point messaging and publish/subscribe messaging. JMS is defined as part of the Java 2 Enterprise Edition 1.3 and 1.4 definitions. It defines a package of Java interfaces, which allows for provider-independence, but does not necessarily allow for provider interoperability.

The JMS APIs are provided with the WebSphere Everyplace Deployment runtime. This runtime also includes a point to point JMS provider based on MQe messaging. The MQe classes for JMS are a set of Java classes that implement the JMS interfaces to enable JMS programs to access MQe systems.

There are several benefits to using JMS as the API to write MQe applications. Some advantages are derived from JMS being an open standard with multiple implementations. Using an open standard provides the following benefits:

- The protection of investment, both in skills and application code
- The availability of people skilled in JMS application programming
- The ability to plug in different JMS implementations to fit different requirements

IBM has several implementations of JMS. Interoperability is provided between them.

More information about the benefits of the JMS API is available at: http://java.sun.com

The JMS application is written to use only references to the interfaces in the `javax.jms` package. All vendor-specific information is encapsulated in implementations of the following JMS administered objects:

- `QueueConnectionFactory`
- `TopicConnectionFactory`
- `Queue`
- `Topic`

Note: The WebSphere Everyplace Development client runtime only provides an MQe point-to-point JMS provider, which supports the `QueueConnectionFactory` and `Queue` objects. The WebSphere Everyplace Development runtime does not provide a JMS provider supporting `TopicConnectionFactory` or `Topic` objects.

These JMS administered objects are stored in a Naming Directory Interface (JNDI) namespace. A JMS application can retrieve these objects from the namespace and use them without needing to know which vendor provided the implementation; in this case, the MQe JMS provider.

WebSphere MQ Everyplace

WebSphere MQ Everyplace (MQe) is a member of the IBM WebSphere MQ family of business messaging products. It exchanges messages with various applications, providing once and once-only assured delivery leveraging the point to point message paradigm.

MQe provides an integrated set of security features enabling the protection of message data both when held locally and when being transferred.

With *synchronous* message delivery, the application puts the message to MQe for delivery to the remote queue. MQe simultaneously contacts the target queue and delivers the message. After delivery, MQe returns immediately to the application. If the message cannot be delivered, the sending application receives immediate notification. MQe does not assume responsibility for message delivery in the synchronous case (non-assured message delivery).

With *asynchronous* message delivery, the application puts the message to MQe for delivery to a remote queue. MQe immediately returns to the application. If the message can be delivered immediately, or moved to a suitable staging post, it is sent. If not, it is stored locally. Asynchronous delivery provides once and once-only assured delivery. After the message is provided to MQe, control is returned to the application. MQe next takes responsibility for assured delivery of the message. Delivery occurs in the background allowing the application to carry on its processing.

MQe also has the ability to exchange messages with WebSphere MQ host queue managers and brokers. To do this, configure a MQe queue manager with bridge capabilities. Without the bridge, a queue manager can communicate directly only with other MQe queue managers. However, it can communicate indirectly through other queue managers in the network that have bridge capabilities.

As mentioned previously, a point-to-point JMS provider is included with MQe. Initially MQe must be bootstrapped using the supplied connection factory. From then on, standard JMS APIs can be used.

MicroBroker – Technical Preview

MicroBroker is a very small footprint, 100% Java message broker, capable of running in resource-constrained environments. It is suitable for embedding in applications and solutions that have a need for messaging, notification and event services.

MicroBroker supports the publish and subscribe messaging paradigm. It provides a messaging infrastructure, which enables lightweight messaging clients to communicate with each other, on one host or across a network, as well as with enterprise brokers through its bridging capabilities.

MicroBroker uses the MQ Telemetry Transport (MQTT) protocol over TCP/IP.

As part of the technical preview capabilities, MicroBroker allows for the bridging of messages to WebSphere MQ servers. The IBM MicroBroker Technical Preview Deployment Guide (available in “Related documentation” on page 48) provides detailed information on the MQ bridging capability. This is accomplished by leveraging a set of MQ JAR files from a WebSphere MQ installation. The usage of these MQ JARs is subject to the same Technical Preview status as the MicroBroker component. In order to provide access to the required MQ packages in our WebSphere Everyplace Deployment environment, the JARs can be packaged as an Eclipse plug-in using our WebSphere Everyplace Client Tooling. In order to provide the MQ jars for the platform, we suggest the following steps:

1. Create a Client Services project choosing an appropriate project name, such as `com.ibm.mq`.
2. Deselect the **Create a Java project** check box. Select **Next**, and then **Finish**.

3. Copy the `com.ibm.mq.jar`, `com.ibm.mq.jms.jar`, and the `connector.jar` from the `<Websphere MQ installation>/Java/lib` directory into the root of the new Client Services project.
4. Open the `build.properties` file and select the three JARs listed in step 3 in the Binary Build section of the `build.properties` editor.
5. Update the `META-INF / MANIFEST.MF` file as shown:

```
Manifest-Version: 1.0
Bundle-Name: IBM WebSphere MQ
Bundle-Version: 1.0.0
Bundle-SymbolicName: com.ibm.mq
Import-Package: javax.jms
Bundle-ClassPath: com.ibm.mq.jar,com.ibm.mqjms.jar,connector.jar
Export-Package: com.ibm.mq.jms
Provide-Package: com.ibm.mq.jms
```

   This Client Services project can now be used in your development environment or exported for use in a client runtime. Please refer to "Using the IBM WebSphere Everyplace Client Toolkit" on page 169 for more information on performing these tasks.

MQ Telemetry Transport – Technical Preview

MQ Telemetry Transport (MQTT) is an open protocol designed for resource-constrained devices and networks, providing publish and subscribe messaging over TCP/IP. Clients operate in conjunction with a suitable message broker, such as the MicroBroker, WebSphere Business Integration (WBI) Message Broker, or WBI Event Broker, which are responsible for the syndication of messages. As a wire protocol, no device API is mandated; rather, the implementations expose a simple semantic including: connect/disconnect, publish, and subscribe/unsubscribe. Provision is made for assurance of message delivery using one of three levels of service; fire and forget, at least once, and exactly once.[2] The Last Will and Testament feature allows abnormal disconnection of a client to be detected, and interested parties to be informed.

By minimizing the requirement on network bandwidth, it is practical to use in wide area networks, which typically have lower link speeds than wired networks. This facilitates not only using MQTT for the collection of data, but also for the presentation of data on handheld devices.

A Java client implementation of the MQTT wire protocol is provided to simplify MQTT client programming. For more information, see: http://www.mqtt.org

WebSphere MQ Everyplace and MicroBroker comparison

While MQe and MicroBroker products are similar in that they provide embedded messaging capabilities; their specific set of features might make a better choice for certain client applications.

Table 7. MQ Everyplace and MicroBroker comparison

	WebSphere MQ Everyplace	**MicroBroker and MQTT**
JMS provider included	Yes. point-to-point provider.	No
Messaging paradigm	point-to-point (queue-based)	Publish and Subscribe (topic-based)

2. Fire and forget is also known as 'at most once'.

Table 7. MQ Everyplace and MicroBroker comparison (continued)

	WebSphere MQ Everyplace	MicroBroker and MQTT
Implementation type	Java-based (platform-independent) implementation	Java-based (platform-independent) implementation
Bridging	Supports bridge to MQe and WebSphere Business Integration Message and Event Brokers	Support bridge to WebSphere MQ, and WebSphere Business Integration (WBI) Message and Event Brokers
Wire protocol	MQe- specific	MQTT standards based
Separate small footprint client	No	Yes; Java and 'c'
QOS for message delivery	At least once and exactly once.	Fire and forget, at least once, and exactly once
Local and remote queues	Yes	No
Build in security	Yes	No

This is not an exhaustive comparison of the two products. See the product documentation for more complete information about these products.

Client Services platform profile components

IBM WebSphere Everyplace Client Toolkit provides Client Services platform profile support in the Rational tooling environment. These platform profiles simplify the creation and configuration of messaging application projects, enabling you to select the target-embedded messaging APIs, and provide automatic management of the requisite messaging libraries. When developing a messaging application, you can select any of the Client Services platform profiles for your Client Services project. The following table provides a list of tasks and the appropriate application service selections for each profile in a Client Services project.

Table 8. Client Services project tasks

Task	Application Services
MQe programming	WebSphere MQ Everyplace
JMS point-to-point messaging	WebSphere MQ Everyplace using Java Messaging Service (JMS)
MQTT programming	WebSphere MQ Telemetry Transport
MicroBroker programming	MicroBroker

Related documentation

WebSphere MQ Everyplace

For more information about MQe, see the following:

WebSphere_MQe.pdf

MQ Telemetry Transport Technical Preview

Documentation is also available on the MQTT Web site at: http://mqtt.org

MicroBroker Technical Preview

For more information about MicroBroker, see the following:

MicroBroker_techpreview.pdf

Developing the application user interface

This section explains the technology associated with constructing an application user interface. Topics include:

- An overview of the technology associated with application user interfaces
- An overview of the tasks and artifacts for a user interface
- Some guidelines to help incorporate your application into the overall client platform

Technology overview

Eclipse

The Eclipse 3.0 SDK provides an open framework to enhance functionality to the Integrated Development Environment (IDE). These features (or plug-ins) are easily versioned and dynamically installed or updated without restarting the IDE. Software developers using Eclipse have often wished for a similar model for their desktop applications. With previous versions of Eclipse, this was possible but difficult, especially when heavily customizing menus, layouts, and other user interface elements.

Eclipse Version 3 introduces the Rich Client Platform (RCP), a refactoring of the fundamental parts of the UI, enabling RCP to be used as a general-purpose application platform. Its internals are the same OSGi run time and GUI toolkit provided by the Eclipse IDE, but now these are easily used by application developers to provide robust, customizable, and portable Java applications. Because of its Eclipse open source license, you can use the technologies that went into Eclipse to create your own commercial-quality programs. The GUI toolkits used by Eclipse RCP are the same used by the Eclipse IDE and enable applications with optimal performance that have a native look and feel on any platform that they run on.

The client platform provides the default product and workbench. The workbench provides the default look-and-feel that incorporates the base menus, images, and application launcher. Application developers can create new applications that run within the confines of the platform. The workbench provides the facilities that developers can use to enable applications to be launched.

There are two main application user interface types that can be hosted within the platform. Web application based user interfaces are displayed within an embedded browser view that is part of a predefined perspective provided by the platform. The web application can either be locally hosted within the provided web container environment, or be running on a remote server. For more information on creating web applications, refer to “Developing Web applications” on page 25.

The second main application user interface type is an application built using Java-based widgets. In order to contribute to the workbench, applications must provide a perspective and declare the perspective to the workbench using the `WctApplication` extension point. The perspective is a standard Eclipse perspective and is composed of one or more views (or editors). Views (and editors) are constructed from UI widgets such as tables, buttons, text fields and more. Wizards and dialogs can also be created by the application to perform specific tasks.

Wizards and dialogs can be launched by widgets (buttons) within a view, or from various menu bars that are available within the platform.

UI toolkits

The following UI toolkits are used by the Eclipse IDE and plug-ins, and work equally well for RCP applications.

The Standard Widget Toolkit (SWT) provides a completely platform-independent API that is tightly integrated with the operating system's native windowing environment. Java widgets actually map to the platform's native widgets. This gives Java applications a look and feel that makes them virtually indistinguishable from native applications. In cases where native function is not provided, the SWT emulates it in a manner in keeping with the platform's normal look and feel. This toolkit overcomes many of the design and implementation trade-offs that developers face when using the Java Abstract Window Toolkit (AWT) or Java Foundation Classes (JFC). AWT gives the least common denominator approach and is therefore functionally limited. JFC is more flexible, but because all widgets are painted by the toolkit, JFC always seems to have trouble precisely emulating a native look and feel.

The JFace toolkit is a platform-independent user interface API that extends and interoperates with the SWT. This library provides a set of components and helper utilities that simplify many of the common tasks in developing SWT user interfaces. For example, it provides the dialogs, wizards, and rich text editors used by the Eclipse IDE. JFace also has tables and trees that utilize a model view controller (MVC) architecture to separate data access login from data display logic. JFace also provides the mechanisms by which plug-ins programmatically contribute to the workbench, which is further discussed in the next topic.

Visual Editor for Java

The visual editor for Java is a code-centric Java editor that helps you design applications that have a graphical user interface (GUI). The visual editor is based on the JavaBeans™ component model and supports visual construction using the Standard Widget Toolkit (SWT), the Abstract Window Toolkit (AWT), or Swing. A developer can use the visual editor for Java to create SWT composites. These SWT composites can be used inside of Eclipse views and perspectives and used in WebSphere Everyplace Deployment.

Understanding the WebSphere Everyplace Deployment User Interface

User interaction in the WebSphere Everyplace Deployment 6.0

The default user interface provided for the Websphere Everyplace Deployment is similar in appearance and action to other user interfaces, such as Microsoft Windows, Macintosh, and Motif. Users employ a ″selection/action″ model to interact with it.

In a selection/action model, selection pertains to each view and is independent of other selections in other views. The view retains what is called selection memory. For example, when a user selects one or more items in a list in one view, and goes to a different view (in a different plug-in or the same one), and then returns to the list in the original view, the selected items are still selected. This selection model helps users remember where they were and what they were doing in the view.

Inactive selection refers to the display of a previous selection at the same time that the active selection is displayed. All actions only take place on active selections, but continuing to display the inactive selection helps users remember the choices they have made.

User interface organization

The following figure illustrates the organization of the user interface.

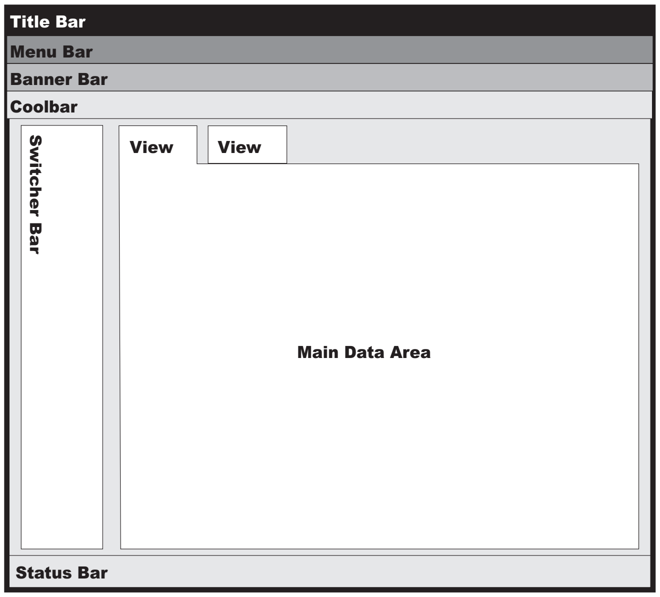


Figure 10. User interface organization

The following parts of the Websphere Everyplace Deployment user interface are displayed by default:
- Title bar
- Menu bar
- Banner bar
- Status bar
- Switcher bar

The main data area contains only a default image when the client platform starts. Once applications have been opened, then the views associated with the application will be displayed.

Title bar

The Title bar displays the program title and icon. The displayed information is part of the branding elements that can be changed. For information on changing the program title, refer to “Managing client configurations” on page 222.

Menu bar

The menu bar contains the set of actions that have been provided by either the default workbench or by other applications.

Usability guidelines recommend that all features of an application be available from the menu items on the menu bar. They also can be available from buttons or context menus. Context menus are displayed when a user right-clicks the background of a user interface component, such as a view or document.

In general, menus should display all menu items that are applicable to the current view.

If	Then
A menu item is not applicable to what is currently selected, but the user can take an action while in the same tab or window to enable the menu item	That menu item should appear dimmed
A menu item is not applicable to what is in the view	That menu item should be hidden

Pull-right menus should always be enabled, even if none of the items in the pull-right menu are available. Users should be able to view the contents of the pull-right menu, even if none of the actions are available.

Banner bar

The banner bar can optionally display a graphic and the application name. The configuration and graphical image used by the banner bar can be changed. For information on changing the program title, refer to "Managing client configurations" on page 222.

Switcher bar

The switcher bar is a vertical bar on the left side of the client window that lists the running applications, each represented by an icon, from which users can select an application to switch to.

When a user clicks an application icon, the client switches to the application in the main data area. Each application represented on the switcher bar has a perspective defined for it. The perspective defines how to populate the window. The switcher bar provides a way for a user to switch from one perspective to another. It prevents multiple applications from having to share a single perspective. As a result, it allows for a cleaner, less cluttered user interface.

In addition to using the switcher bar, users can switch between open applications through the keyboard shortcuts **Ctrl+F8** or **Shift + Ctrl + F8**. The icons on the switcher bar are displayed in a flat list only; no nesting is supported.

Data area

The primary data area of the user interface contains the perspectives and views associated with an application.

Coolbar/Toolbar

The Coolbar/Toolbar area is used to display icons for actions that are available. Usability guidelines suggest that all actions available on the coolbar or toolbar are also available via actions on the menu bars. The Coolbar/Toolbar area is optionally displayed and is configurable by the administrator. Users can also elect to display or hide the Coolbar/Toolbar area.

For information on changing the Coolbar/Toolbar display, refer to "Managing client configurations" on page 222.

Status bar

The status area can be used by applications to display information regarding the status of their application. The components in the status area are associated with a particular perspective and view.

The status bar is optionally displayed and is configurable by the administrator. For information on changing the Status bar display, refer to "Managing client configurations" on page 222.

Creating a simple Rich Client Platform application

The Rich Client Platform is the minimal set of Eclipse SDK plug-ins needed to build an application with a UI. However, rich client applications are free to use any API necessary for their feature set, and can require any plug-ins beyond the bare minimum. The key differentiator of a rich client application from the standard SDK workbench is that the application is responsible for defining which class should be run as the main application. Your RCP application is an Eclipse plug-in. As such, you can use a plugin.xml file to specify your application's start-up class, which creates the workbench window.

In order to create a Java-UI application that can be contributed to the client platform, there is a minimal set of objects that must be created:

- A plug-in for the application
- A view to display some data
- A perspective to contain the view
- An extension point declaration of the application for WebSphere Everyplace Deployment

Note: The above objects are enough to run an application within the development environment. In order to deploy and install the application, a feature and an update site are also required.

The following procedure illustrates how to create an application that can be contributed to the client platform. In these steps, you will make use of the Plug-in Development Environment Plug-in Project, as it provides a set of templates that will reduce the number of steps necessary to build the application.

1. Start Rational Software Development Platform.
2. Create a plug-in project to contain the application that you are providing. You will use a template that provides a view in order to quickly build an application. Later, you can modify the code generated by the template to meet your own requirements.
   a. Select **File > New > Other > Plug-in Development > Plug-in Project** to create the initial project, then select **Next**.

      Note: If you do not see the Plug-in Development entry as one of the selections, select the **Show all wizards** option to enable the display of all of the new project wizards. Selecting a new Plug-in Project will cause a dialog to display inquiring whether you want to enable Eclipse Plug-in Development. You will need to enable this capability to allow appropriate choices to be populated on menus.
   b. On the Plug-in Project panel, enter a name for your plug-in project, such as `MyApplication`. You can use the defaults already entered on the rest of the panel. Select **Next**.

c. Use the default information on the Plug-in Content panel. Select **Next**.

d. On the Templates panel, check **Create a plug-in using one of the templates**. Then select **Plug-in with a view**. Then select **Finish**.

e. A project called `MyApplication` will now be created in your workspace. Selected files will have already been created within the project based on your selections to create a plug-in with a view. The `MyApplication` and `MyApplication.views` packages will have been created in the `src` folder.

3. Create the initial Java class for the perspective for the application. This class organizes the view created by the template.

   a. First, create a package to contain your perspective:

      1) Highlight the `src` folder in your plug-in project.

      2) Select **File > New > Package**. Enter `MyApplication.perspectives` as the Name, then select **OK**.

   b. Now, create the Java class for the perspective:

      1) Select the **MyApplication.perspectives** package that you just created.

      2) Select **File > New > Class**.

      3) Enter a Name for the class, such as `MyPerspective`.

      4) The superclass will remain `java.lang.Object`.

      5) Your class will implement the `org.eclipse.ui.IPerspectiveFactory` interface. Select the **Add...** button next to Interfaces. Begin typing `IPerspectiveFactory` and the panel will complete the name for you as you type. Select **OK**.

      6) Select **Finish** to complete your class creation.

   c. You will need to add some code to the class that you just created in order to layout the view created by the template:

      1) Edit the `MyPerspective.java` file.

      2) Locate the method `createInitialLayout` and replace the method with this code:

```
public void createInitialLayout(IPageLayout layout) {
  String editorArea = layout.getEditorArea();
  layout.addView( "MyApplication.views.SampleView",
            IPageLayout.BOTTOM, 0.7f,editorArea);
  layout.setEditorAreaVisible(false);
 }
```

      This code adds the view created during the project creation to this perspective.

      3) Save and close the Java file.

4. Now add an extension point to the plug-in to define the perspective that you just created:

   a. Open the `plugin.xml` file contained in your project.

   b. Select the Extensions tab within the editor. You should already see some extensions defined, such as `org.eclipse.ui.views` and `org.eclipse.ui.perspectiveExtensions`.

   c. Select **Add...**, then select **org.eclipse.ui.perspectives**, then **OK**.

   d. Once `org.eclipse.ui.perspectives` has been added to the All extensions list, right click on the entry, then select **New > perspective**.

   e. The Extension Element Details information will display. Use the default information for the id and name.

   f. Select the **Browse...** button next to the class and select the **MyApplication.perspectives.MyPerspective** class that you created in Step 3.

5. Create the `WctApplication` extension point to enable contribution of the application to the client platform
   a. Select **Add...** again, uncheck the box for **Show only extension points** from the required plug-ins, then select **com.ibm.eswe.workbench.WctApplication**, then **OK**.
   b. Select the **plugin.xml** tab in the editor.
   c. Replace the content:
   ```
   <extension
                point="com.ibm.eswe.workbench.WctApplication">
           </extension>
   with
           <extension id="MyApplication"
         point="com.ibm.eswe.workbench.WctApplication">
         <DisplayName>MyPerspective</DisplayName>
         <PerspectiveId>MyApplication.perspective1</PerspectiveId>
         <Version>1.0.0</Version>
      </extension>
   ```
   d. Save and close the `plugin.xml` file.

At this point, you can launch the platform from the workbench including this application. **Using Application > Open** will show `MyApplication` as one of the selectable applications. Refer to "Debugging and testing applications" on page 157 for more information on how to launch the client platform.

These are the basic steps towards creating the initial application. Once you have the basic application created, you can continue to enhance your application, adding additional views, menus and menu items, preference pages, dialogs, messages, and more. You can use the same plug-in that you created above, or you can separate your user interface logic into additional plug-ins, depending upon your requirements.

In step 5, you implemented an extension point that defined the perspective ID to contribute to the WebSphere Everyplace Deployment workbench. For more information on the `WctApplication` extension point (and the `WctWebApplication` extension point), refer to "Extension points reference" on page 213.

Extending the capabilities of your application

There are common conventions that have been adopted in order to construct applications that are intuitive and easy to use. By following the conventions and guidelines, you will enable your users to become quickly productive, and will reduce the training required and the frustrations experienced in using the application.

The subsections of this chapter focus on specific areas with suggestions and guidelines in how to build your application to work in a pluggable, cooperative environment. In addition, since WebSphere Everyplace Deployment 6.0 is based on Eclipse, you can also refer to the Eclipse User Interface Guidelines. To view the Eclipse User Interface Guidelines, visit: http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html.

Adding and contributing menus

The views and editors that are created and displayed will typically provide a visual representation of data for the users. In order to control the application environment, such as opening or closing views, or performing specific actions, such as syncing data, you will probably want to consider creating menu and menu items to enable access to the actions.

Menu contributions

You can make a feature you have created available to users by adding it to the menu bar as a menu item. You can create and contribute menu items by implementing one of the following contribution types:

- **Global menu contributions** -- These menu contributions persist across every application. These menu items are universal and can be used from any context. These menu items are also ″retargetable,″ so your application can write code to retarget these global menu items, such as Cut, Copy, and Paste, for application-specific purposes.
- **Part-associated action set contributions** -- These menu contributions are specific to a single application or view. By default, most menu items are local, meaning that if the view associated with those menu items is not displayed, those menu items are not displayed. Contributing view-specific menu items enables you to create discrete menu items for a page of your application that are not shared with other products on the client.

Global menu contributions: You can add a menu item to one of the WebSphere Everyplace Deployment global menus using the `org.eclipse.ui.actionSets` extension point. You can also associate a new action set to a specific perspective or view to ensure that the menu contributions defined in the action set appear in the user interface only when the specified view is active using the `org.eclipse.ui.actionSetPartAssociation` extension point. See the *Platform Plug-in Developer's Guide* for more information on part associations.

For details on the ids used by the WebSphere Everyplace Deployment Top Level Menus, refer to “WebSphere Everyplace Deployment top level menus” on page 211.

If actions are defined by the extension points within plug-ins, but are not associated with a particular perspective, then the menu items/actions will be displayed at all times. By associating menu items and actions with a particular perspective through `perspectiveExtension` extension points, the menu items and actions will be displayed only when the application's perspective is displayed.

The **File** menu provided by the workbench contains entries for Preferences and Exit. Usability guidelines suggest that Exit always be the last menu item. The Preferences action is provided by the workbench, and launches a dialog that aggregates all provided preference pages.

Part-associated action set menu contributions: In addition to having the global menus, each product contributes its own menu items on the existing menus. A product can also contribute entire menus to the menu system.

Context menus: A context menu is the menu that is displayed when a user right-clicks the background of a user interface component, such as a view or editor. Context menus should repeat pertinent menu items that are available on the menu bar or pertinent actions on a dialog box triggered from a menu item. The contents of the context menu must change to be appropriate to the object that has focus or the object that is selected.

A context menu item should not be the only way a user accesses a piece of functionality. All context menu items must have access keys for accessibility.

Indicate the access key by underlining the key text in the menu item label. If possible, use the same access key as the one used for the menu item in the menu bar.

You should give context menus to the following objects:
- Objects in a navigator view, such as folders or document libraries
- Items selected in a list.

Icons in menus: If a menu item is represented by a toolbar button with an icon, include that icon on the menu. If your application does not have a toolbar or the toolbar does not include icons, do not include an icon on the menu. Include icons to the left of the menu items.

Creating a top-level menu

You can write code that performs a task and enable users to trigger that task from a corresponding menu item in the user interface by creating a menu item contribution and adding it to the application menu. The File, Application, View and Help menus appear at the top level. You can contribute a menu item that displays at the top-level as well.

The following steps make use of the templates provided with the Plug-in Development Environment to show you how to create an action, an actionSet, and a top-level menu.

1. Start your Rational Software Development Platform.
2. Create a plug-in project to contain the menu and action that you are providing. You will use a template that creates a Java class for the action, and adds all of the appropriate extension points. Later, you can modify the code generated by the template to meet your own requirements.
   a. Select **File > New > Other > Plug-in Development > Plug-in Project** to create the initial project, then select **Next**.
   b. On the Plug-in Project panel, enter a name for your plug-in project, such as `MyAction`. You can use the defaults already entered on the rest of the panel. Select **Next**.
   c. Use the defaults on the Plug-in Content panel. Select **Next**.
   d. On the Templates panel, check **Create a plug-in using one of the templates**. Select **Hello, World**. Then select **Finish**.
   e. A project called `MyAction` will now be created in your workspace. Selected files will have already been created within the project based on your selections to create a plug-in with a view. The `MyAction` and `MyAction.actions` packages will have been created in the `src` folder.

At this point, you can launch the platform from the workbench including this plug-in (for more information, refer to "Debugging and testing applications" on page 157). A new top level menu, Sample Menu, will be displayed along with File, Application, View and Help. This menu will contain an action labeled Sample Action.

As you launch the platform, you will notice that regardless of the application that you open, the same Sample Menu always appears. If you want to change this menu so that it appears only with a particular application, you will need to create a perspectiveExtension. To illustrate these steps, the following steps assume that the Simple Rich Client Platform application was created in "Creating a simple Rich Client Platform application" on page 55.

1. Open the `plugin.xml` file for the `MyAction` project you just created

2. Now Add an extension point to the plug-in to define the perspective Extension that you want to create.
   a. Open the `plugin.xml` file contained in your `MyAction` project.
   b. Select the Extensions tab within the editor. You should already see the `org.eclipse.ui.actionSets` defined.
   c. Select **Add...**, then select `org.eclipse.ui.perspectiveExtension`, then select **OK**.
   d. Once `org.eclipse.ui.perspectiveExtensions` has been added to the All extensions list, right click on the entry, then select **New > perspectiveExtension**.
   e. The Extension Element Details information will display. Enter `MyApplication.perspective1` as the target id. (This is the perspective created in "Creating a simple Rich Client Platform application" on page 55).
   f. Now select the **MyApplication.perspective1 perspectiveExtension**. Right click and select **New**, then select **actionSet**.
   g. The Extension Element Details information will display. Enter `MyAction.actionSet` as the target id.
   h. Expand the **org.eclipse.ui.actionSets**, and select the **Sample Action Set**.
   i. The Extension Element Details information will display. Change the value of visible from **true** to **false**.
   j. Save and close the `plugin.xml` file.

As you launch the platform, you will notice that the Sample Menu displays only when your application is visible.

You can further limit or associate a menu with a particular view. By using the `org.eclipse.ui.actionSetPartAssociation` extension point, you can assign a particular action set to a view or editor such that the action set displays only when the view or editor is active. To modify the `MyAction` plug-in to associate with a particular view instead of just a perspective, you would need to make the following changes:

1. Open the `plugin.xml` file contained in your `MyAction` project.
2. Select the **Extensions** tab within the editor. You should already see the `org.eclipse.ui.actionSets` defined.
3. Remove the `org.eclipse.ui.perspectiveExtensions` extension point, as it is no longer necessary for the `actionSet` to be associated to the perspective.
4. Select **Add...**.
5. Select **org.eclipse.ui.actionSetPartAssociations**, then **OK**.
6. Once `org.eclipse.ui.actionSetPartAssociations` has been added to the All extensions list, right click on the entry, then select **New > actionSetPartAssocation**.
7. The Extension Element Details information will display. Enter `MyAction.actionSet` as the target id. This is the action set you want to associate with a view.
8. Select the **MyAction.actionSet actionSetPartAssociation**. Right click and select **New**, then select **part**.
9. The Extension Element Details information will display. Enter `MyApplication.views.SampleView` as the part.

When you launch, the SampleMenu will now only display when the Sample View is active. Since MyApplication only has the single view available, you won't really notice an immediate difference. However, if you were to modify MyApplication to add another view, then you would notice that Sample Menu only displays when Sample View is active.

Creating views

A perspective can show one or more views simultaneously. Depending upon the application requirements, the number of views that are simultaneously displayed views may change in response to events occurring within the application. Application developers may force views to maintain a specific size or location, or to remain open, when displayed in the perspective. Alternatively, application developers may allow users to close views. While the WebSphere Everyplace Deployment workbench application enables the ability to “reset” a perspective to its initial configuration, you may want to also provide menu options that enable users to open specific views if they are inadvertently closed.

Creating preferences

Plug-in preferences are key/value pairs, where the key describes the name of the preference, and the value is one of several different types, including Boolean, double, float, int, long, and string. The Eclipse platform provides support for storing plug-in preferences and showing them to the user on pages in the workbench Preferences dialog box.

WebSphere Everyplace Deployment extends the Eclipse capabilities by including the Configuration Admin service that can persist configuration information.

Creating preference pages

The references to Preferences in this section cover a wide range of information, from user choices on how to display information, to configuration options needed to connect to enterprises. The client platform framework provides built-in capabilities to help manage preferences. You may choose to use one or both of these options, or to construct your own mechanisms.

When you write the code that defines the content of the preference page, follow these formatting rules:

- Use group boxes to separate areas, if you feel that grouping is necessary. Capitalize only the first letter of the first word of the group box heading.
- Begin each preferences page with a sentence that describes what the user can do.
- Add a colon after field labels.
- Always provide the Restore Defaults and Apply buttons. You can add other command buttons as necessary

Preference options

Eclipse preferences: The Eclipse framework provides an extensible preference store that permits preference information to be stored at various levels. Preference information is stored as key value pairs. Preference pages are generally provided to set or update the preference information stored within the system. Information stored within the Eclipse preference framework will usually be related to display or operating characteristics that the user may change to suit his choices. Eclipse-based preferences are not connected to the enterprise management system. Refer to the *Platform Plug-in Developer’s Guide* for more information on using Eclipse preferences.

Configuration admin: The OSGi core framework also provides a configuration management capability known as Configuration Admin. Configuration Admin provides capabilities to store preference or configuration information based on key value pairs.

Applications that use Configuration Admin to store information will need to implement the `ManagedService` interface. By implementing this interface, the application will be notified when configuration information changes. Applications that use Configuration Admin to store configuration and preference information can also use the `Metatype` service to provide a metadata description of the information. The metadata can describe the parameter types, default values, and validation logic to be applied for each parameter.

If Configuration Admin is used to store configuration information, system administrators can query and update configuration values via the Enterprise Management Agent.

Configuration information stored using Configuration Admin is common to all instances (work spaces) of the WebSphere Everyplace Deployment platform using the same installation directory (this is equivalent to the Eclipse `ConfigurationScope` preferences). Since Configuration Admin values are common to all instances, it is not recommended that user-specific configuration information be stored using Configuration Admin. Applications requiring persistence of user specific configuration information and preferences should use the Eclipse preferences model.

If configuration information is stored using Configuration Admin, preference pages can be used to provide a user interface to view and modify preferences. The WebSphere Everyplace Deployment platform provides classes to assist in using preference pages to interact with Configuration Admin.

The `com.ibm.eswe.preferences.ConfigAdminPreferencePage` has subclassed the `org.eclipse.jface.preference.FieldEditorPreferencePage` to provide preference pages for configuration information stored within Configuration Admin. The `ConfigAdminPreferencePage` creates a `ConfigAdminPreferenceStore`, that uses Configuration Admin and `Metatype` services to obtain the required data to automatically build the preference page.

To use the `ConfigAdminPreferencePage`, you will need to create your own subclass of the `ConfigAdminPreferencePage`, and supply the Bundle and Persistent ID for the properties that you intend to display. Within your subclass, you can also affect the set of properties displayed, as well as add your own validation logic to the page. You will also need to define a preference page via the standard Eclipse extension point for preference pages, and supply the appropriate metadata files (i.e. `METADATA.XML`, `METADATA.properties`). Refer to "Using the Meta Type Service" on page 183 for more information . The `METADATA.XML` and `METADATA.properties` files are loaded from either the plug-in files or from the plug-in's class path. The `METADATA.*` files must be placed within the `META-INF` directory of either the project or the `src` directory for the project

The `ConfigAdminPreferencePage` supports only scalar metatype definitions for non-factory PIDs. Integer, String, and Boolean fields are handled by default. You will need to provide implementations for `FieldEditors` for types such as `Byte`, `Char`, `Long`, `Float`, `Double`, `BigDecimal` and `BigInteger`

Refer to the Javadoc information for the `com.ibm.eswe.preference` package for more information.

Applying Capitalization and punctuation guidelines

Use appropriate punctuation, such as a periods, exclamation points, or question marks at the end of complete sentences. Add a colon at the end of labels for controls in forms and dialog boxes.

For more information, see the *Eclipse User Interface guidelines*.

The following table describes when to use headline-style capitalization and when to use sentence-style capitalization:

Table 9.

Capitalization style	**Guideline**	**Use for**
Headline	Capitalize the first letter of each word, except the following: • Articles such as: a, an, and, the • Short prepositions that are between words such as: for, in, of, on, and to	• Command buttons (push buttons) • Dialog box title bars • Menu items • Menu titles • Section headers (For example, the header to a section on a form. Section headers should also be bold.) • Tabs • Title bars • ToolTips (When the toolTip is for a toolbar item; all other tooltips use sentence-style capitalization.) • Window titles
Sentence	Capitalize the first letter of the first word and any proper nouns, such as Workplace.	• Check box labels • Dialog box labels • Group box or group bar titles • Radio buttons • Text field labels

Creating helpful messages

Messages

Whenever possible use the standard message dialog boxes provided in the `MessageDialog` class as part of the `org.eclipse.jface.dialogs` package. The error message can be modeless, which means it allows the user to continue to interact with the application or modal, which means it requires that the user respond to the error dialog box before continuing to use the application. The following sections describe the available Eclipse message types.

Critical: Informs users of a serious problem that prevents them from continuing their work. Critical error messages are always modal.

Example code:

```
MessageDialog.openError
(parent.getShell(),
"Error Title",
"Error Message");
```

Warning: Alerts users to a condition that requires a decision before proceeding. In many cases you must add buttons to support the Yes, No, and Yes, No, Cancel conditions. Warning messages are usually modal; that is, users must respond before continuing to interact with the application.

Example code:

```
MessageDialog.openWarning
(parent.getShell(),
"Warning Title",
"Warning Message");
```

Customizing existing applications

Activities

An activity is a logical grouping of function that is centered on a certain kind of task. For example, developing Java software is an activity commonly performed by users of the Eclipse SDK platform, and the Java Development Tooling defines many UI contributions (views, editors, perspectives, preferences, etc.) that are only useful when performing this activity. Before we look at the mechanics for defining an activity, let's look at how they are used to help ″declutter″ the UI.

The concept of an activity is exposed to the user, although perhaps not apparent to a new user. When an activity is enabled in the platform, the UI contributions associated with that activity are shown. When an activity is disabled in the platform, its UI contributions are not shown.

Activities are defined using the `org.eclipse.ui.activities` extension point. Refer to the *Platform Plug-in Developer's Guide* for more information on activities and how to define them.

Activities can be used in a static manner to "hide" UI contributions that other plug-ins may have provided.

WebSphere Everyplace Deployment supports the use of activities.

Using activities

Activities are associated with UI contributions using activity pattern bindings, patterns that are matched against the id of the UI contributions made by plug-ins. The process to "filter out" UI contributions is a two-step process described below:

Step 1: Defining an activity: Activities are defined using the org.eclipse.ui.activities extension point. Activities are assigned a name and description that provide information about an activity. The id of the activity is used when defining pattern bindings or other relationships between activities.

An example activity definition:

```
<extension
 point="org.eclipse.ui.activities">
 <activity
```

```
  name="WebSphere Everyplace Deployment Specific"
description="Filters out WebSphere Everyplace Deployment UI contributions"
  id="WebSphere Everyplace Deployment Activities">
 </activity>
```

Step 2: Binding activities to UI contributions: Activities are associated with UI contributions using pattern matching. The pattern matching used in activity pattern bindings follows the rules described in the `java.util.regex` package for regular expressions. The patterns used by the workbench are composed of two parts. The first part uses the identifier of the plug-in that is contributing the UI extension (the plug-in namespace). The second part is the id used by plug-in itself when defining the contribution (which may or may not also include the plug-in id as part of the identifier). The following format is used:

```
plug-in-identifier + "/" + local-identifier
```

For example, the following activity pattern binding states that all UI contributions from the WebSphere Everyplace Deployment workbench plug-in (`com.ibm.eswe.workbench`) are associated with the WebSphere Everyplace Deployment Specific activity. When this activity is enabled or disabled, the state of the UI contributions within the workbench are affected.

```
<activityPatternBinding
activityId="WebSphere Everyplace Deployment Specific"
pattern="com\.ibm\.eswe\.workbench/.*" />
</activityPatternBinding>
```

The next binding is more specific. It states that the contribution named `install` defined in the `InstallUpdate` Launcher plug-in (`com.ibm.eswe.installupdate.launcher`) is associated with the WebSphere Everyplace Deployment Specific activity. As this activity is enabled or disabled, the specific contribution will be included

```
<activityPatternBinding
activityId="WebSphere Everyplace Deployment Specific"
pattern="com\.ibm\.eswe\.installupdate\.launcher/install" />
</activityPatternBinding>
```

WebSphere Everyplace Deployment does not define any activities for use by applications. However, applications may define activities to group WebSphere Everyplace Deployment UI contributions. Refer to “WebSphere Everyplace Deployment top level menus” on page 211 for specific details about the menu identifiers defined by the client platform.

Integrating existing RCP applications into WebSphere Everyplace Deployment

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

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

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

- Because the workbench advisor and application are already provided, applications that include a workbench advisor or RCP application may no longer need to use those capabilities. However, some application function may have been included in the advisor or application. You may be able to re-use the plug-ins without necessarily running the application
- Actions that may have been defined by the advisor class could be defined as `actionSets` in a new or existing plug-in.
- WebSphere Everyplace Deployment has provided a set of menus as identified in "WebSphere Everyplace Deployment top level menus" on page 211. Note that applications that provide action sets may expect certain other menus to be present. If those menus are not present, then `actionSets` that do not include a menu option will not display their actions.
- To enable applications written as perspectives to contribute to WebSphere Everyplace Deployment, you will need to update the application plug-in that defines the perspective to also define the WebSphere Everyplace Deployment `WctApplication` extension point.
- WebSphere Everyplace Deployment may not include all of the plug-ins typically provided in the Eclipse SDK. If the Eclipse SDK was the target for application development, then attempting to run the application on WebSphere Everyplace Deployment may not be successful. In general, plug-ins that exist in the Eclipse SDK could be added to WebSphere Everyplace Deployment along with the application plug-ins. Note that IBM will not provide support for these plug-ins if added to the platform.

Developing data access and synchronization applications

Databases

Database application developers use JDBC, the application programming interface that makes it possible to access relational databases from Java programs. The JDBC API is part of the Java 2 Platform, Standard Edition (J2SE) and is not specific to any particular database implementation, such as Cloudscape or DB2 Everyplace. It consists of the `java.sql` and `javax.sql` packages, which are sets of classes and interfaces that make it possible to access databases (from a number of different vendors) from a Java application.

The JDBC specification defines several interfaces and types that application developers use to access the database.

- A DataSource is the primary definition of a database, and typically defines database access properties and locations.
- A Connection is the communication object that enables queries and updates to be performed against the database. The preferred method for obtaining a connection in JDBC 3.0 is with a DataSource object, as opposed to using DriverManager in a JDBC 2.0 specification implementation.
- A Statement enables the application developer to affect specific actions on the database. There are specialized types known as PreparedStatement or CallableStatement that provide additional advantages and capabilities. A Statement is created from a Connection object.
- A ResultSet represents the result of a query. The ResultSet object is returned upon successful execution of a query in a Statement.

As previously mentioned, the JDBC APIs are part of the J2SE specification. In order to get a clear picture of the current JDBC APIs it is useful to review the history of them and their relationship to the J2SE specification. J2SE 1.2 defined JDBC 2.0, which included definitions in the java.sql package. The main way of accessing databases was with the `java.sql.DriverManager` interface. Sun then defined the JDBC 2.0 extension package, which introduced the `javax.sql` package and a new DataSource interface for accessing databases in Java, as well as support for connection pooling. JDBC 3.0 was subsequently defined and combined the two components of JDBC 2.0 into one JDBC specification. This was first included in J2SE 1.4.

In addition to knowledge of the development of JDBC APIs, database application developers also need to have a detailed understanding of the Structured Query Language (SQL). SQL is the standard query language used with relational databases and is not tied to a particular programming language. No matter how a particular relational database management system (RDBMS) has been implemented, the user can design databases and insert, modify, and retrieve data using the standard SQL statements and well-defined data types. SQL-92 is the version of SQL standardized by ANSI and ISO in 1992. Entry-level SQL-92 is a subset of full SQL-92 specified by ANSI and ISO that is supported by nearly all major DBMSs today. In 1999, another update to the SQL standard was made available called SQL-1999 or SQL-99. You can find reference information about the particulars of the SQL implementation in Cloudscape in the *IBM Cloudscape Reference Manual* and the implementation in DB2 Everyplace in the *IBM DB2 Everyplace Application Developer's Guide*.

Embedded databases

IBM WebSphere Everyplace Deployment provides two relational databases that are accessible using JDBC interfaces, DB2 Everyplace and Cloudscape.

- DB2 Everyplace features a small footprint relational database and high performance data synchronization solution that enables enterprise applications and data to be securely extended to mobile devices such as personal digital assistants (PDAs), smart phones, other embedded mobile devices, and desktops. With DB2 Everyplace, the mobile work force in industries such as health care, telecommunications, retail, distribution, transportation, and hospitality can now easily access the information they need to perform their work from any location, at any time, right from the palm of their hand. It is especially suitable for embedded devices, where large databases and sophisticated queries are not normally required, but can also be used on desktop platforms. DB2 Everyplace provides transaction support covering updates to multiple tables within a single transaction, encrypted tables, and zero client administration.
- Cloudscape is a 100% pure Java relational database, providing SQL-92, partial SQL-99, and Structured Query Language for Java (SQLJ) support, indexes, triggers, transactions, encryption, and the standard features that one expects of a relational database.

DB2 Everyplace and Cloudscape comparison

DB2 Everyplace and Cloudscape are similar, but have features that might make one a better choice for client needs.

Table 10.

	DB2 Everyplace	**Cloudscape**
Implementation Type	High performance native implementation (see DB2E documentation for a complete list of supported devices)	Java-based (platform independent) implementation
On-Disk Footprint	250 KB	2 MB
Number of connections supported	Allows multiple concurrent connections to a database from the same JVM	Allows multiple concurrent connections to a database
SQL Support	Limited set of SQL data types	Full SQL-92 support, partial SQL-99 support
Schema Support	no	yes
Database Creation Requirements	Directory for database tables must be created prior to use	Cloudscape creates directory
JDBC URL	jdbc:db2e:*location* If no database exists at the location, a new database structure is created	jdbc:derby:*location* Database creation requires the addition of an explicit `create=true` attribute to the URL In addition, the database location can also refer to a zip or JAR file in the case of a read-only database

Refer to the product documentation for more complete information about these products.

Creating database access application development best practices

There are typically two reasons for creating an application that needs to make use of database technologies. One reason for using database technologies is to create a new lightweight client application. Another reason is to adapt an existing server-side application for use as a lightweight client application.

Enterprise Java developers who have written applications requiring the use of JDBC typically rely on obtaining access to the database through a DataSource object. The DataSource object will have already been bound by the Java Naming and Directory Interface (JNDI). The application developer simply needs to locate the DataSource using JNDI, and obtain from the DataSource a connection to the database. The application deployer cooperates with the system administrator to define the mapping and access to a physical database, causing the DataSource object to be bound into JNDI.

Also, in the server environment, database management often falls within the administration realm of a database administrator. A database administrator is typically responsible for creating the physical database, partitioning the database for use among several applications, creating tables and indexes for a particular application, managing the access rights to a database, and monitoring the performance of the database. When the application developer has successfully located a DataSource using JNDI, the developer can obtain a Connection, and begin performing actions against the database.

When adapting a server-side application to run on a client, the application developer often takes more responsibility, performing some roles typically handled by a database administrator. The application developer might be responsible for initially creating the database, creating tables, and configuring the database. In addition, depending upon the set of components available on the client, the application developer might also be responsible for creating the DataSource objects, either directly within the application, or within a JNDI environment on the client. If no JNDI support is available in the client runtime environment, the application developer should use the standard JDBC 3.0 javax.sql.DataSource creation methods. Regardless of how the DataSource object is created or located, application developers still obtain Connection objects from the DataSource, and create Statement objects from the Connection.

In order to ensure minimum changes to applications that use JDBC, some best practices should be followed:

- Use DataSource objects as they exist across the JDBC 3.0 specifications; DriverManager does not exist in the JDBC Optional Package for CDC/Foundation Profile. Limiting the JDBC application usage to the JDBC Optional Package for CDC Foundation Profile subset of JDBC 3.0 provides the most portability from desktops to devices.
- Isolate the DataSource creation or location to a single class. Later, if the environment changes, a change needs to be made in only one place.
- Ensure you close all objects (ResultSet, Statement, Connection) when you have completed work. Servers typically include more sophisticated connection management known as connection pooling, which can accommodate mismanagement of connections. However, on the clients, application developers

are directly responsible for the life cycle of the objects. By promptly closing objects, memory requirements will remain at a minimum.

- Do not hard code a schema identifier during statement creation. DB2 Everyplace does not support schema names, so all statements would need to be changed if an application needed to be migrated to DB2 Everyplace.
- The SQL statements supported by a particular database might not match the statements used within an existing application. The application either needs to adapt statements depending upon database type, or obtain statement information from externalized information.

IBM WebSphere Everyplace Deployment includes the Micro Environment Toolkit, which provides Client Services Platform Profile support in the WebSphere Studio or Rational tooling environment. These platform profiles simplify the creation and configuration of database application projects, enabling you to select the target-embedded database, and provide automatic management of the requisite JDBC libraries. In an Eclipse SDK plug-in development environment, you can use the standard plug-in dependency tooling to provide the necessary access to DB2e and Cloudscape libraries during database application development. Eclipse is an award-winning, open source platform for the construction of powerful software development tools and rich desktop applications.

Client Services platform profile components

IBM WebSphere Everyplace Client Toolkit provides Client Services Platform Profile support in the Rational tooling environment. These platform profiles simplify the creation and configuration of database application projects, enabling you to select the target-embedded database, and provide automatic management of the requisite JDBC libraries. When developing a data access application, any of the Client Services Platform Profiles can be selected for your Client Services project. However, ensure you select the IBM Cloudscape application service for project access to the Cloudscape database engine, or the DB2 Everyplace application service for the DB2 Everyplace database engine. If you are interested to developing code leveraging the DB2 Everyplace ISync APIs, be sure to select the DB2 Everyplace ISync client or the IBM Cloudscape Sync Client for project access to the ISync APIs. For more information on Client Services projects, refer to "Using the IBM WebSphere Everyplace Client Toolkit" on page 169.

Deployment and synchronization

When dealing with databases, you can choose to use a database only as a local data repository or as a repository that actively synchronizes with another node in the topology. In either case, if data needs to be distributed to a database, you need to balance considerations of how much data needs to be distributed and when (once only at initialization, one way from one node to another only on an infrequent basis, frequent exchange between nodes), with the storage capabilities at each node in question, and the networking requirements that would permit the exchange to take place. In addition, if you choose synchronization, application developers should consider database organization, filtering, and conflict resolution policies. Synchronization is useful for exchanging the current state of data between nodes, where transaction boundaries or the order of state changes are not important.

When you use data synchronization, database tables are automatically created. If a database is used only for local storage and will not be synchronized with a server database, you must perform the additional step of creating the initial database.

