Oracle® Fusion Middleware

Developing Applications Using Continuous Integration

14*c* (14.1.2.0.0)
F85500-04
May 2025

Oracle Fusion Middleware Developing Applications Using Continuous Integration, 14*c* (14.1.2.0.0)

F85500-04

Primary Author: Oracle Corporation

Contents

7 Building Oracle Service Bus Projects with Maven

8 Building a Real Application with Maven

List of Figures

List of Tables

Preface

This book describes build automation and continuous integration for applications that you develop and deploy to a Fusion Middleware runtime environment. This book describes the features in Fusion Middleware 14*c* to make it easier for users to automate application build and test and to adopt continuous integration techniques with Fusion Middleware.

Audience

This document is intended for developers and build managers who are responsible for building applications that will be deployed into a Fusion Middleware runtime environment and who want to automate their build processes or adopt, or both continuous integration techniques in the context of Fusion Middleware.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

See the following documents in the Oracle Fusion Middleware documentation set:

- *Developing Applications for Oracle WebLogic Server*
- *Developing Applications with Oracle JDeveloper*
- *Developing Services with Oracle Service Bus*
- *Developing SOA Applications with Oracle SOA Suite*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Automating Application Builds With Maven

Oracle Fusion Middleware (FMW) application developers typically use an Integrated Development Environment (IDE), such as Oracle JDeveloper, Eclipse IDE, or JetBrains IntelliJ IDEA during the development of their application. To ensure application quality, industry best practices suggest using a Continuous Integration (CI) system to automate building and testing application changes.
This typically requires the application builds to be automated so that the CI system can produce the application binaries. Apache Maven is one of the most popular build tools for Java-based applications. As such, most application-centric FMW products provide Maven plug-ins, archetypes, and Project Object Models (POMs) to support automating application builds with Maven.

Oracle FMW artifacts for this release are not published in public Maven repositories, such as the Oracle Maven Repository. You must populate your Maven repository with the Oracle FMW artifacts required prior to trying to build FMW applications with Maven.

Installing and Configuring Maven

To get started with Maven, download and install a Maven distribution from https://maven.apache.org. To use Maven, set the following environment variables to point to the JDK and Maven installation directories and add them to the PATH.

On Linux:

```
export JAVA_HOME=/usr/lib/jvm/jdk-17
export M2_HOME=/usr/lib/apache-maven
export PATH=${M2_HOME}/bin:${JAVA_HOME}/bin:${PATH}
```

On macOS:

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home
export M2_HOME=/opt/apache-maven
export PATH=${M2_HOME}/bin:${JAVA_HOME}/bin:${PATH}
```

On Windows:

```
set "JAVA_HOME=c:\java\jdk-17"
set "M2_HOME=c:\apache-maven"
set "PATH=%M2_HOME%\bin;%JAVA_HOME%\bin;%PATH%"
```

Verify your environment is properly configured by running the following command to ensure that you are picking up the expected versions of both Maven and Java:

```
mvn -v
Apache Maven 3.9.9 (8e8579a9e76f7d015ee5ec7bfcdc97d260186937)
Maven home: /scratch/maven/apache-maven-3.9.9
Java version: 17.0.9.0.3, vendor: Oracle Corporation, runtime: /scratch/java/
jdk-17.0.9.0.3
Default locale: en_US, platform encoding: UTF-8
```

```
OS name: "linux", version: "5.4.17-2136.308.9.el8uek.x86_64", arch: "amd64", family:
"unix"
```

Customizing Maven Settings

Maven uses a settings file to customize Maven to work properly in your environment. You will typically want to create a `settings.xml` file in the following situations:

- You require the use of a proxy server to reach the Internet.
- Your organization requires you to use their internal Maven Repository Manager.
- You need to deploy artifacts into your organization's Maven Repository Manager.

If this is the first time you have used Maven, then typically you will not have a Maven settings file. The Maven settings file, `settings.xml`, is usually kept in the `.m2` directory inside your home directory. However, if you want to point Maven to a different location, then see the Maven documentation.

Here is a sample settings file.

```
<settings
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0 http://maven.apache.org/xsd/
settings-1.1.0.xsd"
  xmlns="http://maven.apache.org/SETTINGS/1.1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <servers>
    <server>
      <id>artifactory</id>
      <username>fred</username>
      <password>gV4akMViBM4vVWxG4F9GTu</password>
    </server>
  </servers>

  <proxies>
    <proxy>
      <id>myproxy</id>
      <protocol>http</protocol>
      <host>myproxy.mycompany.com</host>
      <port>80</port>
      <nonProxyHosts>*.mycompany.com</nonProxyHosts>
      <active>true</active>
    </proxy>
  </proxies>

  <mirrors>
    <mirror>
      <id>repo1</id>
      <name>repo1</name>
      <url>https://artifactory.mycompany.com/artifactory/repo1</url>
      <mirrorOf>central</mirrorOf>
    </mirror>
  </mirrors>
</settings>
```

This example shows three common configuration settings that you may need to use:

- **Proxies**: Enables you to communicate with Maven about the HTTP proxy server that is required to access Maven repositories on the Internet.
- **Servers**: Enables you to communicate with Maven about your credentials for the Maven repository, so that you do not have to enter them every time you want to access the repository.

- **Mirrors**: Informs Maven that instead of trying to access the Maven central repository directly, it should use your organization's Maven repository manager as a mirror (cache) of Maven's central repository.

The Maven settings file can be used for other situations beyond the scope of this document. While some configuration can be placed in either `settings.xml` or the project's `pom.xml` files, Oracle recommends that all project-specific settings be placed in `pom.xml` and leave `settings.xml` for cross-project configuration. For more information, see https://maven.apache.org/settings.html.

Populating the Maven Repository

Whether using your organization's Maven Repository Manager or simply using the local Maven repository, which is located by default in `$HOME/.m2/repository`, you must populate it with the Oracle FMW artifacts prior to trying to build FMW applications with Maven.

Oracle provides the Oracle Maven Synchronization plug-in that allows you to populate the Maven repository from an Oracle Home. When you install a Fusion Middleware product, the Maven archetypes, plug-ins, and POMs are installed with the product so that the Oracle Maven Synchronization plug-in can find them.

This section contains the following topics:

Introduction to the Maven Synchronization Plug-In

Oracle Fusion Middleware provides the Oracle Maven Synchronization plug-in that simplifies the process of setting up repositories and completely eliminates the need to know what patches are installed in a particular environment. This plug-in enables you to populate a Maven repository from a given Oracle home. After you patch your Oracle home, run this plug-in to ensure that your Maven repository matches Oracle home. This ensures that your builds use correct versions of all artifacts in that particular environment.

The Oracle Maven Synchronization plug-in is included in the Oracle WebLogic Server, Oracle Coherence, and Oracle JDeveloper installations. To use the plug-in, you must specify the location of the Oracle home and the location of the Maven repository. The Maven repository can be specified using either a file system path or a URL. The plug-in checks for all Maven artifacts in the Oracle home, ensures that all artifacts are installed in the specified Maven repository, and that the versions match exactly. This means that the version numbers and the files are exactly the same at the binary level, ensuring that all patched files reflect accurately in the Maven repository.

Oracle homes contain `plugins/maven` directories which contain Maven POMs for the artifacts provided by Oracle, archetypes for creating projects, and Maven plug-ins provided by Oracle, for executing various build operations.

Installing the Oracle Maven Synchronization Plug-In

Before you start using the Oracle Maven Synchronization plug-in, you must install it into your Maven repository. You can install it into your local repository on your computer, or you can deploy it into a shared internal repository in your organization's Maven Repository Manager, if you have one.

The plug-in is located in your Oracle WebLogic Server home and consists of two files:

- The Maven Project Object Model (POM) file, which describes the plug-in. It is located at:

  ```
  ORACLE_HOME/oracle_common/plugins/maven/com/oracle/maven/oracle-maven-sync/
  14.1.2/oracle-maven-sync-14.1.2.pom
  ```

- The JAR file, which contains the plug-in. It is located at:

  ```
  ORACLE_HOME/oracle_common/plugins/maven/com/oracle/maven/oracle-maven-sync/
  14.1.2/oracle-maven-sync-14.1.2.jar
  ```

Install and deploy the plug-in.

1. To install the plug-in into your local Maven repository, run the following command from the `ORACLE_HOME/oracle_common/plugins/maven/com/oracle/maven/oracle-maven-sync/14.1.2` directory:

   ```
   mvn install:install-file -DpomFile=oracle-maven-sync-14.1.2.pom -
   Dfile=oracle-maven-sync-14.1.2.jar
   ```

2. To deploy the plug-in, use one of the following methods:
   - The simplest way to deploy the plug-in into a shared internal repository is to use the web user interface provided by your Maven Repository Manager to upload the JAR and POM files into the repository.
   - An alternative method is to use the deploy plug-in, which you can do by using a command like the following, from the `ORACLE_HOME/oracle_common/plugins/maven/com/oracle/maven/oracle-maven-sync/14.1.2` directory:

     ```
     mvn deploy:deploy-file -DpomFile=oracle-maven-sync-14.1.2.pom -
     Dfile=oraclemaven-sync-14.1.2.jar -Durl=http://servername/artifactory/
     repositories/internal -DrepositoryId=internal
     ```

     To use the deploy plug-in, you must define the repository in your Maven `settings.xml` file and define the credentials if anonymous publishing is not allowed. For information about this command, refer to the Maven documentation at `http://maven.apache.org/plugins/maven-deploy-plugin/deploy-file-mojo.html`.

If you would like to use the shorter name for the Oracle Maven Synchronization plug-in, so that you do not have to provide the full coordinates when using it, add an entry to your Maven `settings.xml` as follows:

```
<pluginGroups>
    <pluginGroup>com.oracle.maven</pluginGroup>
</pluginGroups>
```

This allows you to refer to the plug-in using the name `oracle-sync`.

Running the Oracle Maven Synchronization Plug-In

The Oracle Maven Synchronization plug-in supports a single push goal used to populate a repository.

To obtain usage and parameter descriptions, invoke the `help:describe` goal by running the following command:

```
mvn help:describe -Dplugin=com.oracle.maven:oracle-maven-sync -Ddetail
```

This output shows the parameters that are available for the plug-in's push goal. Table 1-1 describes the parameters.

Table 1-1 Push Goal Parameters and Description

Parameter	Description
`dryRun`	A parameter to control whether the plug-in attempts to publish the artifacts to the repository. If you set this to `true`, then the push goal finds all of your POM files and prints out details of what would have been done if this was set to `false`. However, it does not publish any artifacts or make any change to the system.
`failOnError`	If you set this property to `false` and the plug-in fails to process a resource, it continues to process all other resources. Failures are logged as warnings, but the process completes successfully. If you set this property to `true`, when it encounters the first problem, the plug-in immediately exits with an error. This is the default.
`oracleHome`	The path to the Oracle home from which you want to populate the Maven repository.
`overwriteParent`	If you set this property to `true`, the plug-in overwrites POM artifacts with ancestry to oracle-common if they exist in the target repository. The default value of false prevents automatic overwrite of customized POM contents. If any such POMs are encountered during plug-in execution, an error is thrown and handled according to the `failOnError` flag value. To carry over changes, save the existing POMs, run the push goal with `overwriteParent=true`, and manually transfer the changes to the newly pushed POMs.
`pushDuplicates`	If you set this property to `true`, then the plug-in pushes all duplicate locations. That is, if multiple POMs with different Maven coordinates (GAV) are assigned to the same location path, the plug-in pushes them all to the destination repository.
`retryFailedDeploymentCount`	Parameter used to control how many times a failed deployment will be retried before giving up and failing. If a value outside the range 0-10 is specified, then it will be pulled to the nearest value within the range 0-10. If set to 0, there will be no retries. If the artifact being deployed to the remote server has both a POM and a binary file, then the retry count will be reset between deploying the POM and deploying the binary file. This parameter is ignored when pushing to a local repository.
`serverId`	The ID of the server entry in your Maven `settings.xml` file. This is required only if you intend to deploy to a remote repository. The `settings.xml` file should provide the remote artifact repository's deployment information, such as URL, user name, and password.

Replacing Artifacts

Some Maven Repository Managers have a setting that controls whether you can replace an existing artifact in the repository. If your Maven Repository Manager has such a setting, you must ensure that you have set it correctly so that the Oracle Maven Synchronization plug-in is able to update the artifacts in your repository. See your Maven Repository Manager documentation for how to change this setting.

Populating Your Maven Repository

To populate your repository, you must use the push goal. You can specify the parameters given in Table 1-1 on the command line or in your POM file.

This section contains the following topics:

About Running the Push Goal

When you run the push goal, it takes the following actions:

- Checks the Oracle home that you have provided and makes a list of all of the Maven artifacts inside that Oracle home. This is done by looking for POM files in the `ORACLE_HOME/oracle_common/plugins/maven` dependencies directory and its subdirectories, recursively, and in the `ORACLE_HOME/PRODUCT_HOME/plugins/maven` directory and its subdirectories, recursively, for each `PRODUCT_HOME` that exists in the `ORACLE_HOME`.
- Checks if the JAR file referred to by each POM file is available in the Oracle home.
- Calculates an SHA1 checksum for the JAR file.
- Attempts to publish the JAR, POM, and SHA1 files to the repository that you have provided.

The following types of Maven artifacts are installed into your repository:

- Maven dependencies provided by Oracle, which include the following:
  - Client API classes
  - Compilation, packaging, and deployment utilities, for example `wlst`
  - Component JARs that must be embedded in the application
  - Client-side runtime classes, for example, t3 and JAX-WS client runtimes
- Maven plug-ins provided by Oracle that handle compilation, packaging, and deployment
- Maven archetypes provided by Oracle that provide project templates

Populating a Local Repository

To populate a local repository, you must specify `oracleHome` on the command line. For example:

```
mvn com.oracle.maven:oracle-maven-sync:push -DoracleHome=path_to_oracleHome
```

The `localRepository` element in your `settings.xml` file indicates the location of your local Maven repository. If you exclude the `localRepository` element in `settings.xml`, the default location is in the `${HOME}/.m2/repository` directory.

If you want to override the `localRepository` value, then you must specify the override location on the command line as a Maven option. For example:

```
mvn com.oracle.maven:oracle-maven-sync:push
-DoracleHome=path_to_oracleHome -Dmaven.repo.local=alternate_path
```

To specify the parameters in your POM file, you must add a plug-in entry similar to the following:

```
<plugin>
    <groupId>com.oracle.maven</groupId>
    <artifactId>oracle-maven-sync</artifactId>
    <version>14.1.2-0-0</version>
    <configuration>
        <oracleHome>path_to_oracleHome</oracleHome>
    </configuration>
</plugin>
```

After adding the plug-in, run Maven with the following command:

```
mvn com.oracle.maven:oracle-maven-sync:push
```

Populating a Remote Repository

To populate a remote repository, you must specify `serverId` and `oracleHome` on the command-line interface or in the plug-in configuration. You must also have a repository configuration in your `settings.xml` file that matches the `serverId` you provide to the plug-in. If authentication is required for deployment, then you must also add a `server` entry to your Maven `settings.xml` file.

For example:

```
mvn com.oracle.maven:oracle-maven-sync:push -DoracleHome=path_to_oracleHome -
DserverId=internal
```

The corresponding Maven `settings.xml` file with authentication details looks similar to the following:

```
...
<profiles>
  <profile>
    <id>default</id>
    <repositories>
      <repository>
        <id>internal</id>
```

```
        <name>Team Internal Repository</name>
        <url>http://some.host/maven/repo/internal</url>
        <layout>default</layout>
        <releases>
          <enabled>true</enabled>
          <updatePolicy>always</updatePolicy>
          <checksumPolicy>warn</checksumPolicy>
        </releases>
        <snapshots>
          <enabled>true</enabled>
          <updatePolicy>never</updatePolicy>
          <checksumPolicy>fail</checksumPolicy>
        </snapshots>
      </repository>
    </repositories>
  </profile>
</profiles>
...
<server>
  <id>internal</id>
  <username>username</username>
  <password>password</password>
</server>
...
<activeProfiles>
  <activeProfile>default</activeProfile>
</activeProfiles>
```

You must define the target repository in a profile and activate that profile using the `activeProfiles` tag, as shown in the preceding example.

> **Note:**
>
> You should specify an encrypted password in the server section. For details on how to encrypt the server passwords, see: https://maven.apache.org/guides/mini/guide-encryption.html#How_to_encrypt_server_passwords.

To specify the parameters in your POM file, you must add a plug-in entry similar to the following:

```
<plugin>
  <groupId>com.oracle.maven</groupId>
  <artifactId>oracle-maven-sync</artifactId>
  <version>14.1.2-0-0</version>
  <configuration>
    <serverId>internal</serverId>
    <oracleHome>path_to_oracleHome</oracleHome>
  </configuration>
</plugin>
```

After adding the plug-in, run Maven with the following command:

```
mvn com.oracle.maven:oracle-maven-sync:push
```

After you have populated the repository, you may want to perform some operations on the repository manager, such as update indexes or update the archetype catalog. Refer to the

documentation for the repository manager to check if any such operations are necessary or recommended.

Running the Push Goal on an Existing Maven Repository

When you run the push goal against a Maven repository that already has Oracle artifacts in it, the Oracle Maven Synchronization plug-in detects that you have existing Parent POMs in the repository. It does not overwrite these Parent POMs, in case you have modified them, for example, by adding your own settings to them. Instead, it prints a warning message. If you want to overwrite the Parent POMs, then you need to specify the extra parameter `-DoverwriteParent=true` on the push goal.

Things to Know About Patching

Patching is the practice of updating a system with minor changes, usually to fix bugs that have been identified after the software goes into production. Oracle Fusion Middleware uses the OPatch utility to manage the application of patches to installed software in the Oracle home. When you use OPatch to apply a patch, the version number of the installed software may not change.

Maven uses a different approach to patching which assumes that released software will never be changed. When a patch is necessary, a new version of the artifact, with a new version number, is created and distributed as the patch.

This difference creates an issue when you use Maven to develop applications in an Oracle Fusion Middleware environment. Oracle Fusion Middleware provides a mechanism to address this issue.

- Oracle Approach to Patching
- Run the Oracle Maven Synchronization Plug-In Push Goal After Patching

Oracle Approach to Patching

If any problems are found after a release of Oracle Fusion Middleware (for example, 14.1.1) into production, a one-off patch is created to fix that problem. Between any two releases, for example 14.1.1 and 14.1.2, a number of these patches are released. You can apply many combinations of these patches, including all or none of these patches.

This approach gives you a great deal of flexibility. You can apply only the patches that you need, and ignore the rest. However, it can create an issue when you are using Maven. Ensure that the artifacts you are using in your build system are the exact same (potentially patched) versions that are used in the target environment.

The complications arise when you have a number of environments, like test, QA, SIT, and production, which are likely to have different versions (or patches) installed.

Oracle recommends that, in such a situation, you set up one Maven repository for each environment that you want to target. For example, a Maven test repository that contains artifacts that match the versions and patches installed in the test environment and a Maven QA repository that contains artifacts that match the versions and patches installed in the QA environment.

Run the Oracle Maven Synchronization Plug-In Push Goal After Patching

After you patch your Oracle home, run this plug-in to ensure that your Maven repository matches the Oracle home. This ensures that your builds use correct versions for all artifacts in that particular environment. See Running the Oracle Maven Synchronization Plug-In.

Considerations for Archetype Catalogs

By running the Oracle Maven Synchronization plug-in push goal, you may have installed new Maven archetypes into your Maven repository. You might need to run a command to rebuild the index of archetypes. Some Maven Repository Managers do this automatically.

To rebuild your local archetype catalog, run a command similar to the following:

