

## **Oracle® Fusion Middleware**

Developing Applications Using Continuous Integration

12c (12.1.2)

**E26997-02**

February 2014

Describes how to build automation and continuous integration for applications that you develop and deploy to a Fusion Middleware runtime environment. It uses Subversion, Maven, Archiva, Hudson, and Oracle Maven plug-ins to demonstrate continuous integration,

Oracle Fusion Middleware Developing Applications Using Continuous Integration, 12c (12.1.2)

E26997-02

Copyright © 2013, 2014, Oracle and/or its affiliates. All rights reserved.

Primary Author: Helen Grembowicz

Contributing Author: Wortimla RS, Sreetama Ghosh

Contributor: Mark Nelson, Leon Franzen

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

---

---

# Contents

<b>Preface</b> .....	vii
Audience .....	vii
Documentation Accessibility .....	vii
Related Documents .....	vii
Conventions .....	vii
<b>1 Introduction to Continuous Integration</b>	
1.1 Introducing Continuous Integration for Oracle Fusion Middleware .....	1-1
1.2 Version Control with Subversion .....	1-3
1.3 Build Automation and Dependency Management with Maven .....	1-3
1.4 Repository Management with Archiva .....	1-4
1.5 Continuous Integration with Hudson .....	1-5
1.6 Summary .....	1-6
<b>2 Roadmap for Continuous Integration</b>	
2.1 Roadmap .....	2-1
2.2 Overview of the Reference Continuous Integration Environment .....	2-3
2.3 Shared Disk Layout .....	2-4
<b>3 Installing and Configuring Subversion for Version Control</b>	
3.1 Downloading Subversion .....	3-1
3.2 Installing Subversion .....	3-1
3.3 Configuring the Subversion Server as a Service .....	3-2
3.4 Setting Up a Repository .....	3-2
3.4.1 Creating a Repository .....	3-2
3.4.2 Subversion Layout .....	3-4
3.4.3 Importing Existing Projects .....	3-5
3.5 Understanding SVN Workflow .....	3-5
3.6 Considerations for Tagging and Branching .....	3-7
3.7 Subversion Clients .....	3-7
3.7.1 WebSVN .....	3-7
3.7.2 TortoiseSVN .....	3-8
3.8 More Information .....	3-8

## 4 Installing and Configuring the Archiva Maven Repository Manager

4.1	Overview of Archiva .....	4-1
4.2	Downloading Archiva .....	4-1
4.3	Installing Archiva .....	4-2
4.4	Configuring Archiva .....	4-2
4.4.1	Configuring the Server .....	4-2
4.4.2	Starting the Server .....	4-3
4.4.3	Creating an Administrator User .....	4-3
4.4.4	Internal and Snapshot Repositories .....	4-3
4.4.5	Proxy Repository .....	4-4
4.4.6	Configuring Mirror Repositories .....	4-4
4.4.7	Creating Development, Production, Quality Assurance, and Test Repositories .....	4-6
4.4.8	Creating a Deployment Capable User .....	4-7
4.5	More Information .....	4-8
4.6	Maven Repository Manager Administration .....	4-8
4.6.1	Snapshot Cleanup .....	4-8
4.6.1.1	Retention Options .....	4-10
4.6.1.2	Deleting Released Snapshots .....	4-10
4.6.2	Advanced User Management .....	4-10
4.6.3	Backing Up Archiva .....	4-10
4.6.4	Archiva Failover .....	4-10

## 5 Installing and Configuring Maven for Build Automation and Dependency Management

5.1	Setting Up the Maven Distribution .....	5-1
5.2	Customizing Maven Settings .....	5-2
5.3	Populating the Maven Repository Manager .....	5-3
5.3.1	Introduction to the Maven Synchronization Plug-In .....	5-4
5.3.2	Installing Oracle Maven Synchronization Plug-In .....	5-4
5.3.3	Running the Oracle Maven Synchronization Plug-In .....	5-5
5.3.4	Things to Know About Replacing Artifacts .....	5-7
5.3.5	Populating Your Maven Repository .....	5-7
5.3.5.1	Populating a Local Repository .....	5-7
5.3.5.2	Populating a Remote Repository .....	5-8
5.3.6	Running the Push Goal .....	5-9
5.3.7	Things to Know About Patching .....	5-10
5.3.7.1	Oracle's Approach to Patching .....	5-10
5.3.7.2	Maintain One Maven Repository for Each Environment .....	5-10
5.3.7.3	Run the Oracle Maven Synchronization Plug-In Push Goal After Patching .....	5-10
5.3.8	Considerations for Archetype Catalogs .....	5-10
5.3.9	Example settings.xml .....	5-11
5.3.10	Deploying a Single Artifact .....	5-14

## 6 Installing and Configuring Hudson for Continuous Integration

6.1	Prerequisites for Installing and Configuring Hudson .....	6-1
6.2	Downloading Hudson .....	6-1

6.3	Installing Hudson .....	6-2
6.3.1	Installing Hudson on Linux .....	6-2
6.3.2	Installing Hudson on Windows .....	6-2
6.4	Configuring the HTTP Port .....	6-2
6.5	Starting Hudson .....	6-3
6.6	Configuring Maven After Startup .....	6-3
6.6.1	First Time Startup .....	6-3
6.6.2	Configuring the JDK .....	6-4
6.6.3	Specifying the Maven Home .....	6-4
6.6.4	Setting Up Maven for Use by Hudson .....	6-4
6.6.5	Installing Hudson Plug-Ins .....	6-5
6.6.6	Integrating the Repository .....	6-6
6.6.7	Monitoring Subversion .....	6-6
6.7	For More Information .....	6-6

## 7 Understanding Maven Version Numbers

7.1	How Version Numbers Work in Maven .....	7-1
7.2	The SNAPSHOT Qualifier .....	7-2
7.3	Version Range References .....	7-3
7.4	Understanding Maven Version Numbers in Oracle Provided Artifacts .....	7-4
7.4.1	Version Numbers in Maven Coordinates .....	7-4
7.4.2	Version Number Ranges in Dependencies .....	7-5

## 8 Customizing the Build Process with Maven POM Inheritance

## 9 Building Java EE Projects for WebLogic Server with Maven

9.1	Introduction to Building Java EE Project with Maven .....	9-1
9.2	Using the Basic WebApp Maven Archetype .....	9-1
9.2.1	Customizing the Project Object Model File to Suit Your Environment .....	9-3
9.2.2	Compiling Your Project .....	9-4
9.2.3	Packaging Your Project .....	9-4
9.2.4	Deploying Your Project to the WebLogic Server Using Maven .....	9-4
9.2.5	Deploying Your Project to the WebLogic Server Using Different Options .....	9-4
9.2.6	Testing Your Basic WebApp Project .....	9-5
9.3	Using the Basic WebApp with EJB Maven Archetype .....	9-5
9.4	Using the Basic WebService Maven Archetype .....	9-7
9.5	Using the Basic MDB Maven Archetype .....	9-10

## 10 Building Oracle Coherence Projects with Maven

10.1	Introduction to Building Oracle Coherence Projects with Maven .....	10-1
10.2	Creating a Project from a Maven Archetype .....	10-2
10.3	Building Your Project with Maven .....	10-3
10.4	Deploying Your Project to the WebLogic Server Coherence Container with Maven .....	10-4
10.5	Building a More Complete Example .....	10-4

## 11 Building a Real Application with Maven

11.1	Introducing the Example .....	11-1
11.2	Multi-Module Maven Projects .....	11-2
11.3	Getting Started Building a Maven Project .....	11-2
11.4	Creating the GAR Project .....	11-3
11.4.1	The POM File .....	11-3
11.4.2	Creating or Modifying the Coherence Configuration Files .....	11-5
11.4.3	Creating the Portable Objects .....	11-6
11.4.4	Creating a Wrapper Class to Access the Cache .....	11-7
11.5	Creating the WAR project .....	11-7
11.5.1	Creating or Modifying the POM File .....	11-8
11.5.2	Creating the Deployment Descriptor .....	11-9
11.5.3	Creating the Servlet .....	11-9
11.6	Creating the EAR project .....	11-11
11.6.1	The POM File .....	11-11
11.6.2	Deployment Descriptor .....	11-15
11.7	Creating the Top-Level POM .....	11-15
11.8	Building the Application Using Maven .....	11-16

## 12 From Build Automation to Continuous Integration

12.1	Dependency Management .....	12-1
12.1.1	Using SNAPSHOT .....	12-2
12.1.2	Dependency Transitivity .....	12-2
12.1.3	Dependency Scope .....	12-2
12.1.4	Multiple Module Support .....	12-3
12.2	Maven Configuration to Support Continuous Integration Deployment .....	12-3
12.2.1	Distribution Management .....	12-3
12.2.2	Snapshot Repository Settings .....	12-4
12.3	Automating the Build with Hudson .....	12-5
12.3.1	Creating a Hudson Job to Build a Maven Project .....	12-5
12.3.2	Triggering Hudson Builds .....	12-6
12.3.2.1	Manual Build Triggering .....	12-7
12.3.2.2	Subversion Repository Triggering .....	12-7
12.3.2.3	Schedule Based Triggering .....	12-7
12.3.2.4	Trigger on Hudson Dependency Changes .....	12-7
12.3.2.5	Maven SNAPSHOT Changes .....	12-7
12.3.3	Managing a Multi-Module Maven Build with Hudson .....	12-7
12.4	Monitoring the Build .....	12-8
12.4.1	Following Up on the Triggered Builds .....	12-8

---

---

# Preface

This book is about 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 12c 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 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.