There are a few options for creating an initial database without using database synchronization:

- Incorporate code to create the initial database within the application. The advantage of this is that the application is fully responsible for creating the database. In addition, if there is a need to rebuild the database, the code is already present. No additional steps are required by the user of the database. One disadvantage is that the code must be carried along with the application, even though it might never be used again. Another disadvantage is that the population of initial data into the database might be too large, or not appropriate to include within Java code. For read-only databases, this is generally not appropriate; if the data were available to populate into the database, then you probably would not need a database. Database creation code could be provided in a separate OSGi bundle that is then removed from the framework after the database has been constructed.
- Distribute the database files with the application. The database is constructed and populated outside of the client environment. The resulting database files are then distributed with the application, either in directory format or in a zip or JAR file. The advantage of this is that the code to build and populate the database runs in another environment; it does not need to be distributed with the application. This is an ideal choice for distribution of a read-only database, as the data can be distributed using CD, memory cards, or other distribution mechanisms. The disadvantages are that the distribution of the files might be more difficult than distributing code. Also, updates to the database typically require redistribution of database files.
- Distribute Data Definition Language (DDL) (a set of database statements) and require the installation application or the end user to create the database. While this overcomes some of the disadvantages of incorporating database creation code within the application, it requires that the end user be sufficiently knowledgeable to create the database, or the installation application becomes more complex. In addition, this also typically requires additional tools (such as the command-line tools DB2eCLP for DB2 Everyplace, or ij for Cloudscape) to be present on the client or device.

Another option for database creation, and for continual update, is to use database synchronization facilities. DB2 Everyplace and Cloudscape are both capable of synchronizing with the DB2 Everyplace Sync Server, using the IBM ISync technology provided with IBM WebSphere Everyplace Deployment. The initial synchronization activity creates the local database tables and also populates the initial set of data. As data is updated on the client device, synchronization transfers that data to the DB2 Everyplace Sync Server and then to other client devices that are configured to receive it. Database administrators set up the necessary subscriptions for synchronization, and can also set up filtering of data to limit the amount of data distributed to client devices.

Database application developers can use the ISync APIs provided with IBM WebSphere Everyplace Deployment to control the synchronization process for their specific databases. This includes initiating the sync, monitoring events during the process, and managing any conflicts or errors that occur.

A simple iSync sample is available in the Rational Samples Gallery under **Technology Samples > WebSphere Everyplace Deployment**.

It is also important to note some differences in database application programming when using database synchronization:

- Any changes made to the local copy might appear much later in the server. Synchronization requires the application or user to initiate the sync to the sync server; the replication cycle must kick in so that changes are pushed back to the back end database server. Furthermore, the local changes can be rejected for many reasons, for example, because conflicts were found at the database server side.
- There is a latency between synchronizing changes from a client to the server and then from the server to other clients. In other words, if a change was made to a local database, these changes do not appear on another client device until the local database is synchronized successfully to the sync server. Then the sync server successfully replicates the changes to the back end database, the changes come back down the sync server for other clients; the changes show up on another client device when it successfully synchronizes.
- Database synchronization is based on row-level updates synchronized to and from the DB2 Everyplace Sync Server and the client devices. Database synchronization captures only the current state of data for synchronization. Because the database row contains only the current state of the data, there are two situations in which the application might need to provide additional capabilities. The first situation is when the ordering of updates that were applied to the database is important. The second situation is when a historical record of the changes to the rows in the database is required. In either of these situations, the application needs to provide this capability. The application can store each change in value in a separate row in the main table, or use additional tables for history purposes. Optionally, the application can use assured messaging to send the various updates to the server.

Security considerations

Server databases typically reside in a well-secured zone, with limited access to applications residing in other network topologies. As a result, data in the database is secured because of network location and access rights. In addition, database backup or sophisticated data management exists to protect against data loss.

Databases existing on client systems have different levels of protection. Physical device security is the first barrier in preventing unauthorized access to databases. This includes locking up or securing the physical device as well as providing secure passwords to protect against others illicitly using the device to access data. An additional way to protect the data in the database is to use the encryption technologies provided by the databases. The entire database, or specific tables, can be encrypted using a key (password) that must be provided by the user. This protects against someone being able to open the database if they were to obtain the physical media storing the database (such as a CD for read-only databases, memory keys, Compact Flash cards, and so on).

To protect against data loss, especially for local databases or local synchronized databases, you should put a backup or synchronization strategy in place to ensure that data is synchronized on an appropriate schedule. Incidentally, this also reduces the chances of data on the device becoming out of date.

DB2 Everyplace and IBM Cloudscape Documentation

For more information on DB2e, refer to the following documentation:

- DB2e Application and Development Guide.pdf
- DB2e Installation and Users Guide.pdf

For more information on Cloudscape, refer to the following documentation:

- Getting Started with IBM Cloudscape.pdf
- IBM Cloudscape Developers Guide.pdf
- IBM Cloudscape Tools and Utilities Guide.pdf
- Tuning IBM Cloudscape.pdf
- IBM Cloudscape Reference Manual.pdf

SyncML

As desktop computers, laptops, personal digital assistants (PDAs) and advanced phones have become part of our business and personal life, the need to access current, consistent data in multiple locations has become a pressing need. The term synchronization, often abbreviated to sync, is broadly used to address this requirement. This section discusses synchronization, along with a description of SyncML4J, an IBM offering that enables ISVs and developers to implement SyncML based applications.

Resources can either be standalone items on the file system, such a word processor document, or items managed within an application, such as a calendar within a Personal Information Manager (PIM) application. If a single user accesses the resources that are required from a single location, such as when using a standalone desktop computer, there is no synchronization issue; the user is always working with the single, and thus current, version. In a local area network, where multiple users access resources on a shared file system, there is no inherent synchronization facility. If two users open and edit the same file, the last person to save the file overwrites the content input by the other user. As applications have become more sophisticated, they often provide support for multi-user access to the resources they manage; however this usually assumes continuously connected devices on a network.

Contacts, calendars, and memos are three common resources that a user might want access to on a variety of devices, beyond the desktop. The most significant problem with situation is that these devices often operate disconnected from the desktop computer for significant periods and it is possible to edit the data on these devices, as well as on the desktop. To merge the edits from both locations, the data must be synchronized between these two devices.

Previously, these resources were synchronized between a desktop machine and single PDA using the synchronization software that was provided with the PDA. Often this meant data was synchronized to a desktop application provided by the device manufacturer; this might not have been the default application used on the desktop, particularly in a corporate scenario. Facilities were often provided to import data to the PDA desktop counterpart application, but this was usually a one-off activity, with no facility to update or merge ongoing edits between the applications.

The situation became worse as the facilities of mobile phones improved. Now there was a third device, usually from a different manufacturer, to synchronize. One possible solution was to ensure that all three devices ran software from one manufacturer and expect manufacturers to ensure compatibility, but this is not the way the market evolved.

Within this context a consortium of companies began the SyncML Initiative to develop an open synchronization standard appropriate to server, desktop, and handheld devices. The organization developed data synchronization (DS), then device management (DM) specifications and regularly held SyncFests, where software and device manufacturers were able to test interoperability between various servers and devices. In November 2002, the SyncML Initiative was integrated into the Open Mobile Alliance (OMA), with the following mission:

“The mission of the Open Mobile Alliance is to facilitate global user adoption of mobile data services by specifying market driven mobile service enablers that ensure service interoperability across devices, geographies, service providers, operators, and networks, while allowing businesses to compete through innovation and differentiation.”

Technology overview

This section provides an introduction to the SyncML4J toolkit available from IBM for the development of sync clients based on the OMA DS and DM standards.

The DS and DM standards needed to take into account the differing device and the network characteristics. To achieve the widest adoption, the protocol had to be suitable for implementation on resource-constrained devices. As a ‘wire’ protocol, it does not specify either an implementation language or application programming interface (API); rather, the protocol is a sequence of XML packages exchanged between client and server during a sync session. Some key protocol features defined in the specifications include support for:

- Multiple data types, including binary
- XML and WBXML encodings
- Multiple transports, including HTTP, HTTPS, OBEX, IrDA
- Client and server authentication and message integrity

The specifications are available for download on the OMA Web site. The adoption of the specifications is progressing; some manufacturers are shipping devices that are DS enabled, several software vendors have toolkits available, and there are open source, C and Java toolkits available.

The latest IBM offering for DS and DM is called SyncML4J, and is part of IBM WebSphere Everyplace Deployment v6.0. SyncML4J enables the creation of DS and DM clients for the Java 2 Platform. SyncML4J is pure Java, delivered as an Eclipse feature. Eclipse is an award-winning open source platform for the construction of powerful software development tools and rich desktop applications. SyncML4J comprises plug-ins for the runtime libraries necessary for creating data synchronization, applications, and device management client applications.

SyncML4J common

In the same way that the DS and DM specifications are based on a common representation and protocol, SyncML4J is built on common components for protocol handling and transport. All mandatory wire commands are supported, as are Basic and MD5 authentication and HMAC message integrity.

The DM device tree represents all manageable settings on the device. The DM specification defines how the tree is used to maintain account information for the DM agent. SyncML4J uses a similar approach to maintain account information for the DS agent; it also uses the tree to maintain a list of data sources capable of interacting with the DS agent. In this way, the developer has the option to manage the client.

The applications are loosely coupled to the agents, so there is no dependency on a particular user interface (UI) library within the base framework. A variety of UIs can be used to build an application, sharing the framework sync code.

SyncML4J data synchronization

SyncML4J provides support for all the mandatory DS 1.1.2 client wire commands. As a framework, SyncML4J supports user-defined data sources (or databases). These can range from simple opaque resources, such as memos and images, to complex schema-aware data types such as relational databases or PIM databases. The framework enables the data sources and their capabilities to be modeled by implementing the SyncSource and SyncSourceCap interfaces respectively. The implementation is then registered into the device tree as a DSSource node. A new (or existing) DSAcc node models the account information, such as server address and credentials and the set of local databases that can be synchronized by this account. Within the DSAcc, for each DSSource node there is a corresponding DSTarget node, recording the corresponding remote database URI, and credential and anchor information. No support is provided in the framework to assist with conflict resolution or duplicate detection; these are implicit responsibilities of implementers of the SyncSource interface.

SyncML4J device management

`SyncML4J` provides support for all the mandatory DM 1.1.2 client wire commands, together with an API for manipulating the device tree locally. Custom nodes are created by subclassing and implementing the abstract methods in `AbstractInterior`, then adding instances of the class into the device tree.

As previously noted, the device tree represents all manageable settings on the device, including in volatile or non-volatile memory and file or I/O system. Custom nodes enable resources that are external to the framework to be manipulated. For example, you can implement a custom node to set the time and declare it into the device tree as `./device/time`. Subsequent commands to get and replace the value of that node could then trigger JNI code to get and set the actual OS system time.

You can save memory, by virtualizing sub-trees using custom nodes, rather than by providing a one-to-one mapping between persistent device tree nodes and resources. A reference to a URI that is a logical child to the custom node dynamically instantiates an appropriate node, enabling it to be manipulated by the wire commands. For example, to make the files of a computer disk drive manageable using the device tree, rather than populate the tree with hundreds of interior nodes and thousands of leaf nodes, you can implement and add a single custom node to the device tree as `./device/driveC`, referencing the drive root. In this example, as wire commands to manipulate files on the drive are received, the custom node dynamically creates nodes to model the addressed file, to which the wire commands are forwarded.

Client Services platform profile components

IBM WebSphere Everyplace Client Toolkit provides Client Services Platform Profile support in the Rational tooling environment. These platform profiles simplify the creation and configuration of database application projects, enabling you to select the SyncML components and provide automatic management of the requisite SyncML libraries. When developing an SyncML application any of the Client Services Platform Profiles can be selected for your Client Services project, but be sure to select the SyncML4J application service on the Platform Profile wizard panel. For more information on Client Services projects please refer to “Using the IBM WebSphere Everyplace Client Toolkit” on page 169.

Developing Embedded Transaction applications

The Embedded Transaction capability of the WebSphere Everyplace Deployment platform enables the development and deployment of business logic components by supporting a subset of the Enterprise Java Bean (EJB) specification. These business logic components are referred to as Embedded Transaction applications, and are run by the platform's Embedded Transaction Container. The WebSphere Everyplace Deployment platform only supports execution of and access to Embedded Transaction Applications executing within the WebSphere Everyplace Deployment runtime.

Note: Use of the Embedded Transaction development tools requires Rational Application Developer or Rational Software Architect. The Embedded Transaction development tools are not supported with Rational Web Developer

Embedded Transaction applications can be developed using many of the same EJB development tools provided by the Rational Software Development platform. You should therefore refer to the Rational online help section "Developing Enterprise applications" as your initial development tools reference. The following topics discuss the additional development considerations and tool usage required when targeting an Embedded Transaction application for the WebSphere Everyplace Deployment platform.

The following table provides pointers to information on tasks that are unique to, or require special consideration when developing Embedded Transaction applications for the WebSphere Everyplace Deployment platform.

Table 11. Embedded Transaction application tasks

Task	Reference
Understanding Embedded Transaction concepts, including which elements of the EJB specification are supported, and which are not.	"Embedded Transaction concepts" on page 78
Working with Client Services Embedded Transaction projects versus EJB projects, and when to use one versus the other.	"Embedded Transaction projects" on page 79
Developing Embedded Transaction logic. This encompasses any special development considerations when coding and constructing the embedded transaction logic.	"Embedded Transaction specific considerations" on page 83
Performing embedded transaction deployment. The Embedded Transaction container requires additional deployment information beyond that provided for an EJB.	"Embedded Transaction Deployment" on page 88
Importing and exporting Embedded Transaction bundles.	"Importing an Embedded Transaction Bundle" on page 92 "Exporting an Embedded Transaction Bundle" on page 93

Table 11. Embedded Transaction application tasks (continued)

Task	Reference
Debugging and testing the Embedded Transaction application.	"Debugging and testing applications" on page 157
Deploying the Embedded Transaction application to a runtime.	"Deploying projects for local testing" on page 166

Embedded Transaction concepts

The Embedded Transaction Container (ETC) provides tooling and runtime support for local Enterprise Java Beans (EJBs). The current version supports the following features of the EJB 2.0 specification:

- Remote and Local Homes for local EJBs
- Stateless Session Beans
- Entity Beans, both bean managed persistence (BMP) and container managed persistence (CMP) at both the EJB 1.1 and EJB 2.0 specification levels (local homes, use of abstract persistence schema)
- Container-managed transactions
- Entity Bean tooling container managed persistence (CMP) support for container managed field types that implement java.io.Serializable.
- CMP Support for DB2e 8.2.1 and Cloudscape 10.0
- JDBC DataSource support
- JNDI support
- Container-managed Relationships

The following features are not supported:

- Stateful Session Beans
- Pass-by-Copy semantics for mutable serializable objects when running in a single address space
- For EJB 1.1, the Embedded Transaction Container does not persist references to an EJB's remote or remote home interfaces. Note that this capability is not required of EJB 2.0
- Message-driven Beans
- Java Security support
- EJB Query Language
- Home methods
- Bean managed transactions
- Enumeration return type for finders. Collection return type is supported.
- Specification of transaction isolation level
- Support for Collections and Iterators outside of the transactions in which they were created

The programming models for the Embedded Transaction container and WebSphere's J2EE EJB server/container are very similar. But, there are differences between the two models, primarily in how they reduce runtime resource requirements. The development differences are covered in "Embedded Transaction specific considerations" on page 83.

Embedded Transaction projects

Using a Client Services Embedded Transaction project versus an EJB project

Embedded Transaction applications can be developed using either a Client Services Embedded Transaction project or an EJB project. The choice of which to use depends on the application content and its primary usage. In general, applications that primarily target the WebSphere Everyplace Deployment platform or depend on other OSGi services besides core Embedded Transaction support should be developed using a Client Services project.

A Client Services Embedded Transaction project is an extension of the EJB project. Because of this, both types of projects make use of the Rational Software Development EJB tools. In addition to this, a Client Services Embedded Transaction project provides the following support for developing an application that is targeting the WebSphere Everyplace Deployment platform.

- The manifest file required by WebSphere Everyplace Deployment application can be automatically managed by the tools.
- The project's class path is maintained to match the class path environment that will exist in the WebSphere Everyplace Deployment runtime. This is useful for detecting class visibility problems at development time rather than runtime.

An EJB project will not have the WebSphere Everyplace Deployment specific tooling aids listed above, but can still be tested and run on the WebSphere Everyplace Deployment platform. This is accomplished by targeting the project's server to the WebSphere Everyplace Deployment runtime through the project's server properties. The tooling will automatically add the proper manifest entries for Embedded Transaction support. However, if the application references other OSGi services or bundles, the developer will have to manually add these dependencies to the manifest file.

A Client Services Embedded Transaction project can also be tested and run on a platform other than WebSphere Everyplace Deployment by reassigning its targeted runtime through the project server properties. Refer to "Debugging and testing applications" on page 157 for further general information. Refer to "Embedded Transaction specific debugging" on page 93 for Embedded Transaction specific debugging information.

Creating a Client Services Embedded Transaction project

Complete the following steps to create a new Client Services Embedded Transaction project:

1. Select **File > New > Project**. The new project wizard displays.
2. Expand the Client Services folder to list the Client Services project wizards. Select **Client Services Embedded Transaction Project**. Then select **Next**. The Client Services Embedded Transaction Project panel displays
3. Specify a project name in the Name field. Select **Finish** to create a project with default settings.

The additional settings that can be configured through this wizard are described in the following tables, along with their default values. Access the additional wizard panels through the **Next** and **Back** buttons. Selecting **Finish** on any of the wizard pages will create the project with the settings you have specified up to that point.

Client Services Embedded Transaction Project panel

Table 12. Client Services Embedded Transaction Project panel

Option	Description	Default Value
Project name	Enter a name for the new Client Services Embedded Transaction Project	None
Project location	Select **Browse** to select a file system location for the new project	The default location creates the project in your current workspace.
EJB version	The EJB version that the project is intended to use	2.0
Add support for annotated Java classes	Selecting this adds annotated Java support to the project (for more information, refer to **Developing enterprise applications > Annotation-based programming overview** in the Rational online help)	Not selected
Create a default stateless session bean	Selecting this creates a default stateless session bean in the project	Not selected

Client Services Content panel

Table 13. Client Services Content panel

Option	Description	Default Value
Bundle ID	This is a unique bundle symbolic name. The bundle name should be a unique URI, following the Java package naming conventions.	The project name is used as the default value.
Bundle Version	The bundle version. The version is in the form of major, minor, and micro numbers, separated by decimal points.	1.0.0
Bundle Name	A descriptive bundle name.	The default name is constructed by appending ″Bundle″ to the project name.
Bundle Provider	A description of the bundle provider.	None
Runtime Library	The name of the JAR file in which the project’s built contents will be placed.	`deployed-ejb.jar` (value cannot be changed)

Table 13. Client Services Content panel (continued)

Option	Description	Default Value
Generate the Java class that controls the bundle's life cycle	Selecting this generates a bundle activator for the project. When unselected, the Embedded Transaction generic activator ("Providing custom bundle activation" on page 87) will be associated with this bundle. Only select this option if the application requires special life cycle processing.	Unselected
This bundle will contribute to the Rich Client Platform	Select this option if you intend for the bundle to use Eclipse extension points to contribute to the Rich Client Platform. Selecting this option affects the default preference for automatically managing manifest package dependencies, on the Client Services Project Options page. If Rich Client Platform is selected, then the Require-Bundle preference is set, otherwise the Import-Package preference is set.	Selected

Platform Profile panel

Allows the platform profile and associated application services to be selected for the project. By default, the necessary services for supporting Embedded Transaction projects will be selected. Refer to "Platform Profile" on page 174 for a discussion of platform profiles and "Application Profile" on page 174 for a discussion of application services.

Table 14. Platform Profile panel

Option	Description	Default Value
Platform Profile	Select from the list the Platform Profile this Client Services project will target. You can change your selection later in the Client Services property page.	WebSphere Everyplace Deployment (6.0.0) Default
Application Services	Check the Application Services that your Client Services project will require. You can change your selection later in the Client Services property page. Grey entries are required by the Platform Profile and cannot be un-checked.	The "Core OSGi Interfaces" Application Service is required by all Platform Profiles.

Client Services Project Options panel

Selects how to manage project package dependencies. Refer to "Managing Client Services project dependencies" on page 173 for more information on these options.

Table 15. Client Services Project Options panel

Option	Description	Default Value
Attempt to automatically resolve Manifest dependencies	Select this option to enable the tools to automatically manage the package dependency information in the manifest file. Package dependencies in your project's Java code will automatically be reflected through proper updates to the manifest file. When this option is not selected, package dependencies that are not properly reflected in the manifest are flagged with problem markers, along with quick fixes to resolve the problems.	Selected
Give preference to `Require-Bundle`	`Require-Bundle` will be used to automatically fix a package dependency in cases where either `Require-Bundle` or `Import-Package` can be used.	Selected
Give preference to `Import-Package`	`Import-Package` will be used to automatically fix a package dependency in cases where either `Require-Bundle` or `Import-Package` can be used.	Not selected

Converting an EJB project to a Client Services Embedded Transaction project

You can convert an existing EJB project into a Client Services Transaction project by using the Convert Project to Client Services project wizard. Refer to "Convert Project to Client Services Project Wizard" on page 250 for information on how to use this wizard.

This will retain the existing EJB logic of the project, and will add Client Services tooling support. There is no wizard to convert a Client Services Embedded Transaction project to an EJB project. If you wish to retain the original EJB project, you should copy the project before converting it. This can be done as follows:

1. In the Package Explorer or Project Explorer view, right click the project to be copied to display its context menu.
2. Select **Copy**.
3. Right click on an empty space in the project view.
4. Select **Paste**. This displays the Project Copy dialog.
5. Enter a project name for the project copy, and select **OK**.

Embedded Transaction specific considerations

The Embedded Transaction container is targeted for more constrained devices than typical J2EE EJB servers/containers. While the programming models between the two are very similar, there are aspects that are unique to Embedded Transaction Container.

The following tasks involve Embedded Transaction specific considerations:
- Implementing Finder Methods
- Configuring and Using Data Sources
- Locating EJBs
- Conserving JDBC Resources
- Working With User Managed Transactions
- Providing Custom Bundle Activation

There are also deployment related differences, such as the introduction of the `eejb_deploy.xml` file. These differences are covered in "Embedded Transaction Deployment" on page 88.

Implementing finder methods

The Embedded Transaction Container uses the following approach to implementing custom finder function. For each finder method declared on the Home interface (other than `findByPrimaryKey`), you must provide the business logic required to build the corresponding collection. The tool requires that this logic be packaged in an abstract finder helper class, which the tool extends in a concrete JDBC finder helper class. Typically the business logic is encapsulated in a SQL SELECT statement.

The container provides a base JDBC finder class, `com.ibm.pvc.txncontainer.BaseJDBCFinder`, that provides the bulk of the required finder functionality. By extending this class, you only need to supply the actual finder logic. For every custom method, `findXXX`, defined on the Home interface, you must code a corresponding `ejbFindXXX` method on the abstract finder helper class. `BaseJDBCFinder` provide the methods `getTableName()` (returns String specifying the database table name) and `getPreparedStatement(String)` (returns a `PreparedStatement` derived from the appropriate `DataSource`).

Finders can return single items or collections of items.

The following is a sample multi-result finder implementation, which returns a Collection:

```
public abstract class Customer20JDBCFinder extends BaseJDBCFinder
{
 public Customer20JDBCFinder(DataSourceHome arg0, String arg1) {
   super(arg0, arg1);
 }
public Collection ejbFindByFirstName (String firstName) throws FinderException
 {
  final String selectSQL = "select * from customer where fname = ?";

  PreparedStatement pstmt = null;

  try {
   pstmt = getPreparedStatement(selectSQL);
   pstmt.setString(1, firstName);
  }
  catch (SQLException e) {
   throw new FinderException
    ("Problem executing Finder: "
     + selectSQL
     + ", Exception = "
     + e.toString());
  }

  return new BaseJDBCCollection(pstmt, this);
 }
```

The following is a sample single-result finder implementation, which returns a single key:

```
public abstract class Customer20JDBCFinder extends BaseJDBCFinder
{
 public Customer20JDBCFinder(DataSourceHome arg0, String arg1) {
   super(arg0, arg1);
 }
  public Customer20Key ejbFindById (String Id) throws FinderException
 {
     final String selectSQL = "select id from customer where id = ?";

  PreparedStatement pstmt = null;

     try {
        pstmt = getPreparedStatement(selectSQL);
        pstmt.setString(1, Id);
     }
     catch (SQLException e) {
        throw new FinderException
    ("Finder = "
     + selectSQL
     + ", Exception = "
     + e.toString());
     }

     return (Customer20Key) singleResultFinder(pstmt, true);
 }
```

In addition to the SQL application logic, you should ensure the following rules are met:

- Ensure that the finder helper is abstract.
- Ensure that the finder helper extends `BaseJDBCFinder`
- Code the appropriate two-parameter constructor, and invoke `super()`

The Boolean parameter supplied to the `singleResultFinder` method determines whether only one object can match the finder criteria. If true, the container will throw an exception if more than one object matches; if false, one object will be returned from the result set, however, you will have no way of determining which object is selected.

Suppose you have an entity bean to which you want to add additional searching/lookup capabilities. For the purposes of these steps, assume you have an entity bean representing a customer, with the Customer class representing the remote interface, `CustomerBean` the actual implementation, and `CustomerHome` as the home interface. In order to add additional search methods, you need to do the following:

1. Update the `CustomerHome` class to define the new method (e.g. `findByFirstName (String firstName)`.
2. Create a new class extending `BaseJDBCFinder`.

a. The constructor must call `super( arg0, arg1 )`.
b. You must implement a method `ejb<findername>`. For our example, therefore, you must implement the method `ejbFindByFirstName( String firstName )`.
c. Use the method `getPreparedStatement( sqlstring )` to obtain a statement to execute. The base class provides the appropriate setup of requesting a connection, etc.
d. For a collection result, return a `BaseJDBCCollection` object.
e. For a single result finder, return the results of the `singleResultFinder( )` method.

Configuring and using data sources

Creating and binding DataSource instances

Data base vendors provide implementation specific `DataSource` classes for connecting to their databases. The Embedded Transaction Container requires a specific DataSource, `TxnDataSource`, be used when connecting to the database. These specific DataSource are created via the `TxnDataSourceFactory`.

This "wrapping" of the vendor specific `DataSource` can be done before JNDI binding or after lookup. Wrapping before JNDI binding is the best practice, since this is done once, as opposed to wrapping after every lookup.

Refer to "Using declarative JNDI" on page 196 for general information about how to handle the JNDI binding, and to "TxnDataSourceObjectFactory" on page 198 for `TxnDataSource` specific information.

Note: For information on using JDBC DataSource to access JDBC data bases, refer to "Creating database access application development best practices" on page 69.

Advanced topics: While the declarative JNDI means of wrapping and binding is the preferred method, this can also be handled programmatically, as follows:

```
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import com.ibm.db2e.jdbc.DB2eDataSource;
import com.ibm.pvc.txncontainer.TxnDataSourceFactory;
private void createAndBindDataSource() throws NamingException {
 // create DB2e specific Datasource object and set it's jdbc url
 DB2eDataSource db2eDS = new DB2eDataSource();
 db2eDS.setUrl("jdbc:db2e:" + EJBDB_LOC);
 // wrap and bind the vendor specific data source
 DataSource ds = TxnDataSourceFactory.create(db2eDS);
 Hashtable env = new Hashtable();
 env.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.ibm.pvc.jndi.provider.java.InitialContextFactory");
 InitialContext context = new InitialContext(env);
 context.bind(DATASOURCE_NAME, ds);
}
```

Where `EJBDB_LOC` is the physical location of the database, and `DATASOURCE_NAME` is the JNDI name of the DataSource.

Locating and connecting to a DataSource

Session beans and client applications typically require a database connection, and, according to the EJB specification, acquire the DataSource from JNDI in the same way that it does when finding EJB homes (for more information, refer to "Finding EJB homes." The following sample shows how Session Beans and client applications can acquire a Connection in the Embedded Transaction Container environment.

```
import java.sql.Connection;
import javax.sql.DataSource;
import javax.naming.InitialContext;

  protected Connection getConnection() throws SQLException
  {
      DataSource     ds = null;

      try {
       Hashtable env = new Hashtable();
       env.put(Context.INITIAL_CONTEXT_FACTORY,
           "com.ibm.pvc.jndi.provider.java.InitialContextFactory");
       InitialContext context = new InitialContext(env);
           ds = (DataSource) context.lookup(DATASOURCE_NAME);
      }
      catch (Exception e) {
           e.printStackTrace();
           throw new IllegalArgumentException("Cannot lookup DataSource:" + e);
      }

      return ds.getConnection();
  }
```

Where `DATASOURCE_NAME` is the JNDI name of the DataSource.

Locating EJBs

The EJB deployment descriptor includes information about the EJB's JNDI binding. As with J2EE EJBs, the JNDI related values can be set via the Deployment Descriptor editor.

The actual timing of EJB binding is handled by the WebSphere Everyplace Deployment JNDI provider (for more information, refer to "WebSphere Everyplace Deployment JNDI overview" on page 195) and Declarative JNDI (for more information, refer to "EJBObjectFactory" on page 197).

Finding EJB homes

Client applications perform the following operation to access deployed EJBs:

```
// Import the JNDI InitialContext class
import javax.naming.InitialContext;

// Name of the Home
final String jndiName = "java:comp/env/EmployeeFromJDBC";
// Get the reference to the Home
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.ibm.pvc.jndi.provider.java.InitialContextFactory");
InitialContext context = new InitialContext(env);
final EmployeeHome localHome = (EmployeeHome)context.lookup(jndiName);
```

Conserving JDBC resources

EJBs provide abstractions that shield the programmer from the technical details of the underlying data stores. These abstract layers need help managing the underlying data base resources to work within the constraints of the embedded data base engines. For example, DB2e v8.2 supports a maximum of 20 open statements. When working with Container Managed Persistence (CMP) Beans, statements may be opened when a finder method creates a collection and when an iterator is used to walk the contents of that collection. When an iterator has reached its end, the corresponding statement can be closed. While the CMP abstraction layer will automatically close statements when it is certain they are no longer needed, it is up to the application programmer to explicitly dispose of all Collections and Iterators to ensure that the corresponding statements are closed prior to commit or garbage collection implicitly closing them. When a program no longer needs a Collection or Iterator, if that object is an instance of `IDisposable`, the dispose method should be called on it.

For example:

```
        if (myCollection instanceof IDisposable) {
                ((IDisposable)myCollection).dispose();
}
```

Working with user managed transactions

The Embedded Transaction Container provides an implementation of a transaction manager. This `TransactionManager` is used for automatically handling transaction management issues.

Note: Unlike a J2EE `TranactionManager`, the embedded transaction container's transaction manager does not provide an implementation of `javax.transaction.TransactionManager`, nor does it support two phase commit.

Advanced topics

Applications generally do not work directly with the transaction manager. For the advanced cases, the Embedded Transaction Container does provide a User Transaction Factory (`UTFactory`) that enables applications to work with user transactions.

The `UTFactory` provides a static `getUserTransaction()` method that will return an implementation of the `javax.transaction.UserTransaction` interface. This `UserTransaction` instance is used by developers for handling typical transaction-related actions:

- `begin()` – creates a new transaction and associated it with the current thread
- `commit()` – completes the transaction associated with the current thread
- `rollback()` – rolls back the transaction associated with the current thread
- `setRollbackOnly` – after calling this method, a "rollback" is the only possible outcome of the transaction associated with this thread
- `getStatus()` – obtains the status of the transaction associated with this thread
- `setTransactionTimeout()` – is not supported

Providing custom bundle activation

The Embedded Transaction Container tooling provides a default bundle activator if the application does not already contain one. The default bundle activator, `com.ibm.pvc.txncontainer.GenericActivator`, registers the EJB's home interface.

Advanced topics

A custom bundle activator would be required if the application needs to perform OSGi specific operations. A custom bundle activator can be created at project creation by selecting **Generate the Java class that controls the bundle's life cycle** in the Client Services Embedded Transaction Project wizard (For more information, refer to "Creating a Client Services Embedded Transaction project" on page 79) To create a custom bundle activator for an existing project, you can perform the following steps:

1. Add the custom bundle activator class to the package in which the other EJB classes (home, interface, implementation, finders) are located.

   Note: The custom bundle activator class must extend the `com.ibm.pvc.txncontainer.GenericActivator` class and call the `start()`and `stop()` methods of `GenericActivator` inside its own `start()` and `stop()` methods.
2. Open the Bundle Manifest editor on the project's `META-INF/MANIFEST.MF` file, and enter the custom bundle activator class in the Class field of the Overview page. This will update the EJB manifest file `META-INF/MANIFEST.MF`, by setting the `Bundle-Activator` property to the custom bundle activator class.

Custom bundle activator example:

```
import org.osgi.framework.BundleContext;
import com.ibm.pvc.txncontainer.GenericActivator;
/**
* Minimum bundle activator for EJBs
*/
public class MyBundleActivator extends GenericActivator {

       public void start(BundleContext context) throws Exception{
              // custom tasks here
              super.start(context);

       }
       public void stop(BundleContext context){
              // custom tasks here
              super.stop(context);
       }
}
```

Embedded Transaction Deployment

Embedded Transaction applications require a deployment step before they can be run. This is analogous to the deployment step performed on an Enterprise Java Bean (EJB), and should not be confused with the concept of deploying a bundle to the runtime (which involves methods of packaging and delivering the bundle to the runtime). Embedded Transaction deployment involves specifying the proper deployment information in deployment descriptors, and subsequently performing a deployment operation which performs the necessary transformations on the project to enable it to be run by the Embedded Transaction container.

Embedded Transaction applications require additional deployment information beyond that typically provided when deploying an EJB. The standard EJB deployment descriptor, ejb-jar.xml , must contain proper deployment information. The Rational Software Development tools automatically manage this in many cases, and provide an EJB Deployment Descriptor editor. For more information, refer to **Developing enterprise applications > EJB Deployment Descriptor** in the Rational help. Refer to "Embedded Transaction Deployment Descriptor" on page 89 for the additional deployment information you must add for Embedded Transaction applications, and "Embedded Transaction Deployment Editor" on page 90 for information on using the editor to update this information.

Once the proper deployment information is specified, a deployment operation can be carried out to enable the application to be run. "Invoking deployment" describes how and when the deployment operation is carried out.

Invoking deployment

Embedded Transaction deployment can be carried out in the following ways:

- Automatically when running or exporting a project
- Manually through the deploy action
- Through an ant script

Automatic deployment when running or exporting a project

Deployment is automatically run whenever an Embedded Transaction project is selected to be run on a WebSphere Everyplace Deployment runtime that is launched through the tools. Refer to "Debugging and testing applications" on page 157 for information on launching WebSphere Everyplace Deployment runtimes through the tools.

Manual deployment through the Deploy action

Deployment can be manually invoked from the J2EE perspective by right clicking on the project to bring up its context menu. Then select **Deploy**.

Ant script deployment

A Deployment Ant script can be used for batch deployment of projects. It can also be used to customize the deployment operation. For information about tailoring this file, refer to "Customizing the deployment Ant script (ejb-build.xml)" on page 91.

By default, the deployment operation is performed internally by the tools. However, if the project contains a deployment ant script, this Ant script will be run to perform the deployment operation. This provides a means for customizing the deployment operation, by customizing the deployment Ant script.

Perform the following procedure to generate a deployment Ant script:

1. From the Project Explorer or Package Explorer view, right click the target project to display its context menu.
2. Select **Client Services > Create ANT Deploy File**.
3. This creates the deployment Ant script as the file `ejb-build.xml` in the project.
4. Optionally, tailor the Ant script.
5. You can run this script from the tools Explorer view by right clicking on file `ejb-build.xml`, and selecting **Run > Ant Build**.

Embedded Transaction Deployment Descriptor

The Embedded Transaction Container Tooling plug-in auto-generates the XML deployment file. The XML deployment file contains the custom deployment information required to deploy the EJB. The information provided in the deployment file supplements the information already included in the EJB deployment descriptor (`ejb-jar.xml`). To deploy the embedded EJB, you must supply this information as an XML file that conforms to the schema file `eejb-deployment.xsd`, which is shipped with the tooling. Information included is as follows:

jndi-name
: The name through which the deployed Home is accessed via the naming service (e.g., JNDI). It is not assumed that the Home will be bound to a `java:/comp/env/ejb` Context, so the name supplied in the deployment information is used exactly "as is". Required for both session and entity beans.

jdbc-bean
: Supplies the information needed to deploy an entity bean to the Embedded Transaction Container using a JDBC-based DataSource. In this element, the user specifies the name of the `abstract-finder-helper` class that the user has supplied for the EJB, and also specifies the name of the deployed finder-helper class that the tooling will generate. Required by container managed persistence (CMP) entity beans only.

datasource-name
: Specifies the name through which the DataSource providing a connection to the data store can be accessed from the naming service. Required by entity beans only.

table-name
: Specifies the name of the relational database table that provides persistence for the container-managed persistence (CMP) entity bean. This element is not required by bean-managed persistence (BMP) entity beans.

deployed-class
: Specifies the name of the deployed bean implementation class. Required for both session and entity beans.

ejbivar
: Specifies the name of the table column that provides persistence for the entity beans fields.

cmp-field
: The name of the entity bean field to be persisted. The value of this field should match the value specified in the EJB deployment descriptor, `ejb-jar.xml`.

Note: You must complete the database information (DataSource name, table name, column names) for entity beans. All other information is completed by the tooling. No update to the deployment file is necessary for stateless session beans.

Embedded Transaction Deployment Editor

Information in the "Embedded Transaction Deployment Descriptor" on page 89 can be managed through the Embedded Transaction Deployment Editor. This editor can be opened as follows:

1. Locate the project's Embedded Transaction Deployment descriptor file in the Explorer view. This is file `eejb_deploy.xml` in the project's `ejbModule/META-INF` folder.
2. Open the deployment descriptor by double clicking it, or right clicking and selecting **Open**. The Embedded Transaction Deployment editor displays.

The editor will display all of the project's defined beans in the Beans view. Select a bean from the Beans view, and its associated Embedded Transaction deployment information will display on the right hand side for editing.

Customizing the deployment Ant script (ejb-build.xml)

While much of the generated deployment Ant script is specific to the project being deployed, some values are defaulted and can be changed by editing the generated `ejb-build.xml` file.

Customizing for target data base (DB2e and Cloudscape)

Different code is generated based on what data base support is available on the platform to which the EJB will be deployed. The `database.mode` property in the `ejb-build.xml` is used to specify this. The default value is "DB2e". To use Cloudscape, you must modify this property in the `<target name="init">` section, as in the example below:

```
<property name="database.mode" value="derby"/>
```

Advanced topics

The following discusses parameters that would typically only be modified for advanced customization purposes.

As discussed in the EJB specification, bean providers and application assemblers supply a set of ejb-jar files containing a J2EE application to a deployer . The deployer deploys the enterprise beans contained in the `ejb-jar` files in a specific operational environment. This deployment step for the Embedded Transaction Container is provided by the `IntegratedDriver`.

The parameter settings necessary to run the Integrated Driver tool are pre-configured in the Ant script, and the default values usually do not need to be specified by the EJB developer. The Integrated Driver and its required arguments are described below. The `end-to-end.args` property in the `ejb-build.xml` file is used to pass parameters to the Integrated Driver.

Integrated Driver required arguments

The following arguments are required and have the specified defaults.

Table 16. Required arguments

Argument	Description	Example
-cmpJar=<ejb-jar filename>	Specifies the input ejb-jar file (from the Bean Developer). This file must include all necessary .class files and the EJB deployment descriptor per the EJB specification.	-cmpJar = c:/tmp/lib/employee.jar
-deployedJar=<output JAR filename	Contains original bean classes and generated code. This is the deployed EJB Jar.	-deployedJar = c:/tmp/lib/deployed-employee.jar

Table 16. Required arguments (continued)

Argument	Description	Example
-dbType=<database that emitted code will access>	The current valid values are: `DB2e`, `derby` and `none`. The latter value is used when deploying only stateless session beans and bean-managed Entity beans to denote the fact that emitted code is database independent. If no dbType argument is supplied, a default value of none is assumed.	-dbType = DB2e (defaults to none)
-findersRootDir=<root directory of all finder classes>	The directory from which all the EJB finder implementations (.java files) are rooted.	″-findersRootDir =src″
-deploymentXMLFile = <extra deployment information XML file>	An XML file containing deployment information beyond that available in the spec-defined EJB deployment descriptor.	-deploymentXMLFile = eejb_deploy.xml
-deploymentXMLSchemaFile = <XML Schema file for deploymentXMLFile>	Name of the XML Schema file describing deploymentXMLFile.	-deploymentXMLSchemaFile = eejb_deployment.xsd
-classpath=<CLASSPATH to use for tooling-invoked compilations>	The CLASSPATH to use when the tooling is compiling its generated code.	Include at least ″-classpath = jta.jar; txncontainer.jar; <undeployed EJB jar>″, plus any other dependencies your EJB requires

Importing and Exporting Embedded Transaction Bundles

Importing an Embedded Transaction Bundle

You can import an Embedded Transaction bundle that was exported with source. Refer to “Exporting Client Services projects” on page 166 for information on exporting a bundle. If the bundle was not exported with source, it will not be eligible for import.

Complete the following steps to import an Embedded Transaction bundle:

1. Open the Import wizard by selecting **File > Import...** The Import window displays.
2. Select **Embedded Transaction Bundle**, then select **Next**. The Import Client Services Embedded Transaction plug-in/bundle window displays.
3. Select the Embedded Transaction bundle file that you want to import by either selecting **Browse**, or by typing the file name into the Client Services Embedded Transaction plug-in/bundle field.
4. Select **Finish**. A project will be created for the imported bundle. The project name will be the bundle’s symbolic name.

Exporting an Embedded Transaction Bundle

An Embedded Transaction project can be exported as a bundle by using the OSGi bundle export wizard. Refer to "Exporting Client Services projects" on page 166 for directions on how to export a bundle.

Embedded Transaction specific debugging

This section includes Embedded Transaction Container specific debugging information.

For general debugging information, refer to "Debugging and testing applications" on page 157.

Saving source for viewing during debugging

The default behavior specified in the general `ejb-build.xml` file is to delete the auto-generated source code for CMP entity beans. This behavior can be changed if the application programmer would like to keep the source code for debugging purposes. Without this source code, the programmer can not visually step through the code when running the debugger. The following two steps will persist the auto-generated source code, and point the debugger to the source location.

1. The `ejb-build.xml` file within an EJB project contains a "clean" tag that signal which files/directories are to be deleted at the end of a build. The default tag is the following:

   ```
   <target name="clean">
        <delete dir="temp"/>
        <delete dir="tempfiles"/>
   </target>
   ```

   To persist the source files needed for debugging, the "delete" tags must be removed or commented out. If commented out, the tags after the needed modifications will look as follows:

   ```
   <target name="clean">
        <!--
        <delete dir="temp"/>
        <delete dir="tempfiles"/>
        -->
   </target>
   ```

2. In order for the debugger to display the source code to be stepped through, it must know the location of that source code. Setting the source code location is one of the options when launching an Eclipse workbench in debug mode. One typical way of setting this is to:
   a. Select **Debug...** The Create, Manage, and Run Configurations Panel displays.
   b. Select the **Source** tab.
   c. Select the **Add** button. The Add Source panel displays.
   d. Select **directory**. The directory with the source files is located within the `tempfiles` folder in the associated EJB project.

Enabling logging and tracing with the Embedded Transaction Container

Embedded Transaction Container logging enables a developer to associate logged messages with one of six logging levels: fatal, error, warning, info, debug, and trace. These logging levels are ordered in decreasing severity. At runtime, the user can configure the Embedded Transaction Container to specify, on a per-class basis, the level of logging that is enabled. Enabling a given log level implies that all log messages associated with that level, or a higher log level, should be logged. Messages associated with a lower log level are ignored.

If you would like to enable logging of the Embedded Transaction Container, add the following options to the VM arguments section of your launch configuration:

```
-Deejb.logging.priority.com=debug
```

```
-Deejb.logging.logwriters=com.ibm.pvc.utils.logger.ConsoleLogWriter=debug
```

This will turn on debug for all components of the Embedded Transaction Container, and send the output to the console.

Developing Mobile Web Services

Mobile Web Services overview

The IBM WebSphere Everyplace Client Toolkit extends the Rational Software Development Platform through plug-ins that enable you to build applications targeting the WebSphere Everyplace Deployment runtime platform. The IBM WebSphere Everyplace Client Toolkit Web Services plug-in suite enables you to develop applications that consume and are exposed as Web Services targeting the OSGi-based WebSphere Everyplace Deployment runtime platform. For more information on the Rational Software Development Platform, visit http://www.ibm.com/pvc.

The IBM WebSphere Everyplace Client Toolkit Web Services runtime plug-in suite provides functionality similar to libraries that implement the Java 2 Micro Edition Web Services Specification (JSR-172).

To enable you to develop Web Services applications, the Web Services Tools allows you to generate client code that consumes Web Services as well as exposes OSGi services as Web Services providers.

An application that will consume a Web Service needs to identify the service end-point, typically a URL to a Web Services Description Language (WSDL) document and use the interface to invoke the Web Services provider. An application that will be exposed as a Web Services provider must implement a Java interface that defines the Web Service calls.

Technologies

Web Services Description Language (WSDL)

A WSDL document provides the description of the Web Services interface. Web Services can be created using a top-down or bottom-up approach. A top-down approach is used to generate code from a WSDL (typically used for developing Web services clients), whereas a bottom-up approach is used to generate a WSDL from code (typically used for developing Web Services providers). However, the IBM WebSphere Everyplace Client Toolkit Web Services plug-in currently supports only the top-down approach.

For more information about WSDL, please visit http://www.w3.org/TR/wsdl.

Simple Object Access Protocol (SOAP)

SOAP is the message format of the transaction that takes place when a Web Services client that communicates with a Web Services provider. The WSDL defines the restrictions on the format of these messages.

For more information about SOAP please see http://www.w3.org/TR/soap.

JAX-RPC

The Java API for XML-Based Remote Procedure Call (JAX-RPC) enables developers to build Web Services using XML-based RPC functionality according to the SOAP 1.1 specification.

For more information about JAX-RPC, please visit
http://java.sun.com/xml/jaxrpc.

The Web Services Client Programming Model

Similar to the programming model specified in the Web Services for J2ME specification (JSR-172), the WebSphere Everyplace Client Toolkit provides the following capabilities:

1. **A generated stub from the Web Services Description Language (WSDL) description of the service operation.**

   The Mobile Web Services Client wizard generates a static client stub class using the WSDL that is exported from the Web Services provider as its input. The stub is then used to invoke the Web Services provider.

   In addition to the static stub, the Web Services Gateway proxy library (`com.ibm.pvcws.osgi`), a component of WebSphere Everyplace Deployment can be used to generate a dynamic client stub on-the-fly. This dynamic client stub may be used in place of the static client stub, which hard-codes the SOAP message definitions and method calls, in order to build Web Services clients dynamically. Other functionality provided by this proxy library is the ability to provide custom marshallers (serializers) for types that are incompatible with JSR-172.
2. **WSDL-defined API.**

   The WSDL document defines an application programming interface (API) that makes up the complete Web Services client application. This API must be present on both the server and client side to allow the endpoints to communicate properly.
3. **Instantiation of the stub**

   The client application uses an instance of the static or dynamic stub to indirectly access the Web service defined by a given WSDL.

   It is imperative that the WSDL definition reflects the actual interface to the Web service at runtime. The JAX-RPC subset does not perform any version control. Any differences between the defined WSDL and the instance of the Web Service may produce unpredictable results.
4. **Invocation of stub methods that correspond to the implementation of service endpoint operations.**

   The Web Services client application can use an instance of the stub to set stub properties, including the service endpoint. The methods generated in the stub are used to call service endpoint operations.
5. **Packaging the stub with the client application.**

   The generated stub is provided in source form. It is used during application development.

Tools

Tools for Mobile Web Services development

The IBM WebSphere Everyplace Client Toolkit provides tools for creating Mobile Web Services client code, as well as code for exposing an OSGi service as a Web services provider.

The tools provided include:

- A Web Services client wizard that includes a wizard to configure security
- A Web Services provider wizard to expose OSGi services as Web Services providers

- A Web Services provider security wizard to configure security
- Editors to modify WS-Security configurations

Creating Mobile Web Services

Creating Mobile Web Services providers

Any OSGi service can be exposed on the WebSphere Everyplace Deployment Client as a Web Services provider using the IBM WebSphere Everyplace Client Toolkit, provided that the service implements a Java interface.

To create a Web Services provider from scratch, perform the following procedure:

1. Create a new Client Services project named **MyWebServicesProvider**:
   a. Select **File > New > Project**.
   b. Select **Client Services > Client Services Project**.
   c. Select **Next**.
   d. Type a name for the new project (for example, **MyWebServicesProvider**).
   e. Select **Next**.
   f. Select **Next**.
   g. Select **Finish**.
2. Create the code that will be exposed as a Web services provider:
   a. Create a simple Java interface with a method declaration.
   b. Create a Java bean class that implements the interface, and then select it in the workspace.
   c. Select **File > New > Other**.
   d. Select **Client Services > Mobile Web Services > Mobile Web Services Provider**.
   e. Select the Java interface implemented by the selected file.
   f. Select **Finish**.

After selecting **Finish**, the IBM WebSphere Everyplace Client Toolkit will modify the Bundle Activator of the targeted project by adding two methods: `getProvider()` and `exposeService()`. Also, the `start()` method of the Bundle Activator will be modified to invoke `exposeService()` so that the bundle is exposed as a Web services provider at startup time. Please note that this call is not restricted to be on the `start()` method, and can be moved to execute at any time as long as the `BundleContext` is available.

Unlike the WebSphere Application Server Web Services tools, the IBM WebSphere Everyplace Client Toolkit does not generate a WSDL document. Instead, the user will select an OSGi service class in a project and run the tools, to generate code that calls the `com.ibm.pvcws.osgi` plug-in APIs to expose the OSGi service as a Web Services provider. Generation of a WSDL-document occurs at runtime using Java reflection into the OSGi service class.

If the Web Services provider needs to handle non-bean classes or types that are incompatible with JSR-172, custom marshallers need to be implemented. Please refer to the section "Custom serialization (marshalling)" on page 103 for more information.

Creating Mobile Web Services clients

The IBM WebSphere Everyplace Client Toolkit can be used to create Web Services client code that calls Web Services providers via a static or dynamic stub. The IBM WebSphere Everyplace Client Toolkit supports the creation of Web Services clients using the WSDL of the Web Services provider. If the Web Services client will target a WebSphere Everyplace Deployment Client Web Services provider, it is best to create and deploy the Web Services provider first, prior to creating the Web Services client so that its WSDL is available.

