<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Preface</td>
</tr>
<tr>
<td></td>
<td>Audience</td>
</tr>
<tr>
<td></td>
<td>Document Conventions</td>
</tr>
<tr>
<td></td>
<td>Related Documents</td>
</tr>
<tr>
<td></td>
<td>Contact Information</td>
</tr>
<tr>
<td>2</td>
<td>Introducing Pramati Server Deployment Process</td>
</tr>
<tr>
<td></td>
<td>Deployment Methods</td>
</tr>
<tr>
<td></td>
<td>Using Deploy Tool</td>
</tr>
<tr>
<td></td>
<td>Using Drag-and-Drop</td>
</tr>
<tr>
<td></td>
<td>Exploded Archive Deployment</td>
</tr>
<tr>
<td></td>
<td>Using Console</td>
</tr>
<tr>
<td></td>
<td>Using Server Shell Operations</td>
</tr>
<tr>
<td></td>
<td>Configuring Deployment Settings</td>
</tr>
<tr>
<td></td>
<td>Auto Deploy Directory Properties</td>
</tr>
<tr>
<td></td>
<td>Application Validation Properties</td>
</tr>
<tr>
<td></td>
<td>DTD Validation</td>
</tr>
<tr>
<td></td>
<td>Retaining Container Generated Files During Deployment</td>
</tr>
<tr>
<td></td>
<td>Compiler Properties</td>
</tr>
<tr>
<td></td>
<td>Classloader Configuration</td>
</tr>
<tr>
<td></td>
<td>Application Versioning</td>
</tr>
<tr>
<td></td>
<td>Server Startup Hooks</td>
</tr>
<tr>
<td></td>
<td>Application Hooks</td>
</tr>
<tr>
<td>3</td>
<td>Using Deploy Tool</td>
</tr>
<tr>
<td></td>
<td>Starting Deploy Tool</td>
</tr>
<tr>
<td></td>
<td>Using Start &gt; Programs Option in Windows</td>
</tr>
<tr>
<td></td>
<td>Executing Batch File and Connecting it to Server</td>
</tr>
<tr>
<td></td>
<td>Using Server Shell</td>
</tr>
<tr>
<td></td>
<td>Deploy Tool Interface</td>
</tr>
<tr>
<td></td>
<td>Preparing Applications for Deployment</td>
</tr>
<tr>
<td></td>
<td>Viewing Incomplete Tasks</td>
</tr>
<tr>
<td></td>
<td>Resolving Tasks</td>
</tr>
<tr>
<td></td>
<td>Using Deploy Tool to Resolve Tasks</td>
</tr>
<tr>
<td></td>
<td>Refreshing Server Details</td>
</tr>
<tr>
<td></td>
<td>Saving Archives</td>
</tr>
<tr>
<td></td>
<td>Prepared Archives</td>
</tr>
<tr>
<td></td>
<td>Preparing Code for Deployment</td>
</tr>
<tr>
<td></td>
<td>Making Prepared Archives</td>
</tr>
<tr>
<td></td>
<td>Starting Applications</td>
</tr>
<tr>
<td></td>
<td>Deploying Applications</td>
</tr>
<tr>
<td></td>
<td>Accessing Deployed Applications</td>
</tr>
<tr>
<td></td>
<td>Deployment Settings</td>
</tr>
<tr>
<td></td>
<td>Global Configuration Settings</td>
</tr>
<tr>
<td></td>
<td>Archive Specific Customization</td>
</tr>
<tr>
<td></td>
<td>Specifying Deployment Setting in Pramati Deployment Descriptor</td>
</tr>
<tr>
<td></td>
<td>Setting EJB Properties Using Deploy Tool</td>
</tr>
<tr>
<td></td>
<td>Setting Properties for Stateful Session Beans</td>
</tr>
<tr>
<td></td>
<td>Setting Properties for Stateless Session Beans</td>
</tr>
<tr>
<td></td>
<td>Setting Properties for Entity Beans (Container Managed)</td>
</tr>
<tr>
<td></td>
<td>Setting Properties for Message Driven Beans</td>
</tr>
</tbody>
</table>
Migrating Local JNDI Names ......................................................... 21
Deploy Tool Logs ................................................................. 22

4 Validation Conditions .......................................................... 25
Basic Jar Level Validation ........................................................ 25
Basic Bean Class Level Validation ............................................ 26
Validating Entity Beans .......................................................... 27
EJB 2.0 Specific Validation ...................................................... 29
Validating MessageDriven Beans ............................................ 30
Validating Session Beans ....................................................... 31
Validating Web Module ........................................................ 31

5 Validating Applications for Deployment on Pramati Server .... 33
Application Validation ............................................................ 33
Validating Archives ............................................................... 33
Validating Archives Using Deploy Tool .................................... 34
Validating Archives Offline Using Command Shell Interface .... 34
Validating Archives From Remote Shell Prompt ....................... 35

6 Preparing Applications for Deployment .................................... 37
Packaging Semantics for J2EE Application ............................... 38
Packaging J2EE Components for Server .................................. 38
Configuring Resources for Applications .................................. 38
Generating Queries ............................................................... 39
Configuring Security Parameters .......................................... 39
Configuring Transaction Attributes ....................................... 39
Configuring Pramati Message Server ..................................... 40

7 Preparing and Deploying Web Applications .............................. 41
Packaging Web Applications .................................................. 41
Deploying Web Applications ................................................ 41
Running Web Applications ................................................... 41
Web Properties ................................................................. 42
General Properties ............................................................. 42
Environment Properties ....................................................... 42
WAR Properties ................................................................. 42
EJB References ................................................................. 42
Resource References ........................................................ 43
Resource Environment References ....................................... 44
Security Role References ..................................................... 44
Multiple Role Mapping ....................................................... 45
Working With Servlets on Pramati Server ............................... 48
Running Servlets Without Packaging ...................................... 48
Adding Servlets to Running Server ....................................... 48
Accessing Servlets ............................................................. 48
Working With JSPs on Pramati Server ..................................... 48
Packaging and Deploying JSPs on Server ............................... 49
Deploying JSPs on Server ..................................................... 49
Hot Deployment ................................................................. 49
Viewing Deployed Web Clients .............................................. 49
Understanding Pramati Server Web Logs ................................ 49
Limitations in Mapping SQL Data Types to Java Types ........................................ 79

11 Setting Up Concurrency Control for CMP Beans .............................................. 81
   Concurrency Type ............................................. 81
   Repeatable Read ................................................ 81
      Case 1: Exclusive DB, Performance Advantage ........................................ 81
      Case 2: Non-Exclusive DB .................................... 81
   Optimistic Repeatable Read ...................................... 82
      Use Case Scenario ......................................... 82
   Read Committed .............................................. 82
   Read Only .................................................... 83
   Configuring Concurrency Properties in Deploy Tool ................................. 83

12 Setting Up Session Beans .............................................................................. 85
   Session Timeout ................................................ 85
   Method Information ........................................... 85

13 Setting Up Message Driven Beans ................................................................. 87
   Components of Message Driven Server .................................................. 87
   MDB Destinations ............................................. 88
   Pre-configured Destinations for MDB Sample ....................................... 88

14 Drag-and-Drop J2EE Applications ............................................................... 89
   AutoDeploy Directory .......................................... 89
      Configuring AutoDeploy Directory ........................................ 89
   Dropping Archives Into AutoDeploy Directory .................................... 90
   Deploying Archives ............................................ 90
      Redeployment ............................................... 90
   Auto Deployment Poll Interval ........................................ 90
   Limitations of AutoDeploy Directory ........................................... 91
   Pramati Deploy Service ........................................ 91
      Auto-resolution Framework .................................... 91
      Feedback Mechanism ....................................... 92
      Automatic Table Creation ................................... 93
      Relationship Mapping ....................................... 93
      Generating pramati-or-map.xml ................................ 93

15 Deploying Exploded Archives ...................................................................... 95
   Pre-requisites for Deploying Exploded EARs ......................................... 95
   Working of Exploded EAR Deployments ......................................... 96
   Using Command Line ........................................... 96

16 Deploying Applications Using Console ....................................................... 99
   Deploying Applications Using Console ........................................... 99

17 Deploying Applications Using Server Shell ................................................ 101
   Types of Shells .................................................. 101
   J2EE Shell Commands for Managing Applications .................................. 102
   Validating Archives From Server Shell ............................................ 102

18 Deploying Applications Using ANT Handler .............................................. 105
   Getting Started With ANT Handler ........................................ 105
   Pramati ANT Handler Commands ........................................ 106
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connecting to Server</td>
<td>106</td>
</tr>
<tr>
<td>A Sample XML</td>
<td>107</td>
</tr>
<tr>
<td><strong>19 Starting and Accessing Applications</strong></td>
<td></td>
</tr>
<tr>
<td>Starting Applications</td>
<td>109</td>
</tr>
<tr>
<td>Starting Applications From Console</td>
<td>109</td>
</tr>
<tr>
<td>Starting Applications From Server Shell</td>
<td>109</td>
</tr>
<tr>
<td>Starting Applications Automatically</td>
<td>109</td>
</tr>
<tr>
<td>Accessing Deployed Applications</td>
<td>110</td>
</tr>
<tr>
<td>Accessing Applications Across Firewalls</td>
<td>110</td>
</tr>
<tr>
<td>HTTP Tunneling, Port Filtering Firewalls</td>
<td>110</td>
</tr>
<tr>
<td>Configuring Client</td>
<td>111</td>
</tr>
<tr>
<td>Configuring RMI Server</td>
<td>111</td>
</tr>
<tr>
<td>Working With NAT Firewalls</td>
<td>111</td>
</tr>
<tr>
<td>Default Socket Factory</td>
<td>112</td>
</tr>
<tr>
<td>Configuring Server for Firewalls</td>
<td>112</td>
</tr>
<tr>
<td><strong>20 Using Connectors</strong></td>
<td></td>
</tr>
<tr>
<td>Packaging Resource Adapters</td>
<td>113</td>
</tr>
<tr>
<td>Location of Resource Adapter Archive Components</td>
<td>113</td>
</tr>
<tr>
<td>Managing Standard Resource Adapter Deployment Descriptor</td>
<td>113</td>
</tr>
<tr>
<td>Managing Pramati Specific Resource Adapter Deployment Descriptor</td>
<td>114</td>
</tr>
<tr>
<td>Usage Scenario</td>
<td>115</td>
</tr>
<tr>
<td>Client Interfaces</td>
<td>115</td>
</tr>
<tr>
<td>Sample Code</td>
<td>115</td>
</tr>
<tr>
<td>Pramati JCA Implementation</td>
<td>116</td>
</tr>
<tr>
<td>Deploying Resource Adapters</td>
<td>117</td>
</tr>
<tr>
<td>Deploying Applications</td>
<td>118</td>
</tr>
<tr>
<td>Starting Applications</td>
<td>118</td>
</tr>
<tr>
<td>Undeploying Resource Adapters</td>
<td>118</td>
</tr>
<tr>
<td>Deploying Resource Adapters on Cluster Nodes</td>
<td>118</td>
</tr>
<tr>
<td>Creating Connector Resources After Deploying Resource Adapters</td>
<td>119</td>
</tr>
<tr>
<td>Using Server Shell to Create Connector Resources (for a 1.0 Connector)</td>
<td>119</td>
</tr>
<tr>
<td>Using Server Shell to Create Connector Resources (for a 1.5 Connector)</td>
<td>120</td>
</tr>
<tr>
<td>Using J2EE Connector 1.5 Resource Adapters for Inbound Communication</td>
<td>121</td>
</tr>
<tr>
<td><strong>21 Creating Application Hooks</strong></td>
<td></td>
</tr>
<tr>
<td>Specifying Server Application Hooks</td>
<td>123</td>
</tr>
<tr>
<td>ApplicationStateListener Interface</td>
<td>124</td>
</tr>
<tr>
<td>Sample Implementation</td>
<td>125</td>
</tr>
<tr>
<td>Configuring Application Hooks</td>
<td>125</td>
</tr>
<tr>
<td><strong>22 Creating Server Startup Hooks</strong></td>
<td></td>
</tr>
<tr>
<td>Creating Server Hook</td>
<td>127</td>
</tr>
<tr>
<td>Compiling Server Hook Implemented Class</td>
<td>128</td>
</tr>
<tr>
<td>Editing deploy-config.xml</td>
<td>128</td>
</tr>
<tr>
<td><strong>23 Classloader Architecture in Pramati Server</strong></td>
<td></td>
</tr>
<tr>
<td>Classloaders in Pramati Server</td>
<td>129</td>
</tr>
<tr>
<td>Features</td>
<td>130</td>
</tr>
<tr>
<td>Classloader Hierarchy</td>
<td>130</td>
</tr>
<tr>
<td>Classloader Lifecycle</td>
<td>134</td>
</tr>
</tbody>
</table>
Application Classloading in Server ............................................. 135
Packaging jar, war, and ear ....................................................... 135
Placing Application Files in Other Paths .................................. 135
Packaging Helper jars ............................................................ 136
Functionality of MANIFEST.MF .................................................. 136
Effective Path in Case of Multiple MANIFEST.MF Files ............... 137
Case Studies .............................................................................. 137
Deployment Scenarios ............................................................... 138
Undeployment ........................................................................... 138
Redeployment ........................................................................... 138
Stopping Server ........................................................................ 139
Java Clients .............................................................................. 139
Classes/JARs Required for Java Clients to Access EJBs .............. 139
How are Each of These Available? ............................................. 139
Accessing Message Server From Application ............................ 139
Where must Message Class be? ................................................ 139
Specifying JDBC Drivers ............................................................ 139
Distributing JDBC Drivers With Application ............................ 139
setup.bat ................................................................................... 140
JRE lib/ext. ............................................................................... 140
Problems .................................................................................. 140
Excessive Visibility .................................................................... 140
Low Visibility ............................................................................ 141
Mutilated Visibility .................................................................... 141
FAQs ......................................................................................... 141
Classloading Guidelines ............................................................. 143
Declare Dependencies ............................................................... 143
Grouping Dependencies ............................................................ 143
Minimize Visibility .................................................................... 143
Share Libraries .......................................................................... 143
Use Correct CL ......................................................................... 143
Glossary ..................................................................................... 144

24 Configuring Application MBeans on Pramati Server .............. 145
Steps to Register Application MBeans at Time of Deployment .... 145
Samples ...................................................................................... 146

25 Interoperability ................................................................. 147
Overview ................................................................................... 147
RMI-IIOP Interoperability .......................................................... 147
Enabling ORB Runtime in Server .............................................. 147
Configuring EJB Applications to use IIOP ................................. 148
Connecting to Pramati Server From Native Clients Using IIOP .... 149
Deploying EJB on Pramati Server Over IIOP ............................. 149
Writing Clients .......................................................................... 150
Compiling Clients ....................................................................... 151
Running Clients ......................................................................... 151
Connecting to Pramati Server From Java Clients Using RMI IIOP 151
Deploying EJB on Pramati Server .............................................. 151
Writing Clients .......................................................................... 151
Compiling Clients ....................................................................... 152
Running Clients ................................................................. 152
Using CORBA-Based Security for EJBs on Pramati Server .......... 153
Transport Layer Security .................................................. 153
Application Level Authentication ....................................... 153
Application Level Identity Assertion .................................... 154
Custom Implementation for Server Authentication Details Provider .. 154
Connecting to CORBA Objects From Pramati Server ................. 155

26 Using Jikes Compiler and Compilation Server ...................... 157
Using Jikes Compiler ....................................................... 157
Using Compilation Server ................................................ 158

27 Creating Deployment Descriptors .................................... 159
Standard XML Descriptors ............................................... 159
Pramati Specific XML Descriptors ..................................... 159

28 Samples for Deployment ............................................... 161
Deploying iBank Sample Application .................................. 161
Shipped Resource .......................................................... 161
Configuring Resources Using Other Databases ....................... 161
Opening Archives ......................................................... 162
Deploying Application ................................................... 163
Accessing Application .................................................... 163
The Numguess Sample Application ..................................... 164
Opening Archives ........................................................ 164
Deploying Application ................................................... 165
Accessing Application .................................................... 165
The Game of Eight Sample Application ................................ 166
Opening Archives ........................................................ 166
Deploying Application ................................................... 167
Accessing Application .................................................... 167
The OrderBook Sample Application .................................... 168
Shipped Resource ........................................................ 168
Adding Resources Using Other Databases .............................. 168
Opening Archive .......................................................... 169
Deploying Application ................................................... 170
Accessing Application .................................................... 170
The Stock Market Sample Application .................................. 171
Shipped Resource ........................................................ 172
Opening Archive .......................................................... 172
Deploying Application ................................................... 172
Accessing Application .................................................... 173

29 DTDs for Server XMLs .................................................. 175
Deployment Descriptors for EJB .......................................... 175
Deployment Descriptor for Web Applications ......................... 176
DTD for Pramati Java EE 5.0 Server .................................... 177
DTD for Pramati OR-mapping ............................................ 186
Preface

The *Pramati Server Deployment Guide* describes how to deploy Java™ 2 Platform Enterprise Edition (J2EE) applications on Pramati Server and explains how to customize the Pramati Deploy Tool.

Audience

This guide is intended for administrators and J2EE developers who have a basic knowledge of how to deploy various applications on the server.

Document Conventions

The following conventions are commonly used in this guide:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>Pramati Server</td>
</tr>
<tr>
<td>Console</td>
<td>Pramati Server Management Console</td>
</tr>
<tr>
<td>Deploy Tool</td>
<td>Pramati Deploy Tool</td>
</tr>
<tr>
<td>Item A &gt; Item B</td>
<td>Menu and sub-menu selections</td>
</tr>
<tr>
<td>code</td>
<td>Font used for writing code snippets, file names, and URLs</td>
</tr>
</tbody>
</table>

*Note:* Indented paragraphs in italics denote notes

press `<key>` Press the suggested key using the keyboard where suggested

`<install_dir>` Server installation directory

`<node_name>` Name of the node created by you

Screenshots used in this document represent the graphical user interface (GUI) of the Deploy Tool on the Windows NT platform. The GUI has a Java look and feel and may differ according to the operating system used.

Related Documents

To get more information about Pramati Server, refer to the following documentation:

- Pramati Server Installation and Configuration Guide
- Pramati Server Administration Guide
- Pramati Server Security Guide
- Pramati Server Performance Tuning Guide
- Pramati Server Technical Reference Guide
- Server Online Help
- Server API Javadoc (accessible at http://<localhost>/admin/java-doc/index.html, where <localhost> indicates where your server is running.)

**Contact Information**


For additional help with using Deploy Tool and troubleshooting, visit http://www.pramati.com or visit http://www.pramati.com/DevPortalWeb/forum/.
Introducing Pramati Server Deployment Process

Deploying a J2EE application means porting the application to a J2EE server. A J2EE application consists of:

- Enterprise JavaBeans™ (EJB) components such as entity beans, session beans, and Message Driven Beans (MDBs)
- Web components such as JavaServer Pages™ (JSP), HTML pages, servlets, and related files
- Connector components such as resource adapters

The J2EE application design blueprint identifies a deployer as one who ports an application to a specific operational environment. A deployer can use J2EE tools to perform deployment tasks effectively.

Archives, such as Java Archives (JARs), Web Archives (WARs), Enterprise Archives (EARs), and Resource Archives (RARs) can be deployed on an application server. To run an application, you need to start a standalone server or cluster, and deploy the archives.

Deployment Methods

An application can be deployed on the Server in the following ways:

- Using Deploy Tool
- Using the autodeploy directory (Drag-and-Drop)
- Using Exploded Archives
- Using the Pramati Server Management Console (referred to as the Console hereafter)
- Using the Server Shell

Using Deploy Tool

Server ships with a Deploy Tool that enables you to deploy enterprise applications on a local or remote Pramati Server. For further details, read “Using Deploy Tool” on page 9.

Using Drag-and-Drop

The Drag-and-Drop service is used to deploy applications by moving the deployable archive to a particular directory called the AutoDeployer directory. For further details, read “Drag-and-Drop J2EE Applications” on page 89.
Exploded Archive Deployment
Pramati Server now allows applications to be deployed right off the working directories. This feature is used during development stages. For further details, read “Deploying Exploded Archives” on page 95.

Using Console
Using Pramati Server Management Console, applications can be deployed as .ear, .jar, .war, .rar, or .par (Prepared Archive) files. For further details, read “Deploying Applications Using Console” on page 99.

Using Server Shell Operations
You can use the Server Shell to deploy applications. You can use the interactive mode and the batch mode. For further details, read “Deploying Applications Using Server Shell” on page 101.

Configuring Deployment Settings
The Deploy Service of Pramati Server is a core service that is started by the server kernel when you start the Pramati Server. This enables applications to be deployed seamlessly in the Server.

The services framework architecture (see the chapter “Understanding Pramati Services and Kernel Framework” in the Pramati Server Technical Reference Guide) delegates the application deployment process to the Deploy Service that takes care of reading the archives and deploying them based on the properties provided.

The following properties can be configured for the Deploy Service in the deploy-config.xml. This file is generated in the \<install_dir>/server/nodes/default/config directory when you start the Pramati Server.

1 Auto Deploy Directory Properties
2 Validation Properties
3 Compiler Properties
4 Classloading for Multiple applications
5 Application Versioning
6 Startup Hooks
7 Application Hooks

Let us discuss the configuration of each of these properties. All the above properties are specified in various tags in the deploy-config.xml file.

Auto Deploy Directory Properties
The Auto Deploy feature deploys applications automatically from the configured directory.
Chapter 2

Introducing Pramati Server Deployment Process

The following properties can be set in the `<auto-deploy>` tag:

```xml
<auto-deploy>
  <dir>$NODE_DIR/archives/autodeploy</dir>
  <poll-interval-seconds>5</poll-interval-seconds>
</auto-deploy>
```

The `<dir>` tag above specifies the directory where the Deploy Service picks the applications from. The default directory is `<install_dir>/server/nodes/default/archives/autodeploy`, where `default` is the name of the node.

The Deploy Service polls this directory time to time to see if there are any applications to be deployed. The poll interval value can be configured by specifying the `<poll-interval-seconds>` tag above.

To deploy an application, just drag-and-drop an archive into the directory. The AutoDeployer periodically checks for new file additions and deploys the newly added files onto the Server.

**Application Validation Properties**

The validation tags define the validation behavior of the Deploy Service during deployment. The Deploy Service provides for two types of validations:

1. **Application Validation** (`<app-archive-validation>`) - The application components are validated against the application provider's responsibilities in the specification and verified to be valid J2EE components. This process has no dependency on the Server environment and can also be used as a standalone process. It is recommended that validation is done when deploying for the first time.

2. **App Start Validation** (`<app-start-validation>`) - This validates the Server environment for an application. All resources required for the application are validated, any potential namespace clashes due to the start of this application are verified.

**DTD Validation**

DTD validation is also performed during creation of the document object model (DOM) structure from Deployment Descriptors. All validation flags can be controlled based on the application action, such as, turn on validation for deploy, turn it off for re-deploy, and so on.

The `<validation>` tag in the `deploy-config.xml` shows this:

1. **Application Archive Validation** - The `<app-archive-validation>` tag has configuration information using which you can enable application validation. Application validation can be done at deploy time, on subsequent deployments, or the archive, and at application startup. These can be specified by setting the respective attributes to either `true` or `false`:
   ```xml
   <app-archive-validation on-deploy="true" on-redeploy="true" on-start="false"/>
   ```
   This property can also be set using the Deploy Tool.

2. **DTD Validation** - The `<dtd-validation>` tag is used to enable DTD validation of the applications:
   ```xml
   <dtd-validation on-deploy="false" on-redeploy="false" on-start="false"/>
   ```
3 **Application Start Validation**: Application validation at startup can be enabled using the `<app-start-validation>` tag: `<app-start-validation enabled="true"/>

This property can also be done using the Deploy Tool.

4 **Checking Rules on Application Startup**: Rules checks exclusively for CMP beans where the mapping information is validated. The `db-table-check` checks for existence of the tables the beans are mapped against, while the `db-table-mapping-check` checks for all CMP field mapping.

   `<app-start-validation enabled="true">
    <rules>
      <db-table-check enabled="true"/>
      <db-table-mapping-check enabled="true"/>
    </rules>
   </app-start-validation>

**Retaining Container Generated Files During Deployment**

While deploying EJB components, the container generates various implementation classes for Home Interfaces and Remote Interfaces of the beans. These class files are directly placed in the containers classpath and used. The Deploy Service by default doesn’t retain a copy of the generated source file. These generated source files can be retained on deployment for debugging purposes. The `<retain-generated-files>` tag can be used to specify this.

   `<retain-generated-java-files>false</retain-generated-java-files>

The generated source files (.java files) are placed in application directory of the
<br>&lt;install_dir&gt;/server/nodes/default/archives/&lt;app_name&gt;/&lt;app_name_version&gt;/java.

The Deploy Service by default removes all generated files on deployment failures. These can be retained by setting the `<retain-files-on-deploy-failure>` tag to `true` as below:

   `<retain-files-on-deploy-failure>true</retain-files-on-deploy-failure>

Again, this feature can be used while debugging applications. This property can also be set using the Deploy Tool.

**Compiler Properties**

Pramati Server allows the use of different compilers to compile JSPs and generated EJB files. Compilation can happen either in-memory or out of process. The configuration information for the compilation properties can be specified in the `<compilation>` tag:

   `<compilation>
    <out-of-process-compilation enabled="false">
    </out-of-process-compilation>
   </compilation>`
Chapter 2
Introducing Pramati Server Deployment Process

In the above, if the enabled attribute of the <out-of-process-compilation> tag is set to true, the external compiler path and the compiler command (for example javac or jikes) must be specified in the <external-compiler> tag. In case this is not specified, the files are compiled using the in-memory compiler.

Classloader Configuration
The Deploy Service provides the following default behavior:
• One classloader for each application
• Classloader used for Web Components is the child classloader of the ejb/.rar components

To set a single classloader for multiple applications, modify the enabled attribute for the <single-classloader-for-applications> tag to true. This means that a single classloader can now be used for multiple applications as specified in the <application-group> attribute specified below.

The Deploy Service provides for application groups where a group of applications share the same classloader. Any number of groups can then be declared. Absence of any groups after enabling single application classloader would mean a universal application group where all applications share the same classloader. Any application not in the <application-group> attribute specified below gets its own classloader if no groups are specified.

```xml
<single-classloader-for-applications enabled="true">
  <!--sample application group definition -->
  <!--application-group apps="abcd.ear,xyz.jar" -->
  <!--any application not listed above gets its own classloader. If no groups are specified, then a single classloader is used for all applications.-->
</single-classloader-for-applications>
```

Application Versioning
The Deploy Service provides versioning for applications in the archives directory. The first version directory is created when the application is deployed for the first time. All subsequent versions are deployed into different directories. Information on the latest deployed version and the version to be used are maintained here. You can start any previous version.

A failed re-deployment means there is a valid version to use. To disable maintaining multiple directories, the versioning-enabled must be set to false.

```xml
<application-properties versioning-enabled="true"/>
<application name="DevPortalWeb">
  <version use="1" latest-version="1"/>
</application>
<application name="KBAppEJB.jar">
  <version use="1" latest-version="1"/>
</application>
```
To set this property using the Deploy Tool, open a .jar, and select **Application Versioning**.

**Server Startup Hooks**

Startup hooks are used to synchronize with Server lifecycle events. Deploy Service maintains such hooks. A class name entry below would mean that the class is to be used as a startup hook.

```xml
<startup-hooks>
  <!--class-name-->
</startup-hooks>
```

Refer to the interface `com.pramati.services.j2ee.spi.StartupHook` for more details.

**Application Hooks**

Application hooks are used to synchronize with application lifecycle events. Each application can have it's own hook. The `class-name` attribute is the class which implements the application hook interface. The `async` parameter specifies whether or not to call the hook methods in a separate thread. Refer to the interface `com.pramati.services.j2ee.spi.ApplicationStateListener` for more details.

```xml
<application-hooks>
  <!-- application-hook app-name="appName" async="true" class-name="class-Name"-->
</application-hooks>
```
Using Deploy Tool

Pramati Server ships with a Deploy Tool that is installed by default when you install Pramati Server. The Deploy Tool enables you to deploy enterprise applications on any J2EE-compliant application server. Applications can be deployed on local or remote servers.

Starting Deploy Tool

Important: Before you start the Deploy Tool, ensure that the server on which you want to deploy the application is running.

There are three ways to start the Deploy Tool:
• Using the Start > Programs option in Windows
• Executing the batch file and connecting it to Server
• Using the Server shell

Using Start > Programs Option in Windows

To start the Deploy Tool on Windows, select Start > Programs > Pramati 5.0 > Deploy Tool. This opens the Connect to Server dialog box as shown below:

![Connect To Server dialog box](image)
Provide the following details:

- **Server IP**: Specify the Server IP or the host name where the server is running. By default, this is shown as 127.0.0.1.
- **Naming Port**: Specify the port on which the server is listening. By default, this is shown as 9191.
- **User Name**: Specify the username provided while configuring the realm. For the realm `system`, the username is `root`.
- **Password**: Specify the password provided while configuring the realm. For the realm `system`, the password is `pramati`.
- **Realm**: Specify the realm name. The default value is `system`.

Click **Connect** or press **Enter** to open the Deploy Tool interface.

### Executing Batch File and Connecting it to Server

For Windows platforms, run `rundeploytool.bat` located in the directory `<install_dir>/server/bin`.

For Unix platforms, run `rundeploytool.sh` located in the directory `<install_dir>/server/bin`.

This opens the **Connect to Server** dialog box as shown above. Provide the required details and click **Connect** or press **Enter** to open the Deploy Tool interface.

### Using Server Shell

Alternatively, you can also start the Deploy Tool from the command shell of the Server. This is done using the `deploytool` command as below:

```
j2eeadmin@default> deploytool
Loading Server Environment
Starting DeployTool...
DeployTool successfully loaded
```

Deploy Tool invoked from the Server shell opens the Deploy Tool interface without prompting for the server details (the details are taken for the current instance). By default, it opens with a ‘No Archive Open’ message.

### Deploy Tool Interface

Deploy Tool has the following features:

- **Menu bar**: Contains default menu items such as Archive, View, and Help.
- **Toolbar**: Contains the commonly used buttons and components, namely, Open, Close, Save, Deploy, Exit, Settings, and Help.
- **Explore panel**: Lists all the components and properties of the archive or module opened in the Deploy Tool.
• **Display panel**: Displays all the components and properties of the selected archive or module. Click a node in the Explore panel to display all the components and properties for the selected archive or module in the Display panel.

• **Deployment task navigator**: Appears green when there are incomplete tasks to be finished before the application is deployed. Click the arrow in the lower right corner of the Deploy Tool window to open the deployment task monitor and view the list of incomplete tasks.

---

**Preparing Applications for Deployment**

Select **Archive > Open**. The **Open** dialog box appears. Browse for the application that is to be deployed. Deploy Tool can open .jar, .war, .ear, and .rar files.

The tool obtains the following information from the Server:

• JNDI names bound to the Server (both Remote and Local JNDI names)
• Context roots in use
• Security roles in use
• Resources bound to the Server

For an undefined resource, add the resource through the Console. For details, read the Chapter **Configuring Resources** of the *Pramati Server Administration Guide*.