## Related Documents

For more information, see the following documents in the Oracle Other Product One Release 7.0 documentation set or in the Oracle Other Product Two Release 6.1 documentation set:

- *Oracle Fusion Middleware Developing Applications for Oracle WebLogic Server*

## Conventions

The following text conventions are used in this document:

<b>Convention</b>	<b>Meaning</b>
<b>boldface</b>	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	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.



---

---

# Introduction to Continuous Integration

This chapter introduces the core concepts of continuous integration and explores a set of tools that can be used to create a continuous integration environment in the context of Oracle Fusion Middleware.

This chapter contains the following sections:

- [Section 1.1, "Introducing Continuous Integration for Oracle Fusion Middleware"](#)
- [Section 1.2, "Version Control with Subversion"](#)
- [Section 1.3, "Build Automation and Dependency Management with Maven"](#)
- [Section 1.4, "Repository Management with Archiva"](#)
- [Section 1.5, "Continuous Integration with Hudson"](#)
- [Section 1.6, "Summary"](#)

## 1.1 Introducing Continuous Integration for Oracle Fusion Middleware

When enterprises develop applications to support their business needs, they typically employ teams of developers who work together, often in small teams, with each team building a part of the application. These parts are then assembled to create the whole application.

Many modern applications are based on a service-oriented architecture (SOA). This means that developers build services (small pieces of business functionality) that can be assembled in various ways to meet the needs of the business application. Some of the features of SOA that make it popular today are:

- Loose coupling of components of the application, which reduces the impact of change
- Reuse of services, a long time goal of Information Technology development
- The flexibility and agility to easily change the application's behavior as the business need changes

In this new paradigm, many development organizations are also adopting iterative development methodologies to replace the older waterfall-style methodologies. Iterative, agile development methodologies focus on delivering smaller increments of functionality more often than traditional waterfall approaches. Proponents of the new approach claim that the impact is usually less for errors that are found sooner and that the approach is especially suitable to today's environment of constant and rapid change in business requirements.

Many of these techniques also feature the adoption of continuous integration. Organizations have a strong interest in automating their software builds and testing, and continuous integration can help accomplish this.

Continuous integration is a software engineering practice that attempts to improve quality and reduce the time taken to deliver software by applying small and frequent quality control efforts. It is characterized by these key practices:

- A version control system is used to track changes.
- All developers commit to the main code line, head and trunk, every day.
- The product is built on every commit operation.
- The build must be automated and fast.
- There should be automated deployment to a production-like environment.
- Automated testing should be enabled.
- Results of all builds are published, so that everyone can see if anyone breaks a build.
- Deliverables are easily available for developers, testers, and other stakeholders.

Oracle Fusion Middleware 12c provides support for enterprises that adopt continuous integration techniques to develop applications on the Oracle Fusion Middleware platform. Specifically, it provides the following:

- Integration with common version control systems from the development tool, Oracle JDeveloper
- The ability to build projects from the command line using Maven, a build and project management system, so that the build can be scripted and automated
- The ability to create new projects from Maven archetypes
- The ability to download necessary dependencies from Maven repositories
- The ability to parameterize projects so that builds can be targeted to different environments, such as Test, QA, SIT, and production
- The ability to include testing of projects in the Maven build life cycle
- The ability to populate a Maven repository with Oracle-provided dependencies from an existing local Oracle home software installation directory
- The ability to run Maven builds under the control of a continuous integration server like Hudson
- Comprehensive documentation about setting up your build or continuous integration environment, or both, to use with Oracle Fusion Middleware

Choices are available for version control, continuous integration, and other components that enterprises typically use in this kind of environment. Many of these components are free and open source, and others are commercial products. This guide presents a reference environment based on the following set of components:

- Apache Subversion for version control
- Apache Maven for build or project management
- Apache Archiva as the Maven Repository Manager
- Apache Hudson as the continuous integration server

Note that these are not the only choices available. You can use, for example, a different version control system or a different continuous integration server. For most common

alternatives, you should be able to adapt the examples in this guide without much difficulty.

## 1.2 Version Control with Subversion

Subversion is a popular version control system. It was originally created as a logical successor to the Concurrent Versioning System (CVS), which is still widely used today. Subversion is used as the version control system in the examples in this guide for the following reasons:

- It is well integrated with Oracle JDeveloper, the development tool that is most commonly used to build applications for the Oracle Fusion Middleware platform and with other common development tools.
- It works well in various network environments, including virtual private networks and HTTP proxies. Thus, it is well suited for the kind of network environments often encountered in enterprises and their partners, and suppliers.
- It supports various authentication options, including strong authentication with certificates.
- For projects using Oracle SOA Suite, it provides an atomic commit that enables developers to update several files as part of a single check-in or commit operation.

A typical Subversion environment consists of one or more Subversion repositories that store source code artifacts. These are accessed by developers using Subversion clients, either included in their integrated development environments or as standalone clients. Developers can copy artifacts to and from the repositories. When a developer changes an artifact, a new version of the artifact is created in the repository. Developers can view and compare versions of artifacts to see what was changed and who changed it.

## 1.3 Build Automation and Dependency Management with Maven

Maven is a project management and build management system. Maven provides project management in terms of:

- Naming and version numbering
- Dependencies
- Where the source code is stored
- Where builds are stored
- Templates for project types
- The release process

Maven provides build management in terms of:

- How to execute the build
- What to do in each phase
- Parameterization of the build
- An extensible framework

Maven is based on the central concept of a build life cycle. The process for building and distributing a particular artifact or project is clearly defined.

For developers to use Maven, they must learn a small set of commands that enable them to build any Maven project. The Maven Project Object Model (POM) ensures that the project is built correctly.

There are three main life cycles defined:

- **Default** to build the project
- **Clean** to remove all generated artifacts from the project
- **Site** to build documentation for the project

Build life cycles are further defined by a set of build phases. A build phase represents a stage in the life cycle. Build phases are executed sequentially to complete the life cycle. Build phases consist of goals that perform the actual tasks. There is a default set of goal bindings for a standard life cycle phases. Maven plug-ins contribute additional goals to a project. The following table shows the build phase and purpose of each phase.

Build Phase	Purpose
validate	Ensure that the project is correct and all necessary information is available.
compile	Compile the source of the project.
test	Test the compiled source code using a suitable unit testing framework; tests should not require the code to be packaged or deployed.
package	Take the compiled code and package it in its distributable format, such as a JAR, WAR, EAR, SAR, or GAR files.
integration-test	Process and deploy the package if necessary into an environment where integration tests can be run.
verify	Run checks to verify whether the package is valid and meets quality criteria.
install	Install the package into the local repository for use as a dependency in other projects locally.
deploy	For the final release, copy the final package to a remote repository for sharing with other developers and projects.

A typical Maven environment consists of Maven installation on each developer's local machine, a shared Maven repository manager within the enterprise, and one or more public Maven repositories where dependencies are stored. The main Maven repository is known as Maven's central repository. This repository stores many free and open source libraries that are commonly used as dependencies during development projects. Examples include the JUnit unit testing framework; Spring, Struts, and other common user interface libraries; and code coverage and style-checking libraries like Cobertura and PMD.

## 1.4 Repository Management with Archiva

When several developers are working on a project, enterprises often find it useful to establish their own internal Maven repository for two purposes:

- To act as a proxy or cache for external Binary repositories, like Maven's central repository, so that dependencies are downloaded only once and cached locally so that all developers can use them.
- To store artifacts that are built by the developers so that they can be shared with other developers or projects.

Although a Maven repository can be as simple as a file system in a particular layout (directory structure), most organizations find that it is more convenient to use a type of

software called a Maven repository manager. This helps in addressing the purposes previously listed. In this guide, Apache Archiva is used as the Maven Repository Manager. Others are available, either for free and commercially.

In a typical enterprise that use Archiva, Archiva is set up on a server that is accessible to developers and build machines. The enterprise defines the following repositories on this server:

- A mirror of Maven's central repository
- An internal repository to store internally developed artifacts that are completed or published
- A snapshot repository to store internally developed artifacts that are under development and not completed yet.