Following are the steps to create a Web services client from scratch:

1. Create a new bundle project named **MyWebServicesClient**:
   a. Select **File > New > Project**.
   b. Select **Client Services > Client Services Project**.
   c. Select **Next**.
   d. Type a name for the new project (for example, **MyWebServicesClient**).
   e. Select **Next**.
   f. Select **Next**.
   g. Select **Finish**.
2. Create a new client interface and optional stub:
   a. Select **File > New > Other**.
   b. Select **Client Services > Mobile Web Services > Mobile Web Services clients**.
   c. Select **Browse** to select the source folder or project for your client application.
   d. Optionally, enter the package you wish to use for the static stubs or leave it blank to use a default package instead. Please note that this option will not be enabled if you select the **Create Dynamic Stub** check box, as the Web Services runtime component requires that the package name match the WSDL namespace.

      Note: If you intend to deploy a Web service client and a Web service provider in the same runtime, place the Web service client stub in a package that is different from the package of the Web service provider to prevent a runtime conflict.
   e. Enter the URL of the WSDL exposed by the Web Services provider.

      Please note that if the URL is secured with SSL (e.g. HTTPS), you will need to start the Rational Software Development Platform with the appropriate VM arguments to provide a valid client certificate. For example, the following VM arguments can be used:

      ```
      -Djavax.net.ssl.keyStore=<path_to_keystore_file>
      -Djavax.net.ssl.keyStoreType=<keystore_type>
      -Djavax.net.ssl.keyStorePassword=<keystore_password>
      -Djavax.net.ssl.trustStore=<path_to_truststore_file>
      -Djavax.net.ssl.trustStoreType=<truststore_type>
      -Djavax.net.ssl.trustStorePassword=<truststore_password>
      ```

      Optionally, select the **Create Dynamic Stub** check box to use dynamic stubs rather than create static stubs.

      A dynamic stub allows the Web Services client to create and use custom marshallers for WSDL types that are non-bean classes or are incompatible with JSR-172.
   f. Optionally, select the **Configure Security** check box to configure Web Services Security for the client.

g. Select **Next**.

h. If there are any types in the WSDL that may require custom marshalling, you will be presented with the option of generating custom marshaller stubs for each type. This will only be true if the **Create Dynamic Stub** check box is checked.

   For each type selected in the list, two classes will be generated; `MarshalFactory` and `Marshaller`. These classes are implementations of the corresponding classes in the `com.ibm.pvcws.osgi` plug-in. Please refer to the section "Custom serialization (marshalling)" on page 103 for information on how to complete the implementation of custom marshallers.

i. If **Configure Security** was checked on the first page, select **Next** and refer to the section "Securing Mobile Web Services" on page 109 for information on how to configure Web services security. Otherwise, select **Finish**.

   Note: If you receive an exception with the message "Parsing of the specified WSDL failed", followed by an explanation, ensure that the WSDL is accessible through a browser. This exception could either be the result of a firewall message in HTML-form requiring authentication, or could be due to an invalid WSDL. Please consult with your administrator.

Static Mobile Web Services clients

If you generated Web Services client code to use a static stub (i.e., you did not check the **Create Dynamic Stub** box), you will see that a `*Soap_Stub` class was generated. To invoke the Web services provider, you simply need to instantiate a `*Soap_Stub` object and then call the Web services provider methods you wish to exercise. For example:

```
MyWebServiceSoap_Stub stub=new MyWebServiceSoap_Stub();
System.out.println("Name=" + stub.getName());
```

Please note that this release of the IBM WebSphere Everyplace Client Toolkit does not support custom marshalling with static Web Services clients.

Note: The generated stub will contain the URL of the WSDL specified in the tools. However, this is only ideal for consuming Web services at fixed locations. If you intend to host a Web services client in the same WebSphere Everyplace Deployment runtime of the Web services provider, and the Web container is selecting a port dynamically, you must register the Web services client `BundleActivator` as an `HttpSettingListener` (package `com.ibm.pvc.webcontainer.listeners.HttpSettingListener`) in order to obtain the port that the Web container is listening on at startup. Then set the endpoint address accordingly. Following is a listing of the changes that need to be made on the project `MANIFEST.MF` file and the `BundleActivator` class, including a sample implementation of the `HttpSettingListener` interface:

MANIFEST.MF:

```
Require-Bundle: ..., com.ibm.pvc.sharedbundle


import com.ibm.pvc.webcontainer.listeners.HttpSettingListener;

public class MyBundleActivator
extends implements BundleActivator, HttpSettingListener
{
 String endpoint = null;

 public void start(BundleContext context) throws Exception
```

```
{
 /* ********************************************************** */
 /* when running a web service client on the same host as a web */
 /* service provider, we must change the endpoint so that it    */
 /* matches the port!!                                          */
 /* ********************************************************** */
 context.registerService(HttpSettingListener.class.getName(),
     this, null);

 /* ********************************************************** */
 /* create the endpoint URL; please note that you will need to  */
 /* make some changes to prevent a race condition with the set- */
 /* ting of httpPort as the registerService call from above is  */
 /* asynchronous so it may not be properly set by the time we   */
 /* execute the line below.                                     */
 /* ********************************************************** */
 endpoint = "http://localhost:" + httpPort + "/ws/pid/echoSvc";

 /* ********************************************************** */
 /* if using a static client, use the following code           */
 /* ********************************************************** */
 // change the endpoint to use the dynamic port which is set below
 EchoServiceSoap_Stub stub = new EchoServiceSoap_Stub();
 stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,
    endpoint);

 /* ********************************************************** */
 /* if using a dynamic client, use the following code          */
 /* ********************************************************** */
 String service = "com.ibm.pvcws.osgi.proxy.WSProxyService";
 ServiceReference ref = bundleContext.getServiceReference(service);

 if (ref == null)
 {
  System.err.println("Error: WSProxyService does not " +
     "exist.");
  return;
 }

 WSProxyService wsManImpl =
  (WSProxyService)context.getService(ref);
 try
 {
  /* register the wsdl at the new endpoint */
  wsManImpl.register(endpoint + "?wsdl");
 }
 catch (Exception e)
 {
  e.printStackTrace();
  return;
 }
}

/**
 * Implement the Web Container Listener interfaces */
 */

private static boolean validSetting = false;
private static int httpPort = 0, httpsPort = 0;
private static final String PROPERTY_HTTP_PORT = "http.port";
private static final String PROPERTY_HTTPS_PORT = "https.port";
private static Map settingsMap = new HashMap();

public void settingsAdded(String pid, Dictionary properties)
{
 Map settings = new HashMap();
 Enumeration enum = properties.keys();
 while (enum.hasMoreElements())
 {
  String key = (String)enum.nextElement();
  settings.put( key, properties.get( key ) );
 }

 settingsMap.put( pid, settings );

 if (!validSetting)
 {
  scanSettings();
 }
}

public void settingsModified(String pid, Dictionary properties)
{
 Map settings = (Map)settingsMap.get( pid );

 if (settings == null)
 {
  settings = new HashMap();
 }

 Enumeration enum = properties.keys();

 while (enum.hasMoreElements())
 {
  String key = (String)enum.nextElement();
  settings.put( key, properties.get( key ) );
 }

 settingsMap.put( pid, settings );

 scanSettings();
}

public void settingsRemoved(String pid)
{
 settingsMap.remove( pid );
 scanSettings();
}

private void scanSettings()
{
 boolean found = false;
 Iterator iter = settingsMap.keySet().iterator();

 while (!found && iter.hasNext())
 {
  Map settings = (Map)settingsMap.get( (String)iter.next() );
  Integer httpPort_ = (Integer)settings.get(
   PROPERTY_HTTP_PORT );

  if (httpPort_ != null)
  {
   httpPort = httpPort_.intValue();
   validSetting = true;
   found = true;
  }

  Integer httpsPort_ = (Integer)settings.get(
   PROPERTY_HTTPS_PORT );

  if (httpsPort_ != null)
  {
   httpsPort = httpsPort_.intValue();
```

```
        validSetting = true;
        found = true;
       }
      }

      if (!found)
      {
       validSetting = false;
       httpPort = 0;
      }
     }
    }
```

Dynamic Mobile Web Services clients

If you generated Web Services client code to use a dynamic stub (i.e., you checked **Create Dynamic Stub**) you will need to register the WSDL of the Web Services provider using the Gateway plug-in. In order to do this, use the `WSProxyServiceFactory` class to get a new instance of `WSProxyService` and invoke one of the following methods at runtime:

```
boolean register(String url);
boolean register(String url, Dictionary properties);
```

Both methods take the URL of a WSDL resource that describes the Web Services provider. The WSDL will be retrieved and parsed to determine the end point of the Web Services provider, the names of the interface to register, and the data structures and methods used by the interface. An automatically generated ″virtual″ OSGi service will then be registered by the Web Services Gateway-using the class name of the interface derived from the WSDL-defined operation.

After the OSGi service is registered, it can be retrieved from the OSGi service registry and used just as any other local OSGi service. The OSGi service will stay registered until the virtual bundle for that service is stopped or uninstalled. If the `unregister()` method of `WSProxyService` is used, the virtual bundle registering the service will be stopped.

Note: When running a dynamic Web services client and a Web services provider in the same runtime, there will be two implementers of the Web service interface (the client and the provider). As a result, the call to retrieve the OSGi service will return one reference to each. You may set a property on the Web services provider `exposeService` method in order to differentiate between the two. Following is as an example on how to set this property on the Web services provider side:

```
private void exposeService(BundleContext context)
{
    Hashtable props = new Hashtable();
    props.put(org.osgi.framework.Constants.SERVICE_PID,
              "MyWebServiceImpl");
    props.put("com.ibm.pvcws.wsdl", "");
    props.put("NAME", "PROVIDER");
    context.registerService(MyWebService.class.getName(),
                            new MyWebServiceImpl(), props);
    WebServiceProvider provider = getProvider(context);
    provider.exportPid("MyWebServiceImpl");
}
```

Following is a Web services client side code example using a dynamic stub that differentiates between invoking the Web services provider directly or through the Web services runtime stubs:

```
public void start(BundleContext context) throws Exception
{
WSProxyService wsManImpl = WSProxyServiceFactory.newInstance(context, null);

try
{

    // NOTE: since by default the web container listens on a random port,
    // refer to the section Static Web Services clients for information on
    // how to obtain the chosen port programmatically and use that
    // information to form the WSDL URL.

    wsManImpl.register("http://localhost:8777/ws/pid/MyWebServicesImpl?wsdl);

    ServiceTracker tracker = new ServiceTracker(context,
               MyWebService.class.getName(), null);

    tracker.open();

    ServiceReference refs[] = tracker.getServiceReferences();

    for (int i=0; i<refs.length; i++)
    {
            String name = refs[i].getProperty("NAME");
            MyWebService stub = (MyWebService)context.getService(refs[i]);

            if ((name != null) && (name.equals("PROVIDER")))
            {
                /* this is a direct call into the service and is more
                   efficient as it would not execute the web services
                   stubs
                */
                System.out.println("NAME= + stub.getName());
            }
            else
            {
                /* this is an indirect call into the service which is
                   less efficient as it would execute the web services
                   stubs
                */
                System.out.println("NAME= + stub.getName());
             }
         }

    tracker.close();
}
catch (Exception e)
{
    e.printStackTrace();
}
```

If the Web Services client needs to handle non-bean classes or types that are incompatible with JSR-172 style Web Services, custom marshallers need to be implemented. Please refer to the section "Custom serialization (marshalling)" for more information.

Editing Mobile Web Services

Custom serialization (marshalling)

Custom marshalling is required for handling non-bean classes as well as types that are incompatible with JSR-172. Custom marshalling can be implemented for both Web Services providers and Web Services clients. However, this release of the IBM

WebSphere Everyplace Client Toolkit does not automatically generate custom marshaller stubs for Web Services providers; it only does for Web Services clients.

When custom marshalling is required, the developer is responsible for completing the code within the generated `MarshalFactory` and `Marshaller` classes on the Web services client. If custom marshallers are needed for a Web Services provider, the Web Services client custom marshaller code can be reproduced on the Web Services provider without change.

This section describes the Application Program Interfaces (API) used in the serialization of data structures that cannot be automatically serialized by the Web Services engine for OSGi.

The current Web Services engine uses Java reflection and conventions similar to the functionality described in JSR 101 to automatically generate a WSDL to describe services and to serialize data structures for wire transmission. However, in order for the conventions to work, the data structures must have a bean-like structure. Unfortunately, not all data structures have this structure and require specific logic to be serialized and described.

The serialization API has the following features:
- The programmer does not deal with XML processing
- Only the structures that need special processing need custom marshallers

Using the WebSphere Everyplace Client Toolkit, it is easy to write classes that can automatically be serialized. However, when working with legacy classes and classes that contain logic, complications arise. Very few of the standard Java classes lend themselves to automatic serialization. For example, consider the `java.util.Properties` class and the `java.util.Calendar` class. Neither of these two classes follows the WebSphere Everyplace Deployment Web Services conventions. Furthermore, there is a standard mapping of the `Calendar` class into the XML Schema namespace, so even the namespace mapping conventions do not apply. The serialization API allows data structures to be introspected.

It has three parts:
- Class name to QName mapping
- Enumeration of the members that make up a class
- Extractions of members from an instance of a class

As it is an introspection API, the programmer does not need to be involved in the generation of XML for the WSDL or the SOAP messages. In fact, the actual encoding can be changed since none of the encoding constructs manifest themselves in the API.

Another design feature of the serialization API is that it is only needed for those classes that cannot be processed using the standard conventions. For example, when the marshaller of the `Properties` class is written, it is represented as an array of `PropertyEntry` objects where `PropertyEntry` is a class that has two public members, a key and a value, both of type String. Because `PropertyEntry` follows the normal convention for writing serializable data structures it is not necessary to write a custom marshaller for it.

Custom marshallers implement the `ClassDescriberMarshalFactory` interface and are registered in the OSGi service registry. The Web services engine dynamically looks up the marshallers and invokes them when needed. This allows for easy deployment of new marshallers and non-disruptive removal of unnecessary marshallers.

Note: Automatic generation of custom marshallers is not possible when creating Web Services providers, only when creating Web Services clients. However, the same code can be reused on the Web Services provider without change.

MarshalFactory

When the Web Services engine needs to map a class to a `QName` or get a Marshal for a `QName`, it relies on the `MarshalFactory` class for help. The `MarshalFactory` class consists of two methods:

```
public interface MarshalFactory {
    Class getClassForQName(ClassLoader cl, QName qtype);
    Marshal getMarshaller(QName qtype);
}
```

Marshal

The Marshal interface has only one method and should not be implemented directly. Either a `SimpleMarshal` or `ComplexMarshal` should be implemented instead. Each time a new object needs to be serialized, the Web Services engine will call `MarshalFactory.getMarshaller()`. When servicing the call, the `MarshalFactory` can create a Marshal object to handle the specific object, or it can reuse an already created object. The Web Services engine interacts with the two Marshal types in different ways. The engine selects the appropriate behavior by checking the class of the Marshal returned from `getMarshaller()`.

ClassDescriberMarshalFactory

The `ClassDescriberMarshalFactory` is an interface that extends both `MarshalFactory` and `CustomClassDescriber`. It does not add any new methods. Bundles must register services using the `ClassDescriberMarshalFactory` in the service registry to provide custom serializers. Serializers that only register services under the `MarshalFactory` and `CustomClassDescriber` interfaces will be ignored by the web service engine. Examination of the information provided by the `CustomClassDescriber` and the `MarshalFactory` shows that there is some overlap between the information provided by the two interfaces. It is the responsibility of the implementer to ensure that the information is consistent.

Examples

This section contains two examples of custom marshallers. The first example shows a `SimpleMarshal` for Calendar, and highlights the use of QName mapping to map Calendar to `xsd:dateTime`, a mapping defined in JAX-RPC. The second example shows a `ComplexMarshal` for Properties, and also illustrates the use of an intermediate class and the use of QName mapping to return a non imported class.

Calendar example: The serialization of Calendar is illustrated in two parts: the implementation of `CalendarMarshalFactory`, and the implementation of `CalendarMarshal`.

CalendarMarshalFactory

You must register a service of type `ClassDescriberMarshalFactory`, so that it is the interface that the `CalendarMarshalFactory` implements. First, you must setup static member variables to aid in your processing:

```
static QName calendarType = new QName(NamespaceConstants.NSURI_SCHEMA_XSD,
                                      "dateTime");
```

```
static ClassDescriptor cdCalendar = new ClassDescriptor(calendarType, 1, 1, true);

static ClassDescriptor cdCalendarArray = new ClassDescriptor(calendarType, 1,
   Integer.MAX_VALUE, true);

static PartsDescriptor parts = new PartsDescriptor
                                 (new QName[] { new QName("entries") ],
                                  new Class[] { PropertyEntry[].class });
```

Now you can write the methods that correspond to ClassDescriber:

```
public boolean canDescribe(Class c) {
   if (c.isArray()) c =
          c.getComponentType();
   return c.equals(Calendar.class);
}
```

In the canDescribe() method you can describe both Calendar and Calendar[]. Ensure you check for both.

```
public ClassDescriptor getQType(Class c) {
  boolean isArray = false;
  if (c.isArray()) {
    c = c.getComponentType();
    isArray = true;
  }
  if (c.equals(Calendar.class)) {
    return isArray ? cdCalendar :
      cdCalendarArray;
  }
  return null;
}
```

This checks to see if the class in question is Calendar and returns the precomputed ClassDescriptors if appropriate. Observe that the only difference between cdCalendar and cdCalendarArray is that the maxOccurs value is 1 in the non-array case, and MAX_VALUE in the array case.

```
public PartsDescriptor getParts(Class c) {
  return null;
}
```

getParts() always returns null since even if c == Calendar.class. The Calendar class is a simple class and therefore has no parts.

```
public Class getClassForQName(ClassLoader cl, QName qtype)
{
  return calendarType.equals(qtype) ?
    Calendar.class : null;
}
```

This method handles QName mapping. It simply returns the Calendar class if the qtype matches xsd:dateTime.

```
static SimpleMarshal calMarshal = new CalendarMarshal();

Marshal getMarshaller(QName qType)
{
  return calendarType.equals(qType) ?
    calMarshal : null;
}
```

This is the final method to be implemented. In this method, the system returns a precomputed SimpleMarshal for Calendar if the qType is xsd:dateTime. The CalendarMarshal has no reusable state, even if it is used concurrently.

CalendarMarshal

When Calendar is encoded as `xsd:dateTime` it has no members, so you must use a `SimpleMarshal`.

```
public Object deserialize(String value) {
  try {
    Date date = new SimpleDateFormat().parse(value);
    Calendar cal = new GregorianCalendar();
    cal.setTime(date);
    return cal;
  } catch (ParseException e) {
    e.printStackTrace();
  }
  return null;
}
```

To deserialize a Calendar object, take the given string and use `SimpleDateFormat` to create a Calendar. Note that even though the return value is `Object`, we must return a `Calendar` since that is the type this Marshal handles.

```
public String serialize(Object o)
{
  Calendar cal = (Calendar)o;
  String date = new SimpleDateFormat().format(cal.getTime());
  return date;
}
```

In the `serialize()` method we know that `o` is of type `Calendar` since that is the type this Marshal handles. You must convert the string to use `SimpleDateFormat()` and return the result.

```
public Object newArray(int size)
{
  return new Calendar[size];
}
```

The final method returns an array of the correct size. The `Calendar` serializer is ready to register in the service registry.

Properties example: The serialization of Properties is illustrated in two parts: the implementation of `PropertiesMarshalFactory`, and then of `PropertyMarshal`.

This example covers the serializer for Properties. As explained, the Properties class is serialized using an intermediate class called `PropertyEntry`:

```
public class PropertyEntry {
  public String key;
  public String value;
}
```

Because `PropertyEntry` is a class that is easily serialized, you don't need to write a serializer for it. However, since it is an intermediate class, you must return the `PropertyEntry.class` in the QName mapping. In addition, since the conventional class name is used for QName mapping, you don't need a mapping for Properties. It is easy to get confused because the `PropertiesMarshalFactory` maps `PropertyEntry`, a class it is only using internally, but not Properties, the class it is actually serializing. Remember from the QName mapping discussion that mapping is used when the QName doesn't correspond to the convention or the class isn't a class that will normally be imported by the virtual bundle or the service. Properties does not meet either case, but `PropertyEntry` meets the second case. The `getClassForQName` is defined as follows:

```
static QName propertyEntryQType = DefaultMarshalFactory.defaultGetClassQName
    (PropertyEntry.class.getName());

public Class getClassForQName(ClassLoader cl, QName qtype)
{
  return qtype.equals
    (propertyEntryQType) ?
    PropertyEntry.class:null;
}
```

Because the `PropertyMarshalFactory` is almost exactly the same as `CalendarMarshalFactory`, only the `getParts()` method is shown. It differs substantially:

```
static PartsDescriptor parts = new PartsDescriptor(new QName[]
    { new QName("entries") },
    new Class[] { PropertyEntry[].class });

public PartsDescriptor getParts(Class c) {
  if (c.isArray()) c = c.getComponentType();
  if (c.equals(Properties.class)) {
    return parts;
  }
  return null;
}
```

The Properties class is encoded as a complex type with one member (entries) that is an array of `PropertyEntrys`. As a result, the `PartsDescriptor` for Properties is made up of two arrays of one entry. The first array has the name of the members, and the second the class of the members. The `PropertiesMarshal` implements `ComplexMarshal` and has the following members:

```
public Object newArray(int length)
{
  return new Properties[length];
}
```

As with Calendar, this is a simple method that returns an array of the specified size.

```
public Object newHandle() {
  return new Properties();
}
```

Since Properties has a default constructor and has methods to add to it, it can return an instance of Properties.

```
{
  PropertyEntry entries[] = (PropertyEntry[])value;
  for(int i = 0; i < entries.length; i++) {
    ((Properties)handle).put(entries[i].key, entries[i].value);
  }
}
```

As there is only one member for Properties, there is no need to check the index. You must only put the received entries into the Properties object.

```
public Object newInstance(Object handle) {
  return handle;
}
```

There is nothing to do in this routine since handle is already a Properties object.

```
public Object getMember(Object obj, int index) {
  Properties props = (Properties)obj;
  Vector entries = new Vector();
  Enumeration en = props.keys();
  while(en.hasMoreElements()) {
    PropertyEntry entry = new PropertyEntry();
    entry.key = (String)en.nextElement();
    entry.value = props.getProperty(entry.key);
    entries.add(entry);
  }
  return entries.toArray(new PropertyEntry[0]);
}
```

As there is only one member, the index does not need to be checked. Since you are generating the entries member, you must build a `PropertyEntry[]` and return it.

```
public int getMemberCount() {
  return 1;
}
```

There is only one member.

```
QPart entriesPart = new QPart(new QName("entries"),
    DefaultMarshalFactory.defaultGetClassQName(PropertyEntry.class.getName()), 0,
    Integer.MAX_VALUE, true, true);

public QPart getPart(int index) {
  return entriesPart;
}
```

Return the QPart that describes the entries member. The first argument of the Qpart constructor is the standard mapping of the QName. The second says that the array can be an empty array. The third says that it is an unbounded array. The fourth says it can be null. The fifth indicates whether the QPart needs to be qualified (always set it to true).

Final steps: Now that the serializers are prepared, you must register them in the service registry so that they can be used. This is done by putting the classes in a bundle and writing a `BundleActivator` for the bundle:

```
public class MarshalBundleActivator implements BundleActivator {
  public void start(BundleContext bc) throws Exception {
    bc.registerService(
      ClassDescriberMarshalFactory.class.getName(),
      new PropertyMarshalFactory(),
      null);

    bc.registerService(
      ClassDescriberMarshalFactory.class.getName(),
      new CalendarMarshalFactory(),
      null);
  }

  public void stop(BundleContext bc) {}
}
```

To register the serializers, construct the two factories and register them as `ClassDescriberMarshalFactory` services.

Securing Mobile Web Services

Securing Mobile Web Services providers

In order to secure Web Services providers, you must first create the Web Service provider following the instructions in “Creating Mobile Web Services providers” on page 97 and deploy the Web Services provider locally following the instructions in “Deploying Mobile Web Services providers” on page 141. This is required because the WSDL of the Web Service provider is needed to enable security. Once the Web Services provider is running in a local instance of the WebSphere Everyplace Deployment, follow these instructions to secure the Web Services provider:

1. Launch the Web Services provider security configuration wizard:
   a. Select **File > New > Other**.
   b. Select **Client Services > Mobile Web Services > Mobile Web Services Provider Security Configuration**.
   c. Select **Next**.
   d. Enter the source folder containing the Java source code for the Web Services provider application.
   e. Enter the URL of the WSDL exposed by the Web Services provider.
   f. Click **Next**.
2. Configure Web Services Security:
   a. Select the Web Services name and port from the drop down menu.
   b. Under 'How to create Web Services Security configuration', select the appropriate choice. For a test case, select Template configuration, then select the Client type and the Security template from the drop down menus.
      - **Use other WebSphere Everyplace Deployment configuration**

        If you browse a folder including the existing WS-Security configurations for the WebSphere Everyplace Deployment under the other existing project, you can import the configurations into the working project.
      - **Import WAS 6.0 configuration**

        If you browse a folder including the existing WS-Security configurations for the WAS 6.0 client, you can import the configurations into the working project.
      - **Use template configuration**

        If you select both the appropriate server type and the appropriate security template, you can use the predefined configurations in the working project.
   c. Select **Finish**.

After completing the above procedure, the WS-Security Provider Editor appears. Some files may be created or modified, as follows:

- WS-Security-related code is inserted in the file `BundleActivator.java` under the working project
- The files `ibm-webservices-ext.xmi`, `ibm-webservices-bnd.xmi`, `serverSample.jks`, and `wssecurity.xml` are generated in the export directory of the package containing the file `BundleActivator.java`.

At this point, you may re-deploy your Web Services provider with security enabled, but if you want to change the WS-Security configurations in the working project, you can edit them with the WS-Security Provider Editor. To understand more details of how to edit the WS-Security configurations, please refer to "Editing the Mobile Web Services security configuration" on page 112.

Securing Mobile Web services clients

In order to secure Web services clients, refer to "Creating Mobile Web Services clients" on page 98. Ensure that you check **Configure Security**. Perform the following procedure to configure Web services security for a secure Web services client:

1. Select the Web services name and port from the drop down menu.
2. Under 'How to create Web Services Security configuration', select the appropriate choice. For a test case, select **Template configuration**, then select the Client type and the Security template from the drop down menus.
   - **Use other WebSphere Everyplace Deployment configuration**

     If you browse a folder including the existing WS-Security configuration for the WebSphere Everyplace Deployment under the other existing project, you can import the configurations into the working project.
   - **Import WAS 6.0 configuration**

     If you browse a folder including the existing WS-Security configurations for the WAS 6.0 client, you can import the configurations into the working project.
   - **Convert WCTME 5.8 class-based configuration**

     If you browse a WS-Security configuration java file for WCTME 5.8 or 5.7, you can convert the configuration for use with the WebSphere Everyplace Deployment 6.0.
   - **Use template configuration**

     If you select both the appropriate server type and the appropriate security template, you can use the predefined configurations in the working project.
3. Select **Finish**.

After completing the above procedure, the WS-Security Client Editor appears. Some files may be created or modified, as follows:

If the Web services client is static and you specified a package in the Mobile Web Services Client wizard

- The port files '`*.java`' and '`*_Stub.java`' are generated in the specified package. The name of the port file is taken from the `portType` in the WSDL file
- WS-Security related code is inserted in '`*_Stub.java`'
- The files `ibm-webservicesclient-ext.xmi`, `ibm-webservicesclient-bnd.xmi`, `clientSample.jks`, and `wssecurityclient.xml` are generated in the export directory in the specified package

If the Web services client is static and you did not specify a package in the Mobile Web Services Client wizard

- The port files '`*.java`' and '`*_Stub.java`' are generated in the package specified from the contents of the WSDL file
- WS-Security-related code is inserted in the file '`*_Stub.java`'
- The files `ibm-webservicesclient-ext.xmi`, `ibm-webservicesclient-bnd.xmi`, `clientSample.jks`, and `wssecurityclient.xml` are generated in the export directory in the package specified from the WSDL file

If the Web services client is dynamic

- A port file is generated in the package specified in the WSDL file
- WS-Security-related code is inserted in the file `BundleActivator.java` under the working project
- The files `ibm-webservicesclient-ext.xmi`, `ibm-webservicesclient-bnd.xmi`, `clientSample.jks`, and `wssecurityclient.xml` are generated in the export directory in the package specified from the WSDL file

Editing the Mobile Web Services security configuration

In order to understand how to edit the Mobile Web Services security configuration in the working project, this section introduces seven scenarios described by OASIS Web Services Security (WSS) TC. These seven message exchange scenarios are intended to test the interoperability of different implementations performing common operations, and to test the clarity of their meaning and proper application. To avoid confusion, they are called Scenario #1 through Scenario #7.

- **Scenario #1: Basic Authentication**

  The request header contains a Username and Password. The response does not contain a security header.
- **Scenario #2: Basic Authentication with Encryption**

  The request header contains a Username and Password that have been encrypted using a public key provided out-of-band. The response does not contain a security header.
- **Scenario #3: Sign and Encrypt**

  The request body contains data that has been signed and encrypted. The certificate used to verify the signature is provided in the header. The certificate associated with the encryption is provided out-of-band. The response body is also signed and encrypted, reversing the roles of the key pairs identified by the certificates.
- **Scenario #4: Session Key**

  The request body contains data that has been signed and encrypted. The certificate used to verify the signature is provided in the header. The symmetric key used to perform the encryption is provided out-of-band. The response body is also signed and encrypted. The same symmetric key is used to perform the encryption. The certificate used to verify the signature is provided out-of-band.
- **Scenario #5: Overlapping Signature**

  The request body contains data that has been signed twice. First the ticket element is signed. The certificate used to verify this signature is provided out-of-band. Next the entire body is signed. The certificate used to verify this signature is provided in the header. The response body is not signed or encrypted.
- **Scenario #6: Encrypt and Sign**

  The request body contains data that has been encrypted and signed. The certificate associated with the encryption is provided out-of-band. The certificate used to verify the signature is provided in the header. The response body is also encrypted and signed, reversing the roles of the key pairs identified by the certificates.
- **Scenario #7: Signed Token**

  The request body contains data that has been signed and encrypted. The signature also protects an enclosed security token by means of the STR Dereference Transform. The certificate used to verify the signature is provided in the header. The certificate associated with the encryption is provided out-of-band. The response body is also signed and encrypted, reversing the roles of the key pairs identified by the certificates.

To understand more details of the scenarios, see the following materials:

- Web Services Security Interop 1 Scenarios (Scenario #1 through Scenario #3): http://www.oasis-open.org/committees/download.php/2362/wss-intero
- Web Services Security Interop 2 Scenarios (Scenario #4 through Scenario #7): http://www.oasis-open.org/committees/download.php/11375/wss-inter

WebSphere Everyplace Deployment Web Services runtimes and the WebSphere Everyplace Client Toolkit Web Services tools do not support Scenario #4, #5, and #7. As a result, this section only describes how to edit Mobile Web Services Security Configuration for scenarios #1, #2, #3, and #6.

Prerequisites

This section assumes that you have removed all Web Services Security configurations. To do so, perform the following procedures as necessary:

To remove the Mobile Web Services Security Configuration for the Web Services Provider:

1. Open the WS-Security Provider Editor.
2. Expand **Server Service Configuration** on the WS extension tab, and delete the **Actor URI**.
3. Expand **WebSphere Application Server 5.x** on the WS extension tab, and uncheck the **Access the WebSphere Application Server 5.x client** check box.
4. Expand **Request Consumer Service Configuration Details** on the WS extension tab:
   a. Expand **Required Integrity** and remove all items in the list.
   b. Expand **Required Confidentiality** and remove all items in the list.
   c. Expand **Required Security Token** and remove all items in the list.
   d. Expand **Caller Part** and remove all items in the list.
   e. Expand **Add Timestamp** and uncheck the **Use Add Timestamp** check box.
   f. Expand **Property** and remove all items in the list.
5. Expand **Response Generator Service Configuration Details** on the WS extension tab:
   a. Expand **Details** and delete the Actor.
   b. Expand **Integrity** and remove all items in the list.
   c. Expand **Confidentiality** and remove all items in the list.
   d. Expand **Security Token** and remove all items in the list.
   e. Expand **Add Timestamp** and uncheck the **Use Add Timestamp** check box.
   f. Expand **Property** and remove all items in the list.
6. Expand **Request Consumer Binding Configuration Details** on the WS binding tab:
   a. Expand **Trust Anchor** and remove all items in the list.
   b. Expand **Token Consumer** and remove all items in the list.
   c. Expand **Key Locators** and remove all items in the list.
   d. Expand **Key Information** and remove all items in the list.
   e. Expand **Signing Information** and remove all items in the list.
   f. Expand **Encryption Information** and remove all items in the list.
   g. Expand **Property** and remove all items in the list.
7. Expand **Response Generator Binding Configuration Details** on the WS binding tab:
   a. Expand **Token Generator** and remove all items in the list.
   b. Expand **Key Locators** and remove all items in the list.
   c. Expand **Key Information** and remove all items in the list.
   d. Expand **Signing Information** and remove all items in the list.
   e. Expand **Encryption Information** and remove all items in the list.

f. Expand **Property** and remove all items in the list.

To remove the Mobile Web Services Security Configuration for the Web Services Client:

1. Open the WS-Security Client Editor.
2. Expand **Client Service Configuration Details** on the WS extension tab, and delete the **Actor URI**.
3. Expand **WebSphere Application Server 5.x** on the WS extension tab, and uncheck the **Access the WebSphere Application Server 5.x provider** check box.
4. Expand **Request Generator Configuration** on the WS extension tab:
   a. Expand **Details** and delete the **Actor**.
   b. Expand **Integrity** and remove all items in the list.
   c. Expand **Confidentiality** and remove all items in the list.
   d. Expand **Security Token** and remove all items in the list.
   e. Expand **Add Timestamp** and uncheck the **Use Add Timestamp** check box.
   f. Expand **Property** and remove all items in the list.
5. Expand **Response Consumer Configuration** on the WS extension tab:
   a. Expand **Required Integrity** and remove all items in the list.
   b. Expand **Required Confidentiality** and remove all items in the list.
   c. Expand **Required Security Token** and remove all items in the list.
   d. Expand **Add Timestamp** and uncheck the **Use Add Timestamp** check box.
   e. Expand **Property** and remove all items in the list.
6. Expand **Security Request Generator Binding Configuration** on the WS binding tab:
   a. Expand **Token Generator** and remove all items in the list.
   b. Expand **Key Locators** and remove all items in the list.
   c. Expand **Key Information** and remove all items in the list.
   d. Expand **Signing Information** and remove all items in the list.
   e. Expand **Encryption Information** and remove all items in the list.
   f. Expand **Property** and remove all items in the list.
7. Expand **Security Response Consumer Binding Configuration** on the WS binding tab:
   a. Expand **Trust Anchor** and remove all items in the list.
   b. Expand **Token Consumer** and remove all items in the list.
   c. Expand **Key Locators** and remove all items in the list.
   d. Expand **Key Information** and remove all items in the list.
   e. Expand **Signing Information** and remove all items in the list.
   f. Expand **Encryption Information** and remove all items in the list.
   g. Expand **Property** and remove all items in the list.

Editing a Mobile Web Services security configuration for basic authentication (scenario #1)

To edit a Mobile Web Services security configuration in this scenario for the Web Services client, perform the following procedure:

1. Open the WS-Security Client Editor.
2. Expand the **Request Generator Configuration** section of the WS extension tab.
3. Expand the **Security Token** section and select **Add**. From the next menu:

   a. Input an appropriate Name.
   b. Select **Username token** as a Token type.
   c. Select **OK**.
4. Expand the **Security Request Generator Binding Configuration** section of the WS binding tab.
5. Expand the **Token Generator** section and select **Add**.
   a. Input an appropriate Token generator name.
   b. Select **com.ibm.pvcws.wss.internal.token.UsernameTokenGenerator** as a Token generator class.
   c. Select the security token name you specified as the **Security token**.
   d. Select **com.ibm.pvcws.wss.internal.auth.callback.NonPromptCallbackHandler** as a Call back handler.
   e. Input an appropriate User ID and Password.
   f. Select **OK**.
6. Save your changes.

To edit a Mobile Web Services security configuration in this scenario for the Web Services provider, perform the following procedure:
1. Open the WS-Security Provider Editor.
2. Expand the **Request Consumer Service Configuration Details** section of the WS extension tab.
3. Expand the **Required Security Token** section and select **Add**. From the next menu:
   a. Input an appropriate Name.
   b. Select **Username token** as a Token type.
   c. Select **Required** as a Usage type.
   d. Select **OK**.
4. Expand the **Caller Part** section and select **Add**. From the next menu:
   a. Input an appropriate Name.
   b. Select **Username token** as a Token type.
   c. Select **OK**.
5. Expand the **Request Consumer Binding Configuration Details** section of the WS binding tab.
6. Expand the **Token Consumer** section and select **Add**. From the next menu:
   a. Input an appropriate Token consumer name.
   b. Select **com.ibm.pvcws.wss.internal.token.UsernameTokenConsumer** as a Token consumer class.
   c. Select the security token name you specified as the **Security token**.
   d. Check the **Use value type** check box.
   e. Select **Username token** as a Value type.
   f. Select **OK**.
7. Save your changes.

Editing a Mobile Web Services security configuration for basic authentication with encryption (scenario #2)

To edit the Mobile Web Services security configuration in this scenario for the Web services client, perform the following procedure:

1. Open the WS-Security Client Editor.
2. Expand the **Request Generator Configuration** section of the WS extension tab.
3. Expand the **Security Token** section and select **Add**. From the next menu:
   a. Input an appropriate Name.
   b. Select **Username token** as a Token type.
   c. Select **OK**.
4. Expand the **Confidentiality** section and select **Add**: From the next menu:
   a. Input an appropriate Confidentiality Name.
   b. Select **1** as an Order.
   c. Select **Add**.
   d. Select **usernametoken** as a Message parts keyword.
   e. Select **OK**.
5. Expand the **Security Request Generator Binding Configuration** of the WS binding tab.
6. Expand the **Token Generator** section and select **Add**. From the next menu:
   a. Input an appropriate Token generator name.
   b. Select **com.ibm.pvcws.wss.internal.token.UsernameTokenGenerator** as a Token generator class.
   c. Select the security token name you specified as the **Security token**.
   d. Select **com.ibm.pvcws.wss.internal.auth.callback.NonPromptCallbackHandler** as a Call back handler.
   e. Input an appropriate User ID and Password.
   f. Select **OK**.
7. Expand the **Key locators** section and select **Add** to create a key locator used for encryption:
   a. Input an appropriate Key locator name.
   b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
   c. Check the **Use key store** check box.
   d. Input an appropriate Key store storepass and Key store path.
   e. Select **JKS** or **JCEKS** as a Key store type.
   f. Select **Add** to specify the key used for encryption.
   g. Input an appropriate Alias and Key name for the encryption key.
   h. Select **OK**.
8. Expand the **Key information** section and select **Add** to create the key information used for encryption:
   a. Input an appropriate Key information name.
   b. Select **KEYID** as the Key information type.
   c. Select 'com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentGenerator' as a Key information class.
   d. Check Use key locator check box.
   e. Select the key locator name you specified as the **Key locator**.
   f. Select the key name you specified as the **Key**.
   g. Select OK.

9. Expand the Encryption Information section and select **Add**. From the next menu:
   a. Input an appropriate Encryption name.
   b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as a Data encryption method algorithm.
   c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as a Key encryption method algorithm.
   d. Input an appropriate Key information name.
   e. Select the key information name you specified as the **Key information element**.
   f. Select the confidentiality name you specified as the **Confidentiality element**.
   g. Select **OK**.
10. Save your changes.

To edit the Mobile Web Services security configuration in this scenario for the Web services provider, perform the following procedure:

1. Open the WS-Security Provider Editor.
2. Expand the **Request Consumer Service Configuration Details** section of the WS extension tab.
3. Expand the **Required Security Token** section and select **Add**. From the next menu:
   a. Input an appropriate Name.
   b. Select **Username token** as a Token type.
   c. Select **Required** as a Usage type.
   d. Select **OK**.
4. Expand Caller Part section and select **Add**. From the next menu:
   a. Input an appropriate Name.
   b. Select **Username token** as a Token type.
   c. Select **OK**.
5. Expand Required Confidentiality section and select **Add**. From the next menu:
   a. Input an appropriate Required Confidentiality Name.
   b. Select **Required** as a Usage type.
   c. Select **Add**.
   d. Select **usernametoken** as a Message parts keyword.
   e. Select **OK**.
6. Expand the **Request Consumer Binding Configuration Details** section of the WS binding tab
7. Expand the **Token Consumer** section and select Add. From the next menu:
   a. Input an appropriate Token consumer name.
   b. Select **com.ibm.pvcws.wss.internal.token.UsernameTokenConsumer** as a Token consumer class.
   c. Select the security token name you specified as the **Security token**.
   d. Check the **Use value type** check box.
   e. Select **Username token** as a Value type.
   f. Select **OK**.
8. Expand the **Token Consumer** section and select **Add**. From the next menu:

   a. Input an appropriate Token consumer name.
   b. Select **com.ibm.pvcws.wss.internal.token.X509TokenConsumer** as a Token consumer class.
   c. Check the **Use value type** check box.
   d. Select the **X509 certificate token** as a Value type.
   e. Check the **User certificate path settings** check box.
   f. Check the **Trust any certificate** radio button.
   g. Select **OK**.
9. Expand the **Key Locators** section and select **Add** to create a key locator used for decryption. From the next menu:
   a. Input an appropriate Key locator name.
   b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
   c. Check the **Use key store** check box.
   d. Input the appropriate Key store storepass and Key store path.
   e. Select **JKS** or **JCEKS** as a Key store type.
   f. Select **Add** to specify the key used for decryption.
   g. Input the appropriate Alias, Key pass, and Key name to specify the decryption key.
   h. Select **OK**.
10. Expand the **Key Information** section and select **Add** to create the key information used for decryption. From the next menu:
    a. Input an appropriate Key information name.
    b. Select **KEYID** as a Key information type.
    c. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentConsumer** as a Key information class.
    d. Check the **Use key locator** check box.
    e. Select the key locator name you specified as the **Key locator**.
    f. Check the **Use token** check box.
    g. Select the token consumer name you specified as the **Token Consumer**.
    h. Select **OK**.
11. Expand the **Encryption Information** section and select **Add**. From the next menu:
    a. Input an appropriate Encryption name.
    b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as a Data encryption method algorithm.
    c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as a Key encryption method algorithm.
    d. Select **Add** to add the key information.
    e. Input an appropriate Key information name.
    f. Select the key information name you specified as the **Key information** element.
    g. Select the required confidentiality name you specified as the **Required Confidentiality** element.
    h. Select **OK**.
12. Save your changes.

Editing a Mobile Web Services security configuration for sign and encrypt (scenario #3)

To edit a Mobile Web Services security configuration in Scenario 3 for the Web Services client, perform the following procedure:

1. Open the WS-Security Client Editor.
2. Expand the **Request Generator Configuration** section from the WS extension tab.
3. Expand the **Integrity** section and select **Add**. From the next menu:
   a. Input an appropriate Integrity Name.
   b. Select **1** as the Order.
   c. Select **Add**.
   d. Select **body** as the Message parts keyword.
   e. Select **OK**.
4. Expand the **Confidentiality** section and select Add. From the next menu:
   a. Input an appropriate Confidentiality Name.
   b. Select **2** as the Order.
   c. Select **Add**.
   d. Select **bodycontent** as the Message parts keyword.
   e. Select **OK**.
5. Expand the **Add Timestamp** section. From the next menu:
   a. Check the **Use Add Timestamp** check box.
   b. Expand the **Property** section from the Add Timestamp section and select **Add**.
   c. Input `com.ibm.pvcws.wss.timestamp.dialect` as a Property Name and `http://www.ibm.com/wct/webservices/wssecurity/dialect-predefined` as a Property Value.
   d. Expand the **Property** section from the Add Timestamp section and select **Add**.
   e. Input `com.ibm.pvcws.wss.timestamp.keyword` as a Property Name and `SOAPHeaderFirst` as a Property Value.
6. Expand the **Request Consumer Configuration** section from the WS extension tab.
7. Expand the **Required Integrity** section and select **Add**. From the next menu:
   a. Input an appropriate Required Integrity Name.
   b. Select '**Required** as the Usage type.
   c. Select **Add**.
   d. Select **body** as the Message parts keyword.
   e. Select **OK**.
8. Expand the **Required Confidentiality** section and select Add. From the next menu:
   a. Input an appropriate Required Confidentiality Name.
   b. Select **Required** as the Usage type.
   c. Select **Add**.
   d. Select **bodycontent** as the Message parts keyword.
   e. Select **OK**.
9. Expand the **Add Timestamp** section and check the **Use Add Timestamp** check box. From the next menu:

    a. Expand the **Property** section from the **Add Timestamp** section and select **Add**.
    b. Input `com.ibm.pvcws.wss.timestamp.dialect` as a Property Name and `http://www.ibm.com/wct/webservices/wssecurity/dialect-predefined` as a Property Value.
    c. Expand the **Property** section from the Add Timestamp section and select **Add**.
    d. Input `com.ibm.pvcws.wss.timestamp.keyword` as a Property Name and `SOAPHeaderFirst` as a Property Value.
10. Expand the **Security Request Generator Binding Configuration** section from the WS binding tab.
11. Expand the **Token Generator** section and select **Add** to insert the certificate for digital signature verification. From the next menu:
    a. Input an appropriate Token generator name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenGenerator** as the Token generator class.
    c. Check the **Use value type** check box.
    d. Select **X509 certificate token** as the Value type.
    e. Select **com.ibm.pvcws.wss.internal.auth.callback.X509CallbackHandler** as the Call back handler.
    f. Check the **Use key store** check box.
    g. Input an appropriate Key store storepass and Key store path.
    h. Select **JKS** or **JCEKS** as a Key store type.
    i. Select **Add** to specify the certificate to be inserted into the message.
    j. Input the appropriate Alias and Key name to specify the certificate to be inserted into the message.
    k. Select **OK**.
12. Expand the **Key locators** section and select **Add** to create a key locator used for digital signature. From the next menu:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
    c. Check the **Use key store** check box.
    d. Input an appropriate Key store storepass and Key store path.
    e. Select **JKS** or **JCEKS** as a Key store type.
    f. Select **Add** to specify the key used for digital signature.
    g. Input an appropriate Alias, Key pass , and Key name to specify the key used for digital signature.
    h. Select **OK**.
13. Expand the **Key locators** section and select **Add** again to create a key locator used for encryption. From the next menu:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
    c. Check the **Use key store** check box.
    d. Input an appropriate Key store storepass and Key store path.
    e. Select **JKS** or **JCEKS** as a Key store type.
    f. Select **Add** to specify the key used for encryption.

   g. Input an appropriate Alias and Key name to specify the key used for encryption.
   h. Select **OK**.
14. Expand the **Key information** section and select **Add** to create the key information used for a digital signature. From the next menu:
   a. Input an appropriate Key information name.
   b. Select **STRREF** as a Key information type.
   c. Select **com.ibm.pvcws.wss.internal.keyinfo.STRRefContentGenerator** as a Key information class.
   d. Check the **Use key locator** check box.
   e. Select the key locator name you specified as the **Key locator**.
   f. Select the key name you specified as the **Key name**.
   g. Check the **Use token** check box.
   h. Select the token name you specified as the **Token generator**.
   i. Select **OK**.
15. Expand the **Key information** section and select **Add** again to create the key information used for encryption:
   a. Input an appropriate Key information name.
   b. Select **KEYID** as a Key information type.
   c. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentGenerator** as a Key information class.
   d. Check the **Use key locator** check box.
   e. Select the key locator name you specified as the **Key locator**.
   f. Select the key name you specified as the **Key**.
   g. Select **OK**.
16. Expand the **Signing Information** section and select Add. From the next menu:
   a. Input an appropriate Signing information name.
   b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Canonicalization method algorithm.
   c. Select **http://www.w3.org/2000/09/xmldsig#rsa-sha1** as a Signature method algorithm.
   d. Input an appropriate Key information name.
   e. Select the key information name you specified for the digital signature as the **Key information** element.
   f. Select **OK**.
17. Expand the **Part References** section from the Signing Information section and select **Add**. From the next menu:
   a. Input an appropriate Part reference name.
   b. Select the integrity name you specified as the **Integrity part**.
   c. Select **http://www.w3.org/2000/09/xmldsig#sha1** as a Digest method algorithm.
   d. Select **OK**.
18. Expand the **Transforms** section from the Signing Information section and select **Add**. From the next menu:
   a. Input an appropriate Name.
   b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Algorithm.
   c. Select **OK**.