```
mvn archetype:crawl -Dcatalog=$HOME/.m2/archetype-catalog.xml
```

2

Customizing the Build Process with Maven POM Inheritance

Oracle provides a set of common parent Project Object Models (POMs) to enable easy customization of the build process for all projects targeted at a particular product, runtime environment, or for all projects targeted at Oracle Fusion Middleware.

Each of the Oracle-provided Maven archetypes have their parent POM set to an Oracle-provided common parent specific to the target runtime environment that the archetype is for, such as WebLogic Server and Coherence. The common parent POMs, one per product or target runtime environment, in turn have their parent POM set to an Oracle Fusion Middleware common parent.

Inheritance of POMs and Archetypes

The common POMs and Oracle-provided archetypes form the following inheritance hierarchy:

```
com.oracle.maven:oracle-common:14.1.2-0-0
  - com.oracle.weblogic.archetype:wls-common:14.1.2-0-0
    - com.oracle.weblogic.archetype:basic-webapp:14.1.2-0-0
    - com.oracle.weblogic.archetype:basic-webapp-ejb:14.1.2-0-0
    - com.oracle.weblogic.archetype:basic-webservice:14.1.2-0-0
    - com.oracle.weblogic.archetype:basic-mdb:14.1.2-0-0
  - com.oracle.coherence:gar-common:14.1.2-0-0
    - com.oracle.coherence:maven-gar-archetype:14.1.2-0-0
  - com.oracle.soa:sar-common:14.1.2-0-0
    - com.oracle.soa.archetype:oracle-soa-application:14.1.2-0-0
    - com.oracle.soa.archetype:oracle-soa-project:14.1.2-0-0
  - com.oracle.servicebus:project:14.1.2-0-0
    - com.oracle.servicebus:sbar-project-common:14.1.2-0-0
    - com.oracle.servicebus.archetype:oracle-servicebus-project:14.1.2-0-0
    - com.oracle.servicebus:sbar-system-common:14.1.2-0-0
  - com.oracle.adf:adf-parent:14.1.2-0-0
    - com.oracle.adf.archetype:oracle-adffaces-ejb:14.1.2-0-0
```

Customizing the Build Process

If you want to customize your build process, for example, setting some default properties, setting up default settings for a particular plug-in, or defining Maven profiles, then you can add your definitions to the appropriate parent POM.

For example, if you add definitions to `com.oracle.weblogic.archetype:wls-common:14.1.2-0-0`, all projects associated with this parent are affected, which includes all projects that you have created from the WebLogic Maven archetypes (unless you modify their parents) and projects that you have created manually.

Doing this minimizes the number of settings needed in each project POM. For example, if you are going to deploy all of the builds to the same test server, then you can provide the details for

the test server by adding the appropriate build, plug-ins, and plug-in section for `com.oracle.weblogic:weblogic-maven-plugin:14.1.2-0-0` as shown in the following example of a customized parent WebLogic POM:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0   http://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.oracle.weblogic.archetype</groupId>
  <artifactId>wls-common</artifactId>
  <version>14.1.2-0-0</version>
  <packaging>pom</packaging>

  <name>wls-common</name>

  <parent>
    <groupId>com.oracle.maven</groupId>
    <artifactId>oracle-common</artifactId>
    <version>14.1.2-0-0</version>
  </parent>

  <build>
    <plugins>
      <plugin>
        <groupId>com.oracle.weblogic</groupId>
        <artifactId>weblogic-maven-plugin </artifactId>
        <version>14.1.2-0-0</version>
        <executions>
          <execution>
            <id>pre-integration-tests-deploy</id>
            <phase>pre-integration-test</phase>
            <goals>
              <goal>deploy</goal>
            </goals>
            <configuration>
              <user>weblogic</user>
              <password>password</password>
              <verbose>true</verbose>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Similarly, if you want to affect all projects targeted at any Oracle Fusion Middleware runtime, place your customizations in `com.oracle.maven:oracle-common:14.1.2-0-0`.

If you are using a shared internal repository, then after you customize the parent POMs, publish them into your shared Maven repository or repositories.

To see how these customizations are brought into your projects, you can use the following command, from your project's directory, to see the full POM that will be used to build your project:

```
mvn help:effective-pom
```

If you want to define more than one set of shared properties in the parent POM, for example, one set for your test environment, and one for your QA environment, Oracle encourages you to explore the use of Maven profiles. See:

http://books.sonatype.com/mvnref-book/reference/index.html

Profiles enable you to switch various settings on for a particular build by setting a command-line argument, or based on the presence or absence of various properties in the POM.

3

Building Jakarta EE Projects for WebLogic Server with Maven

You can use the WebLogic Maven archetypes to create, build, and deploy WebLogic Server Jakarta EE applications.

- Introduction to Building a Jakarta EE Project with Maven
  A Maven plug-in and four archetypes are provided for Oracle WebLogic Server.
- Using the Basic WebApp Maven Archetype
  To build a Jakarta EE project using the basic WebApp Maven archetype, you create the basic project, then customize, compile, and package it. Then you deploy it and test it.
- Using the Basic WebApp with EJB Maven Archetype
  To build a Jakarta EE project using the basic WebApp with EJB Maven archetype, you create the basic project, then customize, compile, and package it. Then you deploy it and test it.
- Using the Basic WebService Maven Archetype
  To build a Jakarta EE project using the basic WebService Maven archetype, you create the basic project, then customize, compile, and package it. Then you deploy it and test it.
- Using the Basic MDB Maven Archetype
  To build a Jakarta EE project using the basic MDB Maven archetype, you create the basic project, then customize, compile, and package it. Then you deploy it and test it.

Introduction to Building a Jakarta EE Project with Maven

A Maven plug-in and four archetypes are provided for Oracle WebLogic Server.

Table 3-1 describes the Maven coordinates.

Table 3-1 Maven Coordinates with WebLogic Server

Name	GroupId	ArtifactId	Version
WebLogic Server plug-in	`com.oracle.weblogic`	`weblogic-maven-plugin`	`14.1.2-0-0`
Basic WebApp archetype	`com.oracle.weblogic.archetype`	`basic-webapp`	`14.1.2-0-0`
WebApp with EJB archetype	`com.oracle.weblogic.archetype`	`basic-webapp-ejb`	`14.1.2-0-0`
Basic MDB archetype	`com.oracle.weblogic.archetype`	`basic-mdb`	`14.1.2-0-0`
Basic WebServices archetype	`com.oracle.weblogic.archetype`	`basic-webservice`	`14.1.2-0-0`

As with Maven archetypes in general, the Oracle WebLogic Maven archetype provides a set of starting points and examples for building your own applications.

Using the Basic WebApp Maven Archetype

To build a Jakarta EE project using the basic WebApp Maven archetype, you create the basic project, then customize, compile, and package it. Then you deploy it and test it.

This section contains the following topics:

Creating a Basic WebApp Project

To create a new Basic WebApp project using the Maven archetype, you must issue a command similar to the following:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.oracle.weblogic.archetype \
    -DarchetypeArtifactId=basic-webapp \
    -DarchetypeVersion=14.1.2-0-0 \
    -DgroupId=org.mycompany \
    -DartifactId=my-basic-webapp-project \
    -Dversion=1.0-SNAPSHOT
```

This command runs Maven's archetype plug-in's generate goal, which enables you to create a new project from an archetype. Table 3-2 describes the parameters.

Table 3-2 Parameters for the Basic WebApp Project

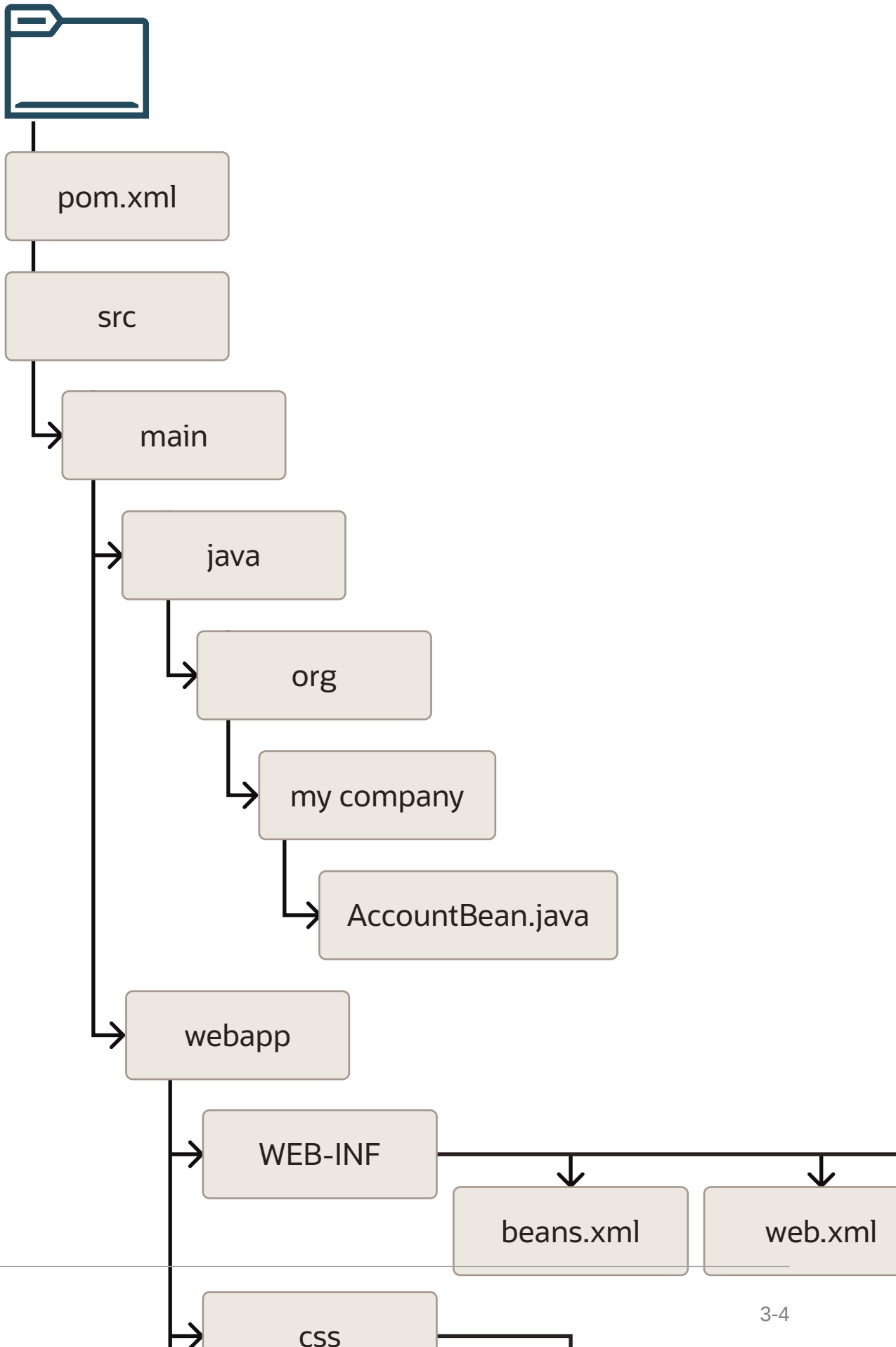
Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new project. This must be `com.oracle.weblogic.archetype`, as shown in the preceding example.
`archetypeArtifactId`	The archetype artifact ID of the archetype that you want to use to create the new project. This must be `basic-webapp`, as shown in the preceding example.
`archetypeVersion`	The version of the archetype that you want to use to create the new project. This must be `14.1.2-0-0`, as shown in the preceding example.
`groupId`	The group ID for your new project. This usually starts with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new project. This is usually an identifier for this project.
`version`	The version number for your new project. This is usually `1.0-SNAPSHOT` for a new project.

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

If you want to limit Maven to look only into a particular repository, you can specify the `-DarchetypeCatalog` option. Specify the value as `local` to look only in your local repository, or specify the `serverId` for the repository you want Maven to look in. This limits the number of archetypes that you are shown and makes the command run much faster.

After creating your project, it contains the following files:


pom.xml
src
main
java
org
my company
AccountBean.java
webapp
WEB-INF
beans.xml
web.xml
css

These files make up a small sample application, which you can deploy as is. You can use this application as a starting point for building your own application.

Table 3-3 describes the files included in the project.

Table 3-3 Files Created for the Basic WebApp project

File	Purpose
`pom.xml`	The Maven Project Object Model (POM) file that describes your new project. It includes the Maven coordinates that you specified for your project. It also includes the appropriate plug-in definitions needed to use the WebLogic Maven plug-in to build your project.
Files under `src/main/java`	An example Contexts and Dependency Injection (CDI) bean that is used by the Web application to store data.
Files under `src/main/webapp`	HTML and other files that make up the web application user interface.

After you have written your project code, you can use Maven to build the project. It is also possible to build the sample as is.

Customizing the Project Object Model File to Suit Your Environment

The Project Object Model (POM) file that is created by the archetype is sufficient in most cases. Review the POM and update any of the settings where the provided default values differ from what you use in your environment.

If you are using an internal Maven Repository Manager, like Artifactory, add a `pluginRepository` to the POM file. The following is an example; you can modify it to suit your environment:

```
<pluginRepositories>
  <pluginRepository>
    <id>artifactory-internal</id>
    <name>Artifactory Managed Internal Repository</name>
    <url>https://artifactory:8082/artifactory/internal/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
```

Compiling Your Jakarta EE Project

To compile the source code in your project, such as Java Beans, Servlets, and JSPs, use the following command:

```
mvn compile
```

This command uses the standard Maven plug-ins to compile your source artifacts into class files. You can find the class files in the `target` directory of your project.

> **Note:**
>
> When compiling the project, you may see the following warning:
>
> ```
> [WARNING] File encoding has not been set, using platform encoding UTF-8,
>       i.e. build is platform dependent!
> ```
>
> You can eliminate this warning by adding the following stanza to the `pom.xml` file.
>
> ```
> <properties>
>   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
> </properties>
> ```

Packaging Your Jakarta EE Project

To build the deployment archive, for example WAR or EAR file, use the following command:

```
mvn package
```

This command uses the standard Maven plug-ins to package your compiled artifacts and metadata into a deployment archive. When you run a Maven goal like `package`, Maven runs not just that goal, but all the goals up to and including the goal you name. If you are not familiar with the Maven build life cycle, for more information, see https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html.

This application is very similar to a standard Jakarta EE application, except that if you have WebLogic deployment descriptors in your project, they are also packaged into the deployment archive. The deployment archive, in this case a WAR file, is available in the `target` directory of your project.

Deploying Your Jakarta EE Project to WebLogic Server Using Maven

To deploy the deployment archive using Maven, use the following command:

```
mvn pre-integration-test
```

This command runs the `deploy` goal in the WebLogic Maven plug-in. This goal supports all standard types of deployment archives.

Deploying Your Jakarta EE Project to WebLogic Server Using Different Options

After you have packaged your project, you can also deploy it to WebLogic Server using any of the other existing (non-Maven) mechanisms. For example, the WebLogic Remote Console, the weblogic.Deployer command-line utility, or a WLST script.

Testing Your Basic WebApp Project

You can test the Basic WebApp by visiting the following URL on the WebLogic Server instance where you deployed it:

1. Enter the following URL:

   `http://servername:7001/basicWebapp/index.xhtml`

2. Provide the **Account Name** and **Amount**, then select **Deposit** to see how the application works.

Using the Basic WebApp with EJB Maven Archetype

To build a Jakarta EE project using the basic WebApp with EJB Maven archetype, you create the basic project, then customize, compile, and package it. Then you deploy it and test it.

To use the Basic WebApp with EJB project using the Maven archetype:

1. Create a new Basic WebApp project using the Maven archetype, running a command similar to the following:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.oracle.weblogic.archetype \
    -DarchetypeArtifactId=basic-webapp-ejb \
    -DarchetypeVersion=14.1.2-0-0 \
    -DgroupId=org.mycompany \
    -DartifactId=my-basic-webapp-ejb-project \
    -Dversion=1.0-SNAPSHOT
```

This command runs Maven's archetype plug-in's generate goal, which enables you to create a new project from an archetype.Table 3-4 describes the parameters.

Table 3-4 Parameters for the Basic WebApp with EJB Project

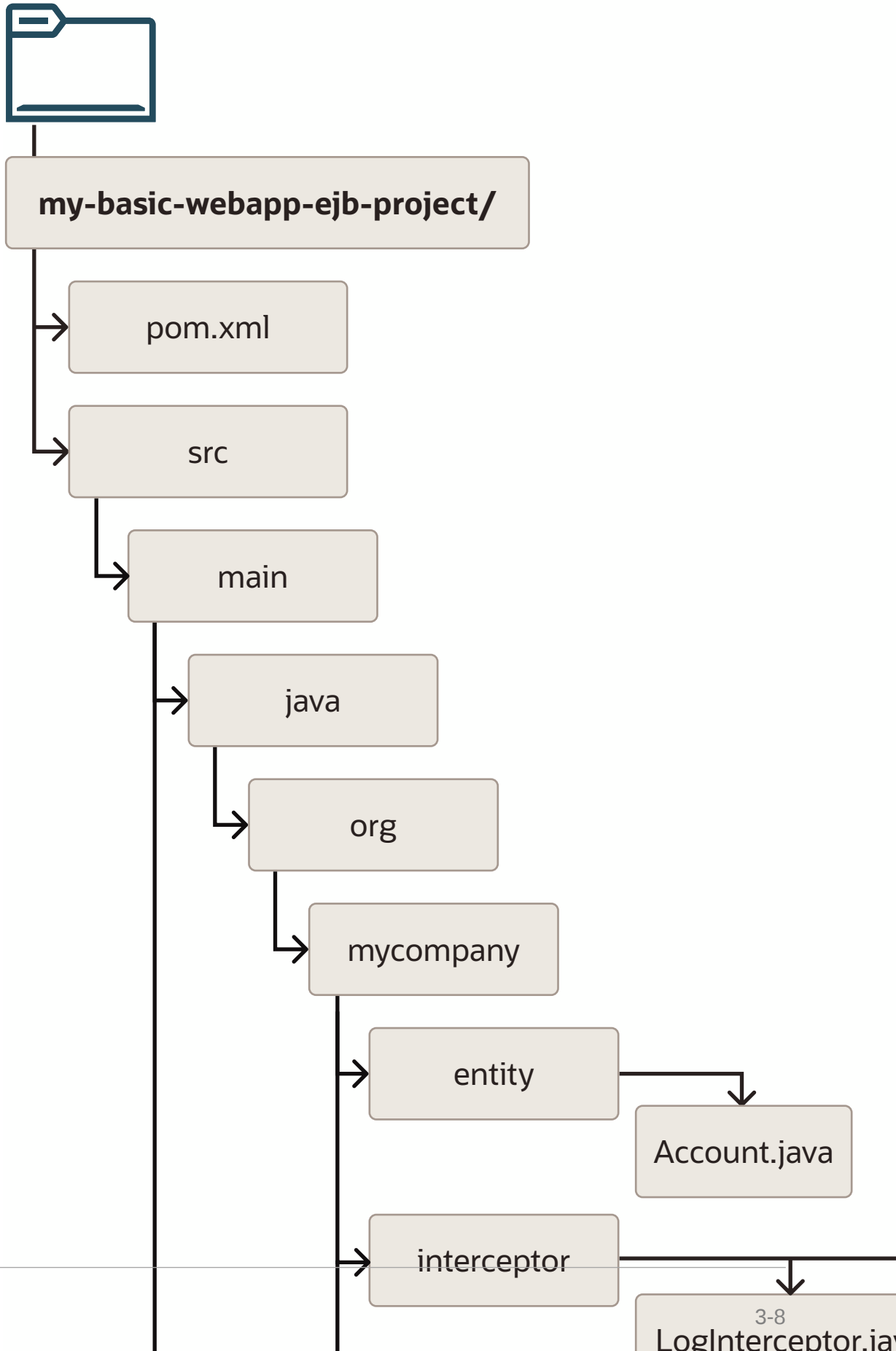
Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new project. This must be `com.oracle.weblogic.archetype`, as shown in the preceding example.
`archetypeArtifactId`	The artifact ID of the archetype that you want to use to create the new project. This must be `basic-webapp-ejb`, as shown in the preceding example.
`archetypeVersion`	The version of the archetype that you want to use to create the new project. This must be `14.1.2-0-0`, as shown in the preceding example.
`groupId`	The group ID for your new project. This usually starts with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new project. This is usually an identifier for this project.
`version`	The version number for your new project. This is usually `1.0-SNAPSHOT` for a new project.

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

After creating your project, it contains the following files:

Figure 3-1 Basic WebApp with EJB Maven Archetype



These files make up a small sample application, which you can deploy as is. You can use this application as a starting point for building your own application.

Table 3-5 describes the files included in the project.

Table 3-5 Files Created for the Basic WebApp with EJB Project

File	Purpose
`pom.xml`	The Maven Project Object Model (POM) file that describes your new project. It includes the Maven coordinates that you specified for your project. It also includes the appropriate plug-in definitions to use the WebLogic Maven plug-in to build your project.
Files under `src/main/java`	An example Enterprise Java Bean that is used by the web application to store data.
Files under `src/main/webapp`	HTML and other files that make up the web application user interface.

2. After you have written your project code, you can use Maven to build the project. It is also possible to build the sample as is.
3. Customize the POM to suit your environment. See Customizing the Project Object Model File to Suit Your Environment.
4. Compile your Basic WebApp with EJB Project. See Compiling Your Jakarta EE Project.
5. Package your Basic WebApp with EJB Project. See Packaging Your Jakarta EE Project.
6. Deploy your Basic WebApp with EJB Project. For information about deploying it using Maven, see Deploying Your Jakarta EE Project to WebLogic Server Using Maven. For information about deploying it using other options, see Deploying Your Jakarta EE Project to WebLogic Server Using Different Options.
7. Test your Basic WebApp with EJB Project.

   You can test the Basic WebApp with EJB by visiting the following URL on the WebLogic Server instance where you deployed it:

   `http://servername:7001/basicWebapp/index.xhtml`
8. Provide the **Account Name** and **Amount**, then select **Deposit** to see how the application works.

Using the Basic WebService Maven Archetype

To build a Jakarta EE project using the basic WebService Maven archetype, you create the basic project, then customize, compile, and package it. Then you deploy it and test it.

To use the Basic WebService project using the Maven Archetype:

1. Create a new Basic WebService project using the Maven archetype by issuing a command similar to the following:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.oracle.weblogic.archetype \
    -DarchetypeArtifactId=basic-webservice \
    -DarchetypeVersion=14.1.2-0-0 \
    -DgroupId=org.mycompany \
    -DartifactId=my-basic-webservice-project \
    -Dversion=1.0-SNAPSHOT
```