There can also be additional repositories depending on the need. For example, there can be additional repositories for particular projects, or for different versions of dependencies needed for different life cycle stages. For example, bug fix to production might use different dependencies from the current version under development.

All developers must configure Maven installations to point to these internal repositories instead of the external repositories, so that developers can use artifacts already stored in the internal repositories and reduce the download and build time. This also helps to ensure that all developers use the same version for various dependencies.

Archiva also provides the ability to manage expiration of artifacts from your snapshot repository. Each time that you execute a build, artifacts are created and stored in the snapshot repository. If you are using continuous integration, you may want to execute builds several times each day. The best practice is to configure Archiva to remove these stored artifacts after a certain amount of time (for example, one week). Alternatively, you can configure Archiva to keep just the last  $n$  versions of each artifact. Either approach helps to automatically manage the expiration and deletion of your snapshot artifacts.

## 1.5 Continuous Integration with Hudson

Hudson is a common continuous integration server product that enables you to automate the build process. Typically this automation include steps such as:

- Initiating a build whenever a developer commits to the version control system
- Checking out the code from the version control system
- Compiling the code
- Running unit tests and collating results (often through JUnit)
- Packaging the code into a deployment archive format
- Deploying the package to a runtime environment
- Running integration tests and collating results
- Triggering the build to the Maven snapshot repository
- Alerting developers through email of any problems

However, it is also possible to use the build system to enforce compliance with corporate standards and best practices. For example, enterprises can include the following steps in the build process:

- Running code coverage checks to ensure that an appropriate number of unit tests exist and are executed
- Running code quality checks to look for common problems
- Running checks to ensure compliance with naming conventions, namespaces, and so on
- Running checks to ensure that documentation is included in the code
- Running checks to ensure that the approved versions of dependencies are used and that no other dependencies are introduced without approval

Hudson provides a web-based console that enables build managers to define, execute, and monitor builds. Builds are executed on one or more build servers. The number of build servers is typically defined based on the volume of builds and the expected time for a build to complete. Hudson also provides APIs and can be extended through a plug-in mechanism, so that additional functionality can be added, as needed.

## 1.6 Summary

This guide describes how to establish a continuous integration environment that supports a large team of developers who develop applications on the Oracle Fusion Middleware 12c platform. This environment includes version control, Maven for build automation and dependency management, Archiva as a Maven repository, and the use of a continuous integration server like Hudson to automate the build process.

All examples in this book are Apache tools: Subversion, Maven, Archiva, and Hudson. However, there are other commercial and open source alternatives that you can use. The intention here is to provide an example that you can refer to and that should be easy enough to adapt to other tools. For example, you may choose to use git for version control or Nexus as your repository manager. The choice of tools in this documentation does not imply that other tools will not deliver equivalent outcome.

---



---

## Roadmap for Continuous Integration

Oracle Fusion Middleware 12c introduces new capabilities for build automation and continuous integration. Continuous integration is both a journey and a destination. If you have not automated your build process before, then you may find yourself at the beginning of the journey. This chapter provides a roadmap to help you to understand the steps you need to take to attain continuous integration.

If you are familiar with build automation or continuous integration already, this chapter provides a summary of the features provided in Oracle Fusion Middleware 12c and helps you relate it with your existing experiences.

This chapter also describes a reference continuous integration environment. This is provided as an example to help you to visualize what your environment may look like after adopting the continuous integration approach and the tools and technologies described in this book.

This chapter contains the following sections:

- [Section 2.1, "Roadmap"](#)
- [Section 2.2, "Overview of the Reference Continuous Integration Environment"](#)
- [Section 2.3, "Shared Disk Layout"](#)

### 2.1 Roadmap

[Table 2–1](#) describes the common steps that you must take to implement continuous integration, and also provides pointers for more information in each step.

**Table 2–1 Roadmap to Attain Continuous Integration**

No.	Task	For More Information
1.	Implement a version control strategy	<p><a href="#">Chapter 3</a> provides details on how to set up a version control environment using Subversion.</p> <p>If you are not currently using version control in your development environment, then you should pay particular attention to repository layout, the Subversion workflow, and tagging and branching, which are described in <a href="#">Chapter 3</a>.</p> <p>For more information about Version Control with Subversion, see:</p> <p><a href="http://svnbook.red-bean.com">http://svnbook.red-bean.com</a></p>

**Table 2–1 (Cont.) Roadmap to Attain Continuous Integration**

No.	Task	For More Information
2.	Implement a binary repository strategy	<p><a href="#">Chapter 4</a> provides details on how to use Apache Archiva to set up a repository for binary artifacts, both those that you are building and those that your builds depend on. You should pay particular attention to understanding the need for multiple repositories, understanding the difference between snapshot and other repositories, and the administration and maintenance required for repositories.</p>
3.	Implement a build automation and dependency management strategy	<p><a href="#">Chapter 5</a> provides a brief introduction to Maven and the installation and configuration steps. If you have never used Maven before, you should do some reading to get familiar with it. Start by reading "What is Maven?" at the official Maven web site at: <a href="http://maven.apache.org/what-is-maven.html">http://maven.apache.org/what-is-maven.html</a></p> <p>There are also a number of resources available online, including:</p> <ul style="list-style-type: none"> <li>■ <i>Maven By Example</i> and <i>Maven: The Complete Reference</i> at: <a href="http://www.sonatype.org/maven">http://www.sonatype.org/maven</a></li> <li>■ <i>Better Builds with Maven</i> at: <a href="http://www.maestrodev.com/better-builds-with-maven/about-this-guide/">http://www.maestrodev.com/better-builds-with-maven/about-this-guide/</a></li> </ul> <p>These resources will help you to form a comprehensive understanding on how to use Maven, some of the principles behind its design, and the kind of things you can do with Maven.</p>
4.	Populate your repository with Oracle artifacts	<p><a href="#">Section 5.3</a> provides details on how to populate your Maven Repository with Oracle-provided artifacts. You should pay particular attention to understanding the Oracle Maven synchronization plug-in, what happens when you apply patches to your Oracle runtime environments, the implications of patching on your build environment, and understanding the role of archetypes.</p>
5.	Ensure that you understand Maven version numbers	<p><a href="#">Chapter 7</a> provides details on the important nuances of Maven version numbers. Ensure that you understand how to use Maven version numbers and version number ranges to specify dependencies, and how Maven resolves dependencies based on the way you specify version numbers.</p>
6.	Learn how to build Java EE applications for WebLogic Server using Maven	<p><a href="#">Chapter 9</a> provides details on how to create Java EE applications using the Oracle WebLogic Maven archetypes, and how to compile, package, and deploy your applications to a WebLogic Server.</p>
7.	Learn how to build Coherence applications using Maven	<p><a href="#">Chapter 10</a> provides details on how to create Coherence GAR applications using the Oracle Coherence Maven archetypes, and how to compile, package, and deploy your applications to the Coherence container on WebLogic Server.</p>



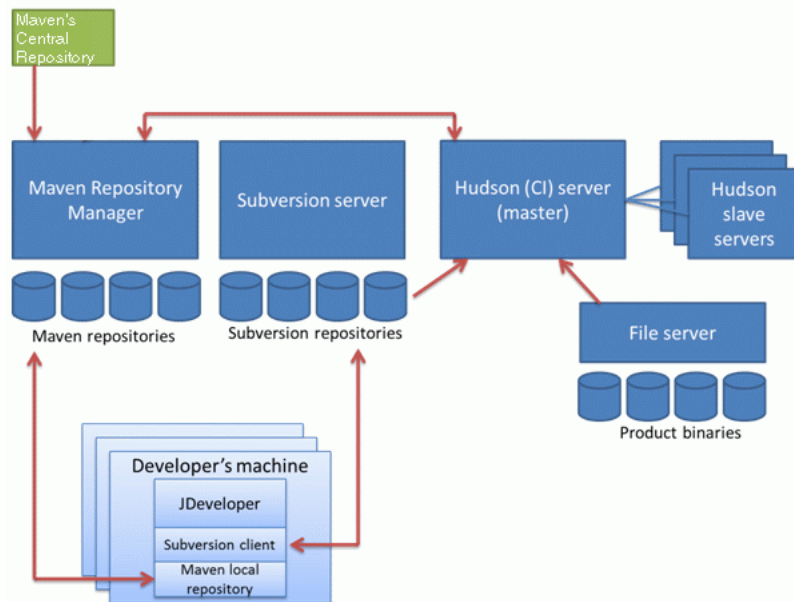
**Table 2–1 (Cont.) Roadmap to Attain Continuous Integration**

No.	Task	For More Information
8.	Learn how to build a whole application using Maven	<a href="#">Chapter 11</a> describes how to bring together many of these concepts to build a more realistic application. The example application has multiple component parts, each targeted to a different Oracle Fusion Middleware runtime environment. It also has dependencies between components, and some custom packaging requirements.
9.	Learn how to customize your build process using Maven POM inheritance	<a href="#">Chapter 8</a> provides details on the Maven POM hierarchy that is included with Oracle Fusion Middleware 12c. The common Oracle 'parent POMs provide an easy way to customize your build process.
10.	Implement a continuous integration strategy	<a href="#">Chapter 6</a> describes how to set up Hudson to create an environment in which to perform continuous integration. <a href="#">Chapter 10</a> expands on this with details of the important considerations for establishing and operating a continuous integration environment.