19. Expand the **Encryption Information** section and select **Add**. From the next menu:
    a. Input an appropriate Encryption name.
    b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as a Data encryption method algorithm.
    c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as a Key encryption method algorithm.
    d. Input an appropriate Key information name.
    e. Select the key information name you specified for encryption as the Key information element.
    f. Select the confidentiality name you specified as the **Confidentiality part**.
    g. Select **OK**.
20. Expand the **Security Response Consumer Binding Configuration** section from the WS binding tab.
21. Expand the **Trust Anchor** section and select **Add**. From the next menu:
    a. Input an appropriate Trust anchor name.
    b. Input appropriate Key store storepass and Key store path.
    c. Select **JKS** or **JCEKS** as a Key store type.
    d. Select **OK**.
22. Expand the **Token Consumer** section and select **Add** to create a token consumer used to validate the certificate for digital signature verification. From the next menu:
    a. Input an appropriate Token consumer name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenConsumer** as a Token consumer class.
    c. Check the **Use value type** check box.
    d. Select the **X509 certificate token** as the Value type.
    e. Check the **User certificate path settings** check box.
    f. Select the **Certificate path reference** radio button.
    g. Select the trust anchor name you specified as the **Trust anchor** reference.
    h. Select **OK**.
23. Expand the **Token Consumer** section and select **Add** to create a token consumer used to validate the certificate for decryption. From the next menu:
    a. Input an appropriate Token consumer name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenConsumer** as a Token consumer class.
    c. Check the **Use value type** check box.
    d. Select the **X509 certificate token** as a Value type.
    e. Check the **User certificate path settings** check box.
    f. Select the **Trust any certificate** radio button.
    g. Select **OK**.
24. Expand the **Key locators** section and select **Add** to create a key locator used for digital signature verification. From the next menu:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as the Key locator class.
    c. Check the **Use key store** check box.

   d. Input the appropriate Key store storepass and Key store path.
   e. Select **JKS** or **JCEKS** as a Key store type.
   f. Select **Add** to specify the key used for digital signature verification.
   g. Input an appropriate Alias and Key name to specify the key used for digital signature verification.
   h. Select **OK**.
25. Expand the **Key locators** section and select **Add** to create a key locator used for decryption. From the next menu:
   a. Input an appropriate Key locator name.
   b. Select **com.ibm.pvcws.wss.internal.keyinfo.X509TokenKeyLocator** as a Key locator class.
   c. Check the **Use key store** check box.
   d. Input an appropriate Key store storepass and Key store path.
   e. Select **JKS** or **JCEKS** as a Key store type.
   f. Select **Add** to specify the key used for decryption.
   g. Input an appropriate Alias, Key pass , and Key name to specify the key used for decryption.
   h. Select **OK**.
26. Expand the **Key information** section and select **Add** to create key information used for digital signature verification. From the next menu:
   a. Input an appropriate Key information name.
   b. Select **KEYID** as a Key information type.
   c. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentConsumer** as a Key information class.
   d. Check the **Use key locator** check box.
   e. Select the key locator name you specified for digital signature verification as the **Key locator**.
   f. Check the **Use token** check box.
   g. Select the token consumer name you specified for digital signature verification as the **Token Consumer**.
   h. Select **OK**.
27. Expand the **Key information** section and select **Add** to create a key information used for decryption. From the next menu:
   a. Input an appropriate Key information name.
   b. Select **STRREF** as the Key information type.
   c. Select **com.ibm.pvcws.wss.internal.keyinfo.STRRefContentConsumer** as the Key information class.
   d. Check the **Use key locator** check box.
   e. Select the key locator name you specified for decryption as the **Key locator**.
   f. Check the **Use token** check box.
   g. Select the token consumer name you specified for decryption as the **Token Consumer**.
   h. Select **OK**.
28. Expand the **Signing Information** section and select **Add**. From the next menu:
   a. Input an appropriate Signing information name.

   b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as the Canonicalization method algorithm.
   c. Select **http://www.w3.org/2000/09/xmldsig#rsa-sha1** as the Signature method algorithm.
   d. Select **Add** to configure key information.
   e. Input an appropriate Key information name.
   f. Select the key information name you specified for digital signature verification as the **Key information** element.
   g. Select **OK**.
29. Expand the **Part References** section under Signing Information section and select Add. From the next menu:
   a. Input an appropriate Part reference name.
   b. Select the required integrity name you specified as the **Required Integrity part**.
   c. Select **http://www.w3.org/2000/09/xmldsig#sha1** as the Digest method algorithm.
   d. Select **OK**.
30. Expand the **Transforms** section from the Signing Information section and select Add. From the next menu:
   a. Input an appropriate Name.
   b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as the Algorithm.
   c. Select **OK**.
31. Expand the **Encryption Information** section and select **Add**. From the next menu:
   a. Input an appropriate Encryption name.
   b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as the Data encryption method algorithm.
   c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as the Key encryption method algorithm.
   d. Select **Add** to configure the key information.
   e. Input an appropriate Key information name.
   f. Select the key information name you specified for encryption as the **Key information** element.
   g. Select the required confidentiality name you specified as the **Required Confidentiality part**.
   h. Select **OK**.
32. Save your changes.

To edit a Mobile Web Services security configuration in Scenario 3 for the Web Services provider, perform the following procedure:
1. Open the WS-Security Provider Editor.
2. Expand the **Request Consumer Service Configuration Details** section on the WS extension tab.
3. Expand the **Required Integrity** section and select **Add**. From the next menu:
   a. Input an appropriate Required Integrity Name.
   b. Select **Required** as a Usage type.
   c. Select **Add**.
   d. Select **body** as a Message parts keyword.

   e. Select **OK**.
4. Expand the **Required Confidentiality** section and select **Add**. From the next menu:
   a. Input an appropriate Required Confidentiality Name.
   b. Select **Required** as a Usage type.
   c. Select **Add**.
   d. Select **bodycontent** as a Message parts keyword.
   e. Select **OK**.
5. Expand the **Add Timestamp** section and check the **Use Add Timestamp** check box. From the next menu:
   a. Expand the **Property** section from the Add Timestamp section and select **Add**.
   b. Input `com.ibm.pvcws.wss.timestamp.dialect` as a Property Name and `http://www.ibm.com/wct/webservices/wssecurity/dialect-predefined` as a Property Value.
   c. Expand the **Property** section under the Add Timestamp section and select **Add**.
   d. Input `com.ibm.pvcws.wss.timestamp.keyword` as a Property Name and `SOAPHeaderFirst` as a Property Value.
6. Expand the **Response Generator Service Configuration Details** section from the WS extension tab.
7. Expand the Integrity section and select **Add**. From the next menu:
   a. Input an appropriate Integrity Name.
   b. Select **1** as the Order.
   c. Select **Add**.
   d. Select **body** as the Message parts keyword.
   e. Select **OK**.
8. Expand the **Confidentiality** section and select **Add**. From the next menu:
   a. Input an appropriate Confidentiality Name.
   b. Select **2** as the Order.
   c. Select **Add**.
   d. Select **bodycontent** as a Message parts keyword.
   e. Select **OK**.
9. Expand the **Add Timestamp** section and check the **Use Add Timestamp** check box. From the next menu:
   a. Expand the **Property** section under the Add Timestamp section and select **Add**.
   b. Input `com.ibm.pvcws.wss.timestamp.dialect` as a Property Name and `http://www.ibm.com/wct/webservices/wssecurity/dialect-predefined` as a Property Value.
   c. Expand the **Property** section under the Add Timestamp section and select **Add**.
   d. Input `com.ibm.pvcws.wss.timestamp.keyword` as a Property Name and `SOAPHeaderFirst` as a Property Value.
10. Expand the **Request Consumer Binding Configuration Details** section from the WS binding tab.
11. Expand the **Trust Anchor** section and select **Add**. From the next menu:
   a. Input an appropriate Trust anchor name.

   b. Input an appropriate Key store storepass and Key store path.
   c. Select **JKS** or **JCEKS** as a Key store type.
   d. Select **OK**.
12. Expand the **Token Consumer** section and select **Add** to create a token consumer used to validate the certificate for digital signature verification:
   a. Input an appropriate Token consumer name.
   b. Select **com.ibm.pvcws.wss.internal.token.X509TokenConsumer** as a Token consumer class.
   c. Check the **Use value type** check box.
   d. Select **X509 certificate token** as a Value type.
   e. Check the **User certificate path settings** check box.
   f. Select the **Certificate path reference** radio button.
   g. Select the trust anchor name you specified as the **Trust anchor reference**.
   h. Select **OK**.
13. Expand the Token Consumer section and select **Add** to create a token consumer used to validate the certificate for decryption:
   a. Input an appropriate Token consumer name.
   b. Select **com.ibm.pvcws.wss.internal.token.X509TokenConsumer** as a Token consumer class.
   c. Check the **Use value type** check box.
   d. Select **X509 certificate token** as a Value type.
   e. Check the **User certificate path settings** check box.
   f. Select the **Trust any certificate** radio button.
   g. Select **OK**.
14. Expand the **Key locators** section and select **Add** to create a key locator used for digital signature verification:
   a. Input an appropriate Key locator name.
   b. Select **com.ibm.pvcws.wss.internal.keyinfo.X509TokenKeyLocator** as a Key locator class.
   c. Select **OK**.
15. Expand the **Key locators** section and select **Add** to create a key locator used for decryption:
   a. Input an appropriate Key locator name.
   b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
   c. Check the **Use key store** check box.
   d. Input an appropriate Key store storepass and Key store path.
   e. Select **JKS** or **JCEKS** as a Key store type.
   f. Select **Add** to specify the key used for decryption.
   g. Input an appropriate Alias, Key pass , and Key name to specify the key used for decryption.
   h. Select **OK**.
16. Expand the **Key information** section and select **Add** to create a key information used for digital signature verification:
   a. Input an appropriate Key information name.
   b. Select **STRREF** as a Key information type.

   c. Select **com.ibm.pvcws.wss.internal.keyinfo.STRRefContentConsumer** as a Key information class.
   d. Check the **Use key locator** check box.
   e. Select the key locator name you specified for digital signature verification as the **Key locator**.
   f. Check the **Use token** check box.
   g. Select the token consumer name you specified for digital signature verification as the **Token Consumer**.
   h. Select **OK**.
17. Expand the **Key information** section and click **Add** to create a key information used for encryption:
   a. Input an appropriate Key information name.
   b. Select **KEYID** as a Key information type.
   c. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentConsumer** as a Key information class.
   d. Check the **Use key locator** check box.
   e. Select the key locator name you specified for encryption as the **Key locator**.
   f. Check the **Use token** check box.
   g. Select the token consumer name you specified for encryption as the **Token Consumer**.
   h. Select **OK**.
18. Expand the **Signing Information** section and select **Add**. From the next menu:
   a. Input an appropriate Signing information name.
   b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Canonicalization method algorithm.
   c. Select **http://www.w3.org/2000/09/xmldsig#rsa-sha1** as a Signature method algorithm.
   d. Select **Add** to select a key information.
   e. Input an appropriate Key information name.
   f. Select the key information name you specified as the **Key information** element.
   g. Select **OK**.
19. Expand the **Part References** section under Signing Information section and select **Add**. From the next menu:
   a. Input an appropriate Part reference name.
   b. Select the required integrity name you specified as the **Required Integrity part**.
   c. Select **http://www.w3.org/2000/09/xmldsig#sha1** as a Digest method algorithm.
   d. Select **OK**.
   e. Expand the **Transforms** section under Signing Information section and select Add:
   f. Input an appropriate Name.
   g. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Algorithm.
   h. Select **OK**.
20. Expand the **Encryption Information** section and select Add. From the next menu:

a. Input an appropriate Encryption name.
b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as a Data encryption method algorithm.
c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as a Key encryption method algorithm.
d. Select **Add** to select a key information.
e. Input an appropriate Key information name.
f. Select the key information name you specified as the **Key information** element.
g. Select the required confidentiality name you specified as the **Required Confidentiality** part.
h. Select **OK**.

21. Expand the **Response Generator Binding Configuration Details** section from the WS binding tab.
22. Expand the **Token Generator** section and select **Add** to insert the certificate for decryption. From the next menu:
    a. Input an appropriate Token generator name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenGenerator** as a Token generator class.
    c. Check the **Use value type** check box.
    d. Select **X509 certificate token** as a Value type.
    e. Select **com.ibm.pvcws.wss.internal.auth.callback.X509CallbackHandler** as a Call back handler.
    f. Check the **Use key store** check box.
    g. Input an appropriate Key store storepass and Key store path.
    h. Select **JKS** or **JCEKS** as a Key store type.
    i. Select **Add** to specify the certificate to be inserted into the message.
    j. Input an appropriate Alias and Key name to specify the certificate to be inserted into the message.
    k. Select **OK**.
23. Expand the **Key locators** section and select **Add** to create a key locator used for digital signature:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
    c. Check the **Use key store** check box.
    d. Input an appropriate Key store storepass and Key store path.
    e. Select **JKS** or **JCEKS** as a Key store type.
    f. Select **Add** to specify the key used for digital signature.
    g. Input an appropriate Alias, Key pass , and Key name to specify the key used for digital signature.
    h. Select **OK**.
24. Expand the **Key locators** section and select **Add** to create a key locator used for encryption:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.SignerCertKeyLocator** as a Key locator class.
    c. Select **OK**.

25. Expand the **Key information** section and select **Add** to create a key information used for digital signature. From the next menu:
    a. Input an appropriate Key information name.
    b. Select **KEYID** as a Key information type.
    c. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentGenerator** as a Key information class.
    d. Check the **Use key locator** check box.
    e. Select the key locator name you specified for the digital signature as the **Key locator**.
    f. Select the key name you specified for the digital signature as the **Key name**.
    g. Select **OK**.
26. Expand the Key information section and select **Add** to create a key information used for encryption:
    a. Input an appropriate Key information name.
    b. Select **STRREF** as a Key information type.
    c. Select **com.ibm.pvcws.wss.internal.keyinfo.STRRefContentGenerator** as a Key information class.
    d. Check the **Use key locator** check box.
    e. Select the key locator name you specified for encryption as the Key locator.
    f. Check the **Use token** check box.
    g. Select the token generator name you specified for encryption as the **Token Consumer**.
    h. Select **OK**.
27. Expand the **Signing Information** section and select **Add**:
    a. Input an appropriate Signing information name.
    b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Canonicalization method algorithm.
    c. Select **http://www.w3.org/2000/09/xmldsig#rsa-sha1** as a Signature method algorithm.
    d. Input an appropriate Key information name.
    e. Select the key information name you specified for the digital signature as the **Key information element**.
    f. Select **OK**.
28. Expand the **Part References** section under Signing Information section and select **Add**. From the next menu:
    a. Input an appropriate Part reference name.
    b. Select the integrity name you specified as the **Integrity part**.
    c. Select **http://www.w3.org/2000/09/xmldsig#sha1** as a Digest method algorithm.
    d. Select **OK**.
29. Expand the **Transforms** section under Signing Information section and select Add. From the next menu:
    a. Input an appropriate Name.
    b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Algorithm.
    c. Select **OK**.
30. Expand the **Encryption Information** section and select **Add**. From the next menu:

a. Input an appropriate Encryption name.
b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as a Data encryption method algorithm.
c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as a Key encryption method algorithm.
d. Input an appropriate Key information name.
e. Select the key information name you specified for encryption as the **Key information element**.
f. Select the confidentiality name you specified as the **Confidentiality part**.
g. Select **OK**.

31. Save your changes.

Editing a Mobile Web Services security configuration for encrypt and sign (scenario #6)

To edit a Mobile Web Services security configuration in Scenario #6 for the Web services client, perform the following procedure:

1. Open the WS-Security Client Editor.
2. Expand the **Request Generator Configuration** section on the WS extension tab.
3. Expand the **Integrity** section and select **Add**. From the next menu:
   a. Input an appropriate Integrity Name.
   b. Select **2** as the Order.
   c. Select **Add**.
   d. Select **body** as a Message parts keyword.
   e. Select **OK**.
4. Expand the **Confidentiality** section and select **Add**. From the next menu:
   a. Input an appropriate Confidentiality Name.
   b. Select **1** as the Order.
   c. Select **Add**.
   d. Select **bodycontent** as a Message parts keyword.
   e. Select **OK**.
5. Expand the **Add Timestamp** section and check the **Use Add Timestamp** check box.
6. Expand the **Request Consumer Configuration** section on the WS extension tab.
7. Expand the **Required Integrity** section and select **Add**. From the next menu:
   a. Input an appropriate Required Integrity Name.
   b. Select **Required** as a Usage type.
   c. Select **Add**.
   d. Select **body** as a Message parts keyword.
   e. Select **OK**.
8. Expand the **Required Confidentiality** section and select **Add**. From the next menu:
   a. Input an appropriate Required Confidentiality Name.
   b. Select **Required** as a Usage type.
   c. Select **Add**.
   d. Select **bodycontent** as a Message parts keyword.

   e. Select **OK**.
9. Expand the **Add Timestamp** section and check the **Use Add Timestamp** check box.
10. Expand the **Security Request Generator Binding Configuration** section on the WS binding tab.
11. Expand the **Token Generator** section and select **Add** to insert the certificate for digital signature verification. From the next menu:
    a. Input an appropriate Token generator name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenGenerator** as a Token generator class.
    c. Check the **Use value type** check box.
    d. Select **X509 certificate token** as a Value type.
    e. Select **com.ibm.pvcws.wss.internal.auth.callback.X509CallbackHandler** as a Call back handler.
    f. Check the **Use key store** check box.
    g. Input an appropriate Key store storepass and Key store path.
    h. Select **JKS** or **JCEKS** as a Key store type.
    i. Select **Add** to specify the certificate to be inserted into the message.
    j. Input an appropriate Alias and Key name to specify the certificate to be inserted into the message.
    k. Select **OK**.
12. Expand the **Key locators** section and select **Add** to create a key locator used for digital signature:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
    c. Check the **Use key store** check box.
    d. Input an appropriate Key store storepass and Key store path.
    e. Select **JKS** or **JCEKS** as a Key store type.
    f. Select **Add** to specify the key used for digital signature.
    g. Input appropriate Alias, Key pass , and Key name to specify the key used for digital signature.
    h. Select **OK**.
13. Expand the **Key locators** section and select **Add** again to create a key locator used for encryption:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
    c. Check the **Use key store** check box.
    d. Input an appropriate Key store storepass and Key store path.
    e. Select **JKS** or **JCEKS** as a Key store type.
    f. Select **Add** to specify the key used for encryption.
    g. Input an appropriate Alias and Key name to specify the key used for encryption.
    h. Select **OK**.
14. Expand the **Key information** section and select **Add** to create a key information used for digital signature:

a. Input an appropriate Key information name.
b. Select **STRREF** as a Key information type.
c. Select **com.ibm.pvcws.wss.internal.keyinfo.STRRefContentGenerator** as a Key information class.
d. Check the **Use key locator** check box.
e. Select the required confidentiality name you specified as the **Key locator**.
f. Select the key name you specified for the digital signature as the **Key name**.
g. Check the **Use token** check box.
h. Select the required integrity name you specified as the **Token Generator**.
i. Select **OK**.

15. Expand the **Key information** section and select **Add** to create a key information used for encryption:
    a. Input an appropriate Key information name.
    b. Select **KEYID** as a Key information type.
    c. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentGenerator** as a Key information class.
    d. Check the **Use key locator** check box.
    e. Select the key locator name you specified for encryption as the **Key locator**.
    f. Select the key name you specified for encryption as the **Key name**.
    g. Select **OK**.
16. Expand the **Signing Information** section and select **Add**. From the next menu:
    a. Input an appropriate Signing information name.
    b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Canonicalization method algorithm.
    c. Select **http://www.w3.org/2000/09/xmldsig#rsa-sha1** as a Signature method algorithm.
    d.
    e. Input an appropriate Key information name.
    f. Select the key information name you specified as the **Key information** element.
    g. Select **OK**.
17. Expand the **Part References** section under the Signing Information section and select **Add**. From the next menu:
    a. Input an appropriate Part reference name.
    b. Select the integrity name you specified as the **Integrity part**.
    c. Select **http://www.w3.org/2000/09/xmldsig#sha1** as a Digest method algorithm.
    d. Select **OK**.
18. Expand the **Transforms** section under the Signing Information section and select **Add**. From the next menu:
    a. Input an appropriate Name.
    b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as an Algorithm.
    c. Select **OK**.
19. Expand the **Encryption Information** section and select **Add**. From the next menu:

a. Input an appropriate Encryption name.
b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as a Data encryption method algorithm.
c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as a Key encryption method algorithm.
d. Input an appropriate Key information name.
e. Select the key information name you specified for encryption as the **Key information** element.
f. Select the confidentiality name you specified as the **Confidentiality part**.
g. Select **OK**.

20. Expand the **Security Response Consumer Binding Configuration** section on the WS binding tab.
21. Expand the **Trust Anchor** section and select **Add**. From the next menu:
    a. Input an appropriate Trust anchor name.
    b. Input appropriate Key store storepass and Key store path.
    c. Select **JKS** or **JCEKS** as a Key store type.
    d. Select **OK**.
22. Expand the **Token Consumer** section and select **Add** to create a token consumer used to validate the certificate for digital signature verification:
    a. Input an appropriate Token consumer name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenConsumer** as a Token consumer class.
    c. Check the **Use value type** check box.
    d. Select **X509 certificate token** as a Value type.
    e. Check the **User certificate path settings** check box.
    f. Select the **Certificate path reference** radio button.
    g. Select the trust anchor name you specified as the **Trust anchor reference**.
    h. Select **OK**.
23. Expand the **Token Consumer** section and select **Add** to create a token consumer used to validate the certificate for decryption:
    a. Input an appropriate Token consumer name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenConsumer** as a Token consumer class.
    c. Check the **Use value type** check box.
    d. Select **X509 certificate token** as a Value type.
    e. Check the **User certificate path settings** check box.
    f. Select the **Trust any certificate** radio button.
    g. Select **OK**.
24. Expand the **Key locators** section and select **Add** to create a key locator used for digital signature verification:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
    c. Check the **Use key store** check box.
    d. Input an appropriate Key store storepass and Key store path.
    e. Select **JKS** or **JCEKS** as a Key store type.
    f. Select **Add** to specify the key used for digital signature verification.

g. Input an appropriate Alias and Key name to specify the key used for digital signature verification.

h. Select **OK**.

25. Expand the Key locators section and select **Add** to create a key locator used for decryption:

   a. Input an appropriate Key locator name.

   b. Select **com.ibm.pvcws.wss.internal.keyinfo.X509TokenKeyLocator** as a Key locator class.

   c. Check the **Use key store** check box.

   d. Input an appropriate Key store storepass and Key store path.

   e. Select **JKS**or **JCEKS** as a Key store type.

   f. Select **Add** to specify the key used for decryption.

   g. Input an appropriate Alias, Key pass , and Key name to specify the key used for decryption.

   h. Select **OK**.

26. Expand the **Key information** section and Select **Add** to create a key information used for digital signature verification:

   a. Input an appropriate Key information name.

   b. Select **KEYID** as a Key information type.

   c. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentConsumer** as a Key information class.

   d. Check the **Use key locator** check box.

   e. Select the key locator name you specified for digital signature verification as the **Key locator**.

   f. Check the **Use token** check box.

   g. Select the token consumer name you specified as the **Token Consumer**.

   h. Select **OK**.

27. Expand the **Key information** section and select **Add** to create a key information used for decryption:

   a. Input an appropriate Key information name.

   b. Select **STRREF** as a Key information type.

   c. Select **com.ibm.pvcws.wss.internal.keyinfo.STRRefContentConsumer** as a Key information class.

   d. Check the **Use key locator** check box.

   e. Select the key locator name you specified for decryption as the **Key locator**.

   f. Check the **Use token** check box.

   g. Select the token consumer name you specified for decryption as the **Token Consumer**.

   h. Select **OK**.

28. Expand the **Signing Information** section and select **Add**. From the next menu:

   a. Input an appropriate Signing information name.

   b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Canonicalization method algorithm.

   c. Select **http://www.w3.org/2000/09/xmldsig#rsa-sha1** as a Signature method algorithm.

   d. Select **Add** to select a key information.

   e. Input an appropriate Key information name.
   f. Select the key information name you specified for digital signature verification as the **Key information** element.
   g. Select **OK**.
29. Expand the **Part References** section from the Signing Information section and select **Add**. From the next menu:
   a. Input an appropriate Part reference name.
   b. Select the required integrity name you specified as the **Required Integrity** part.
   c. Select **http://www.w3.org/2000/09/xmldsig#sha1** as a Digest method algorithm.
   d. Select **OK**.
30. Expand the **Transforms** section under the Signing Information section and select **Add**. From the next menu:
   a. Input an appropriate Name.
   b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Algorithm.
   c. Select **OK**.
31. Expand the **Encryption Information** section and select **Add**. From the next menu:
   a. Input an appropriate Encryption name.
   b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as a Data encryption method algorithm.
   c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as a Key encryption method algorithm.
   d. Select **Add** to select a key information.
   e. Input an appropriate Key information name.
   f. Select the key information name you specified for decryption as the **Key information** element.
   g. Select the required confidentiality name you specified as the **Required Confidentiality** part.
   h. Select **OK**.
32. Save your changes.

To edit a Mobile Web Services security configuration in Scenario #6 for the Web services provider, perform the following procedure:

1. Open the WS-Security Provider Editor.
2. Expand the **Request Consumer Service Configuration Details** section on the WS extension tab.
3. Expand the **Required Integrity** section and select **Add**. From the next menu:
   a. Input an appropriate Required Integrity Name.
   b. Select **Required** as the Usage type.
   c. Select **Add**.
   d. Select **body** as a Message parts keyword.
   e. Select **OK**.
4. Expand the **Required Confidentiality** section and select **Add**. From the next menu:
   a. Input an appropriate Required Confidentiality Name.
   b. Select **Required** as a Usage type.

   c. Select **Add**.
   d. Select **bodycontent** as a Message parts keyword.
   e. Select **OK**.
5. Expand the **Add Timestamp** section and check the **Use Add Timestamp** check box.
6. Expand the **Response Generator Service Configuration Details** section on the WS extension tab.
7. Expand the **Integrity** section and select **Add**. From the next menu:
   a. Input an appropriate Integrity Name.
   b. Select **2** as the Order.
   c. Select **Add**.
   d. Select **body** as a Message parts keyword.
   e. Select **OK**
8. Expand the **Confidentiality** section and select **Add**. From the next menu:
   a. Input an appropriate Confidentiality Name.
   b. Select **1** as the Order.
   c. Select **Add**.
   d. Select **bodycontent** as a Message parts keyword.
   e. Select **OK**.
9. Expand the **Add Timestamp** section and check the **Use Add Timestamp** check box.
10. Expand the **Request Consumer Binding Configuration Details** on the WS binding tab.
11. Expand the **Trust Anchor** section and select **Add**. From the next menu:
    a. Input an appropriate Trust anchor name.
    b. Input an appropriate Key store storepass and Key store path.
    c. Select **JKS** or **JCEKS** as a Key store type.
    d. Select **OK**.
12. Expand the **Token Consumer** section and select **Add** to create a token consumer used to validate the certificate for digital signature verification:
    a. Input an appropriate Token consumer name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenConsumer** as a Token consumer class.
    c. Check the **Use value type** check box.
    d. Select **X509 certificate token** as a Value type.
    e. Check the **User certificate path settings** check box.
    f. Select the **Certificate path reference** radio button.
    g. Select the trust anchor name you specified as the **Trust anchor** reference.
    h. Select **OK**.
13. Expand the **Token Consumer** section and select **Add** to create a token consumer used to validate the certificate for decryption:
    a. Input an appropriate Token consumer name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenConsumer** as a Token consumer class.
    c. Check the **Use value type** check box.
    d. Select **X509 certificate token** as a Value type.

e. Check the **User certificate path settings** check box.
f. Select the **Trust any certificate** radio button.
g. Select **OK**.

14. Expand the **Key locators** section and select **Add** to create a key locator used for digital signature verification:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.X509TokenKeyLocator** as a Key locator class.
    c. Select **OK**.
15. Expand the **Key locators** section and select **Add** to create a key locator used for decryption:
    a. Input an appropriate Key locator name.
    b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
    c. Check the **Use key store** check box.
    d. Input an appropriate Key store storepass and Key store path.
    e. Select **JKS** or **JCEKS** as a Key store type.
    f. Select **Add** to specify the key used for decryption.
    g. Input an appropriate Alias, Key pass, and Key name to specify the key used for decryption.
    h. Select **OK**.
16. Expand the **Key information** section and select **Add** to create key information used for digital signature verification:
    a. Input an appropriate Key information name.
    b. Select **STRREF** as a Key information type.
    c. Select **com.ibm.pvcws.wss.internal.keyinfo.STRRefContentConsumer** as a Key information class.
    d. Check the **Use key locator** check box.
    e. Select the key locator name you specified for the digital signature verification as the **Key locator**.
    f. Check the **Use token** check box.
    g. Select the token consumer name you specified for the digital signature verification as the **Token Consumer**.
    h. Select **OK**.
17. Expand the **Key information** section and select **Add** to create key information used for decryption:
    a. Input an appropriate Key information name.
    b. Select **KEYID** as a Key information type.
    c. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentConsumer** as a Key information class.
    d. Check the **Use key locator** check box.
    e. Select the key locator name you specified for decryption as the **Key locator**.
    f. Check the **Use token** check box.
    g. Select the token consumer name you specified for decryption as the **Token Consumer**.
    h. Select **OK**.

18. Expand the **Signing Information** section and select Add. From the next menu:
    a. Input an appropriate Signing information name.
    b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Canonicalization method algorithm.
    c. Select **http://www.w3.org/2000/09/xmldsig#rsa-sha1** as a Signature method algorithm.
    d. Select **Add** to choose the key information.
    e. Input an appropriate Key information name.
    f. Select the key information name you specified for digital signature verification as the **Key information** element.
    g. Select **OK**.
19. Expand the **Part References** section from the Signing Information section and select **Add**. From the next menu:
    a. Input an appropriate Part reference name.
    b. Select the required integrity name you specified as the **Required Integrity part**.
    c. Select **http://www.w3.org/2000/09/xmldsig#sha1** as a Digest method algorithm.
    d. Select **OK**.
20. Expand the **Transforms** section from the **Signing Information** section and select **Add**. From the next menu:
    a. Input an appropriate Name.
    b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as an Algorithm.
    c. Select **OK**.
21. Expand the **Encryption Information** section and select **Add**. From the next menu:
    a. Input an appropriate Encryption name.
    b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as a Data encryption method algorithm.
    c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as a Key encryption method algorithm.
    d. Select **Add** to choose the key information.
    e. Input an appropriate Key information name.
    f. Select the key information name you specified for decryption as the **Key information** element.
    g. Select the required confidentiality name you specified as the **Required Confidentiality part**.
    h. Select **OK**.
22. Expand the **Response Generator Binding Configuration Details** section on the WS binding tab.
23. Expand the **Token Generator** section and select **Add** to insert the certificate for decryption. From the next menu:
    a. Input an appropriate Token generator name.
    b. Select **com.ibm.pvcws.wss.internal.token.X509TokenGenerator** as a Token generator class.
    c. Check the **Use value type** check box.
    d. Select **X509 certificate token** as a Value type.

   e. Select **com.ibm.pvcws.wss.internal.auth.callback.X509CallbackHandler** as a Call back handler.
   f. Check the **Use key store** check box.
   g. Input an appropriate Key store storepass and Key store path.
   h. Select **JKS** or **JCEKS** as a Key store type.
   i. Select **Add** to specify the certificate to be inserted into the message.
   j. Input an appropriate Alias and Key name to specify the certificate to be inserted into the message.
   k. Select **OK**.
24. Expand the **Key locators** section and select **Add** to create a key locator used for digital signature:
   a. Input an appropriate Key locator name.
   b. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyStoreKeyLocator** as a Key locator class.
   c. Check the **Use key store** check box.
   d. Input an appropriate Key store storepass and Key store path.
   e. Select **JKS** or **JCEKS** as a Key store type.
   f. Select **Add** to specify the key used for digital signature.
   g. Input an appropriate Alias, Key pass , and Key name to specify the key used for digital signature.
   h. Select **OK**.
25. Expand the Key locators section and select **Add** to create a key locator used for encryption:
   a. Input an appropriate Key locator name.
   b. Select **com.ibm.pvcws.wss.internal.keyinfo.SignerCertKeyLocator** as a Key locator class.
   c. Select **OK**.
26. Expand the **Key information** section and select **Add** to create a key information used for digital signature:
   a. Input an appropriate Key information name.
   b. Select **KEYID** as a Key information type.
   c. Select **com.ibm.pvcws.wss.internal.keyinfo.KeyIdContentGenerator** as a Key information class.
   d. Check the **Use key locator** check box.
   e. Select the key locator name you specified for the digital signature as the **Key locator**.
   f. Select the key name you specified for the digital signature as the **Key name**.
   g. Select **OK**.
27. Expand Key information section and) click Add again to create a key information used for encryption:
   a. Input an appropriate Key information name.
   b. Select 'STRREF' as a Key information type.
   c. Select 'com.ibm.pvcws.wss.internal.keyinfo.STRRefContentGenerator' as a Key information class.
   d. Check Use key locator check box.
   e. Select the key locator name you specified for encryption as the **Key locator**.

f. Check Use token check box.
g. Select the token generator name you specified for encryption as the **Token Generator**.
h. Click OK.

28. Expand the **Signing Information** section and select **Add**. From the next menu:
    a. Input an appropriate Signing information name.
    b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Canonicalization method algorithm.
    c. Select **http://www.w3.org/2000/09/xmldsig#rsa-sha1** as a Signature method algorithm.
    d. Input an appropriate Key information name.
    e. Select the key information name you specified for the digital signature as the **Key information** element.
    f. Select **OK**.
29. Expand the **Part References** section from the Signing Information section and select **Add**. From the next menu:
    a. Input an appropriate Part reference name.
    b. Select the integrity name you specified as the **Integrity part**.
    c. Select **http://www.w3.org/2000/09/xmldsig#sha1** as a Digest method algorithm.
    d. Select **OK**.
30. Expand the **Transforms** section from the Signing Information section and select **Add**. From the next menu:
    a. Input an appropriate Name.
    b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as a Algorithm.
    c. Select **OK**.
31. Expand the **Encryption Information** section and select **Add**. From the next menu:
    a. Input an appropriate Encryption name.
    b. Select **http://www.w3.org/2001/04/xmlenc#tripledes-cbc** as a Data encryption method algorithm.
    c. Select **http://www.w3.org/2001/04/xmlenc#rsa-1_5** as a Key encryption method algorithm.
    d. Input an appropriate Key information name.
    e. Select the key information name you specified for encryption as the **Key information** element.
    f. Select the confidentiality name you specified as the **Confidentiality part**.
    g. Select **OK**.
32. Save your changes.

Deploying Mobile Web Services

Web services applications are deployed as OSGi bundles / Eclipse plug-ins. They run in the IBM WebSphere Everyplace Deployment runtime platform and require no extra special handling.

Note: If you intend to deploy a Web service client and a Web service provider in the same runtime, place the Web service client stub in a package that is different from the package of the Web service provider to prevent a runtime conflict. Please refer to "Creating Mobile Web Services clients" on page 98 for information on how to provide a different package name.

Note: If your Web service provider or client needs to handle XML control characters, the data must not be encoded as a string or a runtime exception will occur. To allow your Web service provider or client to handle XML control characters, you must use a different encoding such as `byte[]`.

Deploying Mobile Web Services providers

In order to deploy a Web Services provider, launch an instance of the WebSphere Everyplace Deployment runtime with the Web Services provider plug-in installed.

Following are the steps to deploy a Web services provider from the IDE:

1. From the IDE, run the Web services provider plug-in:
   a. Select **Run > Run**.
   b. Create a new WebSphere Everyplace Deployment instance if necessary.
   c. In the Plug-ins tab, ensure that your Web services provider plug-in is checked.
   d. Select **Run**.
   e. At the `osgi>` prompt in the Console view, type `ss` to display a list of all registered bundles and their associated IDs. From the list, find the bundle ID for your Web services provider plug-in.
   f. At the `osgi>` prompt in the Console view, type `start <bundle ID>`, where `<bundle ID>` is the bundle ID for your Web services provider plug-in. This results in a call to the `exposeService()` method of the Web services provider, and its WSDL will be immediately available for client use.
2. Verify that the Web services provider has started successfully:

   From a browser, enter the following URL:

   ```
   http://<machine>:<port>/ws/pid/<servicepid>?wsdl
   ```

   where `<machine>` is the name of the machine hosting the Web services provider, `<port>` is the port used by the Web container (see Note below), and `<servicepid>` is the name used to register the Web services provider (by default it is the base name of the class that implements the exposed Java interface).

   For example, if the machine hosting the Web service is the local host, the Web container is listening on port 1477, and the Java interface used to expose the Web service provider is `MyWebServiceImp`, the WSDL URL will be:

   ```
   http://localhost:1477/ws/pid/MyWebServiceImpl?wsdl
   ```

   Note: Since by default the Web container port is dynamically selected by the IBM WebSphere Everyplace Deployment runtime, refer to the **Web Container Configuration** section in the **WebSphere Everyplace Deployment for Windows and Linux System Administrator's Guide** in order to find the chosen port or to bind the Web container to a static port instead. If the service is deployed in the same runtime environment, and a dynamic port is preferred, you can refer to the section Static Mobile Web Services clients to learn how to write a Web services client that can programmatically retrieve the chosen port.

There are two properties you may change in the OSGi service:

- `com.ibm.pvcws.wsdl`: The value for this property is either an empty String, or a String containing the actual WSDL (not a URL to the WSDL, as above).

When set as an empty String, it indicates that the service should be exposed as a Web Service, and the WSDL should be generated dynamically. You can also pass a String containing the WSDL that describes the Web Service if the WSDL must be of a form that cannot be auto generated.

If you do not use the empty String, the WSDL must have a location attribute. The value of the location attribute is unimportant since the Gateway will correct the location when it is served to clients.

- `org.osgi.framework.Constants.SERVICE_PID`: This is an optional String that can be used to change the service PID URL part of the Web service WSDL. If changed, make sure to pass the same string to the `exportPid` invocation within the `exposeService` method.

Deploying Mobile Web Services clients

In order to deploy a Web Services client, you must ensure that the client implementation has been completed prior to launching an instance of the IBM WebSphere Everyplace Deployment runtime with the Web Services client plug-in installed.

If you generated Web Services client code that is static you must instantiate the generated `Soap_Stub` object and then call the methods you wish to exercise. Please refer to the section "Static Mobile Web Services clients" on page 99 for information on how to complete a static Web Services client.

If you generated Web Services client code that is dynamic or that requires custom marshalling, you will need to access the Web Service using the Gateway plug-in. Please refer to the sections "Dynamic Mobile Web Services clients" on page 102 and "Custom serialization (marshalling)" on page 103 for information on how to complete a dynamic Web Services client.

Once the client code is complete, launch an instance of the IBM WebSphere Everyplace Deployment runtime with the Web Services client plug-in installed and ensure that the appropriate code gets executed to invoke the Web Services provider.

Note: If you get an exception with the message "Parsing of the specified WSDL failed" followed by an explanation, ensure that the WSDL is accessible through a browser. This exception could either be the result of a firewall message in HTML-form requiring authentication, or could be due to an invalid WSDL. Please consult with your administrator.

Note: When deploying a Mobile Web service client for communicating with a Web service provider that references byte arrays, you may need to differentiate between WSDLs that map byte arrays to either `byte unbounded` or to `base64Binary`. You can set the `javax.xml.rpc.Stub.BYTE_ARRAY_ENCODING` property to `byteUnbounded` or `base64Binary` depending on your needs. If the property is not set, the client will encode byte arrays using `base64Binary`.

For example:

```
// static client
stub._setProperty(javax.xml.rpc.Stub.BYTE_ARRAY_ENCODING,
    javax.xml.rpc.Stub.BYTE_UNBOUNDED);

// dynamic client
Dictionary props = new Hashtable();
props.put(javax.xml.rpc.Stub.BYTE_ARRAY_ENCODING,
          javax.xml.rpc.Stub.BYTE_UNBOUNDED);
```

```
WSProxyService wsImpl =
(WSProxyService)bundleContext.getService(serviceReference);

if (wsImpl == null) {
    System.err.println("Unable to find service Web service proxy");
    return;
}

if (!wsImpl.register(wsdl, props)) {
    System.err.println("Unable to consume WSDL");
    return;
}
```

Note: When deploying a Mobile Web service client for communicating with a Web service provider hosted in WebSphere Application Server 5.1 that references `Hashtables` or `HashMaps`, you may need to set the `javax.xml.rpc.Stub.KEY_VAL_FULLY_QUALIFIED` property to "false" (see the previous Note for an example). This will allow the proper SOAP decoding of the mentioned types. By default, this property is not set.

Validating Mobile Web Services using the Web Services Gateway Utility

Accessing the Mobile Web Services Gateway Utility

In order to access the Web Services Gateway Utility, the `com.ibm.pvcws.wsosgi`, `com.ibm.pvcws.osgi.ui`, and standard `HttpService` plug-ins must be installed and started. The `com.ibm.pvcws.osgi.ui` plug-in creates a servlet using the `HttpService` and registers itself as `wsman`.

To install the Mobile Web Services Gateway Utility:

1. Launch the WebSphere Everyplace Deployment.
2. Add the Mobile Web Services Gateway Utility to your workbench:
    a. Select **Application > Install > Add Folder Location**.
    b. Browse to the `updates/eval` directory in your WebSphere Everyplace Deployment distribution media and select **OK**.
    c. At the Edit Local Location, select **OK**.
    d. Select **Next** to list all available features.
    e. Check the **Web Services Gateway Utility** check box, and select **Next**.
    f. Read the License Agreement. If you accept, select **Next**.
    g. Select the installation location to be `<WED_INSTALL_DIRECTORY>/WED/shared/eclipse`, where `<WED_INSTALL_DIRECTORY>` is the directory you chose to install WebSphere Everyplace Deployment.
    h. Select **Finish**.
    i. Select **Install**.

Select **Yes** when asked to restart the workbench.

To access the utility, select **Application > Open > Web Services Gateway Utility** from the WebSphere Everyplace Deployment workbench.

Consuming Mobile Web Services

From the Web Services Gateway Utility, you may consume a Web service and exercise its calls. The WSDL for the Web service must be accessible via a valid URL. The WSDL is parsed by the Web Services runtime component, and it is used to created and install a virtual OSGi bundle that proxies the Web service.

To consume a Web service:

1. Select the **Consume a Web Service** link.

   You must specify the URL to the WSDL of the Web service. This can be an HTTP or a file based URL. You can either paste the WSDL URL into the **Web Service Description (URL)** field or use the **Quick Selection recall** drop-down menu. Every WSDL that is successfully consumed will go into the **Quick Selection** drop-down menu for future access. You can use the **Clear Selected** or **Clear All** to remove items from the quick selection box.
2. Select the **Create** button to retrieve the WSDL, and create and install a virtual OSGi bundle that will serve as a proxy to the Web service.
3. Once the proxy to the Web service is available, you may invoke the Web service calls by selecting the **test** or **dynamic test** links. You may also remove the proxy to the Web service by selecting the remove link.

Exposing Mobile Web Services providers

From the Web Services Gateway Utility, you may expose existing OSGi services as Web Services providers.

To expose an existing OSGi bundle as a Web service:

1. Select the **Create a Mobile Web Services Provider** link.
2. Select the OSGi bundle from the list that you wish to expose as a Web service and select the **create by sid** or **create by pid** link.
3. Once the OSGi bundle is exposed as a Web Services provider, you may consume its WSDL and exercise its calls.

Listing Mobile Web Services client OSGi bundles

From the Web Services Gateway Utility, you may list all dynamic Web Services clients registered with the Web Services runtime component. Simply select the **List Mobile Web Services Client OSGi Bundles** link to generate the list.

For each Web Services client, the following actions can be performed:

remove	Stops and uninstalls the virtual bundle for this Web Service.
test	If a `com.ibm.wsosgi.proxy.test.WSProxyTestService` is registered for this service then this allows you to use that to interactively test the Web Service. If none is registered then this reverts to dynamic test.
dynamic test	This allows you to run a dynamically created test for any of the methods of the Web Service. You will be able to choose which method to run and then fill in all of parameters via HTML forms. You can then execute the method and view the results.

Listing Mobile Web Services provider OSGi bundles

From the Web Services Gateway Utility, you may list all Web Services providers registered with the Web Services runtime component. Simply select the **List Mobile Web Services Provider OSGi Bundles** link to generate the list.

For each Web Services provider listed, the following actions can be performed:

View WSDL	Displays the WSDL for the Web service.
remove	Removes the Web service from the Web services runtime registry.

Creating client runtime images

Getting started building platforms

Use the Platform Builder tool to create a custom runtime image of the WebSphere Everyplace Deployment platform.

To get started, use the Platform Builder Project Wizard to guide you through the process of building an initial target runtime image. Choose which plug-ins, bundles, features, and application components you want so that you can build the smallest runtime footprint. If you want to refine the runtime image at a later date, you can use the Profile Editor to update the `platform.properties` file and rebuild the image.

Follow these basic steps to build a target platform:

Step		**Refer to:**
1	Ensure that your development system meets all software and hardware requirements and that you are aware of any Platform Builder limitations at the time of this release.	*WebSphere Everyplace Deployment Release Notes*
2	Set the plug-in development target platform location to the WebSphere Everyplace Deployment runtime.	“Setting up the target platform”
3	Use the Platform Builder wizard to create a new platform builder project. You can also use this wizard to build the target runtime image.	“Creating a Platform Builder project” on page 148
4	Update the `platform.properties` file, if necessary.	“Updating a Platform Builder project” on page 149
5	Build the target platform.	“Building a target runtime image” on page 149
6	Export the target runtime image to a local directory.	“Exporting a target runtime image” on page 150
7	Run the target runtime image.	“Running a target runtime image” on page 150

Attention: Use of runtime platforms generated with the WebSphere Everyplace Client Toolkit in a production environment requires a separate license. Contact your IBM sales representative for deployment licensing.

Setting up the target platform

To enable the plug-ins provided with WebSphere Everyplace Deployment for use with the Rational Software Development Platform, you must re-target your workspace to use WebSphere Everyplace Deployment runtime platform. Because of this requirement, it is recommended that you create a new workspace to use when you develop plug-ins for this product.

For information about configuring and registering the target platform, see "Using the IBM WebSphere Everyplace Client Toolkit" on page 169.

Creating a Platform Builder project

To create a Platform Builder project, follow these steps:

Note: When using the Platform Builder wizard, press the **F1** key at any time to get help with option descriptions and default values. At any time during the wizard process, you can click **Finish** to accept default values and build a runtime image.

1. Start the IBM Rational Software Development Platform and select **File > New > Project**. The New Project page is displayed.
2. Double-click to expand **Client Services**, select **Platform Builder Project** and click **Next**. The Platform Builder Project Wizard is displayed.
3. Type a name for your project. Accept the default directory for your new project or clear the **Use default** check box and select a workspace of your own. Click **Next** to continue. The "Platform Options page" on page 151 is displayed.
4. Accept default platform options or update fields and click **Next**. Operating System and Processor type fields are not selectable. Click **Next** to continue. The "Plug-ins/Bundles page" on page 155 is displayed.
5. To select the plug-ins and/or bundles that you want to add to the target platform, do the following:
   a. Select one of the following:
      - **Choose features from list (feature based startup)** to view all feature projects in the current workspace (workspace features) and all WebSphere Everyplace Deployment core components (external features).
      - **Choose plug-ins/fragments from list to (plug-in based startup)** to view all workspace and external plug-ins and fragments.

      Note: You cannot switch between radio buttons to select list items from both lists. If you want to add a standalone workspace plug-in that does not belong to a feature, you must first create a feature project in a workspace and add this plug-in to the feature. To do so, click **File > New > Project > Plug-in Development > Feature Project**.

   b. Select the list items that you want to add to the target runtime image.
   c. Click **Add Required Plug-ins** to resolve any dependencies for selected plug-ins. Please wait.
   d. Resolve all dependency messages and click **Next** to continue. The "Startup Options page" on page 156 page is displayed.
6. Configure WebSphere Everyplace Deployment runtime startup options. Click **Next**. The "Languages page" on page 156 is displayed.
7. Select the languages that you want the target platform to support. Click **Next**. The Summary page is displayed.
8. Review the configuration options that you selected. To change any of your selections, click **Back**. Do one of the following:
   - To generate the platform files and build the runtime image, select the **Build Platform** check box and click **Finish**. A runtime image is built, packaged in the format you specified (`zip`, `tar`, or `tar.gz`), and copied into the project directory.

- To generate the initial platform files only, click **Finish**. Platform Builder generates the initial platform files. You can now edit the `property.properties`, where user settings are stored or build the runtime image.

Errors are located in the `.metadata/.log` file relative to the workspace.

Updating a Platform Builder project

When you create a project using the Platform Builder Wizard, a `platform.properties` file is created. This properties file stores the configuration information associated with the Platform Builder project. Use the Profile Editor to edit the key-value pairs stored in the `platform.properties` file. Each page in the editor provides the same controls and behaviors as specified in the Platform Builder wizard.

To edit the `platform.properties` file, follow these steps:

1. Start the Rational Software Development Platform. If you have not already done so, create a Platform Builder project.
2. Locate the project in the Package Explorer view (**Window > Show View > Project Explorer**). If you cannot locate the project in the Package Explorer view, click **File > Refresh**.
3. Expand the project folder and double-click to open the `platform.properties` file. The Platform Options page is displayed.
4. Modify options on one or more of the following pages: "Platform Options page" on page 151, "Plug-ins/Bundles page" on page 155, "Startup Options page" on page 156, and "Languages page" on page 156.
5. Click **File > Save** for changes to take effect.
6. Rebuild the platform, if necessary. To do so, from the Navigator view, right-click the Platform Builder project name and select **Client Services > Build Platform**.

Building a target runtime image

After you create a Platform Builder project and customize it, you must run a build operation to generate the target runtime image.

You can build a target runtime image in the following ways:

- When creating a new project, select the **Build platform** check box on the Platform Builder Wizard Summary page and click **Finish**.
- If a project already exists, do one of the following:
  - Right-click the Platform Builder project name and select **Client Services > Build Platform**. The Build Platform dialog is displayed. Review the output location for the build and edit if necessary. Changes are updated in the `platform.properties` configuration file. Click **Finish** to start the platform build operation.
  - Right-click the `build.xml` file in the Platform Builder project and select **Run > Ant Build**. Modify options (if necessary) and click **Run** to run an Ant build file. When the build is started, Ant uses the `build.xml` file to read the `platform.properties` file and manipulate the necessary files to result in a target platform package.

    Note: When using the Run Ant function for the first time, the `build.xml` file is preset to use the same JRE as the workspace.

After the build completes, the build file named `platform.`*`xxx`* is stored in the output location you specified (where *xxx* specifies the file type extension you specified (`.tar`, `.tar gz`. or `.zip`).

Exporting a target runtime image

After you build a custom runtime image, you can export the file to a local directory on your system and use local tools, such as `pkzip`, `gzip`, or `tar`, to extract the runtime from the archive based on your chosen format.

If your output directory is in the workspace, copy the file to a local directory. The default output location is as follows: *`your_project`*`/output`. To do so, follow these steps:

1. In the Navigator view, select the file that you want to export.
2. From the main menu, select **File > Export**. The Export wizard opens.
3. Select **File System** and click **Next**.
4. By default, the resource that you selected is exported. Optionally, use the check boxes, in the left and right panes to select the set of resources to export and, and use push buttons, such as **Select Types** to filter the types of files that you want to export.
5. Click **Browse** on the next page of the wizard to select the directory where you want to export the file.
6. Click **Finish**.

Running a target runtime image

To launch the target runtime image, do one of the following:

- On Windows systems, run the `startup.bat` file.
- On Linux systems, run the `startup.sh` file.

Note: Do not install a platform into a directory that contains a semicolon in the directory name.

Platform Builder page options

Platform Builder option descriptions are as follows:

Platform Builder page

Table 17. Platform Builder page

Option	**Description**	**Default value**
Project name	Type a name for your new Platform Builder project.	None
Project contents	You can de-select ″Use default″ and click **Browse** to select a directory for your new Platform Builder project.	The ″Use default″ check box is selected. This creates the project in your current workspace.