Select **Archive > Refresh Server Details** to refresh the Server information displayed in the Deploy Tool. It is also possible to add resources to a running server.
Viewing Incomplete Tasks
You cannot deploy an application without resolving all tasks with regard to the Server environment. The arrow in the lower right corner of the Deploy Tool is green when there are incomplete tasks. Click the green arrow to move to the next incomplete task. To view all the incomplete tasks, choose View > All Tasks.
Complete all pending tasks using the Next Task option and resolve all deployment tasks before deploying an application.

Resolving Tasks
When you open an archive, Deploy Tool automatically detects required mappings and relationships. Incomplete mappings or unresolved references are listed as unresolved tasks. These tasks must be completed before the application can be deployed.
If the archive contains an entity bean or it requires to be connected to a database, use datasources to map the bean fields to the fields in the database table.

Using Deploy Tool to Resolve Tasks
1 In the main menu of the Deploy Tool, choose View > Next Task.
2 Map the datasource name, table name, and fields in the application with the fields in the table. You may be required to resolve other tasks depending on the type and complexity of your application. For more information about setting resources to an application, read “Preparing Applications for Deployment” on page 37.
3 After you resolve all incomplete tasks, the status bar indicates that the number of tasks left is 0. The application is now ready to be deployed.

Note: To ascertain the bean type, select the <name>.jar option (say, bankejb.jar) in the tree view present on the left side. Select the bean you want to know the details about.

Refreshing Server Details
Choose Archive > Refresh Server Details to refresh the Deploy Tool with the most recent Server details. This implies that to add datasources in the Server, you do not need to close and restart the Deploy Tool.

Saving Archives
To save the opened archive, choose Archive > Save from the main menu in the Deploy Tool. The archive is saved along with the pramati-j2ee-server.xml and pramati-or-map.xml, every time it is deployed. You are prompted to save the archive before closing the archive, before closing the Deploy Tool, or on clicking Exit in the main menu.
Prepared Archives

A prepared archive (.par) is an archive that does not require code generation and compilation. It can be distributed across EJB server within the similar environment.

The deployment process involves:

1. Generating runtime code.
2. Compiling the generated code and running the remote method invocation compiler (RMIC). This is a time-consuming operation. The generated and compiled code is saved in <install_dir>/server/nodes/<node_name>/archives (jar) and in <install_dir>/server/nodes/<node_name>/archives/public_html (war).
   *server_name* is the name of the configured server on which the application has been deployed.
3. Starting the prepared applications.

Code generation and RMIC operations usually delay the deployment process. On Pramati Server, you can generate code, perform RMIC, and start the prepared code at any point in time. You can then use the prepared code to build a .par without repeating the whole process.

Preparing Code for Deployment

Select Archives > Prepare from the main menu in the Deploy Tool to generate and build prepared code for both beans and Web components, and to perform RMIC.

Making Prepared Archives

The code that has been prepared using the Prepare option can then be used to make a .par file. Choose Archive > Make Prepared Archive to display all applications that have been prepared using the Prepare or Deploy option.

After preparing the archive, save it in a file with a .par extension at <install_dir>/server/nodes/<node_name>/archives/parchive.

**Example:** Selecting Archive > Make Prepared Archive > BankDesk.ear prepares and places the BankDesk.ear.par under <install_dir>/server/nodes/<node_name>/archives/parchives.

A .par file can be further distributed to any server where the application needs to be deployed. The .par files are particularly useful when applications that have been developed locally require to be deployed on the Internet.

Read “Preparing and Deploying Web Applications” on page 41 for more information about distributing applications.
Starting Applications
An archive that has been prepared for deployment using the Prepare option can be started by using Archive > Start in the main menu. Starting a prepared application completes the deployment process.

Using the Start option to deploy an application is faster than using the Deploy Tool because all the deployment tasks are already completed.

Deploying Applications
The application is ready to be deployed after all the deployment tasks have been resolved. Choose Archive > Deploy from the main menu to deploy the application. The progress bar displays the progress of the deployment process. To undeploy applications, choose Archive > Undeploy from the main menu.

Accessing Deployed Applications
After deploying an application, access it through any browser that supports HTML 3.2 and above. To access a deployed application, open the browser. Enter the location of the deployed application in the following format:

http://<host on which the server is running>:<port number on which the server is listening>/<context root>/<first page of the application>

The deployment process requires the deployer to deploy Web applications and EJBs, and resolve any server-specific dependencies that may arise during deployment.

Deployment Settings
The Deploy Tool allows for run-time configuration of deployment properties. Configuration changes can be done either globally or for each application. Deployment settings done globally are modified in the Deploy Service (that is, the changes are made in the deploy-config.xml) and are valid across all deployments. These changes are persistent. Alternatively, deployment changes done at application level are only applied for the current deployment.

Global Configuration Settings
The following properties can be made to change globally:

- **Intermediate Files**: During Deployment of EJB Applications, various intermediary Java files are generated (for example, bean home implementation classes). These files are normally not required by the container as their compiled versions are made available in the container’s classpath. The generated Java files are at times required for debugging purposes. The deploy service can be configured to retain these files. These files are available in the <install_dir>/server/nodes/<node_name>/archives/<archive_name>/java directory.
• **Validation of Application Archives**: Applications can be validated before deployment. This can be enabled or disabled for applications. Validation of archives during deployment involves DTD validation, and this is done for all kinds of archives like .war, .ear, and .rar. The validation happens for specification compliance of the deployment descriptors, internal consistency of the deployment descriptors, and so on.

• **Validation of Applications at startup**: Application can be validated at startup. This can be enabled or disabled for applications. Application validation involves checking if the mappings involved in the deployment descriptors are in sync with the current server environment. For example, datasources as a resource reference in the application.

Global properties can be set by clicking on the **Settings** icon on the Toolbar of the Deploy Tool:

---

**Archive Specific Customization**

Changes made at the archive level are specific for the current application considered for deployment. However, these changes are not persisted. The following properties can be set:

• **Retain generated files**: Intermediary (.java) files are retained by the container on deployment. These are used for debugging purposes.

• **Enable dynamic proxies for EJBs**: Dynamic Proxies, if used, avoid code generation and RMI calls. The changes made to this property are updated in the `pramati-j2ee-server.xml` file.
• **Force deployment:** If the application is already deployed, this option forces a deployment and overwrites all classes that existed in the previous deployment.

• **Application versioning:** The Deploy Service provides versioning for applications in the archives directory. The first version is created when the application is deployed for the first time. All subsequent versions are deployed into different directories. Information on the latest deployed version and the version to be used is maintained here. Any previous version can be started. A failed re-deployment would always mean there is a valid version to fall back on.

• **Application validation at startup:** This option enables validation of applications at startup. Validation of archives during deployment involves DTD validation, and this is done for all kinds of archives like .war, .ear, and .rar. The validation happens for specification compliance of the deployment descriptors, internal consistency of the deployment descriptors, and so on.

• **Application validation before application startup:** Application archives can be validated at deployment time. This can be enabled by turning this option on. Application validation involves checking if the mappings involved in the deployment descriptors are in sync with the current server environment. For example datasources as a resource reference in the application.

Archive specific deployment properties can be set by clicking **Deployment Properties** in the Explore panel of the application archive.
Note: Deployment changes made at the Archive level take precedence over the Deployment Settings.

Specifying Deployment Setting in Pramati Deployment Descriptor

Another way to specify archive specific deployment properties is to add an entry in the pramati-j2ee-server.xml file. A sample deployment property can be set as below:

```xml
<deployment-properties app-versioning="true" delete-previous-versions="true" forced-deployment="true">
  <jsp-files pre-compilation="true"/>
  <ejb-files retain-generated-code="true"/>
  <validation>
    <dtd-validation on-prepare="false" on-start="false"/>
    <app-validation on-prepare="true" on-start="true"/>
  </validation>
  <deployment-failure retain-files="true"/>
</deployment-properties>
```

Setting EJB Properties Using Deploy Tool

The Deploy Tool allows setting of various properties for Enterprise Java Beans at the archive level. Changes made here are made in Pramati’s deployment descriptors: pramati-or-map.xml and pramati-j2ee-server.xml. These properties are:

- An option to load data from the databases at the beginning of a transaction for CMP beans. This ensures that the bean is updated with fresh data prior to a transaction. This is valid for all levels of isolation, except for repeatable reads where database sharing is non-exclusive.
- An option for setting timeout for concurrent requests on the same Primary Key for CMPs. This timeout value is specified in milliseconds.
- A system property that indicates whether or not a BMP 2.0 bean can be locked on the primary key.

When an archive is opened using the Deploy Tool, the Explore panel on the left displays various components in the application archives. Clicking the EJB jar node expands the tree and lists all the beans created in this archive.

Setting Properties for Stateful Session Beans

For a stateful session bean, the Deploy Tool displays the following information on the right hand panel.
General
The General tab displays the following information:

Bean Information: This section contains general information about:

- EJB Name
- Bean Type
- Home Interface
- Remote Interface
- Local Home Interface
- Local Interface

Deployment Information: The deployment information contains the following:

- JNDI Name
- Local JNDI Name
- Max Pool Size
- Minimum Pool Size
- Low Activity Interval
- Pool Wait Timeout
- Enable Free Pool

Bean Description: A general description about the bean. The bean description can be specified here.

Method Information
This tab displays a list of methods that can be selected to alter a bean’s state.

Setting Properties for Stateless Session Beans
For a stateless session bean, the Deploy Tool displays the following information on the right panel:

Bean Information: This section contains general information about:

- EJB Name
- Bean Type
- Home Interface
- Remote Interface
- Local Home Interface
- Local Interface

Deployment Information: The deployment information contains the following:

- JNDI Name
- Local JNDI Name
- Max Pool Size
- Minimum Pool Size
- Low Activity Interval
- Pool Wait Timeout
- Enable Free Pool

*Bean Description:* A general description about the bean. The bean description can be specified here.
Setting Properties for Entity Beans (Container Managed)
For a Container Managed Entity Bean, the Deploy Tool displays the following information on the right panel, in the form of tabs.

General
The General tab displays the following information:

Bean Information: This section contains general information about:
- EJB Name
- Bean Type
- Home Interface
- Remote Interface
- Local Home Interface
- Local Interface

Deployment Information: The deployment information contains the following:
- JNDI Name
- Local JNDI Name
- Max Pool Size
- Minimum Pool Size
- Low Activity Interval
- Pool Wait Timeout
- Enable Free Pool

Bean Description: A general description about the bean. The bean description can be specified here.

Concurrency
The Concurrency tab contains information about the concurrency policy that needs to be assigned to the selected CMP.

CMF Fields
The CMF Fields tab contains information about the O-R mapping for a particular CMP bean. For a given datasource, the associated schema is displayed in the drop-down list, and the Container Managed Fields table contains the mapping for the CMP field and the corresponding column name in the table.

Finder Queries
The Finder Queries tab contains mapping for a finder method to a corresponding query label. New queries can be created by clicking New at the lower right hand of the window. Clicking New displays a query wizard. Using the Query Wizard, you can interactively create a query without having to manually code it.
Read “Writing SQL Queries for EJBs” on page 63 for more details about the Query Wizard.

**Setting Properties for Message Driven Beans**
For a message driven bean, clicking on the Bean Properties on the explore panel on the left, displays the following tabs on the right hand panel:

**General:**
The General tab displays all information pertaining to a particular MDB.

*Bean Information:* This section contains general information about:
- Bean Name: The name of the Message Driven Beans. This is a drop-down list of all the MDBs in the current archive.
- Bean Class Name: The name of the Message Driven Beans. This is a drop-down list of all the MDBs in the current archive.
- Transaction Type
- Transaction Attribute

*Bean Description:* A general description about the bean. The bean description can be specified here.

**MDBProperties:**
The MDBProperties tab displays the following information:

*Connector Properties:* This section displays the following connector properties:
- Message Listener Interface
- Resource Adapter
- Activation Spec
- A table displaying the various properties, Java types, and their values. You can select and modify these values.

**Migrating Local JNDI Names**
In previous service packs of Pramati Server 4.1, for beans that have local home references, the container generated a local JNDI name and bound local home with this name in Naming Service.

If there was a local reference to this bean in `pramati-j2ee-server.xml` corresponding to this local-ref mapping, the link-name tag used to have the remote JNDI name. The container internally used to resolve this to the actual local JNDI name before it bound the reference.

From Pramati Server 4.1 service pack 5, this assumption is removed from the container. It is now mandatory for local-reference mappings to be present in the `pramati-j2ee-server.xml`. Absence of local home references leaves a task remaining alert at the lower right-hand corner of the Deploy Tool. Pramati Deploy Tool now has a migration utility that does this for older archives.
**Deploy Tool Logs**

Pramati Server Deploy Tool logs messages that are generated while deploying applications. These messages can be helpful while debugging or troubleshooting. By default, the log level for Deploy Tool is ‘severe’. The logs can be viewed by selecting **View > View Logs** from the Deploy Tool menu.

When Deploy Tool is used locally, the logs are generated in the `<install_dir>/server/nodes/<node_name>/logs` directory, with a name `DeployTool.log`. If the Deploy Tool is used for deploying application on a remote machine, the Deploy Tool logs are generated in the `<current_directory>/Temp/DeployTool/logs` directory.

Custom tracing can be provided to the Deploy Tool. This is mainly used in cases when you want to know specifically the path taken by the Deploy Tool at the time of deploying the application.

For example, you can specify logging information to be displayed when the Deploy Tool invokes specific packages of the Server. Again, this is mainly used for debugging purposes. A template of the trace properties can be found at `<install_dir>/server/templates/consoletrace.props`.

The path to the trace properties file has to be provided at the time of starting the Deploy Tool, with the following command:

```
j2eadmin>deploytool -tracefile path/to/consoletrace.props
```

The following is a sample `consoletrace.props`:

```
handlers = first
handler.first.type = file
handler.first.details = trace.log
handler.first.loglevel = all
handler.first.contexttype.packages = .,false;web,true;
web.servlet.core.DefaultHttpTask, and packages ejb, & services.naming
```

Trace properties are explained in the table below:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>handlers</td>
<td>This is just a place holder for the different handlers that are set up. All handlers that are listed specified as a comma separated list. There can be a number of handlers available doing various types of tracing according to individual settings, all of them working simultaneously.</td>
</tr>
<tr>
<td>handler.custom_handler.type</td>
<td>Tracing can happen either to a user specified file (“file”) or to the console (“console”).</td>
</tr>
<tr>
<td>handler.custom_handler.details</td>
<td>If the handler.type had been set to “file”, then the filename must be specified here.</td>
</tr>
<tr>
<td>handler.custom_handler.isAbsolutePath</td>
<td>Set this to true if the path given above is an absolute path, and false otherwise.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>handler.custom_handler.appendIfFileExists</strong></td>
<td>This can be either true or false. If true, trace methods are appended at the end of the file, if it exists. If set to false, the trace file is overwritten.</td>
</tr>
<tr>
<td><strong>handler.custom_handler.loglevel</strong></td>
<td>Loglevels can be specified here. The following log levels are available: event, debug, off, info, user, finer, severe, warning, all.</td>
</tr>
<tr>
<td><strong>handler.custom_handler.contexttype.packages</strong></td>
<td>All relevant trace methods that occur in a given package are logged in the trace file. The connotation used is: &lt;package name&gt;,&lt;true/false&gt;;&lt;packagename1&gt;, true/false. Where package name is the name of the package to be traced (a ‘.’ (dot) denotes all packages), true/false is a boolean value associated with the package. The trace is generated if it is true. If set to false the package is ignored.</td>
</tr>
<tr>
<td><strong>enable.debug</strong></td>
<td>Set this to true if debug messages must appear while tracing. Valid values are true or false.</td>
</tr>
</tbody>
</table>
Validation Conditions

Validation enables application deployers to trap errors much before it propagates through the runtime environment.

The Validator validates application modules (.ear or .jar), and is a checkpoint before starting an application. The static validation which is done to validate the application module checks for the readiness of the module for deployment. It checks for things that the appserver assumes the bean provider takes care of, but probably may have missed due to some factors.

The validation is as per the EJB 1.1 specification, specifically the bean provider responsibility section. The various cases listed based on categories accompanied with the error messages are:

**Basic Jar Level Validation**

*Table 1: Basic jar level validation*

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO EJB-JAR.XML</td>
<td>ejb-jar.xml not found.</td>
</tr>
<tr>
<td>UNIQUE EJB NAMES</td>
<td>Cant have Duplicate ejb names in the same application.</td>
</tr>
<tr>
<td>Assembly Descriptor: Bean Name listed here exists</td>
<td>The assembly descriptor lists a bean which could not be found in the jar - Name.</td>
</tr>
<tr>
<td>Assembly Descriptor: Security role listed here exists</td>
<td>The assembly descriptor lists a security role which could not be found in the roles defined in the jar.</td>
</tr>
<tr>
<td>Assembly Descriptor: Method permission method exists</td>
<td>The assembly descriptor declares permission for a method which does not exist in the bean. Method details.</td>
</tr>
<tr>
<td>Assembly Descriptor: Method declared for container transaction exists</td>
<td>The assembly Descriptor declares container transaction for a method which could not be found on the bean. Method details.</td>
</tr>
</tbody>
</table>
## Basic Bean Class Level Validation

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public Constructor with no arguments</td>
<td>BeanName: Does not have a public constructor with no arguments.</td>
</tr>
<tr>
<td>Class Exists (Bean Class, Home, Remote)</td>
<td>BeanName: Could not load Class (No further validation done on the bean.)</td>
</tr>
<tr>
<td>Could not load Support Class</td>
<td>Bean Name: Could not load Support Class: Support Class Name.</td>
</tr>
<tr>
<td>Class Abstract</td>
<td>BeanName: Bean Class cannot be abstract.(except CMP 2.0).</td>
</tr>
<tr>
<td>Could not load Class fields</td>
<td>BeanName: Could not load Class field: Class Field.</td>
</tr>
<tr>
<td>Class Final</td>
<td>BeanName: Bean Class cannot be declared final.</td>
</tr>
<tr>
<td>Bean Class: Must be Public</td>
<td>BeanName: Bean Class must be declared public.</td>
</tr>
<tr>
<td>Bean Class: Static fields but not final</td>
<td>BeanName: All static fields have to be declared as final.: Field Name.</td>
</tr>
<tr>
<td>Environment Entries Invalid</td>
<td>BeanName: Environment entry has invalid value: Entry type, Value.</td>
</tr>
<tr>
<td>Bean Class: Finalize method</td>
<td>BeanName: Finalize method cannot be declared in the bean.</td>
</tr>
<tr>
<td>Home Interface: extends EJBHOME</td>
<td>BeanName: Home Interface does not extend EJBHOME (No further validation done).</td>
</tr>
<tr>
<td>Home Interface: Public</td>
<td>BeanName: Home Interface must be declared public.</td>
</tr>
<tr>
<td>Home Interface: Corresponding Methods</td>
<td>BeanName: Corresponding method for method &lt;&gt; not found.</td>
</tr>
<tr>
<td>Remote Interface: extends EJBOBJECT</td>
<td>BeanName: Remote Interface does not extend EJBOBJECT (No further validation done).</td>
</tr>
<tr>
<td>Remote Interface: Public</td>
<td>BeanName: Remote Interface must be declared public.</td>
</tr>
<tr>
<td>Method declared in DD present on the remote interface</td>
<td>BeanName: No method defined in the remote interface with the details: Details.</td>
</tr>
<tr>
<td>Create Method Exception</td>
<td>BeanName: Create Method does not throw Create/Remote Exception.</td>
</tr>
<tr>
<td>Remote Interface Corresponding Bean Class Methods</td>
<td>BeanName: No Cores method in bean class for: Remote Method Details.</td>
</tr>
<tr>
<td>Business Method: Not final</td>
<td>BeanName: Business Method cannot be declared final: Method Details.</td>
</tr>
<tr>
<td>Business Method: not public</td>
<td>BeanName: Business Method must be declared Public: Method Details.</td>
</tr>
<tr>
<td>Business Method: not static</td>
<td>BeanName: Business Method cannot be declared Static: Method Details.</td>
</tr>
<tr>
<td>Business Method: Valid Parameter types</td>
<td>BeanName: Invalid RMI-IIOP param type: Param Details.</td>
</tr>
<tr>
<td>Business Method: Valid Names</td>
<td>BeanName: Business Method names cannot start with ejb.</td>
</tr>
<tr>
<td>Business Method: Valid Return types</td>
<td>BeanName: Invalid RMI-IIOP return type: Return type Details.</td>
</tr>
</tbody>
</table>
### Table 2: Basic bean class level validation

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP: Container Managed Fields Valid or not.</td>
<td>BeanName: Invalid field declared as CMF must be primitive, serializable, home./remote.</td>
</tr>
<tr>
<td>BMP: Persistent fields declared</td>
<td>Bean Name: Persistent fields cannot be declared for BMPs</td>
</tr>
<tr>
<td>CMP: at least one CMP field</td>
<td>BeanName: CMP bean must have at least one CM field</td>
</tr>
<tr>
<td>ejbcreate: Not final</td>
<td>Bean Name: ejbcreate must not be final</td>
</tr>
<tr>
<td>ejbcreate: Public</td>
<td>BeanName: ejbcreate must be public</td>
</tr>
<tr>
<td>ejbcreate: Not static</td>
<td>BeanName: ejbcreate must not be static</td>
</tr>
<tr>
<td>ejbCreate: Return type</td>
<td>Bean Name: ejbCreate must return PrimaryKey class.</td>
</tr>
<tr>
<td>ejbCreate: parameter type</td>
<td>Bean Name: ejbCreate must have valid parameter type.</td>
</tr>
<tr>
<td>ejbfndbyPK: Not static</td>
<td>BeanName: ejbfndbyPK must not be static</td>
</tr>
<tr>
<td>ejbfndbyPK: Not final</td>
<td>BeanName: ejbfndbyPK must not be final</td>
</tr>
<tr>
<td>ejbfndbyPK: Public</td>
<td>BeanName: ejbfndbyPK must be public</td>
</tr>
<tr>
<td>ejbfndbyPK: valid return type</td>
<td>BeanName: ejbfndbyPK must have valid return type: return type details</td>
</tr>
<tr>
<td>ejbfndbyPK: valid parameter types</td>
<td>BeanName: ejbfndbyPk must have valid param types: param details</td>
</tr>
<tr>
<td>ejbfndbyPK: Return type</td>
<td>BeanName: ejbfndbyPk must return primary Key Class.</td>
</tr>
<tr>
<td>PKClass: Constructor with no params</td>
<td>BeanName: PKClass must have a constructor with no params</td>
</tr>
<tr>
<td>PKClass: Public</td>
<td>BeanName: PKClass must be declared public</td>
</tr>
</tbody>
</table>

### Validating Entity Beans

### Table 3: Validating entity beans

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Method: Remote Exception</td>
<td>BeanName: Business Method does not throw Remote Exception: Method Detail.</td>
</tr>
<tr>
<td>Method Permission Element: Method Present</td>
<td>BeanName: Method does not exist but declared for permissions.</td>
</tr>
<tr>
<td>Create Method: Return Type Remote Interface</td>
<td>Bean Name: Create method must return remote interface.</td>
</tr>
<tr>
<td>Create Method: Matching ejbCreate Method.</td>
<td>BeanName: Create method has no matching ejbCreate Method.</td>
</tr>
<tr>
<td>Home interface Methods: valid Parameter Types</td>
<td>BeanName: Invalid RMI-IIOp param type: Param Details.</td>
</tr>
<tr>
<td>Home interface Methods: valid Return Types</td>
<td>BeanName: Invalid RMI-IIOp return type: Return Details.</td>
</tr>
<tr>
<td>Assembly Descriptor: Method Permissions - Method Exists</td>
<td>BeanName: Method listed in Method permissions does not exist in the bean.</td>
</tr>
<tr>
<td>Validation Condition</td>
<td>Error Messages</td>
</tr>
<tr>
<td>----------------------------------------------------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>PKClass: Valid RMI-IIOP type</td>
<td>BeanName: PKClass must be a valid RMI-IIOP type</td>
</tr>
<tr>
<td>CMP: PKClass: all fields within CMFs</td>
<td>BeanName: PKClass must be defined within CMFs</td>
</tr>
<tr>
<td>CMP: CMFs within bean fields</td>
<td>BeanName: CMF not found in bean</td>
</tr>
<tr>
<td>CMP: CMFs not transient</td>
<td>BeanName: CMFs must not be declared as transient</td>
</tr>
<tr>
<td>PKClass: must override equals and hashcode</td>
<td>BeanName: PKClass must override equals and hashcode</td>
</tr>
<tr>
<td>MultiObject Finders: No ObjectNotFoundException</td>
<td>BeanName: MultiObjectFinders must not throw ObjectNotFoundException</td>
</tr>
<tr>
<td>CMP: Persistent fields valid</td>
<td>BeanName: Invalid persistent field declared: Field details</td>
</tr>
<tr>
<td>ejbpostcreate - ejbcreate: match params</td>
<td>BeanName: postcreate and create don't match</td>
</tr>
<tr>
<td>Home findByPK: at least one</td>
<td>BeanName: there must be at least one findByPK on the home interface</td>
</tr>
<tr>
<td>Home findByPK: Param Type PKClass</td>
<td>BeanName: the return type of findByPK must be of type PKClass of the bean</td>
</tr>
<tr>
<td>ejbFinders: Not Static</td>
<td>BeanName: EjbFinder methods cannot be declared static</td>
</tr>
<tr>
<td>ejbFinders: Not final</td>
<td>BeanName: EjbFinder methods cannot be declared final</td>
</tr>
<tr>
<td>ejbFinders: Public</td>
<td>BeanName: EjbFinder methods must be declared public</td>
</tr>
<tr>
<td>ejbFinders: valid parameter types.</td>
<td>BeanName: EjbFinder methods must have valid parameter types</td>
</tr>
<tr>
<td>ejbFinders: Return type</td>
<td>BeanName: EjbFinder methods must return primary key or collection of PKs</td>
</tr>
<tr>
<td>HomeFinders: Return type</td>
<td>BeanName: Finder Method must return Remote Interface</td>
</tr>
<tr>
<td>HomeFinders: Finder Exception</td>
<td>BeanName: Finder methods must throw finder exception</td>
</tr>
<tr>
<td>HomeFinders: corresponding ejbFinder method</td>
<td>BeanName: No corresponding ejbFinder method found: finder method details</td>
</tr>
<tr>
<td>HomeFinders: Remote Exception</td>
<td>BeanName: Finder method must throw RemoteException</td>
</tr>
<tr>
<td>ejbPostCreate: Matching create methods</td>
<td>BeanName: all ejbcreate methods must have matching ejbPostCreateMethods</td>
</tr>
<tr>
<td>ejbPostCreate: Not Final</td>
<td>BeanName: ejbPostCreate Methods cannot be declared final</td>
</tr>
<tr>
<td>ejbPostCreate: Not Static</td>
<td>BeanName: ejbPostCreate Methods cannot be declared static</td>
</tr>
<tr>
<td>ejbPostCreate: Return type</td>
<td>BeanName: ejbPostCreate Methods must return void</td>
</tr>
<tr>
<td>ejbPostCreate: Public</td>
<td>BeanName: ejbPostCreate Methods must be declared public</td>
</tr>
<tr>
<td>Home Interface Methods:</td>
<td>BeanName: Home Interface must be only create or finder methods</td>
</tr>
<tr>
<td>Transaction attributes on remote i/f</td>
<td></td>
</tr>
<tr>
<td>*Home Intf: each method corresponding method pre-fix:</td>
<td>BeanName: Home Method &lt;&gt; does not have corresponding method</td>
</tr>
<tr>
<td>ejbHome</td>
<td></td>
</tr>
</tbody>
</table>
Table 3: Validating entity beans

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>Interface</td>
</tr>
<tr>
<td>Interface</td>
<td>Interface</td>
</tr>
</tbody>
</table>

EJB 2.0 Specific Validation

Table 4: Validating Ejb 2.0

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Home i/f</td>
<td>BeanName: Local Home class could not be loaded</td>
</tr>
<tr>
<td>LocalHome: Interface ExtendsEJBLocalHome</td>
<td>BeanName: LocalHome does not extend EJBLocalHome</td>
</tr>
<tr>
<td>LocalHome: InterfacePublic</td>
<td>BeanName: LocalHomeInterface must be declared public</td>
</tr>
<tr>
<td>LocalHomeMethodModifiers</td>
<td>Bean Name: Local Home methods must be public and not static</td>
</tr>
<tr>
<td>LocalHome: MethodNameMatch</td>
<td>Bean Name: Local Home methods do not match with the Bean class</td>
</tr>
<tr>
<td>Local I/f: Public</td>
<td>BeanName: Local Component Interface not declared public</td>
</tr>
<tr>
<td>Local I/f: Extends EJBLocalObject</td>
<td>BeanName: Local Component i/f does not extend EJBLocalObject</td>
</tr>
<tr>
<td>Local I/f: Corresponding methods on the bean class</td>
<td>BeanName: Local i/f methods do not corresponding to the bean class</td>
</tr>
<tr>
<td>CMP 2.0: Cascade Delete not supported for Many - Many relation</td>
<td>BeanName: Many - Many relation cannot specify cascade - delete</td>
</tr>
<tr>
<td>CMP 2.0: CMF field declaration: Accessor methods Upper case get/set</td>
<td>BeanName: Invalid Container Managed Field Accessor</td>
</tr>
<tr>
<td>CMP 2.0: CMFs Public or Protected and Abstract</td>
<td>BeanName: CMFs must be declared abstract</td>
</tr>
<tr>
<td>CMP 2.0: CMF starts with lower case</td>
<td>BeanName: CMFs must start with lower case</td>
</tr>
<tr>
<td>CMP 2.0: CMF valid</td>
<td>BeanName: CMF must be primitive/serializable/remote/home reference</td>
</tr>
<tr>
<td>CMP 2.0: CMF declared in bean</td>
<td>BeanName: CMF must not be declared in the Bean Class</td>
</tr>
<tr>
<td>CMP 2.0: CMR declaration: Accessor methods Upper case get/set</td>
<td>BeanName: Invalid Container Managed Relation Accessor</td>
</tr>
<tr>
<td>CMP 2.0: CMRs Accessors public /protected and Abstract</td>
<td>BeanName: CMR accessors must be declared public/protected and abstract</td>
</tr>
<tr>
<td>CMP 2.0: CMRs must start with lower case</td>
<td>BeanName: CMRs must start with lower case</td>
</tr>
<tr>
<td>CMP 2.0: CMR Local Interface for One - One Collection for One -Many or Many - Many</td>
<td>BeanName: Invalid CMR declaration</td>
</tr>
<tr>
<td>CMP 2.0: CMR fields Virtual</td>
<td>BeanName: CMR accessors must be declared virtual</td>
</tr>
</tbody>
</table>
Validating MessageDriven Beans

Table 4: Validating Ejb 2.0

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP 2.0: CMR Multiplicity</td>
<td>BeanName: CMR Type - EJBLocalObject for One - One or Many - One Collection/Set for One - Many or Many - Many</td>
</tr>
<tr>
<td>EjbFinders: abstract</td>
<td>BeanName: Finder method must not be declared in the bean class</td>
</tr>
<tr>
<td>EjbSelect: throws FinderException</td>
<td>BeanName: Select method must throw FinderException</td>
</tr>
<tr>
<td>EjbSelect: Public and abstract</td>
<td>BeanName: Select method must be abstract and public</td>
</tr>
<tr>
<td>EjbSelect: QL association</td>
<td>BeanName: ejbSelect method not associated with QL</td>
</tr>
<tr>
<td>EjbSelect: Param Length and param matching</td>
<td>BeanName: ejbSelect Method params declared in the BeanClass and those declared in the ejb-jar.xml do not match</td>
</tr>
</tbody>
</table>

Table 5: MDB Validation

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDB: Implements MessageDriven Bean Interface</td>
<td>BeanName: Does not implement javax.ejb.MessageDrivenBean interface.</td>
</tr>
<tr>
<td>MDB: Implements MessageListener Interface</td>
<td>Bean Name: Does not implement the javax.jms.MessageListener</td>
</tr>
<tr>
<td>ejbCreate Method: Return type.</td>
<td>BeanName: Return type must be void.</td>
</tr>
<tr>
<td>ejbCreate Public</td>
<td>BeanName: ejbCreate must be declared public.</td>
</tr>
<tr>
<td>ejbCreate: Not Final</td>
<td>BeanName: ejbCreate must not be declared final.</td>
</tr>
<tr>
<td>ejbCreate: Not Static</td>
<td>BeanName: ejbCreate must not be declared static.</td>
</tr>
<tr>
<td>ejbCreate: Return type</td>
<td>BeanName: ejbCreate must return void.</td>
</tr>
<tr>
<td>ejbRemove: Public, Not Final and Not Static</td>
<td>BeanName: ejbRemove not declared as required.</td>
</tr>
<tr>
<td>MDB: Transaction attribute</td>
<td>BeanName: MDB can have either Required/NotSupported as Transaction attribute.</td>
</tr>
<tr>
<td>MDB: Selector</td>
<td>BeanName: The Message Selector for the MDB must be valid.</td>
</tr>
</tbody>
</table>
Validating Session Beans

Table 6: Validating session beans

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implements Session Bean Interface</td>
<td>BeanName: Does not implement javax.ejb.SessionBean interface</td>
</tr>
<tr>
<td>Create Method</td>
<td>BeanName: No create method defined on the home interface</td>
</tr>
<tr>
<td>ejbCreate Method: Return type.</td>
<td>BeanName: Return type must be void</td>
</tr>
<tr>
<td>Bean Class: No params Constructor</td>
<td>Bean Name: Bean class does not define a no-params public constructor</td>
</tr>
<tr>
<td>Bean Class: Not abstract</td>
<td>Bean Name: Bean class cannot be abstract</td>
</tr>
<tr>
<td>SLBean Create Method</td>
<td>BeanName: More than one create method cannot be declared on home</td>
</tr>
<tr>
<td>SLBean Create Method Parameters</td>
<td>BeanName: SL create method cannot take parameters</td>
</tr>
<tr>
<td>ejbCreate Public</td>
<td>BeanName: ejbCreate must be declared public</td>
</tr>
<tr>
<td>ejbCreate: Not Final</td>
<td>BeanName: ejbCreate must not be declared final</td>
</tr>
<tr>
<td>ejbCreate: Not Static</td>
<td>BeanName: ejbCreate must not be declared static</td>
</tr>
<tr>
<td>ejbCreate: Return type</td>
<td>BeanName: ejbCreate must return void</td>
</tr>
<tr>
<td>StatelessBean: SessionSync Interface</td>
<td>BeanName: SL Session beans cannot implement SessionSync interface</td>
</tr>
<tr>
<td>StatefulBean: Session Bean Interface</td>
<td>BeanName: SF Beans with BMTs must not implement SessionSync interface</td>
</tr>
<tr>
<td>- must have CMT</td>
<td></td>
</tr>
<tr>
<td>Stateful Bean: SessionSync Interface</td>
<td>BeanName: SF Bean implemented SessionSync interface can have required,</td>
</tr>
<tr>
<td>- Transaction Attribute</td>
<td>RequiresNew, or mandatory transaction attribute</td>
</tr>
<tr>
<td>*transaction attributes for all methods on the remote interface ++</td>
<td>BeanName: transaction attributes must be set for all methods on the remote interface except for the EJBOBJECT methods</td>
</tr>
<tr>
<td>*transaction attributes for all methods on the home interface ++</td>
<td>BeanName: transaction attributes must be set for all methods on the home interface</td>
</tr>
</tbody>
</table>

Validating Web Module

Table 7: Validating Web modules

<table>
<thead>
<tr>
<th>Validation Condition</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO WEB.XML</td>
<td>web.xml not found cannot continue.</td>
</tr>
<tr>
<td>Servlet Class:</td>
<td>Servlet class exists and is can be loaded.</td>
</tr>
<tr>
<td>Servlet class:</td>
<td>Servlet class extends the required class with reference to servlet specifications.</td>
</tr>
<tr>
<td>Environment entries</td>
<td>Invalid environment entry.</td>
</tr>
</tbody>
</table>
Validating Applications for Deployment on Pramati Server

The Deploy Service of Pramati Server is a core service started by the Pramati Server Kernel during startup, enabling applications to be deployed seamlessly in the Server. The services framework architecture delegates the application deployment process to the Deploy Service that takes care of reading archives and deploying them considering the archive properties.