## 2.2 Overview of the Reference Continuous Integration Environment

This chapter provides a reference implementation of a continuous integration environment based on Subversion, Maven, Hudson, and Archiva.

[Figure 2–1](#) provides an overview of the recommended development environment. The dark blue components (shown in the middle) make up the shared (server) portion of the development environment.

**Figure 2–1 Reference Continuous Integration Environment Architecture**

The following is a description of the environment:

- The developer's machine:** Each developer has a workstation on which to run an integrated development environment (Oracle JDeveloper) to create source artifacts like Java code, deployment descriptors, BPEL processes, and ADF user interface

projects. JDeveloper includes a Subversion client. This enables the developer to perform actions like checking code in and out of repositories on the Subversion server, checking differences, performing merges, and resolving conflicts. Each developer's machine also has a local Maven repository on their workstation, which holds dependencies that are needed to perform local builds that they may want to perform on their workstation. For example, a developer might want to check a build and run unit tests locally before checking in code to the shared Subversion server.

- **The Maven Repository Manager:** This part of the development environment is the repository for all built artifacts and dependencies, both for those created in your environment and those sourced externally. Typically, there are different repositories for different purposes:
  - The Binary Repository Manager acts as a mirror or proxy for external Maven repositories like Maven's central repository. When artifacts from these repositories are needed for a build, they are downloaded and stored in a repository managed by the Maven Repository Manager.
  - When you build an artifact (for example, a WAR, JAR, or SAR file), it is published into a repository managed by the Maven Repository Manager. Often there are separate repositories for SNAPSHOT (work in progress) artifacts and release (final) artifacts.
- **The Subversion server:** The repository for source artifacts that are created by your developers. Typically there are multiple repositories. For example, there may be one per project.
- **The Hudson continuous integration (parent) server:** The server that manages your continuous integration builds. It is responsible for running builds and collecting and reporting results of those builds.
- **Hudson child servers:** Optional additional Hudson servers that are used to provide additional capacity. If you are running a large number of builds, you can set up several Hudson child servers to share or perform some of the builds.
- **File server:** A storage area network (SAN) or network-attached storage (NAS) that hosts copies of any product binaries that are needed by the Hudson build servers. For example, **ojdeploy** is required to build Oracle ADF applications. All of the Hudson servers have access to this file server.
- **Test servers** (not shown): If you are performing integration tests, you may also have a set of test servers in which you deploy built artifacts like WARs, JARs, and SARs, and execute your integration tests.
- **Maven's Central Repository:** This is an external Maven repository that holds open source dependencies, which you may need for a build. The Maven Repository Manager is able to search these repositories for any dependencies that it does not have in its own repository, and obtain them if they are available.

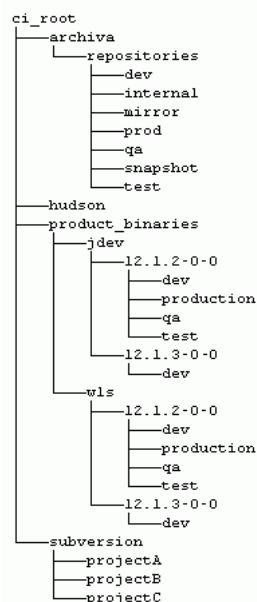
Depending on the size of your environment, these components (excluding the external ones and the developer's machine), might be on a single server, or spread across several machines.

## 2.3 Shared Disk Layout

If you plan to use a shared disk, consider keeping all of your Subversion, Maven, Hudson data, and product binaries on the shared disk.

A suggested directory structure is shown in [Figure 2–2](#). This structure shows only the high-level directories that you would consciously consider creating.

**Figure 2–2** Directory Structure for a Shared Disk



Note that for the product binaries, you must keep one copy for each environment (such as production, development, QA). Although the product binaries are on the same version of the software, they are likely to have different patches installed. Ensure that you can always build using the same set of artifacts, possibly patched, as the environment to which you want to deploy.

Although you may have moved your development environment up to 12.1.3 in the example in [Figure 2–2](#), you still must be able to build against 12.1.2. If you find a bug in production or QA, you must be able to build using the same versions of artifacts as you have installed in those environments.

Your Maven Repository Manager, Archiva in this case, includes the following repositories:

- **Internal:** Stores finished artifacts that you have built in your development environment.
- **Snapshot:** Stores work in progress artifacts that you have built in your development environment.
- **Mirror:** Stores dependencies that have been downloaded from an external repository.
- **Dev, test, qa, prod:** You have one repository for storing the dependencies needed for each target environment. You do this because it is possible that two environments might have the same version of an artifact (for example, 12.1.3-0-0) even though the artifact has been patched in one environment, and is therefore different. See [Section 5.3.7](#) to learn more about this requirement.

The shared disk server must provide sufficient space for product binaries, Subversion repositories, the Archiva repository, Maven binaries, Hudson binaries, configuration, and file storage. At a minimum, Oracle recommends that you allocate at least 40 gigabytes. Factors, such as Archiva snapshot clean up rules and whether or not you

permit check-in of binaries into the source control system, can increase the required space.

---

---

# Installing and Configuring Subversion for Version Control

Subversion is a version control system that keeps track of changes made to files and folders or directories, thus facilitating data recovery and providing a history of the changes that have been made over time. This chapter describes how to install and configure Subversion for version control.

This chapter contains the following sections:

- [Section 3.1, "Downloading Subversion"](#)
- [Section 3.2, "Installing Subversion"](#)
- [Section 3.3, "Configuring the Subversion Server as a Service"](#)
- [Section 3.4, "Setting Up a Repository"](#)
- [Section 3.5, "Understanding SVN Workflow"](#)
- [Section 3.6, "Considerations for Tagging and Branching"](#)
- [Section 3.7, "Subversion Clients"](#)
- [Section 3.8, "More Information"](#)

## 3.1 Downloading Subversion

Although Subversion is an Apache project, Apache does not build their own binary files for any operating system. The following URL provides URLs about the latest stable releases of Subversion built by third parties for all major operating systems:

<http://subversion.apache.org/packages.html>

If possible, use a package manager such as YUM or APT to manage the installation of other software.

On a Windows operating system, Oracle recommends that you use a precompiled binary package such as Silk SVN which is available in the following URL:

<http://www.silksvn.com/en/download>

On Windows, if you install Subversion through the installer package, then ensure that you choose an installer which includes the server binary files.

## 3.2 Installing Subversion

The installation method varies depending upon the platform and distribution method.

For example, if you use YUM, the command is likely to be:

```
sudo yum install subversion
```

On Windows, you can change the default installation path to a shorter location.

```
C:\svn
```

Ensure that the `PATH` variable is correctly set by the installer.

To obtain the version information of `svnserve`, run the following command on the command line:

```
svnserve --version
```

If you cannot find the command, then do the following:

1. Open **Control Panel**.
2. Select **System**, and then **Advanced System Settings**.
3. Under **Advanced**, select **Environment Variables**.
4. Edit the `PATH` variable in the **System variables** pane by adding the path to the Subversion binary directory.

### 3.3 Configuring the Subversion Server as a Service

To configure the Subversion server as a service:

- **On Linux**

The Linux installation process automatically creates an `/etc/init.d/svnserve` script. This starts the server when you start up your system.

To start the service manually, run the following command on the command line:

```
sudo /etc/init.d/svnserve start
```

- **On Windows**

You must register `svnserve` with the service manager. To register `svnserve`, run the following command:

```
sc create svnserver binpath= "C:\svn\svnserve.exe" --service -r "REPOS_PATH"  
displayname="Subversion" depend=Tcpip start=auto
```

In the preceding command, `REPOS_PATH` is the absolute path to the local file system.

### 3.4 Setting Up a Repository

A Subversion repository is a collection of versioned artifacts on the Subversion server.

This section contains the following topics:

- [Creating a Repository](#)
- [Subversion Layout](#)
- [Importing Existing Projects](#)

#### 3.4.1 Creating a Repository

After Subversion is installed, you must create a repository. The command-line utility called `svnadmin` is the primary tool for server-side administrative operations.

- **On Linux**

To create a repository:

1. Create a directory for the repository by running the following command:

```
mkdir -p REPOS_PATH
```

In this command, *REPOS\_PATH* is the absolute path to the local file system.