Platform Options page

Table 18. Platform Options page

Option	**Description**	**Default value**
Platform Profile		
Device	Specifies the type of device on which the target platform runs. Values are as follows: • PC (Windows XP) • PC (Red Hat Enterprise Linux 3.0)	The default value is the operating system on which the Rational Software Development Platform is running.
Operating System	Specifies the operating system installed on the target device. This option cannot be changed (**WindowsXP** or **Linux**).	Based on the **Device** type selected
Processor	Specifies the OSGi-supported processor type. This option cannot be changed (**x86**).	Based on the **Device** type selected
Target Platform	Specifies the target platform runtime—**WebSphere Everyplace Deployment 6.0**.	**WebSphere Everyplace Deployment 6.0**

Table 18. Platform Options page (continued)

Option	Description	Default value
Configuration	**WebSphere Everyplace Deployment (6.0.0) Core**: This WebSphere Everyplace Deployment profile defines services that are available and applicable for a minimal runtime environment consisting of an OSGi framework. **WebSphere Everyplace Deployment (6.0.0) Default**: This IBM WebSphere Everyplace Deployment profile defines services that are available and applicable for a runtime environment consisting of an Eclipse Rich Client Platform framework. The runtime environment defined includes only the minimal set of plug-ins necessary to run the Eclipse Rich Client Platform. **WebSphere Everyplace Deployment (6.0.0) RCP**: This IBM WebSphere Everyplace Deployment profile defines services that are available and applicable for a runtime environment consisting of the IBM WebSphere Everyplace Deployment framework. The runtime environment defined includes the set of plug-ins corresponding to the typical installation of this product.	**WebSphere Everyplace Deployment (6.0.0) Default**

Table 18. Platform Options page (continued)

Option	Description	Default value
JVM	Specifies the type of Java Virtual Machine (JVM) that the target runtime image contains. Values are as follows: • **J2SE** – Specifies the OS-specific J2SE JVM packaged with this product. It is recommended that you use the J2SE default value since the platform has been qualified with this JVM. • **Custom** – Specifies that you want to use a custom JVM. Launches are created to use the JVM specified. • **None** – Specifies *not* to include a JVM. To use this image you must specify the JVM prior to executing the platform for the first time. To do so, follow these steps: 1. Extract your image from the image archive. 2. Edit the `platform.props` file in the root of the extracted image. Replace the value for the **@vm** attribute pair with the `java` or `javaw` executable for the appropriate JVM. 3. Execute the startup script normally to configure the JVM. **Notes**: – This operation must be performed prior to launching startup script to be effective. Subsequent changes to this data are ignored. – The platform startup performs some level of validation against the configured JVM. Errors may result in the platform failing to start. – When no JVM is specified, the existing JVM might not provide the needed functions to run the platform correctly.	**J2SE**

Table 18. Platform Options page (continued)

Option	**Description**	**Default value**
JVM Location	Specifies the location of the JVM that the target runtime image contains.	Based on the **JVM** selected. Click **Browse** to select the location of the custom JVM. This value typically points to a file system location that contains `bin` and `lib` directories.
Build Output		
Output Format	Specifies the file type of the target platform build operation. Available package formats are as follows: • **zip** – Specifies the compressed ZIP format used primarily for Windows devices. • **tar** – Specifies the standard TAR format used on Linux devices. • **tar.gz** – Specifies a zipped version of the TAR format used on Linux devices.	**zip**
Output Location	Specifies the location where Platform Builder stores the target platform after a successful project build.	The default output location is the output directory of the Platform Builder project.

Plug-ins/Bundles page

Table 19. Plug-ins/Bundles page

Option	Description	Default value
Check boxes: • **Choose features from list (feature based startup)** • **Choose plug-in/fragments from list (plug-in based startup)**	Choose feature-based startup to view all feature projects in the current workspace (workspace features) and all WebSphere Everyplace Deployment core components (external features). Choose plug-in-based startup to view all workspace and external plug-ins and fragments. **Note:** You cannot switch between radio buttons to select list items from both lists. If you want to add a standalone workspace plug-in that does not belong to a feature, you must first create a feature project in a workspace and add this plug-in to the feature. To do so, click **File > New > Project > Plug-in Development > Feature Project**.	The plug-ins, bundles, and fragments projects included in the current configuration are selected.
Tree view for check box includes the following entries: • **Workspace Plug-ins** (or **Features**) • **External Plug-ins** (or **Features**)	Lists WebSphere Everyplace Deployment runtime components and user-defined plug-ins and fragments. Select one or more check boxes to specify the items that you want to add to the target platform. Click **Add Required Plug-ins** to add dependent plug-ins, bundles, or fragments for selected entries. The wizard attempts to resolve any discrepancies and prompts you to add or remove selections as needed. **Note:** The **Add Required Plugins** option adds to the selected list any fragments associated with selected bundles, and any dependencies driven by those fragments.	None External Plug-ins are initially configured based on the configuration from the prior panel and the plug-ins are located in the Target Platform. You cannot remove plug-ins/features that are added from the configuration. You can add to the configuration only. Workspace plug-ins are never preconfigured.

Startup Options page

Table 20. Startup Options page

Option	Description	Default value
Program to Run		
Workbench Type	Specify the workbench type for the target platform.	None
Command Setting		
JVM Arguments	*Optional*: Type JVM arguments, if necessary. **Note:** The Platform Builder wizard does not validate these arguments.	None
Program Arguments	*Optional*: Type Program arguments, if necessary. **Note:** The Platform Builder wizard does not validate these arguments.	None
Bootstrap Entries	*Optional*: Type bootstrap entries, if necessary. **Note:** The Platform Builder wizard does not validate these arguments.	None
Profiling	If selected, includes the **–Xprof** option in the JVM Arguments field to enable profiling.	Disabled
Verbose	If selected, includes the **–verbose** option in the JVM Arguments field to enable verbose output.	Disabled
Console	If selected, includes the **–console** option in the Program Arguments field to enable output to the console.	Disabled

Languages page

Select one or more check boxes to specify the languages that you want the target platform to support. You are defaulted to the language of your system, if available. You must select at least one language.

Note: If one language in a fragment is selected, the entire language fragment is added to the target runtime image.

Debugging and testing applications

You can use either the WebSphere Everyplace Deployment Server or the WebSphere Everyplace Client Toolkit's WebSphere Everyplace Deployment Launcher to run and debug applications. Typically, the debugging mechanism you have used in the past will be the easiest to use for debugging plug-ins.

Regardless of the launch mechanism used for debugging, the critical requirements for successful debugging are access to the source for your applications, and Java class files that contain debugging information.

Local debugging and testing

There are two methods available for running and debugging applications in a local instance of the IBM WebSphere Everyplace Deployment runtime. Both are equally capable of running applications, but they have individual capabilities which may make one more preferable.

The WebSphere Everyplace Deployment Launcher is ideal if you are familiar with Eclipse Plug-in development tools. It is also recommended if you are developing bundles that are not J2EE projects, as the WebSphere Everyplace Deployment launcher lets you easily select all bundle projects in your workspace.

The WebSphere Everyplace Deployment Server is ideal if you are familiar with Rational J2EE development tools. It is also recommended if you are developing non-Client Services EJB or Web projects, as the WebSphere Everyplace Deployment Server can easily add them to the list of configured projects to run. Since the server framework coincides with the existing RAD J2EE tools, it may be more natural to use if you are primarily doing J2EE development, especially if you are running the project on both WebSphere Everyplace Deployment and non-WebSphere Everyplace Deployment runtimes.

WebSphere Everyplace Deployment Launcher

The launcher supports the ability to run and debug Client Services projects from your workspace. The IBM WebSphere Everyplace Deployment runtime launch extends the Eclipse runtime workbench launch. It is suggested that you use the WebSphere Everyplace Deployment launcher rather than the Eclipse runtime workbench launcher for running Client Services projects, since it automatically handles setting other parameters for the WebSphere Everyplace Deployment runtime.

Perform the following procedure to run or debug a project using the WebSphere Everyplace Deployment runtime launch:

1. Select either **Run > Run...** or **Run > Debug...** to run or debug using the WebSphere Everyplace Deployment runtime.
2. Select WebSphere Everyplace Deployment under configurations, and select **New** to create a new configuration.

   Note: If WebSphere Everyplace Deployment runtime configurations have already been created, you can directly select one.

3. On the arguments tab, ensure that the JRE selected is **WebSphere Everyplace Deployment v6 JRE** (or the JRE defined in step 2 on page 170 of "Setting up the WebSphere Everyplace Client Toolkit" on page 170).
4. By default, the launcher selects all the external plug-ins/application services from the WebSphere Everyplace Deployment default platform profile. To change the external plug-ins/application services in your WebSphere Everyplace Deployment instance, select the **Profile** tab.
5. By default, the launcher selects all your workspace plug-ins and Client Services projects. To change the project selection, select the **Workspace Plug-ins** section from the Plug-ins tab.
6. Select the appropriate check box to include any fragments with translated properties files. For Group 1 languages, check **Group 1**. For Group 2 languages, check **Group 2**.

   For a list of IBM language groups, refer to "IBM language groups" on page 201.
7. Select either the **Run** or **Debug** button to launch the runtime.

   Note: This launcher works best for Client Services projects. If you have non Client Services EJB or Web projects you can select them to run in Step 5 if you have first set the target runtime of the project.

   Note: If you choose to manually select plug-ins, do not include fragments from Linux in your Windows environment, or fragments from Windows in your Linux environment. If both operating system fragment types are included, the platform will not work correctly. To avoid this problem, use the **Profiles** tab to select your plug-ins.

   Note: If your machine is disconnected from the network (Linux only), then you must manually add the following entry to the `/etc/hosts` file:

   ```
   127.0.0.1 <your_machine_hostname>
   ```

   This allows the Web Container to function correctly when you are disconnected from the network.

For more information on launch option, refer to the section **Running a Plug-in** of the *PDE Guide*.

For more information on debugging, refer to the section **Using the Java integrated development environment > Concepts > Local Debugging** of the *Developing Java Applications Guide*.

WebSphere Everyplace Deployment Server

The server supports the ability to run and debug J2EE projects from your workspace, including Client Services Web projects and Client Services Embedded Transaction as well as existing Rational Web projects and EJB projects. The IBM WebSphere Everyplace Deployment server extends the Rational server framework to launch a WebSphere Everyplace Deployment runtime.

Note: You should be developing in the J2EE or Web perspective when using the Rational server tools to run J2EE applications.

Creating a server

Perform the following procedure to create an IBM WebSphere Everyplace Deployment server:

1. Select **New > Server > Server...** to show the New Server creation dialog

2. Select **IBM > WebSphere Everyplace Deployment v6.0** as the server type and select either **Next** to choose a Platform Profile or **Finish** to use the default.
3. If modifying the profile, perform your selections on this page then select **Finish**.
4. A new server will be created with the default name **WebSphere Everyplace Deployment v6.0 @ localhost** and will be visible from the Servers view.
5. On the arguments tab, ensure that the JRE selected.

Editing a server

You can view the server editor by double-clicking it from the servers view. To modify the Platform Profile, or a set of Application Services, navigate to the **Profile** tab. To modify the advanced features of the launch, select **Advanced** on the main tab. For descriptions of the tabs and fields in this dialog, refer to **Running a Plug-in** of the *PDE Guide*.

Adding projects to a server

Projects that are associated with a server are automatically loaded onto the WebSphere Everyplace Deployment runtime when it is started.

To view the list of J2EE projects associated with a server, or to modify the projects on the server, right-click the server in the Servers view and choose **Add and Remove Projects...** to display the Add and Remove Projects dialog.

If you are working with one J2EE project that you want to test, you can right-click the project and choose **Run on Server**. The wizard adds that project to a new or existing server, and automatically starts the server.

In order to add a non-J2EE Client Services project to a server, you must go to the advanced editing screen (refer to "Editing a server"). You will be able to select Client Services projects from the set of Workspace plug-ins on the Plug-ins tab.

Starting a server

From the Servers view, right click the server and select **Start** to run the server using the IBM WebSphere Everyplace Deployment runtime. Or, select **Debug** to debug using the IBM WebSphere Everyplace Deployment runtime.

Remote debugging and testing

Remote debugging using Rational Software Development Platform

Rational Software Development Platform also provides capabilities to debug code executing within the IBM WebSphere Everyplace Deployment platform. Using Rational Software Development Platform is effective if you have created Client Services projects and will need to debug them remotely.

In order to debug the IBM WebSphere Everyplace Deployment platform, you will need to launch it using specific debug options. You can launch using either a Launch Configuration from the WebSphere Everyplace Deployment launcher in the toolkit, or by using the command line.

To launch a WebSphere Everyplace Deployment platform using a Launch Configuration in the toolkit, create a Launch Configuration as described in "WebSphere Everyplace Deployment Launcher" on page 157. Be sure to use the

Run > Run... option in step 1. Before launching the runtime, enter the following arguments into the VM Arguments field on the Arguments tab:

```
-Xdebug -Xnoagent -Xrunjdwp:transport=dt_socket,server=y,address=zzzz
```

Where `zzzz` is an available port on your system.

You can also launch the WebSphere Everyplace Deployment platform for debugging using the command line. Add the parameters

```
-vmargs -Xdebug -Xnoagent
-Xrunjdwp:transport=dt_socket,server=y,address=zzzz
```

to the command line, where `zzzz` is an available port on your system. Refer to "Starting from the command line" for more information on launching via the command line.

In the Rational Software Development Platform, select **Debug...**, then create a new Remote Java Application launch configuration. Enter the name of any project, and select a Connection Type of **Standard (Socket Attach)**. For the Connection Properties, you can use **localhost** as the host (or supply the IP address or host name for a remote system), and specify the same port number as you specified when launching the WebSphere Everyplace Deployment platform.

Starting from the command line

To start the WebSphere Everyplace Deployment platform from a command line, switch to the `WED_RUNTIME/rcp` directory, and run the `rcplauncher.exe` file on Windows, or the `rcplauncher` file on Linux. You can add additional parameters following the file name, such as

```
rcplauncher -console -nosplash
```

to start the platform with a console window and suppress the splash screen. If you need to supply any Java virtual machine specific arguments, such as when specifying debug options, use the `-vmargs` parameter. The `-vmargs`, followed by any virtual machine specific arguments, must be the last parameter on the command line. Application specific arguments must not follow the `-vmargs` parameter on the command line.

Packaging and deploying applications

This section assumes you have completed some level of code development and are interested in one of the two situations below:

- You have completed code development and are now prepared to start constructing installation artifacts to turn over to an administrator or IT team for distribution to client systems.
- You want to run your code in an actual installed instance of the client platform on your local development system (and not within an instance launched from the workspace)

For information on constructing installation artifacts for distribution, refer to "Packaging applications for distribution."

For information on performing quick deployments of plug-ins to local instances for test purposes, refer to "Deploying projects for local testing" on page 166.

Packaging applications for distribution

Updates to the client platform are provided in the form of features. Features may contain other features, or a set of related plug-ins. The Update Manager component of the client platform handles the installation of the features, and a user interface is provided to manage the installed features.

Features may be provided to the Update Manager by connecting to an update site, or by the included Enterprise Management agent. Update sites organize features for installation. The Update Manager typically connects to an update site to determine the features that are available for installation.

The Enterprise Management agent enables the client platform to be managed remotely. Administrators create software distribution jobs that define the artifacts to be installed, and the Enterprise Management agent handles the installation tasks.

Understanding methods of installation

There are two mechanisms to install applications via features. The application can either be installed from each system by using the Update Manager, or by using an enterprise distribution system.

Local installation

To enable local installation, you will need to provide an update site configuration to the platform. If you provide an installation program, in addition to any other tasks that you perform, you should provide an update site for the users to use to install your application. The update site could be provided on the distribution media, or could be created on the hard drive of the system on which you execute the installation program.

The customers will need to start their client platform, and use the Update Manager to connect to the site, and install the application.

Enterprise installation

In addition to a local installation process, you should also consider providing an installation process to enable enterprise distribution. To enable the enterprise installation, you should clearly identify each file that should be installed, and the appropriate installation location.

The client platform provides an Enterprise Management Agent that connects to Tivoli® Device Manager provided by WebSphere Everyplace Device Manager.

Tivoli Device Manager provides for software distribution as well as configuration jobs to be applied to the client platform. Tivoli Device Manager uses bundles as the distribution artifacts for the Enterprise Management Agent.

While the Enterprise Management Agent is capable of accepting a distribution at a plug-in or bundle level, it is strongly recommended that Eclipse update sites packaged as a bundle using the `NativeAppBundle` tool be used as the distribution artifact.

The update site provided to an administrator for distribution is the same update site that is built for local installation. Therefore, developers can create the same artifacts for enterprise installation as they would create for local installation.

Understanding the types of install artifacts

For any successful application installation, you need to provide the appropriate set of installation artifacts from the following:

Installer/Uninstaller

A program to handle installation (and uninstallation of your application) may be needed if you need to do anything more than provide an update site to allow installation of your application.

Update site

An Update Site is the key mechanism to enable installation of the application. The Rational Software Development Platform provides templates for creating Update Sites.

For more information on update sites, including how to create one, please see the **Getting Started > Update Sites** section of the *PDE Guide*.

Features

A feature is the only level of installable unit that exists. You cannot choose to install only certain plug-ins from a feature. The Plug-in Development Platform provides wizards for creating Features.

Once the feature project is created, you can make additional updates to the feature definition by editing the `feature.xml` file.

Additionally, if a feature has already been created, you can import the feature as a binary project into your workspace. Select **File > Import > External Features** to launch the wizard. Enter the name of an update site to browse, and you can select the features to import.

Importing External Features is useful if you are attempting to create an Update Site, and someone else has already created the features that need to be installed.

To enable WebSphere Everyplace Deployment users to use the Scan for Updates action within the Application Management dialog, you must add an update URL to the `feature.xml` file. Using the Feature Manifest Editor, on the Overview tab, right click on the **Update URLs** entry in the Feature URLs section, then select **New > Update URL**. You can then enter the appropriate update site information in the Properties view that is displayed.

If no update URLs are provided in the `feature.xml` file, the Scan for Updates action will still be displayed as an available action, but will not return any update information.

WebSphere Everyplace Deployment uses the `eclipse` directory to contain all of the features and plug-ins from Eclipse. The `rcp` directory contains the remainder of the features and plug-ins that are part of the client platform. The `shared` directory can be used to install features that are to be used by all users on the client system. Additionally, there is a feature install directory in the workspace associated with each user.

When new versions of features are provided, they will be installed into the same directory as the previous version. The installation directory for a feature upgrade cannot be changed.

Versions for features are specified using `major.minor.service.qualifier`. For example, a version of 4.0.1 has a major version of 4, a minor version of 0, and a service version of 1. An equivalent version is a version that differs first at the service level. A compatible version is a version that differs first at the minor level. For example, using our version 4.0.1 above, a version of 4.0.2 would be an equivalent version, since the service value is the first value that changed. A version of 4.1.2 would be a compatible version, since the minor value is the first value that changed.

The Scan for Updates capability provided as part of the Application Management dialog enables updates of only equivalent or compatible versions, according to the preferences selected in the Manage > Preferences > Install/Update dialog. The default value is for Equivalent feature updates.

A feature version that changes at the major level, for example, a version 5.0.0 that would replace our version 4.0.1, must be installed through the Application > Install mechanism. Scan for Updates will not show this feature as being available.

For additional information on versioning, refer to the **Feature manifest** section of the *Platform Plug-in Developer's Guide*.

Plug-ins

Plug-ins provide the core logic capability for the application, but they must be grouped into features in order to be installed via the Update Manager.

If you choose to use a Feature project, or an Update Site project within your workspace, you will need to provide the plug-ins within the workspace as well. These plug-ins can either be in source form - if you are responsible for developing the plug-ins - or they can be in binary form - if another person will be providing these artifacts to you. If another person is providing the artifacts, then you can import these artifacts as binary plug-ins so that you can use the Feature and Update Site project capabilities. Select **File > Import > External Plug-ins and Fragments** to launch the **Import Wizard**. Once plug-ins exist in projects within the workspace, you can use the Feature Manifest Editor to add plug-ins to the Feature, and the Site Manifest Editor to add features to the Update Site.

Native libraries

Plug-ins may use the Java Native Interface (JNI) to access native library code. Native libraries are by convention placed in a fragment specific to an operating system or architecture to the plug-in providing the Java classes (native libraries can all be placed within a single plug-in). The organization of the fragment is the following:

```
directory\fragment.xml
directory\os\<osgi.os>\<osgi.arch>\*.dll or *.so
```

Where `directory` is typically `<fragment_name>_<fragment_version>`

The value of the `osgi.os` value above is the value corresponding to the value of the Java System property, `osgi.os`. The value of the `osgi.arch` corresponds to the value of the Java System property, `osgi.arch`.

The `osgi.os` value is generally based on the `os.name` property value, although the value of `osgi.os` may alias a set of values for `os.name`. For example, the `osgi.os` value of `win32` is used to represent an `os.name` value of Windows 2000.

The `osgi.arch` value is generally based on the `os.arch` property value.

For the runtime environment, since it is targeting Windows 2000, Windows XP, and Linux, the values of `osgi.os` would be either `win32` or `linux`.

The value for `arch` will be `x86`.

As an example, the DB2 Everyplace component requires native libraries. The plug-in id is `com.ibm.db2e` and the version is 8.2.0. The native libraries required for Windows reside in a fragment for this plug-in, `com.ibm.db2e.win32_8.2.0`. Within this fragment, the directory `os\win32\x86` contains the DLLs required by DB2 Everyplace.

If there is a single native library required by the plug-in, and it is loaded via the `System.loadLibrary()` method, then packaging the fragment or plug-in in this organization is sufficient.

If there are multiple native libraries required by the plug-in, and each one is individually loaded by the `System.loadLibrary()` method, and the DLLs do not depend on each other or statically or dynamically load each other, this organization is sufficient.

If there are multiple native libraries required by the plug-in, and there are dependencies between the libraries such that a `System.loadLibrary()` method call loads one of the libraries, but that library statically or dynamically loads the other libraries, then an additional step is required. Because these directories are not provided on the System `PATH` or `LIBPATH`, the operating system is unable to handle the loading of the shared library. To cause these directories to be added to the system `PATH` or `LIBPATH` so that the operating system can load these libraries, update the `rcpinstall.properties` file to add the appropriate plug-in or fragment directory to the `library.path.prepend` or `library.path.append` properties. Refer to "Configuring native library references" on page 223 for more information.

As an example, there are multiple native libraries required for DB2 Everyplace. Because these libraries have dependencies between them, they must be present on the system `PATH` or `LIBPATH`. Therefore, the fragment directory, `C:/Program Files/IBM/WED/rcp/eclipse/plugins/com.ibm.db2e.win32.x86_8.2.1.20050620/os/win32/x86`, has been added to the `library.path.append` property in the `rcpinstall.properties` file.

Code that behaves differently between operating systems that are aliased to `win32`, such as Windows 2000 or Windows XP, should be handled within the native libraries, and should not be placed into separate plug-ins/fragments, or separate `osgi.os` directories, since changing the operating system name for the runtime environment will prevent proper loading of components such as DB2 Everyplace or SWT.

Configuration file updates

Most of the changes required when installing an application can be accomplished by providing the appropriate plug-ins. However, some of the configuration files used by the WebSphere Everyplace Deployment platform may need to be updated. Changes to these files can be made either by installation programs, or by using an Install Handler associated with the application Feature being installed. An Install Handler is invoked at certain checkpoints within the installation process. During the applicable checkpoints, code provided within the Install Handler can make the required changes to the configuration files.

Installation instructions

You should make sure that any installation instructions for your application are clearly provided to your customers. Your customers will need to know, for example, the location of the update site from which to install the application, any preferences that may need to be updated, how to start your application, and so on.

Enterprise distribution instructions

You should also consider the needs of enterprises to use an enterprise distribution mechanism to install the application. Any artifacts that must be installed should be clearly identified. This will allow the administrator to easily define the necessary steps to distribute your application. You should supply Eclipse update sites to the enterprise administrator to allow for distribution of your application.

Using Ant tasks to build a deployable bundle

ANT is a scripting framework often used in task automation for Java. It is commonly used to automate the building of code from source into the resulting binary artifacts.

The WebSphere Everyplace Client Toolkit provides an ANT task to assist in building workspace projects. Projects containing EJBs and Web Applications require more pre-deployment processing than other types of Client Services or plug-in projects. The `bde.exportJarBundle` ANT task ("bde.exportJarBundle" on page 220) is provided to perform those steps. The output of the ANT task is a JAR file that can be used directly within the Eclipse framework, or placed on an update site for distribution.

The Plug-in Development Environment (PDE) within the Rational Software Development Platform provides a framework for building features and plug-ins using ANT technology. For most plug-in type projects, the automated PDE build technology will dynamically create build scripts based upon the `build.properties` file associated with a project. For projects containing EJBs or Web Applications, the default generated build script is not sufficient. To prevent the PDE build framework from generating an incorrect script, you will need to configure the `build.properties` to indicate that a custom build script is being used:

1. Select the `plugin.xml` or the `MANIFEST.MF` for the project, right click, and then select **PDE Tools > Create ANT Build File**. A `build.xml` file is created in the project.
2. Open the `build.properties` file in the project (using the Build Properties Editor).
3. In the upper right of the Build page, select the **Custom Build** option.
4. Save the `build.properties` file. The `build.properties` will now contain a line `custom = true`.

Once you have created the `build.xml` file, you will need to edit the `build.xml` file to modify the `build.jars` and `gather.bin.parts` tasks to produce the correct output for an ANT build. The `build.jars` task typically compiles the project source, and puts the resulting class files into a JAR within the directory indicated by the ANT script variable `${build.result.folder}`. The `garther.bin.parts` task typically creates a plug-in folder in the directory specified by the `${destinationation.temp.folder}`, and copies all the required files for the binary export into the plug-in folder. For EJB and Web Application projects you want to build for the client platform, you can use the `bde.exportJarBundle` task from within the `build.jars` task, and then unzip the resulting output jar during the `gather.bin.parts` task.

Deploying projects for local testing

In order to run your application plug-ins in a locally installed instance of the client platform, and not an instance started from the workspace application launcher, you will need to export your plug-ins to the local file system.

This section covers how to export your projects to the local file system for local testing. The steps suggested here provide the quickest route for exporting your projects to the local file system. Please note the following:

- Using this method to deploy your plug-ins to the local instance does not allow the plug-ins to be removed through the Application Management menu
- Removing the plug-in requires that you use local system tools to physically remove the directory or directories that contain the plug-ins
- The preferred method for creating binaries for distribution to others is to create features and/or update sites (as covered in "Packaging applications for distribution" on page 161

Exporting Client Services projects

Select **File > Export > OSGI bundle files**. This wizard will allow you to select multiple projects for export. You can select to export bundle (select Export Jar as Export Type) or export directory structure which will be understood by both the WebSphere Everyplace Deployment platform and the Plug-in Development Environment. Specify the output directory, and/or any other options. Select the **Finish** button.

Note: The name of the exported file will be named, by default, `Bundle-SymbolicName_Bundle-Version.jar`. If you prefer to use a different naming mechanism, you can change it. Go to **Windows > Preferences > WebSphere Everyplace Client Toolkit > Export** to specify a different naming mechanism.

Exporting plug-ins from the Rational Software Development Platform

The Rational Software Development Platform provides an Export wizard that will allow you to export a Client Services plug-in project to a local runtime for deployment.

Select **File > Export > Deployable Plug-ins and fragments**. This wizard will allow you to select multiple projects for deployment. You will also be able to select different output formats:

- A single ZIP file containing all plug-ins
- Individual JARs for each plug-in for use on an update site
- A directory structure

The directory structure option is the preferred approach for deployment to a local runtime, as the export format is understood by both the WebSphere Everyplace Deployment platform and the Plug-in Development Environment of the Rational Software Development Platform.

When using the directory structure option, you should refer to an eclipse directory, such as `<installation_root>/rcp/eclipse`. The export process will automatically put your plug-in into the plug-ins directory.

Notes:

1. While your plug-in may build successfully in the workspace, the Export mechanism runs a separate compilation of your plug-in. If you have added entries to the Java build path for your plug-in, you should also make sure that the `build.properties` file in your plug-in project contains any required extra JAR files in the `jars.extra.classpath` property. The `build.properties` file can be updated either through the Build Properties Editor or the Bundle Manifest Editor.
2. If you will be debugging these plug-ins from another IDE, then you should make sure that you check the **Include source code** option in the **Export Wizard**, and that the **Compile source with debug information** option is selected in the Build Options available in the **Export Wizard**.

Using the IBM WebSphere Everyplace Client Toolkit

Getting started with the IBM WebSphere Everyplace Client Toolkit

The information in this section guides you through the process of setting up the IBM WebSphere Everyplace Client Toolkit and creating a sample bundle with IBM WebSphere Everyplace Client Toolkit. After you create a sample bundle, refer to “Roadmap of major tasks” on page 170 for instructions on how to create your own bundle.

IBM WebSphere Everyplace Client Toolkit overview

The IBM WebSphere Everyplace Client Toolkit provides the tools necessary to create and test OSGi bundles, Web Applications, and Embedded Transaction Applications for use on the WebSphere Everyplace Deployment platform. WebSphere Everyplace Deployment is built on Eclipse 3.0, which is built on top of OSGi. For this release, Eclipse 3.0 plug-ins run as OSGi bundles. The Plug-in Developer Environment (PDE) provided with Eclipse 3.0 provides many features that are useful in the development of OSGi bundles. The WebSphere Everyplace Toolkit is built upon the solid base provided by the Eclipse PDE.

WebSphere Everyplace Client Toolkit allows developers to focus on the creation of bundles, without requiring them to become OSGi bundle internals experts. In its simplest form, the toolkit is designed for the developer who wants to develop and store ten or twenty bundles with automated assistance purely within the Eclipse environment.

The primary WebSphere Everyplace Client Toolkit features include:

- **Enhanced Bundle Manifest Editor** - Provides graphical editing of the `MANIFEST.MF` file
- **OSGi validation** - The toolkit provides an OSGi manifest header validation logic based on the OSGi implementation in Eclipse 3.0. This logic maximizes the errors caught at development time, rather than at deployment
- Automatic management of Java Build Path and OSGi Manifest fields
- Run / Debug support for Local and Remote Runtimes

Using the IBM WebSphere Everyplace Client Toolkit, developers build applications and services as ″bundles″ that run on WebSphere Everyplace Deployment runtime. A bundle may be packaged as a JAR file with information in the manifest file that relates information to IBM WebSphere Everyplace Client Toolkit about the bundle, such as the services and packages the bundle imports and/or exports. A bundle may also be packaged in a plug-in structure. For more information, see “Packaging and deploying applications” on page 161 for more information.

Supported platforms

The WebSphere Everyplace Client Toolkit runs on the following platforms and supports application development for the WebSphere Everyplace Deployment runtime on these same platforms.

- Windows XP
- Red Hat Enterprise Linux WS 3.0 with GNOME desktop

The WebSphere Everyplace Client toolkit feature must be installed in Rational Application Developer, Rational Software Architect, or Rational Web Developer 6.0.

Roadmap of major tasks

The following table provides a reference to documentation covering important bundle development tasks and considerations.

Table 21. Bundle Development Documentation

Task	Reference
Create a new Client Services project	"New Client Services Project Wizard" on page 245 "New Client Services Fragment Project Wizard" on page 248
Convert an existing project to a Client Services project	"Convert Project to Client Services Project Wizard" on page 250
Edit the manifest file	"Bundle Manifest editor" on page 244
Update Client Services project properties	"Client Services Project Properties page" on page 252
Exporting Client Services projects	"Deploying projects for local testing" on page 166
Run and debug a Client Services project	"WebSphere Everyplace Deployment Runtime Launch Configuration dialog" on page 253

Setting up the WebSphere Everyplace Client Toolkit

Before you use the IBM WebSphere Everyplace Client Toolkit, verify that you have done the following:

Note: Upon the first startup of a workspace, you will see a dialog that prompts you to automatically set the preferences for the WebSphere Everyplace Client Toolkit. Selecting **yes** in this dialog, causes steps 1 and 2 below to be done automatically. You may also skip them when performing a manual set up.

1. Set the plug-in development target platform location to the WebSphere Everyplace Deployment development runtime.
   a. Select **Window > Preferences > Plug-in Development > Target Platform**.
   b. Set the Location field for the target platform to `RATIONAL_HOME/sdpisv/eclipse/plugins/com.ibm.pvc.wct.runtimes_ 6.0.0/eclipse`.
   c. Select **OK**.
2. Set the default workbench JRE to be the IBM WebSphere Everyplace Client JRE. WebSphere Everyplace Deployment runtime ships with J2SE 1.4.2.
   a. Select **Window > Preferences > Java > Installed JREs**.
   b. Click **Add...**
   c. Set JRE type to **Standard VM**.
   d. Enter a JRE name of your choosing, such as `J2SE 1.4.2`.
   e. Set the JRE home directory to `RATIONAL_HOME/spdisv/plugins/com.ibm.pvc.wct.runtimes.j2se.win32.x86_1.4.2.SR2/rcp/eclipse/plugins/com.ibm.rcp.j2se.win32.x86_1.4.2.SR2/jre`.

f. The JRE system libraries view should now display the JRE libraries.
g. Click **OK**.
h. Click the check box next to the JRE you just created to select it as the default workbench JRE.
i. Click **OK**.

3. Insure that the bundle manifest problem markers are enabled. This will need to be done the first time a preexisting workspace is used with the WebSphere Everyplace Client toolkit.
   a. Click the **Filters** icon on the Problems view toolbar.
   b. In the problem type checklist, ensure that the following types are selected:
      - Bundle Manifest Problem
      - Bundle Manifest Reference Problem

Creating a sample Client Services project

Refer to the following instructions to create a sample Client Services project that includes the Client Services Pizza JSP Web Application Sample project:

1. Select **Window > Open Perspective > Other > Plug-in Development** from the menu bar.
2. Select **Help > Samples Gallery**.
3. Select **Application Samples > WebSphere Everyplace Deployment > Pizza JSP Web Application**.
4. Select **Finish** on the Pizza JSP Web Application Sample project naming dialog.

   The Pizza JSP Web Application Sample project appears in the Package Explorer view.
5. Launch the WebSphere Everyplace Deployment runtime.
   a. Select **Run > Run...**
   b. Select **WebSphere Everyplace Deployment** under Configurations. Click **New**.
   c. Open the **Profile** tab and select the **WebSphere Everyplace Deployment (6.0.0) Default** profile.
   d. Click **Run**.

   Note how you did not have to explicitly install the Pizza JSP Web Application into WebSphere Everyplace Deployment runtime. This is because, as part of standard Eclipse behavior, the WebSphere Everyplace Deployment runtime launches all bundles in the user's workspace (plus any enabled external plug-ins found in the Target Platform folder).This behavior is controlled by the Plug-ins tab of your WebSphere Everyplace Deployment.
6. Now, with the Pizza JSP Web Application already running on the WebSphere Everyplace Deployment runtime, verify its behavior by launching the application in the runtime's browser. Select **Application > Open > Pizza JSP Web Application** from the WebSphere Everyplace Deployment runtime:

Setting Manifest Editor preferences

Validation of the bundle manifest file can be controlled via the WebSphere Everyplace Client Toolkit preferences. You can access this panel by selecting **Window > Preferences > WebSphere Everyplace Client Toolkit**. On this panel you can set the notification level for different possible problems in the manifest editor. You can set the notification level to error, warning, or ignore.

If **error** is selected, the problem will be marked as an error in the problems view and compilation of the bundle will not succeed.

If **warning** is selected, the problem will be marked as a warning in the problems view and compilation will proceed.

If **ignore** is selected, the problem will be ignored and no notification will be displayed.

The following preferences can be set

Table 22. Manifest editor preferences

Option	**Description**	**Default value**
Incorrect manifest syntax	The syntax of the manifest header is incorrect or invalid	Error
Unresolved bundle references	The specified bundle reference does not exist in the PDE target platform or workspace	Error
Invalid reference values	The specified reference is invalid for the given manifest headers	Error
Invalid fragment references	The specified reference is invalid in the fragment manifest file	Warning
Required attributes not defined	A required attribute for the manifest header is missing	Error
Unknown attributes	The specified attribute of the manifest is not defined in the OSGi specification	Warning
Unknown attribute values	The attribute value defined in the manifest is incorrect or invalid	Warning
Secondary dependencies	The secondary dependency is referenced, but is not defined in the bundle's manifest file	Warning
Unused references	The given reference is defined but not used in the bundle	Warning
Unknown resources	The given resource cannot be resolved	Error

Turning on build automatically

For optimum launch performance, the 'build automatically' preference should be enabled. To enable 'build automatically', select **Project** from the main menu bar. If **Build Automatically** is checked, the preference is already enabled. If it is not checked, select the option to enable it.

Concepts

The WebSphere Everyplace Client Toolkit extends the RSDP integrated development environment to support the development, testing, and deployment of Eclipse plug-ins and OSGi™ bundles. The WebSphere Everyplace Client Toolkit

adds wizards and editors that collectively provide an OSGi bundle developer with the tools needed to complete the following tasks:
- Identify bundle package and service imports and exports.
- Construct the OSGi manifest to include package and service imports and exports.
- Launch the WebSphere Everyplace Deployment runtime from the Integrated Development Environment (IDE).

Client Services Project

A Client Services project contains a bundle and an associated Application Profile. In addition, Client Services projects can:
- Automatically update the Java Build Path.

  WebSphere Everyplace Client Toolkit can automatically update the project's Java Build Path to reflect the project's Platform Profile and Application Services settings. Refer to "Platform Profile" on page 174 and "Application Services" on page 174 for more information.
- Provide a default bundle activator.

  The toolkit can create a default bundle activator class. You can tailor the default bundle activator class by editing the source file for the class folder.
- Automatically update the Manifest file.

  The toolkit can automatically update the Manifest file in the Client Services project to contain appropriate OSGi metadata for the project. Client Services manages or provides initial default values for the metadata fields by:
  – Setting `Bundle-Name` to the project name on project creation
  – Setting `Bundle-Version` to `1.0.0` on project creation
  – Setting `Bundle-Activator` to the default bundle activator if one was created
  – Updating `Import-Packages` and `Require-Bundle` to reflect the packages imported by the project's classes. Refer to "Automatic management of manifest package dependencies" on page 175 for more information
  – Setting `Import-Services` to include all of the services from the Application Services selected for the project. Refer to "Platform Profile" on page 174 for more information.

Bundle Manifest Editor

The existing Eclipse plug-in manifest editor packaged with the Eclipse PDE is used to edit the `plugin.xml`, `MANIFEST.MF`, and `build.properties` files. It provides a user interface for entering plug-in data, and specific manifest headers. WebSphere Everyplace Client Toolkit provides a Bundle Manifest Editor, which extends the PDE plug-in manifest editor to include graphical editing of the `MANIFEST.MF` file. Certain fields stored in the `MANIFEST.MF` file are edited (graphically) on tabs of the existing Eclipse plug-in manifest editor. The fields managed include such fields as importing/exporting services and import/export package. The manifest editor provides a Graphical User Interface to modify OSGi Manifest Headers.

Managing Client Services project dependencies

The following describes how to best use the tooling provided by the WebSphere Everyplace Client Toolkit to manage the dependencies in a Client Services project. These dependencies include the Java Build Path and the manifest file. When developing Eclipse plug-ins or OSGi bundles, the Java Build Path, the packages used by the bundle's code, and the manifest are all related.
- The Java Build Path must contain the necessary libraries and bundles (plug-ins) that contribute the packages and classes used by the project's bundle code

during the compilation process. If this is not the case, the tools will tag the code with problem markers indicating that a referenced package or class cannot be found.

- The manifest must contain references to the packages and bundles that the bundle code is using. This is how the OSGi framework manages the class path of the bundle at runtime. A reference to a particular bundle implementation is done through a Require-Bundle manifest entry. A reference to a required package is done through an Import-Package manifest entry. Failure to properly resolve these dependencies in the manifest can cause the bundle to fail at runtime with a "class not found" error (`NoClassDefFoundError`).

The following mechanisms are available to help manage both the Java Build Path and the manifest file:

Platform Profile

The Platform Profile provides a method for you to define the Application Services used by the runtime environment, the build-time environment for a Client Services project, and the set of bundles that can run on the platform.

A Platform Profile defines a set of Application Services. Application Services enable you to focus on the logical service requirements of the service instead of the requirements of the actual underlying bundles.

When you create a Client Services project, you select a Platform Profile and a set of Application Services for the project. The WebSphere Everyplace Client Toolkit updates the Java Build Path for the project to reflect the Java Runtime Environment (JRE) of the platform, based on the Platform Profile you selected. The Application Services you selected are automatically added to the Java Build Path of the project. "Application Profile" refers to the Platform Profile and the set of Application Services you selected for the project.

The WebSphere Everyplace Client Toolkit provides Platform Profiles that represent the bundles available in the WebSphere Everyplace Deployment runtime.

Application Services

An Application Service represents a logical service, such as an XML parser or logging service that consists of one or more bundles. Application Services enable you to focus on the logical requirements of the service instead of the requirements of the actual underlying bundles.

Application Profile

Application Profile refers to the Platform Profile and the set of Application Services that you selected for the project. The Platform Profile and Application Services you select define the runtime and build time environment for the project. WebSphere Everyplace Client Toolkit can optionally update the Java Build Path for the project to reflect the Java Runtime Environment (JRE) of the platform, based on the Platform Profile you selected. WebSphere Everyplace Client Toolkit places the bundles from the Application Services you selected into the Java Build Path of the project.

Secondary Dependencies

Secondary Dependencies allow users to include bundles in the build path without adding references to the Manifest. By using this mechanism to include these bundles in the build path, users can compile and develop their code with the convenient features of Eclipse JDT. Once the user references one of the packages in a bundle in the Secondary Dependency list, WebSphere Everyplace Client Toolkit can display a warning to inform the user the project is compiling against classes which are not available during runtime (For information on how to change the severity of this notification, see “Setting Manifest Editor preferences” on page 171). The WebSphere Everyplace Client Toolkit also provides “quick fixes” to resolve this problem by adding corresponding references to the Manifest. The user may also set the tool to automatically add dependencies to the Manifest, whenever they are found.

The Application Services dependencies selected from a project’s Platform Profile are added to the Secondary Dependencies for dependency management. For more information on how to add Secondary Dependencies manually, please refer to “Bundle Resources page” on page 245.

Note: Plug-in dependencies can be selected through both the Plug-in Manifest editor and the Bundle Manifest editor through the Dependencies page. Selecting a plug-in dependency adds the plug-in to the project’s Java Build Path and updates the manifest with a `Require-Bundle` entry. This is appropriate for code that is extending the Rich Client Platform, but does not properly handle bundles that export their packages using `Export-Package`. If you are not strictly developing against RCP plug-ins, it is suggested that you represent your bundle dependencies using Secondary dependencies.

Automatic management of manifest package dependencies

Manifest package dependencies are automatically managed for Client Services projects by default. This capability can be disabled through the new project wizard when creating a Client Services project and through the Client Services properties page of an existing project. When enabled, this option automatically updates the `Require-Bundle` and `Import-Package` entries in the project’s manifest file based on changes in the project’s Java code. When new package dependencies are added to the project’s Java code, the manifest file is automatically updated with the proper entries. It is important to note that this mechanism requires the developer to use the Platform Profile or secondary dependencies to represent the project’s bundle dependencies. Bundles that are placed on the Java Build Path through the project’s Java properties page will not be handled by this mechanism.

If this option is disabled, the tools will flag the manifest file with error markers if packages are used in the Java code without being referenced in the manifest file. These errors can then be selectively fixed through the quick fix mechanism.

WebSphere Everyplace Client Toolkit will search for manifest package dependencies by default. This capability can be disabled through the new project wizard when creating a Client Services project and through the Client Services properties page of an existing project. If this option is disabled, the user can still manually search the project for manifest dependencies by using the **Compute** button in the Secondary Dependencies panel on the Bundle Resource Page of the Bundle Manifest Editor.

Bundle registration job

The WebSphere Everyplace Client Toolkit provides more advanced prerequisite dependency checking than found in the Eclipse PDE. This dependency checking requires the toolkit to register the bundles, plug-ins and fragments in your target platform.

You will see a registration job window when you reset your target platform. This job is required to finish in order to load the dependency checking with all the required information. You can select **Run in Background** to use the toolkit during this registration process.

Until the database has been loaded completely, the dependency checking, along with the functionality that uses the dependency checking, will not be available. Both the Platform Builder wizard and the WebSphere Everyplace Deployment Launcher utilize the dependency checking and will not be available until all the bundles, plug-ins and fragments have been registered.

Advanced PDE export

The export functionality for plug-ins, features, and update sites can be extended to include Client Services Web projects and Embedded Transaction Container projects.

To enable or disable this extended functionality, perform the following procedure:

1. Go to **Windows > Preferences > WebSphere Everyplace Client Toolkit > Export**.
2. To enable the functionality, check the **Use Advanced PDE Export** check box. To disable it, deselect the check box to use the basic PDE Export.

   If you choose not to utilize the advanced export, you cannot export plug-ins and features, or build update sites that reference Client Services Web projects or Embedded Transaction Container projects.

Creating and using Client Services applications

This section describes the tasks for creating and using Client Services projects to create bundles.

Creating a Client Services project

Complete the following steps to create a new Client Services project:

1. Select **File** >**New** >**Project**.

   The new project wizard is displayed.
2. Select **Client Services > Client Services Project**.
3. Click **Next**.

   The **New Client Services Project** panel is displayed.
4. Specify a project name in the **Project name** field.
5. Click **Next**.

   The second **New Client Services Project** panel is displayed.
6. Modify the bundle properties as necessary.
7. Select whether or not the application is being designed for the Rich Client Platform. By default, the option to contribute to the RCP is pre-selected.
8. Click **Next**.
9. Select a **Platform Profile** from the pull-down menu.

   A description of the platform profile you selected is displayed in the **Description** field.
10. Select the boxes next to the **Application Services** you want to use.

    Note: If the platform that you selected in step 9 requires specific services, the services are automatically selected in the **Application Services** field. You can select any additional services that the application uses.

11. Click **Next**.

    The Client Services Options page is displayed. See the table, *Client Services Project Options* in "New Client Services Project Wizard" on page 245 for additional information.

    Note: In some cases, a Client Services Project cannot automatically determine the correct Manifest dependency settings. For example, if your code uses the `java.lang.Class.forName(...)` API, the tool cannot correctly calculate the required Manifest dependency settings at compile time. In this case you should disable the **Attempt to automatically resolve Manifest dependencies** option and configure the dependency section of the Manifest manually by using the Manifest Editor.
12. Click **Finish**.

    WebSphere Everyplace Client Toolkit creates a Client Services project.

Creating a Client Services fragment project

1. Select **File** >**New** >**Project**.

   The new project wizard is displayed.
2. Select **Client Services** on the left panel and **Client Services Fragment Project** on the right panel.
3. Click **Next**.

   The **New Client Services Fragment Project** panel is displayed.
4. Specify a project name in the **Project name** field.
5. Select the Parent Bundle for the project by selecting the **Browse** button next to the Bundle ID input box in the Parent Bundle Section.
6. If the fragment is intended for the Rich Client Platform, select **Next**. Otherwise deselect **This fragment will contribute to the Rich Client Platform** and select **Next**.
7. Select a **Platform Profile** from the pull-down menu.

   A description of the platform profile you selected is displayed in the **Description** field.
8. Select the boxes next to the **Application Services** you want to use.

   Note: If the platform that you selected above requires specific services, the services are automatically selected in the **Application Services** field. You can select any additional services that the application uses.
9. Click **Next**.

   The Client Services Options page is displayed. See the table, *Client Services Project Options* in "New Client Services Project Wizard" on page 245 for additional information.

   Note: In some cases, a Client Services project cannot automatically determine the correct Manifest dependency settings. For example, if your code uses the `java.lang.Class.forName(...)` API, the tool cannot correctly calculate the required Manifest dependency settings at compile time. In this case you should disable the **Attempt to automatically resolve Manifest dependencies** option and configure the dependency section of the Manifest manually by using the Manifest Editor.
10. Click **Finish**.

    The IBM WebSphere Everyplace Client Toolkit creates a Client Services Fragment project.

Converting a Java project into a Client Services project

Complete the following steps to convert a Java project into a Client Services project:

1. Select **File**->**New**->**Other...**.

   The **New** project wizard is displayed.
2. Select **Client Services > Convert Project to Client Services Project**.
3. Select **Next**.

   The **Convert Project to Client Services** panel is displayed.
4. Select the Java project that you want to convert.
5. Click **Next**.

   The second **Convert Project to Client Services** window is displayed.
6. Select a **Platform Profile** from the pull-down menu.

   A description of the platform profile you selected is displayed in the **Description** field.
7. Select the boxes next to the **Application Services** you want to use.

   Note: If the platform that you selected in step 5 above requires specific services, the services are automatically selected in the **Application Services** field. You can select any additional services that the application uses.
8. Click **Next**.

   The Client Services Options page is displayed. See the table, *Client Services Project Options* in "Convert Project to Client Services Project Wizard" on page 250 for additional information.

   Note: In some cases, a Client Services Project cannot automatically determine the correct Manifest dependency settings. For example, if your code uses the `java.lang.Class.forName(...)` API, the tool cannot correctly calculate the required Manifest dependency settings at compile time. In this case you should disable the **Attempt to automatically resolve Manifest dependencies** option and configure the dependency section of the Manifest manually by using the Manifest Editor.
9. Click **Finish**.

   The IBM WebSphere Everyplace Client Toolkit converts the Java project into a Client Services project.

Setting Client Services project properties

Client Services projects have some project-specific properties. To modify the project properties, complete the following steps:

1. Right-click on your project and select **Properties**.
2. Select **Client Services**.

   The Application Profile tab allows you to choose the Platform Profile that you want to use with this project. You may also select the Application Services that you want to be made available to this bundle.

   The Options tab allows you to control the behavior of the WebSphere Everyplace Client Toolkit for this project.

   See the table, *Client Services Project Options* in "New Client Services Project Wizard" on page 245 for additional information.
3. Make the changes you desire and click OK to commit your changes.

Using Platform Services

Using the HTTP Service

WebSphere Everyplace Deployment provides two methods to enable content serving via an HTTP Server. The Web Container provides the ability to run web applications serving content via servlets and JSPs. For more information on the Web Container, refer to “Developing Web applications” on page 25.

In addition, the WebSphere Everyplace Deployment platform enables you to provide services on the Internet by providing an HTTP Web server. The basic Web Server model is the `HttpService`, which is an implementation of the OSGi specification for `HttpService`. The `HttpService` implements an HTTP 1.0 Web server with a Servlet 2.1 engine. The `HttpService` provides a complete implementation of the Servlet 2.1 specification.

The `com.ibm.osg.service.http` plug-in provides the implementation of the HTTP Service.

The HTTP Service enables other bundles to dynamically register and unregister servlets and other static resources such as GIF files. You can register HTML files, GIF files, class files, or any resources found via a URL.

handleSecurity plug-in

You must provide an HTTP Context when you register a resource or a servlet with the HTTP Service. The HTTP Context defines how resources are accessed and how access to the resources is authenticated. The handleSecurity method of HTTP Context performs the authentication for servlets and resources.

IBM’s HTTP Service implementation supports a pluggable default handleSecurity method. This method defines the behavior of the default HTTP Context handleSecurity method.