Application Validation

The Deploy Service validates the archive before it is prepared for deployment. There can be two levels of validation:

Archive Validation: Verifying specification compliance of the archive. For example, `ejbCreate()` in a bean class must throw `CreateException`, CMP 2.0 bean class must be abstract.

Application Start Validation: Verifying of deployment information before starting the archive in the server environment. For example, checking the validity of the JNDI bindings, existence of the tables in the database for CMP Beans.

When the application validation fails, the Deploy Service does not let the application to be deployed.

Validating Archives

Validator is a tool that is shipped along with Pramati Server. You can use this tool to validate your archive based on certain pre-defined conditions. Severity levels are assigned against each validation condition. You can take corrective action depending on the message that appears against each validation condition. The levels are:

- Success - Validation is successful.
- Warning - There is a minor problem in the validation. However, the tool proceeds with the validation and you can deploy the archive.
- Failed - Validation has failed for that condition. You cannot deploy this archive until the condition is resolved.

You can validate an archive using the following methods:

- Using the Deploy Tool
- Offline Validation
- From the Command Shell
Validating Archives Using Deploy Tool

The Deploy Tool automatically validates an archive when you open it for deploying. To validate an archive using the Deploy Tool:

1. Start the Deploy Tool.
2. Select **Archive > Open** to open the required archive.

The Deploy Tool automatically validates the archive before opening it. After completion, the validation status appears on the screen.

You can also open an archive and select **Archive > Validate**.

The status and progress of the validation appears on the screen.

Validating Archives Offline Using Command Shell Interface

You can validate an archive independent of the Deploy Tool from the command shell. At the prompt, enter:

```
java com.pramati.tools.validator.earvalidator.EARValidator <name and location of the archive>
```

The archive is validated. Validation can be enabled or disabled when an archive is opened again. The options that you can specify along with this command are:

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>After the validation is complete, a report of all the severity levels is generated and displayed.</td>
</tr>
</tbody>
</table>
Validating Applications for Deployment on Pramati Server

Example:

```java
java com.pramati.tools.validator.earvalidator.EARValidator -a -oreport.txt orderApp.ear
```

Validating Archives From Remote Shell Prompt

For information about validating applications from the shell prompt, read “Deploying Applications Using Server Shell” on page 101.
Preparing Applications for Deployment

J2EE applications can consist of one or more of the following components:
- EJB components such as Entity beans, Session beans, and Message Driven Beans (MDBs)
- Web components such as HTMLs, Servlets, JSPs, and related files
- Connector components such as resource adapters

Before deploying an application, the following tasks have to be performed:
- Package J2EE applications
- Add external resources
- Perform O-R Mapping
- Generate queries
- Add security parameters
- Configure transactions
- Configure Message Server information if an application requires a messaging system

These tasks can be performed using the Console or through the command shell.
For information about using the Console, read the Pramati Server Administration Guide.
For information about using the Server Administration Service, read the *Pramati Server Installation and Configuration Guide*.

**Packaging Semantics for J2EE Application**

The J2EE architecture specifies a set of packaging requirements for enterprise applications that define a scalable and modular application assembly. This is very portable and can be deployed on any J2EE-compliant platform. Packaging makes the application server-agnostic.

A J2EE application is typically assembled as an archive of one or more J2EE components - Web application, servlets, JSPs, HTML pages, classes, and other resources. The archive can be composed of one or more J2EE modules, where each module represents a basic unit of application with a set of components of the same type. Information about modules is obtained from the *application.xml*. The flexibility and extensibility of the J2EE component model facilitates the packaging and deployment of J2EE components as individual components, component libraries, or applications.

**Packaging J2EE Components for Server**

A J2EE module that contains one or more J2EE components of the same component type, such as Web, EJB, or Connector, can be packaged together with a single deployment descriptor appropriate to that container type. The Deploy Tool reads the *ejb-jar.xml* and *web.xml* files for EJB and Web components, and the *ra.xml* for connectors. The packaged .ear must also contain the *application.xml* file. Refer to the appendix for more information about DTDs for these XMLs.

**Configuring Resources for Applications**

Pramati Server provides various resources to interact with databases, send mails, and encapsulate URLs. Resources are created on the server on which the J2EE applications are deployed. Resources provided by the Server are:

- JDBC connection factory for obtaining connections to a database
- Datasource cluster for providing auto failover from primary to secondary source
- Mail resource for sending mails from applications using the mail session object
- JMS adapter for enabling Java programs to create, send, receive, and read messages using the messaging system
- JCA Connection Factory for defining a set of system-level contracts between the server and resource adapter
- URL resource for obtaining URL connections

An application may also need security parameters such as realms, roles, and user names. These parameters can be set using the Console.
For more information about creating and managing resources, read Configuring Resources in the Pramati Server Administrator Guide.

**Generating Queries**
Queries are typically defined for EJBs with Container Managed Persistence (CMP). You can define multiple queries for a single data resource and driver.

You can define the following two types of queries:

- **EJB QL**: Enterprise JavaBeans Query Language, which is used to define queries for CMP2.0 beans
- **SQL**: Used to define queries for CMP1.1 beans

**Configuring Security Parameters**
Implementation of security involves configuring attributes that define the security policy for deployment. The Console enables you to define security parameters for the following elements:

- Security realms
- Users
- Access Control Lists (ACLs) and permissions for Server resources
- Secured Socket Layer (SSL) protocol
- Mutual authentication
- Host name verification
- Audit providers
- Custom filters
- Security context propagation

For more information on Server Security policy, read the Pramati Server Security Guide.

**Configuring Transaction Attributes**
The EJB architecture supports distributed transactions, which eliminates the need for developers to deal with complex issues of failure recovery and multi-user programming.

The transaction system ensures that a unit of work is either completed, or rolled back if incomplete. Transactions enable developers to design an application where units of work appear to execute serially.

You can use tools on the Console to configure the JavaTransaction API (JTA). The configuration process involves specifying the following attributes:

- Transaction timeouts and limits
- Transaction Manager behavior
For more information on Transactions, read *Managing Transactions* in the *Pramati Server Administrator Guide*.

**Configuring Pramati Message Server**

The Message Server enables Java applications to create, send, receive, and read messages. If an application requires a messaging system, configure the embedded Message Server in the Server using the Console. Pramati Message Server provides default values for some configuration attributes and the user provides values for the other attributes.

Use the Console to configure the following attributes of the Message Server:

- Destinations such as Queues and Topics
- Connection Factories
- Connections
- Persistence stores

For more information on Pramati Message Server, read *Configuring Embedded Message Server* in the *Pramati Server Administrator Guide*. 
Preparing and Deploying Web Applications

A Web application contains application resources such as servlets, JSPs and JSP tag libraries, and static resources such as HTML pages and image files. A Web application can also define links to resources outside the application such as EJBs.

Web applications are deployed on Pramati Web Container, which manages the execution of Web components. Server supports deployment of Servlets 2.4 and JSP 2.0 components.

When you install and start Server, the Web container is installed and started by default. You can configure a node in a cluster to be a pure Web node.

*Note: To view Web container details, select Monitor > Web Server in the Explore panel of the Console.*

Packaging Web Applications

Web applications can be deployed on Server as .war or .ear files. A .ear contains a .war that may contain the Web deployment descriptor, Web resources, utility classes, and supporting classes.

Deploying Web Applications

Web applications can be deployed on Server as standard .war or .ear files. Using the Deploy Tool, set properties such as resource references, resource environment references, and security role references to get applications in a ready-to-deploy state.

Running Web Applications

Web components such as JSPs, servlets, HTML pages, image files, and stylesheets can be deployed and run as applications. These applications can also be run in an exploded form.

Package these files as .war or .ear files and deploy them on Server. Application-related information is extracted to the document root. When a client makes a request, the application is accessed from this location.

An open structured application need not be packaged into .war or .ear files. You only need to copy the application files to the Server document root. You can also deploy exploded directories. Instead of copying, Pramati Server serves the application from the deployed directory.
Web Properties

Once you have opened an archive in the Deploy Tool, click the Web Properties node in the explore panel to set properties for the .war files. Web applications are handled according to the Servlet 2.4 specification, which describes a standard manner of grouping the various components of a Web-based application, such as JSPs, HTTP Servlets, HTML pages, and image files.

A Web application can also access external resources such as EJBs and JSP tag libraries. A server can host any number of Web applications.

General Properties

Click the General tab in the Web Properties node to open the General panel. This panel contains the following attributes:

- Web Archive: Displays the names of the Web archives. This field is not editable.
- Context Root: Contains the context root specified for the .war. The name of the .war is set as the context root, by default.

The context root must be unique for a virtual host in the Server namespace. The context root is the root under which the archive content is available. To modify the context root name, click the name and type the new context root name. To avoid using any context root while accessing the application, use the context root ‘/’. You can access the application with http://<host>:port.

Environment Properties

Click Environment tab in the Web Properties node to open the Environment Properties panel. You can assign environment values to defined environment entries. This enables applications to access resources and external information easily without requiring explicit knowledge of how external information is named or organized.

WAR Properties

Clicking on the .war node on the left panel displays the information corresponding to the tree node selected on the right window. This displays all the details of the Web components in the .war file like Name, Date, Time, Size, Ratio, Packed, Path, and context root of the .war.

EJB References

When you click the EJB References tab in the Web Properties node, the EJB References panel is displayed, which is used to declare references to enterprise beans. This panel contains the following attributes:

- Web Archive: Displays all .war files that are to be deployed in the application. This field is not editable.
- EJB Ref Name: Displays the EJB reference names that are to be mapped to Link or JNDI names.
• **Type:** Can be Remote or Local.

  Local references refer to the referred bean only through local interfaces. The referring Web component and the referred bean must be in the same VM. The drop-down list used to select the link name contains only local bean names that are available in the Server and the current application.

  Remote references refer to the referred bean only through the remote interface. The referring bean can be in a different VM. The drop-down list used to select the link name contains all the bean names and JNDI names both in Server and in the current application.

• **Link Name:** Provides the list of link names, which are used to look up the referred beans. The link name is the EJB name of the referred bean.

When you select the Web archive row in this panel, the EJB-ref type, the bean interfaces, and an optional description are displayed. The description field is editable.

**Resource References**

Click the **Resource Ref** tab in the Web Properties node to view the Resource Reference panel. All the defined resources are obtained from the Server.

This panel contains the following attributes:

• **Web Archive:** Displays all .war files that are to be deployed in the application. This field is not editable.

• **Resource Ref:** Displays the resource reference of the .war. It is usually in the format type/resourcename, where type represents the resource type and resourcename the name of the resource.

• **Defined Resource:** The available resources of the specified type are provided as a drop-down list. The Resource type is picked up automatically from the `web.xml` file. It includes resources that were directly bound on the server as well as resources of the same type from the deployed .rar files.

When you select a .war node, the resource type, the resource authorization, and an optional description are displayed.

The following types of resources can be directly mapped to the Server:

• `javax.sql.DataSource`
• `javax.jms.QueueConnectionFactory`
• `javax.jms.TopicConnectionFactory`
• `javax.mail.Session, java.net.URL`

If the resource is bound by a .rar, the type may be any other type representing the JCA connection factory class.
**Resource Environment References**
Click the Res Env Ref tab in the Web Properties node. This panel contains a declaration of the Web application’s reference to an administered object that is associated with a resource in the resource’s environment.

It contains the resource environment reference name and an indication of the resource environment reference type expected by the Web application code. For JMS, the resource environment reference can be selected from `javax.jms.Queue` and `javax.jms.Topic`. The drop-down list displays only resource environment type objects.

The deployer can select the value assigned to the .war’s resource environment reference.

**Security Role References**
The Security related views are available only if the selected application contains references to Security.

To open the Security panel, click the Security node in the Deploy Tool. Assign realm roles to the security roles defined in the wars. The available realm roles are obtained from the Security service. This panel contains the following attributes:

*Note: If you face a problem, check Monitor > Logs in the Console.*
Multiple Role Mapping

There might be a situation when you define a `<method-permission>` element in the `ejb-jar.xml`. The element specifies that one or more security roles are allowed to invoke one or more enterprise bean methods. The method-permission element consists of an optional description, a list of security role names or an indicator to state that the method is unchecked for authorization, and a list of method elements.

The security roles used in the method-permission element must be defined in the security-role elements of the deployment descriptor, and the methods must be methods defined in the enterprise bean's component and/or home interfaces.

From this version on, you can map roles to multiple realms using the Deploy Tool. To do so, use the following steps:

1. Start the Deploy Tool.
2. Select an archive using **Open**.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Realm</td>
<td>Specifies the realm name.</td>
</tr>
<tr>
<td>Module</td>
<td>Displays the WAR name or the application-level role name in the security role.</td>
</tr>
<tr>
<td>Role Name</td>
<td>Displays all the role names that have already been defined for the application.</td>
</tr>
<tr>
<td>Role Link</td>
<td>Specifies the name of the security role.</td>
</tr>
</tbody>
</table>
3 In the left-hand panel, click **Security**, and then select **Application Level Role Mapping**. The Deploy Tool UI looks like:

4 Select a **Role** to be mapped.

5 Assign the **Role** to an available Group. Use >> to move the **Group** to the **Selected** list.
6 Assign the Role to an available User. Use >> to move the User to the Selected list. The Deploy Tool UI looks like:

![Deploy Tool UI](image)

7 Click Save.

8 Perform the steps for each Role you wish to map. This can be done for the different realms available as a drop-down list by selecting the desired realms and roles.

9 Upon saving the archive, the changes made are reflected in the pramati-j2ee-server.xml:

```xml
<role-mapping>
    <module-name>ejb_secEbmp_sec_ejb.jar</module-name>
    <role-name>VP</role-name>
    <role-link>
        <group-principal>everybody</group-principal>
        <group-principal>administrator</group-principal>
        <user-principal>root</user-principal>
    </role-link>
</role-mapping>
```
Working With Servlets on Pramati Server

To run a servlet, package it in a .war file along with the helper classes, the Java classes used by the servlet. Deploy the .war file and access the servlet from the browser.

Running Servlets Without Packaging

Servlets can be run on the Server without packaging them into .war or .ear files. To do this, store the servlet class in the document root of the virtual host in the Server.


If the servlet has a package, then the same needs to be created under `web-inf/classes` directory.


Note: The location of the document root of the Server is specified in the web-config xml.

The Server classpath must be updated to contain all the helper classes that the servlet accesses.

The servlet helper classes are located at `<install_dir>/server/nodes/<node_name>/archives/public_html/servlet`.

Adding Servlets to Running Server

Servlets can be dynamically added to the running Server. When a request for a Servlet is made, the Server looks for the servlet in the document root's `web-inf/classes` directory of Server.

If the servlet is found, the Server returns the request, else it throws a file not found error.

Accessing Servlets

To access a servlet running on Server, specify `http://<host-name>:<port>/<servlet_name>`, as `http://localhost:8181/contextroot/<servlet_path>/<servlet_name>`.

Here, `host-name` is the name or IP of the machine on which Server is running, `port` is the port on which the Web container handles HTTP requests, `servlet_path` is the fully qualified path for the servlet class, and `servlet_name` is the name of the Servlet that is being accessed.

Example: If sample is the context root and `package1.package2.Test` is the fully qualified path for the Servlet class, the link is: `http://localhost:8181/sample/package1/package2/Test`.

Working With JSPs on Pramati Server

You can deploy JSPs that are packaged in a .war or .ear file, or deploy the JSP directly on the Server without packaging.
Packaging and Deploying JSPs on Server
You can package a JSP in a .war or .ear file and deploy it on an application server. After deployment, the application contents are extracted to the document root of the server. This application can then be accessed from this location.

Deploying JSPs on Server
Pramati Web Container provides a simple mechanism for running JSPs without having to first package them. The Web container provides an access area from where JSPs can be directly served. To do this, save the JSP file in the document root of the Server. The default location is 
<install_dir>/server/nodes/<node_name>/archives/public_html/.

Example: If abc.jsp is placed in public_html/games/ directory, it can be directly accessed using the following URL: http://localhost:8181/games/abc.jsp.

All class files used by the JSP are stored at:
<install_dir>/server/nodes/<node_name>/archives/public_html/classes

Hot Deployment
JSPs can be modified and accessed without stopping the application. To do this, deploy the application on Server.
Send a request for the JSP. Modify it, and send the request again. The changes are reflected.

Viewing Deployed Web Clients
When you deploy an archive on Server, the following structure and files are created:
• Archives: Is a directory created at <install_dir>/server/nodes/<node_name>, which contains all the deployed classes and files. <node_name> is the name of the configured node on which the application has been deployed.
• Web archives: Are extracted and stored at <install_dir>/server/nodes/<node_name>/archives/public_html/context_root in the appropriate package hierarchy. war_name is the name of the .war.

When you undeploy the application, all these files are removed from the specified location.

Understanding Pramati Server Web Logs
You can monitor Server activities by enabling the Web activity logging and analyzing this data to get useful information. Once the service is enabled, the Server:
• Initiates the logging process
• Logs the activities in a standard format
• Views and analyzes the log data
Logging Web Activity
When the Server starts, you can specify if the Web activities are to be logged. To log Web activities, open the `web-config.xml` file. The log file is located at `<install_dir>/server/nodes/<node_name>/config`.

In the `web-log` parent tag, set the value of `log-on` tag to `true`. For example:
```
<web-log>
    <log-on>true</log-on>
    <dir-path>$NODE_DIR\abc</dir-path>
</weblog>
```

*Note: You can set this option before starting the Server.*

Log Format Used by Pramati
Pramati Server uses the Apache Combined Log Format (CLF) to log Web activities.

The format string consists of percent directives which indicate the information that Server can log. Literal characters can be placed in the format string and are directly copied into the log output. A back-slash must precede the quote character to prevent it from being interpreted as the end of the format string. The format string may also contain the special control characters `\n` for new-line and `\t` for tab.

In Apache CLF, information is stored in the format string as: 

```
"%h %l %u %t "%r" %>s %b
"%{Referer}i"
"%{User-agent}i"
```

*Table 2: Values for Apache combined log format*

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>The IP address of the client (remote host) which made the request to the server.</td>
</tr>
<tr>
<td>l</td>
<td>Remote log name.</td>
</tr>
<tr>
<td>u</td>
<td>The userid of the person requesting for the document as determined by http authentication. The same value is typically provided to CGI scripts in the REMOTE_USER environment variable.</td>
</tr>
<tr>
<td>t</td>
<td>The time at which Server finished processing the request.</td>
</tr>
<tr>
<td>r</td>
<td>The request line from the client enclosed in double quotes. Contains information regarding the method used by the client, the name and type of resource, and the protocol used.</td>
</tr>
<tr>
<td>s</td>
<td>The status code sent back to the client by Server. Code beginning with 2 indicates that the request resulted in a successful response, code beginning with 3 indicates a redirection, code beginning with 4 indicates that an error has been caused by the client, codes beginning with 5 indicates an error in the server.</td>
</tr>
<tr>
<td>b</td>
<td>This indicates the size of the object returned to the client. This does not include the response headers.</td>
</tr>
<tr>
<td>(Referer)i</td>
<td>The &quot;Referer&quot; HTTP request header. This gives information about all the sites and files that the client has accessed.</td>
</tr>
<tr>
<td>(User agent)i</td>
<td>The User-Agent HTTP request header. The client browser identifies itself with this information.</td>
</tr>
</tbody>
</table>
For example, 127.0.0.1 - username [10/Mar/2002:13:55:36 -0700] "GET /pramati.gif HTTP/1.0" 200 2326 "http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I ;Nav)".

**Monitoring Web Activity Reports**

When logging of Web activity is initiated, a log file is generated. The information in this file is stored in the CLF.

The log file is located at the path specified while setting the value of log-on tag to `true`.

*Note: The contents are purged when the file size is big before logging new information in this file. This is taken care of by the operating system. The server automatically saves the contents of this file before purging.*

For example, 127.0.0.1 - username [10/Mar/2002:13:55:36 -0700] "GET /pramati.gif HTTP/1.0" 200 2326 "http://www.example.com/start.html" "Mozilla/4.08 [en] (Win98; I ;Nav)".

**Using Log Analyzers**

Log analyzers are used to analyze the log file information. Some analyzers available free or at a price are:

- [http://www.mrunix.net/webalizer](http://www.mrunix.net/webalizer) *(free)*
- [http://www.softjam.co.uk/acatalog/Log_Analysers.html](http://www.softjam.co.uk/acatalog/Log_Analysers.html) *(at a price)*

**Manual Logging of Web Activity Using Servlet APIs**

To log information regarding a specific activity being performed for an application, you can use standard Servlet APIs that allow a fuller control over what is to be logged.

The generic servlet class that has to be used for this purpose is `javax.servlet Class GenericServlet`.

To use the APIs, the `com.pramati.web.logging` property must be set to `true`. To set this property, while starting the Server, at the java prompt, type:

```
java -Dcom.pramati.web.logging=true com.pramati.Server -verbose
```

To log whatever information is passed into the Server Logs, you can call any of the following two methods:

```
public void log(java.lang.String message, java.lang.Throwable t)
```

This method writes an explanatory message and a stack trace for a given throwable exception to the servlet log file, prepended by the servlet's name.

**Table 3: Parameters for generating servlet logs**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>A string that describes the error or exception.</td>
</tr>
</tbody>
</table>
public void log(java.lang.String msg)

This method writes the specified message to a server log file, prepended by the servlet's name.

The log file gets saved at `<install_dir>/server/nodes/<node_name>/logs`.

**Enhanced Logging**

Deploy Tool now logs messages, errors, and debug statements. The logs can be viewed from View Menu > View Logs.

By default, the Deploy Tool's log file is located at `<user.dir>/Temp/DeployTool/logs/DeployToolTrace.log`. The location of the file can be changed by setting the system property `com.pramati.deploytooltracefile` to point to a properties file that contains the new location or by using the option `tracefile` command line to point to the `tracefile.props` containing the logging options.
Preparing and Deploying EJB Applications

EJB applications include EJB components like session beans, entity beans and MDB’s. EJB applications are deployed on Pramati EJB Container. Pramati Server supports deployment of both EJB 1.1 and 2.0 beans.

If EJB applications contain stateless, stateful, and Bean Managed Persistence (BMP) beans compliant to EJB 1.1 standards, they are automatically migrated to EJB 2.1 at the time of deployment.

This section discusses how to implement, assemble, deploy, and invoke EJB applications that are deployed on Pramati Server.

Packaging EJB Applications
EJB applications can be packaged as .jar or .ear files.

Deploying EJB Applications
Deploying EJB applications refers to making beans available to clients. EJB applications can be deployed on Pramati Server as either standard .jar or .ear files. To get an application to a ready-to-deploy state, you need to resolve EJB information with regard to Server environment such as resource references, resource environment references, concurrency, and security role references.

Configuring Bean Properties
The bean properties node facilitates specifying the properties of each bean. To set the bean properties using the Deploy Tool, click the bean properties node in the node tree to set the properties for each bean. This opens a panel on the right, which displays a table listing all the beans in the application with their properties.

Depending on the selected application, the attributes that can be set here are general properties, EJBs, resources, resource environment, sessions, bean pools, run as security information, JNDI name, and MDB destinations.
Configuring General Properties

The **General** properties panel appears by default when you click the Bean Properties node. The panel displays the following attributes:

- **JAR**: The name of the Java archive is displayed in this field. This field is not editable.
- **Bean**: The Bean name is displayed here. This field is not editable.
- **JNDI Names**: This field displays the JNDI names for the beans. JNDI names must be unique across the Server. The deploy tool provides JNDI names to the beans by automatically retrieving the EJB names from the file, `ejb-jar.xml`.

  If the EJB name is already bound, the name `ejb-name@jarname` is provided, where `ejb-name` represents the name of the Bean, and `jarname` represents the name of the JAR. In case the above name is also bound, the names provided are `ejb-name@jarname1`, `ejb-name@jarname2`, `ejb-name@jarname3` and so on such that the JNDI name provided is always unique. JNDI name can be modified by entering the value in the field.

  *Note: No JNDI names are provided for MDBs as they do not have a client view.*

Beans having remote interfaces are the only beans that have JNDI names.

Beans having local interfaces, have a local bean name and no look up can be performed on these. They are accessed using local EJB references.
The Deploy Tool does not allow a busy local bean name to be used as a JNDI name for other objects, saving the user from confusion. Similarly, a busy JNDI name cannot be used as a local name of a bean. Server uses internal naming conventions for local EJB bindings.

- **SSL**: Select or deselect the option to set the Secure Sockets Layer protocol as enabled or disabled. The SSL protocol provides authentication and message protection at the transport layer.

- **CLB**: Select or deselect the option to mark a bean if it is to be deployed in a clustered environment. This facilitates a transparent failover for non-idempotent beans.

### Configuring EJB

This panel is displayed when you click the **EJB** tab in the Bean Properties node. This panel lists the beans which refer other beans in the container. The attributes provided in this tab are:

<table>
<thead>
<tr>
<th>JAR</th>
<th>Bean</th>
<th>Reference Name</th>
<th>Type</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>bankEJB.jar</td>
<td>TransferBean</td>
<td>ejbMyChecking</td>
<td>remote</td>
<td>CheckingBean</td>
</tr>
<tr>
<td>bankEJB.jar</td>
<td>TransferBean</td>
<td>ejbMyLodger</td>
<td>remote</td>
<td>LodgerBean</td>
</tr>
</tbody>
</table>

- **JAR**: The name of the Java archive is displayed in this field. This field is not editable.
- **Bean**: The Bean name is displayed here. This field is not editable.
- **Reference Name**: This field displays the EJB reference names that are to be mapped to a link name. All the link names and the JNDI names bound on the Server and set in the General tab in the Bean properties node are available for mapping.
• **Type:** The EJB reference can be set as either remote or local. References of type local can refer to the referred bean only through the local interfaces; and both referred and referring bean must be in the same virtual machine. The drop-down list for selecting the link name contains only local bean names available in the server and in the current application. References of type remote refer to referred bean only through remote interface and the referring bean can be in a different virtual machine. The drop-down list for selecting link name contains all the bean, and JNDI names in the server as well as current application.

• **Link Name:** Select from the list of Link Names provided in the drop-down list. This becomes the local EJB name of the referred bean.

Selecting any of the beans row displays the EJB reference type, the interfaces of the bean such as LocalHome or RemoteHome and also an optional description in the panel.

### Configuring Resources

Clicking the **Resource** tab facilitates mapping of resource references onto the resources defined on the server. In case a new resource is added using the Management Console, it gets added to the running Server.

The attributes in this panel are:

- **JAR:** The name of the Java archive is displayed in this field. This field is not editable.
• **Bean**: The Bean name is displayed here. This field is not editable.

• **Reference Name**: This field displays the resource reference name. It is named as type and resourcename, where type represents the resource type and resourcename is the name of the resource.