This command runs Maven's archetype plug-in's generate goal, which enables you to create a new project from an archetype. Table 3-6 describes the parameters.

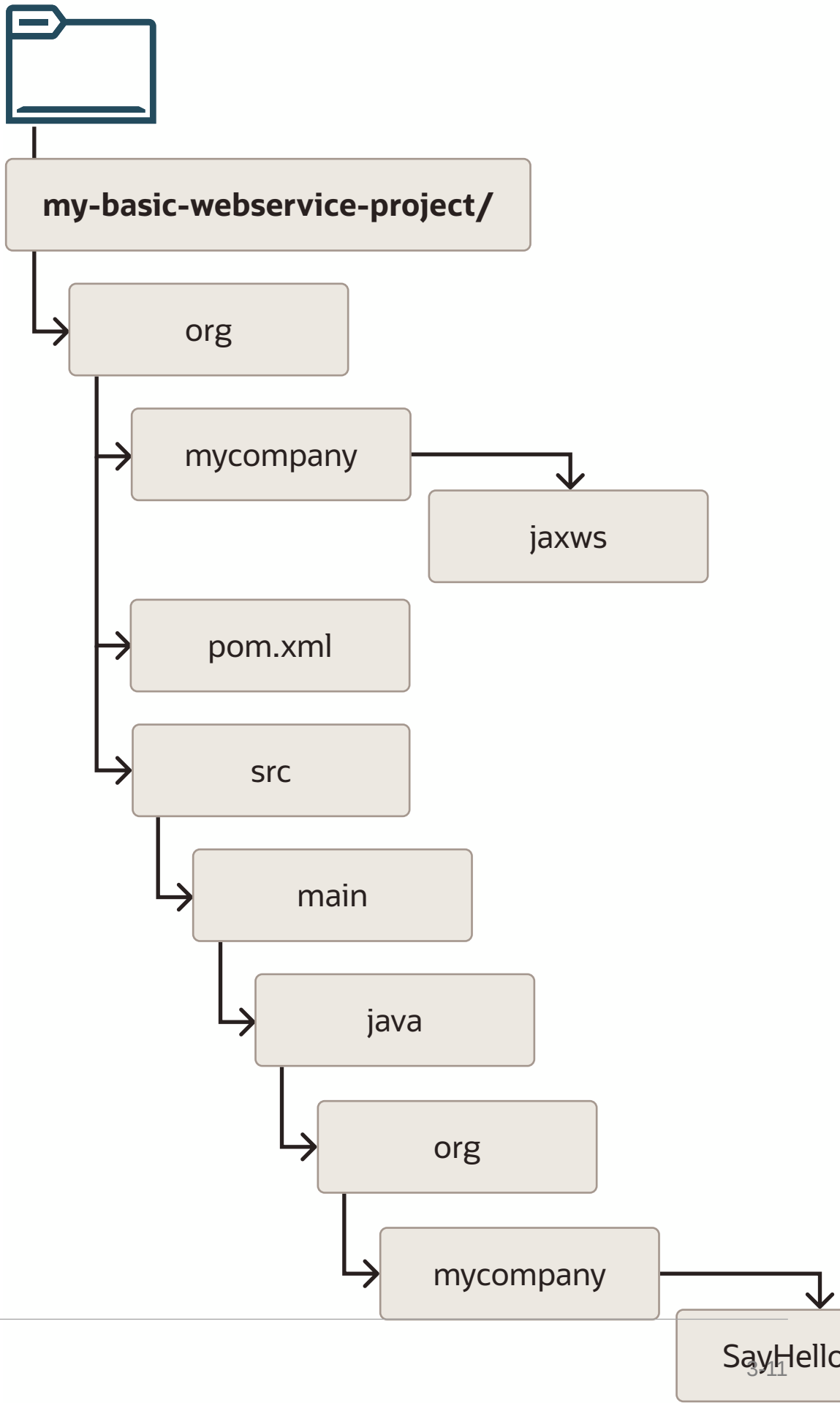
Table 3-6 Parameters for the Basic WebService Project

Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new project. This must be `com.oracle.weblogic.archetype`, as shown in the preceding example.
`archetypeArtifactId`	The artifact ID of the archetype that you want to use to create the new project. This must be `basic-webservice`, as shown in the preceding example.
`archetypeVersion`	The version of the archetype that you want to use to create the new project. This must be `14.1.2-0-0`, as shown in the preceding example.
`groupId`	The group ID for your new project. This usually starts with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new project. This is usually an identifier for this project.
`version`	The version number for your new project. This is usually `1.0-SNAPSHOT` for a new project.

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

After creating your project, it contains the following files:


my-basic-webservice-project/
org
mycompany
jaxws
pom.xml
src
main
java
org
mycompany
SayHello.java

These files make up a small sample application, which you can deploy as is. You can use this application as a starting point for building your own application.

Table 3-7 describes the files included in the project.

Table 3-7 Files Created for the Basic WebService Project

File	Purpose
`pom.xml`	The Maven Project Object Model (POM) file that describes your new project. It includes the Maven coordinates that you specified for your project, and it also includes the appropriate plug-in definitions to use the WebLogic Maven plug-in to build your project.
`SayHello.java`	An example Web Service.

2. After you have written your project code, you can use Maven to build the project. It is also possible to build the sample as is.
3. Customize the POM to suit your environment. See Customizing the Project Object Model File to Suit Your Environment.
4. Compile your Basic WebService Project. See Compiling Your Jakarta EE Project.
5. Package your Basic WebService Project. See Packaging Your Jakarta EE Project.
6. Deploy your Basic WebService Project. For information about deploying it using Maven, see Deploying Your Jakarta EE Project to WebLogic Server Using Maven. For information about deploying it using other options, see Deploying Your Jakarta EE Project to WebLogic Server Using Different Options.
7. Test your Basic WebService Project.

   You can test the Basic WebService by visiting the following URL, on the WebLogic Server instance where you have deployed it:

   `http://servername:7001/basicWebservice/SayHello`
8. You can access the WSDL for the web service, and you can open the WebLogic Web Services Test Client by selecting the **Test** link. This enables you to invoke the web service and observe the output.
9. To test the web service, select **SayHello** operation in the left hand pane, then enter a value for **arg0**, and select **Invoke**.
10. Scroll down to see the test results.

Using the Basic MDB Maven Archetype

To build a Jakarta EE project using the basic MDB Maven archetype, you create the basic project, then customize, compile, and package it. Then you deploy it and test it.

To use the Basic MDB project using the Maven Archetype:

1. Create a new Basic MDB project using the Maven archetype, by running a command similar to the following:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.oracle.weblogic.archetype \
    -DarchetypeArtifactId=basic-mdb \
    -DarchetypeVersion=14.1.2-0-0 \
    -DgroupId=org.mycompany \
    -DartifactId=my-basic-mdb-project \
    -Dversion=1.0-SNAPSHOT
```

This command runs Maven's archetype plug-in's generate goal, which enables you to create a new project from an archetype. Table 3-8 describes the parameters.

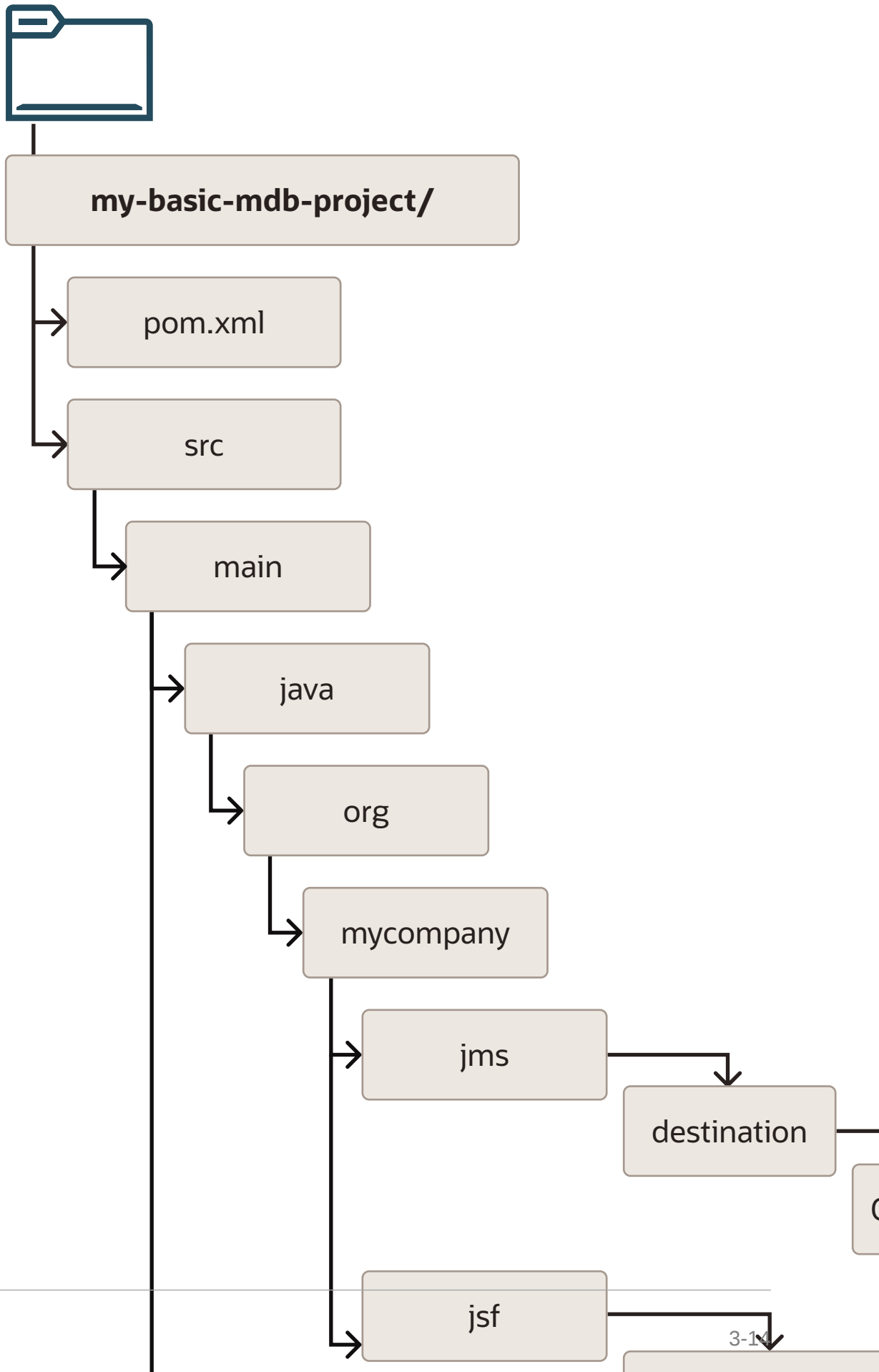
Table 3-8 Parameters for the Basic MDB Project

Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new project. This must be `com.oracle.weblogic.archetype`, as shown in the preceding example.
`archetypeArtifactId`	The artifact ID of the archetype that you want to use to create the new project. This must be `basic-mdb`, as shown in the preceding example.
`archetypeVersion`	The version of the archetype that you want to use to create the new project. This must be `14.1.2-0-0`, as shown in the preceding example.
`groupId`	The group ID for your new project. This usually starts with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new project. This is usually an identifier for this project.
`version`	The version number for your new project. This is usually `1.0-SNAPSHOT` for a new project.

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

After creating your project, it contains the following files:


my-basic-mdb-project/
pom.xml
src
main
java
org
mycompany
jms
destination
jsf

These files make up a small sample application, which you can deploy as is. You can use this application as a starting point for building your own application.

Table 3-9 describes the files included in the project.

Table 3-9 Files Created for the Basic MDB Project

File	Purpose
`pom.xml`	The Maven Project Object Model (POM) file that describes your new project. It includes the Maven coordinates that you specified for your project. It also includes the appropriate plug-in definitions to use the WebLogic Maven plug-in to build your project.
Files under `src/main/java`	An example Message Driven Bean that is used by the web application to store data.
Files under `src/main/webapp`	HTML files that make up the web application user interface.
`configure_resources.py`	WLST online script used by the build to create the JMS resources required to support the MDB.

2. After you have written your project code, you can use Maven to build the project. It is also possible to build the sample as is.
3. Customize the POM to suit your environment. See Customizing the Project Object Model File to Suit Your Environment.
4. Compile your Basic MDB Project. See Compiling Your Jakarta EE Project.
5. Package your Basic MDB Project. See Packaging Your Jakarta EE Project.
6. Deploy your Basic MDB Project. For information about deploying it using Maven, see Deploying Your Jakarta EE Project to WebLogic Server Using Maven. For information about deploying it using other options, see Deploying Your Jakarta EE Project to WebLogic Server Using Different Options.
7. Test your Basic MDB Project.

   You can test the Basic MDB by visiting the following URL on the WebLogic Server instance where you deployed it:

   `http://servername:7001/basicMDB/index.xhtml`
8. Provide the **Account Name** and **Amount**, then select **Deposit**:
9. As indicated in the user interface, you must check the WebLogic Server output to find the message printed by the MDB. It looks like the following example:

   ```
   The money has been deposited to frank, the balance of the account is 500.0
   ```

4

Building Oracle Coherence Projects with Maven

You can use the Oracle Coherence archetypes to create, build, and deploy Oracle Coherence applications.

- Introduction to Building Oracle Coherence Projects with Maven
  Oracle Fusion Middleware provides a Maven plug-in and an archetype is provided for Oracle Coherence Grid Archive (GAR) projects.
- Creating a Coherence Project from a Maven Archetype
  You can create a new Coherence project using the Coherence Maven archetype.
- Building Your Coherence Project with Maven
  After you have written your project code, you can use Maven to build the project.
- Deploying Your Coherence Project to the WebLogic Server Coherence Container with Maven
  To deploy your GAR to a Coherence Container in a WebLogic Server environment, you must add some additional configuration to your project's POM file.
- Building a More Complete Coherence Example
  In a real application, you are likely to have not just a GAR project, but also some kind of client project that interacts with the Coherence cache established by the GAR.

Introduction to Building Oracle Coherence Projects with Maven

Oracle Fusion Middleware provides a Maven plug-in and an archetype is provided for Oracle Coherence Grid Archive (GAR) projects.

Table 4-1 describes the Maven coordinates.

Table 4-1 Maven Coordinates with Coherence

Name	GroupId	ArtifactId	Version
GAR plug-in	`com.oracle.coherence`	`gar-maven-plugin`	`14.1.2-0-0`
GAR archetype	`com.oracle.coherence.archetype`	`gar-maven-archetype`	`14.1.2-0-0`

Table 4-2 describes the goals supported by the Oracle Coherence plug-in.

Table 4-2 Oracle Coherence Goals

Goal	Purpose
generate-descriptor	Generates the project's POF configuration file.
package	Packages the basic GAR assets, including library dependencies into a JAR archive.

Table 4-2 (Cont.) Oracle Coherence Goals

Goal	Purpose
repackage	Repackages the packaged JAR archive with optional metadata and GAR extension.

Creating a Coherence Project from a Maven Archetype

You can create a new Coherence project using the Coherence Maven archetype.

1. To create a new Coherence project using the Coherence Maven archetype, issue a command similar to the following:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.oracle.coherence.archetype \
    -DarchetypeArtifactId=gar-maven-archetype \
    -DarchetypeVersion=14.1.2-0-0 \
    -DgroupId=org.mycompany \
    -DartifactId=my-gar-project \
    -Dversion=1.0-SNAPSHOT
```

This command runs Maven's archetype plug-in's generate goal which lets you create a new project from an archetype. Table 4-3 describes the parameters.

Table 4-3 Parameters for the Coherence Projects

Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new project. This must be `com.oracle.coherence.archetype`.
`archetypeArtifactId`	The artifact ID of the archetype that you want to use to create the new project. This must be `gar-maven-archetype`.
`archetypeVersion`	The version of the archetype that you want to use to create the new project. This must be `14.1.2-0-0`.
`groupId`	The group ID for your new project. This usually starts with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new project. This is usually an identifier for this project.
`version`	The version for your new project. This is usually `1.0-SNAPSHOT` for a new project.