Configuring HTTP Service

HTTP Service configurations described here are specific to the IBM implementation of HTTP Service that is included with WebSphere Everyplace Deployment.

The HTTP Service can use either Java system properties for simple configuration, or it can use `ConfigurationAdmin` for more sophisticated configuration. The following Java system properties govern the configuration of the HTTP Service:

Table 23.

`com.ibm.osg.service.http.address`	`HttpService` property, which if set defines the host address for the default ports that the `HttpService` listens on. If this property is defined then the `HttpService` will only listen for requests that come through this IP address. The default value is ALL, which indicates all available IP addresses on the device will be used. The value of this property may be a resolved name or IP address (e.g. www.ibm.com, 192.168.0.101, local host)

Table 23. (continued)

`com.ibm.osg.service.http.defaultports`	If this property is set, and no HTTP Service port configuration exists in Configuration Admin, then HTTP Service will listen on the default port specified by `org.osgi.service.http.port`. If this property is not set and no HTTP Service port configurations exist in Configuration Admin, then HTTP Service will not listen on any ports.
`org.osgi.service.http.port`	Default HTTP port. See `com.ibm.osg.service.http.defaultports`.
`org.osgi.service.http.port.secure`	Default HTTPS port. See `com.ibm.osg.service.http.defaultports`. If not set then no port will be opened to listen for HTTPS requests. The default is not to listen for HTTPS requests.

The IBM implementation of HTTP Service has several configuration parameters. There are general configuration parameters that affect all of HTTP Service and there are additional parameters for configuring individual ports. These additional parameters enable you to configure the HTTP Service to listen on multiple ports.

The HTTP Service manages its configuration parameters with the Configuration Admin Service. The Configuration Admin Service relies on a PID (Persistent Identity) to identify which service to configure. The PID resides in a service property called `service.pid`.

General configuration

These parameters affect all of HTTP Service and are configured with a Managed Service.

The PID for the Managed Service for HTTP Service's general configuration is:
`com.ibm.osg.service.http.Http`

Table 24. HTTP Service General Configuration Parameters

Parameter Name	**Java Type**	**Definition**	**Valid Values**
http.minThreads	`Integer`	Specifies the minimum number of threads in the thread pool.	Valid values are between 0 and 63. The default value is 4.
http.maxThreads	`Integer`	Specifies the maximum number of threads in the thread pool.	Valid values are between 0 and 63. The default value is 20.
http.threadPriority	`Integer`	Specifies the priority of threads in the thread pool.	A valid value is any integer with a valid `Thread` priority. Refer to the Thread class for valid values. The default value is `Thread.NORM_PRIORITY`.

Port configuration

A Managed Service Factory configures these individual ports. A Managed Service Factory allows multiple instances of a configuration for a given service. Each configuration that you make under the Managed Service Factory PID directs HTTP Service to register another HTTP Service object with the configured parameters.

Note: If no individual configuration exists, then HTTP Service will not register an HTTP Service object and therefore, will not listen on any port unless the property, `com.ibm.osg.service.http.defaultports`, is set.

The PID for the Managed Service Factory for the individual port configuration is:

`com.ibm.osg.service.http.HttpFactory`

Table 25. HTTP Service Individual instance parameters

Parameter Name	Java Type	Definition	Valid Values
http.port	`Integer`	Specifies the port to listen to for HTTP requests.	The default value is 80.
http.scheme	`String`	Specifies the scheme to use.	Valid values are `http` and `https`. The default value is `https`. **Note:** To configure a port to use HTTPS, a `java.net.ssl. SSLServerSocketFactory` must be registered as an OSGi service.
http.timeout	`Integer`	Specifies the number of seconds to wait before reclaiming a Keep-Alive thread.	Valid values are between 0 and 600. The default value is 30. Specify 0 to disable Keep-Alive support.
http.address	String	Specifies the host address to listen to for HTTP requests.	A host name or IP address that is a valid address for the device. A value of ″ALL″ indicates all valid addresses on the device should be used. The default value is ″ALL″.
service.ranking	Integer	Specifies the service ranking for the configuration	Valid values are 0 to`integer.MAX_VALUE`

Configuring the parameters using Admin Utility for OSGi

Bundles configure the parameters with the Configuration Admin API. You can also configure the parameters with the Admin Utility for OSGi User Interface. Refer to “Using Admin Utility for OSGi” on page 192 for more information on installing and launching the admin utility. To configure HTTP Service using the admin utility, under **Bundles** find and click the bundle file with the name **HTTP Service**.

On the HTTP Service page you will see HTTP Service General Settings and HTTP Service Settings that you can configure.

Configuring the HTTPService for multiple ports

You can configure the HTTPService for multiple ports. A different HttpService registration represents each configured port. An application that uses the HTTPService must register with each port to be accessible on that port.

For example, you might only want to register a Servlet with sensitive data with the HttpService which supports HTTPS on port 443. You might want to register a Servlet that does not contain sensitive data with both the 80 and the 443 port HttpServices.

To ensure that you register a servlet with the correct HttpService instance, use a **ServiceTracker** with a Filter. A Filter enables your bundle to specify exactly which services you want to know about to the ServiceTracker. Your BundleActivator can extend the **ServiceTracker** class or implement the **ServiceTrackerCustomizer** interface to provide the addingService(), modifiedService(), and removingService() methods. ServiceTracker will call these methods to notify your bundle when these service registration events occur. For example, if your bundle is tracking HttpService registrations, it would be notified when HttpServices are registered and unregistered and your bundle can then use these services accordingly.

The following example shows how to create the Filter and ServiceTracker:

```
//To track any HttpService registrations using port 80.  We would look for the property
 "http.port=80".
public void start(BundleContext context)
{
        Filter filter = context.createFilter("(&(objectClass=org.osgi.service.http.HttpService)
        (http.port=80))");
        ServiceTracker httpServiceTracker = new ServiceTracker(context,filter,this);
}
//To track any HttpService registrations using HTTPS.  We would look for the property
"http.scheme=https".
public void start(BundleContext context)
{
        Filter filter = context.createFilter("(&(objectClass=org.osgi.service.http.HttpService)
(http.scheme=https))");
        ServiceTracker httpServiceTracker = new ServiceTracker(context,filter,this);
}
```

For an example on how to use a ServiceTracker, refer to the **Samples Gallery > Technology Samples > WebSphere Everyplace Deployment > OSGi > Service Tracker**.

For additional documentation, please see the OSGi™ Service Platform release 3 specification.

Using the Log Service and Log Reader Services

IBM WebSphere Everyplace Deployment provides an implementation of the OSGi Log Service specification in the `com.ibm.osg.service.log` plug-in. In addition to the Log Service implementation, this bundle contains an implementation of the OSGi Log Reader service. The Log Reader service enables applications to register a `LogListener` and receive notifications of all `LogEvents`.

The WebSphere Everyplace Deployment platform, through its log redirection capabilities, registers a `LogListener` to receive notification of the `LogEvents`, and records the events using the JDK logger into platform log files.

Configuring Log Service

Log Service configurations described here are specific to the IBM implementation of Log Service. Log Service has a number of configuration options. You can control the log size and the log threshold (Error, Warning, Informational, or Debug severity level). When you specify a threshold level, only entries that are less than or equal to the current threshold are logged. `Error` is the lowest threshold level and `Debug` is the highest level.

The Log Service manages its configuration parameters with the Configuration Admin Service.

The PID for the Managed Service for Log Service's configuration parameters is: `com.ibm.osg.service.log.Log`

Table 26. Log Service Configuration Parameters

Parameter Name	Java Type	Definition	Valid Values
log.size	`Integer`	Specifies the maximum number of entries in the log. Old entries are discarded after the log reaches this number.	Valid values are between 10 and 2000. The default value is 100.
log.threshold	`Integer`	Specifies the lowest level that you want to retain entries in the log. Any level higher than what is specified will not be kept in the log.	Valid values are: LogService.LOG_ERROR LogService.LOG_WARNING LogService.LOG_INFO LogService.LOG_DEBUG

Configuring the parameters using Admin Utility for OSGi

Bundles configure the parameters with the Configuration Admin API. You can also configure the parameters with the Admin Utility for OSGi User Interface. Refer to the section on Using the Admin Utility for OSGi for more information on installing and launching the admin utility. To configure Log Service using Admin Utility for OSGi, under **Bundles** find and click the bundle file with the name **Log Service**.

On the LogService page you will see Log Service Settings. You can configure the log size and level of logging.

Using the Meta Type Service

The OSGi Configuration Admin Service allows bundles to persistently store their configurations. The OSGi Meta Type Specification allows for a programmatic description of a bundle's metadata. The IBM-defined `MetaType` Service ties those two pieces together and enables an administrative bundle to dynamically discover what another bundle's configuration looks like and make changes to it. For example, the Admin Utility for OSGi utility uses the Meta Type Service to dynamically generate the configuration screens for the `LogService`, `HttpService`, and `WebContainer`. Likewise, the Configuration Admin Preferences pages use Configuration Admin and the Meta Type Service to automatically build preference pages based on the configuration information.

The `MetaType` Service acts as a middle-man between an administrative bundle and a configurable bundle. The admin bundle may ask the `MetaType` Service for a `MetaType` Provider for a given bundle object. The `MetaType` Provider can then be queried to discover the Object Class Definitions and Attribute Definitions contained within.

For this scheme to work, configurable bundles must provide a way for the MetaType Service to discover the bundle's configuration information. Each configurable bundle must have a METADATA.XML file in the/META-INF directory of their plug-in/bundle. This METADATA.XML file contains a description of the configuration in XML format and is packaged in the bundle's JAR file.

This file describes the data in a format defined by METADATA.DTD. Within the METADATA.XML file is a list of all supported Locales. An additional set of files, METADATA_<locale>.properties should contain the translatable strings for a locale in key=value format. The METADATA.XML file contains strings for the default language only, which are used by default if the properties file for the desired locale can not be found.

The Meta Type Service is registered with OSGi under the class name com.ibm.osg.service.metatype.MetaTypeServiceand provides a way to define and retrieve org.osgi.service.metatype.MetaTypeProvider objects for an OSGi bundle. For more information on the package org.osgi.service.metatype see the OSGi Release 3 specification. The MetaTypeProvider data for each bundle is stored as a bundle resource at /META-INF/METADATA.XML.

The Meta Type data stored in /META-INF/METADATA.XML is used to define information on how to configure a bundle using the OSGi service org.osgi.service.cm.ConfigurationAdmin. For example, the Managed Service PIDs, the Managed Service Factory PIDs, and the Attributes associated are defined in the METADATA.XML file.

MetaData XML sample

The following is an example of the XML syntax for the file /META-INF/METADATA.XML.

The DTD for the METADATA.XML file is provided in the <Rational SDP Install>\sdpisv\eclipse\plugins\com.ibm.pvc.wct.runtimes_<version>\rcp\eclipse\plugins\com.ibm.osg.service.metatype_1.1.0.<version>\schema directory. In addition, a sample template for the METADATA.XML file is provided. Using the XML capabilities of the Rational Software Development Platform, you can link the METADATA.DTD to any METADATA.XML file that you create.

For a complete definition of the XML format see the METADATA.DTD and the METADATA_TEMPLATE.XML in the API document directory: TECHHOME/smf/client/docs/metatype/apidoc

```
<?xml version="1.0" encoding="UTF-8"?>
<METADATA>
        <ATTRIBUTEDEFINITION ID='log.size' TYPE='Integer' DEFAULTVALUE='100'>
                <NAME>
                      <STRING KEY='LOG_SIZE' DEFAULT='Log Size'/>
                </NAME>
                <DESCRIPTION>
                       <STRING KEY='LOG_SIZE_DESC' DEFAULT='The size of the log, in number of
                        entries'/>
                </DESCRIPTION>
                <CARDINALITY TYPE='SCALAR' SIZE='' />
                <VALIDATE>
                        <RANGE MIN='10' MAX='2000' />
                </VALIDATE>
        </ATTRIBUTEDEFINITION>

        <OBJECTCLASSDEFINITION
                ID='com.ibm.osg.service.log.Log'
```

```
            TYPE='PID'>
            <NAME>
                   <STRING KEY='LOG_SETTINGS' DEFAULT='Log Service Settings'/>
            </NAME>
            <DESCRIPTION>
                   <STRING KEY='LOG_SETTINGS_DESC' DEFAULT='The Log Service settings that may
                  be configured'/>
            </DESCRIPTION>
            <ATTRIBUTE ID='log.size' REQUIRED='yes'/>
            <ATTRIBUTE ID='log.threshold' REQUIRED='yes'/>
       </OBJECTCLASSDEFINITION>

       <LOCALES>
</METADATA>
```

Using the XML parser services

Applications requiring use of XML Parsers should use the XML Parser Service interface. By using the XML Parser Service interface, applications are able to dynamically select the parser at runtime, and are notified of parser service events by the XML Parser Service. However, to use the XML Parser service, you must modify existing applications.

Applications can use the standard JAXP calls without using the service interfaces. The APIs providing the parser factories use the underlying service interfaces. In this situation, applications will not be able to dynamically choose their parser at runtime, and they will not receive event notifications if parser services are removed. Applications might receive `javax.xml.parsers.FactoryConfigurationError` if parser actions are attempted and no parsers exist.

Typical usage of a SAX Parser involves obtaining a reference to the SAX Parser Factory, and then obtaining a new parser:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

The following example shows how you can obtain a reference to the SAX Parser Factory using XML Parser Services:

```
ServiceReference ref =
      context.getServiceReference(SAXParserFactory.class.getName() );
SAXParserFactory factory = context.getService( ref );
SAXParser parser = factory.newSAXParser();
```

It is not necessary to issue the `newInstance` call once the factory reference is obtained.

Similarly, when using the `DocumentBuilderFactory`, the typical sequence using JAXP is:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

The following example shows how you use the `DocumentBuilderFactory` using the XML Parser Service.

```
ServiceReference ref =
      context.getServiceReference( DocumentBuilderFactory.class.getName() );
DocumentBuilderFactory factory = context.getService( ref );
DocumentBuilder builder = factory.newDocumentBuilder();
```

It is not necessary to issue the `newInstance` call once the factory reference is obtained.

SAX interfaces permit setting properties on the `SAXParser` object. Since MicroXML does not support validation, attempting to set the validation property on MicroXML will result in a `org.xml.sax.SAXNotSupportedException` being thrown. If your code is not specific to either parser, then you should be prepared to receive exceptions when attempting to set features, and handle them appropriately.

Attempts to set other features can result in exceptions if different parser implementations have been plugged in. If specific parser attributes are required, you can specify them when requesting the parser factory service.

It is possible to perform validation of XML using DTD or XSD when the XML resources are loaded from a plug-in. Because the resource is loaded from a plug-in, you need to provide indication of the location of the DTD file. Create an `InputSource` object for the XML document that requires parsing. Use the `Bundle.getEntry()` or `Bundle.getResource()` method to obtain a URL or load the resource from a stream. Set the System ID for the `InputSource` to the URL used to load the file. By using this process, you do not need to create an `EntityResolver` instance to locate the DTD.

Use the following approach to permit DTD or Schema validation against files contained within bundles:

1. Set up an `InputSource` to specify the stream and the desired URL.
2. Do not use only an input stream, because an attempt to find a DTD or Schema file will result in unexpected URLs.

```
ServiceReference ref =
            context.getServiceReference(SAXParserFactory.class.getName() );
SAXParserFactory factory = context.getService( ref );
SAXParser parser = factory.newSAXParser();

      XMLReader xmlReader = parser.getXMLReader();
      xmlReader.setContentHandler(this);
         xmlReader.setErrorHandler(this);
         xmlReader.setFeature("http://xml.org/sax/features/namespaces", true);
           xmlReader.setFeature("http://xml.org/sax/features/validation", true);
         xmlReader.setFeature("http://apache.org/xml/features/validation/schema",
true );

           URL url = getClass().getResource(uri);
           java.io.InputStream is = url.openStream();
[1]        InputSource input = new InputSource(is);
[2]        input.setSystemId( url.toExternalForm() );
           xmlReader.parse(input);
```

Creating help for the application

If you are creating a set of Help information for your application, and you intend on using the built-in WebSphere Everyplace Deployment help plug-ins, you should use the Eclipse PDE to create a help plug-in. The Help Plug-in provides for XML configuration of the Table of Contents, and content specified as HTML.

For more information on creating a help plug-in, refer to the section **Plug-in Help** in the *Platform Plug-in Developer's Guide* located in the Eclipse Help system.

Using logging and tracing

WebSphere Everyplace Deployment provides three methods for logging and tracing messages for developers - via the Eclipse provided logging interfaces, the OSGi LogService interface, or the `java.util.logging` interface of the JRE.

Eclipse logging

For plug-in developers the Eclipse logging and tracing mechanism consists of just a few objects and methods. The following code provides a simple example of logging from a plug-in:

```
import org.eclipse.core.runtime IStatus;
import org.eclipse.core.runtime Status;
import org.eclipse.core.runtime Platform

IStatus status= new Status (IStatus.ERROR,
                           "Test",
                           0,
                           "Testing Eclipse Error Logging",
                           (Throwable) null);
getDefault().getLog().log(status);
```

In order to send a message to the Eclipse logging system, a Status object must be constructed and populated with all required data. Once the Status object is instantiated it is then passed to the Eclipse logger via the `log()` method. The handle to the Eclipse log is located via the `getLog()` method of the `Plugin` class (the `Plugin` class is accessed via the `getDefault()` method here).

All messages logged via the Eclipse logging mechanism are stored in the `<workspace>/.metadata/.log file`.

Note: If you are running your target platform out of the PDE, then the workspace that is being used can be determined by reviewing the launch configuration for the Run-time Workbench. On the first panel of the configuration screen the location can be found on the **Arguments** tab, in the **Workspace Data** section, in the **Location** field.

Eclipse tracing

Eclipse's tracing mechanism is based on methods from the Eclipse Platform class, `inDebugMode()` and `getDebugOption(String)` and one tracing configuration file called ″`.options`″ located in the plug-in directory. Details on Eclipse tracing can be found in the online help at **Running with Tracing** in the **PDE Guide > Getting Started > Basic Plug-in Tutorial > Running a plug-in >Running with tracing**.

Once the trace files have been setup appropriately for the plug-in and tracing has been enabled via the workbench configuration screens, the following code shows an example of how the tracing setup is used by the developer:

```
if ( Platform.inDebugMode() ) {
         if ("true".equalsIgnoreCase (Platform.getDebugOption("T2")) {
             IStatus status= new Status(IStatus.INFO,
                                   "Test",
                                   0,
                                   "Testing Eclipse Error Logging",
                                   (Throwable)null);
             getDefault().getLog().log(status);
         }
       }
```

The above example begins by checking to see if debugging has been enabled for this plug-in, if it has been enabled, then the example confirms that trace level "T2" is set to "true", if and only if these two checks are true will the Status object be created and logged to the Eclipse logger.

While the above example illustrates the trace check and a log entry created to the Eclipse logger, plug-in developers often use `System.out` or `System.err` to write out trace information. The Eclipse log file is typically reserved for Error or Warning conditions that occur during application execution.

OSGi logging

OSGi provides a service based interface for logging for OSGi applications. By obtaining a `LogService` object from the Framework service registry, a bundle can start logging messages to the Log Service object by calling on the `LogService` methods. A Log Service object can log any message, but it is primarily intended for reporting events and error conditions.

The following example demonstrates the use of a log method to write a message into the log:

```
logService.log(LogService.LOG_INFO, "Test");
```

There are four levels of messages defined by the OSGi LogService interface: `ERROR(1)`, `WARNING(2)`, `INFO(3)` and `DEBUG(4)`. The number in parenthesis is the integer value for each of the levels.

For more detailed information on the OSGi LogService interface please refer to the **Platform Plug-in Developer's Guide > Reference > OSGi API Reference in Eclipse 3.0**.

For information on managing the logging levels of the runtime platform, please see "Troubleshooting" on page 205.

Enabling your plug-in for startup

Plug-ins in the Eclipse framework generally take on one of the following status:

- **INSTALLED** - The plug-in has been recognized, but either has not been requested to start, or is incapable of starting because of missing pre-requisites
- **RESOLVED** - all of the pre-requisites exist and the plug-in is ready to start
- **ACTIVE** - The plug-in has been started and its capabilities are available to the workbench.

The framework uses settings known as Start Levels to organize the startup of the plug-ins. The framework exists at a specified level and the plug-ins are assigned to start at a specific level. The framework begins at level 0 with no plug-ins started. Next, the Framework will move to start level 1. All bundles assigned to start level 1 will then be started in some order. The framework will progress through each of the levels until it reaches the designated framework start level.

A specific set of plug-ins must always be started (ACTIVE) for the platform to execute properly. The plug-ins required at startup are specified in the config.ini file in the configuration directory. The `osgi.bundles` property defines the set of bundles required at startup. For any plug-in specified in this property, it must be able to resolve and start based upon the other plug-ins at the same start level. Plug-ins in this property should generally be limited to those essential for startup.

To improve startup performance, the WebSphere Everyplace Deployment platform will typically only activate (move to the ACTIVE state) plug-ins as they are referenced. A plug-in will automatically start when referenced if it contains the property `Eclipse-Autostart:true` in its `META-INF\MANIFEST.MF` file. You can enable or disable this value by updating the values in the Plug-in Activation section of the Runtime within the Plug-in Manifest Editor.

Plug-ins that do not provide packages for use by other plug-ins will not start on first usage. These plug-ins must be explicitly started.

The Workbench will perform the task of starting all bundles that do not contain the `Eclipse-AutoStart` attribute.

Web Applications are a special case. Since the Web Applications will appear in the application drop-down menu, or in the desktop perspective, when web applications are requested, then they will be started. Because of this, Web Applications could contain the Eclipse-AutoStart attribute to defer startup until requested.

The startup state of plug-ins that do not contain the `Eclipse-AutoStart` attribute is retained when the WebSphere Everyplace Deployment platform is shut down. Once a plug-in moves to `ACTIVE` state, it will start in all successive launches. You should not, however, depend upon this mechanism to make sure that your plug-ins are started, as the workspace location that saves this information could be removed due to problems. Plug-ins that contain `Eclipse-AutoStart` attributes will be stopped when the framework shuts down.

This table summarizes the settings of `Eclipse-AutoStart` attribute and the associated startup actions:

Table 27.

Eclipse-AutoStart attribute	**Startup action**
Eclipse-Autostart attribute not present in the `MANIFEST.MF` file	Workbench will automatically start plug-in during startup processing
Eclipse-AutoStart: true present in `MANIFEST.MF`	Eclipse will automatically activate the plug-in when used. Plug-in will be stopped when the workbench exits
Eclipse-AutoStart:false present in `MANIFEST.MF`	Plug-in will not be automatically started by any means. Plug-in will be stopped when the workbench exits. (Generally not used)

This table summarizes the various plug-in types and the recommended settings for the Eclipse-AutoStart attribute:

Table 28.

Plug-in type	**Recommended Eclipse-Autostart Settings**
Plug-in implementing extension points	Eclipse-AutoStart:true
Plug-in providing services for use by other bundles	Do not set Eclipse-AutoStart attribute
Plug-in providing packages for other plug-ins to use	Do not set Eclipse-AutoStart attribute

Table 28. (continued)

Plug-in type	Recommended Eclipse-Autostart Settings
Web Application appearing in the application list	Eclipse-AutoStart:false (The workbench will automatically start the web application when requested)
Web Application not appearing in the application list, but referred to from other web applications	Do not set Eclipse-AutoStart attribute

The cases where a plug-in would not be started automatically by one of the rules would include the plug-in that contains `Eclipse-AutoStart:false`, or it contains `Eclipse-AutoStart:true`, but does not provide any packages. In these cases, applications will be responsible for explicitly starting the required plug-ins. This can be done by using the `org.eclipse.core.runtime.Platform` object to locate a bundle by its symbolic name, and issuing methods such as startup against the resultant Bundle object.

It is not recommended that the `osgi.bundles` property be updated to include application specific bundles.

Using runtime developer tools

Using the Platform Manager

The Platform Manager tool is a troubleshooting tool which can be used to view the applications and features installed on the WebSphere Everyplace Deployment runtime. In case a particular application or feature does not function as desired the user can use the tool to view the status of application/feature dependencies (plug-ins) and start or stop them as desired.

The Platform Manager provides a single view which lists the installed applications, web applications, features and runtime plug-ins.

Installing the Platform Manager

The Platform Manager tool is available on the update site provided with the installation media. Perform the following procedure to install the Platform Manager troubleshooting tool:

1. Start WebSphere Everyplace Deployment.
2. Select **Application > Install**.
3. You should see two sites already configured. Make sure that the **updates/eval** site is checked.
4. If you do not have the media or a network attached drive, you will need to obtain the Update Site address from your IT personnel.
   a. Select **New Remote Site**.
   b. Add a name to identify the site, and enter the URL that you were provided.
   c. Click **OK**.
5. In the Sites to include in search list, select only the site that you just added, then click **Next**.
6. From the list presented, select the **Platform Manager** feature, then click **Next**.
7. Read the license terms provided, and if acceptable, then click **Next**.

8. Select the appropriate installation site.
9. Select **Finish**.
10. When prompted to install features that are unsigned, click **Install**.
11. When prompted to restart your workbench, save your work and select **Yes**.

Configuration view

The Configuration view provides the user with the list of plug-ins installed on the runtime. For each plug-in the following information is provided - the plug-in ID, status of the plug-in, the plug-in name and the location where the plug-in was installed. After you have installed the Platform Manager tool, starting and stopping any plug-in is easy:

1. Select **Plug-ins**.
2. Select the plug-in to start or stop.
3. To start a plug-in, click the **Start** button.

   If the plug-in is in the *Resolved* state it will be started and the status of the plug-in will be updated to *Active* to reflect the change. If an error occurs during plug-in activation the Platform Manager tool will display an error message to inform the user and log the error to the platform log.
4. To stop a plug-in, click the **Stop** button.

   If the plug-in is in the *Active* state it will be stopped and the status of the plug-in will be updated to *Resolved* to reflect the change. If an error occurs during plug-in de-activation the Platform Manager tool will display an error message to inform the user and log the error to the platform log.

Note: There are certain plug-ins which, if stopped, will cause the WebSphere Everyplace Deployment runtime to not function correctly. It is recommended that users not stop the following set of plug-ins:

- `org.eclipse.core.runtime` ″Core Runtime″
- `org.eclipse.core.resources` ″Core Resource Management″
- `org.eclipse.osgi` ″OSGi System Bundle″
- `org.eclipse.osgi.services` ″OSGi Release 3 Services″
- `org.eclipse.osgi.util` ″OSGi R3 Utility Classes″
- `org.eclipse.ui` ″Eclipse UI″
- `org.eclipse.ui.forms` ″Eclipse Forms″
- `org.eclipse.ui.workbench` ″Workbench″
- `org.eclipse.jface` ″JFace″
- `org.eclipse.swt` ″Standard Widget Toolkit″
- `org.eclipse.update.core` ″Install/Update Core″
- `org.eclipse.update.ui` ″Install/Update UI″
- `org.eclipse.update.configurator` ″Install/Update Configurator″

In addition to the Start and Stop actions for each plug-in, additional actions are provided for each plug-in:

- **Headers –** Displays the contents of the headers of the `MANIFEST.MF` file. If the plug-in id not originally contain a `MANIFEST.MF` file, then this action shows the contents of the platform generated `MANIFEST.MF` file.
- **Diagnostics –** Displays an errors that may cause a bundle to not resolve properly in the platform. If used on a bundle in the RESOLVED or ACTIVE statue, it will display “No unresolved constraints”.
- **Services –** Displays the services registered by the plug-in

- **Packages –** Displays the packages exported and imported by the plug-in.

Two other actions apply to the entire platform:

- **Garbage Collection –** Causes the Garbage Collection daemon to run on the platform
- **Platform Properties –** Provides a dialog to display, create, edit, and remove Java system properties.

The Configuration view also provides the user with the list of applications, web applications and features installed on the runtime. For each application or feature installed the tool will also list the plug-in dependencies and allow the user to start or stop plug-ins as desired. Viewing the list of plug-in dependencies for an installed application, web application or feature is easy:

1. Select **Applications** (for non-web applications), **Web Applications** (for web applications) or **Features** (for features)
2. Expand the tree to view the list of applications and web applications. For features, the Platform Manager tool will list the plug-ins provided by the feature.
3. Select the application or feature plug-in to view the list of application/feature dependencies.
4. To start a plug-in, click the **Start** button.

   If the plug-in is in the *Resolved* state it will be started and the status of the plug-in will be updated to *Active* to reflect the change. If an error occurs during plug-in activation the Platform Manager tool will display an error message to inform the user and log the error to the platform log. The platform log is located into the `<user workspace>\.metadata`directory.
5. To stop a plug-in, click the **Stop** button.

   If the plug-in is in the *Active* state it will be stopped and the status of the plug-in will be updated to *Resolved* to reflect that. If an error occurs during plug-in de-activation the Platform Manager tool will display an error message to inform the user and log the error to the platform log.

Using Admin Utility for OSGi

The Admin Utility for OSGi is a web application that provides simple administrative access to the WebSphere Everyplace Deployment platform focusing on its OSGi capabilities. It enables you to manage the life cycle of bundles, view the installed bundles and registered services, and manage framework and bundle start levels. You can also use Admin Utility for OSGi to configure the IBM implementation of HTTP Service, Log Service and WebContainer.

The following sections describe the tasks you can perform within Admin Utility for OSGi.

Installing the Admin Utility for OSGi

To install the Admin Utility for OSGi, perform the following procedure:

1. Start WebSphere Everyplace Deployment.
2. Select **Application > Install**. You should see two sites already configured. Make sure that the **updates/eval** site is checked.
3. If you do not have the media or a network attached drive, you will need to obtain the Update Site address from your IT personnel.
   a. Select **New Remote Site**.

   b. Add a name to identify the site, and enter the URL that you were provided.
   c. Select **OK**.
4. In the Sites to include in search list, select only the site that you just added, then click Next.
5. From the list presented, select the **Admin Utility for OSGi** feature, then click **Next**.
6. Read the license terms provided, and, if acceptable, select **Next**.
7. Select the appropriate installation site.
8. Select **Finish**.
9. When prompted to install features that are unsigned, select **Install**.
10. When prompted to restart your workbench, save your work and select **Yes**.

Starting Admin Utility for OSGi

To access the admin utility, select **Application > Open > Admin Utility for OSGi**. This starts the admin utility in a browser window.

The Admin Utility for OSGi initially displays all of the bundles that are installed on the client.

Managing start levels

The admin utility provides another way to manage start levels of both the framework and the individual bundles. From the main page of Admin Utility for OSGi you can view and edit the framework active start level and the initial bundle start levels. You can also view the start level values for each individual bundle.

To change the start level of a bundle, you must go into the bundle's information page by selecting the bundle's name in the list.

Managing bundles

You can start, stop, update, or uninstall bundles with Admin Utility for OSGi.

1. Select a bundle by checking the **Select** check box.
2. Click **Start, Stop, Update**, or **Uninstall**. Messages display in the **Admin Messages** area.

The admin utility prevents you from unintentionally disabling bundles that are directly needed to run the utility. For example, you can not perform bundle operations on Admin Utility for OSGi itself, or any bundle that provides an implementation of the OSGi Http Service, such as HTTP Service or Web HTTP Service.

Use care when manipulating bundles that the HTTP Service implementation depends on, because it might cause the admin utility to stop functioning properly.

Viewing a bundle's information

Using Admin Utility for OSGi, you can view information about a specific bundle by clicking on the bundle name.

Configuring a bundle

Some bundles are configurable through Admin Utility for OSGi. Select the bundle from the list to view the bundle's information. If a bundle is configurable, configuration input fields display at the bottom of the form. If the bundle is not configurable the text ″No Configuration Information″ displays.

A bundle is configurable if it contains a `METADATA.XML` file as one of its resources. When using Admin Utility for OSGi to configure bundles, the `ConfigurationAdmin` and `MetaType` bundles must be active. For more information, see "Using the Meta Type Service" on page 183.

Viewing registered services

Each bundle contains registered services. You can view the services that a bundle has registered, or the services on which a bundle depends, on the bundle's information page. To view a list of services and the bundles that own them, select **Services** from the **admin utility**menu.

User administration

With Admin Utility for OSGi, you can use the User Admin Service to manipulate user definitions. You can add, modify, or delete properties and credentials for existing users.

To add a user: Click Create New User. The Create New User input fields display.

Enter a user name and click **Create User**.

To delete a user: First select the check box for that user and click **Delete**. You will need to refresh the data before removing the user from the form.

To create a new group: Click Create New Group. The Create New Group form displays.

Add members to the group by selecting a member in the **Available Roles** box and clicking either **Basic** or **Required** to move the member to the **New Group Roles** box.

Click **Create Group** to finish creating the group.

Using the log viewer

You can view logs using the admin utility by selecting Log Viewer from the menu. The log displays the log level, time, message, and the name of the service or bundle for which the log entry was created.

You can install multiple Log Reader Services at a time. Log Readers can display different types of log messages. Select the desired Log Reader Service from the drop-down list at the top of the window.

The size of the log and the types of events it collects can be configured through the admin utility.

Using the JNDI Manager

The JNDI Manager is a IBM WebSphere Everyplace Client Toolkit rich client application consisting of one perspective and two views – the Binding List view and the Binding Detail view. The purpose of the Binding List view is to provide easy access to controls which allow for looking up objects bound within the JNDI on the platform. It also provides controls to add, edit or remove those objects. The Binding Detail view is where specific binding information is entered if a binding is to be added or edited.

Installing the JNDI Manager

To install the JNDI Manager, perform the following procedure:

1. Start WebSphere Everyplace Deployment.

2. Select **Application > Instal**l. You should see two sites already configured. Make sure that the **updates/eval** site is checked
3. If you do not have the media or a network attached drive, you will need to obtain the Update Site address from your IT personnel.
   a. Select **New Remote Site**.
   b. Add a name to identify the site, and enter the URL that you were provided.
   c. Select **OK**.
4. In the Sites to include in search list, select only the site that you just added, then select **Next**.
5. From the list presented, select the **JNDI Manager** feature, then select **Next**.
6. Read the license terms provided, and, if acceptable, select **Next**.
7. Select the appropriate installation site.
8. Select **Finish**.
9. When prompted to install features that are unsigned, select **Install**.
10. When prompted to restart your workbench, save your work and select **Yes**.

Starting the JNDI Manager

To access the admin utility, select **Application > Open > JNDI Manager**. This will open the main view of the utility.

The main view is the Binding List view, which provides controls to choose which JNDI provider to utilize and which location to view within the selected JNDI provider. Both of these controls are editable drop down selection controls that are automatically pre-populated with default values. The values of these selection controls can be edited by the user.

If there is an error in connecting to the provider or retrieving the bindings from the specified location, there is a red message shown in the button of the Binding List View to notify the connection failure, the table which shows the bindings will be empty, and all action buttons will be disabled.

You can create new bindings for the specific provider, or you can select existing bindings and edit or remove them. The bindings are listed in sorted order. Selecting the header for the Name column switches the sort order from ascending to descending and vice versa.

The Binding Detail view provides controls to add or update a binding in the JNDI provider. The Name and Value fields allow the user to specify (or edit) a binding name and value (simple String object value) in the JNDI provider. When adding a new binding, the user (and is selected by default) will add the binding to the currently selected location (shown in the Binding List view).

Configuring Enterprise Definitions (JNDI)

WebSphere Everyplace Deployment JNDI overview

The WebSphere Everyplace Deployment runtime provides a simple Java object JNDI registry to support the enterprise object definition needs of web applications, EJB applications, and messaging applications. The WebSphere Everyplace Deployment JNDI provider enables a local naming directory for objects running in the client platform to communicate via standard Java naming APIs. The runtime client JNDI implementation is very lightweight and does not support federation of other name spaces, rather it provides a simple hierarchical name space for client applications. In most cases, applications leveraging JNDI do not need to interact directly with JNDI Name objects and simply use `String` representations of the names to be bound or located. If your application needs to directly interact with JNDI `CompoundName` objects, please note that due to the lightweight implementation, only a restricted set of JNDI syntax properties is supported for use when creating a `CompoundName` object. In order to ensure the correct JNDI syntax properties are used, simply use the provided `NameParser` implementation from the WebSphere Everyplace Deployment JNDI provider when a `CompoundName` object is needed.

The WebSphere Everyplace Deployment JNDI provider can be directly accessed via the provider's `InitialContextFactory` class as shown below:

```
try {
 Hashtable env = new Hashtable();
 env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.ibm.pvc.jndi.provider.java.InitialContextFactory");
 InitialContext context = new InitialContext(env);

} catch (NamingException e) {
 e.printStackTrace();
}
```

This JNDI provider is also registered as the default JNDI provider for the WebSphere Everyplace Deployment platform so even if no provider is specified it will still use the above `InitialContextFactory` to generate the `InitialContext` object.

The WebSphere Everyplace Deployment JNDI provider does not persist objects or their state information across platform restarts, so the platform administrator is responsible for binding the objects each time the platform starts and configuring those objects as needed before binding them into the JNDI registry. While the application itself could programmatically register the objects that it needs each time the platform starts, the WebSphere Everyplace Deployment client provides another declarative model for JNDI bindings.

Objects that need to be bound into JNDI can be declared using Eclipse extension points, to be described in detail shortly, so that when a lookup request is made for a specific object via its JNDI name the JNDI provider will locate the declarative definition, create the object and return it to the client application on-demand. This "lazy" creation of objects provides for faster platform startup and memory allocation based on actual need, rather than expected need.

This capability is based on two characteristics of the bundles/plug-ins:

1. A `plugin.xml` must exist and must provide an entry for the `com.ibm.pvc.jndi.provider.java.binding` extension point
2. If the bundle/plug-in has a `Manifest.mf` file, it must contain the `Eclipse-AutoStart: true` entry.

The WebSphere Everyplace Client Toolkit leverages this declarative JNDI capability to automatically generate the required `plugin.xml`, and `Manifest.mf` entries for EJBs so that the WebSphere Everyplace Deployment JNDI provider can locate their declarative information upon a lookup of their JNDI name.

Using declarative JNDI

The declarative JNDI component is based on an Eclipse extension point. The use of the Eclipse extension point registry provides the ability for objects to dynamically be added and removed from the JNDI registry by providing extension points as a part of the `plugin.xml` files of installation artifacts.

This JNDI binding extension point is called `com.ibm.pvc.jndi.provider.java.binding`.

An example of the usage of this extension point would be similar to the following:

```
<extension
    point = "com.ibm.pvc.jndi.provider.java.binding">
    <binding
    jndi-name="java:comp/env/jdbc/dsname"
 objectFactory-id="com.ibm.pvc.jndi.provider.java.genericobjectfactory">
    </binding>
  </extension>
```

Note that a required component of the `com.ibm.pvc.jndi.provider.java.binding` extension point is the `objectFactory-id`. The WebSphere Everyplace Deployment client provides three `ObjectFactory` implementations:

- `EJBObjectFactory`
- `GenericObjectFactory`
- `TxnDataSourceObjectFactory`

The `EJBObjectFactory` is used exclusively for embedded transaction container bundles, while the `GenericObjectFactory` allows for an XML description of any Java object, including primitive constructor parameters, and the ability to call methods on the object once it has been created, but before it is bound into the JNDI registry and returned to a client application. The `TxnDataSourceObjectFactory` provides the ability to create and bind into JNDI transaction capable `DataSource` objects that are required by the embedded transaction container.

Note: No matter what specific object factory is used, all JNDI objects declaratively described are required to provide the `com.ibm.pvc.jndi.provider.java.binding` in their `plugin.xml` files. Some object factories may also provide another extension point to that needs to be implemented as well, such as the `GenericObjectFactory`.

EJBObjectFactory

The `EJBObjectFactory` is responsible for managing the life-cyle of EJBs in the WebSphere Everyplace Deployment runtime. Upon a client lookup of an EJB, the JNDI provider will use the `EJBObjectFactory` to locate the EJB and start it, which in turn will cause the EJB to register with the EJB container and be bound into JNDI. The WebSphere Everyplace Client Toolkit will automatically generate the appropriate `plugin.xml` entries for EJBs as a part of the deployment process, and will also add the `Eclipse-AutoStart`: true entry into the Manifest file to ensure that the EJB will not be started automatically upon platform start, rather it will be started by the `EJBObjectFactory` upon JNDI client lookup.

GenericObjectFactory

This factory provides an Eclipse extension point (described via a schema definition file) that will allow for the description of Java objects to bound into JNDI upon a client JNDI lookup. An example of the use of the new extension point to instantiate a DB2 Everyplace DataSource object is as follows:

```
<extension point="com.ibm.pvc.jndi.provider.java.genericobject">
  <object
    jndi-name="java:comp/env/jdbc/dsname"
         class="com.ibm.db2e.DB2eDataSource">
```

```
        <method name="setUrl">
    <method-parameter
             type="String"
      value="jdbc:db2e:oedb">
    </method-parameter>
        </method>
     </object>
   </extension>
```

The `com.ibm.pvc.jndi.provider.java.genericobject` extension point definition allows for:

- the jndi-name of the object
- one class name to be specified per object entry,
- a list of parameters including type (supported types listed below) and value to be used to create the constructor call to be executed to create this object,
- a list of methods to be called against this object including parameters with type (supported types listed below) and value to be used in the method calls.

The `jndi-name` and `class` attributes are the only required attributes.

The list of valid types for the parameters is as follows:

- **Objects:** `Boolean, String, Integer, Short, Long, Float, Double`
- **Primitives:** `boolean, int, short, long, float, double`

TxnDataSourceObjectFactory

In order to create transaction capable data sources which are required by the embedded transaction container of the WebSphere Everyplace Deployment runtime, standard DataSource objects, such as the `com.ibm.db2e.DB2eDataSource` are passed to the `com.ibm.pvc.txnconatiner.TxnDataSourceFactory.create()` method and a suitable DataSource for use in embedded transactions is returned. The `TxnDataSourceObjectFactory` declarative JNDI component allows for this object transformation to be declaratively described via eclipse extension points. It leverages the same `com.ibm.pvc.jndi.provider.java.genericobject` extension point as the `com.ibm.pvc.jndi.provider.java.genericobjectfactory`, so the definition of the data source is the same, but in the definition of the binding itself, the `TxnDataSourceObjectFactory` is used.

Note: Since the point of this `ObjectFactory` it to create transaction capable DataSources, the value for the class element of the `genericobject` extension point must implement `javax.sql.DataSource`. If the class provided does not, the `Factory` will throw an `InvalidObjectException` and the JNDI object being located will not be found.

The following examples shows how to create a `DB2eDataSource` that can be used by the embedded transaction container.

```
extension point="com.ibm.pvc.jndi.provider.java.binding">
    <binding
    jndi-name="java:comp/env/jdbc/txndsname"
 objectFactory-id="com.ibm.pvc.txncontainer.TxnDataSourceObjectFactory">
    </binding>
  </extension>
```

```
<extension point="com.ibm.pvc.jndi.provider.java.genericobject">
   <object
     jndi-name="java:comp/env/jdbc/txndsname"
     class="com.ibm.db2e.DB2eDataSource">
     <method name="setUrl">
       <method-parameter
```

```
        type="String"
  value="jdbc:db2e:oedb">
       </method-parameter>
     </method>
   </object>
</extension>
```

Packaging

The schema files needed for development are included in the `com.ibm.pvc.wct.extension.schemas` plug-in which is part of the WebSphere Everyplace Client Toolkit installation package and therefore is available for use in the Rational development environment. These schema files are not needed at runtime and are therefore not shipped as a part of the WebSphere Everyplace Deployment runtime.

The `ObjectFactory` implementations are delivered via 2 different plug-ins. The `GenericObjectFactory` is shipped as a part of the JNDI provider (`com.ibm.pvc.jndi.provider.java`) plug-in, while the `EJBObjectFactory` and the `TxnDataSourceObjectFactory` implementations are packaged with the Embedded Transaction Container runtime (`com.ibm.pvc.txncontainer`) - since it is only useful when dealing with the embedded transaction container.

The availability of these object factory implementations in terms of platform deployment is inconsequential, as they are delivered with the base components as needed. If other object factory implementations are developed, care will need to be taken to ensure that they are available on the client runtime as needed, as the `plugin.xml` references alone will not cause any dependencies to be registered during deployment which could result in JNDI definitions on the runtime platform with no associated object factory to bind the object into JNDI.

Extending declarative JNDI

The declarative JNDI solution leverages the Eclipse extension point registry to provide a means to declare a list of objects that should be bound in JNDI if/when a client application attempts to locate them. In order to provide an extensible mechanism for this the declarative JNDI component makes use of the JNDI `ObjectFactory` interface as a way of abstracting the JNDI provider from the specific implementation code needed to instantiate different objects to be bound in JNDI. The `javax.naming.spi.ObjectFactory` interface is very simple and contains only one method `getObjectInstance()`.

`getObjectInstance()` is called by the `lookup()` method of our JNDI provider implementation if it is unable to locate the object requested in it's current registry of JNDI objects. The JNDI provider will first read the list of object factories from an Eclipse extension point. It will then read the list of descriptions of objects to be bound into JNDI. With this information, the JNDI provider will determine if the name that is attempting to be located is in the list of names to be lazily bound, if it is, it will instantiate the `ObjectFactory` based on the id provided in the extension point registry for that name and call `getObjectInstance()` with a null object, the current name and context. The returned object from this method invocation will then be bound into the JNDI object registry and returned from the lookup method of the JNDI provider.

In the WebSphere Everyplace Deployment runtime Object factories are registered via the Eclipse extension point registry which allows for any number of `ObjectFactory` implementations to be registered and available to the JNDI Provider.

This extension point is `com.ibm.pvc.jndi.provider.java.objectfactory` and an example of the usage of this extension point would be similar to the following:

```
<extension
point="com.ibm.pvc.jndi.provider.java.objectfactory">
 </extension>
```

Life cycle Management of JNDI registry

It is required that JNDI be notified when a lazily bound object is removed from the environment so that it can be sure to unbind the object from the JNDI registry. This requirement will be met by registering a listener with the extension registry which will notify JNDI when new objects are registered and also when currently registered objects are unregistered. When this notification occurs the JNDI provider will add the new object or remove the existing object from its registry.

It is the responsibility of the JNDI provider to manage the life cycle of the binding, not of the instantiated object. The `ObjectFactory` implementation is responsible for managing the life cycle of the created object by registering any appropriate listeners, and managing the cleanup of any associated resources if the contributing bundle artifact is de-activated. It may do so by registering its own Extension Registry listener (`org.eclipse.core.runtime.IRegistryChangeListener`) for the exposed extension point.

Globalizing your application

You can globalize an application that you build on the WebSphere Everyplace Deployment platform by using the International Components for Unicode (ICU) technology. ICU4J is a set of Java classes that extend the capabilities provided by the J2SE class libraries in the areas of Unicode and internationalization support. The ICU4J classes (at version 3.2.0) are provided in the `com.ibm.icu.icu4j`, and enable you to:

- Support multiple locales
- Support bidirectional text layouts
- Create translatable plug-ins

The following Web site provides more information about the icu4j package: http://www-306.ibm.com/software/globalization/icu/index.jsp

Support for multiple locales

A locale represents a geographic place. A user's geographic location implies certain preferences for operating system and application settings, such as language character sets, date, time, and currency formats, and the direction in which text is displayed.

The default locale for a WebSphere Everyplace Deployment application is the same as the locale for the operating system of the machine on which the client is running. If you design your application to support multiple locales, the user can specify `-nl <locale code>` as a command line option when starting the client to change the default locale for the client application running on their machine.

Things to keep in mind when implementing support for multiple locales:

- Call `java.util.Locale.getDefault()` to get the current locale. If a user supplies the `-nl` parameter when starting the client, it resets the default locale value. Calling `java.util.Locale.getDefault()` would return the newly specified locale code.

- Use icu4j whenever possible to generate locale-sensitive data dynamically. The ICU4J classes provide the following objects among others:
  - DateFormat
  - MeasureFormat
  - MessageFormat
  - NumberFormat
- Do not expect the same behavior you witness in the locale you are developing in to occur in another locale. For example, ″`i`″.to `UpperCase()` does not return ″`I`″ in the Turkish locale.
- Keep in mind that sort orders vary in each locale. Call `com.ibm.icu.text.Collator` to compare international text.

Note: The Javadoc information for the icu4j.jar package is available from the following Web site: http://oss.software.ibm.com/icu4j/doc/index.html

IBM language groups

IBM identifies the following language groups:

Table 29. Group 1 languages

Locale code	**Language**
de	German
es	Spanish
fr	French
it	Italian
ja	Japanese
ko	Korean
pt_BR	Portuguese (Brazil)
zh	Chinese (Simplified)
zh_TW	Chinese (Traditional)

Table 30. Group 2 languages

Locale code	**Language**
ar	Arabic
cs	Czech
da	Danish
el	Greek
fi	Finnish
hu	Hungarian
iw	Hebrew
nl	Dutch
no	Norwegian
pl	Polish
pt	Portuguese
ru	Russian
sv	Swedish

Table 30. Group 2 languages (continued)

Locale code	Language
tr	Turkish

Table 31. Group 3 languages

Locale code	Language
be	Belorussian
bg	Bulgarian
ca	Catalan
et	Estonian
hi	Hindi
hr	Croatian
is*	Icelandic
lt	Lithuanian
lv	Latvian
mk	Macedonian
ro	Romanian
sk	Slovak
sl	Slovenian
sq	Albanian
sr	Serbian
th	Thai
uk	Ukrainian

*Applies to selected OS/400® requirements only.

Supporting preferred fonts and bidirectional layouts

Each locale has a preferred set of fonts. Display your application text using these fonts whenever possible to maintain a unified look and feel for the user. Bi-directional (bi-di) support is important for Arabic and Hebrew.

To support preferred fonts and bi-di:

1. Instead of using a hard-coded font to format the text you display in the application, retrieve the font that is preferred for a specific locale by calling the `org.eclipse.jface.resource.JFaceResources` class. The `JFaceResources` class provides the following methods for retrieving preferred fonts:
   - getBannerFont()
   - getDialogFont()
   - getHeaderFont()
   - getTextFont()
   - getViewerFont()
2. Test your plug-in in RTL mode to be sure it responds correctly.