• **Resource**: The resource type is automatically selected from the `ejb-jar.xml` file. The drop-down list displays the resources only of resource reference type. It includes resources directly bound to the server, and those from the .rars deployed on the Server.

Selecting a row displays the resource type, the resource authorization, and an optional description in the lower half of the panel.

### Configuring Resource Environment

Click the **Resource Environment** tab in the Bean Properties node. It displays details of an enterprise bean that refers to an administered object. This object is associated to a resource in the bean’s environment.

The panel contains the resource environment reference name, the resource environment reference type expected by the enterprise bean code, the bean name, and the archive which the bean belongs to. For JMS, the resource environment reference can be selected from the types `javax.jms.Queue` and `javax.jms.Topic`. The drop-down list displays only the objects of resource environment type.

The deployer can select the value assigned to the bean's resource environment reference.

### Configuring Sessions

The **Session** tab displays the following attributes:

• **JAR**: The name of the Java archive is displayed in this field. This field is not editable.

• **Bean**: The Bean name is displayed here. This field is not editable.

• **Timeout (min)**: This field represents the time for which the pool size is kept under observation, and then set for each bean.

After the specified time, if the actual usage of bean instances is below the minimum pool size, the pool size is automatically reduced to the minimum pool size. This provides an optimized use of bean pool.
Configuring Bean Pool Properties

The **Bean Pool** tab displays the following attributes:

- **JAR**: The name of the Java archive is displayed in this field. This field is not editable.
- **Bean**: The Bean name is displayed here. This field is not editable.
- **Max Size**: This is the maximum number of bean instances to be retained in the bean pool. The default size is 10.
- **Min Size**: This is the minimum number of bean instances to be retained in the bean pool. The default size is 0.
- **Low Activity Interval (s)**: Low Activity Interval (LAI) represents the time for which the pool size is kept under observation, and then set for each bean. The object–level caching size and the number of beans that can be maintained in the pool for reuse is determined by the LAI in seconds. After the specified time, if the actual usage of bean instances is below the minimum pool size, the pool size is automatically reduced to the minimum pool size. This provides an optimized use of bean pool.
• **Wait Time (s):** The wait time in seconds represents the time for which the pool size is kept under observation, and then set for each bean. After the specified time, if the actual usage of bean instances is below the minimum pool size, the pool size is automatically reduced to the minimum pool size. This provides an optimized use of bean pool.

• **Use Free Pool:** Select or deselect the option to set the free pool usage as enabled or disabled.

### Configuring Environment Properties

All the environment entries that can be accessed from the EJB code are displayed here. An environment entry is scoped to the bean. This means:

- The environment entry cannot be accessed from other enterprise beans at runtime.
- Other enterprise beans can use the same named environment entries without conflict.

The following fields are displayed in the Environment Properties panel:

#### EJB Environment

*Table 1: EJB Environment properties*

<table>
<thead>
<tr>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jar</td>
<td>The Java archive name is displayed here. This field is not editable.</td>
</tr>
<tr>
<td>Bean</td>
<td>The bean name is displayed in this field. This field is not editable.</td>
</tr>
<tr>
<td>Entry Name</td>
<td>Name of the environment entry.</td>
</tr>
<tr>
<td>Type</td>
<td>The Java type of environment entry.</td>
</tr>
<tr>
<td>Value</td>
<td>The value assigned for the environment entry. This field is editable.</td>
</tr>
</tbody>
</table>

#### Web Environment

*Table 2: Web environment properties*

<table>
<thead>
<tr>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>War</td>
<td>The Web archive name is displayed here. This field is not editable.</td>
</tr>
<tr>
<td>Bean</td>
<td>The bean name is displayed in this field. This field is not editable.</td>
</tr>
<tr>
<td>Entry Name</td>
<td>Name of the environment entry.</td>
</tr>
<tr>
<td>Type</td>
<td>Java type of environment entry.</td>
</tr>
<tr>
<td>Value</td>
<td>The value assigned for the environment entry. This field is editable.</td>
</tr>
</tbody>
</table>
Configuring Security Role References

The Security related views are available only if the selected application contains references to Security.

Click Security node in the Deploy Tool to bring up the Security panel. This enables assigning of realm roles to the security roles defined in the .jars. The available realm roles are obtained from the Security service started by the Server.

The attributes in this panel are:

Table 3: Attributes of Security Role References

<table>
<thead>
<tr>
<th>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Realm</td>
<td>This specifies the realm name. Select from the available realms in the drop-down list.</td>
</tr>
<tr>
<td>Module</td>
<td>This displays the JAR name, or the application level role name in the security role.</td>
</tr>
<tr>
<td>Role Name</td>
<td>This field displays all role names that have been already defined for the application. The management console may be used to define the security roles.</td>
</tr>
<tr>
<td>Role Link</td>
<td>Select from the role links available in the drop-down list whose value is the name of the security role.</td>
</tr>
</tbody>
</table>

Note: If there is a problem in obtaining the security service from the Server, check the log and Status window in the Management Console.
Run As Roles

Select the Run As tab in the Bean Properties node. The deployer uses this to map a user to the Run As role for a bean. The panel displays the following attributes:

- **JAR**: The Java archive names are displayed in the first field. This field is not editable.
- **Bean**: The bean names are displayed in this field. This field is not editable.
- **Run As**: This field displays all the run as role names of the bean.
- **User**: The drop-down list displays all the users of the actual role mapped to role name, corresponding to the run as role names.

*Note: This panel appears only if Security Roles have been added.*

Viewing Deployed EJBs

All configured server instances on the local machine are displayed under the nodes directory in the server installation directory. This directory also provides references that can be used for viewing deployed applications on these servers.

Deploying a jar or ear on Server creates the following directory structure:

- An archives directory is created at `<install_dir>/server/nodes/<node_name>` and contains all the deployed classes and files.
  
  Here, `<node_name>` is the name of the configured node on which the application has been deployed. Example: `c:/pramati_server/nodes/default/archives`, where default is the name of the configured server.

- The EJB classes deployed on Server are extracted and stored in: `<install_dir>/server/nodes/<node_name>/archives/archive_name/archive_version/classes` in the appropriate package hierarchy.
  
  Here, `<node_name>` is the name of the configured node on which the application has been deployed. The name of the archive deployed is represented by `archive_name`.

- The classes directory for an application is located under `<install_dir>/server/nodes/<node_name>/archives/archive_name`. The client jar required for java clients, for MyApp.ear is stored at `classes/MyAppClient.jar`.

- A directory for the jar files is created at `<install_dir>/server/nodes/<node_name>/archives/<archive_name>/<jar_name>`. It stores the `ejb-jar.xml` file, and also the `queries.props` file.
  
  Here, `<node_name>` is the name of the configured node on which the application has been deployed. The name of the archive deployed is represented by `<archive_name>`, and the name of the jar in the application by `<jar_name>`.
• The xml files - application.xml, pramati-j2ee-server.xml, web.xml, ejb-jar.xml, ra.xml, and pramati-or-map.xml are stored at <install_dir>/server/nodes/<node_name>/archives/<archive_name>. Here, <node_name> is the name of the configured node on which the application has been deployed. The name of the archive deployed is given by <archive_name>. Undeploying the application removes the above files from their respective locations.
Writing SQL Queries for EJBs

You can use the Query wizard to create SQL queries for CMP 1.1 entity beans. Queries are designed specific to datasource and driver. Multiple queries can be defined for a single datasource and driver.

In the **Deploy Tool** panel, click a CMP bean node. Select the **Finder Queries** tab and click **New**.

This brings up the **Query Wizard**, which helps generate queries.
Managing Database Connectivity

The first screen in the Query wizard displays the database you are connected to. Choose the data resource from the .dat resource name drop-down.

Enter the following details:

- **Data Resource Name**: Displays all the data resources connected to the Server. Select a name from the drop-down list provided.
- **Driver**: Displays the driver on which the query has been defined. This field is not editable.
- **URL**: Displays the URL. This field is not editable.
- **User Name**: Displays the user name if present. This field is not editable.
- **Password**: Displays the password if present. This field is not editable.

Click **Next**. This opens a panel that lists the tables to be queried.
Query Tables
The query name is used to map the finder methods when entity beans are defined or deployed. To add tables from the Available tables list, click \[\text{Add}\]. To delete tables from the Selected tables list, click \[\text{Remove}\].

Specify Output Fields
When you click Next after selecting the tables, a screen is displayed for specifying the output fields:

Select the mode as either Distinct or All, and modify the Variable Name using the editable field.
Click **Next**.

### Search Criteria

The Specify search criteria panel is used to create a search criterion for a query.

*Note: This is the first screen in the Query Wizard when it is invoked using Bean Wizard. The table and resource to be used is already selected.*

Click **Prev** to modify the following settings:

- **Field 1**: Specifies the database field name
- **Field 2**: Provides the following options as the second argument:
  - Field: Specifies the database field
  - Subquery: Selecting subquery activates a drop-down list using which you can select a subquery
  - Literal: Specifies a literal expression in the query
  - Parameter: Specifies a user-defined value in the query
- **Operator**: Provides a drop-down list with options such as:
  - Comparison: Includes all the comparative operators
  - Range: Includes BETWEEN and NOT BETWEEN
  - Pattern: Includes LIKE and NOT LIKE
  - List: Includes IN and NOT IN
  - Value: Includes IS NULL and IS NOT NULL

Specify an operator and add the defined search criterion to the WHERE clause table by clicking **Add**. When you add more than one clause, the first column in the displays an and/or toggle text. Click this text to join all the clauses.
To delete a search criterion, click **Delete**.

If the query contains a parameter that is provided at run time, the **Query Results** tab does not display the results.

**Output Fields and Sorted Query Results**

This panel displays the options for selecting the output fields and the sorting order for the query results.

This screen provides the following options:

- **All**: Displays all the occurrences in the output if database rows are returned several times by various search criteria
- **Distinct**: Displays only one database row in the output
- **Sort Result By**: Specifies the field on which to sort. Click a field name from the list and click the right arrow button to add it to the Field name table. The ascending/descending toggle sets the sort order.
Viewing Generated Queries
Click Finish to generate the query.

<table>
<thead>
<tr>
<th>Generated SQL query: findById</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT ACCOUNT.USERID, ACCOUNT.EMAIL, ACCOUNT.FIRSTNAME, ACCOUNT.LASTNAME, ACCOUNT.ST</td>
</tr>
<tr>
<td>FROM ACCOUNT</td>
</tr>
<tr>
<td>WHERE</td>
</tr>
<tr>
<td>ACCOUNT.USERID &gt; 101</td>
</tr>
<tr>
<td>ORDER BY ACCOUNT.LASTNAME ASC, ACCOUNT.FIRSTNAME ASC</td>
</tr>
</tbody>
</table>

Modifying SQL Queries
Return to the screen where you defined the query by clicking Prev and modify the query as required.

Query Viewer
To open Query Viewer, click a bean node in Deploy Tool, select the Finder Queries tab, and click Import. This displays the Designed Queries and the Generated SQL for the same as below:
Choose the name of the desired query from **Designed Queries**.

The query is displayed in the **Generated SQL**. It is a good idea to first define a set of queries that are likely to be used in finder methods. You can delete queries by clicking **Delete**.

**Importing Queries**

When you click **Import** in the Query Viewer panel, the Import Queries dialog box opens.

Browse and import queries from a remote `queries.props` into the local props file that contains the query you want to import. This is useful for transferring queries written earlier to a deployment site or while relocating .ears. Click **OK**. A query that is imported for the first time is displayed in blue in the Import Query dialog box along with the generated SQL statement. Duplicate queries cannot be imported into the Deploy Tool.

**Compiling EJB QL Queries**

EJB QL is used to define queries for CMP 2.0 entity beans. It is a query specification language for the finder and select methods of CMP beans.

EJB QL can be compiled to a target language of a database or other persistent store such as SQL. This enables you to move query execution to the native language facilities provided by the
persistent store instead of requiring queries to be executed on the runtime representation of the entity beans' state. Query methods can be made optimizable as well as portable.
Setting Up CMP Beans

This section discusses the various attributes of container-managed persistence (CMP) entity beans, their relationships, object relational (OR) mappings, finder queries, and how to use these entity beans with Server.

Container Managed Persistence

CMP provides a standard mechanism for implementing persistent business components in the J2EE component model. It defines the life cycle of entity bean instances and management model for entity bean's persistence state and relationships. It ensures that an entity bean is portable across all J2EE-compliant EJB containers.

Persistence is provided by the container transparently through generated implementation classes for entity beans. Entity beans are logically independent of the underlying datasource where the persistence state is stored.

Entity beans rely on CMP to generate methods that perform persistent data access for entity bean instances. These generated methods transfer data between entity bean instances and the underlying resource manager. Persistence is handled by the container at runtime. The advantage of CMP is that the entity bean can be logically independent of the datasource in which the entity is stored.

Object Relational Mapping

OR mapping for CMP 1.1 beans involves mapping the beans to their database schemas. OR mapping for CMP 2.0 beans involves mapping the beans to the database schemas and container-managed relationships (CMR).

Mapping Container-Managed Persistence Fields

The Deploy Tool enables you to map CMP fields defined in the bean with the corresponding fields in the persistent store. While mapping the Java type of the CMP field to the database type of the mapped column, ensure that the mapped types match.

*Note: CMP 1.1 beans can have a Byte Array as container managed field.*

**Example:** A CMP field with type `java.lang.String` must **not** be mapped with a number column type. It is recommended to map this type to `varchar`. Refer to the Type Conversion tables for more information.
Click a CMP bean node, and click the **CMFields** tab. Use the existing tables so that the Server generates the appropriate queries to create, load, store, and remove CMP beans.

To map a CMP bean to the database schemas:
- Select a datasource from the Data Source drop-down list.
- Select a schema name from the Schema Name drop-down list.
- Select a table name from the Table Name drop-down list.
- Select the appropriate database attributes from the DB Attribute field and map them to the CMP fields in the table.

**Finder Queries**

Applications that use CMP 2.0 beans, specify finder queries in a vendor-neutral language - Enterprise Java Bean Query Language (EJB QL) - during packaging. There are no standard finder query languages for applications that use CMP 1.1 beans. Queries for finder methods for such applications are specific to vendors.

Queries for finder methods in CMP 1.1 beans are resolved during deployment for applications deployed on Server. These queries define the behavior of the finder methods.

*Note: For more information on how to define SQL Queries, read “Writing SQL Queries for EJBs” on page 63.*
Mapping Finder Queries

This panel appears only when a CMP 1.1 bean is selected. Click a bean node and select the Finder Queries tab.

The Finder Queries panel enables you to map the find methods to queries. Select a query to display the query in the Query Text field.

To create a new query, click New. This opens the Query Wizard, which helps generate SQL queries for CMP 1.1 beans. A generated query is stored in the bean’s Bean Properties file. When you save or deploy the .jar containing the bean, a file called queries.props is generated and stored in the .jar.

To import existing queries, click Import. This opens the Query Viewer, which helps to import existing queries. This facility is used when transferring queries prepared during development to a deployment site or when relocating .ears.
Container Managed Relationships

Entity beans can have either uni-directional or bi-directional relationships with other beans. You can define and store these relationships in the `ejb-jar.xml` file.

Server supports the following four types of relationship mappings according to the EJB 2.0 specification:

- One-to-One
- One-to-Many
- Many-to-One
- Many-to-Many

Note: Server does not support relationships between beans that are mapped to different datasources.

Uni-directional Relationships

When two beans have a uni-directional relationship, navigation is determined by the direction specified.

Example: If entity A and entity B are in a one-to-one, uni-directional relationship and if the direction is from entity A to entity B, then entity A is aware of entity B, but entity B is not aware of entity A. This type of relationship is implemented with a container managed relationship (CMR) field on the entity bean from which navigation is possible and where there is no related CMR field on the target entity bean.

Server supports uni-directional self-relationships.

Bi-directional Relationships

When two beans have a bi-directional relationship, navigation is possible from either bean.

Example: If entity A and entity B are in a one-to-one bi-directional relationship, both beans are aware of each other.

Server supports bi-directional self-relationships.

Preloading Container Managed Relationships

When a CMR accessor method is invoked, the container does not prefetch the related entity's state but it only associates the identity. That is, the obtained foreign key. Container loads the bean's state when a business method or getter or setter is invoked. This is synonymous with Lazy Loading.

In the case of Lazy Loading the number of database round trips are more which has an impact on performance. Preloading of CMRs aims at minimizing the database round trips for relationship access hence improving their performance.
Performance Advantage With Container Managed Relationship Pre-loading
In case of Lazy Loading, there would be two database round trips for a one-to-one relationship. One query to associate the target identity and another query to load the target entity's state. While pre-loading a single query associates the identity as well as the state during relationship access.

In the case of one-to-many relationship access, the number of database roundtrips can be reduced to one single query from n-number of queries using pre-loading option. Pre-loading is not done for non-transactional relationship access.

Configuring Relationship Preloading in Deploy Tool
Pre-loading is disabled by default. It can be enabled by checking against the the CMR field that is required to be preloaded.

Click the CMRPreloading tab to edit the pre-loading properties for relationships. The CMRPreloading tab is displayed only when the entity bean has relationships.
Deferred Primary Key Mapping

Server enables you to represent columns that are not mapped to CMFields as deferred Primary Key (PK) fields. During deployment, a deployer can choose one or more columns from the corresponding table to define the PK. The container provides a key generation algorithm for these deferred keys, which is used to insert unique values for PK columns. By default, a String value key generator is provided. It is also possible to plug in an integer value key generator.

The Server does not support deferred PKs that contain Foreign Key (FK) constraints. Using entity beans with deferred PK-type specifications limits the client application programming model because clients that are written before deploying the entity bean may not use methods that are based on the PK-type.

For deferred PKs, the PK-type is specified as `java.lang.Object` in the deployment descriptor.

Removing Beans That Have Relationships

When a bean that has a relationship with another bean is removed, the container automatically removes the relationship.

When you call the `Remove` method, only the entity on which the method is invoked is removed. The deletion is not cascaded to other entity beans.

To cascade the deletion of one entity object automatically to another, use the cascade-delete mechanism.

Cascade Delete

Cascade Delete is used to specify that the lifetime of one or more entity objects in a relationship is dependent upon the lifetime of another entity object.

Cascade Delete can only be specified for an EJB-relationship-role element contained in an EJB-relation element if the other EJB-relationship-role element in the same EJB-relation element specifies a multiplicity of `One`. This option cannot be specified for a many-to-many relationship.

This option deletes only the entity object or objects in the relationship for which it is specified. It does not cascade delete other entity objects, unless they are participants in relationship roles for which cascade-delete has also been specified.

Resolving JOIN Conditions

Server resolves relationships automatically. You can also resolve relationships manually by defining JOIN conditions using the Deploy Tool. To define a many-to-many relationship, define a third table which contains the FK constraints of the two tables at the database level.

Choose the `CMRelations` tab in the bean node to view the CMRs. This panel appears only for CMP 2.0 beans. Server resolves relationships automatically between CMP 2.0 beans.
Use this panel to further edit the JOIN conditions as well as view existing CMRs.

To edit the JOIN conditions using the Deploy Tool:
• Select the relation from the **Relationships** field.
• Click the **JOIN Condition** button. This displays the JOIN Condition window.
• Edit the **JOIN Condition** and click **Add**.
• **Close** the JOIN Condition panel. The JOIN condition is edited in the relation.

**Example:** The first table, `SalesOrder`, contains `SalesOrderId`, which is defined as the PK. The second table, `LineItem`, contains `SalesOrderId`, which is the FK related to the PK of `SalesOrder`.

Specify the join condition using the Deploy Tool as shown above.

**Deferred Primary Key Mapping**
This panel appears only if the selected bean is a EJB 2.0 bean and if its PK class is `java.lang.Object`. 
Select the bean node in the .jar and choose the **PrimaryKey** tab.

The attributes in this panel are:

- **Data Source**: Provides a list of the available data sources.
- **Table Name**: Provides a list of tables in the database.
- **PK**: Has a yellow key as its icon when enabled. Click a key icon for a database attribute to make it a PK for the table. Multiple database attributes can function as the PK class. The CMFields and CMRelations tabs are enabled only after the PK class is selected.
  
  **Note**: When the Primary Key panel opens during deployment, the datasource and Table Name fields are disabled in the CMFields panel.
- **DB Attribute**: Displays all the database attributes in the table. Select those DB attributes or columns that do not correspond to a CMP field.

**Data Conversions on Query Results**

The data conversion table lists all the conversions from SQL types to Java types that are supported through `getXXX` methods of Prepared Statement. For example, you can read an SQL VARCHAR value as a string using `getString`, but you cannot read an SQL FLOAT as `java.sql.Date`. 
If you attempt an illegal conversion, or if a data conversion fails, an `SQLException` is raised. For example, you cannot call `getInt` on an SQL `VARCHAR` value `foo`.

**Using ResultSet.getXXX Methods to Retrieve Common SQL Data Types**

This table is shown at the end of this section (Table 3: Details of conversion from SQL to Java types). Cross in the table indicates the method can be used to retrieve the SQL type. `R` indicates the recommended method for retrieving the SQL type.

**Limitations in Mapping SQL Data Types to Java Types**

You must provide appropriate Java mappings for the common SQL data types. Ensure that you have type information to store and retrieve parameters and recover results from SQL statements.

A Java data type may not be isomorphic to the SQL data type. For example, you can represent both fixed length and variable length SQL arrays as variable length Java arrays as Java has no fixed length arrays. Server uses Java strings though they are not a precise match for any SQL CHAR type.

The following table lists the default Java mappings for various common SQL data types. These types are not entirely supported by all databases.

<table>
<thead>
<tr>
<th>SQL Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>String</td>
</tr>
<tr>
<td>LONGVARCHAR</td>
<td>String</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>BIT</td>
<td>boolean</td>
</tr>
<tr>
<td>TINYINT</td>
<td>byte</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>BINARY</td>
<td>byte[]</td>
</tr>
</tbody>
</table>

The following table lists reverse mapping from Java types to SQL types.

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>VARCHAR or LONGVARCHAR</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>byte</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARBINARY or LONGVARBINARY</td>
<td>boolean</td>
</tr>
<tr>
<td>BIT</td>
<td>SMALLINT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TINYINT</td>
<td>short</td>
</tr>
<tr>
<td>SMALLINT</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>VARCHAR or LONGVARCHAR</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>NUMERIC</td>
</tr>
<tr>
<td>byte</td>
<td>TINYINT</td>
</tr>
<tr>
<td>VARBINARY or LONGVARBINARY</td>
<td>boolean</td>
</tr>
<tr>
<td>BIT</td>
<td>SMALLINT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TINYINT</td>
<td>short</td>
</tr>
<tr>
<td>SMALLINT</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Default Java Mappings for common SQL Data Types

Table 2: Details for Reverse Mapping from Java to SQL Types
The mapping for a String is VARCHAR but, it is a LONGVARCHAR if the given value exceeds the limit of VARCHAR values in the driver. This is the case for byte[], VARBINARY, and LONGVARBINARY.

Table 2: Details for Reverse Mapping from Java to SQL Types

<table>
<thead>
<tr>
<th>Java</th>
<th>Type</th>
<th>Java</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>INTEGER</td>
<td>long</td>
<td>BIGINT</td>
</tr>
<tr>
<td>float</td>
<td>REAL</td>
<td>double</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
<td>java.sql.Time</td>
<td>TIME</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>TIMESTAMP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Details of the conversions from SQL to Java Types

<table>
<thead>
<tr>
<th>Methods\Types</th>
<th>tinyint</th>
<th>smallint</th>
<th>integer</th>
<th>bigint</th>
<th>real</th>
<th>float</th>
<th>double</th>
<th>decimal</th>
<th>numeric</th>
<th>bit</th>
<th>char</th>
<th>varchar</th>
<th>longvarchar</th>
<th>binary</th>
<th>varbinary</th>
<th>longvarbinary</th>
<th>date</th>
<th>time</th>
<th>timestamp</th>
</tr>
</thead>
<tbody>
<tr>
<td>getByte</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getShort</td>
<td>x</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getInt</td>
<td>x</td>
<td>x</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getLong</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getFloat</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getDouble</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getBigDecimal</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getBoolean</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getString</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getBytes</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>getDate</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getTime</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getTimestamp</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getAsciiStream</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getUnicodeStream</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getBinaryStream</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>R</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getobject</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Enterprise applications often try to balance scalability and concurrency issues. Concurrency defines how the EJB container must manage concurrent access to an entity bean.

This section discusses selecting the concurrency type in the Deploy Tool.

**Concurrency Type**

Pramati Server features different concurrency modes enabling pessimistic and optimistic concurrency at the application server level and deferring of locking services to the corresponding database.

When writing business methods for entity beans, the bean provider need not worry about concurrent access from multiple transactions. The bean provider can assume that the container ensures appropriate synchronization for entity objects that are accessed concurrently from multiple transactions.

Concurrent transactions use locks to synchronize access to entity beans. These locks are either maintained by the application server or pushed to the underlying persistent store.

**Repeatable Read**

This is the default configuration for entity beans. A single bean instance is created irrespective of the number of concurrent requests. Server ensures serialization of requests to the bean instance. The bean is either loaded fresh before the start of a transaction or obtained from a cache, depending on the type of access to the database.

**Case1: Exclusive DB, Performance Advantage**

An exclusive lock on the instance state in the database is postponed till an `ejbStore()` callback occurs as part of the transaction commit operation. A delayed lock implies that locks to the database are maintained for a limited time.

Performance is enhanced because of limited database resource usage. The entity bean’s state can be cached achieving load optimization. This mechanism must be used only when updates to the database happen through entity bean.

**Case2: Non-Exclusive DB**

At the start of a transaction, a bean is loaded fresh and an exclusive lock acquired on the particular row in the database. This mechanism must be used when the updates to the database are done externally from non-EJB or non-application clients.
Optimistic Repeatable Read

In this isolation level multiple instances of beans are created for each concurrent transaction. When a transaction is committed and when there are any updates in the transaction, a verified update is done. The container validates the state of the cache against the database while committing each transaction.

• If the bean was modified in the transaction, a verified update is performed. The database state is updated with the new values only after ensuring that the data that was read at the start of the transaction is still unmodified in the database.
• If the bean was not modified in the transaction, the container performs a row check with the read state to ensure that no other transaction has modified the instance since the load of the data by this transaction.
• In both cases, if the verified update or row check fails, the transaction is rolled back.
• When two or more transactions happen at the same time, the first transaction is allowed to change the bean and the other transactions are rolled back. In such cases, the application has to retry the transactions. There is no automatic retry mechanism provided by the container.

Use Case Scenario

This concurrency strategy must be used when there are high concurrent reads and low concurrent updates and the cost of rollbacks are sustainable. This strategy yields higher data integrity than Read Committed.

Performance Advantage

For a standalone deployment with exclusive DB access, a load from DB is avoided and concurrent transactions that require shared lock for get accessor methods can go ahead and access the bean.

Read Committed

In this isolation level, multiple instances of beans are created for each concurrent transaction. At the time of committing the transaction, the bean's container-managed fields are compared and an update is done only if the state is dirty.

Use Case Scenario

This option is used when the concurrent transactions are high in reads and low in updates; it can lead to non-repeatable reads and lost updates.

Performance Advantage

Concurrent transactions that require shared lock for get accessor methods can go ahead and access the bean, an update call is avoided if the bean state was not modified as part of the transaction.
Read Only
Read only isolation level provides optimization for data marked as read only in the system. Since the data is never changed, no subsequent loads/stores are required once the bean is loaded.

For non-exclusive database access, refresh intervals can be specified using the following property in the `pramati-or-map.xml` file as: `<ro-refresh-interval>`

Use Case Scenario
Read only access of CMP beans can be used when your application can afford to have a little stale information, say, if the bean doesn’t get modified within the application.

Performance Advantage
Concurrent accesses are allowed on a bean. This also saves the time spent on database access. Methods for the Read Only strategy can be transactional.

Configuring Concurrency Properties in Deploy Tool
Click the Concurrency tab to edit the concurrency properties for entity beans. The concurrency modes supported in Server are:

- Optimistic Repeatable Read - set this to verify all, modified, or modified and accessed fields
- Repeatable Read
- Read Committed
The optimizations can be performed using either of the two database access methods:

- **Exclusive**: Optimization in this case is performed assuming that the bean exclusively accesses the database table. The data in this case, is not modified by any external source. Since, container has exclusive access to the database, it can cache the entity state and achieve load optimization.

- **Non-Exclusive**: Container cannot cache the entity state. Since the data is modified from external sources, the Container has to use database locking to achieve data integrity.

You can either select to update all the CMP fields, or only the modified CMP fields.

You can also use this screen to specify whether you want to insert the bean after `ejbcreate`, `ejbPostcreate`, or at the end-of-transaction.
A session bean instance represents work being performed for the client that calls it. Its fields maintain conversational state on behalf of the session object’s client represented by a specific client/session object pair. It is a reusable component that contains and implements business logic, rules, algorithms, and workflow. Its lifetime is controlled by the client. This section describes the various session bean attributes that must be specified during deployment using the Deploy Tool.

Session Timeout
In the Deploy Tool, click the Session Timeout tab in the Bean Properties node. This panel is displayed only for stateful session beans. The attributes in this panel are:

- Jar: Is the name of the Java Archive name. This field is not editable.
- Bean: Is the name of the bean name. This field is not editable.
- Session Timeout: Enables the Server to remove the session object after the specified time limit if there is no session activity for that time.

Method Information
Method information is relevant only in a clustered environment. To get information on methods, click the stateful session bean node in the .jar and click the Methods tab. All methods of the EJB home and EJB remote interface are listed here.
Select/deselect the methods that alter the bean state.
When you check a method, it implies that the method affects the state of the bean. After every such method call, the state is persisted so that when the bean fails over the console node, the correct state is used.
This information is used in a clustered environment to support a successful failover of Stateful Session Beans.
Since MDB is a stateless component that is invoked by the container on a Java Message Service (JMS™) message, it has neither a home nor a component interface and is an instance of the MDB class.

An MDB instance is created by the container to process messages where the MDB itself is the consumer. The container controls an MDB’s lifetime.

Pramati Message Server is a pure Java implementation of the JMS version 1.0.2 and is embedded in the Server.

Message Server maximizes the portability of client applications that are deployed on J2EE servers without requiring the application developer to learn about message service.

This section discusses the various service components and how to set MDB destinations during application deployment.

### Components of Message Driven Server

The Message Server contains the following components:

- **JMS interface**: Provides basic vendor-independent administrative control features.
- **JMS Client**: Produces or consumes messages. It is the end user of a JMS.
- **Message**: Is an object created by the client, which carries information for another client.
- **Administered Object**: Is a pre-configured JMS object created by the administrator for clients. There are two types of administered objects: destinations and connection factories.

Message Server functions as a broker between the clients, messages, and administered objects, besides providing administrative control features. It manages interactions between these components in a centralized manner and implements the following messaging attributes:

- **Persistence**
- **Durability**
- **Priority**
MDB Destinations

In the Deploy Tool, click the **MDBProperties** tab for a bean under the jar node. This panel is displayed only if the selected bean is an MDB. You can use this panel to select destinations for the MDBs in the jar.

![Deploy Tool screenshot](image)

The destination specifies the type of the JMS destination such as Queue or Topic.