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

After creating your project, it contains the following files:

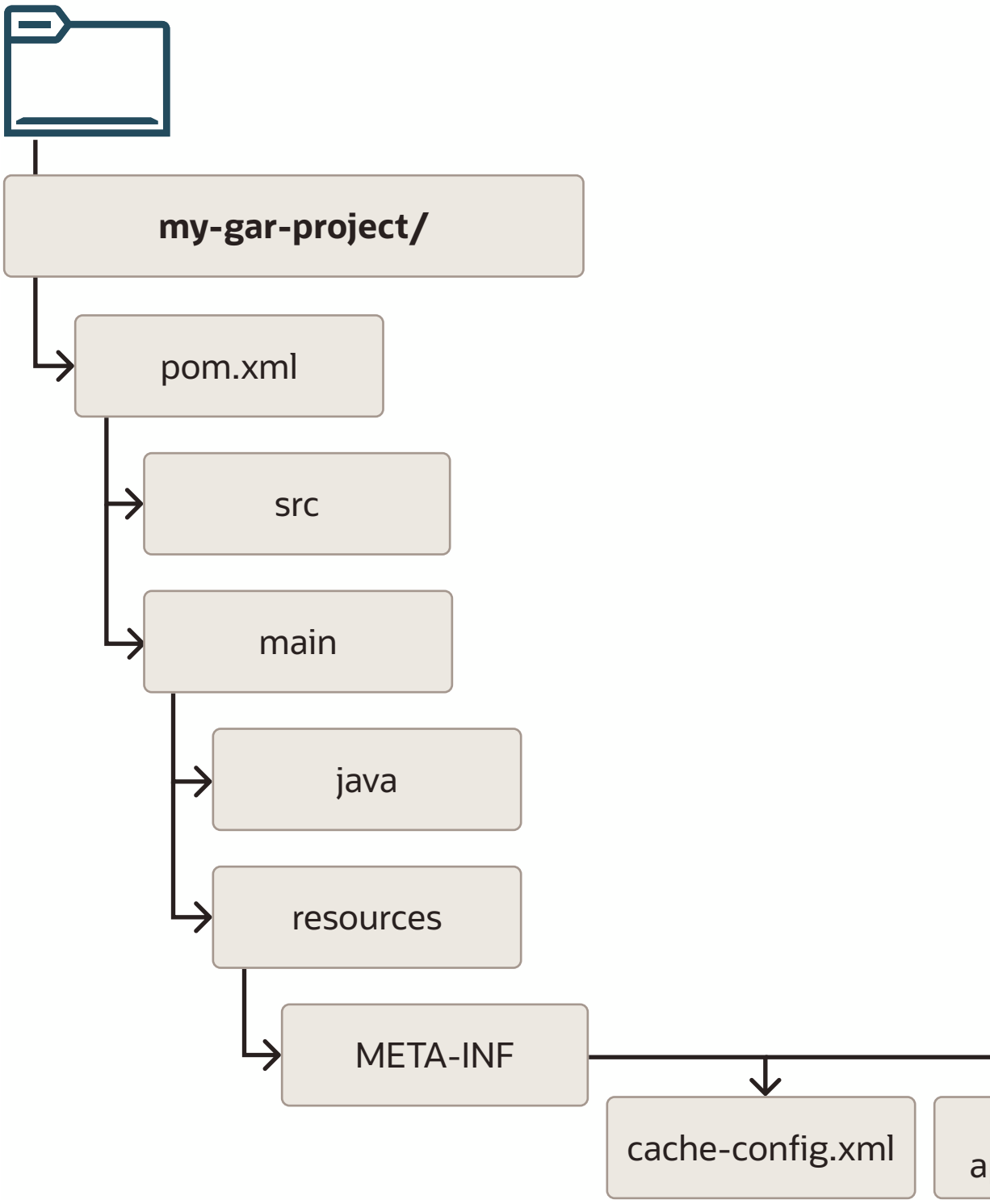


Table 4-4 describes the files included in the project.

Table 4-4 Files Created for the Coherence Project

File	Purpose
`pom.xml`	The Maven Project Object Model (POM) file that describes your new project. It includes the Maven coordinates that you specified for your project and the appropriate plug-in definitions to use the Coherence Maven plug-in to build your project into a gar file.
`cache-config.xml`	A starter Coherence cache configuration file.
`coherence-application.xml`	A starter Coherence GAR deployment descriptor for your GAR file.
`pof-config.xml`	A starter Coherence Portable Object Format (POF) configuration file. The POF configuration file is processed and inserted into the final GAR file if the plug-in option `generatePof` is set to `true`. By default, POF configuration metadata will not be generated.

2. If you are using POF in your project, then you must add the following parameter into your project's POM file:

Parameter	Purpose
`generatePof`	The POF configuration file is generated and inserted into the final GAR file if this plug-in option is `true`. The configuration file is generated by scanning all classes in the GAR's classpath annotated with the class `com.tangosol.io.pof.annotation.Portable`. By default, POF configuration metadata is not generated.

3. To generate a GAR with a correctly generated `pof-config.xml` file, add the following to your GAR plug-in configuration in the POM:

```
<build>
  <plugins>
…
    <plugin>
      <groupId>com.oracle.coherence</groupId>
      <artifactId>gar-maven-plugin</artifactId>
      <version>14.1.2-0-0</version>
      <extensions>true</extensions>
      <configuration>
         <generatePof>true</generatePof>
      </configuration>
    </plugin>
…
  </plugins>
</build>
```

Building Your Coherence Project with Maven

After you have written your project code, you can use Maven to build the project.

1. To compile the source code in your project, run the following command:

```
mvn compile
```

2. To package the compiled source into a GAR, run the following command. Note that this command runs all steps up to package, including the compile.

```
mvn package
```

Deploying Your Coherence Project to the WebLogic Server Coherence Container with Maven

To deploy your GAR to a Coherence Container in a WebLogic Server environment, you must add some additional configuration to your project's POM file.

Take these steps:

1. Add instructions to use the Oracle WebLogic Maven plug-in to deploy the GAR, as shown in the following example:

```
<plugin>
  <groupId>com.oracle.weblogic</groupId>
  <artifactId>weblogic-maven-plugin</artifactId>
  <version>14.1.2-0-0</version>
  <executions>
    <execution>
      <id>deploy-to-server</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>deploy</goal>
      </goals>
      <configuration>
        <adminurl>t3://localhost:7001</adminurl>
        <user>weblogic</user>
        <password>password</password>
        <!--
          The location of the file or directory to be deployed
         -->
          <source>${project.build.directory}/${project.build.finalName}.$
{project.packaging}</source>
        <!--
          The target servers where the application is deployed
         -->
        <targets>AdminServer</targets>
        <verbose>true</verbose>
        <name>${project.build.finalName}</name>
      </configuration>
    </execution>
  </executions>
</plugin>
```

2. After you have added this section to your POM, use the following command to compile, package, and deploy your GAR to WebLogic Server:

```
mvn verify
```

Building a More Complete Coherence Example

In a real application, you are likely to have not just a GAR project, but also some kind of client project that interacts with the Coherence cache established by the GAR.

Refer to Building a Real Application with Maven to see an example that includes a Coherence GAR and a web application (WAR) that interacts with it.

5

Building ADF Projects with Maven

You can use the Oracle Application Development Framework (ADF) Maven archetypes to create, build, and deploy Oracle ADF applications.
For more information about using the Oracle ADF development plug-in with Maven, see "Building and Running with Apache Maven " in *Developing Applications with Oracle JDeveloper*.

- Introduction to Building Oracle ADF Projects with Maven
  Oracle Fusion Middleware provides two Maven plug-ins and an archetype for Oracle ADF projects.
- Creating an ADF Application Using the Maven Archetype
  You can create a new Oracle ADF application using the Oracle ADF application Maven archetype.
- Building Your Oracle ADF Project with Maven
  After you have written your code, you can use Maven to build the project.

Introduction to Building Oracle ADF Projects with Maven

Oracle Fusion Middleware provides two Maven plug-ins and an archetype for Oracle ADF projects.

Table 5-1 describes the Maven coordinates.

Table 5-1 Maven Coordinates with Oracle ADF

Name	GroupId	ArtifactId	Version
ADF ojmake plug-in	`com.oracle.adf.plugin`	`ojmake`	`14.1.2-0-0`
ADF ojdeploy plug-in	`com.oracle.adf.plugin`	`ojdeploy`	`14.1.2-0-0`
ADF archetype	`com.oracle.adf.archetype`	`oracle-adffaces-ejb`	`14.1.2-0-0`

JDeveloper also has extensive support for Maven. This documentation covers Maven use outside of JDeveloper. For more details about using Maven within JDeveloper, see "Building and Running with Apache Maven" in *Developing Applications with Oracle JDeveloper*.

Creating an ADF Application Using the Maven Archetype

You can create a new Oracle ADF application using the Oracle ADF application Maven archetype.

To do so, issue a command similar to the following:

```
mvn archetype:generate \
-DarchetypeGroupId=com.oracle.adf.archetype \
 -DarchetypeArtifactId=oracle-adffaces-ejb \
 -DarchetypeVersion=14.1.2-0-0 \
 -DgroupId=org.mycompany \
```

```
-DartifactId=my-adf-application \
-Dversion=1.0-SNAPSHOT
```

This command runs Maven's `archetype:generate` goal which allows you to create a new project from an archetype. Table 5-2 describes the parameters.

Table 5-2 Parameters for the Oracle ADF Project

Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new project. This must be `com.oracle.adf.archetype`, as shown in the example.
`archetypeArtifactId`	The artifact ID of the archetype that you want to use to create the new project. This must be `oracle-adffaces-ejb`, as shown in the example.
`archetypeVersion`	The version of the archetype that you want to use to create the new project. This must be `14.1.2-0-0`, as shown in the example.
`groupId`	The group ID for your new project. This usually starts with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new project. This is usually an identifier for this project.
`version`	The version for your new project. This is usually `1.0-SNAPSHOT` for a new project.

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

Building Your Oracle ADF Project with Maven

After you have written your code, you can use Maven to build the project.

1. To compile your project, run the following command:

   ```
   mvn compile
   ```

   This command runs the ojmake plug-in.

2. To package the project into an EAR file, run the following command (note that this actually runs all steps up to package, including the compile again):

   ```
   mvn package
   ```

   This command runs the ojdeploy plug-in.

6

Building Oracle SOA Suite and Oracle Business Process Management Projects with Maven

You can use the Oracle SOA Suite and Oracle Business Process Management Maven archetypes to create, build, and deploy Oracle SOA Suite and Oracle Business Process Management applications.
For more information about using the Oracle SOA Suite development plug-in with Maven, see Using the Oracle SOA Suite Development Maven Plug-In in *Developing SOA Applications with Oracle SOA Suite*.

Introduction to Building Oracle SOA Suite and Oracle Business Process Management Projects with Maven

Oracle Fusion Middleware provides a Maven plug-in and two archetypes are provided for Oracle SOA Suite and Oracle Business Process Management.

Table 6-1 describes the Maven coordinates.

Table 6-1 Maven Coordinates with Oracle SOA Suite

Name	GroupId	ArtifactId	Version
SOA plug-in	`com.oracle.soa.plugin`	`oracle-soa-plugin`	`14.1.2-0-0`
SOA Application archetype	`com.oracle.soa.archetype`	`oracle-soa-application`	`14.1.2-0-0`
SOA Project archetype	`com.oracle.soa.archetype`	`oracle-soa-project`	`14.1.2-0-0`

Table 6-2 describes the goals supported by the Oracle SOA Suite plug-in.

Table 6-2 Oracle SOA Suite Plug-In Goals

Goal	Purpose
compile	Runs the SCA composite validation routine on your project---this is somewhat equivalent to a traditional compile operation in that it inspects the source artifacts and produces errors and warnings. However, it does not produce any compiled version of the source artifacts.
sar	Creates a SOA archive (SAR) file from the project.
deploy	Deploys the SAR file to a runtime environment. Note that this goal is mapped to the pre-integration-test phase in the default life cycle, not the deploy phase, as deployment to a runtime environment is normally done in the pre-integration-test phase in Maven.
test	Runs SCA tests in the composite. Note that this goal is mapped to the integration-test phase, not the test phase, as it depends on the composite (SAR) having been deployed to a runtime environment.
undeploy	Undeploys a composite (SAR) from a runtime environment. Note that this goal is not mapped to any phase in the default Maven life cycle.

The SOA Application archetype allows you to create a new SOA application with a single SOA Project in it. This can be imported in JDeveloper for editing.

The SOA Project archetype allows you to add a new SOA Project to an existing SOA Application.

Creating a New SOA Application and Project from a Maven Archetype

You can create a new SOA application (containing a single SOA project) using the SOA Maven archetype.

To do so, run a command similar to the following:

```
mvn archetype:generate \
     -DarchetypeGroupId=com.oracle.soa.archetype \
     -DarchetypeArtifactId=oracle-soa-application \
     -DarchetypeVersion=14.1.2-0-0 \
     -DgroupId=org.mycompany \
     -DartifactId=my-soa-app \
     -Dversion=1.0-SNAPSHOT \
     -DprojectName=my-project
```

This command runs Maven's archetype plug-in's generate goal which allows you to create a new SOA Application from an archetype. Table 6-3 describes the parameters.

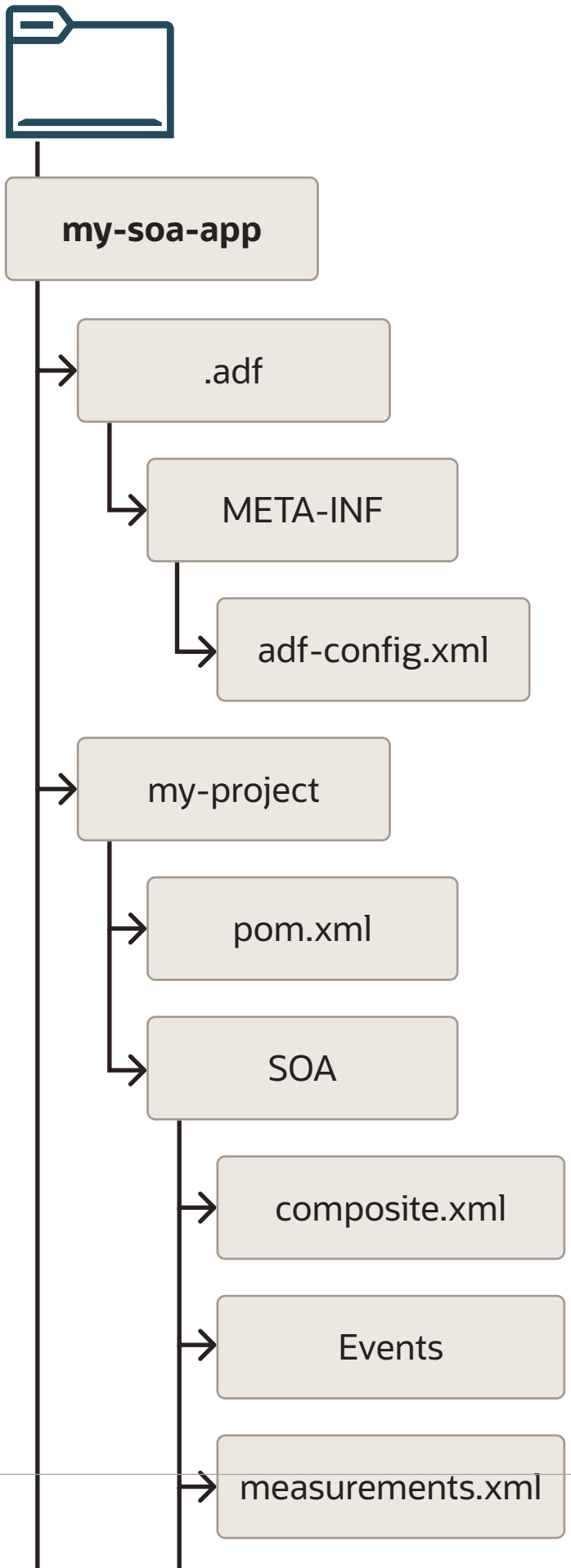
Table 6-3 Parameters for the Oracle SOA Suite Application

Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new SOA application. This must be `com.oracle.soa.archetype`, as shown in the previous example.
`archetypeArtifactId`	The artifact ID of the archetype that you want to use to create the new SOA application. This must be `oracle-soa-application`, as shown in the previous example.
`archetypeVersion`	The version of the archetype that you want to use to create the new SOA application. This must be `14.1.2-0-0`, as shown in the previous example.
`groupId`	The group ID for your new SOA application. This would normally start with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new SOA application. This would normally be an identifier for this SOA application.
`version`	The version for your new SOA application. This would normally be `1.0-SNAPSHOT` for a new project.
`projectName`	The name for the SOA project inside your new SOA application. This should be different to the name of the SOA application (`artifactId`).

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

After creating your application, it contains the following files, assuming you named your application `my-soa-app` and your project `my-project`, as shown in the previous example:


my-soa-app
.adf
META-INF
adf-config.xml
my-project
pom.xml
SOA
composite.xml
Events
measurements.xml

The generated project contains files and a handful of empty directories. The files are described in Table 6-4.

Table 6-4 Files Created for the Oracle SOA Suite Application and Project

File	Purpose
`pom.xml`	The Maven Project Object Model (POM) file that describes your new application. It includes the Maven coordinates that you specified for your application, and a reference to the SOA project inside the application.
`PROJECT``/pom.xml`	The Maven POM file that describes your new project. It includes the Maven coordinates that you specified for your project, and the appropriate plug-in definitions to use the SOA Maven Plug-In to build your project into a SAR file.
`PROJECT``/composite.xml`	Composite metadata.
`.adf/META-INF/adf-config.xml`	The definitions for MDS repositories that may be needed to build your composites.
Others	The remainder are the standard files that are created in any new composite. These are the same files as you would find in a new SOA Application and SOA Project created in JDeveloper.

Creating a SOA Project in an Existing SOA Application from a Maven Archetype

You can create a new SOA project (in an existing SOA application) using the SOA Maven archetype.