For example:

```
mkdir -p /ciroot/subversion/repository
```

2. Create a repository on a given path by running the following command:

```
svnadmin create REPOS_PATH
```

In this command *REPOS\_PATH* is the absolute path to the local file system.

For example:

```
svnadmin create /ciroot/subversion/repository
```

- **On Windows**

To create a repository:

1. Create a directory for the repository by running the following command:

```
mkdir REPOS_PATH
```

In this command, *REPOS\_PATH* is the absolute path to the local file system.

For example:

```
mkdir C:\ciroot\subversion\repository
```

2. Create a repository on a given path by running the following command:

```
svnadmin create REPOS_PATH
```

In this command, *REPOS\_PATH* is the absolute path to the local file system.

For example:

```
svnadmin create C:\ciroot\subversion\repository
```

Access to the repository is controlled by file permissions and the user referenced for accessing the repository through the SVN client. Ensure that user and group permissions for all files in the new repository reflect the type of access control that you want to have over the repository contents.

By default, anonymous, read-only access is enabled for a new repository. This means that anyone with SSH access, regardless of repository permissions settings, can check out repository files. You can modify this in the *REPOS\_PATH/conf/svnserve.conf* file.

Now that you have created a repository, you can use the Subversion client to perform standard operations against the new repository by using the following base URL:

```
svn+ssh://USER@HOST/REPOS_PATH
```

For example:

```
svn ls svn+ssh://mycompany@localhost/ciroot/subversion/repository
```

In addition to `svn+ssh`, there are several other protocols that are supported by Subversion. Refer to the Subversion documentation for information on how to configure other protocols. `svn+ssh` might not be available on Windows by default.

## 3.4.2 Subversion Layout

Although Subversion does not require any particular subdirectory structure within a repository, it is a good idea to follow an established convention, as this book does. The typical repository layout should resemble the following figure:

```
root
-- projectA
    -- subprojectA1
        -- trunk
        -- tags
        -- branches
    -- subprojectA2
        -- trunk
        -- tags
        -- branches
-- projectB
(etc.)
```

Development of the main code line occurs in the `trunk` directories. When a release is made, the current trunk source is copied into the `tags` directory, to a tag corresponding to the release. Subversion copy operations are not expensive in terms of storage because the server tracks changes internally.

The following is an example of a tag:

```
my-project/tags/3.0.5
```

In the preceding example, `3.0.5` indicates the release version to which this tag corresponds to.

A tag is important for future work that might be necessary for patch creation or bug-fix releases. Another importance of a release tag is to facilitate investigation regarding issues in the associated release.

If a patch or subsequent change of a tag is considered necessary, then you must create a branch. A branch is a copy of a location elsewhere in the repository and does not differ in composition from a tag. After a copy of the tag is made under the `branches` directory, you can check out the code and modify it as necessary. When changes are complete, the new release is made from the branch and a corresponding tag is created.

This Project-A example outlines the general workflow for patch management of source code:

In Project-A, the main code line is managed under `project-A/trunk`. The current version developing under the `trunk` directory is version 2.1. The three previous releases of Project-A are 1.0, 1.1, and 2.0. A problem is discovered in version 1.0 that requires a patch release.

To address the problem, the `project-A/tags/1.0` tag is copied, using the `svn copy` command, to the `project-A/branches/1.0.1-SNAPSHOT`. The `SNAPSHOT` designation is a Maven device indicating a version that is not yet released, as shown in the following figure.



When the branch code fix is complete, the branch is copied from `project-A/branches/1.0.1-SNAPSHOT` to a `project-A/tags/1.0.1` tag. The release build can then be made from the tag.

```
root
  -- project-A
    -- trunk
    -- tags
      -- 1.0
      -- 1.1
      -- 2.0
    -- branches
      -- 1.0.1-SNAPSHOT
  -- projectB
    -- trunk
    -- tags
      -- 1.0
    -- branches
```

For more information on directory structure conventions, see "chapter 2" of *Version Control with Subversion* in the following URL:

<http://svnbook.red-bean.com/>

### 3.4.3 Importing Existing Projects

If you have existing projects that you want to manage in your repository, you can import them using the SVN client's `import` command:

```
svn import LOCAL_PATH REPOSITORY_URL/REPOSITORY_PATH
```

For example:

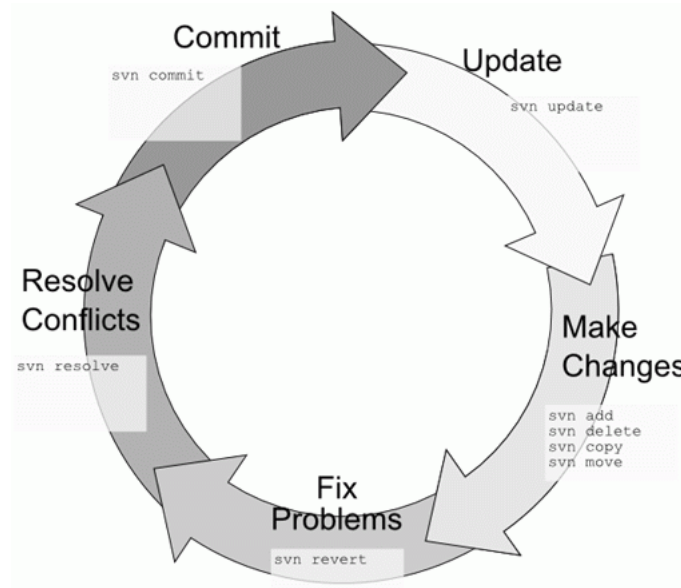
```
svn import /checkouts/project-a
svn+ssh://user@svn.mycompany.com/ciroot/subversion/repository/project-a/trunk/ -m
"initial import"
```

## 3.5 Understanding SVN Workflow

To modify code, you usually perform the following operations:

1. Update the working copy using the `svn update` command.
2. Make changes. Use the `svn add`, `svn delete`, `svn copy`, and `svn move` commands as needed to edit your files.
3. Review changes through the `svn status` and `svn diff` commands.
4. Fix mistakes. You can revert and abandon changes using the `svn revert` command.
5. Resolve conflicts. When they are resolved, mark them using the `svn resolve` command.
6. Commit changes using the `svn commit` or `svn ci` command.

Figure 3–1 shows the complete life cycle of an SVN operation:

**Figure 3–1 SVN Workflow**

In a continuous integration development process, this workflow remains largely unchanged. Committed change sets tend to be smaller and occur more frequently than in a noncontinuous integration process. You must commit the active trunk or branch code for the target release so that the continuous integration system can perform an integration build. Avoid creating a personal branch, with the intention of merging back to the main-line code base in the future. The personal branch and merge technique defers integration and runs counter to continuous integration precepts.

To begin working on a Subversion managed project, you must first check out the files into your local file system. The SVN client copies the project files to your system, including Subversion metadata in `.svn` directories located in each subdirectory. Run the following command to check out files:

```
svn co REPOSITORY_URL/REPOSITORY_PATH LOCAL_DIRECTORY
```

In the preceding command:

- `REPOSITORY_URL` is the URL to the Subversion repository.
- `REPOSITORY_PATH` is the path to the directory being checked out.
- `LOCAL_DIRECTORY` is the path to the local directory in which the checked out project is stored.

The `test-project` example demonstrates main-line code development on a project:

```
svn checkout
  svn+ssh://user@svn.mycompany.com/subversion/repository/test-project/trunk
test-project
```

In this case, a directory called `test-project` is created and the project contents are recursively copied into the directory from the server.

You can make any number of changes to the checked out files. When you are ready to commit the changes to the repository, check in the files or directories that you want to commit. The file or directory set being checked in does not have to correspond to what was checked out as long as all components are members of the checked out directory. Run the following commands to commit the changes:

```
svn commit -m "Added code and test case" test-project/src/main/java
test-project/src/test/resources/testdata.xml
```

If changes are made in the period between the checkout or last update, and the commit, then the operation fails and a message detailing the reason is shown. You must resolve the conflict and update the system to mark the conflict as resolved. To do so, fix the conflict and run the following command:

```
svn resolve test-project/src/test/resources/testdata.xml
```

After you have resolved any conflict, proceed with a normal check-in operation.

After the project is checked out once on your system, there is no need to perform subsequent checkouts on that source code. To stay synchronized with the Subversion repository content, you can run the `svn update` command on a checked out directory or even on individual files.

Before committing local changes to the repository operation, run `svn update` to integrate any changes committed to the code by others, since your last checkout or update, by running the following command:

```
svn update
```

Finally, commit your changes by running the following command:

```
svn commit -m "description of the updates"
```

## 3.6 Considerations for Tagging and Branching

Tagging creates a named point-in-time copy of a branch. Tagging should be done on two occasions:

- Whenever a project is released
- Whenever an important milestone occurs

It is important to tag releases, as tags provide a simple mechanism for patching releases. When a bug is found in a release, you can branch from the tag for that release, implement the fix, and then create a patch for the release. Tag this new (patched) release as well, in case you find an issue with it later and need to fix that new issue.