**Pre-configured Destinations for MDB Sample**

Pramati Message Server has the following pre-configured destinations for the MDB sample provided:

- JMSQueue
- JMSTopic
- StockTopic
- BuyQueue
- SellQueue

Pramati Message Server has four connection factories for the MDB sample:

- JMSQueueConnectionFactory
- JMSTopicConnectionFactory
- XAQueueConnectionFactory
- XATopicConnectionFactory
Drag-and-Drop J2EE Applications

Drag-and-Drop service is used to deploy applications by moving the archives to be deployed to a particular directory. Shell scripts, ANT scripts, and OS shell commands can be used to move or copy archives to the AutoDeploy directory.

The Deploy Service is used to deploy applications. Deployments are filtered using file name filters. Server allows deployment of files with .ear, .jar, .war, and .rar extensions, as also exploded directories.

Drag-and-Drop service provides support to configure multiple directories as AutoDeploy directory. AutoDeployer keeps track of the directory configured to deploy archives. Any archive added to this directory is deployed automatically, or a modified archive redeployed.

AutoDeploy Directory

The AutoDeploy directory is used to deploy applications on the Server.

When an archive is dropped into the directory, it is deployed using the Deploy Service. The archive gets redeployed on modification, automatically.

The following types of archives can be dropped into the AutoDeploy:

- Archives with complete mappings
- New archives with incomplete mappings

Configuring AutoDeploy Directory

AutoDeploy directory can be configured in `deploy-config.xml` located in `<install_dir>/server/nodes/<node_name>/config/deploy-config.xml`.

A sample code is provided below:

```xml
<deploy-config>
  <auto-deploy>
    <dir>${NODE_DIR}/archives/autodeploy</dir>
    <poll-interval-seconds>5</poll-interval-seconds>
  </auto-deploy>
</deploy-config>
```
Dropping Archives Into AutoDeploy Directory

The following archives can be dropped into the AutoDeploy directory:

- Completely mapped archives: Archives containing `pramati-j2ee-server.xml` with complete mappings. In case of CMP beans, `pramati-or-map.xml` with complete mappings and Web components with complete mappings.
- Standard archives: Archives that do not have any vendor specific XML, with standard J2EE XMLs.

*Note: Archives containing MDB archives with incomplete mappings for MDB’s, or archives containing CMP bean archives with partially mapped `pramati-or-map.xml` cannot be deployed in the AutoDeploy directory.*

Deploying Archives

Valid archives moved into the AutoDeploy directory get deployed in the Server. In case of incomplete mappings, the deployment mappings are verified by an auto-mapping resolution framework and the validity of the archive is checked by a validation mechanism. This framework verifies and completes the mappings by assuming default data-source and mail-resource picked from the resource configuration.

If the archive has CMP beans, tables must be created before the deployment process. Server creates the tables automatically at the time of deployment, identifies the container managed relationship (CMR) between the beans, and specifies foreign-key (FK) constraints for the database tables.

Redeployment

An archive is redeployed when the content is modified. The auto deployer identifies the changes automatically.

Auto Deployment Poll Interval

The auto deployer keeps polling for the new archives added to the AutoDeploy directory and verifies the timestamps of the existing archives. If a new archive is added, it gets deployed automatically. An archive is redeployed if the timestamp of the existing archive is modified.

The AutoDeploy directory has a timeout parameter poll interval, which is the time taken by the auto deployer to search for new deployments and modified deployments. It must be specified in seconds and can be configured in the `deploy-config.xml`. 
Limitations of AutoDeploy Directory

- Directories cannot be dropped into the directory.
- Archives containing MDBs with incomplete mappings cannot be dropped into the directory. Destination or queues are not mapped with default mappings automatically.
- Archive containing CMP beans with partially mapped `pramati-or-map.xml` cannot be dropped into the directory.

Pramati Deploy Service

The Deploy Service comprises of auto-resolution, automatic table creation, and validation framework.

Auto-resolution Framework

The auto-resolution framework verifies deployment information during deployment of an archive and resolves incomplete mappings. The framework helps in smooth deployment and finishes the deployment tasks left out by the application deployer. Any archive that has to be deployed, must have deployment information specific to the target server environment.

Deployment Information

The following information is required for an application to be deployed in the Server environment:

- JNDI mapping
- Reference mapping
- Bean pool size
- Destination or queue mapping for MDB
- Table mapping for CMP beans
- CMP field vs DB field mappings
- Join conditions for CMR
- Context root and virtual host for Web components
- Connection factories for connector components

Server Specific Deployment Information

Server specific deployment information is specified in:

- `pramati-j2ee-server.xml` containing the jndi-names, pool-sizes, reference-mappings for beans, context root information in case of Web components
- `pramati-or-map.xml` containing Datasource mappings, CMP field mappings, relationships and table mappings in case of CMP beans
- `Queries.props` in case of CMP1.1 beans for query mappings
**Note:** An incomplete deployment mapping becomes a deployment task.

**Functional Aspects**
The Auto Resolution process performs the following functions:
- Verification of deployment information
- Auto-mapping of incomplete information

**Verification of Deployment Information**
Verification of deployment information is a two-step process:

1. **Verification of deployment information from the standard J2EE XMLs:**
   - The Application provider specifies application specific deployment information in the standard J2EE XML. The standard J2EE XMLs are:
     - `ejb-jar.xml` for bean components
     - `web.xml` for Web components
     - `ra.xml` for connector components
   - These XMLs contain information about resource references, security references and EJB references defined in the archive, which are to be mapped to the JNDI names in Server environment. In case of an invalid information, the validity of the standard J2EE XMLs, with regards to the packaged components, is checked and feedback is given to you.

2. **Validation of deployment information given in the proprietary deployment XMLs:**
   - Server specific deployment information is validated against the environment, and invalid or incomplete information is marked as tasks that are completed during the auto-mapping phase.

**Auto-mapping**
Auto-mapping is the completion of deployment tasks with default mappings. These default mappings are specified in the server specific deployment descriptors at the time of deployment.

- Default datasource and mail resource are obtained from the `resource-config.xml` from the default resources section, while the default pool sizes for `ejb-container` are obtained from the `ejb-config.xml`.

- For CMP 2.0 beans, `pramati-or-map.xml` is generated by the automatic table creator. All the mappings required for the deployment of an archive with CMP beans are completed.

- For .wars, the `ejb-link` element that has `ejb-references` must point to a valid bean in a .jar.

**Feedback Mechanism**
Feedback is sent to you in two scenarios:
- Invalid deployment information provided in the standard J2EE XMLs
- Incomplete mappings for an unsupported type of archive for auto-resolution
Invalid deployment information provided in the standard J2EE XMLs

Verification process identifies an incomplete or invalid application deployment information in the standard J2EE XMLs, and gives the feedback about the information to be specified. For example, if the `ejb-jar.xml` contains an invalid resource type for a resource reference, at the time of deployment, you receive the following error message: *The resource-ref <ref-name> in <bean-name> defined in ejb-jar.xml of <jar-name> contains an invalid resource-type <res-type>*.

Incomplete mappings for an unsupported type of archive for auto-resolution

If the server specific deployment XMLs contain incomplete mappings for an unsupported type of archive like CMP 1.1, you receive feedback about the type of mappings to be specified. For example, for MDBs, destination or queue mappings are mandatory. In the absence of these mappings, deployment process stops with a message prompting you to complete mappings.

Automatic Table Creation

Creation of tables during deployment is essential for deployment. Mapping tables and CMP fields for an archive with lot of CMP beans is a tedious task as each bean field has to be mapped to a DB field. The Server provides automatic table info generation, table creation, and `or-map.xml` generation.

Creating Tables

Table creation involves:

- Table name generation
- DB-field name generation
- Type mapping of Java types to Database types
- SQL query generation

Table name information is generated from the bean name and database fields are generated from the CMP field name. Java types defined for the CMP fields are mapped to the appropriate database types by a type-mapping process. For table creation, SQL queries are generated with all the information obtained for table-name, database field name and database types.

Relationship Mapping

Relationships can be specified within beans that use CMRs. Information on CMR is available in `ejb-jar.xml`, using which foreign-key relationships are obtained and specified in the SQL queries.

Generating `pramati-or-map.xml`

Deployment information for CMP beans is available in `pramati-or-map.xml` that is generated using all the information described above. This XML is generated to complete the table mappings, CMP field to database field mappings, and relationship mappings.
For relationship mapping, join conditions are generated along with foreign-key constraints in the tables.
Enterprise applications that are in development phase often go through a lot of modifications and several rounds of testing. This requires re-deployment of the application archives as and when bugs are fixed or new features are added. It can be a real pain to deploy applications as they have to be archived and deployed every time changes are made.

Pramati Server now allows enterprise applications to be deployed right off the working directories. Here, the application is in an unzipped form (or unarchived form) and the deployment descriptors and application class files reside in the same directory. In short, applications need not be archived to be deployed every time changes are made.

**Pre-requisites for Deploying Exploded EARs**

To deploy an extracted application, the archive must be in a form as described below:

- The root application directory must be named same as the archive name. For example, `BankDesk.ear`.
- All the jar/war/rar modules inside must follow the same naming conventions.
- All required Pramati specific deployment descriptors must be present and all mappings must be complete.
For example, to deploy an application, say BankDesk (shipped with the Server installable and available in the /samples directory) as an extracted ear, the directory structure must be as follows:

![Diagram of directory structure]

**Working of Exploded EAR Deployments**

The deploy service selects an appropriate deploy handler for an archive and creates an application object. Once done, the steps are the same as other deployments as all manipulations are done through the application object.

Deployment features not supported by this implementation are:

- Standalone archive validation (without a server instance running) from the command shell using `com.pramati.j2ee.deploy.validator.earvalidator.EARValidator`.
- Offline client jar generation of exploded ear archives.

**Using Command Line**

The relevant commands in the J2EE Shell are:

- deploy
- prepare

All current flags that are available with these commands are fully valid for exploded .ear deployments.

```
j2eadmin> deploy BankDesk.ear -validateappstart true
```

Where `BankDesk.ear` is the name of the exploded ear directory. Two new flags are available for this feature (valid for the above commands `deploy` and `prepare`). They are:
• **-classes**: Specifies that the application classes are to be found in a directory called \texttt{classes} under the module directory. This option does not take an argument for the \texttt{classes} directory. It just says that the application classes is under a \texttt{classes} directory under the directories module.

  For example, for the structure described above for \texttt{BankDesk.ear}, the class files (with full package structure) can be under a directory called \texttt{classes} under the directory \texttt{BankEJB.jar}. Once this flag is set, all application classes must be under the \texttt{classes} directory in all the jar/war/rar modules throughout the application. If this flag is not set, it is assumed that the application classes are present in the modules itself.

  \texttt{j2eeadmin} > deploy BankDesk.ear -classes

• **-utiljars**: \texttt{-utiljars <utilJarDir>} specifies the Util jar directory. This must be a directory name relative to the application root.

  \texttt{j2eeadmin} > deploy BankDesk.ear -utiljars myUtilJars

  Where \texttt{myUtilJars} is a directory under the \texttt{BankDesk.ear} directory. The \texttt{utilJar} directory is a directory containing the utility classes used throughout the application.
The following types of applications can be deployed using the Console:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ear</td>
<td>EJB applications</td>
</tr>
<tr>
<td>.war</td>
<td>Web applications</td>
</tr>
<tr>
<td>.jar</td>
<td>Java applications</td>
</tr>
<tr>
<td>.rar</td>
<td>Resource archive applications. (rars) can be deployed on the Server in two ways - independently as a .rar file or after being packaged within an ear file. If the rar is deployed independently, the resource adapter exists at the server level and any application can use its configured connection factories. If the rar is deployed as an ear, the resource adapter is specific to that archive.</td>
</tr>
</tbody>
</table>

**Deploying Applications Using Console**

To deploy an application on the Pramati Server Management Console:

1. Select the **Configure** tab on the Console.
2. Select the type of application to be deployed from the options provided under the heading **Applications**, say **Web Applications**.
3. This opens a screen using which you can:
   - Browse for, and select an archive you wish to deploy. Click **Deploy**.
   - Or,
   - Deploy an exploded Web archive. Provide mandatory details like the Application Name, the Source Directory of the Application, and the Context Root for the application. You can also select a virtual host optionally. Click **Deploy**.

In case someone else had deployed an application earlier, you can view details related to it.

This completes the deployment process and the related information is displayed in respective screens.
Deploying Applications Using Server Shell

Pramati Server provides a command line interface, the Pramati Server Shell, using which you can perform various Server-related tasks, such as deploying and checking status, from local and remote machines.

Types of Shells
The Server provides the following shells:
- J2EE Shell
- Resource Shell
- JMS Shell
- Kernel shell
- SQL shell
- OS shell
- Security shell

These shells are not dependent on the location of the Server. The local and remote shells provide input and output facility. Further details on each of these shells can be read in the Pramati Server Administration Guide.

This chapter discusses the various J2EE Shell commands that can be used for managing applications.
**J2EE Shell Commands for Managing Applications**

The Server command line interface displays a prompt on which the following commands may be executed within the Server VM for managing applications:

*Table 1: Description of J2EE Shell commands*

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help [command]</td>
<td>Displays help messages. If command name is provided, it provides a more detailed usage help for that command.</td>
</tr>
<tr>
<td>classloader_debug</td>
<td>Sets classloader debugging on/off.</td>
</tr>
<tr>
<td>deploy</td>
<td>Deploys the given ear/jar/rar/war on the server.</td>
</tr>
<tr>
<td>deploygroup</td>
<td>Deploys multiple applications as part of a group.</td>
</tr>
<tr>
<td>deploytool</td>
<td>Brings up the deploy tool connected to the Server for specifying bean references and deploying applications.</td>
</tr>
<tr>
<td>distribute_app</td>
<td>Distributes a deployed application or par file to a Server.</td>
</tr>
<tr>
<td>list</td>
<td>Displays the list of all deployed applications on the Server.</td>
</tr>
<tr>
<td>make_par</td>
<td>Makes a prepared archive (*.par) file which is distributable, and saves it in the parchives directory, under the archives directory.</td>
</tr>
<tr>
<td>prepare</td>
<td>Prepares the ear/jar/war/rar at the given path.</td>
</tr>
<tr>
<td>remove</td>
<td>Removes the application from the Server if it is not running.</td>
</tr>
<tr>
<td>start</td>
<td>Starts the application that is already deployed but not active.</td>
</tr>
<tr>
<td>status</td>
<td>Shows details of J2EE services - naming, resources, transactions, EJB and Web containers, load balancer (for the Web request dispatcher, if enabled), and cluster (if the service is enabled), socket, and session. Use the resource-name for resource service.</td>
</tr>
<tr>
<td>stop</td>
<td>Similar to the undeploy function, but only stops the currently running application and does not remove the ear from the file system. The application may be restarted without redeploying it.</td>
</tr>
<tr>
<td>undeploy</td>
<td>Undeploys the application(s) from the Server. Stops the application, if necessary, and removes the application from the archives directory in the server installation directory.</td>
</tr>
<tr>
<td>validateappstart</td>
<td>Validates an archive that has been deployed but not started.</td>
</tr>
<tr>
<td>validatearchive</td>
<td>Validates the ear/jar/war at the given path.</td>
</tr>
<tr>
<td>who_load_me</td>
<td>Determines the loading classloader for a class.</td>
</tr>
</tbody>
</table>

**Validating Archives From Server Shell**

To validate an archive from the Server Shell, type:

```
java com.pramati.j2ee.deploy.validator.earvalidator.EARValidator <name and location of the archive>
```
**Note:** Before validating an archive from the command prompt, ensure that the classpath has been set. To set the classpath, run `setup.sh` on Unix or `setup.bat` on Windows located at `<install_dir>/server/bin`.

The following options can be specified with this command:

**Table 2: Commands for Validating an Archive**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>Generates and displays a report of all the severity levels after validation is complete.</td>
</tr>
<tr>
<td>-w</td>
<td>Generates and displays a report of messages with Warning and Failed severity levels.</td>
</tr>
<tr>
<td>-f</td>
<td>Generates and displays a report of messages with Failed severity level. This is the default option.</td>
</tr>
<tr>
<td>-v</td>
<td>Generates the results of the validation in the verbose mode.</td>
</tr>
<tr>
<td>-o</td>
<td>Saves the results of the validation to a text file.</td>
</tr>
</tbody>
</table>

For example, at the command prompt, type `java com.pramati.tools.validator.earvalidator.EARValidator -a -oreport.txt orderApp.ear`. 
Deploying Applications Using ANT Handler

Developers who regularly use ANT to manage application development and deployment can use the Pramati ANT Handler to prepare and distribute applications on local and remote Pramati Server instances. The following sections describe how to install ANT handler, an example that uses it to deploy an application, and the Server commands.

Note: For more information on ANT, visit http://jakarta.apache.org/ant/index.html.

ANT is typically used to build application classes. Web components get packaged in a .war, enterprise beans are packaged in a .jar, and .wars and .jars are packaged in a .ear. ANT enables this process to be automated, without any intervention.

The XML file, or the build.xml that contains the instructions, is made up of targets. Each target has a task or set of tasks associated with it. A custom target that handles remote server tasks is provided as part of the ANT integration in Pramati Server. These remote tasks include commands Deploy, Prepare and Start, and Stop and Remove.

Getting Started With ANT Handler

To use the ANT handler, the following jars are required:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pramati_ant.jar</td>
<td>Is the Pramati ANT Handler that enables remote shell commands from a customized ANT task.</td>
</tr>
<tr>
<td>pramati_client_all.jar</td>
<td>Includes all the classes that are required for Java clients to access the Server.</td>
</tr>
</tbody>
</table>

1 These jar files are located at <install_dir>/server/lib/pramati_client/.

2 Set <ANT_Home> environment variable to the directory containing the bin and lib directories of ANT.

3 Copy pramati_ant.jar under <ANT_Home>/lib.
   
   Note: Essentially, pramati_ant.jar files must be specified in the ANT classpath. Only the archives that are in a ready-to-deploy state, that is, the archives that have all the required deployment descriptors, can be deployed or started using the Pramati ANT handler.

4 Ensure that the Server is started and run the ANT script using the command:
   
   <path>ant -buildfile <ANT script filename>

Example: Using the sample ant.xml to Perform a Server Task:
Type `D:\ant\ant.xml prepare_app`.

This prepares the application `BankDesk.ear` on the server as specified in the `ant.xml` (see sample below).

**Pramati ANT Handler Commands**

All command operations in the Pramati Server Remote Shell are executable as an ANT task on Pramati Server. Here is the list of commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prepare</td>
<td>Prepares the application mentioned in the commands attribute.</td>
</tr>
<tr>
<td>start</td>
<td>Starts the application mentioned in the commands attribute if it is already deployed.</td>
</tr>
<tr>
<td>deploy</td>
<td>Prepares and starts the application mentioned in the commands attribute.</td>
</tr>
<tr>
<td>stop</td>
<td>Stops the running application mentioned in the commands attribute.</td>
</tr>
<tr>
<td>remove</td>
<td>Removes application archive from Server installation directory. The application has to be re-prepared.</td>
</tr>
</tbody>
</table>

**Connecting to Server**

Defining a task requires connection to the Server. Server can be contacted by giving its IP, port, realm, user, and password. These are specified as attributes of the task as shown here:

```xml
<property name="ip" value="127.0.0.1"/>
<property name="port" value="9191"/>
<property name="realm" value="system"/>
<property name="user" value="root"/>
<property name="password" value="pramati"/>
```

Specify the properties as listed in the table below when you need to connect to the Server:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>ip</td>
<td>Is the address of the remote server.</td>
<td>Yes</td>
</tr>
<tr>
<td>port</td>
<td>Is the port on which the server is running.</td>
<td>Yes</td>
</tr>
<tr>
<td>realm</td>
<td>Is the security realm of the remote server.</td>
<td>Yes</td>
</tr>
<tr>
<td>user</td>
<td>Is the username to login to the remote server.</td>
<td>Yes</td>
</tr>
<tr>
<td>password</td>
<td>Is the password to login to the remote server.</td>
<td>Yes</td>
</tr>
<tr>
<td>wait</td>
<td>Waits for commands to be executed. Default is True.</td>
<td>No</td>
</tr>
<tr>
<td>classpath</td>
<td>Is the classpath required to execute the task.</td>
<td>No</td>
</tr>
<tr>
<td>commands</td>
<td>Are the commands to be executed.</td>
<td>Yes</td>
</tr>
<tr>
<td>verbose</td>
<td>Generates messages when the task executes. The default value for this attribute is False.</td>
<td>No</td>
</tr>
</tbody>
</table>
A Sample XML

A sample XML is provided below. With this XML, you can prepare, start, stop, and remove the sample application BankDesk.ear that is shipped with Pramati Server.

```
<project name="AntShell" default="serverTasks">
	<taskdef name="pramati" classname="com.pramati.AntTaskHandler"/>

<!-- invoked from outside, probably after the processing to compile & package -->

<target name="serverTasks" depends="prepare_app,start_app,stop_app,remove_app"/>

<target name="prepare_app">
	<pramati verbose="true"
		ip ="localhost"
		port ="9191"
		realm ="system"
	
type ="root"
		type ="pramati"
		wait ="true"
		classpath ="D:\ant\lib\pramati_client_all.jar"
		commands ="prepare D:\PServer50\server\samples\ejb11\BankDesk\ear\BankDesk.ear"
	/>

</target>

<target name="start_app">
	<pramati verbose="true"
		ip ="localhost"
		port ="9191"
		realm ="system"
	
type ="root"
		type ="pramati"
		wait ="true"
		classpath ="D:\ant\lib\pramati_client_all.jar"
		commands ="start BankDesk.ear"
	/>

</target>

<target name="stop_app">
	<pramati verbose="true"
		ip ="localhost"
		port ="9191"
		realm ="system"
	
type ="root"
		type ="pramati"
		wait ="true"
		classpath ="D:\ant\lib\pramati_client_all.jar"
		commands ="stop BankDesk.ear"
	/>

</target>

<target name="remove_app">
	<pramati verbose="true"
		ip ="localhost"
		port ="9191"
		realm ="system"
	
type ="root"
		type ="pramati"
		wait ="true"
		classpath ="D:\ant\lib\pramati_client_all.jar"
		commands ="remove BankDesk.ear"
	/>

</target>
```


wait = "true"
classpath = "D:\ant\lib\pramati_client_all.jar"
commands = "stop BankDesk.ear"

</target>
</project>

Note: The classpath attribute inside <pramati> element must definitely include pramati_client_all.jar.
Starting and Accessing Applications

This chapter explains how to start and access applications deployed on Pramati Server from the Console, from the Server shell, or automatically at the time of Server startup.

Starting Applications

Starting Applications From Console
To start an application from the Console:

1. Select Configure > (type of) Application from the list of deployed applications in the Explore panel.
2. Click Start against the selected application to start the application.

Starting Applications From Server Shell
To start a deployed application from the Server shell, type `start[ear-name]` on the j2eeadmin:> prompt.

You can also use wildcard characters to specify names of applications such as .ear, order*.ear.

Starting Applications Automatically
You can set the deployed applications to start automatically when Server starts in the following ways:

- Set the `<auto-start>` tag in pramati-j2ee-server.xml to true and save the file, as shown below:
  ```xml
  <pramati-j2ee-server>
  ...
  ...
  ...
  <auto-start>true</auto-start>
  </pramati-j2ee-server>
  ```

- When you deploy an application using the Deploy Tool, select ‘Start the application on Server Start’ option to start the application automatically when Server starts.

- To start all the deployed applications automatically when Server starts, type `runserver -startapps` at the command prompt.
Accessing Deployed Applications

After the applications have been deployed, they can be accessed using any browser that supports HTML 3.2.

To access a deployed application, enter the location of the deployed application in the browser window as: http://<host on which the server is running>:<web port number on which Server is listening>/<context root>/<first page of application>

For example, http://localhost:8181/demo/demoindex.htm displays the first page of the application.

Accessing Applications Across Firewalls

In typical deployment scenarios, Servers are located behind firewalls. Any RMI transport layer that attempts to open direct sockets to hosts on the Internet is not allowed to do so. The default RMI transport provides two alternate HTTP–based mechanisms that enable a client to invoke a method on a remote object residing across the firewall. These are:

- HTTP tunneling when the RMI call is within the HTTP protocol
- Using the default socket factory

This section contains information about:

- HTTP tunneling (Port Filtering firewalls)
- Configuring the client
- Configuring the RMI Server
- Working with NAT firewalls
- Default Socket Factory
- Configuring Server for firewalls
- Hosting Scenarios

HTTP Tunneling, Port Filtering Firewalls

To get across a firewall, the transport layer embeds an RMI call within the firewall–trusted HTTP protocol. The RMI call data is sent outside as the body of an HTTP POST request, and the information returned is sent back in the body of the HTTP response.

The transport layer formulates the POST request in two ways:

- If the firewall proxy forwards an HTTP request to an arbitrary port on the host machine, then it is forwarded directly to the port on which the RMI service is listening. The default RMI transport layer listens on a server socket that understands and decodes RMI calls inside POST requests.
- If the firewall proxy forwards HTTP requests to well-known HTTP ports, then it is forwarded to the HTTP server listening on port 80 of the host machine, and a CGI-script forwards the request to the target RMI server port on the same machine.
Configuring Client
While there is no special configuration necessary to enable the client to send RMI calls through a firewall, the client can disable the packaging of RMI calls as HTTP requests. To do this, set the boolean value of `java.rmi.server.disableHttp` property to `true`.

Configuring RMI Server
A client outside the domain of Server host attempting to invoke methods on remote objects requires to first locate Server. The remote references that Server exports must contain the fully qualified name of Server host. Some firewall proxies do not forward the host name if the host name is specified as the IP address of the host.

On some platforms and network environments, the fully qualified name of the host may or may not be available with Server VM. If so, the fully qualified name of the host must be specified with the property `java.rmi.server.hostname` while starting Server.

For example, to start the RMI Server class `ServerImpl` on the machine `docs.pramati.com`, run `java -Djava.rmi.server.hostname=chatsubo.javasoft.com ServerImpl`

If the Server does not support RMI clients that are behind firewalls preventing them from forwarding to arbitrary ports, use the following configuration:

- HTTP server on port 80
- A CGI-script located at the aliased URL path `/cgi-bin/java-rmi.cgi`

This CGI-script:

- Invokes the local Java interpreter to execute a class internal to the transport layer that forwards the request to the appropriate RMI Server port.
- Defines properties in the VM using the names and values of the CGI 1.0-defined environment variables.

A sample script is supplied in the RMI distribution for Unix, Solaris, and Windows Operating Systems. The script must specify the complete path to the Java interpreter on Server.

Working With NAT Firewalls
Network Address Translation (NAT) or IP masquerading implies that the IP of Server host is mapped to a global IP outside the firewall, so that any request arriving at the global IP is transparently directed to the local IP.

For the Server to receive calls from the global IP across the firewall, point `java.rmi.server.hostname` to the global IP while starting the Server.

For example, if your system’s global IP is 192.12.1.143, start the Server using:

```
java -Djava.rmi.server.hostname=192.12.1.143 com.pramati.Server
```
**Default Socket Factory**

The RMI transport layer extends the `java.rmi.server.RMISocketFactory` class to provide a default implementation of a socket factory. The implementation provides resources to client and server sockets, and creates sockets that transparently provide firewall-tunneling mechanism.

Client sockets automatically attempt HTTP connections to hosts that cannot be contacted through a direct socket. Server sockets automatically detect if a newly accepted connection is an HTTP POST request, and if so, returns a socket that exposes only the body of the request to the transport and format its output as an HTTP response.

Client-side sockets having this default behavior are provided by the `java.rmi.server.RMISocketFactory.createSocket` method. Server-side sockets having this default behavior are provided by the `java.rmi.server`.

**Configuring Server for Firewalls**

Administrators must consider the following points while using Server with firewalls:

- The naming service port must always be open.
- HTTP Tunneling drastically reduces performance of network communication and must be avoided if possible.

To configure Pramati Server or a Cluster node to run behind a firewall:

1. Open the HTTP port. In a typical firewall configuration, this is the default port that is open.
2. Open the Naming service port. This is mandatory if Naming service lookups have to occur from the client-side. If HTTP tunneling is used, no other ports need to be open.
3. Open the port on which remote objects are to be exported if HTTP tunneling is not used. This port is already configured as part of Server. See the tag `<export-port>` under `<server-nodes>` in Server configuration file. The value is by default zero, when it exports remote objects on random ports. Specifying an unused port here exports all remote objects onto this port. Hence, to enable the client to talk to remote objects behind a firewall, this port must be open.
4. To enable dynamic downloading of stubs to clients across firewalls, the class file server port must also be open. This port can be specified in `server-config.xml` under `<class-file-server-port>`. By default it is 5020.
   *Note: If dynamic downloading of the EJB stubs is not required, there is no need to open this port across the firewall.*
5. The configuration file of Server must have the global IP of the node specified in the tag `<host-ip>`. When starting Server, the following command line argument is given to the JVM:

   ```
   java -Djava.rmi.server.hostname = <GLOBALIP> com.pramati.Server
   ```
Using Connectors

A J2EE application may consist of connector components such as resource adapters. This section describes how to deploy a resource adapter, define a connection factory, and test the resource adapter using an application.

Packaging Resource Adapters
A resource adapter is packaged in a .rar file, which can be deployed on any J2EE server. A .rar file may be contained in a .ear file as a module, or may exist as a standalone file.

Location of Resource Adapter Archive Components
After deployment, the standalone resource adapter files are located at:

<install_dir>/server/nodes/<server_name>/archives/<rar name>/<rar version dir>

If the adapter is part of an application, the files are located at:

<install_dir>/server/nodes/<server_name>/archives/<application name>/<application version dir>/<rar name>/

A rar contains the following files and directories:

<table>
<thead>
<tr>
<th>Directory/Files</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.jar</td>
<td>Contains the implementation classes for connecting to the EIS.</td>
</tr>
<tr>
<td>ra.xml</td>
<td>Contains all the details of the resource adapter. It is located in the Meta-inf folder.</td>
</tr>
<tr>
<td>*.html</td>
<td>Refers to HTML files.</td>
</tr>
<tr>
<td>library files</td>
<td>Any system specific libraries required by the RAR classes.</td>
</tr>
</tbody>
</table>

Managing Standard Resource Adapter Deployment Descriptor
A resource adapter has an associated descriptor file containing adapter-specific parameters. These parameters are defined based on the J2EE Connector Architecture (henceforth referred to as JCA) specification.

The resource adapter descriptor file, ra.xml, is located at:

<install_dir>/server/nodes/<server_name>/archives/<rar name>/<rar version dir>/META-INF
If the resource adapter is part of an application, the file is located at:

<install_dir>/server/nodes/<server_name>/archives/<application name>/<application version dir>/RAR_name/META-INF

A resource adapter descriptor file contains:

- Generic information on the resource adapter such as name, type of EIS, and vendor name
- Connection factory details such as, connection factory interface, connection factory implementation class, connection interface, and connection implementation class
- Configuration properties such as name, type, and value
- Authentication information
- Inbound communication information for message listener endpoints

**Managing Pramati Specific Resource Adapter Deployment Descriptor**

In addition to the descriptor file, there is a Pramati specific descriptor file, pramati-j2ee-server.xml, with a tag called `connector` for the resource adapter. It contains the name of the resource adapter and the configuration details of all the connection factories that are bound to it. The configuration details are:

- JNDI name.
- User name/password, required if the adapter is container managed.
- The pool properties for the connection factory, like max pool size and the connection request timeout in seconds.
- Configuration properties as name-value pairs containing resource adapter and EIS-specific information. These properties are defined in the deployment descriptor. You can change the values to connect to different EISs.
- If the connector is a JCA 1.5 connector, additional properties are required to define the connector resource. These include the connection definition data - the Managed Connection Factory implementation class, and the Connection and Connection Factory classes and their implementations.
  