Creating translatable plug-ins

Customize your plug-in to display appropriately to an international audience by:

- Separating out all hard-coded text strings that appear in the user interface, in error messages, message boxes, or titles that display in title bars
- Enabling the date and number objects you use to be formatted based on the user's preferred locale

To ensure that a plug-in is translatable, do the following:

1. Define all translatable strings in a `.properties` file associated with the plug-in. For example, instead of defining the name of the plug-in in the `plugin.xml` file, define it in a file called `plugin.properties`, which associates the `.properties` file to the `plugin.xml` file. For example, define the name of the plug-in in the `plugin.xml` file as follows:
   ```
   <plugin   name="%plugin.name"
   ```
2. In the `plugin.properties` file, include the following text to define the `%plugin.name` keyword:
   ```
   plugin.name = My Super-useful Plugin
   ```
3. Use the Rational Software Development Platform or a similar product to search your source code for all translatable strings. Replace each string with a keyword and define the keyword in a `.properties` with the same name as the file that contained the translatable string.
4. Identify the locale-specific objects in your plug-in, then organize them into categories and store them in different `ResourceBundle` objects accordingly. For example, you can store a series of String objects in a `PropertyResourceBundle`, which is backed up by a set of properties files, or you can manage all locale-specific objects using a `ListResourceBundle`, which is backed up by a class file. Though the `ListResourceBundle` object requires you to code and compile a new source file to support any additional locales, `ListResourceBundle` objects are useful because unlike properties files, they can store any type of locale-specific object, not just text objects.
5. Use the standard Java library classes that localize the formatting for numbers, dates, times, and currencies. None of these object types can be displayed or printed without first being converted to a String. The formatting classes enable you to use the proper format for the user's locale when converting an object into a String object. For example, if you use the factory methods provided by the `icu4j.jar` package `NumberFormat` class, you can get locale-specific formats for numbers, currencies, and percentages. The `DateFormat` class provides predefined formatting styles for dates and times that are locale-specific and easy to use.
6. Create a plug-in fragment to contain all .properties files associated with a specific language group. IBM uses the following convention when naming the language fragments:
   ```
   Language pack fragment containing IBM Group 1 languages =
      <plugin id>.nl1_<version>
   Language pack fragment containing IBM Group 2 languages =
      <plugin id>.nl2_<version>
   ```
7. Provide a feature to group the Language packs you are including in your application. Use the following convention when naming the features:
   ```
   Language pack feature containing IBM Group 1 languages =
      <feature id>.nl1_version
   Language pack feature containing IBM Group 2 languages =
      <feature id>.nl2_version
   ```
   Specify any translatable text in the `feature.xml` file in an associated `feature.properties` file.

See the eclipse.org Web site for more details on internationalizing a plug-in at http://www.eclipse.org/articles/Article-Internationalization/how2I18n.html.

Troubleshooting

Logging

You use logs to gather information about problems that might happen while using WebSphere Everyplace Deployment for Windows and Linux. This section describes how to access logs, adjust logging levels, and configure how WebSphere Everyplace Deployment manages log files.

Logging framework

The OSGi framework, Eclipse framework and the 1.4 JDK (JSR47) all provide different systems for logging messages. Applications in some cases also leverage standard error and standard out for message delivery, and these messages must also be captured to ensure all data is available during the problem determination stage. The WebSphere Everyplace Deployment runtime `com.ibm.pvc.wct.internal.logredirector` plug-in, hereafter referred to as logRedirector, provides the ability to collect all messages logged in the WebSphere Everyplace Deployment runtime into one persistent log file. The logRedirector captures messages logged from the OSGi logService, the Eclipse logging APIs, and standard error and standard out and redirects them to the JDK 1.4 `java.util.logging`. A java.utils.logging log file manager is also provided with the WebSphere Everyplace Deployment runtime (`<`*`installation directory`*`>/rcp/loggerboot.jar`) which supports configuration of the JDK logging and manages the persistent log file in `<USER_HOME>\IBM\RCP\<INSTALL_ID>\<USER_NAME>\logs\rcp.log.*` where * represents the different generations of log files, such as rcp.log.0.

Each of the logging systems available in the WebSphere Everyplace Deployment runtime has its own definition of logging levels. In order to bring all of the messages from these disparate systems together a mapping was needed between the logging levels. The following section describes this mapping.

OSGi to JDK level mapping:

Because `java.util.Level` doesn't provide a DEBUG level, OSGi DEBUG messages will be written as FINEST messages. ERROR messages are mapped to SEVERE messages. INFO and WARNING messages are mapped directly.

Table 32. Log level mapping between the OSGi log and the java.util.Level

OSGi Log Level	java.util.Level
ERROR	SEVERE
WARNING	WARNING
INFO	INFO
DEBUG	FINEST

Eclipse to JDK level mapping:

Table 33. Log level mapping between the Eclipse log and the java.util.Level

Eclipse Log Level	java.util.Level
CANCEL	SEVERE

Table 33. Log level mapping between the Eclipse log and the java.util.Level (continued)

Eclipse Log Level	java.util.Level
ERROR	WARNING
WARNING	INFO
OK	FINEST

Manually adjusting the logging level

The logging framework default `java.util.logging` configuration properties are stored in the `plugin_customization.ini` file, which resides in the `<`*`installation directory`*`>/rcp directory`. The properties are in <key, value> format, and are set to the following by default:

Table 34. Default log properties at startup time

Properties	plugin_customization.ini default values
Handlers	java.util.logging.ConsileHandler com.ibm.rcp.core.logger.boot.RCPFileHandler
level	WARNING
com.ibm.rcp.core.logger.boot. RCPFileHandler.level	FINEST
com.ibm.pvc.wct.internal.logredirector.level	FINEST

The `plugin_customization.ini` file allows one level of customization of the WebSphere Everyplace Deployment runtime logging framework. All properties defined for `java.util.logging.LogManager` are supported. This file can be used to set additional or alternate handlers, log levels for handlers and for loggers. Default handlers are: `java.util.logging.ConsoleHandler` and `com.ibm.rcp.core.logging.RCPFileHandler`. To replace these handlers use a property such as:

```
handlers=<a list of handlers separated by white spaces>
```

To add additional handlers to the default simply include the default handlers in the list.

Logger namespaces are hierarchical so `com.ibm.level=Severe` will turn off all of our logging below SEVERE by default and then you can override this for individual points in the hierarchy that you are interested in. The following is an example of typical levels:

```
com.level=INFO
com.ibm.level=FINEST
```

The first entry sets the level for the logger for the com package and all loggers created from packages that are children of it to INFO. The second entry overrides this setting for the logger created for the package com.ibm, setting it to FINEST.

For more information on configuring the JDK logger please refer to the JDK documentation for the java.util.logging package.

The **logRedirector** component of the WebSphere Everyplace Deployment runtime also allows for configuration of the level of Eclipse Log and OSGi Log Service messages that are sent to the console and to the persistent log file. The system property `-Dlogredirector.level` is read to configure the logging level of the logredirector. This can be added into the `<`*`installation directory`*`>\rcp\rcpinstall.properties` file in order to have the property set at platform launch time. If this property cannot be accessed, the loggerRedirector will use the default setting "WARNING".

The standard error and standard out messages are also logged to the persistent log file and sent to the console. To configure what level the standard error and standard out messages should be logged to the JDK logger, the following 2 properties can be set: `logredirector.err.level` and `logredirector.out.level`. These can be set in the `rcpinstall.properties` listed above, and both default to INFO.

Log file management

This section provides information on how WebSphere Everyplace Deployment manages the log file.

The persistent log file for the WED platform is managed by a `java.util.logging FileHandler`. The `logger.properties` file located in the `<`*`installation directory`*`>/rcp/` directory can be used to change the management policy for the log file. The default settings for the log file manage are shown in the following table.

Table 35. Log properties and their default values

Configuration variables	**logger.properties default values**
log.append	false
log.generations	12
log.size	2000000
logfile.formatter	java.util.logging.SimpleFormatter

If `log.append` is set true, the new messages will be written from the end of the log file, while the default is to create a new log file each time the runtime is started. The maximum number of generations of log files is 12. Any setting that is bigger than 12 is treated as 12. The maximum size of log file is 2000000. Any setting that is bigger than 2000000 is treated as 2000000.

Note: A value of 0 for the size will be treated as an unlimited file size. If you would like to stop all logging from the WebSphere Everyplace Deployment runtime to the file system, you can change the `com.ibm.rcp.core.logger.boot.RCPFileHandler.level` entry of the `<`*`installation directory`*`>/rcp/plugin_customization.ini` file from its default of FINEST to OFF. When you have updated this setting, and restarted the platform, logging data will no longer be written to a persistent file.

The `logfile.formatter` sets the log file format, which defaults to the `SimpleFormatter`. For more information on these configuration options, please refer to the `java.util.logging.FileHandler` JDK documentation.

Tracing

Several of the WebSphere Everyplace Deployment runtime components, such as the Embedded Transaction Container, and the Web Container, have detailed tracing capabilities for enhanced problem determination in specific areas of the runtime.

For specific trace enablement information for the WebSphere Everyplace Deployment runtime components please refer to the specific documentation for each component.

The Enterprise Management Agent provides extensive tracing for debugging purposes. For WebSphere Everyplace Deployment 6.0 a Java system property is used by the agent to enable the debug tracing.

Add the following system property to enable debug tracing.

```
com.ibm.osg.service.osgiagent.osgiagent.debug=true
```

For the changes to take effect, the runtime client must be restarted.

Refer to "Configuring Java system properties" on page 223 for more information.

Using the IBM Support Assistant

IBM Support Assistant is designed to help answer questions and gather data for service personnel if needed.

IBM Support Assistant provides the following functions:

- The ability to perform a federated search across information repositories
- Convenient access to support-related web resources
- A way to automate data collection and transmission to IBM to expedite problem resolution.

The search component can be used to execute a federated search that concurrently accesses multiple search locations and returns results in a hierarchical arrangement. The support links component is a consolidated list of IBM web links organized by brand and product. The service component allows for the gathering of problem determination data and assists in the process of creating a problem management record with IBM Support.

You access the IBM Support Assistant from the WebSphere Everyplace Client Toolkit by selecting the Help menu option, then selecting IBM Support Assistant. A User's Guide is available from the Help tab on the navigation bar of the IBM Support Assistant page.

To capture problem determination:

1. When you have opened the IBM Support Assistant select the **Service** tab.
2. Click **Invoke Collector**. ISA gathers the problem determination information. When the collector completes a panel is displayed noting the location of the archive that was created that contains all of this information. This archive will be needed for detailed problem determination with IBM Support.
3. From the Service tab, you can send the collected information to IBM Support using the Send System Data option. You select a geographic location, and the data is sent to one of the following destination servers depending upon your selection:

North America and Asia Pacific:testcase.boulder.ibm.com\ps\toibm\pvc

EMEA: ftp.emea.ibm.com

Data collection details:

When the collector is invoked from the Service tab of the IBM Support Assistant, the collected data is the same as that collected with a Rational product. Included in the collected data is:

- Installation configurations, properties and logs
- RAD/RWD/RSM logs
- Performance Tester configurations and logs
- User workspace log
- RAD/RWD/RSM/Performance Tester Workspace files
- System Data
- Network data

For more information refer to the IBM Support Assistant User's Guide, which is available via the WebSphere Everyplace Deployment runtime at Help > IBM Support Assistant.

Reference information

WebSphere Everyplace Deployment top level menus

The WebSphere Everyplace Deployment platform defines a set of default menu items. This section provides the identifiers that are required to be used for either of the following cases:

- You want to add additional menu items in a specific location to one of the pre-existing menus
- You want to define activities to allow you to group menu items within an activity group.

The Top Level Menu Items as shown in the table below are defined by WebSphere Everyplace Deployment. These menu items cannot be removed through the use of activities.

Table 36. Top level menus

Menu Name	Menu ID	Plug-in ID
File	file	com.ibm.eswe.workbench
Application	application	com.ibm.eswe.workbench
View	view	com.ibm.eswe.workbench
Help	help	com.ibm.eswe.workbench

File menu

The menu items, markers, and separators defined for the File menu are defined in the following table.

Table 37. File menu

Menu Group	Menu Item	ID	Plug-in ID
	group marker	fileStart	
	group marker	fileEnd	
	separator		
	group marker	prefStart	
prefStart	Preferences	preferences	com.ibm.eswe.workbench
	group marker	prefEnd	
	separator		
	group marker	additions	
	separator		
	Exit	quit	com.ibm.eswe.workbench

Application menu

The menu items, markers, and separators defined for the Application menu are defined in the following table.

Table 38. Application menu

Menu Group	Menu Item	ID	Plug-in ID
	group marker	applicationStart	
applicationStart	Open	open	com.ibm.eswe.workbench
applicationStart	Close	closePerspective	com.ibm.eswe.workbench
applicationStart	Close All	closeAll	com.ibm.eswe.workbench
applicationStart	Reset	ResetPerspective	com.ibm.eswe.workbench
	group marker	applicationEnd	
	separator		
	group marker	additions	
install	Install	install	com.ibm.eswe.installupate.launcher
management	Application Management	management	com.ibm.eswe.installupate.launcher

View menu

The menu items, markers, and separators defined for the View menu are defined in the following table.

Table 39. View menu

Menu Group	Menu Item	ID	Plug-in ID
	group marker	viewStart	com.ibm.eswe.workbench
viewStart	Show Icon Labels	show_iconlabels	com.ibm.eswe.workbench
viewStart	Show Large Icons	show_largeicons	com.ibm.eswe.workbench
viewStart	Show Small Icons	show_smallicons	com.ibm.eswe.workbench
viewStart	Show Application Switcher	show_switcher	com.ibm.eswe.workbench
viewStart	Show Banner	show_banner	com.ibm.eswe.workbench
viewStart	Show Coolbar	show_coolbar	com.ibm.eswe.workbench
	group marker	viewEnd	
	Separator		
	group marker	additions	

Help menu

The menu items, markers, and separators defined for the Help menu are defined in the following table:

Table 40. Help menu

Menu Group	Menu Item	ID	Plug-in ID
	group marker	helpStart	
helpStart	Help Contents	helpContents	com.ibm.eswe.workbench
	group marker	helpEnd	
	separator		
	group marker	additions	
	separator		
escGroup	IBM Support Assistant	com.ibm.esupport.client.Browser	com.ibm.esupport.client

Table 40. Help menu (continued)

Menu Group	Menu Item	ID	Plug-in ID
	About WebSphere Everyplace Deployment	about	

Extension points reference

This section describes the extension points WebSphere Everyplace Deployment provides for application development. The following extension points can be used to extend the capabilities of the platform infrastructure:

Applications

- "com.ibm.eswe.workbench.WctApplication"

Web applications

- "com.ibm.eswe.workbench.WctWebApplication" on page 214

JNDI Binding

- "JNDI Binding" on page 217

JNDI generic object

- "JNDI generic object" on page 218

JNDI object factory

- "JNDI object factory" on page 219

Applications

com.ibm.eswe.workbench.WctApplication

This extension point provides for the definition of an application to be launched.

Since: WCTME Enterprise Offering 5.8.0

Configuration markup:

```
<!ELEMENT extension EMPTY>
 <!ATTLIST extension
 point CDATA #REQUIRED
 id    CDATA #IMPLIED
 name  CDATA #IMPLIED>
```

- `point` - Fully qualified identifier of the target extension point
- `id` - ID identifying this instance of the extension point.
- `name` - Name associated with the extension point

```
<!ELEMENT DisplayName (#CDATA)>
```

`DisplayName` - Display Name to use for the application in the **Application > Open** and **Application Switcher** menus, or on the Desktop perspective. Required.

```
<!ELEMENT PerspectiveId (#CDATA)>
```

`PerspectiveId` - ID for the perspective to be launched when the application is selected.

```
<!ELEMENT Icon (#CDATA)>
```

`Icon` - Relative path of `icon` to be used in the **Application > Open** and **Application > Switcher** menus. Optional.

If specified, application developers should provide both a 16x16 and a 32x32 pixel color image. If the image size is not 16x16 or 32x32 or if the `<Icon>` element is not specified, then the default web application images will be used. Image name should contain either ’16x16’ or ’32x32’ to denote the size of the image.

Examples:

1. This application shows a rich client application to run on the WebSphere Everyplace Deployment runtime.

```
<extension point="com.ibm.eswe.workbench.WctApplication">
    <DisplayName>Order Entry Rich Client sample</DisplayName>
    <PerspectiveId>com.ibm.eswe.orderentry.OrderEntryPerspective
    </PerspectiveId>
</extension>
```

2. This application shows a rich client application that defines an icon.

```
<extension point="com.ibm.eswe.workbench.WctApplication">
    <DisplayName>Order Entry Rich Client sample</DisplayName>
    <PerspectiveId>com.ibm.eswe.orderentry.OrderEntryPerspective
    </PerspectiveId>
    <Icon>OEWctApp_32x32.gif</Icon>
</extension>
```

Web applications

com.ibm.eswe.workbench.WctWebApplication

This extension point provides the definition of a web application to be launched.

Since: Enterprise Offering 5.8.0

Configuration markup:

```
<!ELEMENT extension EMPTY>
      <!ATTLIST extension
      point CDATA #REQUIRED
      id    CDATA #IMPLIED
      name  CDATA #IMPLIED>
```

- **point** - Fully qualified identifier of the target extension point
- **id** - ID identifying this instance of the extension point. If the web application is using an `IUrlProvider` implementation to generate the URL, then the id should not contain any decimals (’.’)..
- **name** - Name associated with the extension point

```
<ELEMENT DisplayName (#CDATA)>
```

DisplayName - Display Name to use for the application in the **Application > Open** and **Application Switcher** menus. Required.

```
<!ELEMENT Url (#CDATA)>

 <!ATTLIST Url
      provider CDATA #IMPLIED
      local    CDATA #IMPLIED
      secured  CDATA #IMPLIED>
```

Url - The `Url` text portion of the element specifies either the context root and application specific path for a Local application, or the entire URL for a remote application. Required.

The `Url` element will contain the following attributes:

- A `provider` attribute that specifies the name of a class that will return the `Url` to be displayed. If the provider attribute is present, the text value of the `Url` element is not required. Any provider must implement the `IUrlProvider` interface and use the `IPageDescriptor` API to retrieve application information required to construct the `Url`.
- A `local` attribute that indicates whether the content of the `Url` text portion is intended to be run against the local web container, or is a full URL. Values are true or false. This replaces the Local element that previously existed. The default is true. If the provider attribute is specified, the workbench will do nothing with this value (except for setting it up within the `IPageDescriptor` implementation)
- A secured attribute used only `if local="true"` that indicates that HTTPS should be used by the browser to connect to the web application. Values are true or false. The default is false. If the provider attribute is specified, the workbench will do nothing with this value (except for setting it up within the `IPageDescriptor` implementation)
- The `id` attribute defined as part of the extension element is required if the provider attribute of the `Url` is used.

Note: The value of the id attribute must be a simple string with no special characters (e.g. '.').

```
<!ELEMENT Icon (#CDATA)>
```

Icon - Relative path of icon to be used in the **Application > Open** and **Application > Switcher** menus. Optional.

If specified, web application developers should provide both a 16x16 and a 32x32 pixel color image. If the image size is not 16x16 or 32x32 or if the `<Icon>` element is not specified, then the default web application images will be used. Image name should contain either '16x16' or '32x32' to denote the size of the image.

```
<!ELEMENT BrowserOptions>
 <!ATTLIST BrowserOptions
       browser CDATA #IMPLIED
       showAddressbar    CDATA #IMPLIED
       showToolbar  CDATA #IMPLIED
       showHistory  CDATA #IMPLIED
       showHome  CDATA #IMPLIED
       showPageCtrl  CDATA #IMPLIED
       showPrint  CDATA #IMPLIED
       showBookmark  CDATA #IMPLIED
       userid  CDATA #IMPLIED
      password  CDATA #IMPLIED >
```

BrowserOptions- Configures browser options. Optional

The `BrowserOptions` element contains the following attributes:

- A browser attribute that specifies the type of browser to use. The supported values are "platform", "MSIE" and "Mozilla". The default for Windows is MSIE, the default for Linux is Mozilla.

  Note: Mozilla on Windows is not supported for this release.

- A `showAddressbar` attribute that specifies whether or not the browser address bar displays. The supported values are “true” or “false”. The default is “true”.
- A `showToolbar` attribute that specifies whether or not the browser tool bar displays. The supported values are ″true″ or ″false″. The default is ″true″. If `showToolbar` is set to ″false″, then none of the toolbar buttons (Print, Stop, etc...) will display.
- A `showHistory` attribute that specifies whether or not the browser Back and Forward buttons display. The supported values are ″true″ or ″false″. The default is ″true″.
- A `showHome` attribute that specifies whether or not the browser Home button bar displays. The supported values are “true” or “false”. The default is “true”.
- A `showPageCtrl` attribute that specifies whether or not the browser Stop and Refresh buttons display. The supported values are ″true″ or ″false″. The default is ″true″.
- A `showPrint` attribute that specifies whether or not the browser Print button displays. The supported values are “true” or “false”. The default is “true”.
- A `showBookmark` attribute that specifies whether or not the browser Bookmark button displays. The supported values are “true” or “false”. The default is “true”.
- A `userid` attribute that specifies the `username` to use to replace the `%USERID%` tag in the web application URL.
- A `password` attribute that specifies the `password` to use to replace the `%PASSWORD%` tag in the web application URL.

Examples

1. The following example uses the `Url` element to specify the web application URL, and shows the browser address bar while hiding the Home button:

```
<?eclipse version="3.0"?>
<plugin>
   <extension
     point="com.ibm.eswe.workbench.WctWebApplication">
    <DisplayName>%webapp.name</DisplayName>
    <Url local="true" secured="false">/OrderEntry</Url>
    <BrowserOptions browser="platform"
          showAddressBar="true"
          showHome="false"/>
     <Icon>icons/OEwctwebapp_32x32.gif</Icon>
  </extension>
</plugin>
```

2. The following example specifies a secured web application URL, and removes the browser address bar and toolbar:

```
<?eclipse version="3.0"?>
<plugin>
   <extension

   point="com.ibm.eswe.workbench.WctWebApplication">
     <DisplayName>%webapp.name</DisplayName>
     <Url local="true" secured="true">/OrderEntry</Url>

     <BrowserOptions browser="platform"
          showAddressBar="false"
          showToolbar="false"/>
    <Icon>icons/OEwctwebapp_32x32.gif</Icon>
   </extension>
</plugin>
```

3. The following example uses the Mozilla browser, and removes Print button from toolbar:

```
<?eclipse version="3.0"?>
<plugin>
   <extension      point="com.ibm.eswe.workbench.WctWebApplication">
    <DisplayName>%webapp.name</DisplayName>
    <Url local="true">/OrderEntry</Url>
    <BrowserOptions browser="Mozilla"
    showPrint="false"/>
    <Icon>icons/OEwctwebapp_32x32.gif</Icon>
   </extension>
</plugin>
```

4. The following example shows how to use the `UrlProvider` capability:

```
<?eclipse version="3.0"?>
<plugin>
   <extension
   id=MyApplication   point="com.ibm.eswe.workbench.WctWebApplication">
    <DisplayName>%webapp.name</DisplayName>
    <Url provider="myApplication.myProvider"/>
    <BrowserOptions browser="platform"
        showAddressBar="true"
        showToolbar="true"
        showHistory="true"
        showPageCtrl="true"
        showHome="false"
        showPrint="true"/>
<Icon>icons/OEwctwebapp_32x32.gif</Icon>
   </extension>
</plugin>
```

This extension point will be used to display the menu items that appear in the **Application > Open** menu, and to display the web applications within the Desktop view.

JNDI Binding

com.ibm.pvc.jndi.provider.java.binding

This extension point provides the ability to define objects to be bound into JNDI upon client JNDI lookup.

Since: WebSphere Everyplace Deployment for Windows and Linux 6.0

Configuration markup:

```
<!ELEMENT extension (binding+)>
    <!ATTLIST extension
    point CDATA #REQUIRED
    id    CDATA #IMPLIED
    name  CDATA #IMPLIED>

<!ELEMENT binding EMPTY>
    <!ATTLIST binding
    jndi-name        CDATA #IMPLIED
    objectFactory-id CDATA #IMPLIED>
```

- `jndi-name` - The location in JNDI where this object should be bound
- `objectFactory-id` - The `javax.naming.spi.ObjectFactory` implementation responsible for creating and binding this object into JNDI

Examples:

```
  <extension point="com.ibm.pvc.jndi.provider.java.binding">

<binding
```

```
  jndi-name="java:comp/env/jdbc/dsname"
  objectFactory-id="com.ibm.pvc.jndi.provider.java.genericobjectfactory">
    </binding>
 </extension>
```

API information: No Java code is required for this extension point.

Supplied implementation: No implementations of this extension point are provided with the platform.

JNDI generic object

com.ibm.pvc.jndi.provider.java.genericobject

This extension point provides the ability to define Java objects in a declarative XML way such that they can be created and bound into JNDI upon client JNDI lookup.

The application using this extension point is responsible for providing:

- the jndi-name of the object
- one class name to be specified per object entry
- a list of parameters including type (supported types listed below) and value to be used to create the constructor call to be executed to create this object
- a list of methods to be called against this object including parameters with type (supported types listed below) and value to be used in the method calls

The `jndi-name` and `class` attributes are the only required attributes.

The list of valid types for the parameters is as follows:

- **Objects**: Boolean, String, Integer, Short, Long, Float, Double
- **Primitives**: boolean, int, short, long, float, double

Since: WebSphere Everyplace Deployment for Windows and Linux 6.0

Configuration markup:

```
<!ELEMENT extension (object+)>
    <!ATTLIST extension
    point CDATA #REQUIRED
    id    CDATA #IMPLIED
    name  CDATA #IMPLIED>

<!ELEMENT object (constructor-parameter* , method*)>
    <!ATTLIST object
    jndi-name CDATA #IMPLIED
    class     CDATA #IMPLIED>
```

- `jndi-name` - The location in JNDI where this object should be bound
- `class` - The Java class to be instantiated

```
<!ELEMENT constructor-parameter EMPTY>

<!ATTLIST constructor-parameter

type  CDATA #IMPLIED

value CDATA #IMPLIED>
```

- `type` - The type of the parameter to the constructor
- `value` - The value for the constructor parameter

```
   <!ELEMENT method (method-parameter*)>
       <!ATTLIST method
       name CDATA #IMPLIED>
```

- **name** - The name of the method on the Java object to call once the object has been instantiated

```
...<!ELEMENT method-parameter EMPTY>
.......<!ATTLIST method-parameter
.......type  CDATA #IMPLIED
.......value CDATA #IMPLIED>
```

- `type` - The type of the parameter to the method
- `value` - The value for the method parameter

Examples:

The following example shows the use of the new extension point to instantiate a DB2 Everyplace DataSource object:

```
<extension point="com.ibm.pvc.jndi.provider.java.genericobjectfactory">
  <object

    jndi-name="java:comp/env/jdbc/dsname"
         class="com.ibm.db2e.DB2eDataSource">

        <methods name="setUrl">

<method-parameter
         type="String"
        value="jdbc:db2e:oedb">

</method-parameter>

        </method>
     </object>
   </extension>
```

API information: No Java code is required for this extension point.

Supplied implementation: No implementations of this extension point are provided with the platform.

JNDI object factory

com.ibm.pvc.jndi.provider.java.objectfactory

This extension point provides a way to extend the list of available ObjectFactories to be used by JNDI to dynamically bind objects into the JNDI object registry upon client JNDI lookup.

Since: WebSphere Everyplace Deployment for Windows and Linux 6.0

Configuration markup:

```
   <!ELEMENT extension (objectfactory+)>
       <!ATTLIST extension
       point CDATA #REQUIRED
       id    CDATA #IMPLIED
       name  CDATA #IMPLIED>

<!ELEMENT objectfactory EMPTY>
<!ATTLIST objectfactory
class CDATA #REQUIRED
id    CDATA #REQUIRED>
```

- `class` - The name of the class that implements `javax.naming.spi.ObjectFactory` to be instantiated and used to dynamically bind objects into the JNDI registry.
- `id` - The id that binding extension point implementors will use to reference this `ObjectFactory`

Examples:

```
<extension point="com.ibm.pvc.jndi.provider.java.objectfactory">

<objectfactory
     id="com.ibm.pvc.txncontainer.EJBObjectFactory"
     class="com.ibm.pvc.txncontainer.EJBObjectFactory">

  </objectfactory>

 </extension>
```

API information: Classes that leverage this extension point must implement the `javax.naming.spi.ObjectFactory` interface.

Supplied implementation: There are 2 implementations of this extension point provided with the platform:

- The Generic Object Factory - used to instantiate Java objects and bind them into the JNDI registry
- The EJB Object Factory - used to instantiate and bind EJBs into the JNDI registry

WebSphere Everyplace Client Toolkit ANT task and types

The toolkit provides an ANT task and types that can be used to export projects into JAR files compatible with update sites. This ANT task can be used with any Client Services project.

bde.exportJarBundle

Description

The `bde.exportJarBundle` task provides the capability to export any Client Services project to a JAR that can be included in an update site.

Parameters

Table 41. Parameters

Attribute	Description	Required
projects	Comma separated list of projects to be exported by this task invocation	yes
exportFileNames	Comma separated list of file names, with a one-to-one correspondence with the list of projects to export. If this attribute is not specified, then the default naming scheme is `<Bundle-SymbolicName>_<Bundle-version>.jar`	optional

Nested elements

"filedirectory" - This required element is used to specify a target directory to contain the exported JAR file(s).

"buildpolicyinfo" - - This optional element can be used to specify properties to apply during the export process.

Examples

The following example invokes the task to export two projects from the workspace. The resulting file names will be `com.ibm.pvc.tools.samples.db2e_1.0.0.jar` and `com.ibm.pvc.tools.samples.mqeclient_1.0.0.jar` (since both projects have versions of 1.0.0). This is the default export file name format. Both JARs will be exported to the current directory.

```
<bde.exportJarBundle projects="com.ibm.pvc.tools.samples.db2e,
                              com.ibm.pvc.tools.samples.mqeclient">

 <filedirectory path="." />
      <buildpolicyinfo includeSource="false"
                       abortOnErrors="false"
                       isDebugCompilation="true"
                       isVerboseCompilation="true" />
</bde.exportJarBundle>
```

filedirectory

Description

Defines the target directory to contain the exported JAR file(s).

Parameters

Table 42. Parameters

Attribute	Description	Required
path	Specifies the file directory location to contain the exported JAR files	yes

buildpolicyinfo

Description

Defines a set of properties to apply during the compilation and export process.

Parameters

Table 43. Parameters

Attribute	Description	Required
includeSource	Specifies whether to include source in the output jar. A value of true indicates source should be included.	optional

Table 43. Parameters (continued)

Attribute	Description	Required
abortOnErrors	Specifies whether to immediately stop the process when errors are encountered. A value of true indicates that errors should terminate processing.	optional
isVerboseCompilation	Specifies whether compilation should be performed in verbose mode. A value of true indicates that verbose compilation should be used.	optional
isDebugCompilation	Specifies whether debug information should be included in the compiled class files. A value of true indicates the debug information should be included.	optional

Managing client configurations

This section describes the tasks you can perform to configure the WebSphere Everyplace Deployment runtime on the user's machine.

Understanding the client file layout

When WebSphere Everyplace Deployment is installed on a machine, the installer creates a directory structure in the installation directory. This section describes the layout of the installation directory.

This section describes the layout of the files installed on the runtime.

```
<installation directory>/
    _uninst/                 - Files required for uninstalling the product
    eclipse/                 - Platform components
         .eclipseproduct
         configuration
         features/
         links/
         plugins/
    rcp/                     - Platform components
          eclipse/
             .eclipseproduct
             features/
             plugins/
    shared/                  -Site to contain applications shared across multiple
                              configurations
         eclipse/
             features/
             plugins/
    license/                 - Product licenses in multiple languages
    migration/               - WCTME EO 5.8.1 migration utility
```

The `.eclipseproduct` marker files are used by the `installupdate` plug-in. When an installation site is marked with `.eclipseproduct` marker it is filtered so that a use cannot uninstall it. A user is allowed to install into these directories, but only after a warning. An ISV should add the `.eclipseproduct` marker to their installation directories to obtain the same filtering.

You can also control how these features are updated by adding these base features to your features. The root features has the ability to control the update site that is used. See <includes> <search-location> in the feature manifest documentation for more information.

Configuring Java system properties

Applications might require specific system properties to be set at startup when running the WebSphere Everyplace Deployment platform. To minimize the number of parameters that you must specify on the command line, you can add configuration lines to the `rcpinstall.properties` file, which resides in the <*installation directory*>`/rcp/` directory.

Refer to "Updating the rcpinstall.properties file" on page 228 for detailed information.

You can also specify properties on the command line when you launch the platform. Refer to "Configuring the platform launcher" on page 236 for more information.

Configuring Java VM arguments

Applications might require the addition of JVM specific arguments when the platform starts. To minimize the number of parameters that must be specified on the command line, you can add vmarg.* configuration lines to the `<installation directory>/rcp/rcpinstall.properties` file.

Refer to "Updating the rcpinstall.properties file" on page 228 for detailed information.

You can also specify VM arguments on the command line when you launch the platform. Refer to "Configuring the platform launcher" on page 236 for more information.

Configuring native library references

The recommended approach is that all native library objects be included in operating system/processor specific fragments. In general, this is sufficient to allow the application code and the operating system to locate the desired library. However, there might be cases where it is not possible to organize the libraries within a fragment, or the library loading requirements inhibit this approach. Therefore, library search path must be updated.

You will need to update the `library.path.append` or `library.path.prepend` lines in the `rcpinstall.properties` file to specify the directory locations containing the libraries.

Refer to "Updating the rcpinstall.properties file" on page 228 for detailed information.

Configuring Java Bootclasspath libraries

The recommended approach is that all class libraries that are needed by applications reside within plug-ins or fragments. If there are cases in which the libraries must be placed on the Java bootclasspath, then you will need to update the `–Xbootclasspath` line(s) in the `rcpinstall.properties` file.

Refer to "Updating the rcpinstall.properties file" on page 228 for detailed information.

Configuring workbench options

The default WebSphere Everyplace workbench provides configuration options that control the display of the Banner Bar, Cool Bar, and Status Bar in the user interface. You can modify the display options for each bar.

To enable or disable the display of these objects in the user interface, you will need to edit the `plugin_customization.ini` file, which resides in the `<installation_directory>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<version_no>` directory.

Use a text editor to open and edit this file. Change the values for `bannerVisibile`, `coolBarVisible`, and `statusLineVisibile` to true or false as appropriate.

Setting **bannerVisible=true** and **coolBarVisible=true** provides the capability to display these areas. The user will also have options to display or hide these areas. If these are set to false, they will not be displayed and the user will not be able to change the settings

`statusLineVisible` either hides or displays the status bar. If this is set to true, it will always be displayed, and the user will not be able to hide this area.

The other content in this file controls the appearance of the banner that is displayed in the Banner Bar. Refer to "Specifying platform branding" on page 231 for more information.

Configuring the web container

The default configuration for the web container listens only for HTTP requests received on localhost on a port dynamically selected during platform startup. If you need to make changes to this configuration, refer to the contents of this section.

The following properties are available for configuration of the web container.

Table 44. Java system properties

Option	**Default**	**Description**	**Java System Property**
HTTP Port	0	Defines the port used by the HTTP Service listener to listen for requests	com.ibm.pvc.webcontainer.port or com.ibm.osg.webcontainer.port
HTTPS Port	-1		com.ibm.pvc.webcontainer.port.secure or com.ibm.osg.webcontainer.port.secure
HTTP Timeout	60 sec	Defines the value used for socket time-outs	com.ibm.pvc.webcontainer.http.timeout or com.ibm.osg.webcontainer.http.timeout

Table 44. Java system properties (continued)

Option	Default	Description	Java System Property
HTTP Address	localhost	Defines the host address for the default ports that the Web Container listens on. If this property is defined then the Web Container will only listen for requests that come through this IP address. The special value ALL indicates all available IP addresses on the device will be used. The value of this property may be a resolved name or IP address (e.g. www.ibm.com, 192.168.0.101, localhost).	com.ibm.pvc.webcontainer.http.address or com.ibm.osg.webcontainer.http.address
HTTP redirect port	-1		com.ibm.pvc.webcontainer.http.redirectPort
Min HTTP Threads	4		com.ibm.pvc.webcontainer.http.minThreads
Max HTTP Threads	20		com.ibm.pvc.webcontainer.http.maxThreads
Max Keep Alive Connections	20		com.ibm.pvc.webcontainer.http.maxKeepAliveConnections
Max Keep Alive Requests	50		com.ibm.pvc.webcontainer.http.maxKeepAliveRequests
Keep Alive Timeout	20 sec		com.ibm.pvc.webcontainer.http.keepAliveTimout

Table 45. ConfigurationAdmin keys

Option	ConfigurationAdmin key
HTTP Port	http.port
HTTPS Port	https port
HTTP Timeout	http.timeout
HTTP Address	http.address
HTTP redirect port	http.redirectPort
Min HTTP Threads	http.minThreads
Max HTTP Threads	http.maxThreads
Max Keep Alive Connections	http.maxKeepAliveConnections
Max Keep Alive Requests	http.maxKeepAliveRequests
Keep Alive Timeout	http.keepAliveTimeout

Configuring the web container to use a dynamic port

If the value of either the `com.ibm.pvc.webcontainer` or `com.ibm.osg.webcontainer` Java system properties is equal to 0, then the web container selects a random port when the web container plug-in is started. This allows for multiple instances of the Web Container to be running at the same time on the same machine.

Note: This port will be broadcast to all plug-ins that register an `HttpSettingListener` service.

Configuring with system properties

For simple configuration changes, you can effect the changes by adding Java system properties to the platform. For each of the configuration properties identified in "Configuring the web container" on page 224, add the appropriate Java System Property to the `rcpinstall.properties` file. Refer to "Configuring Java system properties" on page 223 for information on adding Java System Properties.

Configuring with ConfigurationAdmin

The Web Container is configurable using the OSGi Configuration Admin service.

The PID of the Web Container ManagedServicefactory is `com.ibm.pvc.webcontainer`.

Using the Admin Utility for OSGi tool to access ConfigurationAdmin

Applications can configure the Web Container using the Configuration Admin service API. You can also configure these parameters using the Admin Utility tool. To configure the Web Container using the Admin Utility tool, perform the following steps:

1. Launch the WebSphere Everyplace Deployment platform.
2. Install the Admin Utility for OSGi feature. The Admin Utility for OSGi feature is located on the evaluation site.
3. Select **Application > Open > Admin Utility for OSGi** to run the application.
4. When the application is started, select the **Web Container** from the list of installed plug-ins. The Http Service Settings will display a list of configurable options.

   Users can configure the following Web Container settings:
   - HTTP Port
   - HTTPS Port
   - HTTP Timeout
   - HTTP Address
   - HTTP Redirect Port
   - Minimum HTTP Threads
   - Maximum HTTP Threads
   - Maximum Keep-Alive Connections
   - Maximum Keep-Alive Requests
   - Maximum Keep-Alive Timeout
5. If you want to use settings other than the default, enter the new value for the setting and click **Configure**.

Configuring HTTPS

The secure hypertext transfer protocol (HTTPS) is a communications protocol used to encrypt data between computers over the internet. HTTPS uses a Secure Socket Layer (SSL) to transfer encrypted HTTP data.

Configuring client plug-ins with the Enterprise Management Agent

You can change the configuration of a client plug-in that has registered itself as a Managed Service with the framework using the Enterprise Management Agent. A Managed Service represents a client of the Configuration Admin service. Plug-ins which have registered themselves as a Managed Service (e.g., Web Container) will receive configuration update notifications from the `ConfigurationAdmin` service.

The Enterprise Management Agent provides administrators with a means to discover these types of plug-ins and change their configuration.

Note:

- Registration as a Managed Service does not guarantee that a plug-in can be discovered by the Enterprise Management Agent. `ConfigurationAdmin` must be populated with the plug-ins configuration data. Refer to the OSGi specification for more details.
- Administrators choosing to configure the client plug-in using the Enterprise Management Agent must populate the `ConfigurationAdmin` with the plug-ins configuration data. This can be done either programmatically or by using a tool like Admin Utility for OSGi.

The following steps describe how to change the configuration of a client plug-in, making use of multiple jobs to perform the task. These tasks should be performed on a system running a Device Management Server using the DM Console application.

1. Click **Devices**.
2. Select **Use New Query and Return anything** as your search criteria, and then click **OK**. The DM console will show a list of enrolled devices.
3. Select your device, right click and select **Submit Job**.
4. Click Next, then select the **Job Type as Node Discovery** (use the default settings for all the other job attributes).
5. Select **Next**, then select **Add Group**.
6. Type `./OSGi/BundleConfiguration/<`*`Plugin_PID`*`>` as the Target URL.

   where <*`Plugin_PID`*> is the PID of the plug-in. For example, `com.ibm.pvc.webcontainer` is the PID of the Web Container
7. Enter a search depth of 5, then click **Next**.
8. Click **OK**. The job has been submitted. Click **Close**.
9. You will need to wait for the job to complete (this will depend upon the configured polling interval). Once the job has completed, select the device, then click **View Inventory**.
10. Then select **Management Tree**.
11. Select the configuration entry you wish to configure and click **Submit Job**.
12. Click **Next**. Leave the defaults as supplied in the panel, then click **Next**.
13. Enter 1 for the **Command Number Field**.
14. Set the **Data** value for the configuration entry, then click **Next**.
15. Click **OK**.

16. Click **Close**, then click **Close** again. and wait for the job to complete.

Configuring platform configuration files

Specifying other platform configuration properties

The following system properties can be specified as with any other Java system properties. These will control operation of the platform. Some of the properties are not currently set and can be specified if the platform behavior needs to be changed.

com.ibm.osg.service.deviceagent.nativeinstall.default

Defines the target update site used by the Enterprise Management Agent when installing updates created by the NativeAppBundle tool with a -Eclipse=default parameter. The first configured site that ends with this value will be the target installation site for the distributed features. This value is case sensitive, and must end with a slash.

Current Setting: `/apps/eclipse/`

Default Setting: <none>

provisioning.configFile
Defines the path to the provisioning config file that lists the features to install and the sites to install from.

provisioning.errorFile
Defines the file where provisioning errors will be written. The installer uses this file to log any errors that occur during provisioning. If no errors occurred, then this file is not created.

Updating the rcpinstall.properties file

This section describes how you can update one or more element of a user's WebSphere Everyplace Deployment client platform by modifying the `rcpinstall.properties` file. The installer creates and populates the `rcpinstall.properties` file. Users should not make changes to this file. Only ISVs and developers should make changes to the file, however, they should not change the installed values.

The `rcpinstall.properties` file resides in the same directory as the launcher, which is `<`*`installation directory`*`>/rcp`. This file must meet all requirements of a Java property file. Any non-ASCII characters must be escaped with \uxxxx. You can generate these values with your favorite Unicode 16 editor and converting with the Java program native2ascii. The `rcpinstall.properties` file can contain the following properties:

Note: Unless indicated otherwise, all properties should appear only once within the `rcpinstall.properties` file. If a property appears more than once, only the first occurrence of the property is used.

Table 46. rcpinstall.properties

Property	**Description**
vm=<JVM executable file>	REQUIRED. This must point to a JVM executable file. The preference is javaw .
rcp.install.id=<id>	REQUIRED. This is a unique id that the installer creates for each installation.
cp=<classpath>	REQUIRED. This must include <install_directory>/eclipse/startup.jar

Table 46. rcpinstall.properties (continued)

Property	**Description**
Xbootclasspath.prepend=<path>	Specifies the **-Xbootclasspath/p**:*path* See the documentation for the Java application launcher.
Xbootclasspath.append=<path>	Specifies the **-Xbootclasspath/a**:*path* See the documentation for the Java application launcher.
-D<prop>=<value>	See the documentation for the Java application launcher. Additional System properties may be added at the bottom of the file.
library.path.append=<path>	Modifies PATH (on Windows) or LD_LIBRARY_PATH (on Linux)
library.path.prepend=<path>	Modifies PATH (on Windows) or LD_LIBRARY_PATH (on Linux)
application=<application plugin id>	REQUIRED. This is equivalent to the eclipse runtime property `–data`. You can override this for the user and serial.multiuser configurations by using the `-data` argument on the command line.
install.configuration={serial.multiuser, service, user}	REQUIRED. Specifies whether the installation is a service, serial multi-user, or a user. After a platform has been installed, this value should not be changed.
vmarg.<name>=<value>	You can provide VM specific arguments here. The <value> is passed unaltered as a vmarg to the JVM. The <name> is only used as a unique identifier of the vmarg within this file. Syntax of the argument is not checked.
library.preload=<value>	REQUIRED for Linux. This will have been populated by the installer if needed.

An example of a typical `rcpinstall.properties` file might look like the following:

```
#Fri Jun 03 10:22:45 CDT 2005
install.configuration=user
vm=C\:/Program Files/IBM/WEDMGMT/rcp/eclipse/plugins/com.ibm.rcp.j2se.win32.
    x86_1.4.2.SR1/jre/bin/javaw.exe
library.path.append=;C\:/Program Files/IBM/WEDMGMT/rcp/eclipse/plugins/com.
    ibm.tivoli.agentext.win32.x86_1.8.0.20050601/InventoryScanner;
    C\:/Program Files/IBM/WEDMGMT/rcp/eclipse/plugins/com.ibm.db2e.win32.
    x86_8.2.1-20050601/os/win32/x86;C\:/Program Files/IBM/WEDMGMT/rcp/
    eclipse/plugins/com.ibm.mobileservices.isync.win32.x86_8.2.1-20050601/
    os/win32/x86;C\:/Program Files/IBM/WEDMGMT/rcp/eclipse/plugins/
    com.ibm.tivoli.agentext.win32.x86_1.8.0.20050601/InventoryScanner
-Dprovisioning.errorFile=C\:/Program Files/IBM/WEDMGMT/rcp/deploy/error.log
rcp.installId=1117812134309
Xbootclasspath.append=C\:/Program Files/IBM/WEDMGMT/rcp/loggerboot.jar
rcp.install.id=1117812134309
vmarg.com.ibm.jre.Xj9=-Xj9
-Dprovisioning.configFile=C\:/Program Files/IBM/WEDMGMT/rcp/deploy/
     install.propertiesapplication=com.ibm.eswe.workbench.WctWorkbenchApplication
cp=C\:/Program Files/IBM/WEDMGMT/eclipse/startup.jar
```

```
-Djava.util.logging.config.class=com.ibm.rcp.core.logger.boot.LoggerConfig
-Dcom.ibm.osg.service.osgiagent.logfileloc=C\:/Program Files/IBM/WEDMGMT/rcp
-Dcom.ibm.osg.service.deviceagent.nativeinstall.default=C\:/Program Files/IBM/
    WEDMGMT/shared/eclipse
```

Updating the config.ini file

The `config.ini` file is located in the <*installation directory*>`/eclipse/configuration` directory. The installer provides information and values for this file that the platform needs. The remaining configuration files are located in the `.config` directories. You can add a `config.ini` file to the `.config` directory. Values in the `.config/config.ini` file will override the values in the global `eclipse/configuration/config.ini` file.

The location of the `.config` directory is dependent upon the installation configuration you chose:

If the installation configuration is **service**, then the location of the `.config`directory is: <*installation directory*>`/eclipse/.config/config.ini.`

If the installation configuration is **serial multiuser**, then the location is: <*installation directory*>`/eclipse/.config/config.ini.`

If the installation configuration is **user**, then the location is: `<user.home>/IBM/RCP/<`*rcp.install.id*`>/<user.name>/.config/config.ini.`

<user.home> and <user.name> are Java system properties.

<rcp.install.id> is from the rcpinstall.properties file.

The following is a list of the properties found in the `config.ini` file:

osgi.framework.extensions
: Extensions to the base Eclipse framework.

  Current Setting: =com.ibm.jxesupport

  Default setting: <none>

osgi.parentClassloader
: The definition of the parent class loader for OSGi bundles.

  Current Setting: ext

  Default setting: boot

osgi.splashPath
: The comma separated list of URLs to search for the file named `splash.bmp`.

  Current Setting:

  ```
  platform:/base/../rcp/eclipse/plugins/com.ibm.pvc.wct.platform,
  platfomr:/base/../rcp/eclipse/plugins/com.ibm.pvc.wct.platfomr.nl1,
  platfomr:/base/../rcp/eclipse/plugins/com.ibm.pvc.wct.platfomr.nl2
  ```

  Default Setting: <none>

osgi.frameworkClassPath
: A comma separated list of class path entries that define the OSGi framework implementation.

  Current Setting: `core.jar, console.jar, osgi.jar, resolver.jar, defaultAdaptor.jar, eclipseAdaptor.jar.`

Default Setting: <none>

osgi.bundles

A comma separated list of bundles that will be installed and optionally started once the system is up and running. For more information on the syntax for this property, refer to the Platform Plug-in Developer's Guide.

Current Setting:

```
org.eclipse.core.runtime@2:start,
org.eclipse.update.configurator@3:start
com.ibm.pvc.wct.platform.autostart@3:start
```

Default Setting: <none>

osgi.bundles.defaultStartLevel

Assigns the default start level to any bundles that do not explicitly have a start level assigned.

Current Setting: 4

Default Setting: 4

eclipse.exitOnError

Indicates whether the workbench should exit immediately if it receives a Framework ERROR event.

Current Setting: false

Default Setting: true

eclipse.product

Sets the identifier of the product being run, which identifies the branding information associated with the workbench.

Current Setting: `com.ibm.pvc.wct.platform.WctWorkbenchProduct`

Default Setting: <none>

java.protocol.handler.pkgs

Java System property that specifies classes that will be used to handle different URL types.

Current Setting: com.ibm.net.ssl.www.protocol

Default Setting: <not set>

Specifying platform branding

You can modify the user interface of the client workbench to include your own branding. Use the instructions in this section to modify such elements as the title bar, splash screen, icons and images, and the About dialog.

Note: In this section we indicate the changes that need to be made relative to the installed `com.ibm.pvc.wct.platform` plug-in and its files. Refer to "Distributing branding updates" on page 235 for additional considerations on how to make these changes.

The following diagram depicts the location of various objects that you can modify when specifying platform branding.

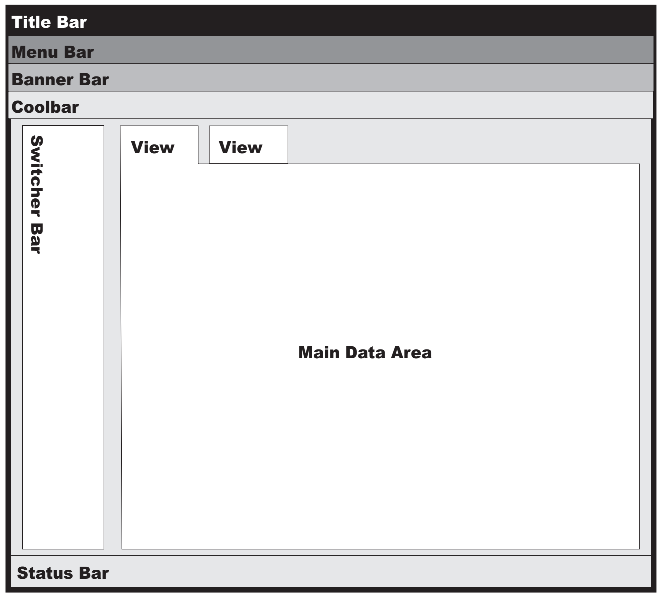


Figure 11. platform branding options

Changing the title bar

The title bar appears at the top of the client platform and usually contains the name of the workbench, and a small graphic.