If you do not tag a release, then it is very difficult to obtain the exact code line that was built into that release.

---

---

**Note:** Treat tagged releases as read-only artifacts. You must not continue merging into a release after it is tagged.

---

---

## 3.7 Subversion Clients

This section describes two popular Subversion clients:

- [WebSVN](#)
- [TortoiseSVN](#)

### 3.7.1 WebSVN

WebSVN provides a web-based view of a repository and supports visual differences, blame, and search.

WebSVN can be downloaded from the following location:

<http://www.websvn.info/>

### 3.7.2 TortoiseSVN

TortoiseSVN is a free Windows Subversion client that integrates with Windows Explorer. All standard Subversion client operations can be performed through the Windows user interface. Folder and file icon decorators indicate the status of Subversion files. Command-line tools are mapped with menu items and options are configurable through dialog boxes. Tortoise also provides sophisticated graphical diff and merge tools that can be helpful for resolving conflicts.

TortoiseSVN can be downloaded from the following location:

<http://tortoisesvn.net/>

## 3.8 More Information

This document is meant as a quick guide for starting and running Subversion. For a detailed guide, see *Version Control with Subversion* in the following location:

<http://svnbook.red-bean.com>

---

---

**Note:** Oracle strongly recommends reading *Version Control with Subversion* if you are new to Subversion.

---

---

---

---

# Installing and Configuring the Archiva Maven Repository Manager

This chapter describes the installation and basic configuration of Apache Archiva. Archiva is one of several choices for an artifact repository, an important component of a Maven-based continuous integration build system.

If you are not familiar with Maven Repository Managers or artifact repositories, see [Section 1.4](#) for more details.

After you have completed installation and configuration of Archiva (as detailed in this chapter) and Maven (as detailed in [Chapter 5](#)), you should populate your Archiva repository with the Oracle-provided artifacts. Refer to [Section 5.3](#) for more details.

This chapter contains the following sections:

- [Section 4.1, "Overview of Archiva"](#)
- [Section 4.2, "Downloading Archiva"](#)
- [Section 4.3, "Installing Archiva"](#)
- [Section 4.4, "Configuring Archiva"](#)
- [Section 4.5, "More Information"](#)
- [Section 4.6, "Maven Repository Manager Administration"](#)

## 4.1 Overview of Archiva

Archiva is distributed as a standalone installation that is bundled with Jetty. A WAR file distribution is also provided so that Archiva can be installed into an existing application server.

This chapter describes the process of installing the standalone version. Instructions for WAR file installation and configuration are available in the official Archiva documentation in the following location:

<http://archiva.apache.org/docs/1.3.6>

## 4.2 Downloading Archiva

You can download the latest standalone Archiva release either as a .zip file or tar.gz file from the following location:

<http://archiva.apache.org/download.html>

## 4.3 Installing Archiva

Unpack the distribution to the target installation directory. This location depends on your preference and target operating system. Oracle recommends that you create a common location for continuous integration related workspaces. For example, unpack the distribution in the following location:

```
/ciroot/archiva
```

- **On Linux**

Run the following command:

```
sudo mkdir -p /ciroot/archiva ; sudo tar xzvf apache-archiva-1.3.6-bin.tar.gz  
--strip-components 1 -C /ciroot/archiva
```

After you run the command, ensure that you change the owner of the files to match your user and group. For example, if you are using `oracle` as the user id and `oracle` as the group name, you would run the following command:

```
chown -R oracle:oracle /ciroot/archiva
```

- **On Windows**

Create a directory to create the Archiva installation files:

```
mkdir c:\ciroot\archiva
```

Extract the Archiva zip file that you downloaded into this new directory.

## 4.4 Configuring Archiva

This section provides details on how to configure Archiva, not just in general, but also some specific configuration for use in a Fusion Middleware environment.

This section contains the following topics:

- [Configuring the Server](#)
- [Starting the Server](#)
- [Creating an Administrator User](#)
- [Internal and Snapshot Repositories](#)
- [Proxy Repository](#)
- [Configuring Mirror Repositories](#)
- [Creating Development, Production, Quality Assurance, and Test Repositories](#)
- [Creating a Deployment Capable User](#)

### 4.4.1 Configuring the Server

The Archiva Jetty instance starts up with a default HTTP port of 8080. If you want to change this, before startup, modify `/ciroot/archiva/conf/jetty.xml`. Change the connector configuration's `SystemProperty` value for `jetty.port` to a different value, for example, 8081:

For example:

```
<Call name="addConnector">  
  <Arg>  
    <New class="org.mortbay.jetty.nio.SelectChannelConnector">
```

```

<Set name="host">
  <SystemProperty name="jetty.host"/>
</Set>
<Set name="port">
  <SystemProperty name="jetty.port" default="8081"/>
</Set>

```

## 4.4.2 Starting the Server

After the server is configured, you can start it from the command-line interface.

To start the server:

- Run the following command:

```
/ciroot/archiva/bin/archiva start
```

On 64-bit Linux systems, you may receive an error message similar to this:

```
./archiva: /ciroot/archiva/bin/./wrapper-linux-x86-32:
/lib/ld-linux.so.2: bad ELF interpreter: No such file or directory
```

If you receive this error, install the `glibc.i686` package (using `yum` for example) and try again.

- Check the log output while the server is starting, to ensure that it starts as expected, by running the following command on the command line:

```
tail -f /ciroot/archiva/logs/*
```

After the startup tasks complete, the Archiva server is available in the following location:

<http://localhost:8081/archiva>

## 4.4.3 Creating an Administrator User

When you visit the Archiva home page for the first time, you are prompted to set the administration password. Specify the full name, email address, and password of the administration user.

The screenshot shows a web browser window with the URL `localhost:8081/archiva/security/addadmin.action`. The page title is "Apache Archiva \ Create Admin ...". The main content area is titled "Create Admin User" and features the Archiva logo on the left. The form contains the following fields:

- Username: `admin`
- Full Name\*: `CI Admin User`
- Email Address\*: `adminuser@mycompany.org`
- Password\*: `*****`
- Confirm Password\*: `*****`

A "Create Admin" button is located at the bottom right of the form. There is also a search bar in the top right corner with the text "Search for:".

## 4.4.4 Internal and Snapshot Repositories

Archiva starts up with two hosted repositories configured:

- Internal

The internal repository is for maintaining fixed-version released artifacts deployed by your organization, which includes finished versions of artifacts, versions that

are no longer in development, and released versions. Note that in-development versions of the same artifacts may exist in the snapshot repository in the future.

- **Snapshot**

The snapshot repository holds the work-in-progress artifacts, which are denoted with a version with the suffix `SNAPSHOT`, and artifacts that have not yet been released.

#### 4.4.5 Proxy Repository

In addition to hosting your internally deployed artifacts, the internal repository is configured to serve as a proxy for the public Maven Central repository by default. A proxy repository can be used instead of directly referring to a third-party repository. A proxy caches previously requested artifacts locally. This reduces the load on public servers, which is recommended; especially if you run builds from a clean repository. If you place too much load on the public server, it may throttle or ban your host from placing additional requests. For significant build performance improvement, fetch dependencies from a less loaded, more proximate, proxy server.

If you require third party artifacts from other public repositories, then add them to your repository as additional Proxy Connectors.

#### 4.4.6 Configuring Mirror Repositories

Because you will typically want to share cached third-party proxied artifacts among multiple repositories, you should separate the cached artifacts from your project artifacts by moving them into a separate repository.

To create mirror repositories:

1. Remove the proxy connections from the internal repository.
  - a. Under the **Administration** menu, click **Proxy Connections**.
  - b. Delete the **Central Repository** and **maven2-repository.dev.java.net** proxy connectors by clicking the red **X** on each entry.
2. Add a new mirror repository.
  - a. From the **Administration** menu, click **Repositories**.
  - b. From the top right corner, click **Add**.
  - c. Specify the following information in the **Admin: Add Managed Repository** dialog box:
    - **Identifier:** mirror
    - **Name:** Mirror
    - **Directory:** `/ciroot/archiva/data/repositories/mirror`
    - Select **Releases Included**, **Block Re-deployment of Released Artifacts**, and **Scannable**.
  - d. Click **Add Repository**.



### Admin: Add Managed Repository

Identifier\*:   
 Name\*:   
 Directory\*:   
 Index Directory:   
 Type:   
 Cron\*:   
 Repository Purge By Days Older Than:   
 Repository Purge By Retention Count:   
 Releases Included  
 Block Re-deployment of Released Artifacts  
 Snapshots Included  
 Scannable  
 Delete Released Snapshots

3. Add proxy connectors to the mirror repository.
  - a. Under the **Administration** menu, click **Proxy Connections**.
  - b. Click **Add**.
  - c. Select **mirror** in Managed Repository.
  - d. Select **central** for Remote Repository.
  - e. Click **Add Proxy Connector**.