To do so, run a command similar to the following, while in the SOA application directory:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.oracle.soa.archetype \
    -DarchetypeArtifactId=oracle-soa-project \
    -DarchetypeVersion=14.1.2-0-0 \
    -DgroupId=org.mycompany \
    -DartifactId=my-second-project \
    -Dversion=1.0-SNAPSHOT
```

This command runs Maven's archetype plug-in's generate goal which allows you to create a new SOA Project from an archetype. Table 6-5 describes the parameters:

Table 6-5 Parameters for the Oracle SOA Suite Project

Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new SOA application. This must be `com.oracle.soa.archetype`, as shown in the previous example.
`archetypeArtifactId`	The artifact ID of the archetype that you want to use to create the new SOA application. This must be `oracle-soa-project`, as shown in the previous example.
`archetypeVersion`	The version of the archetype that you want to use to create the new SOA application. This must be `14.1.2-0-0`, as shown in the previous example.

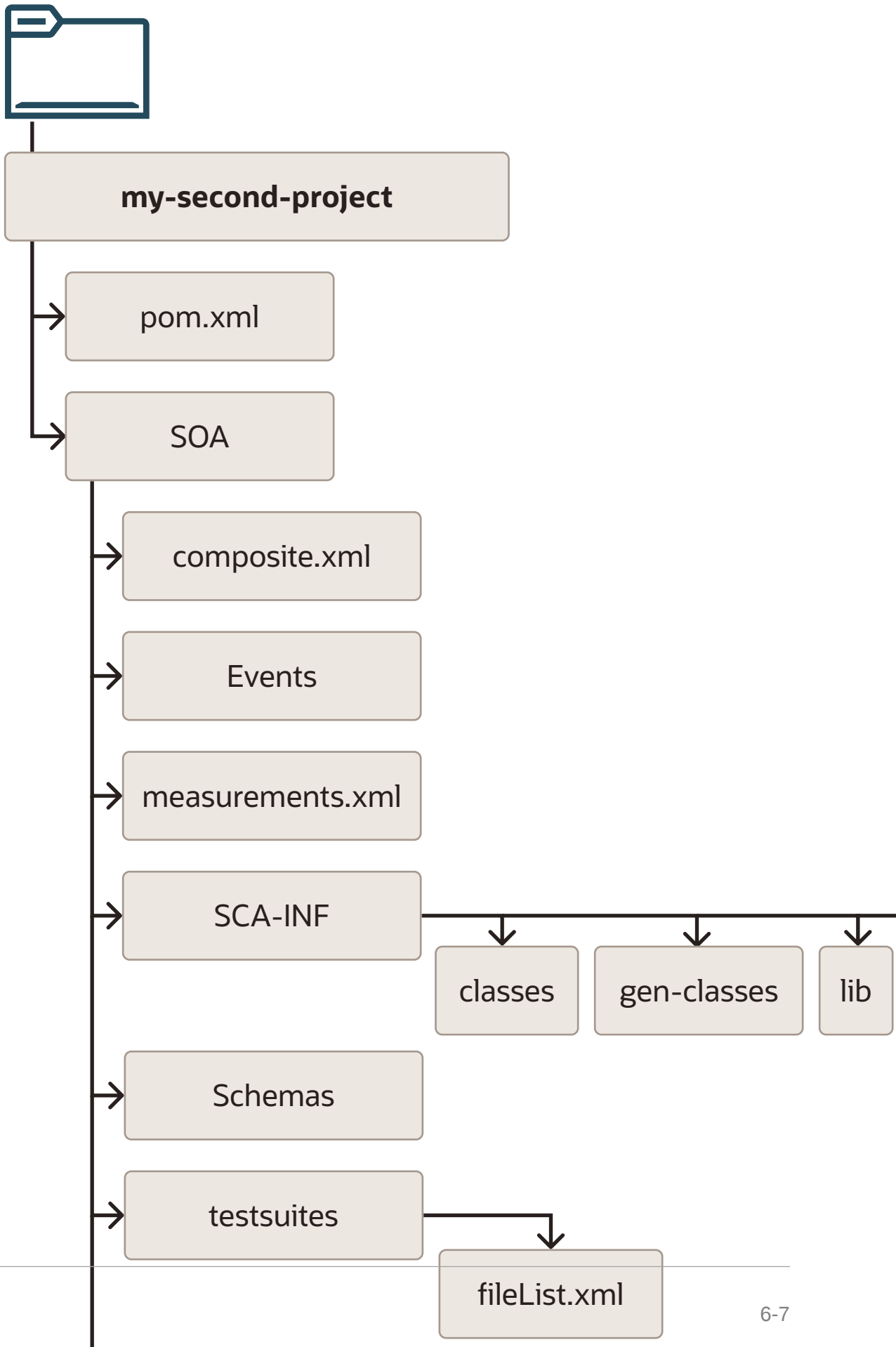
Table 6-5 (Cont.) Parameters for the Oracle SOA Suite Project

Parameter	Purpose
`groupId`	The group ID for your new SOA project. This would normally start with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new SOA project. This would normally be an identifier for this SOA project.
`version`	The version for your new SOA project. This would normally be `1.0-SNAPSHOT` for a new project.

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

After creating your new project, it contains the following files, assuming that you named your project `my-second-project`:


my-second-project
pom.xml
SOA
composite.xml
Events
measurements.xml
SCA-INF
classes
gen-classes
lib
Schemas
testsuites
fileList.xml

The generated project contains files and a handful of empty directories. The files are described in Table 6-4.

The `mvn generate` command also updates your SOA Application POM to add the new project. For example, if you created this project in the application in Creating a New SOA Application and Project from a Maven Archetype, you would see the following list in the SOA Application POM:

```
<modules>
  <module>my-project</module>
  <module>my-second-project</module>
</modules>
```

When you have a SOA Application with multiple SOA Projects like this (a Maven multi-module project), Maven builds your projects one by one, in the order they are listed in the SOA Application POM.

Editing Your SOA Application in Oracle JDeveloper

You can edit your SOA application in Oracle JDeveloper to configure SOA composites.

To edit your application, you open the project in Oracle JDeveloper, then edit the application:

1. Open the **File** menu, then select **Import…**.
2. In the **Import** dialog box, select the **Maven Project** option and click **OK.**

   The **Import Maven Projects** dialog appears.
3. In the **Root Directory** field, enter the path to the application you want to import into JDeveloper.
4. In the **Settings File** field, enter the path to your Maven `settings.xml` file. The default value is most likely correct unless you are using a non-standard location for your Maven settings file.
5. Click **Refresh** to load a list of projects available at that location.
6. Select the projects that you want to import. Also, select **Also import source files into application** and **Update existing JDeveloper Projects to synch with imported POM files**.
7. Click **OK** to complete the import.

Your projects are then opened in JDeveloper.

Building Your SOA Project with Maven

After you have written your project code, you can use Maven to build the project.

To do so:

1. To run the SCA validation on your project, run this command:

   ```
   mvn compile
   ```
2. To build the SAR file, run this command:

   ```
   mvn package
   ```

What You May Need to Know About Building SOA Projects

Some SOA composite projects require access to an MDS repository in order to be built. This includes all composites that contain a Human Task or Business Rule component. These components refer to WSDL or XSD files, or both, in MDS.

To build these projects, you need to provide the build with access to an MDS repository. This can be either a file-based or a database-based MDS repository.

The MDS repository connection details are specified in the *SOA Application*`/.adf/META-INF/adf-config.xml` file. This means that any SOA Project which requires access to MDS, must be located inside a SOA Application.

If you create a new project using the SOA Maven Application archetype or using JDeveloper, the `adf-config.xml` file will contain the following default MDS repository configuration:

```
<metadata-store-usages>
  <metadata-store-usage id="mstore-usage_1">
    <metadata-store class-name="oracle.mds.persistence.stores.file.FileMetadataStore">
      <property name="metadata-path" value="${oracleHome}/integration"/>
      <property name="partition-name" value="seed"/>
    </metadata-store>
  </metadata-store-usage>
</metadata-store-usages>
```

This example defines a file-based MDS repository in the location `${oracleHome}/integration`. If you run this build in Maven, the `oracleHome` variable may not be defined. In that case, you need to specify it on the Maven command line, as shown in the following example:

```
mvn compile -DoracleHome=MW_HOME/soa -DappHome=dir_for_application_for_proj
```

Notice that the value of `oracleHome` points to the `soa` directory in the Oracle Home in which you installed the SOA Quickstart or JDeveloper. That directory contains the *seed* MDS repository.

Alternatively, you can just update the `adf-config.xml` file to provide the full path to the MDS repository that you want to use.

If you want to use a database-based MDS repository, you must alter the configuration to specify the JDBC values, similar to that shown in the following example:

```
<metadata-store-usage id="mstore-usage_1">
  <metadata-store class-name="oracle.mds.persistence.stores.db.DBMetadataStore">
     <property name="jdbc-userid" value="your_prefix_mds"/>
     <property name="jdbc-password" value="<password>"/>
     <property name="jdbc-url"
             value="jdbc:oracle:thin://@database.server:1521/service_name"/>
     <property name="partition-name" value="soa-infra"/>
  </metadata-store>
</metadata-store-usage>
```

Deploying Your SOA Project to the SOA Server with Maven

To deploy your SOA project to the SOA server with Maven, you deploy the SAR file and edit the project POM.

To deploy the SAR file, run the following command:

```
mvn pre-integration-test
```

Table 6-6 describes the parameters that you can specify for the deployment. These may be specified either in the POM file for the project or on the command line.

Table 6-6 Parameters for Deploying a SOA Project

Parameter	Purpose
`serverURL`	The URL of the Administration Server in the SOA domain.
`sarLocation`	The location of the SAR file.
`overwrite`	Whether deployment should overwrite any existing composite with the same revision.
`configplan`	(Optional) The name of the SOA configuration plan to use, if any.
`forceDefault`	Whether deployment should make this revision the default revision.
`regenerateRuleBase`	Whether the base rule dictionary should be regenerated.
`composite.partition`	The SOA partition that the composite will be deployed into.
`user`	User name to be used for deployment.
`password`	Password to be used for deployment.

To specify the parameters:

- On the command line: Use the format `-Dparameter=value`, as shown in this example (note that the whole command would be entered on one line):

```
mvn pre-integration-test -DserverURL=http://test.server:7001
                         -DsarLocation=deploy/sca_my-project_rev1.0.sar
                         -Doverwrite=true
                         -DforceDefault=true
                         -Dcomposite.partition=test
                         -Duser=weblogic
                         -Dpassword=<password>
```

- In your project POM file: Specify replacement values for the defaults already specified in the parameters section of the project POM:

```
<properties>
        <!-- these parameters are used by the compile goal -->
        <scac.input.dir>${project.basedir}/SOA/</scac.input.dir>
        <scac.output.dir>${project.basedir}/target</scac.output.dir>
        <scac.input>${scac.input.dir}/composite.xml</scac.input>
        <scac.output>${scac.output.dir}/out.xml</scac.output>
        <scac.error>${scac.output.dir}/error.txt</scac.error>
        <scac.displayLevel>1</scac.displayLevel>
        <!-- if you are using a config plan, uncomment the following line and update
to point
             to your config plan -->
        <!--<configplan>${scac.input.dir}/configplan.xml</configplan>-->
        <!-- these parameters are used by the deploy and undeploy goals -->
        <composite.name>${project.artifactId}</composite.name>
        <composite.revision>1.0</composite.revision>
        <composite.partition>default</composite.partition>
        <serverUrl>${oracleServerUrl}</serverUrl>
        <user>${oracleUsername}</user>
        <password>${oraclePassword}</password>
        <overwrite>true</overwrite>
        <forceDefault>true</forceDefault>
        <regenerateRulebase>false</regenerateRulebase>
        <keepInstancesOnRedeploy>false</keepInstancesOnRedeploy>
```

```
        <!-- these parameters are used by the test goal -->
        <!-- if you are using the sca-test (test) goal, you need to uncomment
             the following line and point it to your jndi.properties file. -->
       <!--<jndi.properties.input>${basedir}/jndi.properties</
jndi.properties.input>-->
        <scatest.result>${scac.output.dir}/testResult</scatest.result>
        <!--  input is the name of the composite to run test suites against -->
        <input>project12</input>
</properties>
```

Running SCA Test Suites with Maven

You can run SCA test suites with Maven.

To run your SCA Test Suites as part of the Maven build process:

1. Create a `jndi.properties` file (as you would if you were executing SCA Test Suites from ANT, for example) in your SOA composite project directory. This file contains the following information:

   ```
   java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
   java.naming.provider.url=t3://test.server:7003/soa-infra
   java.naming.security.principal=weblogic
   java.naming.security.credentials=<password>
   dedicated.connection=true
   dedicated.rmicontext=true
   ```

2. Uncomment the `jndi.properties` entry in the SOA composite project POM (`pom.xml`) and ensure that it points to the file you just created.
3. The SOA Maven Plug-In runs the SCA Tests in the integration-test phase. To compile and package your composite, deploy it to a server and run the SCA Tests, run this command:

   ```
   mvn verify
   ```

What You May Need to Know About Deploying SOA Composites

When you create a SOA composite, you may use new resources, such as WebLogic data sources, JMS queues, and topics. These resources may not be present in the runtime environment where you want to deploy your composite. This means that you may not be able to successfully run any instances of your composite, for example to run test cases.
While it is possible to manually create these resources through the WebLogic console, this would not be appropriate for an automated build environment. To address this issue, you can create WLST scripts and run them as part of the build to ensure that any necessary resources are created and configured on the runtime environment. You can run the WLST scripts at the appropriate time in your build using the `weblogic-maven-plugin:wlst` goal.

The following is an example of a WLST script to create a data source. You could add it to your project as `misc/create-datasource.py`:

```
# Copyright 2012, 2024 Oracle Corporation.
# All Rights Reserved.
#
# Provided on an 'as is' basis, without warranties or conditions
# of any kind, either express or implied, including, without
# limitation, any warranties or conditions of title,
# non-infringement, merchantability, or fitness for a particular
# purpose. You are solely responsible for determining the
# appropriateness of using and assume any risks. You may not
# redistribute.
```

```
#
# This WLST script can be used as part of a continuous
# integration build process before deploying a SCA composite, to
# create any necessary Jakarta EE data sources on the WebLogic
# Server.
#
# In addition to creating the data source, this script will also
# update the resource adapter and redeploy it.
#
import time

#
# Edit these parameters before running this script.
#

# admin server url
url = 't3://localhost:7001'

# username to connect to the admin server
username = 'weblogic'

# password to connect to the admin server
password = 'password'

# the name for the EIS, as defined in the JDev DB Adapter wizard
eisName = 'eis/db/myDS'

# the server to target where the DbAdapter is deployed
serverName = 'soa_server1'

# the name for the data source
dsName = 'myDS'
# the JNDI name for the data source
jndiName = 'jbdc/myDS'

# the database url for the data source
dbUrl = 'jdbc:oracle:thin:@localhost:1521:orcl'

# the database user
dbUser = 'mark'

# the database password
dbPassword = 'password'

# the database driver to use
dbDriver = 'oracle.jdbc.xa.client.OracleXADataSource'

# the host where node manager is running
nmHost = 'localhost'

# the port to connect to node manager (5556 is the default port)
nmPort = '5556'

# the user to connect to node manager
nmUser = 'weblogic'

# the password to connection to node manager
nmPassword = 'password'

# the name of the weblogic domain
domain = 'soa_domain'
```

```
#
# Do not edit below this line.
#

uniqueString = ''
appName = 'DbAdapter'
moduleOverrideName = appName+'.rar'
moduleDescriptorName = 'META-INF/weblogic-ra.xml'

#
# method definitions
#
def makeDeploymentPlanVariable(wlstPlan, name, value, xpath,
                               origin='planbased'):
"""
Create a variable in the Plan.

This method is used to create the variables that are needed in the Plan in order to add
an entry for the outbound connection pool for the new data source.
"""
try:
  variableAssignment = wlstPlan.createVariableAssignment(name,
moduleOverrideName,moduleDescriptorName)
  variableAssignment.setXpath(xpath)
  variableAssignment.setOrigin(origin)
  wlstPlan.createVariable(name, value)
except:
  print('--> was not able to create deployment plan variables successfully')

def main():
  '''
  Copyright 2012, 2024 Oracle Corporation. All Rights Reserved.

  Provided on an 'as is' basis, without warranties or conditions
  of any kind, either express or implied, including, without
  limitation, any warranties or conditions of title,
  non-infringement, merchantability, or fitness for a particular
  purpose. You are solely responsible for determining the
  appropriateness of using and assume any risks. You may not
  redistribute.

  This WLST script can be used as part of a continuous
  integration build process before deploying a SCA composite, to
  create any necessary Jakarta EE data sources on the WebLogic
  Server.

  In addition to creating the data source, this script will also
  update the resource adapter and redeploy it.
  '''

  print('!!! WARNING !!! This script will make changes to your WebLogic domain. Make
sure you know what you are doing. There is no support for this script. If something bad
happens, you are on your own! You have been warned.')

  #
  # generate a unique string to use in the names
  #
  uniqueString = str(int(time.time()))

  #
  # Create a JDBC Data Source.
  #
```

```
  try:
    print('--> about to connect to weblogic')
    connect(username, password, url)

    print('--> about to create a data source ' + dsName)
    edit()
    startEdit()
    create(dsName, 'JDBCSystemResource')
    cd('/JDBCSystemResources/%s/JDBCResource/%s' % (dsName, dsName)
    set('Name', dsName)
    cd('JDBCDataSourceParams/%s' % dsName)
    set('JNDINames', jarray.array([String(jndiName)], String))
    cd('../JDBCDriverParams/%s' % dsName)
    set('URL', dbUrl)
    set('DriverName', dbDriver)
    set('PasswordEncrypted', dbPassword)
    cd('../JDBCConnectionPoolParams/%s' % dsName)
    set('TestTableName', 'DUAL')
    cd('Properties/%s' % dsName)
    create('user', 'Property')
    cd('Properties/user')
    set('Value', dbUser)
    cd('/JDBCSystemResources/%s/JDBCResource/%s/ JDBCDataSourceParams/%s' % (dsName,
dsName, dsName))
    set('GlobalTransactionsProtocol', 'TwoPhaseCommit')
    cd('/JDBCSystemResources/%s' % dsName)
    obj_name = ObjectName('com.bea:Name=%s,Type=Server' % serverName)
    set('Targets', jarray.array([obj_name], ObjectName))
    save()
    print('--> activating changes')
    activate()
    print('--> done')

    #
    # update the deployment plan
    #
    cd('/AppDeployments/DbAdapter')
    print('--> about to update the deployment plan for the DbAdapter')
    startEdit()
    planPath = get('PlanPath')
    appPath = get('SourcePath')
    print('--> Using plan ' + planPath)
    plan = loadApplication(appPath, planPath)
    print('--> adding variables to plan')
    makeDeploymentPlanVariable(plan, 'ConnectionInstance_eis/DB/' + dsName +
'_JNDIName_' + uniqueString, eisName, '/weblogic-connector/outbound-resource-adapter/
connection-definition-group/[connection-
factoryinterface="javax.resource.cci.ConnectionFactory"]/connection-instance/[jndi-
name="' + eisName + '"]/jndi-name')

    makeDeploymentPlanVariable(plan, 'ConfigProperty_xADataSourceName_Value_' +
uniqueString, eisName, '/weblogic-connector/outbound-resource-adapter/
connectiondefinition-group/[connection-factory-
interface="javax.resource.cci.ConnectionFactory"]/
connection-instance/[jndi-name="' + eisName + '"]/connection-properties/properties/
property/[name="xADataSourceName"]/value')
    print('--> saving plan')
    plan.save()
    save()
    print('--> activating changes')
    activate(block='true')
```

```
    cd('/AppDeployments/DbAdapter/Targets')
    print('--> redeploying the DbAdapter')
    redeploy(appName, planPath, targets=cmo.getTargets())
    print('--> done')
  except:
    print('--> something went wrong, bailing out')
    stopEdit('y')
    exit()

  #
  # disconnect from the admin server
  #
  print('--> disconnecting from admin server now')
  disconnect()

#
# this is the main entry point
#
if __name__ == '__main__' or __name__ == 'main':
  main()
```

To run this script during the pre-integration-test phase of your build, you would include a `plugin` section similar to the following in your SOA Project POM:

```
<plugin>
  <groupId>com.oracle.weblogic</groupId>
  <artifactId>weblogic-maven-plugin</artifactId>
  <version>14.1.2-0-0</version>
  <executions>
    <execution>
      <id>wlst-create-datasource</id>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>wlst</goal>
      </goals>
      <configuration>
        <middlewareHome>/opt/soa1412</middlewareHome>
        <fileName>${project.basedir}/misc/create-datasource.py</fileName>
      </configuration>
    </execution>
  </executions>
</plugin>
```

What You May Need to Know About ADF Human Task User Interface Projects

If you add an ADF Human Task User Interface project to your SOA Application in JDeveloper, the application level POM is updated to add the new ADF project as a module and some `<plugin>` definitions are added to the `<build>` section to build the ADF project.

When you create an ADF Human Task project, some JDeveloper libraries are added to the build path for that project in JDeveloper. Additionally, JDeveloper checks to see if there are matching POMs for those libraries in your local Maven repository. If not, it creates those POMs for you. These new library POMs (if any are needed) are created with the Maven `groupId = com.oracle.adf.library`.

Note that the Maven repository used by JDeveloper can be specified in `Tools/Preferences/Maven/Repository`; it may not be the default repository in `$HOME/.m2`.

Note the following:

- If you want to build these ADF Human Task projects on another machine, for example, a build server, which is using a different Maven repository (local or remote), copy these new POMs to that Maven repository.
- The server on which you build the ADF Human Task projects must have access to a JDeveloper installation, because the ojdeploy Maven plug-in, which is used to package the ADF project into an EAR file, depends on the JDeveloper Oracle Home being present.
- If you want to deploy the EAR file as part of the application build, you need to add a new `<plugin>` section to invoke the `weblogic-maven-plugin:deploy` goal in the appropriate phase of your build, most likely pre-integration-test. Note that the EAR file is located in the *appHome*`/deploy` directory, not in the ADF project's directory. This is due to the fact that a single EAR file may contain multiple ADF Human Task project WAR files.
- If you target the deployment to the SOA server (as shown in the following example), the ADF URI is automatically registered in the appropriate MBean so that the SOA or BPM Workspace application can find it.