  This information is not required for a 1.0 connector because there is only one Managed Connection Factory in a 1.0 connector, while there can be multiple in a 1.5 connector.

This file is located at:

<install_dir>/server/nodes/<server_name>/archives/<rar name>/<rar version dir>

If the resource adapter is part of an application, the file is located at:

<install_dir>/server/nodes/<server_name>/archives/<application name>/<application version dir>.
**Usage Scenario**

A connector (or a resource adapter) can act as a communication bridge between an EIS and J2EE applications running on an application server. This communication can be in two modes if the connector is a JCA 1.5 connector.

If it's a J2EE 1.0 connector, the communication is initiated only by the application, while in case of a 1.5 connector the application can initiate communication and the EIS can also initiate contact with the application using a message delivery model.

To integrate a client with an EIS in a typical enterprise scenario:

1. Obtain a resource adapter specific to the EIS and deploy it on the application server.
2. Define a connection factory for the resource adapter in the server namespace. For a standalone adapter, you can define a connection factory at any time. For a resource adapter that is part of an application, you can define a connection factory only when you deploy the application. To deploy the application, resolve the application-specific tasks and the resource adapter specific task of defining the connection factory.
3. Use the client API provided by the resource adapter, to access data from the information system.

The resource adapter functions as a factory for connections. An application client uses a connection factory to access a connection instance, which it then uses to connect to the EIS.

**Client Interfaces**

The client interfaces used by enterprise applications and resource adapters to connect to heterogeneous EIS are of two types:

- **CCI**: Defined according to the JCA specification to support a common client API across heterogeneous EIS
- **Non-CCI**: Specific to the resource adapter and the associated EIS

The CCI has two main interfaces:

- `javax.resource.cci.ConnectionFactory` to implement connection factory
- `javax.resource.cci.Connection` to implement connection

`ConnectionFactory` is an interface that allows an application client to obtain a connection to an EIS instance.

**Sample Code**

```java
// obtain application server specific naming context
Context ic = new InitialContext();
javax.resource.cci.ConnectionFactory cf = javax.resource.cci.ConnectionFac-
tory)ic.lookup("myconnfact");
javax.resource.cci.Connection con = cf.getConnection();
//use the connection object to access data using CCI
```
Example of non-CCI interfaces are:
for ConnectionFactory interface -
javax.sql.DataSource;
for Connection Interface -
java.sql.Connection
// obtain application server specific naming context
Context ic = new InitialContext();
javax.sql.DataSource ds =
(javax.resource.DataSource)ic.lookup("myconnfact");
java.sql.Connection con = cf.getConnection();
con.doSomething();........
//finally close the connection
con.close();
Also,
for ConnectionFactory interface -
com.myeis.SomeConnectionFactory;
for Connection Interface -
com.myeis.MyConnection
// obtain application server specific naming context
com.myeis.SomeConnectionFactory cf = (com.myeis.SomeConnectionFactory)ic.lookup("myconnfact");
com.myeis.MyConnection myCon = cf.getConnection();
//use the connection object to access data using the API provided by the connection interface
myCon.doSomething();........
//close the connection
myCon.close();

**Pramati JCA Implementation**

Pramati Server implements the system contracts for an application server according to the JCA 1.5 specification. The Pramati JCA implementation supports connection, transaction, and security contracts. It has the following features:

- Supports deployment of both standalone resource adapters and resource adapters that are part of an application.
- Local transactions are optimized according to the JCA specification. For one resource adapter, a local transaction is started instead of a global Java Transaction API to avoid global transaction overheads.
• Security contract supports only password credential interface and basic password authentication mechanism that is specific to an EIS.

• Message driven beans support for other messaging types - Message driven beans are not restricted any more to JMS - they can implement any custom message listener interface and receive messages from any EIS. For example, an EJB 2.1 MDB is capable if receiving messages from a SOAP engine, or from an instant messaging system.

• JCA 1.5 compliant JMS adapter - A JCA 1.5 JMS adapter enables standards based connectivity between a JMS server and a J2EE server. This does away with the need for proprietary interfaces, which used to be the case until now.

• Inbound communication - Enables EISs (Enterprise Information Systems) to initiate communication with J2EE application components like MDBs and perform work instances inside the application server system.

• Transaction inflow - Enables distributed transactions started inside the EIS to be propagated to the application server. Enables EISs to control the lifecycle of such transactions.

• Message inflow - Message driven beans are not restricted any more to JMS - they can implement any custom message listener interface and receive messages from any EIS. For example, an EJB 2.1 MDB is capable if receiving messages from a SOAP engine, or from an instant messaging system.

You can download the JCA 1.5 specification from http://java.sun.com/j2ee/download.html#connectorspec.

Deploying Resource Adapters

RARs can be deployed on Server in two ways:

• As an independent .rar file: The resource adapter exists at the server level and any application can use its configured connection factories and message delivery mechanism.

• Packaged in a .ear file: The resource adapter is specific to that archive. When you undeploy the ear, it undeploys the rar also. Deploying a .rar in an ear requires at least one connection factory configuration. This is displayed as an incomplete task in the Deploy Tool.

The recommended way of deploying a Resource Adapter (standalone, or packaged inside another .ear) is to open it in the Deploy Tool. Once opened, click the rar node in the tree and select Configuration.

Set the following attributes:

• JNDI Name: JNDI name for the ConnectionFactory to be configured

• Pool Size: The minimum and maximum pool size

• User Name and Password: The user name and password required for authentication

• Authorization: Specify authorization type as Container or Component. Selecting Container enables the username and password fields. Selecting Component disables these fields.
If the resource adapter is a JCA 1.5 Resource Adapter, an additional set of properties need to be configured to create a connection factory. The Deploy Tool automatically detects the version of the Resource Adapter and displays the appropriate panels for configuration.

For a 1.5 Resource Adapter, the additional configuration is selecting the Connection Definition, that is, the ManagedConnectionFactory implementation and the Connection and Connection Factory classes and their implementations. Once this is selected, the property section comes up, as described above.

*Note: A Resource Adapter can be deployed just like any other application on the Server.*

The **Property** panel defines one property set for each ManagedConnectionFactory instance for creating connections to various underlying EIS instances.

- **Property**: Name of the property. This field is not editable.
- **Java Type**: Java type of the property set. This field is not editable.
- **Value**: Value of the properties in this field.

**Deploying Applications**

1. At the Server shell, type `deploytool`. The Deploy Tool UI appears.
2. Click **Archive > Open**. Specify the name and location of the enterprise application.
3. Click **Archive > Deploy**. The application is deployed.

**Starting Applications**

1. In the Console, click **Applications**. A list of all the deployed applications appears.
2. Click **Start** to start the enterprise application.

**Undeploying Resource Adapters**

1. At the default prompt, type `deploytool`.
2. Select **Archive > Undeploy > <resource adapter name>**. The resource adapter is undeployed. The connection factories that are bound to the resource adapter are unbound.

*Note: You can type 'list' at the ressh (the resource administration shell) prompt to verify that the connection factories are no longer bound for a particular resource adapter once it is undeployed.*

**Deploying Resource Adapters on Cluster Nodes**

Deploying a resource adapter on a cluster node is similar to deploying it on a standalone server. The additional factors are:

- A resource adapter deployed on one node is available to all the nodes of the cluster
- If a cluster node N2 is shutdown when the resource adapter is deployed on another cluster node N1, the resource adapter is available on N2 after you start it
• Connection factories are available on all the nodes of a cluster

Creating Connector Resources After Deploying Resource Adapters

Connector resources (or connection factories) can be created post-deployment of a resource adapter also. This can be done using either:
• The Server shell
• The Management Console GUI

Using Server Shell to Create Connector Resources (for a 1.0 Connector)

Type `ressh` at the Server shell prompt to go to the resource administration shell. Your prompt must change to `resourceadmin@(servername)>

Even though most of the commands in the `ressh` work in two modes, you can only use the interactive mode for this command.

Type `add_jca_cf` and press Enter. It prompts you for further details as follows:

```
resourceadmin@server>  - rarName * #
```

Enter the resource adapter name for which you want to create a connection factory. The name you enter here must be the name of a resource adapter that you have deployed earlier.

```
resourceadmin@default>  - jndiName * #
```

Enter a unique name for the connection factory. This is the name by which the connection factory is bound in the server's JNDI namespace.

```
resourceadmin@default>  - user #
```

Enter the name of a user if you want the connection factory to be server authored. Press Enter without entering anything if you want the credentials for connecting to be provided by the application code at runtime.

```
resourceadmin@default>  - password * #
```

Enter the password, keeping in mind the things mentioned above.

```
resourceadmin@default>  - maxPoolSize [40] #
```

Enter a valid value for the maximum pool size that your connection factory must have. The default value is 40.

```
resourceadmin@default>  - connectionRequestTimeoutSeconds [10] #
```

Enter a valid value for the maximum time in seconds that your application waits for a connection from the connection factory. The default value is 10 seconds.

Next, you are prompted for entering configuration data for the managed connection factory. Note that this step is optional for some resource adapters. Whether the MCF has any
configurable properties or not depends on the resource adapter provider/EIS. Enter the relevant values.

 Added JCA Connection Factory [jndi name]

This message indicates that your JCA 1.0 connection factory was created successfully. If there were any errors while creating, a stack trace is displayed. You can refer to the stack trace and error messages to detect the problem.

**Using Server Shell to Create Connector Resources (for a 1.5 Connector)**

Type `ressh` at the server shell prompt to go to the resource administration shell. Your prompt must change to `resourceadmin@(servername)>

Even though most of the commands in the ressh work in two modes, you can only use the interactive mode for this command.

Type `add_jca_cf` and press Enter. It prompts you for further details as follows:

```
resourceadmin@server> - rarName * #
```

Enter the resource adapter name for which you want to create a connection factory. The name you enter here must be the name of a resource adapter that you have deployed earlier.

```
resourceadmin@default> - jndiName * #
```

Enter a unique name for the connection factory. This is the name by which the connection factory is bound in the Server's JNDI namespace.

```
resourceadmin@default> - user #
```

Enter the name of a user if you want the connection factory to be server authored. Press Enter without entering anything if you want the credentials for connecting to be provided by the application code at runtime.

```
resourceadmin@default> - password * #
```

Enter the password, keeping in mind the things mentioned above.

```
resourceadmin@default> - maxPoolSize [40] #
```

Enter a valid value for the maximum pool size that your connection factory must have. The default value is 40.

```
resourceadmin@default> - connectionRequestTimeoutSeconds [10] #
```

Enter a valid value for the maximum time in seconds that your application waits for a connection from the connection factory. The default value is 10 seconds.

```
Outbound connection definition * #
```

A list of all available connection definitions are shown before this prompt. This lets you choose the connection implementation that you want for your connection factory. Type the number corresponding to the connection definition data set and press Enter.

Based on your selection, you are prompted for entering configuration data for the managed connection factory. Note that this step is optional for some resource adapters. Whether the MCF
has any configurable properties or not depends on the resource adapter provider/EIS. Enter the relevant values.

Added JCA Connection Factory [jndi name]

This message indicates that your JCA 1.5 connection factory was created successfully. If there were any errors while creating, a stack trace is displayed. You can refer to the stack trace and error messages to detect the problem.

**Using J2EE Connector 1.5 Resource Adapters for Inbound Communication**

The J2EE Connector Architecture 1.5 specification defines additional contracts for an application server as well as a Resource Adapter provider (usually an EIS), which facilitate the EIS to initiate communication to the J2EE application based on a message delivery model. In this model, the resource adapter acts like a bridge between the EIS and the J2EE application. The EIS may be a legacy message oriented middleware (MOM) system, or it may be a non-RDBMS information system, for example.

This new architecture also embraces the J2EE standard message driven beans (MDBs). Prior to J2EE 1.4, MDBs were restricted to being message listeners from JMS-based messaging providers. J2EE 1.4, the EJB 2.1, and JCA 1.5 specification has extended MDBs to implement any Message Listener interface, meaning that MDBs in a J2EE application can now receive messages from any messaging provider with custom APIs.

The requirements for this to happen are:

- The Messaging provider must provide a JCA 1.5 compliant resource adapter capable of inbound communication.
- The application server must support the JCA 1.5 specification and allow such MDBs to be deployed. Pramati Server 5.0 supports the JCA 1.5 and EJB 2.1 specifications completely.

Also, JMS-based providers are also required to provide a resource adapter for communication between applications and the JMS provider. If the JMS provider does not have a 1.5 JCA resource adapter, Pramati Server provides a generic wrapper adapter for such providers.
This chapter describes the functions provided by Pramati Server for Server Application Hooks. These hooks provide a listener mechanism for lifecycle events for specific applications.

An application hook defines a callback interface that can be implemented by application developers to receive callbacks at certain points in their application lifecycle. Such lifecycle events might include the following:

1. Prepare
2. Prestart
3. Start
4. Prestop
5. Stop
6. Remove

The application hook can use application classes in all these methods except the `applicationRemoved()` since the methods are always called in the context of the application classloader.

In case the application hook class uses any application classes, the hook class must be placed in such a location (with the application/outside it) that the application classes are visible to it.

*Note: For further information on the classloading hierarchy and any known caveats, read “Classloader Architecture in Pramati Server” on page 129.*

**Specifying Server Application Hooks**

The application developer might want to perform certain tasks on the different events in an application's lifecycle like start, stop, prepare, and remove. Pramati Server provides a way to configure multiple hooks for each application. The appropriate hook is invoked when the application is deployed, started, prepared, or stopped.
To create and use Server application hooks, the developer must do the following:

1. Provide an implementation for the methods in the `com.pramati.services.j2ee.spi.ApplicationStateListener` interface.
2. Register the implementation class with the Server by specifying it in the `deploy-config.xml`, located in the `<install_dir>/server/nodes/<node_name>/config` directory. For example:
   ```xml
   <application-hooks>
     <application-hook app-name="a.ear" async="true" class-name="com.pramati.test.AppHookTwo"/>
   </application-hooks>
   
   3. Place the implemented class in the application archive.

4. Restart the Server.

**ApplicationStateListener Interface**

The Server provides an application state listener (hook) interface, found under `com.pramati.services.j2ee.spi` package. Methods available are shown in the table below. The implementation class of the `ApplicationStateListener` interface must be listed in the `deploy-config.xml` that is picked up by the Pramati Extension Classloader.

The following methods are available with the StartupHook interface:

```java
/**
 * This method is called just prior to starting the application. This method can
 * use the application classes because the method is called in the context of
 * the application classloader/
 * public void applicationPreStart();

 /** This method is called after the application has started successfully*/
 public void applicationStarted();

 /** This method is called just prior to stopping the application*/
 public void applicationPreStop();

 /** This method is called after the application has stopped.* /
 public void applicationStopped();

 /** This method is called after the application has been prepared successfully.*/
 public void applicationPrepared();

 /** This method is called after the application has been removed, i.e.,
   undeployed.*/
```
public void applicationRemoved();

**Sample Implementation**

```java
public class SampleApplicationHook implements com.pramati.services.j2ee.spi.ApplicationStateListener {

    public void applicationStarted()
    {System.out.println(" Application Started.");}
    public void applicationStopped()
    {System.out.println(" Application Stopped.");}
    public void applicationPrepared()
    {System.out.println(" Application Prepared.");}
    public void applicationRemoved()
    {System.out.println(" Application Removed");}
    public void applicationPreStart()
    {System.out.println("Before application start");}
    public void applicationPreStop()
    {System.out.println("Before application stop");}
}
```

*Note: To compile application hooks, make sure you have pramati_spi.jar in your classpath. This can be found in the `<install_dir>/server/lib/pramati` directory.*

**Configuring Application Hooks**

To configure an application hook, make the following entry in `deploy-config.xml` under the `</deploy-config>` tag:

```
<deploy-config>
    <application-hooks>
        <application-hook app-name="applicationName" async="true" class-name="SampleHook"/>
    </application-hooks>
</deploy-config>
```

Here, `app-name` defines the application for which the hook or listener is to be registered. The `async` attribute specifies whether or not to start the hook method in a separate thread. Setting the value of the `async` attribute to `true` invokes the hook methods in a separate thread.

You can also have multiple hooks for each application. For example:

```
<application-hooks>
    <application-hook app-name="a.ear" async="true" class-name="com.pramati.test.AppHookTwo"/>
    <application-hook app-name="b.ear" async="true" class-name="com.pramati.test.AppHookOne"/>
```
</application-hooks>
Creating Server Startup Hooks

Server Hooks provide a way in which you can get callbacks for specific server lifecycle events. They are used in cases where you want to perform certain tasks during the server’s lifecycle. Startup Hooks are used to synchronize with server lifecycle events. The Deploy Service is used to maintain these hooks.

Server Hooks can be used in the following scenarios:
• Tasks that need to be performed while the server is starting up.
• Tasks that need to synchronize with global server startup events like autostart applications getting started.

To create and use Server Startup Hooks, do the following:

1 Provide an implementation for the `com.pramati.services.j2ee.spi.StartupHook` interface.

2 Register the implementation class with the Server by specifying it in the `deploy-config.xml` located in the `<install_dir>/server/nodes/<node_name>/config` directory. For example:
   ```xml
   <startup-hooks>
     <class-name>com.pramati.test.MyStartupHook</class-name>
   </startup-hooks>
   ```

3 Place the implemented class (`.jar`) in `<install_dir>/server/lib/ext` directory. In case the class is not in a jar, it has to be placed in a `classes` directory under the `<install_dir>/server/lib/ext` directory. The classes directory can be created manually under the `lib/ext` directory, if not present.

4 Restart the Server.

Creating Server Hook

Creating the Server hook requires you to implement the `com.pramati.services.j2ee.spi.StartupHook` interface. This interface must implement the following methods:
• `start()`: This method is executed on successful start of the server. Tasks that need to be performed at the start of the server can be specified here.
• `stop()`: This method is executed on server shutdown. Tasks that need to be performed on server shutdown can be specified here.
• postAutoStartAppsStart(): This method is executed when all the applications deployed on the server and specified as auto-start are started.
• preAppsShutdown(): This method is executed before all applications running on the server are shutdown.

Here is a sample of a class that implements StartupHook interface:

```java
package com.server.hooks.ServerHook;
import com.pramati.services.j2ee.spi.StartupHook;

public class ServerHook implements StartupHook {
    public void start() {
        System.out.println("Server Startup Hook Started");
    }
    public void stop() {
        System.out.println("Server Startup Hook Stopped");
    }
    public void postAutoStartAppsStart() {
        System.out.println("Server Startup Hook Post Startup of Apps");
    }
    public void preAppsShutdown() {
        System.out.println("Server Shutdown...");
    }
}
```

Compiling Server Hook Implemented Class
To compile the Server Hook implementation you need to have pramati_spi.jar in your classpath. This library is available in `<install_dir>/server/lib/pramati`.

Editing deploy-config.xml
The implemented class must be specified in the deploy-config.xml. This class is picked up by the Deploy Service. The tag looks as below:

```xml
<startup-hooks>
    <class-name>com.server.hooks.ServerHook</class-name>
</startup-hooks>
```
Classes are the basic building blocks of Java, outside of which no executable code can be written. A classloader refers to a component that is used to load classes.

Classloaders form a basic component of Java whose function is to load classes at runtime to the Java Virtual Machine (JVM). In other words, a classloader is an object that loads classes into memory, and is responsible for navigating and loading class files at runtime.

Classloaders implement `java.lang.ClassLoader` and allow different portions of the container, and the Web applications running on the container, to have access to different repositories of available classes and resources.

**Classloaders in Pramati Server**

The Server classloader is used for loading classes at runtime for each application that has been deployed on Server. There are various application artefacts that are provided and generated by the Server. A comprehensive list of application artefacts are as follows:

- EJB module
  - Beans
  - Homes
  - Remote interfaces
  - Helper classes packaged within the module
  - Helper classes in ext JARs
- Web module
  - JSPs
  - Servlets
  - Util classes
- Generated during deployment
  - IMPL generated classes – which are executed at the Server side
  - Client.jar – RMI stubs for home and remote interfaces
- Generated at runtime for Java clients
  - Client.jar, which is available only after deployment
Features

The Classloader used by the Deploy Service features:

- **Extension classloader** - Parent of all application classloaders and has the capability to pick up classpath on the fly detecting the addition of jars to the classpath.

- **Application Groups** - Ability to share classes among applications using an application group. A group can be defined in `deploy-config.xml`.

- **Application level classloader** - One single classloader with preloaded classes for all of the applications which would enable even EJB to pick up Web classes.

- **Module level classloader** - Classloader for each Web module, enabling explicit redeployment of a single module without bringing the whole application down.

Application classes precedence can be set by a flag for an application or globally for all applications. You can get the application to load its own classes in the system classpath. This would be available only on demand and not as a default option.

Classloader Hierarchy

In the earlier versions of JDK, there were no relationships between classloaders. Java 1.2 introduced classloaders in a parent-child relationship, where a parent classloader branches out to child classloaders. This relationship between parent and child classloaders is akin to the object relationship of super classes and subclasses. This led to increased security as it was possible to assign modification rights attributes to classloaders.
The CLs can be added at any time during code execution. Every CL, other than the Bootstrap CL, *has a parent.

When a CL is asked to load a particular class or resource, it delegates the request to its parent CL first, which, in turn, sends it to its own parent CL up the hierarchy. If the requested class/resource is not found, the request is sent back to the parent.

**Bootstrap**

The Bootstrap CL is the parent in the Java classloader hierarchy. It has ‘null’ parent. The JVM creates the Bootstrap CL, which loads the Java Development Kit (JDK) internal classes and java.* packages included in the JVM. (For example, the Bootstrap CL loads java.lang.String.) It loads the basic runtime class of JVM – rt.jar, along with any classes from JAR files present in the System Extensions directory ($JAVA_HOME/jre/lib/ext). Bootstrap serves as the default parent for Extension CL and System CL.

*Note: Some JVMs may implement this as more than one classloader, or it may not be visible as a CL.*

**JDK Extension**

The Extension CL is a child of the Bootstrap CL. The Extension CL is the parent of all Application CLs and has the capability to pick up classpath, and detect the addition of jars to the classpath. It loads any JAR files placed in the Extension directory of the JDK. This is a convenient means of extending the JDK without adding entries to the classpath. However, anything in the Extension directory must be self-contained and can only refer to classes in the Extension directory or JDK classes. It loads the classes present in the directory path .jre/lib/ext.

**System**

The System CL is the child of the Extension CL. It loads the classes from the classpath environment variable.

**Pramati Extension CL**

The CL .jre/lib/ext directory can be changed by pointing to a node specific directory. The Extension CL is also used during loading of external classes like DataSources and startup hooks etc. Jar files, classes directories or zip files can be dropped without the server being stopped in the .jre/lib/ext dir.

**Application**

The Application classloaders are created by pramati server to manage application class loading. Different strategies are handled to suite specific requirements.

*Note: The behavior of the above structure is not related to .jre/lib/ext. The jars don't get picked up when you copy them manually. The classpaths have to be set for the jars to be picked.*
Application Classes Precedence
By setting a flag for the whole application or a Web module, you can get the application to load its own classes over the same class in the system class path. This would be available only on demand and not as a default option. This is usually required for applications that need to use their own version of classes rather than the ones which the server uses.

Application Groups
Ability to share classes among applications using an application group. A group can be defined in deploy-config.xml. Grouping enables an application to access the classes packaged in another application in the same group. When an application belonging to a group is restarted, all started applications of that group have to be restarted because the applications use a common classloader.

When the Deploy Service tries to deploy an application, it loads the ejb/rar classloader and Web classloader. After which the application loads the classes through these classloaders. The ejb/rar classloader loads all jars and all jars inside rars. The Web classloader loads all jars required for the Web module. The Bootstrap classloader, Ext classloader and the System classloader are loaded by the Java runtime environment.

The Pramati Ext classloader loads all other external classes from the directory specified in the server-config.xml. The default directory is $install_dir$/server/libext. The rar classloader loads all the standalone rars.

A single classloader can be used to load multiple applications, if the applications are setup in a logical group. Classloader for multiple applications can be set in deploy-config.xml.

```xml
<classloader-for-multiple-apps enabled="false">
  <application-group apps="abcd.ear,xyz.jar"/>
</classloader-for-multiple-apps>
```

In the above case if classloader-for-multiple-apps is false, a single classloader is loader for each application. In the above case, if no group is defined, a universal group for all the applications is assumed.

Any application not mentioned in the applications list above gets its own dedicated classloader. If no group is specified, then a single classloader is used.

Single Classloader for Applications
The Deploy Service can be instructed to use a single classloader at the time of deployment of Web, connector and bean components within an application from the pramati-j2ee-server.xml file.

```xml
<use-single-classloader-for-app>
  true
</use-single-classloader-for-app>
```
CLs have preloaded classes for all applications that enable even an EJB to pick up Web classes, once this feature is enabled. The Web loader would be the same instance and not use the EJB CL as a parent to enable Web classes access by EJB.
Preferring Deployed Classes
This option is used to specify the preference of the classes for the classloader at the time of deployment of the application.

<prefer-deployed-classes for="application"/>

In this case if the application has classes that are also present in the system classpath, the application classes are picked up by the classloader. In this case, the application also includes Web modules.

<prefer-deployed-classes for="web-module"/>

If the Web module classes are to be preferred over the other classes, the above option can be specified.

<prefer-deployed-classes for="none"/>

In this case there is no preference for the classloader and the classes are loaded normally.

Setting Up the Ext Directory
By default the external classes are loaded from <install_dir>/server/libext. But a different ext directory can be specified, from where the classes can be auto loaded. This can be done by using <ext-classloader-dir> from server-config.xml. Since server-config.xml is specific to a server instance, multiple ext directory can be set for multiple server instances.

Classloader Lifecycle
When the Server starts,
1  BootStrap CL loads all the JDK related classes
2  Extension CL loads all the classes present in the JRE/LIB/ext folder
3  System CL loads all the classes set in the Classpath

All classes that form part of Server are placed in System Classpath.

All the EJB modules (jars) within an application have a single CL, whereas each Web module (war) has a separate CL. The structure is such that the System Classpath acts as the parent for the EJB modules, which, in turn, serves as the parent for each of the Web modules. In other words, each instance of the Web module is a direct child of the EJB module.

It is common for Web applications to call EJBs. Server application classloader architecture allows JSPs and servlets to see the EJB interfaces in their parent classloader. This architecture also allows Web applications to be redeployed without redeploying the EJB tier. It is more common to change JSP pages and servlets than to change the EJB tier.
Application Classloading in Server

The System CL is responsible for loading classes from the directories and jars listed on the command-line and/or the `java.class.path` system property when the JVM is invoked. This CL can always be found via the static method `ClassLoader.getSystemClassLoader()`. If not specified, any user-instantiated CL has this loader as its parent.

Every application receives its own classloader hierarchy, and the parent of this hierarchy is System CL. This isolates applications so that one application cannot see the CLs or classes of another. In CLs, no sibling or friend concepts exist. Application CLs can only see their parent classloader - the System CL. This allows Server to host multiple isolated applications within the same JVM.

Server automatically creates a set of CLs when an application is deployed. The base application CL loads any jar files in the application. A child CL is created for each Web application.

Packaging jar, war, and ear

Pramati Server classloading is centered on the concept of an application. An application is normally packaged in an Enterprise Archive (ear) containing application classes. Everything within an ear is considered part of the same application. Other applications include:

- An Enterprise JavaBean (EJB) jar. All the EJB module files such as the enterprise beans, homes, remote interfaces, helper classes packaged within the module, and helper classes in ext jars are packaged as a jar.
- A Web application war. All the Web module files such as .jsp, .html, Util classes, and servlets etc., are packaged as a war.

An ear is a package of all the jars and wars.

If you deploy a jar and a war separately, they are treated as two applications by the Server. If they are deployed together within an ear file, they are treated as one application. This produces a classloader arrangement that allows servlets and JSPs to find the EJB classes.

All jars must be explicitly placed in the system classpath.

Placing Application Files in Other Paths

Application files may also be placed in the:

- System classpath
- Pramati’s Extension directory (`<install_dir>/lib/ext`)

You can place a driver such as the Oracle JDBC Driver `classes12_01.zip` in the system classpath.
Packaging Helper jars

Applications usually have shared utility classes. If you create or acquire utility classes that you plan to use in more than one application, you must package them with each application as separate jar files. The jar files must be self-contained and must not have any references to the classes in the EJB or Web components. Common types of shared utility classes are data access objects or JavaBeans, which are passed between the Web and EJB tier.

You can also add shared utility classes to the Java system classpath. If you modify your utility classes and they are in the Java system classpath, however, you have to restart the Server after you modify the utility classes.

Helper jars may include the utility jars that serve to provide some additional functions, and may be duly referenced by using the path entries in MANIFEST.MF.

Functionality of MANIFEST.MF

Every jar is a zip file that contains the META-INF directory within itself. This META-INF directory is the home of MANIFEST.MF file.

Besides other information, the MANIFEST.MF contains details regarding:

- Main-Class: For example, com.pramati.j2ee server
- Class-Path: For example, a.jar

The J2EE specification provides the manifest classpath entry as a means for a component to specify that it requires an auxiliary jar of classes. You only need to use this manifest classpath entry if you have additional supporting jar files as part of your jar or war file. In such cases, when you create the jar or war file, you must include a manifest file with a classpath element that references the required jar files.

The following is a simple manifest file that references a utility.jar file:

```
Manifest-Version: 1.0 [CRLF]
Classpath: utility.jar [CRLF]
```

In first line of the manifest file, you must always include the Manifest-Version attribute, followed by a new line (CR | LF | CRLF) and then the Classpath attribute. More information about the manifest format can be found at: http://java.sun.com/j2se/1.4/docs/guide/jar/jar.html#JAR.

The manifest classpath entries refer to other archives relative to the current archive in which these entries are defined. This structure allows multiple wars and jars to share a common library jar.

The manifest file itself must be located in the archive at META-INF/MANIFEST.MF.

For more information, see http://java.sun.com/docs/books/tutorial/jar/basics/manifest.html.
Effective Path in Case of Multiple MANIFEST.MF Files

The effective path to be used if there are references to multiple MANIFEST.MF files is according to the order in which they appear.

For example, let's say we have the following jar files: a.jar ; b.jar ; c.jar and d.jar.
The classpath in the MANIFEST.MF of a.jar is: b.jar /..folder/c.jar, and the classpath of b.jar refers to d.jar. The effective class path to be used is: a.jar;b.jar;d.jar;c.jar.

Important
In other words, if e.jar is loaded at run time, and makes a call to f.jar, which needs g.jar to be loaded first, you get an exception if g.jar is placed lower in the CL hierarchy than f.jar.

What Happens When:
e.jar calls f.jar - The search begins only after the request reaches the parent BootStrap CL that looks for f.jar in the Extension CL, and then System CL where it finds it.
f.jar calls for g.jar - Once found, f.jar makes a call to g.jar, which, since it is not in the same CL, the search again takes the 'highest to lowest' approach – in the meanwhile, e.jar, not being able to run f.jar, gives an exception.
Hence, placing the jar files in the proper hierarchy is of utmost importance so as to not provide these exceptions.

Case Studies

Case 1
There are two classes class1 and class2, and there are two jar files EJB_JAR1 and EJB_JAR2.
The two classes are required only by EJB_JAR1, and not by EJB_JAR2.
The contents for EJB_JAR1 would be the beanclasses + class1 + class2 + ejb_jar.xml (present in the META-INF folder of the JAR file.)

Case 2
There are two classes class1 and class2, and there are 'n' number of jar files EJB_JAR1... EJB_JARn in EAR1.
The two classes are required only by all the jar files.
The contents for EAR1 would be EJB_JAR1 + ... + EJB_JARn + common.jar (that has class1 and class2.)

Note: Remember, the MANIFEST.MF must have a reference to the classpath entry of common.jar.
**Deployment Scenarios**

In code generation and compilation stage of deployment process, certain classes and jars are generated.

- **IMPL generated classes**: IMPL generated classes are executed at server side. EJB module jar and all utility jars referred by EJB module jar through manifest entries are set in the classpath.

- **Client.jar**: The Client.jar contains RMI stubs for Home and Remote interfaces for all EJBs in an application. This jar is required to be placed in the client classpath if the client is running on another VM.

  This jar is located at `<install_dir>/server/nodes/default/archives/<application name>/classes`, where 'default' could be a folder created by a user with a different name.

**Undeployment**

Undeployment of an application unbinds those resources currently being used by the running application, from the Naming Service. It deletes generated files, Remote/Home interfaces, and stubs from the file system.

**Redeployment**

Server allows you to deploy newer versions of application components such as EJBs while Server is running. A class must be reloaded when:

**a) Modifications are made in JSPs**

This is the case when after a class has been loaded using a CL, the JSP page is modified.

After making the changes in the JSP and after recompiling the page, if you try to load the class again using the CL, the JSP page does not reflect the changes. The reason is that although recompilation updates the file, the CL does not load the new file because it picks up JSP pages loaded earlier using a map to its existing paths.

To overcome this, the JSP engine uses smart code generation to check the time stamp of the modified file against the existing file. It changes the class name with an incremental value and places it in the same path, so that the CL picks up the updated file.

**b) Application is redeployed**

When you have to deploy a new version of an already deployed application on your system, you need to create a new application CL.

You cannot use the existing CL because classes might have changed in the new version, which you must reload.

Smart code generation skips classes that have not been modified, loads new classes and creates a new CL structure.
Stopping Server
Stopping a Server stops all services currently running on Server and the current CL hierarchy ceases to exist.

Java Clients

Classes/JARs Required for Java Clients to Access EJBs
The classes and jars that are required by the Java clients for accessing EJBs are:
• Application Client.jar
• Application Helper jars

How are Each of These Available?

Application client.jar
During the preparation phase, RMI Stubs of Home and the Remote Interfaces of EJBs are placed in the Client.jar.

Application helper jars
These are to be provided by the application, with a reference in the client Classpath.

Accessing Message Server From Application
If a J2EE application that is being accessed through a Java client throws a runtime exception, the client needs to have that exception in its Classpath. A typical example is the message class that is used in typecasting the message.

Where must Message Class be?
This class is needed by the accessing points – beans, jsps, servlets, etc.
These application exceptions need to be packaged as separate jar files as util.jar within the EJB module, and the same jar file must also be placed on the client VM at runtime.
To avoid a classcast exception while casting a JMS message content to a particular class, the class must be packaged in the EJB module's jar in which the MDB resides. This is the preferred method to avoid any classcast exceptions.
The class can also be packaged in the utility.jar, which is referred by the EJB module through the Classpath entries in MANIFEST.MF.

Specifying JDBC Drivers

Distributing JDBC Drivers With Application
The JDBC drivers must not be packaged in ears. Instead, they must be available to the system classpath.
setup.bat
Packaging the JDBC drivers within the `setup.bat` is the best option as all the entries are directly stored in the system classpath.

JRE lib/ext
The JDBC drivers can be placed in JRE lib/ext.

*Note: The use of pramati’s lib/ext directory is highly recommended.*

**Problems**
Some of the common classloader problems are:

**Excessive Visibility**
This occurs when a class is duplicated and surfaced as:

**ClassCastException**
The cause of this error is simple, say, casting an Integer to String. Often, the situation is one of "same class, different loader" type – where the source and target types have the same class name.

**Class Duplication**
This problem arises when:

- Cross-application dependencies are managed by copying dependent jars into each application.
- An application is reloaded, the container creates a new classloader and uses it to reload the application classes. In this scenario, however, the original classes may still be accessible.
- Some JNDI implementations do not serialize objects that are locally bound and retrieved. Thus, when applications in the same JVM store and retrieve an object, that instance is shared, and the duplicate class problem can occur.

Normally, the existence of duplicate classes is not a problem. In order for an exception to occur, an instance created in one CL must be passed into the CL of the other. This can be achieved through any storage facility visible to both classloaders, including:

- Static fields
- Global collections
- JNDI
Low Visibility
This situation occurs when a required class is not visible from within the current scope. The problem can surface in the following ways.

ClassNotFoundException
This error occurs during dynamic loading using any method that explicitly loads a class (for example, `Class.forName()` or `ClassLoader.loadClass()`).

NoClassDefFoundException
This error occurs:
- When the application code tries to instantiate an object using the new operator.
- If dependencies of a previously loaded class cannot be resolved. This is confusing due to invisible dependencies.

Mutilated Visibility
Although rare, these are serious errors.

IncompatibleClassChangeError
This error indicates that a superclass or interface has changed.

ClassCircularityError
This error indicates that the current class exists as one of its own superclasses or super interfaces.

UnsupportedClassVersionError
This error occurs when loading a class compiled in a more recent JDK version than the one in which it is running.

VerifyError
This error occurs when the code in the class violates one of the constraints imposed by the JVM.

ClassFormatError
This error indicates that the class file format is invalid, often due to corruption.

FAQs
If a message is being cast, where does the class come from?
To avoid a classcast exception while casting a JMS message content to a particular class, the class must be packaged in the EJB module's jar in which the MDB resides. This is the preferred method to avoid any classcast exceptions.
The class can also be packaged in the utility.jar, which is referred by the EJB module through the Classpath entries in MANIFEST.MF.
What goes into the client jar during preparation?

During the preparation phase, RMI Stubs of Home and the Remote Interfaces of EJBs are placed in the Client.jar.

Where to place application exceptions when using Java clients?

If a J2EE application that is being accessed through a Java client throws a runtime exception, the client needs to have that exception in its Classpath. These application exceptions need to be packaged as separate jar files (util.jar) within the EJB module, and the same jar file must also be placed on the VM client so as to be able to catch them at runtime. Refer to the exceptions in the section on Common Classloader Problems.

Where to place application exceptions when Concentrators access EJBs in Pramati Server?

Concentrators are clients as other Web servers, portal applications etc. that run on other servers and try to access EJBs deployed on Pramati Server. To allow concentrators to access EJBs in the Server, do the same as for handling exceptions.

How to specify the Classpath option when starting Server from a remote VM using Java program?

If the Server is started from a remote VM using a Java program, the classpath option must be specified giving the complete path for running the application. A code sample for the same is provided below:

```java
// code snippet for starting J2ee server.
ArrayList commandList=new ArrayList();
if(System.getProperty("os.name").startsWith("Windows")) file://for windows & NT
   commandList.add(javaHome + File.separator + "bin" + File.separator + "javaw");
else file://any other OS
   commandList.add(javaHome + File.separator + "bin" + File.separator + "java");
   commandList.add("-D<a>=<b>"); // add jvm options; for every jvm option add one entry like this.
   commandList.add("-classpath");
   commandList.add(); file://add server classpath here;
   // that is the class path set by the setup.bat file
   commandList.add("-Dinstall.root=/server");
   commandList.add("com.pramati.J2eeServer");
   // following two lines are not required for default server.
   commandList.add("-config");
   commandList.add("/server/nodes/StandAlone//config/server-config.xml"); // for stand alone server.
```
commandList.add("/server/nodes/Cluster//config/cluster-config.xml"); // for cluster node.
commandList.add("-username"); // this is not required if username password is
commandList.add("""); // "root" and "pramati" respectively.
commandList.add("-password"); //
commandList.add("""); //
commandList.add("-redirect");
commandList.add("-noCommandLine");
Runtime.exec(commandList.toArray(new String[0]));
// end of code snippet

Classloading Guidelines
Here are some of the best practices for classloading:

Declare Dependencies
Make dependencies explicit. Hidden or unknown dependencies are left behind when you move
your application to another environment.

Grouping Dependencies
Ensure that all dependencies are visible at the same level or above. If you move a library, make
sure all dependencies are still visible.

Minimize Visibility
Dependency libraries must be placed at the lowest visibility level that satisfies all dependencies.

Share Libraries
Avoid duplicating libraries. Use the parent attribute to share classes across a set of applications.
Use the tag in the global application.xml file to share classes across all applications.

Keep configurations portable. Choose configuration options in the following order:
1  Standard J2EE options
2  Options that can be expressed within your ear file
3  Server level options
4  J2SE extension options

Use Correct CL
If you call Class.forName(), always pass the CL returned by
Thread.currentThread().getContextClassLoader. If you are loading a properties file, use
Thread.currentThread().getContextClassLoader().getResourceAsStream().
Glossary

classloader
An object that performs classloading at runtime.

classloading
The process of installing a class file into the JVM to create a class.

BootStrap CL/Primordial CL
JVMs in-built CL responsible for storing the core system classes.

User-defined CL
An instance of a subclass of CL with user defined behavior (overridden methods) for certain aspects of class loading. It is responsible for loading all classes other than the core system classes.
Pramati Server supports deployment and registration of custom application MBeans through an entry in the `pramati-j2ee-server.xml`. The MBean classes must be accessible from the application classloader; they must be present in the application itself or placed in the extension directory of the Server.

As the JMX specification does not specify standard interaction between application MBeans and a J2EE Server, this feature is implemented in proprietary way in Pramati Server. Each Pramati application has a custom deployment descriptor called `pramati-j2ee-server.xml`. This XML file is present at the top level of each archive.

Using the `<mbean>` tag available, given the MBean implementation class name and object name, the Server instantiates and registers the MBean when the application is deployed. Following is the DTD of the element `mbean`:

```
<!ATTLIST mbean class NMTOKEN #REQUIRED>
<!ATTLIST mbean name CDATA #REQUIRED>
<!ATTLIST mbean web-module-name CDATA #IMPLIED>
```

Using this element, the deployer can specify which MBean implementation needs to be instantiated and registered with the Server's MBean Server when the application gets deployed. The Server unregisters this MBean whenever the application is undeployed.

Note: The Server takes control of the MBean LifeCycle and registers the MBean as a part of deployment.

The application need not, and must not, try to make an explicit `registerMBean()` call.

### Steps to Register Application MBeans at Time of Deployment

Following steps must be performed to register an application MBean at the time of deployment:

1. **In case of standalone war:** Put the MBean classes in WEB-INF/lib or WEB-INF/classes in case of .war. The .ear can contain several .wars and the MBean classes can be in WEB-INF/lib of any of the .wars. No manifest entries for specifying these
classes are required. But the .war that has the MBean classes must be pointed out by specifying its name in the MBean entry of pramati-j2ee-server.xml.

2 In case of .ear: Put the MBean classes in a .jar inside the .ear. The .ear's MANIFEST entry must refer to this .jar.

Samples

Following is a sample pramati-j2ee-server.xml that intends to register an MBean class SimpleStandard with ObjectName PRAMATI:type=SimpleStandard. This MBean class is expected to be in WEB-INF/lib or WEB-INF/classes in case of a .war.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pramati-j2ee-server PUBLIC '-//Pramati Technologies //DTD Pramati J2ee Server 5.0//EN' 'http://www.pramati.com/dtd/pramati-j2ee-server_5_0.dtd'>
<pramati-j2ee-server version="5.0">
  <description>No Description</description>
  <vhost-name>default</vhost-name>
  <auto-start>TRUE</auto-start>
  <web-module>
    <name>mbeanTestWeb</name>
    <module-name>mbeanTestWeb.war</module-name>
  </web-module>
  <mbeans>
    <mbean class="SimpleStandard" name="PRAMATI:type=SimpleStandard"/>
  </mbeans>
</pramati-j2ee-server>
```

Following is the directory structure of the .war file:

```
$ mbeanTestWeb.war
- index.jsp
- WEB-INF
  - lib
    - *SimpleStandard.jar*
- pramati-j2ee-server.xml
- WEB-INF
  - web.xml
```

The SimpleStandard.jar contains the MBean classes.
Overview

J2EE Interoperability is a set of API and specifications in the Java EE platform that lets developers build interoperable components, that work with CORBA and other non J2EE components, access systems written in non Java languages and platforms in a standard way.

RMI-IIOP Interoperability

The Internet Inter ORB Protocol (IIOP) is an open, vendor independent protocol used by CORBA objects communicating over the network. CORBA IIOP allows heterogenous applications to communicate seamlessly with each other and other services. It is an Object Management Group (OMG) standard - http://www.omg.org. J2EE and CORBA can work together to provide CORBA services across the network to J2EE components like EJBs. The Java IDL mapping/RMI IIOP specification defines how normal EJB components written to work with the RMI API can be made to work over CORBA/IIOP. This requires an underlying IIOP runtime which is usually provided by the application server/EJB container. Pramati Server comes with full support for CORBA/IIOP, including EJB access over IIOP from external clients. This document describes how to configure Pramati Server to use the IIOP protocol; specifically, how to make an EJB application use IIOP.

This involves the following steps:

1. Enable the ORB runtime in the server.
2. Configure the EJB application to use IIOP.

Note: In this document the terms IIOP and RMI/IIOP are used interchangeably while referring to IIOP used in the EJB/J2EE context. ‘Pure’ RMI is referred to as RMI.

Enabling ORB Runtime in Server

The ORB runtime is a service which plugs into the extensible Pramati Services Framework. The ORBInitializer service can be used to plug in any third party ORB for use with Pramati Server. If the ORB requires additional configuration, it can also be specified in the configuration file itself. Pramati Server comes bundled with JacORB (http://www.jacorb.org), a free, high performance Java ORB. It is completely pluggable and can be replaced with any other ORB if required. However, it is recommended that JacORB be used since it has been tested well with Pramati.
The `<install_dir>/server/nodes/<node_name>/config/server-config.xml` file has the following entry:

```xml
<service name="ORBInitializer" enabled="true" class-name="com.pramati.iiop.ORBInitializerImpl">
  <requires/>
  <property name="orb-initial-port" value="900"/>
  <property name="com.pramati.iiop.nameservice.wrapper"
    value="com.pramati.iiop.jacorb.JacORBNameServiceWrapper"/>
  <property name="org.omg.CORBA.ORBClass" value="org.jacorb.orb.ORB"/>
  <property name="org.omg.CORBA.ORBSingletonClass"
    value="org.jacorb.orb.ORBSingleton"/>
  <property name="com.pramati.orb.ssl.port" value="5711"/>
</service>
```

The enabled attribute in the tag is set to false by default, indicating that the service is disabled. Set it to true. The other properties shown above are:

- **orb-initial-port**: The port number where the ORB runtime listens for requests.
- **org.omg.CORBA.ORBClass**: The name of the ORB implementation class.
- **org.omg.CORBA.ORBSingletonClass**: The name of the ORB singleton implementation class.
- **com.pramati.iiop.nameservice.wrapper**: If the ORB requires a separate naming service to be started (as a separate process or in the server JVM) Pramati provides a wrapper class which can be plugged in into the ORB service. To do this the `com.pramati.services.orb.spi.NameServiceWrapper` interface has to be implemented and the impl class name specified as the value for the above property. By default Pramati provides a wrapper class for JacORB, the bundled ORB.
- **com.pramati.orb.ssl.port**: If the components deployed under IIOP are required to use transport layer security (SSL/TLS) then this port must be set. See “Using CORBA-Based Security for EJBs on Pramati Server” on page 153 for how to set up an EJB component to use CORBA security.

Starting the server after enabling the ORBInitializer service automatically bootstraps the ORB runtime and makes the server ready to deploy components over IIOP.

**Note:** Enabling this service does not disable or interfere with the normal RMI communication subsystem of the server. EJB components can be still deployed over pure RMI. The service merely configures the server to use the proper ORB runtime.

### Configuring EJB Applications to use IIOP

An EJB application can be deployed on Pramati over only RMI (which is the default), only RMI/IIOP, or both simultaneously. This can be configured using the `pramati-j2ee-server.xml` file through the `<transport-type>type</transport-type>` tag, where type can be either rmi or iiop. If this tag is left out, the application is deployed over RMI.
The tag can occur at three levels in the XML:

- At the individual EJB level
- At the individual EJB JAR level
- At the global application level

This gives the application deployer the flexibility to selectively deploy select EJBs over a particular protocol if required. However, the most common scenarios would be probably where the whole application is deployed over one protocol.

IIOP configuration doesn’t require any additional tweaking. After adding this tag with the value set to `iiop` the application can be deployed. The following are the differences between deployment of EJBs over RMI and RMI/IIOP:

<table>
<thead>
<tr>
<th>RMI/JRMP</th>
<th>RMI/IIOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJBs use RMI JRMP for communication.</td>
<td>EJBs use RMI IIOP for communication.</td>
</tr>
<tr>
<td>EJBs are bound in the server’s JNDI naming service.</td>
<td>EJBs are bound in the server’s CosNaming service.</td>
</tr>
</tbody>
</table>

*Note: Invoking `isIdentical` on EJBs may not work as expected.*

## Connecting to Pramati Server From Native Clients Using IIOP

J2EE components like EJBs can be deployed so as to be accessible over the IIOP protocol on Pramati Server. This means that any CORBA object can access these EJBs, using the Interoperable Naming Service or other means. This walks you through writing, compiling, and running such a client.

### Deploying EJB on Pramati Server Over IIOP

Assume that EJB is a Stateless Session Bean with the following remote interface and a JNDI name "HelloBean".

```java
public interface HelloRemote extends EJBObject {
    public String sayHello(String name) throws RemoteException;
}
```

On deploying the EJB, Pramati Server automatically generates the required IIOP stubs along with the Interface Definition Language (IDL) files for the EJB interfaces. The IDL files are required by clients written in other languages to generate stubs in their respective languages.

The IDL files and IIOP stubs are present in the `<install_dir>/server/nodes/<node_name>/archives/<your EJB application name>/<your EJB application version>/classes` directory.
Writing Clients

We'll develop a C++ client to access the EJB. The IDL files mentioned previously, an IDL to C++ compiler and a C++ ORB are required.

- Generate the C++ stubs from the IDL files. The specifics for this depend on the IDL to C++ compiler you are using.
- Write the client: Some sample code for is shown below - you might need to include appropriate libraries depending on the C++ ORB you are using.

```cpp
#include <CORBA.h>
#include "HelloRemote.h" //Generated by the compiler
#include "HelloHome.h" //Generated by the compiler
using namespace std;

int main (int argc, char *argv[]) {
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var ns = orb->resolve_initial_references("NameService");
    CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow(ns);
    CosNaming::Name name;
    name.length (1);
    name[0].id = CORBA::string_dup ("HelloBean");
    name[0].kind = CORBA::string_dup ("");
    CORBA::Object_var obj;
    cout << "Looking up HelloBean in the CosNaming service" << flush;
    try {
        obj = nc->resolve (name);
    } catch (CosNaming::NamingContext::NotFound &exc) {
        cout << "NotFound" << endl;
        exit (1);
    } catch (CosNaming::NamingContext::CannotProceed &exc) {
        cout << "CannotProceed" << endl;
        exit (1);
    } catch (CosNaming::NamingContext::InvalidName &exc) {
        cout << "InvalidName" << endl;
        exit (1);
    }
    HelloRemoteHome_var home = HelloRemoteHome::_narrow (obj);
    HelloRemote_var remote = home->create();
    CORBA::WString_Value result = remote->sayHello("Dude");
}
cout << "Result of calling HelloBean: " << result << endl;
return 0;
}

**Compiling Clients**

Compile this client using your favorite C++ compiler, including all ORB libraries as required. The header files and classes for the EJB (earlier generated by the IDL to C++ compiler) must also be present in the include path.

**Running Clients**

This client initializes the ORB and resolves the NamingService first. Pramati Server comes with JacORB's inbuilt naming service implementation - the IOR for must be used to initialize the C++ ORB also. On accessing the EJB using the name "HelloBean", the C++ client gets hold of a CORBA object reference which was earlier bound in the NamingService by Pramati's EJB container. Method calls on this object reference are delegated to the internal servants (actual EJBs) inside the EJB container and the result returned.

**Connecting to Pramati Server From Java Clients Using RMI IIOP**

J2EE components like EJBs can be deployed on Pramati Server and made accessible over the IIOP protocol. This enables EJBs to be invoked over the IIOP protocol from Java (and other language) clients. This document walks you through developing such a Java RMI IIOP client.

**Deploying EJB on Pramati Server**

*Note: Refer to relevant documentation on how to deploy an EJB over IIOP.*

The deployment process generates IIOP stubs as well as the required IDL (Interface Definition Language) files. These files are present in `<install_dir/server/nodes/<node>/archives/<your EJB application name>/<your EJB application version>/classes`.

Assume that EJB is a Stateless Session Bean with a JNDI name "HelloBean" and the following remote interface -

```
public interface HelloRemote extends EJBObject {
    public String sayHello(String name) throws RemoteException;
}
```

**Writing Clients**

Pramati Server comes bundled with JacORB, a free, high performance Java ORB. The IIOP backbone inside Pramati Server is provided by this ORB. However, with this client you can use any ORB you want, since the client code uses only the standard CORBA Java APIs and the EJB interfaces.

Sample code for a Java client is shown below:

```
import java.io.*;
```
import java.rmi.RemoteException;
import java.util.*;
import javax.ejb.CreateException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
public class JavaRMICORBAClient {
    public static void main(String[] args) throws NamingException, RemoteException, CreateException {
        Properties props = new Properties();
        props.setProperty(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.cosnaming.CNCtxFactory");
        // replace this with the correct ip:port
        props.setProperty(Context.PROVIDER_URL, "iiop://localhost:900");
        InitialContext ic = new InitialContext(props);
        HelloRemoteHome home = (HelloRemoteHome) PortableRemoteObject.narrow(ic.lookup("HelloBean"), HelloRemoteHome.class);
        HelloRemote remote = home.create();
        String result = remote.sayHello(");
        System.out.println("Result of calling the bean deployed over IIOP: "+ result);
    }
}

Replace the value of the property Context.PROVIDER_URL in the properties object with the correct IP port combination, or the IOR (or corbaloc) of the NamingService of the server. Note that the client is identical to any normal RMI client which is used to invoke EJBs - except for the fact that the PortableRemoteObject.narrow method is mandatory in case of RMI IIOP and not in plain RMI. Thus the RMI IIOP programming model allows existing EJB applications/client to be made easily accessible over IIOP to CORBA objects, without any changes in the code.

**Compiling Clients**

Compile the client after including all required EJB interfaces in the classpath.

**Running Clients**

Run the client using java JavaRMICORBAClient.
Using CORBA-Based Security for EJBs on Pramati Server

EJBs can be configured to use the IIOP protocol on Pramati Server by configuring the pramati-j2ee-server.xml file. This enables the EJBs to be accessible over IIOP to both Java and non Java CORBA clients across the network and serve requests.

Such invocations need to be secured in cases where the underlying network is unsafe, or the EJB’s themselves require certain security protocols to be followed. The CSIv2 specification describes the security requirements of CORBA components. The EJB 2.1 specification describes the same with respect to EJBs and mandates a conformance level of 0 as defined in the CSIv2 specification. In order to interoperate with CORBA clients, EJBs have to be configured to use CORBA security semantics. There are three levels to this security:

- Transport layer security
- Application level authentication
- Application level identity assertion

Transport Layer Security

Transport layer security is provided by the SSL/TLS protocols. To configure the EJB container to use SSL/TLS over IIOP, the following tag has to be provided in the pramati-j2ee-server.xml under the tag for the EJB whose security is being configured.

```
<ior-security-descriptor>
  <transport-config integrity="required" confidentiality="required"
     establish-trust-in-client="supported" establish-trust-in-target="supported"/>
  
  <as-context auth-method="username_password" required="true"/>
</ior-security-descriptor>
```

This specifies which secure transport options are required for that particular EJB.

*Note: To enable SSL over JRMP, set the <is-secure>true</is-secure> tag to true.*

Application Level Authentication

Application level authentication involves client authentication based on tokens. The CSIv2 specification defines a GSS username password (GSSUP) mechanism for transmitting such data.

```
<ior-security-descriptor>
  
  <as-context auth-method="username_password" required="true"/>

</ior-security-descriptor>
```

This tag specifies that the username-password mechanism is to be used, and that is it mandatory.
The standalone application client has to keep login.props file (which has the authentication details) under the client working directory.

A sample of login.props file follows:

```
java.naming.security.principal=<username>
java.naming.security.credentials=<password>
com.pramati.naming.realm=<realm name>
```

If the file does not exist, then the client container creates a new file with the default properties so that you can edit it later.

**Application Level Identity Assertion**

Identity assertion is a mechanism that enables an entity to act on behalf of another. This entity is usually an intermediate between the actual originator of the request and the target whom the request is meant for. The intermediary may or may not independently authenticate itself to the target at the transport or AS layers (according to the security settings between them), but this authentication is independent of the asserted identity.

```
<ior-security-descriptor>
    ......
    <sas-context caller-propagation="none"/>
    ......
<ior-security-descriptor>
```

This tag specifies whether or not caller (or identity) propagation is supported.

Sometimes the server may act as client to another server, when forwarding the request. In that case it does not send the authentication details of the client who made the request. Instead, it sends its own authentication details to the server to which the request is getting forwarded.

In that case the server uses the authentication details provided by the Server Authentication Details provider. The default provider reads the authentication details from the remote-server-login.props file located under the server's working directory. In this release it uses the same properties to make requests to all the other servers.

It contains the same properties that kept in login.props as mentioned above.

**Custom Implementation for Server Authentication Details Provider**

The custom implementation of the Server Authentication Details provider must implement the interface com.pramati.security.interop.AuthenticationDetails. The implementation class name must be specified using the System property com.pramati.security.server.authenticationdetailsimpl. The implementation class must be in the classpath of the server. The client information is propagated to the server with identity assertion token.
The identity assertion uses the following types supported by CSIv2 Specification:

- ITT Principal Name
- ITT X509 Certificate Chain
- ITT X501 Distinguished Name
- ITT Absent
- ITT Anonymous

The server passes the information in one of the above identity tokens, depending on how client got authenticated to it. If for example the client did not authenticated to the server, then it uses the ITT Anonymous identity token while forwarding the request.

**Connecting to CORBA Objects From Pramati Server**

Pramati Server comes with a bundled Java ORB (JacORB) and an Interoperable Naming Service. The ORB runtime in the server is initialized to use JacORB - so that any ORB initialization calls, even from inside applications - default to using this ORB. J2EE components like EJBs and JSPs can access CORBA objects on other processes/systems. These CORBA objects might be written in other languages, or might be J2EE components like EJBs themselves. In case of CORBA objects written in other languages (or even Java), a reference to the object has to be obtained using any of the following mechanisms available through CORBA:

- The Interoperable Naming Service
- A stringified IOR as a Java string
- An IOR in a file, available on a local filesystem or downloadable over a protocol like http or ftp
- A corbaloc address. For example, `corbaloc:iiop:1.2@192.168.1.11:900#MyObject`

In case of EJBs on other systems made accessible over IIOP, the normal RMI IIOP programming model or the Java IDL programming model can be used.

Additionally, EJB references in J2EE applications can be configured at deployment time to point to remote EJBs deployed over IIOP on other servers. Such links usually take the form of a corbaloc address as specified above. This configuration can be used if the remote EJBs are available over IIOP and the EJBs or Web components on the local server have ejb-refs defined which must point to those remote EJBs. These links can be added in the `pramati-j2ee-server.xml` file inside the EJB or Web application on Pramati Server.
Using Jikes Compiler and Compilation Server

The Server can perform out-of-process compilation in two different ways:

- By using an external compiler, which leads to creation of a new process each time a new compilation is to be done. In such a case the path to an external compiler like JIKES can be specified. The JDK compiler is used by default. The value for the use attribute can be javac, jikes as below.
  ```xml
  <external-compiler use="javac" path="C:\jdk1.3.101\bin\javac.exe">
  <external-compiler use="jikes" path="C:\Jikes\bin\jikes.exe">
  ```
- By using a Compilation Server provided by Pramati Server, which creates a persistent VM, to perform compilations. In such a case the com.sun.tools.javac.v8.Main class, present in tools.jar is used to perform compilation. Note that the tools.jar must be present in the Server classpath.

Using Jikes Compiler

The Jikes™ compiler translates Java source files as defined in the Java Language Specification into the bytecode instruction set and binary format defined in the Java VM specification.


To configure Server to use Jikes, change the external-compiler tag in deploy-config.xml present in <install_root>/server/nodes/<node_name>/config/ to specify that you want to use Jikes compiler. This is applicable only for out of process compilation. The use attribute must be set to jikes and the path must point to the correct path of the jikes executable:

```xml
<compilation>
  <out-of-process-compilation enabled="true">
    <external-compiler use="jikes"
      path="C:\jikes-1.18\bin\jikes.exe"/>
    <property name="initial-heap-size-megabytes" value="120"/>
    <property name="maximum-heap-size-megabytes" value="200"/>
  </out-of-process-compilation>
</compilation>
```
Using Compilation Server

When a request is received for a JSP file the jsp file is parsed to create a servlet and servlet is compiled before loading and serving to the client. This compilation can either be in-process or out-process. In-process compilation involves carrying out the compilation in the same VM in which server is running. The in-process compilation implemented by us presently has a few drawbacks.

Since the compilation is done in the same VM in which the server is running the memory requirement of server increases. Out-process compilation involves starting a new separate VM and carrying out the compilation in that VM. The out-process compilation suffers with following drawbacks:

- Since the VM for compilation is started each time a request for compilation comes to the server there is a significant delay and overhead involved in compilation.
- When an application is started for the first time there would be large number of compilation requests. Starting and stopping a new VM for each request leads to high initial loading time.

The server has a feature called Compilation Server that runs in a new VM. Every time a compilation request comes, the request is sent to the compilation server. The Compilation Server does the compilation and writes the response on its own System.out and System.err.

The following tag in deploy-config.xml file shows how to configure a compilation server:

```xml
<compilation-server>
  <options>
    <property name="initial-heap-size-megabytes" value="120"/>
    <property name="maximum-heap-size-megabytes" value="200"/>
    <property name="idle-timeout-minutes" value="5"/>
    <property name="number-of-trials" value="5"/>
  </options>
</compilation-server>
```

Note that:

- If an illegal value is specified for idle-timeout-minutes property, the default value used by the Server is 5 minutes.
- If an illegal value is specified for the number-of-trials property, the default value used by the Server is 2.
- If illegal values have been specified for initial-heap-size-megabytes or maximum-heap-size-megabytes, the Server uses the default value of 120m for initial heap size and 200m for maximum heap size.

If out-of-process compilation is disabled by setting the 'enabled' attribute to "false" of the <out-of-process-compilation> tag and J2SE1.4 is being used, the default behavior of the Server would be to use compilation.
Creating Deployment Descriptors

Deployment descriptors are standard XML files that are packaged with J2EE applications. They define components and operating parameters for applications. This chapter describes how to manually create the various deployment descriptors that are needed to deploy any application. Deployers need to resolve several server-specific dependencies before deploying an application.

Standard XML Descriptors

Several standard deployment descriptors need to be created for a J2EE application. These files can be created and modified using any text editor. The following XML elements are used to define these deployment descriptors:

- `application.xml`: Contains the elements required by ears
- `ejb-jar.xml`: Contains the elements required by jars
- `web.xml`: Contains the elements required by wars
- `ra.xml`: Contains the elements required by connectors

These XML descriptors must be packaged according to the J2EE packaging specification.

Pramati Specific XML Descriptors

All deployment mappings in the Server must be stored in proprietary XML files such as:

- `pramati-j2ee-server.xml`: Contains all elements required for deployment mappings except O-R mapping
- `pramati-or-map.xml`: Contains all elements required for O-R mapping

These XMLs are located at `<install_dir>/server/dtds`. To create proprietary files for an application:

1. Open the two DTD files, `pramati-j2ee-server_5_0.dtd` and `pramati-or-map_3_0.dtd` in a text editor.
2. Refer to these files to create the proprietary files, which must follow the same format and carry the same name. For example, the template for auto-start