For configuring a mirror repository in a remote repository, complete steps 1-3. However, select **maven2-repository.dev.java.net** in step 1-b.

After completing these steps, you should see the following:

**Repository Proxy Connectors**

 **mirror**  
Mirror

**Proxy Connector** ⊕ ⏴ ⏵ ✖

 **central**  
Central Repository  
<http://repo1.maven.org/maven2>

[Settings](#)

**Proxy Connector** ⊕ ⏴ ⏵ ✖

 **maven2-repository.dev.java.net**  
Java.net Repository for Maven 2  
<http://download.java.net/maven/2/>

[Settings](#)

To configure the anonymous guest user to enable read privileges for the new repository:

1. Under the **Management** menu, click **User management**.
2. Click **Guest**.
3. Click **Edit Roles**.

4. Select the **Repository Observer** role next to **mirror**.
5. Click **Submit** to save your changes.

#### 4.4.7 Creating Development, Production, Quality Assurance, and Test Repositories

You must create a separate repository for each Oracle Fusion Middleware environment that you want to target with a Maven build. Oracle's support for one-off patching (see [Section 5.3.7](#)), means that it is possible that you could have two different environments (for example, production and test), which are at the same version but have some different files due to different one-off patches applied.

To ensure that your Maven builds are using the correct version of files, create, configure, and create a group Maven repository for each target environment.

1. Create a repository:
  - a. From the **Administration** menu, click **Repositories**.
  - b. To add a new repository, click **Add** from the top right corner.
  - c. From the **Admin: Add Managed Repository** dialog box, specify the following details:
    - **Identifier:** Provide an identifier, like dev, prod, qa, or test.
    - **Name:** Provide a name.
    - **Directory:** Add a directory path like `/ciroot/archiva/data/repositories/${IDENTIFIER}`, where `${IDENTIFIER}` matches the string that you provided in **Identifier**.
    - Deselect **Block Re-deployment of Released Artifacts**.
    - Select **Releases Included** and **Scannable**.

##### Admin: Add Managed Repository

Identifier\*:

Name\*:

Directory\*:

Index Directory:

Type:

Cron\*:

Repository Purge By Days Older Than:

Repository Purge By Retention Count:

Releases Included

Block Re-deployment of Released Artifacts

Snapshots Included

Scannable

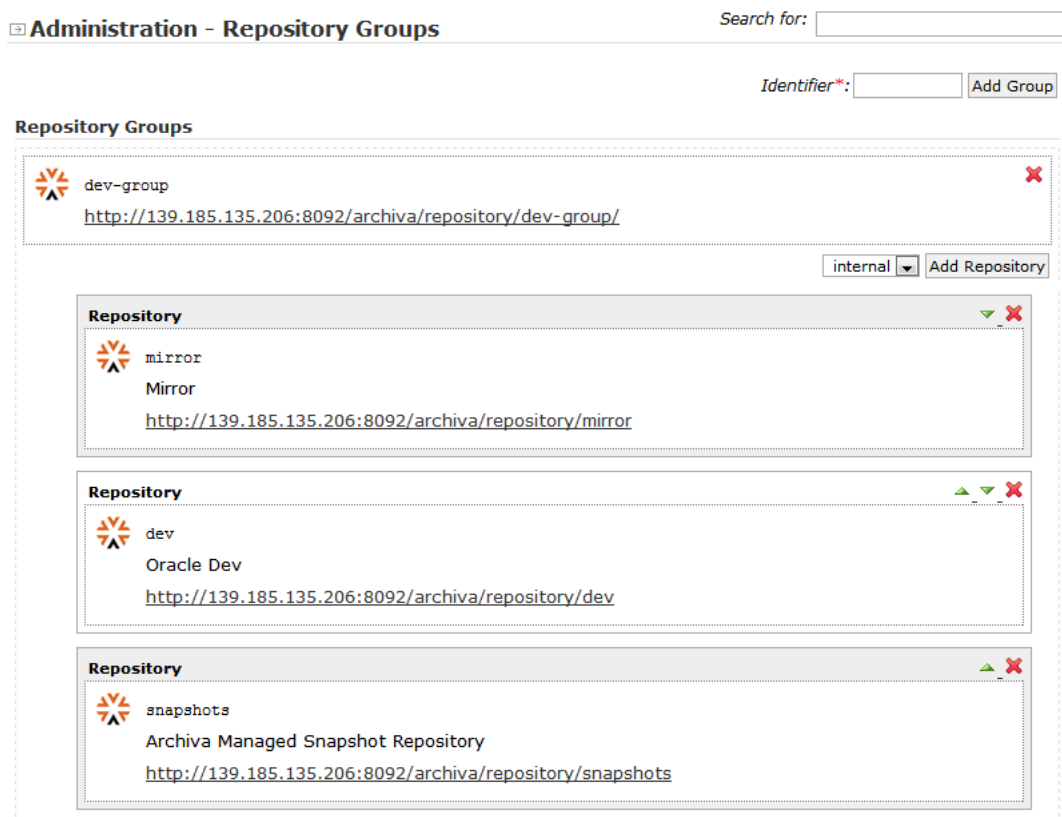
Delete Released Snapshots

- d. Click **Add Repository**.
2. To configure the anonymous guest user to have read privileges for the new repository:
  - a. Under **Manage**, click **User Management**.

- b. Select **guest**.
  - c. Select **Edit Roles**.
  - d. Select the **Repository Observer** role next to the appropriate repository entry.
  - e. Click **Submit** to save your changes.
3. To create a corresponding group for the new repository:
    - a. In the **Administration** menu, click **Repository Groups**.
    - b. In the top right corner, click **Add Group**.
    - c. In the **Identifier** field, specify a name that matches the repository that you created, with the addition of **-group**, for example, **dev-group**.
    - d. Click **Add Group**.
    - e. Select your new repository, like **dev**, from the drop-down menu next to **Add Repository** and click **Add Repository**.

Repeat steps 3-a to 3-d to add **mirror** and **snapshots**.

The following figure shows the Repository Groups page.



4. Repeat the repository and group creation steps 1-3 for each repository type: **test**, **qa**, and **prod**.

#### 4.4.8 Creating a Deployment Capable User

To support deployment in your internal repository, you must add at least one user with appropriate permissions:

1. Under **Management**, click **User Management**.

2. Click **Create New User** to add a user. Then, specify the required details like name, email address, and password. After the user is added, you are directed to a **Role Administration** dialog box for that user.
3. In the **Role Administration** dialog box, under **Resource Roles**, select the **Repository Manager** role for **Snapshot** and **Internal Repositories**.
4. Click **Submit** to save your changes.

---

---

**Note:** The Repository Manager role, while allowing you to upload artifacts, also allows you to change the repository configuration.

To customize or change the role, in the **User Roles** section, add a new more limited role and assign it to the appropriate users.

---

---

Typically, you want to create a new user for each individual with access to the repository. For Hudson, to publish build output to the repository, each user who accesses the repository should have their own user id, and you should create an additional user with deployment permissions.

After you have completed installation and configuration of Archiva (as detailed in this chapter) and Maven (as detailed in [Chapter 5](#)), you should populate your Archiva repository with the Oracle-provided artifacts, as described in [Section 5.3](#).

## 4.5 More Information

The user guide for Archiva 1.3.6 is available at:

<http://archiva.apache.org/docs/1.3.6/userguide/>

Other releases are available in the Archiva home page at:

<http://archiva.apache.org>

## 4.6 Maven Repository Manager Administration

This section contains the following topics:

- [Snapshot Cleanup](#)
- [Advanced User Management](#)
- [Backing Up Archiva](#)
- [Archiva Failover](#)

### 4.6.1 Snapshot Cleanup

Archiva retains an instance of a particular snapshot-versioned artifact for every successfully deployed job. When you request a snapshot artifact, the most recent snapshot is obtained. Maven examines the associated metadata in the repository to determine the correct copy to download. The Maven Repository Manager maintains each copy with a unique timestamp and build number.