To modify the product title bar, you modify the name attribute of the product extension in the `plugin.properties` file. The `plugin.properties` file resides in the following directory:

`<installation directory>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<version_no>`

The following is an example of the product extension in the `plugin.properties` file. To rename the title bar, you can specify the new name in the `product name` attribute:

```
<extension
 id="WctWorkbenchProduct"
 point="org.eclipse.core.runtime.products">
 <product
  name="%product.name"
        application ...
```

Changing the splash screen

When a user launches the workbench, a splash screen image is displayed. You can replace the splash screen image with your own image. The splash screen image is a file called `splash.bmp`. The splash screen must have the file extension .bmp. There are no constraints regarding the size of the image, but for reference, the standard Eclipse splash screen image is 500 x 300 pixels.

You can have a different splash screen for each locale that the product supports. When the application starts, the launcher determines the locale of the machine, and then selects the splash screen image from the appropriate language directory in the `plugins` directory. For example, a splash screen for the French locale would reside in the `nl/fr` directory.

To replace the splash screen image, replace the `spash.bmp` file that resides in the <*installation directory*>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform.nl1_<*version_no*> /nl//<*locale*> directory. If the launcher does not find a splash screen image for your locale, then the launcher selects the default image from the <*installation directory*>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<*version_no*> directory.

Changing the product images

On Windows systems, a small image is associated with the product and is displayed in the title bar, next to the product title. You can modify this image to be consistent with your branding.

Standard Widget Tools (SWT) allows a set of images to be associated with a shell with the expectation that all the images in the set will have the same appearance but be rendered at different sizes. These images are provided to the SWT shell, which is then able to select the most appropriate one for each specific use. For example, the smaller image (16 X 16) is used for the title and task bars while the larger image (32 X 32) is used in the Alt-Tab application switcher.

To modify the images, replace images **16.gif**, **32.gif** and **48.gif** files, which reside in the <*installation directory*>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform.nl1_<*version_no*> directory.

Changing the About dialog

To specify branding of the About dialog, you must replace the image shown in the About Dialog box, and also the text that displays next to the image.

Changing the About dialog image:

The image shown in the About dialog is supplied by a file called `about.bmp`. You can replace this file with the image you want to use, but it must be a file with a **.bmp** extension.

You can have a different About dialog images for each locale that the product supports. When a user opens the About dialog, the system detects the locale of the machine, and then selects the About dialog image from the appropriate language directory in the `plugins` directory. For example, a splash screen for the French locale would reside in the `nl/fr` directory.

To replace the About dialog image, replace the `about.bmp` file in the <*installation directory*>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform.nl1_<version_no>/ nl//<*locale*> directory. If the system does not find a About dialog image for your locale, then the system selects the default image from the <*installation directory*>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<*version_no*> directory.

Changing the About dialog text:

The text that is displayed next to the image in the About dialog is contained in the `plugin.properties` file. The `plugin.properties` file is locale-specific. For each locale, you can modify the `productAboutText` property in the `plugin.properties` file, which resides in the <*installation directory*>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<*version_no*>/ nl/<*locale*> directory. If there is no `plugin.properties` file for your locale, then you can modify the default `plugin.properties` file, which resides in the `<`*`installation directory`*`>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<`*`version_no`*`>` directory.

Changing the background image

The background image is the image that displays in the main data area of the workbench when no applications are opened. You can modify the background to display your own image.

To display your own image, replace the `default_background.gif` file in the `<`*`installation directory`*`>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<`*`version_no`*`>/nl/<`*`locale`*`>` directory. If there is no `default_background.gif` file for your locale, then you can modify the `default_background.gif` file, which resides in the `<`*`installation directory`*`>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<`*`version_no`*`>` directory.

You can also change the image path and file name by modifying the `defaultBackgroundImage` property in the `plugin_customization.ini` file, which resides in the `<`*`installation directory`*`>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<`*`version_no`*`>` directory.

To point to an image in the `nl` directory, you can set the value of `defaultBackgroundImage` to `$nl$/new_default_background.gif`.

Changing the Switcher Bar

You can change the image that is displayed in the switcher bar to an image of your choice. The image that you provide will need to be tiled to fill in the full area of the switcher bar.

To display your own image, replace the `switcherbar_background.gif` file in the `<`*`installation directory`*`>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<`*`version_no`*`>/nl/<`*`locale`*`>` directory. If there is no `switcherbar_background.gif` file for your locale, then you can modify the `switcherbar_background.gif` file, which resides in the`<`*`installation directory`*`>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<`*`version_no`*`>` directory.

You can also change the image path and file name by modifying the `switcherBarImage` property in the `plugin_customization.ini` file, which resides in the `<`*`installation directory`*`>/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_<`*`version_no`*`>` directory.

To point to an image in the **nl** directory, you can set the value of `switcherBarImage` to `$nl$/new_switcherbar_background.gif`.

Changing the Banner bar

The banner area resides at the top of the workbench window, directly below the menu bar. You can customize all of the visual elements of the banner to suit your needs. You can change the default height of the banner, or hide the banner in environments where screen real estate are a concern.

You modify the banner area by setting properties in the `plugin_customization.ini` file, which resides in the *<installation directory>*`/rcp/eclipse/plugins/com.ibm.pvc.wct.platform_`*<version_no>* directory.

The banner area consists of an image on the right, an image on the left, and a tiled background image. Switching between applications will change the name and images displayed in the banner area. The system derives the application name to use from the `WctWebApplication` and `WctApplication` extension points within each application.

The following table lists all of the properties that you can configure for your banner in the `plugin_customization.ini` file:

Table 47. banner properties

Preference	Type	Default	Description
bannerVisible	Boolean	true	Determines whether or not the banner is visible.
bannerHeight	Integer	60	Determines the height of the banner area in pixels
bannerBackgroundColor	n:n:n where 0 <=n <=255	0:0:0	RGB value for the banner background. Visible if no bannerTileImage is specified.
bannerTileImage	string		Filename of the horizontally tiled background image (for example a texture).
bannerRightImage	string		Filename of the right-aligned image on the banner.
bannerLeftImage	string		Filename of the left-aligned image on the banner.
applicationTitleVisible	Boolean	true	Determines whether the application title is displayed.
applicationTileXpos	integer	61	x offset in pixels for the page title (measured from the left).
applicationTitleYpos	n | centered where n=0	5	y offset in pixels for the page title (measured from the left)
applicationTitleColor	n:n:n where 0 <=n <=255	255:255:255	RGB value for the page title text.
applicationTitleFontSize	N	14	Size of the page title text in points.

Distributing branding updates

While each of the changes discussed here can be made to the actual files located on disk, you will most likely want to distribute these changes to all clients that you are responsible for. If you want to make branding changes, it is recommended that you make the changes prior to distributing the client. The recommended approach for making these changes is the following:

- Make a copy of the `com.ibm.pvc.wct.platform` plug-in, and any language fragments of that plug-in that you require, creating a new symbolic name for the plug-in (and updating the host plug-in for each of the fragments)
- Follow the instructions above in order to make any branding changes that you require
- Create a feature to include your new plug-in
- Update the `rcpinstall.properties` file to remove the `com.ibm.pvc.wct.platform` feature and replace it with your new feature.
- Provide access to the updated distribution site for your users.

Configuring the platform launcher

The options that you might want to configure are:

```
-console
-debug
-consoleLog
-application
-vmargs
-configuration
-data
-nosplash
```

`-console` enables your to debug and display the OSGi console. You can add `logredirector.level=INFO` to the `rcpinstall.properties` file. These two options work well together when you need to debug problems.

`-consolelog` will cause every console message to be duplicated. It is better to use the -console option instead.

`-application` temporarily overrides the `rcpinstall.properties` file.

`-configuration` temporarily override the `rcpinstall.properties` file.

Arguments based on Java VM, Eclipse, or OSGi are passed through, except for the following:

- The `-vm` property is required in the `rcpinstall.properties` file. When the -vm argument is passed to the `rcplauncher.exe` the argument is stripped.
- If either the `-console` or `-consoleLog` argument is specified, then `javaw.exe` specified in the `rcpinstall.properties` file will be converted to `java.exe` by `<launcher>.exe`.
- If the platform is running as a service the `-nosplash` option is always active.

For a complete list of runtime options, refer to the Platform Plug-in Developer's Guide, installed with the Rational Software Development Platform.

To configure the platform launcher to use additional arguments, you must add the argument to the path of the executable. To add arguments to the executable, you modify the shortcut from the desktop icon, or if the WebSphere Everyplace Deployment client is installed as a service, you modify the service's image path from the registry.

The following example modifies the WebSphere Everyplace Deployment Client that is running as a Windows service to display a console during launch. When you follow this example, you enable additional debugging information and provide access to the OSGi console. At the osgi> prompt, type "`help`" to see a list of possible commands:

1. From the **Start** menu select **Run**.
2. In the Run dialog, type `regedit`. The registry is displayed.
3. Select **HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/WEDMgmntService**.
4. Double click on the entry named **ImagePath**. The Edit String dialog is displayed.
5. Type `-console` at the end of the path and click **OK**.
6. Open the Services Control Panel.
7. Open the properties for WebSphere Everyplace Deployment Management Service.
8. Click the **Log on** tab and select the **Allow service to interact with desktop** option.
9. Start the service.

For Linux, you can get access to the OSGi console and additional debugging messages using the following steps:

1. Stop the service if it is running.
2. From a command line interface, type `mgmtservice -debug`.
3. When finished, type exit at the osgi> prompt and the service will terminate.

OSGi

OSGi specification

One activity of the OSGi Alliance was to define a Java framework that:

- Enables multiple applications to coexist within a single JVM
- Manages the life cycle of components within the framework
- Specifies a set of required and optional services on the platform

WebSphere Everyplace Deployment is built on the Eclipse Rich Client Platform, which includes an OSGi framework. The framework is based upon the OSGi™ Service Platform Release 3 specification with additional extensions provided by the Eclipse 3.0.x implementation of the OSGi framework. Application developers partition applications into services and other resources. Services and resources are packaged into bundles, which are files that serve as the delivery unit for applications. Bundles have manifests with special headers that enable you to share classes and services at the package level. Within the Eclipse based platforms, all plug-ins are OSGi bundles, so you can think of the terms plug-in and bundle as being interchangeable.

Working with OSGi bundles

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

Creating OSGi bundles

This section describes how to create an OSGi bundle. For more detailed information about writing bundles, refer to the OSGi Service Platform Release 3

Bundles: A bundle is the smallest unit of management for the Framework. Bundles are Java Archive (JAR) files with a manifest that contains special headers. These headers describe the bundle to the OSGi framework and list the bundle's dependencies, such as the packages and services required by the bundle. Bundles can register services with the OSGi framework that other bundles can use.

The descriptive information in the manifest file differentiates bundles from other JAR files. Non-bundle JAR files often keep very little information in the manifest file. However, a bundle's manifest file usually contains descriptive information, such as the bundle's name and version, and a list of the packages and services it requires.

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

- `INSTALLED` - the bundle has been installed, but all of the bundle's dependencies have not been met. The bundle requires packages that have not been exported by any currently installed bundle.
- `RESOLVED` - the bundle is installed, and its dependencies have been met, but it is not running. If a bundle is started and all of the bundle's dependencies are met, the bundle skips this state.
- `STARTING` - a temporary state that the bundle goes through while the bundle is starting.
- `ACTIVE` - the bundle is running.
- `STOPPING` - a temporary state that the bundle goes through while the bundle is stopping.
- `UNINSTALLED` - the bundle no longer exists in the framework.

Conventions for creating bundles: When you create bundles, use the following conventions:

- Clean up objects and threads properly during your stop method. The framework does not terminate lingering threads.
- Return promptly from `BundleActivator` `start()` and `stop()` methods. These methods are invoked synchronously by the framework. Delays in returning from these methods will affect the ability of the framework to process other bundle actions. It is recommended that substantial activities be handed off to another thread for processing, or be delayed until first service invocation.
- Return promptly from Framework and Bundle Listeners events. These event methods are invoked by the framework. Delays in returning from these methods may adversely affect performance of the framework.
- Allow for service life cycle events. The OSGi framework provides the ability to dynamically install and remove bundles. As a result, it is possible in some frameworks that services may not always be present. A service is only present when the bundle that registered the service is available. See "Getting and un-getting services from the OSGi Framework" on page 243 for conventions to solve this problem.

Creating manifest files: Each bundle must contain either a `plugin.xml` or a manifest file. The bundle's manifest file contains data that the framework needs to correctly install and activate the bundle.

Note: A `plugin.xml` may contain similar information, however, a `plugin.xml` also contains extensions and extension points

If a bundle contains only a `plugin.xml`, the Eclipse platform will generate a `MANIFEST.MF` equivalent when the platform starts. When you specify data in a manifest file, you must use the headers that were defined by the OSGi™ specification. You can use user-defined headers; however, the framework ignores any headers that it does not understand. Refer to the OSGi Service Platform Release 3 specification for more information about the OSGi Manifest file format and syntax.

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

The following headers are defined in the OSGi Service Release 3 specification and by the Eclipse 3.0.x extensions to the OSGi framework.

- Import-Package

  Use this header to specify the names of any package that you want your bundle to import from the runtime. If you do not specify the package your bundle needs in this header, you may get a `NoClassDefFound` exception when the bundle loads.

  For information on how to edit this header, refer to "Bundle Resources page" on page 245.

  Note: You must also specify the package you want to import (using Import-Package) in the Export-Package header of the bundle that contains the package.
- Export-Package

  Use this header to specify the name of any package that you want your bundle to export to the runtime. If you do not specify the packages needed by other bundles in this header, the dependent bundles may not resolve.

  For information on how to edit this header, refer to "Bundle Resources page" on page 245.
- Require-Bundle

  Use this header to specify the specific bundles that provides packages you use in your bundle. If you do not specify the bundle which provides the packages you need, you may get a `NoClassDefFound` exception when the bundle loads.

  For information on how to edit this header, refer to "Bundle Resources page" on page 245.

  Note: You must also specify the packages you want to access from your bundle in the Provide-Package header of the bundle that contains the package.
- Provide-Package

  Use this header to specify the names of any package that you want to provide to other bundles. If you do not specify the packages needed by other bundles in this header, the dependent bundles may not resolve.

  For information on how to edit this header, refer to "Bundle Resources page" on page 245
- Bundle-Activator

  Use this header to specify the fully-qualified name of the BundleActivator class.

  A bundle designates a special class to act as a Bundle Activator. The Framework must instantiate this class and invoke the start and stop methods to start or stop the bundle as needed. The bundle's implementation of the `BundleActivator`

Interface enables the bundle to initialize a task, such as registering services, when the bundle starts and to perform clean-up operations when the bundle stops.

The `org.eclipse.core.runtime.Plugin` class implements the `org.osgi.framework.BundleActivator` interface. When creating Client Services projects, a subclass of `Plugin` will be created and will become the `BundleActivator` for the plug-in.

You may define your own class to implement the `org.osgi.framework.BundleActivator` interface.

You can specify this header in the Class field on the Overview Page of the Bundle Manifest Editor. For more information, refer to "Overview page" on page 244.

- Bundle-SymbolicName

  The `Bundle-SymbolicName` manifest header can be used to identify a bundle. The Bundle Symbolic Name and Bundle Version allow for a bundle to be uniquely identified in the Framework. It does not replace the need for a Bundle-Name manifest header, which provides a human readable name for a bundle.

  You can specify this header in the ID field on the Overview Page of the Bundle Manifest Editor. For more information, refer to "Overview page" on page 244.

Refer to the OSGi Service Platform Release 3 for descriptions of other bundle headers, such as the following, which provide bundle description information:
- Bundle-Name
- Bundle-Description
- Bundle-Copyright
- Bundle-Vendor
- Bundle-Version
- Bundle-DocUrl
- Bundle-ContactAddress
- Bundle-Fragment

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

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

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

Understanding services

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

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

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

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

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

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

In the following example, three bundles are created:
- InterfaceBundle
- ServiceImplBundle
- ServiceConsumerBundle

Interface Bundle

The `InterfaceBundle` exports the `com.ibm.osg.example.mtservice` package that contains the `com.ibm.osg.example.mtservice.MyTestService` interface. The `InterfaceBundle` adds an Export-Package: `com.ibm.osg.example.mtservice` to its `MANIFEST.MF` file. Since this bundle has no initialization or startup needs, no `BundleActivator` is required for this bundle. This interface defines a service than can print a message:

```
package com.ibm.osg.example.mtservice;
public interface MyTestService {
     // One method is provided by the service.
     // This method will simply print
     // the message to standard out.
     public void printMessage(String message);
}
```

Service Implementation Bundle

The `ServiceImplBundle` provides an implementation of the `MyTestService` (Other bundles could provide alternative implementations). The `ServiceImplBundle` exports no packages, but does contain an Import-Package: com.ibm.osg.example.mtservice in its MANIFEST.MF file so that it can have access to the MyTestService interface.

The following class provides the implementation for our service. In the following example, a service called com.ibm.osg.example.mtservice.MyTestService registers with the framework. This implementation of the service prints the message to the standard output. Generally, packages containing service implementation classes should not be exported to other bundles.

```
package com.ibm.osg.example.mytestservice;
public class MyTestService implements com.ibm.osg.example.mtservice.MyTestService{
     public void printMessage(String message){
          System.out.println("MyTestService - " + message);
     }
}
```

The following BundleActivator class registers the com.ibm.osg.example.mtservice.MyTestService service with the framework.

```
package com.ibm.osg.example.mytestservice;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

public class MyBundleActivator implements BundleActivator {

    ServiceRegistration registration;

    /*Create a new instance of the TestService
    and then use the BundleContext object to
    register it.
Store the registration object
    to use to unregister the service when the
    bundle is
stopped by the framework.
   */
     public void start(BundleContext context)
     {
       MyTestService testservice = new MyTestService();
       if( registration == null ){
                registration =
    context.registerService(
                   "com.ibm.osg.example.mtservice.MyTestService",
                   testservice,
                   null);
           }
 }
     public void stop(BundleContext context) {
  if ( registration != null ){
              registration.unregister();
          }
          registration=null;
     }
}
```

The ServiceConsumer bundle, like the ServiceImplBundle, must contain an Import-Package: com.ibm.osg.example.mtservice in its MANIFEST.MF file. It may optionally contain an Import-Service: com.ibm.osg.example.mtservice.MyTestService. This is recommended as the tools will use this to ensure the framework has the proper prerequisites, but this is not required. See the section “Getting and un-getting services from the OSGi Framework” on page 243 for an example of the ServiceConsumer bundle.

Getting and un-getting services from the OSGi Framework: Bundles register and unregister services. Bundles that depend on services must account for the possibility that the requested service might not be available. The service can register or unregister with the framework at any time. You can use a `ServiceTracker` to enable your bundles to query or listen for service registrations and to react accordingly.

```
package com.ibm.osg.example.mygetservice;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.util.tracker.ServiceTracker;
import com.ibm.osg.example.mtservice.MyTestService;

public class MyBundleActivator
       implements BundleActivator, Runnable
{
     private boolean done=false;
    private ServiceTracker testServiceTracker;

      // Bundle Activator Start Method
      public void start(BundleContext context)
      {
         /* Here we initialize and open our ServiceTracker.
            It will track any service registering under
            the "com.ibm.osg.example.mtservice.MyTestService"
            interface.
         */

      testServiceTracker =
          new ServiceTracker(context,
                        "com.ibm.osg.example.mtservice.MyTestService",
                        null);
          testServiceTracker.open();

       // Here we start a thread that will continue
       // to use our service until
       // the bundle is stopped.

       Thread t = new Thread(this);
          t.setName("mygetservice thread");
          t.start();

     }
    /*Bundle Activator Stop Method -- here we stop
    the thread and close the
    ServiceTracker*/

    public void stop(BundleContext context)
    {
           done=true;
           testServiceTracker.close();
      }
    //Here is a method that uses the service
    //we are tracking.  First we get
    //the service
    //from the tracker, then we call its printMessage
    //method.

      public void useService(String message){
           MyTestService testService = (MyTestService)
                         testServiceTracker.getService();

           if( testService != null )
     {
                // If the service is available then use it.
                testService.printMessage(message);
   }
           else{
                // If the service is not available then perform an acceptable action.
                // Here we just print the message to standard out and indicate the service
                // was not available.
                System.out.println("No MyTestService available - " + message);
           }
      }

    // Simply continues to use the test service
    // every second until the done flag is set.
  public void run(){
           int i = 0;
   done = false;
   while (!done) {
    useService("message from test " + i++);
                try{
                     Thread.sleep(1000);
                }
                 catch( InterruptedException ie ){
                }
           }
      }
}
```

For an example that uses `ServiceTrackers` and getting services, refer to the Service Tracker example in the **Samples Gallery > Technology Samples > WebSphere Everyplace Deployment > OSGi** section.

IBM WebSphere Everyplace Client toolkit

This section provides reference information for using the IBM WebSphere Everyplace Client toolkit.

Editors

The WebSphere Everyplace Client Toolkit provides the following editors:

Bundle Manifest editor

The Bundle Manifest editor enables you to define information in the `plugin.xml` and/or `MANIFEST.MF` file.

Bundle manifest editor extends the PDE plug-in manifest editor. For more information on PDE plug-in manifest editor, refer to the *PDE Guide*.

For Client Services projects, the Bundle Manifest Editor is the default editor for `plugin.xml` and `META-INF/MANIFEST.MF` files. The editor is opened when either of these files are opened. If you find that the bundle specific information is missing, close the editor, and use the **Open with** operation to select the Bundle Manifest Editor.

Overview page: Bundle manifest editor's Overview page includes all sections of the Overview page of plug-in manifest editor, and an additional Bundle Manifest Information section. The following table lists the information you can enter into this section:

Table 48. Bundle Manifest Information headers

Manifest header	**Description**
Bundle-Category	Enables bundles to be listed in different categories.

Table 48. Bundle Manifest Information headers (continued)

Manifest header	**Description**
Bundle-ContactAddress	Represents the e-mail address of a person that a user can contact regarding the bundle.
Bundle-Copyright	Represents the copyright year for the bundle.
Bundle-Description	Specifies the purpose or function of the bundle.
Bundle-DocURL	Specifies a URL that contains additional information about the bundle.
Bundle-UpdateLocation	Specifies the location where an updated version of the bundle resides.

Bundle Resources page: You can specify packages and services to import and export. You can also specify the secondary dependencies this bundle might use. Refer to "Packages" on page 240 for more information about packages. For more information about secondary dependencies, refer to "Secondary Dependencies" on page 174.

Table 49. Bundle Resources

Bundle Resource Section	**Description**
Export Packages	Describes packages from the bundle that other bundles can use.
Export Services	Describes services that other bundles can use.
Import Packages	Describes the packages that the bundle requires. Another bundle must export these packages.
Import Services	Describes the services the bundle can use.
Secondary Dependencies	Describes the list of plug-ins this bundle might use.

The Bundle Resource page contains the following buttons:

- The **Add** button opens a dialog that lists the available choices to add for each section.
- The **Remove** button removes any item that is selected in the box next to them.
- The **Set Version** button opens a dialog that will let you specify the version for the import and export packages.
- The Compute button will compute any dependencies in your project's code that are not already included in the `MANIFEST.MF`. This is meant to be used if you do not enable **Search for dependencies automatically** upon resource changes in the Client Services Project Options page.
- The Up and Down buttons move bundles up and down in the list. Since Secondary Dependencies are resolved upon the first matching bundle, the order can be important if two bundles contain the same packages.

Wizards

The WebSphere Everyplace Client Toolkit provides a variety of wizards:

New Client Services Project Wizard

Use this wizard to create a new Client Services project. This wizard can be accessed as follows:

1. Select **File > New > Project**. The new project wizard displays.
2. Expand the **Client Services** folder and select **Client Services Project**.

Refer to the following tables for a description of the options and their default values.

Table 50. Client Services Project page

Option	**Description**	**Default value**
Project name	Enter a name for your new Client Services Project.	None
Project Contents	You may de-select ″Use default″ and click **Browse** to select a file system location for your new Client Services Project.	The ″Use default″ option creates the project in your current workspace.
Create a Java project	Select this if the project will contain Java code. Deselect this if the project will only contain non-Java resources.	Selected to create a Java project.
Source Folder Name	Folder name for Java source files.	src
Output Folder Name	Folder name for Java class files.	bin

This is followed by the Client Services Content page:

Table 51. Client Services Content page

Option	**Description**	**Default Value**
Bundle ID	This is a unique bundle symbolic name. It is suggested that you update this from the default value. The bundle name should be a unique URI, following the Java package naming conventions.	The project name is used as the default value.
Bundle Version	The bundle version. The version is in the form of major, minor, and micro numbers, separated by '.'.	1.0.0
Bundle Name	A descriptive bundle name.	The default name is constructed by appending "Bundle" to the project name.
Bundle Provider	A description of the bundle provider.	None
Runtime Library	The name of the JAR file in which the project's built contents will be placed.	The project name is used as the JAR file base name.
Generate the Java class that controls the bundle's life cycle	Selecting this will generate a bundle activator for the project. You will probably want to override the default class name of the bundle activator class.	Selected

Table 51. Client Services Content page (continued)

Option	Description	Default Value
This bundle will contribute to the Rich Client Platform	Select this option if you intend for the bundle to use Eclipse extension points to contribute to the Rich Client Platform. Selecting this option has the following affects: • If selected, the generated bundle activator class will have additional support for the Rich Client Platform. • It affects the default preference for automatically managing manifest package dependencies, on the Client Services Project Options page. If Rich Client Platform is selected, then the Require-Bundle preference is set, otherwise the Import-Package preference is set.	Selected

This is followed by the Platform Profile page:

Table 52. Platform profile options

Option	Description	Default value
Platform Profile	Select from the list the Platform Profile this Client Services Project will target. You can change your selection later in the Client Services property page.	WebSphere Everyplace Deployment (6.0.0) Default
Application Services	Check the Application Services that your Client Services Project will require. You can change your selection later in the Client Services property page. Grey entries are required by the Platform Profile and cannot be un-checked.	The ″Core OSGi Interfaces″ Application Service is required by all Platform Profiles.

This is followed by the Client Services Project Options page:

Table 53. Client Services Project Options

Option	Description	Default value
Search for dependencies automatically upon resource changes	Select this option to enable the tools to search for package dependencies whenever the user modifies source files. When this option is deselected, the tooling will not search for any unresolved or unused dependencies in your project.	Selected
Attempt to automatically resolve Manifest dependencies	Select this option to enable the tools to automatically manage the package dependency information in the manifest file. Package dependencies in your project's Java code will automatically be reflected through proper updates to the manifest file. When this option is deselected, package dependencies that are not properly reflected in the manifest are flagged with problem markers, along with quick fixes to resolve the problems.	Selected

Table 53. Client Services Project Options (continued)

Option	Description	Default value
Give preference to Require-Bundle	Require-Bundle will be used to automatically resolve a package dependency in cases where either Require-Bundle or Import-Package can be used.	Selected by default for projects that are contributing to the Rich Client Platform.
Give preference to Import-Package	Import-Package will be used to automatically resolve a package dependency in cases where either Require-Bundle or Import-Package can be used.	Selected by default for projects that are not contributing to the Rich Client Platform.

New Client Services Fragment Project Wizard

Use this wizard to create a new Client Services fragment project. This wizard can be accessed as follows:

1. Select **File > New > Project**. The new project wizard displays.
2. Expand the **Client Services** folder and select **Client Services Fragment Project**.

Refer to the following tables for a description of the options and their default values.

Table 54. Client Services Fragment Project page

Option	Description	Default value
Project name	Enter a name for your new Client Services fragment project.	None
Project Contents	You may deselect ″Use default″ and click **Browse** to select a file system location for your new Client Services Fragment Project.	The ″Use default″ option creates the project in your current workspace.
Create a Java project	Select this if the project will contain Java code. Deselect this if the project will only contain non-Java resources.	Selected to create a Java project.
Source Folder Name	Folder name for Java source files.	src
Output Folder Name	Folder name for Java class files.	bin

This is followed by the Client Services Fragment Content page:

Table 55. Client Services Fragment Content page

Option	Description	Default Value
Fragment ID	This is a unique bundle symbolic name. It is suggested that you update this from the default value. The bundle name should be a unique URI, following the Java package naming conventions.	The project name is used as the default value.

Table 55. Client Services Fragment Content page (continued)

Option	Description	Default Value
Fragment Version	The fragment version. The version is in the form of major, minor, and micro numbers, separated by '.'.	1.0.0
Fragment Name	A descriptive bundle name.	The default name is constructed by appending "Fragment" to the project name.
Fragment Provider	A description of the fragment provider.	None
Runtime Library	The name of the JAR file in which the project's built contents will be placed.	The project name is used as the JAR file based name.
Parent Bundle	Specify the ID, version, and optional version match rule for the parent bundle that this fragment contributes to. You may use the Browse button to select a parent from a dialog of other bundles in the target platform.	None
This bundle will contribute to the Rich Client Platform	Select this option if you intend for the fragment to use Eclipse extension points to contribute to the Rich Client Platform. Selecting this option affects the default preference for automatically managing manifest package dependencies, on the Client Services Project Options page. If Rich Client Platform is selected, then the Require-Bundle preference is set, otherwise the Import-Package preference is set.	Selected

This is followed by the Platform Profile page:

Table 56. Platform profile options

Option	Description	Default value
Platform Profile	Select from the list the Platform Profile this Client Services Project will target. You can change your selection later in the Client Services property page.	No default value
Application Services	Check the Application Services that your Client Services Project will require. You can change your selection later in the Client Services property page. Grey entries are required by the Platform Profile and cannot be unchecked.	The "Core OSGi Interfaces" Application Service is required by all Platform Profiles.

This is followed by the Client Services Project Options page:

Table 57. Client Services Project Options

Option	Description	Default value
Search for dependencies automatically upon resource changes	Select this option to enable the tools to search for package dependencies whenever the user modifies source files. When this option is deselected, the tooling will not search for any unresolved or unused dependencies in your project.	Selected
Attempt to automatically resolve Manifest dependencies	Select this option to enable the tools to automatically manage the package dependency information in the manifest file. Package dependencies in your project's Java code will automatically be reflected through proper updates to the manifest file. When this option is deselected, package dependencies that are not properly reflected in the manifest are flagged with problem markers, along with quick fixes to resolve the problems.	Selected
Give preference to Require-Bundle	Require-Bundle will be used to automatically resolve a package dependency in cases where either Require-Bundle or Import-Package can be used.	Selected by default for projects that are contributing to the Rich Client Platform.
Give preference to Import-Package	Import-Package will be used to automatically resolve a package dependency in cases where either Require-Bundle or Import-Package can be used.	Selected by default for projects that are not contributing to the Rich Client Platform.

Convert Project to Client Services Project Wizard

Use this wizard to convert a Java or Plug-in project to a Client Services project. This wizard can be accessed as follows:

1. Select **File > New > Other**. The new wizard displays.
2. Expand the Client Services folder and select **Convert Project to Client Services Project**.

Table 58. Convert Existing Project Page

Option	Description	Default Value
Available projects	Select the project to be converted.	None
Update Java build path for required bundles	For Java projects, selecting this option will convert the project's Java Build Path to use the Plug-in Dependencies class path container. Deselecting this option will leave the Java Build Path as is.	Selected

This is followed by the Platform Profile page:

Table 59. Platform profile options

Option	Description	Default value
Platform Profile	Select from the list the Platform Profile the Client Services Project will target. You can change your selection later in the Client Services property page.	No default value
Application Services	Check the Application Services that your Client Services Project will require. You can change your selection later in the Client Services property page. Grey entries are required by the Platform Profile and cannot be unchecked.	The ″Core OSGi Interfaces″ Application Service is required by all Platform Profiles.

This is followed by the Client Services Project Options page:

Table 60. Client Services Project Options

Option	Description	Default Value
Search for dependencies automatically upon resource changes	Select this option to enable the tools to search for package dependencies whenever the user modifies source files. When this option is deselected, the tooling will not search for any unresolved or unused dependencies in your project.	Selected
Attempt to automatically resolve Manifest dependencies	Select this option to enable the tools to automatically manage the package dependency information in the manifest file. Package dependencies in your project's Java code will automatically be reflected through proper updates to the manifest file. When this option is deselected, package dependencies that are not properly reflected in the manifest are flagged with problem markers, along with quick fixes to resolve the problems.	Selected
Give preference to Require-Bundle	Require-Bundle will be used to automatically resolve a package dependency in cases where either Require-Bundle or Import-Package can be used.	Selected by default for projects that are contributing to the Rich Client Platform.

Table 60. Client Services Project Options (continued)

Option	**Description**	**Default Value**
Give preference to Import-Package	Import-Package will be used to automatically resolve a package dependency in cases where either Require-Bundle or Import-Package can be used.	Selected by default for projects that are not contributing to the Rich Client Platform.

Client Services Project Properties page

Use this properties page to update the properties of a Client Services Project. Both platform profile selections and project options can be updated. To access this page, right click on the project in the Package Explorer view, select **Properties**, and then select **Client Services**.

Table 61. Application Profile tab

Option	**Description**	**Default Value**
Platform Profile	Select a platform profile this Client Services project will target.	None
Application Services	Select the Application Services that the project requires.	The "Core OSGi Interfaces" Application Service is required by all Platform Profiles.

Table 62. Options tab

Option	**Description**	**Default Value**
Search for dependencies automatically upon resource changes	Select this option to enable the tools to search for package dependencies whenever the user modifies source files. When this option is deselected, the tooling will not search for any unresolved or unused dependencies in your project.	Selected
Attempt to automatically resolve Manifest dependencies	Select this option to enable the tools to automatically manage the package dependency information in the manifest file. Package dependencies in your project's Java code will automatically be reflected through proper updates to the manifest file. When this option is deselected, package dependencies that are not properly reflected in the manifest are flagged with problem markers, along with quick fixes to resolve the problems.	Selected

Table 62. Options tab (continued)

Option	Description	Default Value
Give preference to Require-Bundle	Require-Bundle will be used to automatically resolve a package dependency in cases where either Require-Bundle or Import-Package can be used.	Selected by default for projects that are contributing to the Rich Client Platform.
Give preference to Import-Package	Import-Package will be used to automatically resolve a package dependency in cases where either Require-Bundle or Import-Package can be used.	Selected by default for projects that are not contributing to the Rich Client Platform.

Dialogs

The WebSphere Everyplace Client toolkit includes the following dialogs:

WebSphere Everyplace Deployment Runtime Launch Configuration dialog

The WebSphere Everyplace Client toolkit supports launching a local instance of the IBM WebSphere Everyplace Deployment runtime. This supports the ability to both run and debug Client Services projects from your workspace. The WebSphere Everyplace Deployment runtime launch extends the Eclipse run-time workbench launch. It is suggested that you use the WebSphere Everyplace Deployment launcher for running Client Services projects, since it automatically handles setting up the proper native library environment for the WebSphere Everyplace Deployment runtime.

Perform the following procedure to run or debug a project using the WebSphere Everyplace Deployment runtime launch:

1. Select **Run > Run...** to run under the WebSphere Everyplace Deployment runtime, or select **Run > Debug...** to debug under the WebSphere Everyplace Deployment runtime.
2. Select **WebSphere Everyplace Deployment** under configurations, and click **New** to create a new configuration.

   Note: If WebSphere Everyplace Deployment runtime configurations have already been created, you can directly select one.
3. On the arguments tab, insure that the JRE selected is the J2SE 1.4.2.
4. By default, the launcher selects all the plug-ins/application services from the WebSphere Everyplace Deployment default platform profile. To change the plug-ins/application services in your WebSphere Everyplace Deployment instance, please go to the Profile tab.
5. Click either the **Run** or **Debug** button to launch the runtime.

Refer to the **Running a Plug-in** section of the *PDE Guide* for further information on launch options.

Debugging a remote WebSphere Everyplace Deployment runtime: To debug an IBM WebSphere Everyplace Deployment runtime that has been started through other means than the WebSphere Everyplace Deployment runtime launcher, refer to “Remote debugging and testing” on page 159.

WebSphere Everyplace Client Toolkit Preference Dialogs

To access and set preferences for the WebSphere Everyplace Client Toolkit, select **Window > Preferences > WebSphere Everyplace Client Toolkit**.

Refer to the following tables for a description of the options and their default values of the resulting dialog screens:

Manifest Editor Preferences

Table 63. Manifest editor preferences

Option	Description	Default value
Incorrect manifest syntax	The syntax of the manifest header is incorrect or invalid	Error
Unresolved bundle references	The specified bundle reference does not exist in the PDE target platform or workspace	Error
Invalid reference values	The specified reference is invalid for the given manifest headers	Error
Invalid fragment references	The specified reference is invalid in the fragment manifest file	Warning
Required attributes not defined	A required attribute for the manifest header is missing	Error
Unknown attributes	The specified attribute of the manifest is not defined in the OSGi specification	Warning
Unknown attribute values	The attribute value defined in the manifest is incorrect or invalid	Warning
Secondary dependencies	The secondary dependency is referenced, but is not defined in the bundle’s manifest file	Warning
Unused references	The given reference is defined but not used in the bundle	Warning
Unknown resources	The given resource cannot be resolved	Error

Export Preferences

Table 64. Manifest editor preferences

Option	Description	Default value
JAR Format	The naming schema for exporting Client Service projects as OSGi Jar bundles	`{BundleSymbolicName}_{version}`
Use Advanced PDE Export	Select this option to enable the Advanced PDE export that allows users to export Client Service Web and Embedded Transaction Projects as plug-ins or in features and update sites	Selected

Advanced topics

Platform Profile and Extension Points

Creating a Platform Profile

An XML-based platform profile descriptor file describes the Platform Profile. You can define your own Platform Profile by creating a platform profile descriptor file and adding it to the tools through the platform profile extension point. Refer to the `com.ibm.pvc.tools.bde.profiles` plug-in for platform profile examples. A schema for platform profiles is also available in the `com.ibm.pvc.tools.bde` plug-in under `schema/PlatformProfile.xsd`.

The following information assumes that you have knowledge of the OSGi Alliance concepts, such as bundles and services.

You can externalize the descriptive text in the profile for internationalization. The Platform Profile uses the same mechanism for externalizing strings as Rational Software Development Platform uses for `plugin.xml`. The tools consider words within text strings that begin with the percent character (%) to be keywords for referencing corresponding text from a properties file. The default properties file base name is the same as the Platform Profile base name. You can explicitly specify the name of the properties file through the `PlatformProfile` elements' `PropertyFile` attribute. Properties files for specific languages follow the Java resource bundle naming conventions.

Refer to the "Platform Profile Document Type Definition" for the platform profile format. Refer to "Platform Profile elements and attributes" on page 256 for descriptions of the elements and attributes.

Platform Profile Document Type Definition: The following Document Type Definition (DTD) defines the platform profile format.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT PlatformProfile
(JCL,ApplicationService*,RequiredApplicationService*,RequiredRuntimeService*)>
<!ATTLIST PlatformProfile
  ID           CDATA #REQUIRED
  Name         CDATA #REQUIRED
  Provider     CDATA #IMPLIED
  Description  CDATA #IMPLIED
  PropertyFile CDATA #IMPLIED
>
<!ELEMENT JCL (#PCDATA)>
<!ATTLIST JCL
  Name         CDATA #REQUIRED
  Type         (ECLIPSE_JRE | J9_WCE | J9_WME) #REQUIRED
>

<!ELEMENT ApplicationService (Bundle*,ImportService*, DynamicImport-Package*,
      ClasspathLibrary*, BootLibrary*, NativeLibrary*)>
<!ATTLIST ApplicationService
  Name         CDATA #REQUIRED
  ID           CDATA #IMPLIED
  Description  CDATA #IMPLIED
  Version      CDATA #IMPLIED
>

<!ELEMENT RequiredApplicationService EMPTY>
<!ATTLIST RequiredApplicationService
  ID           CDATA #REQUIRED
>
```

```
<!ELEMENT RequiredRuntimeService EMPTY>
<!ATTLIST RequiredRuntimeService
  ID            CDATA #REQUIRED
>

<!ELEMENT Bundle (#PCDATA)>
>

<!ELEMENT ImportService (#PCDATA)>

<!ELEMENT DynamicImport-Package (#PCDATA)>

<!ELEMENT ClasspathLibrary (#PCDATA)>
<!ATTLIST ClasspathLibrary
  ReferenceType (PLUGIN_RELATIVE | PLATFORM_PLUGIN_RELATIVE
      | CLASSPATH_VARIABLE | ABSOLUTE) #REQUIRED


<!ELEMENT BootLibrary (#PCDATA)>
<!ATTLIST BootLibrary
  Position (PREPEND | APPEND) #IMPLIED
  ReferenceType (PLUGIN_RELATIVE | PLATFORM_PLUGIN_RELATIVE
      | CLASSPATH_VARIABLE | ABSOLUTE) #REQUIRED
>

<!ELEMENT NativeLibrary (#PCDATA)>
<!ATTLIST NativeLibrary
  OS ( Windows 2000 | Windows2000 | Win2000 | Windows XP | WindowsXP
 | WinXP | Linux ) #REQUIRED
  Processor ( x86 ) #REQUIRED
  ReferenceType (PLUGIN_RELATIVE | PLATFORM_PLUGIN_RELATIVE
      | CLASSPATH_VARIABLE | ABSOLUTE) #REQUIRED
>
```

Platform Profile elements and attributes: In the following descriptions, *development time* refers to the development environment of a Client Services project and *runtime* refers to the WebSphere Everyplace Deployment Client v6.0 runtime environment.

PlatformProfile element

The `PlatformProfile` element contains the definition of a Platform Profile. Refer to the following table for the elements in the PlatformProfile element and their descriptions.

Table 65. Elements in the PlatformProfile element

Element	Description
JCL	Defines the Java Class Library that the platform uses.
ApplicationService	Defines each Application Service provided by the platform.
RequiredApplicationService	Defines the Application Services that are required at development time and runtime.
RequiredRuntimeService	Defines the Application Services that are required at runtime, but are not required at development time.

Refer to the following table for the attributes in the PlatformProfile element and their descriptions.

Table 66. Attributes in the PlatformProfile element

Attributes	Description
ID	Specifies a unique identification, in the format of a Uniform Resource Locator (URL), for the platform profile.
Name	Specifies a descriptive name for the Platform Profile.
Provider	Specifies the provider name.
Description	Provides a text description of the Platform Profile.
PropertyFile	Specifies a file name of the properties file to be used for resource strings. This name should not include the `.properties` extension. If this attribute is not included, the default value is the base file name of the platform profile.
DefaultProduct	Specifies the id of a default Eclipse Product for the WebSphere Everyplace Deployment Launcher
DefaultApplication	Specifies the id of a default Eclipse Application for the WebSphere Everyplace Deployment Launcher.

JCL element

The `JCL` element defines the Java Class Library (JCL). The WebSphere Everyplace Client Toolkit will set this JCL in the class path of a Client Services project that uses this platform. Client Services also uses the JCL as a Client Services server that is configured to use this platform. The WebSphere Everyplace Client Toolkit identifies which JCL to use based on the value of this element and the value of the Type attribute. Refer to the following table for the attributes in the JCL element and their descriptions.

Table 67. Attributes in the element

Attributes	Description
Name	Specifies a descriptive name for the Java Class Library.
Type	Specifies the type of the Java Class Library. WebSphere Everyplace Client Toolkit uses this attribute and the value of the JCL element to determine which Java Class Library to use. Refer to the following list for valid values: • `ECLIPSE_JRE` indicates that the server uses the default Java Runtime Environment (JRE) that was selected in the WebSphere Studio Java Properties. If you specify this value, the value of the JCL element is ignored. • `J9_WCE` indicates that the server uses the IBM J9 Java Virtual Machine (JVM) and the Java Class Library indicated by the value of the JCL element. The value of the JCL element must an IBM JCL. • Custom Environment Java Class Libraries. • `J9_WME` indicates that the server uses the IBM J9 Java Virtual Machine (JVM) and the Java Class Library indicated by the value of the JCL element. The value of the JCL element must be one of the WebSphere Everyplace Micro Environment Java Class Libraries.

ApplicationService element

The ApplicationService element defines an Application Service provided by the platform. Bundle elements define the set of bundles that are associated with the Application Service. ImportService elements define the service import dependencies for users of this Application Service. DynamicImport-Package elements define the dynamic import package requirements for users of the Application Service. ClasspathLibrary elements define class path libraries used by the Application Services. BootLibrary elements define boot class path libraries used by the Application Service. NativeLibrary elements define the native libraries used by the Application Service.

Refer to the following table for the attributes in the ApplicationService element and their descriptions.

Table 68. Attributes in the ApplicationService element

Attributes	**Description**
Name	Specifies a descriptive name for the Application Service.
ID	Specifies a unique identification used to refer to this element.
Version	Specifies the version of the Application Service. The version format is major.minor.service, where major, minor, and service are integers that represent the version of the Application Service.
Description	Provides a text description of the Application Service.

ReferenceType attribute

Refer to the following table for a list of the ReferenceType attribute values and their descriptions.

Table 69. ReferenceType attribute form values and descriptions

Value	**Description**
PLUGIN_RELATIVE	Specifies a resource reference that is relative to a plug-in in the tools configuration. The first part of the reference path must be the plug-in ID. The remainder of the path represents the location of the resource within the plug-in. For example, a reference to myLib.jar under the lib directory, within the com.mycompany.myplugin plug-in is com.mycompany.myplugin/lib/myLib.jar. **Note:** This value is not supported for the ReferenceType attribute for Bundle elements. It is supported for the BootLibrary and NativeLibrary elements.
PLATFORM_PLUGIN_RELATIVE	The same as PLUGIN_RELATIVE, except the specified resource is relative to a plug-in from the current plug-in development target platform.
CLASSPATH_VARIABLE	Specifies a resource reference that is relative to the value of a class path variable that is defined in the Rational Software Development Platform environment. For example, if you defined the class path variable MY_BUNDLES as C:/SMF/bundles, then the reference MY_BUNDLES/myBundle.jar refers to the file C:/SMF/bundles/myBundle.jar.

Table 69. ReferenceType attribute form values and descriptions (continued)

Value	Description
ABSOLUTE	Specifies an absolute file reference to the resource. This is an operating system specific reference, such as `C:/SMF/bundles/myBundle.jar`.

Bundle element
The `Bundle` element value references an OSGi bundle or Eclipse plug-in. The value of the Bundle element is the symbolic name of the bundle. The bundle is expected to be in the current plug-in development target platform.

ImportService element
The `ImportService` element specifies a service that a project using the associated Application Service must import. The value of the `ImportService` element is the service interface class. When a project selects the Application Service associated with this element, the WebSphere Everyplace Client Toolkit add this service to the project Manifest file under the Import-Service header.

DynamicImport-Package element
The `DynamicImport-Package` element specifies a dynamic package specification that a project using the Application Service associated with this element must import. When a project selects the Application Service associated with this element, the WebSphere Everyplace Client Toolkit adds this dynamic package to the project Manifest file under the `DynamicImport-Package` header.

ClasspathLibrary element
The `ClasspathLibrary` element references a library JAR file that is needed on the development time and runtime class path by the Application Service. The `ReferenceType` attribute of the `ClasspathLibrary` defines the form of the library reference. Refer to Table 69 on page 258 for a list of the `ReferenceType` attribute values and their descriptions.

BootLibrary element
The `BootLibrary` element references a library JAR file that is needed on the runtime boot class path by the Application Service. The `ReferenceType` attribute of the `BootLibrary` defines the form of the library reference. Refer to Table 69 on page 258 for a list of the `ReferenceType` attribute values and their descriptions. The `Position` attribute of the `BootLibrary` defines where the library appears in the boot class path. Refer to the following table for a list of the `Position` attribute values and their descriptions.

Table 70. Position attribute values and descriptions

Value	Description
PREPEND	Specifies that the boot library should be prepended to the boot class path.
APPEND	Specifies that the boot library should be appended to the boot class path. This is the default value if the Position attribute is not specified.

NativeLibrary element
The `NativeLibrary` element references a file or directory that contains native libraries used by the Application Service. The `ReferenceType` attribute of the `NativeLibrary` defines the form of the library reference. Refer to Table 69 on page 258 for a list of the `ReferenceType` attribute

values and their descriptions. The OS attribute of the NativeLibrary defines what operating system this location is appropriate for. The Processor attribute of the NativeLibrary defines what processor type this location is appropriate for.

RequiredApplicationService element

The RequiredApplicationService element specifies an Application Service that must be present at development time by a project that uses this platform and that must be present at runtime on a WebSphere Everyplace Deployment Client v6.0 server that runs this platform. The WebSphere Everyplace Client Toolkit automatically include this Application Service in the set of services that a project uses when the project selects the platform profile. The WebSphere Everyplace Client Toolkit automatically install and start this service on local Client Services servers that are configured to use this platform.

You can reference the Application Service with the ID attribute. The value you specify for the ID attribute must match the ApplicationService element's ID attribute in the platform profile.

RequiredRuntimeService element

The RequiredRuntimeService element specifies an Application Service that must be present at runtime on a WebSphere Everyplace Deployment server that runs this platform. The WebSphere Everyplace Client Toolkit automatically installs and starts this service on local WebSphere Everyplace Client servers that are configured to use this platform.

You can reference the Application Service with the ID attribute. The value you specify for the ID attribute must match the ApplicationService element's ID attribute in the platform profile.

Extension points

WebSphere Everyplace Client toolkit includes the "com.ibm.pvc.tools.bde.platformprofilerepository" extension point.

com.ibm.pvc.tools.bde.platformprofilerepository:

Platform Profile:

Identifier:

com.ibm.pvc.tools.bde.platformprofilerespository

Description:

This extension point allows a plug-in to contribute platform profiles.

Configuration markup:

```
<! ELEMENT RepositoryLocation (#PCDATA)>
```

The value of RepositoryLocation is a path to a directory, which is relative to the directory where the plug-in resides, that contains the Platform Profile descriptor files.

Examples:

The following example contributes Platform Profiles that reside in the resources/profiles directory for the plug-in.

```
<extension point="com.ibm.pvc.tools.bde.platformprofilerepository">
  <RepositoryLocation>
     resources/profiles
  </RepositoryLocation>
</extension>
```

Appendix. Notices

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

IBM might have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

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

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

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

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

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

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

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

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) 2004, 2005. Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *2004, 2005* All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

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

EveryPlace
IBM
Rational
Rational Suite
WebSphere
Workplace
Workplace Client Technology

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

JavaScript™ is a trademark of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds.

Microsoft and Windows are trademarks of Microsoft Corporation.

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