```
<plugin>
  <groupId>com.oracle.weblogic</groupId>
  <artifactId>weblogic-maven-plugin</artifactId>
  <version>14.1.2-0-0</version>
  <executions>
    <execution>
      <id>deploy-human-task-ui</ui>
      <goals>
        <goal>deploy</goal>
      </goals>
      <phase>pre-integration-test</phase>
      <configuration>
        <adminurl>t3://localhost:7001</adminurl>
        <user>weblogic</user>
        <password>password</password>
        <source>${project.basedir}/deploy/adf1.ear</source>
        <verbose>true</verbose>
        <name>${project.build.finalName}</name>
        <targets>soa_server1</targets>
      </configuration>
    </execution>
  </executions>
</plugin>
```

- To be able to deploy the EAR file, you also need to set up the appropriate MDS configuration in *appHome*`/.adf/META-INF/adf-config.xml`. This is most likely a database-based MDS store, as shown in What You May Need to Know About Building SOA Projects. For the build to work correctly, and the deployed application to function correctly, this database must be accessible from both the build server and the runtime server. This may not be practical in production environments, so you may need to define multiple deployment profiles for the ADF project.

Undeploying Your SOA Project

You can undeploy your composite using the undeploy goal.

Table 6-7 describes the parameters for the undeploy command:

Table 6-7 Parameters for the Undeploy Goal

Parameters	Purpose
`composite.name`	The name of the composite you want to undeploy.
`composite.revision`	The revision of the composite you want to undeploy.
`composite.partition`	The partition that holds the composite you want to undeploy.
`user`	User name to be used for undeployment.
`password`	Password to be used for undeployment.

To undeploy a SAR file, run the following command, specifying the appropriate values for your environment. Enter this command on one line.

```
mvn com.oracle.soa.plugin:oracle-soa-plugin:undeploy
    -DserverURL=http://test.server:7001
    -Dcomposite.name=my-project
    -Dcomposite.revision=1.0
    -Dcomposite.partition=test
    -Duser=weblogic
    -Dpassword=<password>
```

You should run the undeploy goal against a SOA Project, not a SOA Application.

What You May Need to Know About the SOA Parent POM

The SOA Parent POM is provided as a point of customization. It has Maven coordinates `com.oracle.soa:sar-common:14.1.2-0-0`. If you want to set some environment-wide defaults, for example, the URL, user name, and password for your test server, then you can put these in the SOA Parent POM. The SOA Parent POM is provided as a point of customization. It has Maven coordinates `com.oracle.soa:sar-common:14.1.2-0-0`. If you want to set some environment-wide defaults, for example, the URL, user name, and password for your test server, then you can put these in the SOA Parent POM.

The SOA Parent POM contains the following properties:

```
<properties>
  <!--
  These two properties are defined in com.oracle.maven:oracle-common, you can overwrite
them here.
  Users who do not want to add plain text password in their properties or pom
  file, should use the userConfigFile and userKeyFile options for deployment.
  <oracleUsername>USERNAME</oracleUsername>
  <oraclePassword>PASSWORD</oraclePassword>
  -->

  <!-- Change the default values according to your environment -->
  <oracleServerUrl>http://localhost:8001</oracleServerUrl>
  <oracleServerName>soa_server1</oracleServerName>
  <oracleMiddlewareHome>/opt/soa1412</oracleMiddlewareHome>
</properties>
```

You can set these properties or define any other properties that you want to have available to SOA Projects. To refer to a property in your SOA Project POM, use the syntax `$propertyName`, for example `$oracleServerName` would be replaced with `soa_server1` in the previous example.

7

Building Oracle Service Bus Projects with Maven

You can use the Oracle Service Bus Maven archetypes to create, build, and deploy Oracle Service Bus applications.
For more information about using the Oracle Service Bus development plug-in with Maven, see "Using the Oracle Service Bus Development Maven Plug-In" in *Developing Services with Oracle Service Bus*.

- Introduction to Building Oracle Service Bus Projects with Maven
  Oracle Service Bus provides a Maven plug-in and three archetypes.
- Creating an Oracle Service Bus Application from a Maven Archetype
  You can create a new Oracle Service Bus application (containing an OSB Project and an OSB System Resources project) using the OSB Application Maven archetype.
- Editing Your OSB Application in Oracle JDeveloper
  You can edit your application in Oracle JDeveloper to define OSB resources.
- Creating an Oracle Service Bus Project from a Maven Archetype
  You can create a new Oracle Service Bus Project inside an existing OSB application using the OSB Project Maven archetype.
- Building Your OSB Project with Maven
  After you have written your project code, you can use Maven to build the your OSB project.
- Deploying Your Project to the Oracle Service Bus Server with Maven
  You can deploy your OSB project to the Oracle Service Bus server with Maven.
- What You May Need to Know About the Oracle Service Bus Parent POM
  The OSB Parent POM is provided as a point of customization. For example, you can use it to set some environment-wide defaults, such as the URL, user name, and password for your test server, then you may want to put these in the OSB Parent POM.

Introduction to Building Oracle Service Bus Projects with Maven

Oracle Service Bus provides a Maven plug-in and three archetypes.

The Maven coordinates are described in Table 7-1.

Table 7-1 Maven Coordinates with Oracle Service Bus

Name	GroupId	ArtifactId	Version
OSB plug-in	`com.oracle.servicebus.plugin`	`oracle-servicebus-plugin`	`14.1.2-0-0`
OSB Application archetype	`com.oracle.servicebus.archetype`	`oracle-servicebus-application`	`14.1.2-0-0`
OSB Project archetype	`com.oracle.servicebus.archetype`	`oracle-servicebus-project`	`14.1.2-0-0`

Table 7-1 (Cont.) Maven Coordinates with Oracle Service Bus

Name	GroupId	ArtifactId	Version
OSB System Resources archetype	`com.oracle.servicebus.archetype`	`oracle-servicebus-system`	`14.1.2-0-0`

Table 7-2 describes the goals supported by the Oracle Service Bus plug-in supports the following goals:

Table 7-2 Oracle Service Bus Plug-In Goals

Goal	Purpose
package	Creates a service bus archive (SBAR) file from the project.
deploy	Deploys the SBAR file to a runtime environment. Note that this goal is mapped to the pre-integration-test phase in the default life cycle, not the deploy phase, as deployment to a runtime environment is usually done in the pre-integration-test phase in Maven.

The custom packaging type `sbar` is defined, representing an Oracle Service Bus archive.

Creating an Oracle Service Bus Application from a Maven Archetype

You can create a new Oracle Service Bus application (containing an OSB Project and an OSB System Resources project) using the OSB Application Maven archetype.

To do so, run a command similar to the following:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.oracle.servicebus.archetype \
    -DarchetypeArtifactId=oracle-servicebus-application \
    -DarchetypeVersion=14.1.2-0-0 \
    -DgroupId=org.mycompany \
    -DartifactId=my-servicebus-application \
    -Dversion=1.0-SNAPSHOT \
    -DprojectName=my-project \
    -DconfigJar=myjar.jar
```

This command runs Maven's archetype plug-in's generate goal which allows you to create a new project from an archetype. Table 7-5 describes the parameters.

Table 7-3 Parameters for the Oracle Service Bus Project

Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new project. This must be `com.oracle.servicebus.archetype`, as shown in the preceding example.
`archetypeArtifactId`	The artifact ID of the archetype that you want to use to create the new project. This must be `oracle-servicebus-application`, as shown in the preceding example.

Table 7-3 (Cont.) Parameters for the Oracle Service Bus Project

Parameter	Purpose
`archetypeVersion`	The version of the archetype that you want to use to create the new project. This must be `14.1.2-0-0`, as shown in the preceding example.
`groupId`	The group ID for your new project. This usually starts with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new project. This is usually an identifier for this project.
`version`	The version for your new project. This is usually `1.0-SNAPSHOT` for a new project.
`projectName`	The name for the OSB project to create inside the application. This should be different from the name of the application (that is, `artifactId`), and it cannot be `System`, which is reserved for system resources.
`-DconfigJar`	Used to specify a precompiled `sbconfig.jar` or `.sbar` file. This setting allows deployment of individual jar iles without previous fresh build. It can be an absolute path. This setting requires `-DprojectName` to be set as well.

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

After creating your application, it contains the following files:

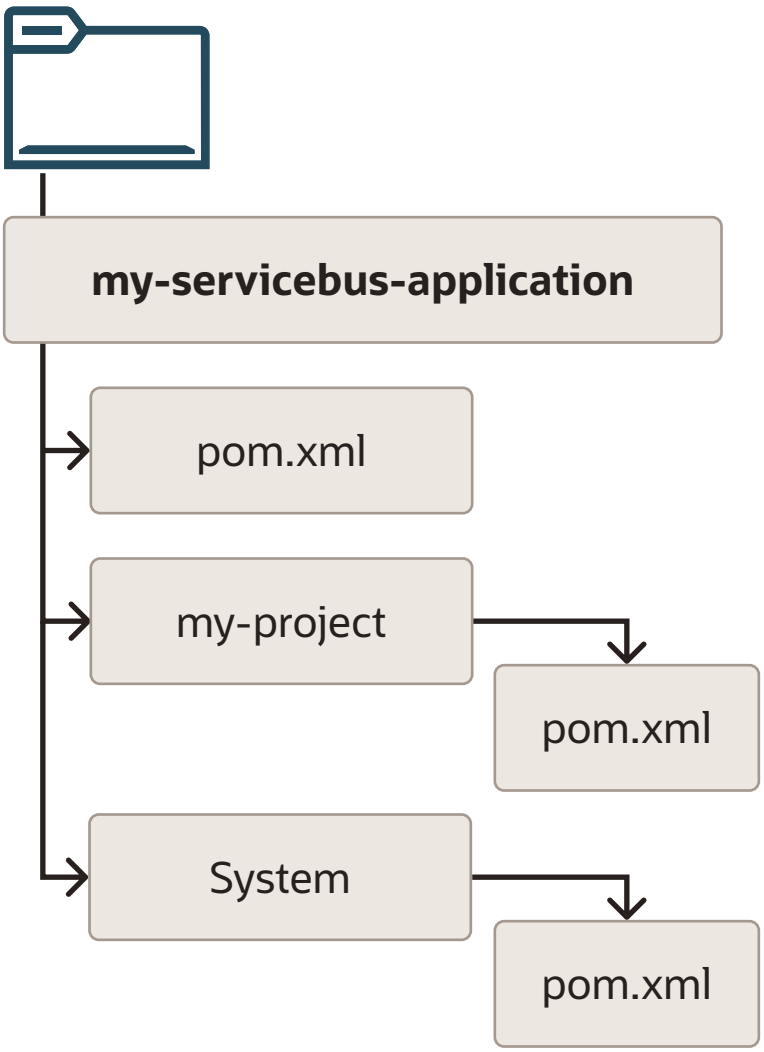


Table 7-4 describes the files included in the project.

Table 7-4 Files Created for the Oracle Service Bus Project

File	Purpose
`pom.xml`	The Maven Project Object Model (POM) file that describes your new application. It includes the Maven coordinates that you specified for your application. This POM is used to group all of the OSB projects that form part of this application.
`my-project/pom.xml`	The Maven POM file that describes your new project. It includes the Maven coordinates that you specified for your project, and the appropriate plug-in definitions to use the Oracle Service Bus Maven plug-in to build your project into an SBAR file.
`System/pom.xml`	A Maven Project Object Model (POM) file for building OSB system resources into a SBAR file.

Editing Your OSB Application in Oracle JDeveloper

You can edit your application in Oracle JDeveloper to define OSB resources.

To edit your application, first open the application in JDeveloper:

1. Open the **File** menu, then select **Import…**.
2. In the **Import** dialog box, select the **Maven Project** option and click **OK**.

   The **Import Maven Projects** dialog appears.
3. In the **Root Directory** field, enter the path to the application you want to import into JDeveloper.
4. In the **Settings File** field, enter the path to your Maven `settings.xml` file. The default value is most likely correct unless you are using a non-standard location for your Maven settings file.
5. Click **Refresh** to load a list of projects available at that location.
6. Select the projects that you want to import. Also select **Update existing JDeveloper Projects to synch with imported POM files**.
7. Click **OK** to complete the import.

Your applications are opened in JDeveloper.

When you import an Oracle Service Bus application (or project) into JDeveloper, you have the choice of creating a new application (or project) directory, or simply creating the JDeveloper project files (`jws` and `jpr` files) in the existing location:

- To create a new copy of the application (or project) in a new directory, select the **Also import source files into application** option and provide a new directory in the import dialog box.
- To create the JDeveloper files in the existing directory, do not select the **Also import source files into application** option, and select the existing directory when prompted for the project location.

Creating an Oracle Service Bus Project from a Maven Archetype

You can create a new Oracle Service Bus Project inside an existing OSB application using the OSB Project Maven archetype.

To do so, run a command similar to the following, from your OSB Application root directory:

```
mvn archetype:generate \
 -DarchetypeGroupId=com.oracle.servicebus.archetype \
 -DarchetypeArtifactId=oracle-servicebus-project \
 -DarchetypeVersion=14.1.2-0-0 \
 -DgroupId=org.mycompany \
 -DartifactId=my-second-project \
 -Dversion=1.0-SNAPSHOT
```

This command runs Maven's archetype plug-in's generate goal, which allows you to create a new project from an archetype. Table 7-5 describes the parameters.

Table 7-5 Parameters for the Oracle Service Bus Project from a Maven Archetype

Parameter	Purpose
`archetypeGroupId`	The group ID of the archetype that you want to use to create the new project. This must be `com.oracle.servicebus.archetype`, as shown in the preceding example.
`archetypeArtifactId`	The artifact ID of the archetype that you want to use to create the new project. This must be `oracle-servicebus-project`, as shown in the preceding example.
`archetypeVersion`	The version of the archetype that you want to use to create the new project. This must be `14.1.2-0-0`, as shown in the preceding example.
`groupId`	The group ID for your new project. This usually starts with your organization's domain name in reverse format.
`artifactId`	The artifact ID for your new project. This is usually an identifier for this project. It cannot be System, which is reserved for system resources.
`version`	The version for your new project. This usually is `1.0-SNAPSHOT` for a new project.

You can also run the command without any arguments, as shown in the following example. In this case, Maven displays a list of available archetypes and prompts you to enter the required information.

```
mvn archetype:generate
```

Note that OSB Projects must be located inside an OSB Application.

After creating your project, it contains the following files:

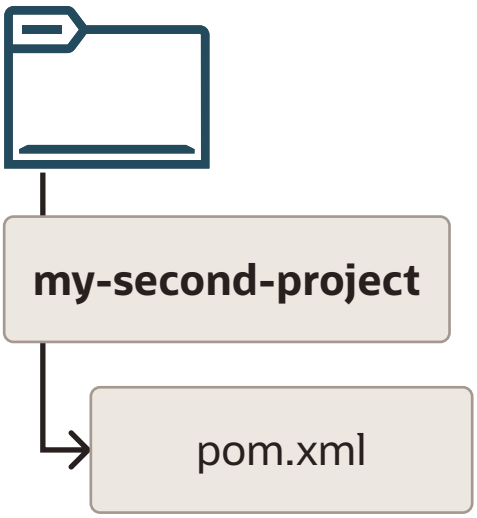


Table 7-6 describes the files included in the project.

Table 7-6 Files Created for the Oracle Service Bus Project

File	Purpose
`pom.xml`	The Maven Project Object Model (POM) file that describes your new project, it includes the Maven coordinates that you specified for your project, and it also includes the appropriate plug-in definitions to use the OSB Maven plug-in to build your project into a SBAR file.

Maven also updates the OSB Application POM file to include this new project. If you ran the preceding command in the application you created in Creating an Oracle Service Bus Application from a Maven Archetype, you would see the following in your OSB Application POM:

```
<modules>
   <module>my-project</module>
   <module>my-servicebus-project</module>
   <module>my-second-project</module>
</modules>
```

Building Your OSB Project with Maven

After you have written your project code, you can use Maven to build the your OSB project.

To build the SBAR file, run the following command:

```
mvn package -DoracleHome=/path/to/osbhome
```

The preceding command creates a SBAR file from your project and places it in:

project`/.data/maven/sbconfig.sbar`

The parameter in Table 7-7 may be specified for the packaging. You can specify it either in the POM file for the project or on the command line, as shown in the preceding example.

Table 7-7 Parameters for the Oracle Service Bus Plug-In package Goal

Parameter	Purpose
`oracleHome`	The location of the Oracle Home for Oracle Fusion Middleware.

Deploying Your Project to the Oracle Service Bus Server with Maven

You can deploy your OSB project to the Oracle Service Bus server with Maven.

To deploy the SBAR file, run the following command:

```
mvn pre-integration-test
```

You can specify the parameters in Table 7-8 for the deployment. You can specify them either in the POM file for the project or on the command line.

Table 7-8 Parameters for the Oracle Service Bus Plug-In deploy Goal

Parameter	Purpose
`oracleHome`	The location of the Oracle Fusion Middleware Oracle Home where OSB is installed.
`oracleServerUrl`	The URL of the server in the OSB domain.
`customization`	(optional) The name of the OSB customization file to use, if any.
`oracleUsername`	User name to be used for deployment.
`oraclePassword`	Password to be used for deployment.
`configJar`	The name of the `.sbar` or `sbconfig.jar` file to be deployed. Use this parameter to specify a precompiled `sbconfig.jar` or `.sbar` file. You can use an absolute path. This setting allows deployment of individual jar files without needing to build the file. If you use this setting, you must also specify the `projectName` parameter.
`projectName`	The name of the project. Use this parameter to specify the project for which the `sbconfig.jar` set by the `configJar` parameter is deployed. This parameter matches the `artifactId` in the POM file of the respective project. This setting is ignored when `DconfigJar` is not present.

To specify the parameters:

- On the command line: Use the format `-Dparameter=value`, as shown in this example (note that you enter the whole command on one line):

```
mvn pre-integration-test
     -DoracleServerUrl=http://test.server:7001
     -DoracleUsername=weblogic
     -DoraclePassword=<password>
     -DconfigJar=D:\ServicebusApp\SBProject\.data\maven\sbconfig.sbar
     -DprojectName=SBProject
```

- In your project POM file: Add a `plugin` section, as shown in the following example:

```
<plugins>

   <plugin>
     <groupId>com.oracle.servicebus</groupId>
     <artifactId>oracle-servicebus-plugin</artifactId>
     <version>14.1.2-0-0</version>
     <extensions>true</extensions>
     <configuration>
       <oracleHome>/u01/osbhome</oracleHome>
       <oracleServerUrl>http://test.server:7001</oracleServerUrl>
       <oracleUsername>weblogic</oracleUsername>
       <oraclePassword><password></oraclePassword>
     </configuration>
   </plugin>
 </plugins>
```

What You May Need to Know About the Oracle Service Bus Parent POM

The OSB Parent POM is provided as a point of customization. For example, you can use it to set some environment-wide defaults, such as the URL, user name, and password for your test server, then you may want to put these in the OSB Parent POM.