```xml
<!ELEMENT auto-start (#PCDATA)>
```
3 Save the newly created files in the EAR in the application directory.

Note: For more information on the DTD files shipped with Server, read ‘DTDs for Server XML’.
Samples for Deployment

Server ships with sample programs located at `<install_dir>/server/samples`. You can use the sample code to build your own applications. The sample applications are:

- iBank
- Numguess
- Game of Eight
- OrderBook
- StockMarket

This section explains how to deploy a sample application using the Deploy Tool.

**Deploying iBank Sample Application**

The iBank sample uses EJBs v1.1 and JSPs to create a simple online bank application. You can create an account, transfer amounts between accounts, view ledger entries, and summary of all accounts.

The application uses three enterprise beans, LedgerBean, TransferBean, and CheckingBean and a thin HTML client.

**Shipped Resource**

The sample ships with a datasource called demo, which has been created using Informix Cloudscape database.

**Configuring Resources Using Other Databases**

You can create resources using databases other than Cloudscape. The iBank sample ships with the database script files, which create the required tables on Oracle, IBM Informix, and MS SQL servers. The script files are located at:

- `<install_dir>/server/samples/oracle_ejb.sql`
- `<install_dir>/server/samples/informix_ejb.sql`
- `<install_dir>/server/samples/mssql_ejb.sql`

Modify these scripts to use other databases. To execute the scripts and prepare the tables for the application, select the database and continue as explained.
Oracle Database
1. Open the SQL*PLUS editor.
2. To access the database, enter the username and password as scott and tiger, respectively.
3. Run the following command from the SQL prompt:
   `<install_dir>/samples/oracle_ejb.sql`.

IBM Informix Database
1. Start the SQL Editor.
2. Connect to the database and run `<install_dir>/samples/informix_ejb.sql`.

MS SQL Server
1. Create the database protondemo using SQL Server Enterprise Manager.
2. Open Query Analyser in SQL Server 7.0 or iSQL in SQL Server 6.5.
3. Select protondemo as the database and run
   `<install_dir>/samples/mssql_ejb.sql`.
4. Create an ODBC32 datasource, protondemo, which connects to the protondemo database.

To add a resource on other databases:
- In the Console, click **Resources**.
- Select **DataSource**. In the new page, click **plus**.
- In the Create Datasource page, enter JNDI name as demo.
- Enter the URL of the database, database driver name, user name, and password.
- Click **Next**. In the new page, click **Next** to continue. This creates the data resource.

Opening Archives
Open the sample archive from `<install_dir>/server/samples/ejb11/BankDesk/ear/BankDesk.ear`. The .ear contains a .jar and a .war. The .jar contains three EJBs and the .war includes some JSPs.
Deploying Application
To deploy the application, choose Archive > Deploy from the Deploy Tool menu bar.

Accessing Application
To start the application, enter the URL as http://127.0.0.1:8181/bankWeb/bank.jsp in the browser, where bankWeb is the context root of the application.

From an HTML client, you can create an account, transfer funds from one account to another, view ledger entries, and view the summary of all the accounts.
The Numguess Sample Application
The Numguess.jsp is a game application. The program generates a random integer between 1 and 100. When a HTML client submits a random guess, the number is compared and a message is returned to the client, indicating if the guess is less than or greater than the actual number. This operation continues till the guess matches the actual number.

Opening Archives
Open the sample archive from `<install_dir>/server/samples/ejb11/jspDesk/war/jspDesk.war`.

The .war contains:
- numguess.jsp: Accepts inputs from the client, refers it to the bean, and returns the results
- NumberGuessBean.java: Is a bean that generates a random number, compares it with the actual number, and returns the result to the calling JSP
- Image files used in the HTML client page
Deploying Application
Choose Archive > Deploy from the main menu of the Deploy Tool to deploy the application on Server.

Accessing Application
To start the application, type the URL as http://localhost:8181/jspDeskWeb/numguess.jsp in the browser and run the .jsp. Here, jspDeskWeb is the context root of the application.
Enter random numbers between 1 and 100 in the page that is displayed until a match occurs.
The Game of Eight Sample Application

In this application, a jumbled magic square of numbers from 1 to 8 is ordered using an HTML client. The program checks for the correct order each time a tile is moved. When all the tiles have been ordered, a winning message and the time taken to solve it is displayed.

Opening Archives

Open the enterprise archive from `<install_dir>/server/samples/ejb11/jspDesk/ear/jspDesk.ear`. The sample application uses the following components:

- `gameofeight.jsp`: This is the client view of the game, which collects information about the sequence of tiles after every move, refers to the bean to check against the required order, and reports the result back to the client. The JSP includes HTML tags that compose the tiles of the magic square.
- `Eight.java`: Embeds routine methods to check the sequence each time a tile is moved and each time a new square is formed and matches that sequence to the correctly ordered square.
- Image files (.gif) used to compose the HTML pages.
Deploying Application
Choose **Archive > Deploy** from the main menu of the Deploy Tool to deploy the application.

Accessing Application
To start the application, enter the URL as `http://localhost:8181/jspDeskWeb/gameofeight.jsp` in the browser. Here `jspDeskWeb` is the context root of the application. In the page, move the tiles to order them.
The OrderBook Sample Application

This sample application demonstrates how to work with version 2.0 EJBs. The sample uses beans, JSPs and HTML pages to create a simple online application that enables you to add customers, view customers, place orders for products, and view orders that have been placed.

Shipped Resource

The sample ships with a datasource called demo, which has been created using Informix Cloudscape database.

Adding Resources Using Other Databases

You can create resources using databases other than Cloudscape. The sample ships with the following database script files, which create the required tables on Oracle, IBM Informix and MS SQL servers:

- `<install_dir>/server/samples/oracle_ejb.sql`
- `<install_dir>/server/samples/informix_ejb.sql`
• `<install_dir>/server/samples/mssql_ejb.sql`
  
  *Note: Modify these scripts to use other databases. Refer to the iBank documentation for more information about executing the scripts and preparing the tables for the application. It is not necessary to execute these scripts again if you have executed them for the iBank sample.*

To add a resource to other databases:

- Click **Resources** in the Console.
- Select **DataSource**. In the new page, click **plus**.
- In the Create Datasource page, enter the JNDI name as `demo`.
- Type the URL for the database is located, the database driver name, the user name, and password.
- Click **Next**. In the new page. Click **Next** to continue. This creates the data resource.

### Opening Archive

Open the `<install_dir>/server/samples.ejb20/OrderBook/ear/OrderBook.ear`. The application uses the following beans:

- **AddressBean.java**: It is a CMP entity bean with local interfaces. The bean has a 1←→N relationship with the CustomerBean.
- **CustomerBean.java**: It is a CMP entity bean with local and remote interfaces. The bean has a 1←→N relationship with the SalesOrderBean.
- **LineItemBean.java**: It is a CMP entity bean with local interfaces. The bean has a N→1 relationship with the ProductBean.
- **OrderSessionBean.java**: It is a session bean with remote interfaces. This is the single point of entry for JSPs.
- **ProductBean.java**: It is a CMP entity bean with local interfaces. The bean has a 1→N relationship with the LineItemBean.
- **SalesOrderBean.java**: It is a CMP entity bean with local and remote interfaces. The bean has a 1←→N relationship with the LineItemBean.
Deploying Application

Choose Archive > Deploy from the Deploy Tool menu bar to deploy the application.

Accessing Application

To start the application, type the URL as http://127.0.0.1:8181/OrderAppWeb/index.htm in the browser, where OrderAppWeb is the context root of the application.
Using the HTML client, try to:

- Add a customer
- View information about a customer
- Place an order against a customer
- View an order and cancel the order

When you place an order for a product, an entry is created against a CustomerID in the SalesOrder table. A LineItem is generated with the OrderID and ProductID. When you cancel an order, the SalesOrder and LineItem are deleted from the database.

**The Stock Market Sample Application**

The sample illustrates the implementation of MDBs on Pramati JMS. The sample uses both Publish-Subscribe and Point-to-Point messaging techniques and requires both Server and Pramati Message Server.

A StockServer receives stock quotes continuously from the topic StockMarket. The Java client, StockServer.java, accesses stock information through the bean SubscriberBean.java. This bean delivers clients’ buy/sell actions to the BuyQueue and SellQueue resources on the Message Server.
After receiving these messages, the beans `BuyAgentMDB.java` and `SellAgentMDB.java` update the resource, StockMarket. The update is reflected on the Java client `StockServer.java`. Clients subscribe to SellQueue or BuyQueue based on their action.

The application demonstrates:

- How to use MDB
- How to send/publish messages to JMS Queues and Topics that the bean subscribes to without invoking an MDB directly.
- How to create a client application

**Shipped Resource**

The sample ships with four JMS sources. The destinations that the application requires are:

- Queues - BuyQueue and SellQueue
- Topics - StockMarket and JMSTopicConnectionFactory

**Opening Archive**

Open the enterprise archive from `<install_dir>/server/samples.ejb20/StockMarket/ear/StockMarket.ear`. The archive contains three MDBs and a Java client:

- `BuyAgentMDB`: Simulates the purchasing of a stock option and displays the message that a particular stock option has been bought. This class implements `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener`.
- `SellAgentMDB`: Simulates the selling of a stock option and displays a message in the server console if a particular stock option has been sold. This class implements `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener`.
- `SubscriberBean`: Simulates a subscription for the stock server, which sends messages containing the company name and the price fluctuation to the subscriber. The subscriber’s decision is based on this information. This class implements `javax.ejb.MessageDrivenBean` and `javax.jms.MessageListener`. This bean has the resource reference as `QueueConnectionFactory` and the resource environment reference as `BuyQueue` and `SellQueue`, which are used by the bean to make buy and sell decisions.
- `StockServer.java`: It is a Java client, which provides information about stock price fluctuations and the company name to subscribers.

**Deploying Application**

- Start the Console.
- Click **Connect**. In the new page, select **Start JMS**.
- Provide all the required information, click **Start**.

This starts Pramati JMS along with the Server. Start Server before starting the Deploy Tool.
Note: Refer to the Pramati Server Administration Guide for more information about starting JMS.

To deploy the application using the Deploy Tool, choose Archive > Deploy from the main menu.

**Accessing Application**

To start the application:

1. At the command prompt, set the server classpath by running either setup.bat for the Windows or setup.sh for the Linux platform.

2. Add `<install_dir>/server/samples/mdb/classes` to the classpath as:
   - On Windows, set the classpath as:
     ```
     classpath=<install_root>/server/samples/mdb/classes;%classpath%
     ```
   - On Linux, set the classpath as:
     ```
     export CLASSPATH=$install_root/server/samples/mdb/classes:%$CLASSPATH
     ```

3. To compile the `StockServer.java` file, type:
   ```
   javac -d <install_dir>/server/samples/ejb20/StockMarket/classes
   and
   <install_dir>/server/samples/ejb20/StockMarket/JavaModule/com/pramati/samples/mdb/StockServer.java
   ```

4. Add `jms.jar`, `jdbc2_0_stdext.jar` and `jndi.jar` from the directory `<install_dir>/server/lib/ext` and `pramati_jms_client.jar`, `pramati_jms_server.jar`, and `pramati_jms_xa.jar` from the directory `<install_dir>/server/lib/ext` to the classpath.

5. Open the `StockServer.java` file and modify the String Topic Name value from `StockMarket` to `StockTopic`.

6. To recompile the `StockServer.java` file, type:
   ```
   javac -d<install_dir>/server/samples/ejb20/StockMarket/classes
   and <install_dir>/server/samples/ejb20/StockMarket/JavaModule/com/pramati/samples/mdb/StockServer.java
   ```

7. To start the stock server, type:
   ```
   java com.pramati.samples.mdb.StockServer
   ```

This brings up the Stock Server dialog box, which displays the stock symbol and price details along with online buying and selling information.
Deployment descriptors are standard text files formatted using XML notation and packaged with J2EE applications. The deployment descriptors define components and operating parameters for applications. The DTDs shipped with Server are:

- EJB 1.1 and EJB 2.0
- Servlets 2.2 and servlets 2.3
- Pramati J2EE Server
- Pramati OR-mapping

These files are packaged in pramati_util.jar located at <install_dir>/server/lib/pramati. The contents of DTD elements are case-sensitive. Comments specify additional requirements for syntax and semantics that cannot be specified with the DTD syntax.

**Deployment Descriptors for EJB**

A deployment descriptor informs the EJB server about the classes that constitute bean implementation, home interface, and remote interface. If there is more than one EJB in the package, the deployment descriptor also indicates how the EJBs interact with one another.

An ejb-jar file created by the bean provider contains one or more EJBs. It does not contain application assembly instructions. An ejb-jar file produced by an application assembler contains one or more EJBs along with application assembly information, which describes how the beans are combined into a single application deployment unit.

*Note: The J2EE specification defines how enterprise beans and other application components contained in multiple ejb-jar files can be assembled into a single application unit. Server supports both EJB 2.0 and 1.1 applications and ships with both DTDs.*

A deployment descriptor captures declarative information, that is not included in the EJB code. The information contained in a typical deployment descriptor are:

- Enterprise beans structural information: Describes the structure of an EJB and declares the external dependencies. It is mandatory for the ejb-jar file producer to provide structural information in the deployment descriptor. This information cannot be modified because doing so could break existing beans.
• Assembly information for an application: Describes how the enterprise bean in the ejb-jar file is composed into a larger application deployment unit. It is not mandatory for the ejb-jar file producer to provide assembly information in the deployment descriptor. Assembly-level information can be changed without breaking the EJB’s function, but it may alter the behavior of an assembled application.

The following configuration and deployment information are contained in the EJB application deployment descriptor:

• EJB name, class, and type
• Remote home and local home interface for EJB 2.0 beans only
• EJB remote and local interface
• Re-entrance indication
• State management type and transaction demarcation type for a session bean
• Bean persistence management, primary key class, primary key-class element, abstract schema name, and CMFs for an entity bean
• CMRs for EJB 2.0 beans only
• Finder query
• Select queries for EJB 2.0 beans only
• Environment entries
• Resource manager connection factory references
• Resource environment references
• EJB references
• Local references for EJB 2.0 beans only
• Security role information
• Destination, message selector, and acknowledgment mode for MDB
• Method permission information
• Transaction attributes

Refer to the ejb-jar_1_1.dtd.xml and ejb-jar_2_0.dtd.xml files to view the XML DTDs that are shipped with Server. These files are located in pramati_util.jar.

**Deployment Descriptor for Web Applications**

Web deployment descriptors contain information about the elements and configuration of Web applications for application developers, application assemblers, and deployers. Web containers are also required to support the 2.2 version of the deployment descriptor for backwards compatibility of applications written to the 2.2 version of the API.

Server ships with DTDs for applications written to both Servlet 2.2 and 2.3 version of the API.
The following configuration and deployment information are contained in the Web application deployment descriptor:

- ServletContext init parameters
- Session configuration
- Servlet declaration
- Servlet mappings
- Application lifecycle listener classes
- Filter definitions and filter mappings
- MIME type mappings
- Welcome file list
- Error pages
- Security information
- Taglib
- Syntax to look up JNDI objects such as env-entry, ejb-ref, ejb-local-ref, resource-ref, and resource-env-ref

Refer to the web-app_2_2.dtd.xml and web-app_2_3.dtd.xml files to view the XML DTD files shipped with Server. These files are located in pramati_util.jar.

**DTD for Pramati Java EE 5.0 Server**

Pramati Server supports Java EE 5.0 version, ships with the DTD, and supports all the DTDs required in the earlier versions. Support for the earlier DTD versions ensures that applications written for previous versions can be deployed on products supporting the current version. It also ensures that there are no restrictions on combining different versions of deployment descriptors in a single application.

```xml
<!ELEMENT auto-start (#PCDATA)>

<!ELEMENT client-jar-path (#PCDATA)>
```
<!--
This element represents the external connection factory that is used in mapping Message Driven bean destinations. Example: <conn-factory>myConnFactory</conn-factory>
myConnFactory is the name of the connection factory on the external JMS Server.
Used in: destination-mapping
-->  
<!ELEMENT conn-factory (#PCDATA)>

<!--
The description element is used to provide text describing the parent element.
-->  
<!ELEMENT description (#PCDATA)>

<!--
This element represents the external destination(queue/topic) that is used in mapping Message Driven bean destinations. Example: <destination-link>myQueue</destination-link>
myQueue is the name of the destination on the external JMS Server.
Used in: destination-mapping
-->  
<!ELEMENT destination-link (#PCDATA)>

<!--
This element provides the external mapping information related to the external JMS server for Message Driven Bean destinations
Used in: ejb
-->  
<!ELEMENT destination-mapping (destination-link, conn-factory, destination-user-name?, destination-password?)>

<!--
This tag represents the password used to create the JMS connection for this MDB. If tag is absent, null password is used.
Used in: destination-mapping
-->  
<!ELEMENT destination-password (#PCDATA)>

<!--
This tag represents the user name used to create the JMS connection for this MDB. If tag is absent, null user name is used.
Used in: destination-mapping
-->  
<!ELEMENT destination-user-name (#PCDATA)>

<!--
This element represents an EJB inside a deployable ejb module for an application. Contains information for its creation and maintenance like pool sizes, jndi-name - by which the server identifies the bean and so on.
Used in: ejb-module
-->  

<!--
ejb-link element is ejb jndi name or ejb local name based on whether it is ejb-ref or ejb-local-ref.
Used in: ejb-local-ref, ejb-ref
-->  
<!ELEMENT ejb-link (#PCDATA)>

<!--
This element represents a ejb-local-ref for the EJB.
Used in: ejb, web-module
-->  
<!ELEMENT ejb-local-ref (ejb-ref-name, ejb-link)>

<!--
This element contains the information for an ejb-module for the application. This module contains related information for the ejb components mostly related to the mapping information for them.
Used in: pramati-j2ee-server
-->  
<!ELEMENT ejb-module (name, ejb+)>  

<!--
This element represents a ejb-local-ref for an EJB/web module.
Used in: ejb, web-module
-->
<!ELEMENT ejb-ref (ejb-ref-name, ejb-link)>

<!--
This element represents the name of the ejb reference.
Used in: ejb-ref, ejb-local-ref
-->  
<!ELEMENT ejb-ref-name (#PCDATA)>

<!--
This element specifies whether access to the bean methods is secure or not.
Used in: ejb
Valid Values: true, false
-->  
<!ELEMENT is-secure (#PCDATA)>

<!--
This element specifies whether bean is clustered or not
Used in: ejb
Valid Values: true, false
-->  
<!ELEMENT is-clustered (#PCDATA)>

<!--
This element represents the name given to the EJB that is unique in the
server name space.
Used in: ejb
-->  
<!ELEMENT jndi-name (#PCDATA)>

<!--
This element represents the low activity interval for the pools used by the
server
Used in: ejb
-->  
<!ELEMENT low-activity-interval (#PCDATA)>

<!--
This element represents the maximum messages for a server session pool for a
MessageDrivenBean.
Used in: server-session
<!ELEMENT max-messages (#PCDATA)>

<!--
This element represents the max pool size for the pools used by the server.
Used in: ejb, server-session, thread-pool
-->  
<!ELEMENT max-pool-size (#PCDATA)>

<!-- represents the name of the finder method as read by loading the bean home
class.
Used in: query-mapping
-->  
<!ELEMENT method-name (#PCDATA)>

<!-- represents the name of the parameter of the finder method when used in
query-mapping. It represents the method signature in pramati specific format
when used in persistant-methods. The format is:
methodName:param1:param2;...
For parameters of type array, [] is used at the end of base class of the
array. For example, java.lang.String[] for string array.
An example of method name in persistant-methods is
addItem:java.lang.String[];int[];String;
while in query-mapping it is addItem.
Used in: query-mapping, persistant-methods
-->  
<!ELEMENT method-param (#PCDATA)>

<!--
This element represents the min pool size for the pools used for the EJB.
Used in: ejb, server-session, thread-pool
-->  
<!ELEMENT min-pool-size (#PCDATA)>

<!--
This element represents the name of the deployable module
Used in: ejb-module, web-module, role-mapping
-->  
<!ELEMENT module-name (#PCDATA)>

<!--
The element name specifies the name of the concerned entity.
Used in: ejb-module, web-module, ejb
--> 
<!ELEMENT name (#PCDATA)>

<!-- persistant-methods element is used to store the information of the methods which affect the state of a stateful session bean.
--> 
<!ELEMENT persistant-methods (method-name*)>

<!-- The pramati-j2ee-server element is the root element of the pramati-j2ee-server xml document. This encapsulates all the information required for an application to be deployed on to Pramati Server.
As an example resource mapping information like mapping a resource onto an actual resource on the server or mapping a security role onto an actual server role.
All this information is persisted in this file and is picked up along with the application whenever Pramati Server uses this application.
--> 
<!ELEMENT pramati-j2ee-server (description?, vhost-name, auto-start?, client-jar-path?, realm-name?, (ejb-module | web-module)+, role-mapping*)>

<!-- query-mapping element appears if the ejb is an ejb1.1 bean. It is used to map the querie-name in the queries.props to the finder method-name. It may also contain some method-param elements.
Used in: ejb
--> 
<!ELEMENT query-mapping (method-name, query-name, method-param*)>

<!-- represents the name of query which is present in the queries.props and have to be mapped to the method-name in the query-mapping element.
Used in: query-mapping
--> 
<!ELEMENT query-name (#PCDATA)>

<!-- This element specifies the security realm on which the application is deployed the default realm is 'system'.
-->
Used in: pramati-j2ee-server
-->
<!ELEMENT realm-name (#PCDATA)>

<!--
This element gives the details for a resource env reference.
Used in: ejb,web-module
-->
<!ELEMENT resource-env-ref (resource-env-ref-name, resource-env-ref-type?,
resource-env-ref-link)>

<!--
This element specifies the link for the resource env ref declared.
Used in: resource-env-ref
-->
<!ELEMENT resource-env-ref-link (#PCDATA)>

<!--
This element specifies the name for the resource env ref declared.
Used in: resource-env-ref
-->
<!ELEMENT resource-env-ref-name (#PCDATA)>

<!--
This element specifies the type of the resource env ref declared.
Used in: resource-mapping
-->
<!ELEMENT resource-env-ref-type (#PCDATA)>

<!--
This element specifies the resource link in the server in the case of
resource-mapping. This represents the actual resource name in the server
namespace.
Used in: resource-mapping
-->
<!ELEMENT resource-link (#PCDATA)>

<!--
This element gives the details about the ejb resource mapping onto a resource
on the server specifying the name and type of the resource.
Used in: ejb,web-module
-->
<!--
<!ELEMENT resource-mapping (resource-name, resource-type?, resource-link)>

<!--
This element gives the name of the resource used in resource mapping for an
EJB. This name is used in the application code.
Used in: resource-mapping
-->  
<!ELEMENT resource-name (#PCDATA)>

<!--
This element specifies the type of the resource used in resource mapping.
Used in: ejb
-->  
<!ELEMENT resource-type (#PCDATA)>

<!--
This element specifies the actual role on the server to which the role in the
module maps to.
Used in: role-mapping
-->  
<!ELEMENT role-link (#PCDATA)>

<!--
This element specifies the role mapping information for mapping module level
roles onto actual server roles.
Used in: pramati-j2ee-server
-->  
<!ELEMENT role-mapping (module-name, role-name, role-link)>

<!--
This element specifies the role name declared in the module which would be
mapped onto an actual role in the server.
Used in: role-mapping
-->  
<!ELEMENT role-name (#PCDATA)>

<!--
This element specifies the principal/identity that would be used to invoke
methods on the EJB.
Used in: ejb
-->
<!ELEMENT run-as-principal (#PCDATA)>

<!--
This element specifies the server-session pool details for a MessageDrivenBean.
Used in: ejb
-->
<!ELEMENT server-session (max-messages, min-pool-size, max-pool-size)>

<!--
session-timeout element is an optional element which appears if the ejb is a stateful session bean.
Used in: ejb
-->
<!ELEMENT session-timeout (#PCDATA)>

<!--
This tag has been deprecated and is now not used internally.
This element specifies the details for the thread pool used for a MessageDrivenBean for an EJB.
Used in: ejb
-->
<!ELEMENT thread-pool (min-pool-size, max-pool-size)>

<!--
This element specifies the virtual host on which the application is deployed.
Example: <vhost-name>www1.pramati.com</vhost-name>
by default 'default' is the value given and all applications are deployed using the same.
Used in: pramati-j2ee-server
-->
<!ELEMENT vhost-name (#PCDATA)>

<!--
This element contains the information for a web-module for the application
This module contains related information for the Web components.
Used in: pramati-j2ee-server
-->
<!ELEMENT web-module (name, module-name, ejb-ref*, ejb-local-ref*, resource-mapping*, resource-env-ref*)>
New Tag added in Pramati Server 6.0.
Used in: pramati-j2ee-server
-->
<!ELEMENT ejb-module (name, enable-freepool?, pool-waittimeout-millis?, pk-waittimeout-millis?, lock-bmp?, sfsb-allow-transactional-remove?, transport-type*, ejb+, message-destination-mapping*)>
<!ELEMENT message-destination-mapping (message-destination-name, message-destination-link)>
<!ATTLIST pramati-j2ee-server version CDATA #IMPLIED>pramati-or-map_3_0.dtd
pramati-or-map_3_0.dtd is a proprietary DTD used to perform O-R mapping before applications are deployed. It is located at <install_dir>/server/templates/dtds.

**DTD for Pramati OR-mapping**

A proprietary DTD, pramati-or-map_3_0.dtd is used for O-R mapping before deploying applications. It is located at <install_dir>/server/templates/dtds.

<!--
This is the XML DTD for the Pramati O-R mapping document. All Pramati O-R mapping documents must include a DOCTYPE of the following form:
<!DOCTYPE or-mapping PUBLIC '-//Pramati Technologies //DTD Pramati OR Map 3.0//EN' 'http://www.pramati.com/dtd/pramati-or-map_3_0.dtd'>
-->
<!--
The following conventions apply to all Pramati O-R mapping document elements unless indicated otherwise.
- In elements that contain PCDATA, leading and trailing whitespace in the data may be ignored.
- In elements whose value is an "enumerated type", the value is case sensitive.
-->
<!-- cmp-field element represents the name of the container managed field of the entity bean.
Used in: field-mapping
-->  
<!ELEMENT cmp-field (#PCDATA)>

<!-- Deprecated tag. Kept to have backward compatibility of pramati-or-mapping.xml. Now the functionality corresponding to this has been replaced by isolation-level tag.-->
<!ELEMENT concurrency-type (#PCDATA)>

<!ELEMENT datasource-name (#PCDATA)>

<!ELEMENT db-field (#PCDATA)>

<!ELEMENT deferred-key (db-field)>


<!ELEMENT ejb-jar (jar-name, ejb*, relationship*)>

<!ELEMENT ejb-name (#PCDATA)>

<!-- insert-strategy element can have three values:
1. ejbcreate - represents that the record must be created in the persistent store after ejbCreate call on the bean.
2. ejbpostcreate - represents that the record must be created in the persistent store after ejbPostCreate call on the bean.
3. end-of-transaction - represents that the record must be created in the persistent store at the end of the transaction.
Default value: ejbcreate
Used in: ejb

<!-- verified-update-strategy element can have three values and is used when isolation-level value is optimistic-repeatable-read.
1. all - update query compares all the cmp-fields for old values.
2. modified - update query compares only modified cmp-fields in a transaction for old values.
Default value: modified
Used in: ejb

<!-- update-strategy element can have two values
1. all - While updating persistent store, update all cmp fields.
2. modified - While updating persistent store, update only modified cmp fields.
Default value: all
Used in: ejb

<!-- exclusion-type element can have two values
1. exclusive - represents that the bean exclusively accesses and updates the table and optimizations can be made based on the assumption that no other source modifies the database table.
2. non-exclusive - represents that the database table can be accessed by some other sources than just the bean.
Used in: ejb

<!ELEMENT insert-strategy (#PCDATA)>
<!ELEMENT verified-update-strategy (#PCDATA)>
<!ELEMENT update-strategy (#PCDATA)>
<!ELEMENT exclusion-type (#PCDATA)>
<!-- field element represents the values of lhs or rhs table field in the join condition. 
Used in: lhs, rhs -->
<!ELEMENT field (#PCDATA)>

<!-- field-mapping element represents a field mapping node consisting of a cmp-field vs db-field mapping. 
Used in: ejb -->
<!ELEMENT field-mapping (cmp-field, db-field)>

<!-- isolation-level element represents the isolation level to be followed for accessing and updating the bean. For ejb1.1 beans currently supported value for isolation level are repeatable-read, optimistic-repeatable-read, read-commited. 
For ejb2.0 the only supported values for isolation-level is repeatable-read. Please refer to Pramati Server Documentation for details. 
Used in: ejb -->
<!ELEMENT isolation-level (#PCDATA)>

<!-- jar-name element represents the name of a jar in the enterprise archive. 
Used in: ejb-jar -->
<!ELEMENT jar-name (#PCDATA)>

<!-- join element encapsulates a set of join conditions for the given relationship. 
Used in: relationship -->
<!ELEMENT join (join-condition+)>

<!-- join-condition element represents one join condition for a relationship. 
Used in: join -->
<!ELEMENT join-condition (lhs, rhs)>
<!-- lhs element represents the left hand side part of the join condition. It consists of a table name and a field name to be equated to the right hand side counterpart. Used in: join-condition -->
<!ELEMENT lhs (table, schema?, field)>

<!-- or-mapping element is the root element of Pramati O-R mapping (pramati-or-map.xml) document. -->
<!ELEMENT or-mapping (ejb-jar*)>

<!-- relationship element represents a relationship node with one or more join conditions encapsulated in a join node. Used in: ejb-jar -->
<!ELEMENT relationship (relationship-name, join+)>

<!-- relationship-name represents the name of the relationship. Used in: relationship -->
<!ELEMENT relationship-name (#PCDATA)>

<!-- rhs element represents the right hand side of the equality in the join condition. It consists of a table name and a table field. Used in: join-condition -->
<!ELEMENT rhs (table, schema?, field)>

<!-- schema element represents the values of lhs or rhs schema in the join condition. Used in: rhs, lhs -->
<!ELEMENT schema (#PCDATA)>

<!-- schema-name element represents the name of the schema which contains the table to which the bean is mapped for its field mapping. Used in: ejb -->
<!ELEMENT schema-name (#PCDATA)>
<!-- table element represents the values of lhs or rhs table in the join condition. Used in: rhs, lhs -->
<!ELEMENT table (#PCDATA)>

<!-- table-name element represents the name of the table to which the bean is mapped to for its field mapping. Used in: ejb -->
<!ELEMENT table-name (#PCDATA)>
<!ATTLIST or-mapping version CDATA #IMPLIED>