You can set these properties or define any other properties that you want to have available to OSB Projects. To refer to a property in your OSB Project POM, use the syntax `$propertyName`. For example, `$oracleServerName` would be replaced with `osb_server1` in the following example.

Projects that are created from the OSB archetypes automatically use values from the OSB Parent POM if you do not override them.

The following is an example of an OSB Parent POM which defines some properties:

```
<properties>
  <!--
   These two properties are defined in
   com.oracle.maven:oracle-common, you can overwrite them here.
   Users who do not want to add plain text password in their
   properties or pom file, should use the userConfigFile and
   userKeyFile options for deployment.
   -->
  <!-- <oracleUsername>USERNAME</oracleUsername> -->
  <!-- <oraclePassword>PASSWORD</oraclePassword> -->

  <!--
   Change the default values according to your environment
   -->
  <oracleServerUrl>t3://localhost:7001</oracleServerUrl>
  <oracleServerName>osb_server1</oracleServerName>
  <oracleHome>/u01/osbhome</oracleHome>
</properties>
```

8

Building a Real Application with Maven

Many real world applications include modules that are targeted to be deployed on different runtime environments. For example, you may have a web application that uses data stored in a Coherence cache.
This chapter describes how to build such a web application.

- Introducing the Maven Example Application
  The example application that you build in this chapter displays a list of people, with their names and age, on a web page. It also allows you to add a new person. The details of the people are stored in a Coherence cache.
- About Multi-Module Maven Projects
  Maven lets you create projects with multiple modules. Each module is in effect another Maven project. At the highest level, you have a POM file that tells Maven about the modules and lets you build the whole application with one Maven command.
- Building a Maven Project
  To build the example project, you create the directory, and then create the GAR, WAR, and EAR projects.

Introducing the Maven Example Application

The example application that you build in this chapter displays a list of people, with their names and age, on a web page. It also allows you to add a new person. The details of the people are stored in a Coherence cache.

This application contains the following parts:

- A Coherence GAR project, which contains a Person POJO (Plain Old Java Object), which you need to build into a Portable Object, a utility class to access the cache, and Coherence cache definitions
- A Jakarta EE web application, which you need to build into a WAR, which contains a servlet and a deployment descriptor
- A project to assemble the GAR and WAR into an EAR and deploy that EAR to WebLogic Server

In this example, you can see how to build a multi-module Maven project, with dependencies between modules, and how to assemble the application components into a deployable EAR file that contains the whole application.

The aim of this chapter is to show how to use Maven to build whole applications, not to demonstrate how to write web or Coherence applications, so the content of the example itself, in terms of the servlet and the Coherence code, is quite basic. For specific steps,, refer to Building Jakarta EE Projects for WebLogic Server with Maven and Building Oracle Coherence Projects with Maven.

About Multi-Module Maven Projects

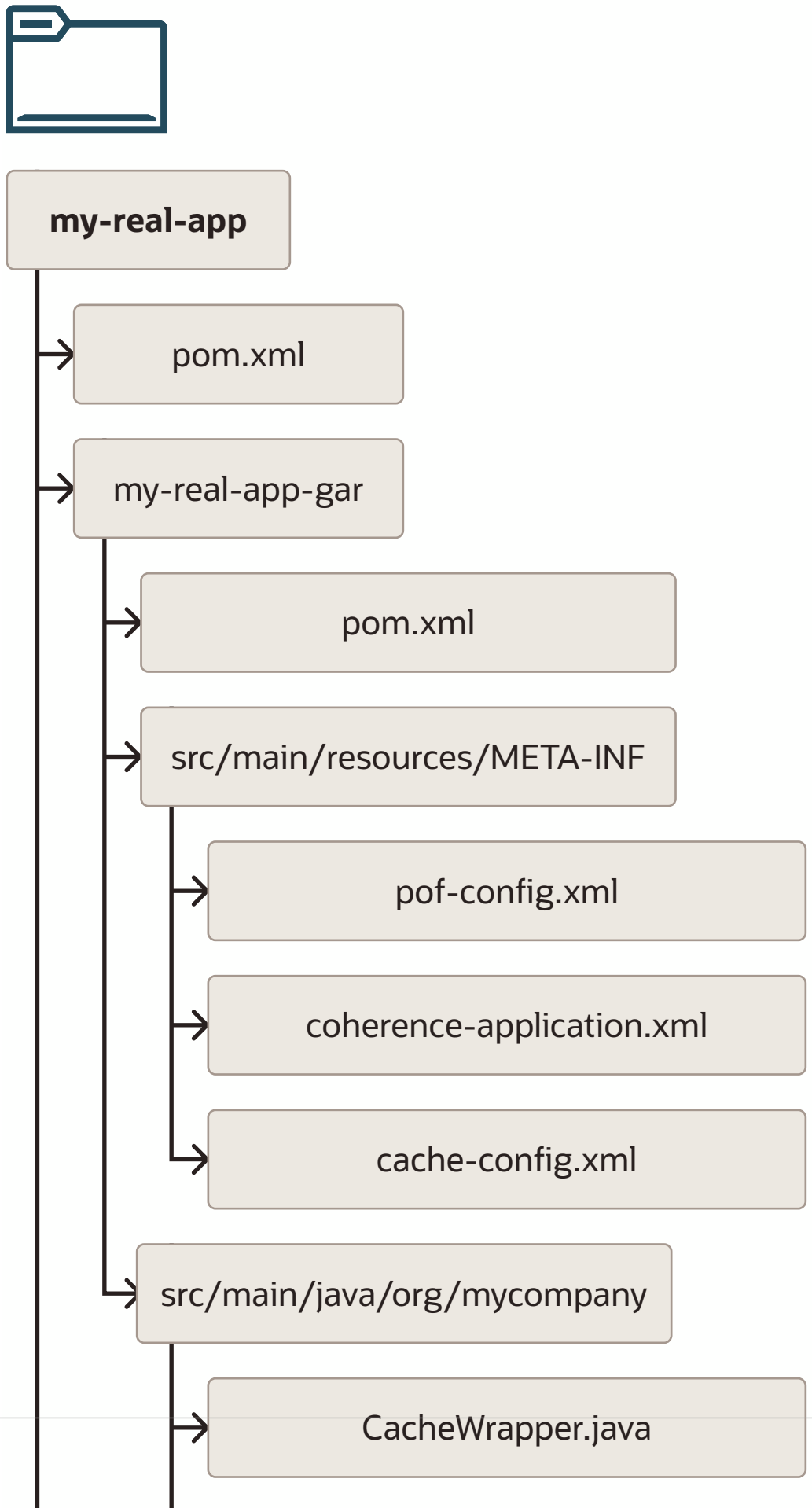
Maven lets you create projects with multiple modules. Each module is in effect another Maven project. At the highest level, you have a POM file that tells Maven about the modules and lets you build the whole application with one Maven command.

Each of the modules are placed in a subdirectory of the root of the top-level project. In the example, the top-level project is called `my-real-app` and the three modules are `my-real-app-gar`, `my-real-app-war`, and `my-real-app-ear`. The Maven coordinates of the projects are shown in Table 8-1.

Table 8-1 Maven Coordinates and Packaging Types for Example Application

GroupId	ArtifactId	Version	Packaging
org.mycompany	my-real-app	1.0-SNAPSHOT	pom
org.mycompany	my-real-app-gar	1.0-SNAPSHOT	gar
org.mycompany	my-real-app-war	1.0-SNAPSHOT	war
org.mycompany	my-real-app-ear	1.0-SNAPSHOT	ear

The following are the files that make up the application:


my-real-app
pom.xml
my-real-app-gar
pom.xml
src/main/resources/META-INF
pof-config.xml
coherence-application.xml
cache-config.xml
src/main/java/org/mycompany
CacheWrapper.java

At the highest level, the top-level POM file points to the three modules:

- The *my-real-app-gar* directory contains the Coherence GAR project. It contains its own POM, the Coherence configuration files, a POJO/POF class definition (`Person.java`) and a utility class that is needed to access the cache (`CacheWrapper.java`).
- The *my-real-app-war* directory contains the web application. It contains its own POM, a servlet, and a deployment descriptor. This project depends on the `my-real-app-gar` project.
- The *my-real-app-ear* directory contains the deployment descriptor for the EAR file and a POM file to build and deploy the EAR.

Building a Maven Project

To build the example project, you create the directory, and then create the GAR, WAR, and EAR projects.

While it is more natural to start with the top-level POM file, doing this will cause issues if you choose to use the archetypes to create the projects. As such, the top-level POM is created at the end, even though the individual project POM files depend on it.

This section includes the following topics:

Creating a Directory for the Projects

Create a directory to hold the projects, using the following command:

```
mkdir my-real-app
cd my-real-app
```

> **Note:**
>
> Throughout the rest of this chapter, paths shown are relative to this directory.

Creating the GAR Project

This section includes the following topics:

- Creating a Wrapper Class to Access the Cache

Creating the Initial GAR Project

You can create the GAR project either using an archetype, as described in Creating a Coherence Project from a Maven Archetype, or you can create the directories and files manually:

- To use the archetype, run the following command:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.oracle.coherence.archetype \
    -DarchetypeArtifactId=gar-maven-archetype \
    -DarchetypeVersion=14.1.2-0-0 \
    -DgroupId=org.mycompany \
    -DartifactId=my-real-app-gar \
    -Dversion=1.0-SNAPSHOT
```

- To create the project manually, use the following commands to create the necessary directories:

```
mkdir -p my-real-app-gar/src/main/resources/META-INF
mkdir -p my-real-app-gar/src/main/java/org/mycompany
```

Creating or Modifying the POM File

If you use the archetype, you already have a POM file. Modify that file to match the following example. If you create the project manually, create the POM file (`my-real-app-gar/pom.xml`) with the following contents:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>my-real-app-gar</artifactId>
  <packaging>gar</packaging>

  <parent>
    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <dependencies>
    <dependency>
      <groupId>com.oracle.coherence</groupId>
      <artifactId>coherence</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>com.oracle.coherence</groupId>
        <artifactId>gar-maven-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <generatePof>true</generatePof>
```

```
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <forceCreation>true</forceCreation>
      </configuration>
    </plugin>
  </plugins>
 </build>
</project>
```

Examine the POM file to understand each part of the file:

- The Maven coordinates `groupId` and `version` are inherited from the top-level POM and the POM only needs to specify the `artifactId` and packaging type. Notice that the packaging is `gar` because you use the Coherence GAR Maven plug-in to build this project into a Coherence GAR file.

  ```
  <artifactId>my-real-app-gar</artifactId>
  <packaging>gar</packaging>
  ```

- The coordinates of the parent project are set. These coordinates point back to the top-level POM to allow for inheritance of configuration.

  ```
  <parent>
    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
  </parent>
  ```

- The `dependencies` section identifies any dependencies that this project has. In this case, you depend only on the Coherence library, that is, `com.oracle.coherence:coherence:14.1.2-0-0`. Notice that the dependency declaration does not need to include the version because that was inherited from the top-level POM file's `dependencyManagement` section. The scope provided means that this library is just for compilation and does not need to be packaged in the artifact that you build (the GAR file) as it is already provided in the runtime environment.

  ```
  <dependencies>
    <dependency>
      <groupId>com.oracle.coherence</groupId>
      <artifactId>coherence</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  ```

- The `plugins` section includes any plug-ins that you want to run during the build that are not automatically included or included plug-ins for which you need to specify additional configuration, like the `maven-jar-plugin`. In this case, you must set `generatePof` to `true` so that the plug-in looks for POJOs with POF annotations and generates the necessary artifacts. You also must tell the `maven-jar-plugin` to always regenerate the GAR file. Notice that the versions of the plug-ins are not provided because those are inherited from the top-level POM file's `pluginManagement` section.

  ```
  <plugins>
    <plugin>
      <groupId>com.oracle.coherence</groupId>
      <artifactId>gar-maven-plugin</artifactId>
      <extensions>true</extensions>
  ```

```
    <configuration>
      <generatePof>true</generatePof>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <configuration>
      <forceCreation>true</forceCreation>
    </configuration>
  </plugin>
</plugins>
```

Creating or Modifying the Coherence Configuration Files

There are three Coherence configuration files that you need in your GAR project. If you use the archetype, the files already exist, but you need to modify them to match the following examples. If you create the project manually, create these files in the following locations:

```
my-real-app-gar/src/main/resources/META-INF/pof-config.xml
my-real-app-gar/src/main/resources/META-INF/coherence-application.xml
my-real-app-gar/src/main/resources/META-INF/cache-config.xml
```

- The following example shows the contents for the `pof-config.xml` file:

  ```
  <?xml version="1.0"?>
  <pof-config
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://xmlns.oracle.com/coherence/coherence-pof-config"
          xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-pof-config
  coherence-pof-config.xsd">
    <user-type-list>
      <include>coherence-pof-config.xml</include>
      <user-type>
        <type-id>1001</type-id>
        <class-name>org.mycompany.Person</class-name>
      </user-type>
    </user-type-list>
  </pof-config>
  ```

  This file requires adding the Person type if you created it with the archetype.

- The following example shows the contents for the `coherence-application.xml` file:

  ```
  <?xml version="1.0" encoding="ISO-8859-1"?>
  <coherence-application xmlns="http://xmlns.oracle.com/weblogic/coherence-
  application">
    <cache-configuration-ref>META-INF/cache-config.xml</cache-configuration-ref>
    <pof-configuration-ref>META-INF/pof-config.xml</pof-configuration-ref>
  </coherence-application>
  ```

  This file requires little or no modification if you created it with the archetype.

- The `cache-config.xml` file must be updated if you have used the archetype.

  In this file, create a cache named People, with a caching scheme named `real-distributed-gar` and a service name of `RealDistributedCache`, which uses the local backing scheme and is automatically started. If you are not familiar with these terms, see Building Oracle Coherence Projects with Maven. The following shows an example of the file:

  ```
  <?xml version="1.0"?>
  <cache-config
  ```

```
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://xmlns.oracle.com/coherence/coherence-cache-config"
      xsi:schemaLocation="http://xmlns.oracle.com/coherence/coherence-cache-config
    coherence-cache-config.xsd">
    <caching-scheme-mapping>
      <cache-mapping>
        <cache-name>People</cache-name>
        <scheme-name>real-distributed-gar</scheme-name>
      </cache-mapping>
    </caching-scheme-mapping>

    <caching-schemes>
      <distributed-scheme>
        <scheme-name>real-distributed-gar</scheme-name>
        <service-name>RealDistributedCache</service-name>
        <backing-map-scheme>
          <local-scheme/>
        </backing-map-scheme>
        <autostart>true</autostart>
      </distributed-scheme>
    </caching-schemes>
  </cache-config>
```

Creating the Portable Objects

Create the Person object, which will store information in the cache. Create a new Java class in the following location:

```
my-real-app-gar/src/main/java/org/mycompany/Person.java
```

The following is the content for this class:

```
package org.mycompany;

import com.tangosol.io.pof.annotation.Portable;
import com.tangosol.io.pof.annotation.PortableProperty;

@Portable
public class Person {
  @PortableProperty(0)
  public String name;

  @PortableProperty(1)
  public int age;

  public Person() {}

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getName() { return this.name; }
  public int getAge() { return this.age; }
}
```

This POJO tells Coherence what to do with the class. Because the focus of this chapter is on building applications with Maven, it does not go into the details of writing Coherence applications. For more information on Coherence, refer to Building Oracle Coherence Projects with Maven.

Creating a Wrapper Class to Access the Cache

Create a small wrapper class that you can use to access the cache. Create another Java class in this location:

```
my-real-app-gar/src/main/java/org/mycompany/CacheWrapper.java
```

The following is the content for this class:

```
package org.mycompany;

import java.util.Map;
import java.util.Set;

import org.mycompany.Person;
import com.tangosol.net.CacheFactory;

public class CacheWrapper {
  private static CacheWrapper INSTANCE = new CacheWrapper();
  public static CacheWrapper getInstance() {
    return INSTANCE;
  }

  public Set<Map.Entry<Object, Object>> getPeople() {
    return CacheFactory.getCache("People").entrySet();
  }

  public void addPerson(int personId, String name, int age) {
    CacheFactory.getCache("People").put(personId, new Person(name, age));
  }
}
```

Later, you can use this class in a servlet to get data from the cache and to add new data to the cache.

Creating the WAR Project

This section includes the following topics:

- Creating the Initial WAR Project
- Creating or Modifying the POM File
- Creating the Deployment Descriptor
- Creating the Servlet

Creating the Initial WAR Project

You can create the WAR project either using an archetype as described in Building Jakarta EE Projects for WebLogic Server with Maven, or you can create the directories and files manually.

- To use the archetype, run the following command:

```
mvn archetype:generate \
  -DarchetypeGroupId=com.oracle.weblogic.archetype \
  -DarchetypeArtifactId=basic-webapp \
  -DarchetypeVersion=14.1.2-0-0 \
  -DgroupId=org.mycompany \
  -DartifactId=my-real-app-war \
  -Dversion=1.0-SNAPSHOT
```

If you use the archetype, then you must remove any unnecessary files included in the project. For example, the generated `AccountBean.java` file needs to be removed because the updated POM file will not include all of the dependencies needed to compile it.

- To create the project manually, use the following commands to create the necessary directories:

```
mkdir -p my-real-app-war/src/main/webapp/WEB-INF
mkdir -p my-real-app-war/src/main/java/org/mycompany/servlets
```

Creating or Modifying the POM File

If you use the archetype, the POM file already exists. Modify that file to match the following example. If you created the project manually, then create the POM file (`my-real-app-war/pom.xml`) with the following contents:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>my-real-app-war</artifactId>
  <packaging>war</packaging>

  <parent>
    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>my-real-app-gar</artifactId>
      <version>${project.version}</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>jakarta.servlet</groupId>
      <artifactId>jakarta.servlet-api</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

Examine the POM file to understand each part of the file:

- As you saw in the GAR project, you only need to set the `artifactId` and packaging type coordinates for this project because the `groupId` and `version` are inherited from the top-level POM file. Notice that the packaging for this project is `war`.

```
<artifactId>my-real-app-war</artifactId>
<packaging>war</packaging>
```

- Define the parent, as you did in the GAR project:

```
<parent>
  <groupId>org.mycompany</groupId>
  <artifactId>my-real-app</artifactId>
  <version>1.0-SNAPSHOT</version>
```

```
  <relativePath>../pom.xml</relativePath>
</parent>
```

- List the dependencies for this project. In this case, there are two dependencies: the GAR project to access the POJO and utility classes you defined there, and the Servlet API. Because the GAR project is built as part of this example application, you use the `${project.groupId}` and `${project.version}` built-in property references to specify the `groupId` and `version` of this dependency.

```
<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>my-real-app-gar</artifactId>
    <version>${project.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

Creating the Deployment Descriptor

The web application has a simple Jakarta EE deployment descriptor that sets the `display-name` for the web application. It resides at the following location:

```
my-real-app-war/src/main/webapp/WEB-INF/web.xml
```

The following are the contents of this file:

```
<web-app version="3.1"
         xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <display-name>my-real-app-war</display-name>
</web-app>
```

Creating the Servlet

To create the servlet, locate the `MyServlet.java` file:

```
my-real-app-war/src/main/java/org/mycompany/servlets/MyServlet.java
```

The servlet displays a list of people that are currently in the cache and allows you to add a new person to the cache. The aim of the section is to learn how to build these types of applications with Maven, not to learn how to write Jakarta EE web applications, hence the use of a simplistic servlet.

The following is the content for the servlet class:

```
package org.mycompany.servlets;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Map;
import java.util.Set;

import javax.servlet.http.HttpServlet;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;

import org.mycompany.Person;
import org.mycompany.CacheWrapper;

@WebServlet(name = "MyServlet", urlPatterns = "MyServlet")
public class MyServlet extends HttpServlet {
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
    throws ServletException, IOException {
    String id = request.getParameter("id");
    String name = request.getParameter("name");
    String age = request.getParameter("age");
    if (name == null || name.isEmpty() ||
        age == null || age.isEmpty() ||
        id == null || id.isEmpty()) {
      // no need to add a new entry
    } else {
      // we have a new entry - so add it
      CacheWrapper.getInstance().addPerson(Integer.parseInt(id),
        name, Integer.parseInt(age));
    }
    renderPage(request, response);
  }

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
    throws ServletException, IOException {
    renderPage(request, response);
  }

  private void renderPage(HttpServletRequest request,
                          HttpServletResponse response)
    throws ServletException, IOException {
    // get the data
    Set<Map.Entry<Object, Object>> people =
      CacheWrapper.getInstance().getPeople();

    PrintWriter out = response.getWriter();
    out.write("<html><head><title>MyServlet" +
              "</title></head><body>");
    out.write("<h2>Add a new person</h2>");
    out.write("<form name=\"myform\" method=\"POST\">");
    out.write("ID:<input type=\"text\" name=\"id\"/><br/>");
    out.write("Name:<input type=\"text\" name=\"name\"/><br/>");
    out.write("Age:<input type=\"text\" name=\"age\"/><br/>");
    out.write("<input type=\"submit\" name=\"submit\" " +
              "value=\"add\"/>");
    out.write("</form>");
    out.write("<h2>People in the cache now</h2>");
    out.write("<table><tr><th>ID</th><th>Name" +
              "</th><th>Age</th></tr>");
    // for each person in data
    if (people != null) {
      for (Map.Entry<Object, Object> entry : people) {
        out.write("<tr><td>" +
          entry.getKey() +
          "</td><td>" +
          ((Person) entry.getValue()).getName() +
```

```
            "</td><td>" +
            ((Person) entry.getValue()).getAge() +
            "</td></tr>");
        }
      }
      out.write("</table></body></html>");
    }
}
```

Check if the user has entered any data in the form. If so, then add a new person to the cache using that data. Note that this application has fairly minimal error handling. To add the new person to the cache, use the `addPerson()` method in the `CacheWrapper` class that you created in your GAR project.

Print out the contents of the cache in a table. In this example, assume that the cache has a reasonably small number of entries, and read them all using the `getPeople()` method in the `CacheWrapper` class.

Creating the EAR Project

The EAR project manages assembling the WAR and the GAR into an EAR.

This section includes the following topics:

- Creating the Initial EAR Project
- About the POM File for the Example Application
- About the Deployment Descriptor for the Example Application

Creating the Initial EAR Project

Create the EAR project manually using the following command:

```
mkdir -p my-real-app-ear/src/main/application/META-INF
```

There are two files in this project: a POM file and a deployment descriptor:

```
my-real-app-ear/pom.xml
my-real-app-ear/src/main/application/META-INF/weblogic-application.xml
```

About the POM File for the Example Application

The following are the contents of the POM file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <artifactId>my-real-app-ear</artifactId>
  <packaging>ear</packaging>

  <parent>
    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../pom.xml</relativePath>
  </parent>
```

```
  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>my-real-app-gar</artifactId>
      <version>${project.version}</version>
      <type>gar</type>
    </dependency>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>my-real-app-war</artifactId>
      <version>${project.version}</version>
      <type>war</type>
    </dependency>
  </dependencies>

  <build>
    <finalName>my-real-app-ear</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <executions>
          <execution>
            <id>copy-gar-locally</id>
            <phase>prepare-package</phase>
            <goals>
              <goal>copy</goal>
            </goals>
            <configuration>
              <artifactItems>
                <artifactItem>
                  <groupId>${project.groupId}</groupId>
                  <artifactId>my-real-app-gar</artifactId>
                  <version>${project.version}</version>
                  <type>gar</type>
                </artifactItem>
              </artifactItems>
            </configuration>
          </execution>
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ear-plugin</artifactId>
        <configuration>
          <outputFileNameMapping>@{artifactId}@-@{version}@.@{extension}@</
outputFileNameMapping>
          <archive>
            <manifest>
              <addClasspath>true</addClasspath>
            </manifest>
          </archive>
          <artifactTypeMappings>
            <artifactTypeMapping type="gar" mapping="jar"/>
          </artifactTypeMappings>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <profiles>
    <profile>
```

```
      <id>integration-test</id>
      <activation>
        <property>
          <name>skipITs</name>
          <value>false</value>
        </property>
      </activation>
      <build>
        <plugins>
          <plugin>
            <groupId>com.oracle.weblogic</groupId>
            <artifactId>weblogic-maven-plugin</artifactId>
            <executions>
              <!--Deploy the application to the server-->
              <execution>
                <id>deploy for integration testing</id>
                <phase>pre-integration-test</phase>
                <goals>
                  <goal>deploy</goal>
                </goals>
                <configuration>
                  <adminurl>${wls.admin.url}</adminurl>
                  <user>${wls.admin.user}</user>
                  <password>${wls.admin.pass}</password>
                  <!--The location of the file or directory to be deployed-->
                  <source>${project.build.directory}/${project.build.finalName}.$
{project.packaging}</source>
                  <!--The target servers where the application is deployed-->
                  <targets>${wls.ear.targets}</targets>
                  <verbose>true</verbose>
                  <name>${project.build.finalName}</name>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
</project>
```

Examine the POM file to understand each part of the file:

- As in the previous projects, you only need to specify the `artifactId`, packaging type, and the parent:

```
<artifactId>my-real-app-ear</artifactId>
<packaging>ear</packaging>

<parent>
  <groupId>org.mycompany</groupId>
  <artifactId>my-real-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <relativePath>../pom.xml</relativePath>
</parent>
```

- The dependencies on the WAR and GAR projects are listed:

```
<dependencies>
    <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>my-real-app-gar</artifactId>
        <version>${project.version}</version>
```

```
        <type>gar</type>
    </dependency>
    <dependency>
        <groupId>${project.groupId}</groupId>
        <artifactId>my-real-app-war</artifactId>
        <version>${project.version}</version>
        <type>war</type>
    </dependency>
</dependencies>
```

- The first plug-in configuration is for the `maven-dependency-plugin`. Configure it to copy the GAR file from the `my-real-app-gar` project's output (target) directory into the EAR project:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-dependency-plugin</artifactId>
  <executions>
    <execution>
      <id>copy-gar-locally</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>copy</goal>
      </goals>
      <configuration>
        <artifactItems>
          <artifactItem>
            <groupId>${project.groupId}</groupId>
            <artifactId>my-real-app-gar</artifactId>
            <version>${project.version}</version>
            <type>gar</type>
          </artifactItem>
        </artifactItems>
      </configuration>
    </execution>
  </executions>
</plugin>
```

- The second plug-in configuration is for the `maven-ear-plugin`. You need to tell it to treat a GAR file like a JAR file by adding an `artifactTypeMapping`, as in the following example:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-ear-plugin</artifactId>
  <configuration>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
      </manifest>
    </archive>
    <artifactTypeMappings>
      <artifactTypeMapping type="gar" mapping="jar"/>
    </artifactTypeMappings>
  </configuration>
</plugin>
```

- The `profile` section adds a third plug-in configuration for the `weblogic-maven-plugin` that tells it how to deploy the resulting EAR file. Notice that the `adminurl`, `user`, `password`, and `targets` parameters' values are set to properties that will be defined in the top-level POM file.

  The profile is activated by adding `-DskipITs=false` to the Maven command line. This makes it easier to build the EAR file without needing to deploy it on every build.

```
<profile>
  <id>integration-test</id>
  <activation>
    <property>
      <name>skipITs</name>
      <value>false</value>
    </property>
  </activation>
  <build>
    <plugins>
      <plugin>
        <groupId>com.oracle.weblogic</groupId>
        <artifactId>weblogic-maven-plugin</artifactId>
        <executions>
          <!--Deploy the application to the server-->
          <execution>
            <id>deploy for integration testing</id>
            <phase>pre-integration-test</phase>
            <goals>
              <goal>deploy</goal>
            </goals>
            <configuration>
              <adminurl>${wls.admin.url}</adminurl>
              <user>${wls.admin.user}</user>
              <password>${wls.admin.pass}</password>
              <!--The location of the file or directory to be deployed-->
              <source>${project.build.directory}/${project.build.finalName}.$
{project.packaging}</source>
              <!--The target servers where the application is deployed-->
              <targets>${wls.ear.targets}</targets>
              <verbose>true</verbose>
              <name>${project.build.finalName}</name>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</profile>
```

After you have completed the POM project, add a deployment descriptor.

About the Deployment Descriptor for the Example Application

The WebLogic deployment descriptor for the EAR file is located in this file:

```
my-real-app-ear/src/main/application/META-INF/weblogic-application.xml
```

The following are the contents:

```
<weblogic-application>
       <module>
             <name>GAR</name>
             <type>GAR</type>
             <path>my-real-app-gar-1.0-SNAPSHOT.gar</path>
       </module>
</weblogic-application>
```

This deployment descriptor provides the details for where in the EAR file, the GAR file should be placed, and what it should be called.

Creating the Top-Level POM

Create the top-level POM. This is located in the `pom.xml` file in the root directory of your application and contains the following:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.mycompany</groupId>
  <artifactId>my-real-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>

    <fmw.version>14.1.2-0-0</fmw.version>
    <coherence.version>${fmw.version}</coherence.version>
    <weblogic.version>${fmw.version}</weblogic.version>

    <wls.admin.url>t3://localhost:7001</wls.admin.url>
    <wls.admin.user>weblogic</wls.admin.user>
    <wls.admin.pass>password</wls.admin.pass>
    <wls.ear.targets>AdminServer</wls.ear.targets>
  </properties>

  <modules>
    <module>my-real-app-gar</module>
    <module>my-real-app-war</module>
    <module>my-real-app-ear</module>
  </modules>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.oracle.coherence</groupId>
        <artifactId>coherence</artifactId>
        <version>${coherence.version}</version>
      </dependency>
      <dependency>
        <groupId>jakarta.servlet</groupId>
        <artifactId>jakarta.servlet-api</artifactId>
        <version>4.0.4</version>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>com.oracle.coherence</groupId>
          <artifactId>gar-maven-plugin</artifactId>
          <version>${coherence.version}</version>
        </plugin>
```

```
        <plugin>
          <groupId>com.oracle.weblogic</groupId>
          <artifactId>weblogic-maven-plugin</artifactId>
          <version>${weblogic.version}</version>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-dependency-plugin</artifactId>
          <version>3.8.1</version>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-ear-plugin</artifactId>
          <version>3.3.0</version>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-enforcer-plugin</artifactId>
          <version>3.5.0</version>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <version>3.4.2</version>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-war-plugin</artifactId>
          <version>3.4.0</version>
        </plugin>
      </plugins>
    </pluginManagement>

    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-enforcer-plugin</artifactId>
        <executions>
          <execution>
            <id>enforce-java-version</id>
            <goals>
              <goal>enforce</goal>
            </goals>
            <configuration>
              <rules>
                <requireJavaVersion>
                  <version>[17,18),[21,22)</version>
                  <message>You must use JDK 17 or JDK 21 to build the project</message>
                </requireJavaVersion>
              </rules>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Set the coordinates for the project. These match the parent coordinates that you specified in each of the three projects. Note that the packaging is `pom`. This tells Maven that this project itself does not create an artifact. In this example, it simply organizes the build for a set of

modules, as named in the `modules` section. There is one module entry for each of the three subprojects.

Because this POM is the parent of the other three, and because POM's inherit from their parents, you can add any common properties to this POM and they will be available in all the three subprojects. In this case, you are adding three sections:

- `properties` section: This allows you to define variables that can be used in most locations in the current POM file and any module POM files. The first three variables are ones used to configure the `maven-compiler-plugin`. You could have declared the `maven-compiler-plugin` as a normal plug-in and added configuration to accomplish the same thing, but using these properties makes for less configuration—a Maven best practice.

  Next, you have defined the `fmw.version` property to hold the version of Oracle Fusion Middleware that you will use. For convenience, you also define the `coherence.version` and `weblogic.version` properties and set their values to the value of the `fmw.version` property.

  Finally, you have defined properties that specify the WebLogic Server URL, credentials, and location to which to target the application for testing.

- `dependencyManagement` section: This section allows you to centrally manage the versions of dependencies used by the project and all of its modules. In this case, the project only has two external dependencies that are needed for compilation but already provided at runtime: the core Coherence dependency and the Jakarta Servlet API.

- `pluginManagement` section: This section allows you to centrally manage the versions of the plug-ins used by the project and all of its modules. In this example, these include the following plug-ins:

  - `gar-maven-plugin`: Assembles the Coherence GAR file
  - `weblogic-maven-plugin`: Deploys the EAR file to WebLogic Server for integration testing
  - `maven-dependency-plugin`: Copies the Coherence GAR file so that it can be included in the EAR file
  - `maven-enforcer-plugin`: Ensures that the build uses a compatible Java version
  - `maven-ear-plugin`: Assembles the Jakarta EE EAR file
  - `maven-jar-plugin`: Participates in assembling the Coherence GAR file
  - `maven-war-plugin`: Assembles the Jakarta EE WAR file

- `plugin` section: This section configures the Maven Enforcer Plugin to make sure that the build is run with either JDK 17 or JDK 21.

Building the Application Using Maven

You are now ready to build and deploy the application. Before running Maven to build and deploy the application, there are a few things you may need to know. The following sections will cover these topics:

- What You May Need to Know About Coherence Networking
- What You May Need to Know About Maven Dependency Resolution
- Running Maven to Build and Deploy the Application
- Accessing the Application

What You May Need to Know About Coherence Networking

By default, Coherence tries to select the appropriate network interface to use and tries to form its cluster using IP multicast. On some machines, Coherence may select the wrong interface or Coherence may decide that the selected interface does not support IP multicast. This can cause the WebLogic Maven Plugin deploy command to hang until it times out. Once deployed, the server may also print messages like the following:

```
2024-11-06 18:48:27.677/52.215 Oracle Coherence GE 14.1.2.0.0 <Warning>
      (thread=Cluster, member=n/a): Delaying formation of a new cluster;
multicast networking
      appears to be inoperable on interface 192.168.1.179 as this process isn't
receiving even its
      own transmissions; consider forcing IPv4 via -
Djava.net.preferIPv4Stack=true
```

Should you run into such issues, the easiest solution for this simple example it to set Java System Property `coherence.wka` to `127.0.0.1` when starting WebLogic Server. This will cause Coherence to use unicast instead of multicast and use the network interface associated with 127.0.0.1. To pass this system property to WebLogic Server, simply set the `JAVA_OPTIONS` environment variable prior to running the `startWebLogic` script. On macOS or Linux, do the following:

```
export JAVA_OPTIONS="-Dcoherence.wka=127.0.0.1"
```

On Windows, do the following:

```
set "JAVA_OPTIONS=-Dcoherence.wka=127.0.0.1"
```

What You May Need to Know About Maven Dependency Resolution

When building Maven projects, there are two ways that Maven uses to resolve dependencies:

- Fetching dependencies from a Maven repository
- Passing dependencies between modules in a multi-module build

To fetch a dependency from a Maven repository, the dependency must exist in either a remote repository (for example, Maven Central) or in the local Maven repository. Any dependency that isn't available in the local Maven repository is fetched from the remote repository and added to the local Maven repository. To put build artifacts in the local Maven repository, you typically run `mvn install`. All artifacts in the local repository can be fetched by Maven builds without any restrictions.

In this example, we have three modules—some of which have dependencies on other modules in the build. The `my-real-app-war` module depends on the `my-real-app-gar` module and the `my-real-app-ear` module depends on both the `my-real-app-gar` and `my-real-app-war`. If you try to build the `my-real-app-war` module by itself before building and installing the `my-real-app-gar` module artifacts in the local Maven repository, you will get an error like this:

```
[ERROR] Failed to execute goal on project my-real-app-war: Could not resolve
      dependencies for project org.mycompany:my-real-app-war:war:1.0-SNAPSHOT:
The following
      artifacts could not be resolved: org.mycompany:my-real-app-gar:jar:1.0-
SNAPSHOT (absent):
      Could not find artifact org.mycompany:my-real-app-gar:jar:1.0-SNAPSHOT ->
```

```
[Help
      1]
```

There are two ways to resolve this issue:

- Run `mvn install` in the `my-real-app-gar` module directory
- Run the build with any target, such as `mvn package`, from the top-level `my-real-app` directory

Maven understands multi-module builds and creates an in-memory structure called the Maven Reactor that it uses to pass around information about the build to the various modules. Part of this information is about how to resolve cross-module dependencies that may not be present in the local Maven Repository. This allows Maven to build the `my-real-app-war` module using the reactor information about the `my-real-app-gar` module that is not in the local Maven repository.

Running Maven to Build and Deploy the Application

You can now build the application using Maven by using one or more of the following commands (in the top-level directory `my-real-app`):

```
mvn compile
mvn package
mvn verify
mvn install
mvn install -DskipITs=false
```

Maven runs all of the phases up to and including the one named. Table 8-2 shows the effects of each command.

Table 8-2 Effects of Maven Commands

Command	Details
`mvn compile`	Compiles the Java source into target class files.
`mvn package`	1. Compiles the Java source into target class files. 2. Runs any unit tests you may have written. 3. Creates the archive (WAR, GAR, and so on) containing compiled files and resources (deployment descriptors, configuration files, and so on).
`mvn verify`	1. Compiles the Java source into target class files. 2. Runs any unit tests you may have written. 3. Creates the archive (WAR, GAR, and so on) containing compiled files and resources (deployment descriptors, configuration files, and so on). 4. Deploys the EAR file to the WebLogic Server environment.[1] 5. Runs any integration tests you may have written. 6. Verifies the integration test results, if any.

Table 8-2 (Cont.) Effects of Maven Commands

Command	Details
`mvn install`	1. Compiles the Java source into target class files. 2. Runs any unit tests you may have written. 3. Creates the archive (WAR, GAR, and so on) containing compiled files and resources (deployment descriptors, configuration files, and so on). 4. Deploys the EAR file to the WebLogic Server environment.[2] 5. Runs any integration tests you may have written. 6. Verifies the integration test results, if any. 7. Copies the artifact, if any, and the POM file to the local Maven repository.

[1] Deployment only happens if the `-DskipITs=false` argument is passed to Maven.
[2] Deployment only happens if the `-DskipITs=false` argument is passed to Maven.

Remember, the typical Maven build will skip deploying the application unless you add `-DskipITs=false`. This makes it easier to build the application without having to deploy each time. After the application is built and successfully deployed, it is time to test the application to ensure that it is working properly.

Accessing the Application

After the application is successfully deployed, simply use the `/my-real-app-war/MyServlet` URL context path to see the application; for example, `http://127.0.0.1:7001/my-real-app-war/MyServlet`.