For example, the contents of the repository directory for an artifact should look similar to the following:

```
maven-metadata.xml
test-artifact-2.1-20110928.112713-14.jar
test-artifact-2.1-20110928.112713-14.pom
```

```
test-artifact-2.1-20110924.121415-13.pom
test-artifact-2.1-20110924.121415-13.jar
```

The corresponding repository metadata should look similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
  <groupId>com.my.company</groupId>
  <artifactId>test-artifact</artifactId>
  <version>2.1-SNAPSHOT</version>
  <versioning>
    <snapshot>
      <timestamp>20110928.112713</timestamp>
      <buildNumber>14</buildNumber>
    </snapshot>
    <lastUpdated>20110928112718</lastUpdated>
    <snapshotVersions>
      <snapshotVersion>
        <extension>jar</extension>
        <value>2.1-20110928.112713-14</value>
        <updated>20110928112713</updated>
      </snapshotVersion>
      <snapshotVersion>
        <extension>pom</extension>
        <value>2.1-20110928.112713-14</value>
        <updated>20110928112713</updated>
      </snapshotVersion>
      <snapshotVersion>
        <extension>jar</extension>
        <value>2.1-20110924.121415-13</value>
        <updated>20110924121415</updated>
      </snapshotVersion>
      <snapshotVersion>
        <extension>pom</extension>
        <value>2.1-20110924.121415-13</value>
        <updated>20110924121415</updated>
      </snapshotVersion>
      ...
    </snapshotVersions>
  </versioning>
</metadata>
```

The `/metadata/versioning/snapshot` element contains the information for the latest snapshot that is fetched when you request the snapshot artifact for `test-artifact-2.1-SNAPSHOT`. You can directly request a specific snapshot of your requirement by referencing timestamp and build numbers in your version, for example, `2.1.-20110928.112713-14`.

Usually, only the latest snapshot is required for proper operation of continuous integration builds. Retention of older instances of a snapshot is helpful for troubleshooting purposes when the continuous integration server indicates that a snapshot dependency change has broken the integration process. It is sometimes useful to pull slightly older builds from the repository, after the last working build, to identify the problem.

If no recurring cleanup operation occurs, snapshot instances can accumulate quite rapidly over the lifetime of a project. To keep storage requirements of the repository manager under control, delete older snapshots. Set options regarding retention policy according to available storage and performance requirements.

#### 4.6.1.1 Retention Options

In a continuous integration environment, where builds are often triggered by checking in artifacts, there is the potential for a large number of builds to be executed. Each of these builds, at least the successful ones, results in some artifacts being published into the repository. These can start consuming a lot of space, and it is important to manage them.

Archiva provides two different options for automatically cleaning up old snapshots on a per-repository basis:

- Repository Purge by Number of Days Older  
Archiva automatically deletes snapshots older than the specified number of days. Archiva always retains the most recent snapshot, no matter how old it is.
- Repository Purge by Retention Count  
To use this method, you must set the `purge-by-days-older` value to 0. Archiva retains only the most recent snapshot instances up to this value. Older instances that exceed this count are deleted.

#### 4.6.1.2 Deleting Released Snapshots

Once the corresponding version is released, a snapshot of that version is no longer needed. Not only does this save space, but it also ensures that your dependency references are up-to-date.

Any existing continuous integration builds that refer to the snapshot fail with a missing dependency message after the dependency is deleted from the repository manager. This failure reminds you that a dependency reference is stale and encourages you to fix the problem.

### 4.6.2 Advanced User Management

Archiva uses Apache Redback for its user management infrastructure. To use Archiva's authentication and role management system with your organization's existing user management system, you must provide additional configuration with Redback. Redback has limited support for LDAP and other authentication systems.

Complete details are available in the following location:

<http://archiva.apache.org/redback/>

### 4.6.3 Backing Up Archiva

You should provide a mechanism for backing up your Archiva file store and configuration so that you can restore it if a file system failure or corruption occurs.

The choice of backup solutions may be affected by your failover method.

### 4.6.4 Archiva Failover

Although Archiva does not provide a failover solution, it is important for you to maintain a failover system that stays current. Depending on your preference, you can either set up an identically configured backup system with a separate file system that is synchronized with the primary systems or configure both systems to use the same shared file system.

For more information, see the Archiva page:

<https://cwiki.apache.org/ARCHIVA/high-availability-archiva.html>

---

# Installing and Configuring Maven for Build Automation and Dependency Management

Maven is a build management tool that is central to project build tasks such as compilation, packaging, and artifact management. Maven uses a strict XML-based rule set to promote consistency while maintaining flexibility. Because most Java-centric continuous integration systems integrate well with Maven, it is a good choice for an underlying build system. This chapter describes how to install and configure Maven.

This chapter contains the following sections:

- [Section 5.1, "Setting Up the Maven Distribution"](#)
- [Section 5.2, "Customizing Maven Settings"](#)
- [Section 5.3, "Populating the Maven Repository Manager"](#)

## 5.1 Setting Up the Maven Distribution

A distribution of Maven 3.0.4 is included with Oracle Fusion Middleware. After you install Oracle WebLogic Server, you can find Maven in your Oracle home in the following location:

```
ORACLE_HOME/oracle_common/modules/org.apache.maven_3.0.4
```

This is a copy of the standard Maven 3.0.4 release, without any modifications.

Alternatively, you can download and install your own copy of Maven from the Maven website:

<http://maven.apache.org>

Oracle Fusion Middleware supports Maven 3.0.4 or higher.

After installation, add Maven to your operating system's PATH environment variable:

- On UNIX:

You must update your shell startup script, your `.profile` or `.bash_profile`, to update the path.

For example, if you have installed Oracle WebLogic Server in `/u01/fmwhome` and you are using the bash shell, then you must add the following to the PATH environment variable:

```
export M2_HOME=/u01/fmwhome/oracle_common/modules/org.apache.maven_3.0.4
export PATH=${M2_HOME}/bin:$PATH
```

You also need to set the `JAVA_HOME` environment variable to point to your JDK installation. For example:

```
export JAVA_HOME=/u01/jdk1.7.0_21
```

- **On Windows:**

Edit your `PATH` environment variable and add the correct path to Maven at the beginning of the `PATH` environment variable.

For example, if you have installed WebLogic Server in `c:\fmwhome`, then you must add the following:

```
C:\fmwhome\oracle_common\modules\org.apache.maven_3.0.4\bin
```

You also need to set the `JAVA_HOME` environment variable to point to your JDK installation, for example, `export JAVA_HOME=/u01/jdk1.7.0_21`.

## 5.2 Customizing Maven Settings

You must create a Maven settings file if:

- You are working behind a firewall or proxy server.
- Your organization has its own internal Maven Repository Manager.

The Maven settings file is called `settings.xml` and is usually kept in the `.m2` directory inside your home directory. However, if you want to point Maven to a different location, see the Maven documentation.

If you have installed Maven for the first time, either as part of the Oracle WebLogic Server installation, or by downloading it from the Maven website, you will not have a settings file yet.

- **On UNIX:**

If your user name is `bob`, then the directory path should look similar to the following:

```
/home/bob/.m2/settings.xml
```

- **On Windows:**

If your user name is `bob`, then the directory path should look similar to the following:

```
C:\Users\bob\.m2\settings.xml
```

The following is an example of a Maven settings file:

```
<settings>
<proxies>
  <proxy>
    <active>true</active>
    <protocol>http</protocol>
    <host>proxy.mycompany.com</host>
    <port>8080</port>
    <nonProxyHosts>mycompany.com</nonProxyHosts>
  </proxy>
</proxies>
<servers>
  <server>
    <id>maven.mycompany.com</id>
    <username>me@mycompany.com</username>
```



```

        <password>{COQLCE6DU6GtcS5P=}</password>
    </server>
</servers>
<mirrors>
  <mirror>
    <id>archiva</id>
    <name>Internal Archiva Mirror of Central</name>
    <url>http://archiva.mycompany.com/repositories/internal</url>
    <mirrorOf>central</mirrorOf>
  </mirror>
</mirrors>
</settings>

```

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

- **Proxy** 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 internal Maven repository manager as a mirror (cache) of Maven's central repository.

If you are not familiar with these terms, review the introduction in [Chapter 1](#). For more information about available Maven settings, see the Maven documentation at:

<http://maven.apache.org/settings.html>

## 5.3 Populating the Maven Repository Manager

After you have configured your Maven Repository Manager, for example, you set up Archiva in [Chapter 4](#), you will want to populate it with Oracle artifacts.

For this reason, a Maven Synchronization plug-in is provided, which will allow you to populate a local or shared Maven repository from an Oracle home. When you install a Fusion Middleware 12c product, the Maven archetypes, plug-ins, and POMs are installed with the product so that the synchronization plug-in can find them.

This section contains the following topics:

- [Introduction to the Maven Synchronization Plug-In](#)
- [Installing Oracle Maven Synchronization Plug-In](#)
- [Running the Oracle Maven Synchronization Plug-In](#)
- [Things to Know About Replacing Artifacts](#)
- [Populating Your Maven Repository](#)
- [Running the Push Goal](#)
- [Things to Know About Patching](#)
- [Considerations for Archetype Catalogs](#)
- [Example settings.xml](#)
- [Deploying a Single Artifact](#)

### 5.3.1 Introduction to the Maven Synchronization Plug-In

Oracle Fusion Middleware 12c provides a **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, you should 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 same at the binary level, ensuring that all patched files reflects accurately in the Maven repository.

Oracle homes in 12c contains `maven` directories which contains Maven Project Object Models (POMs) for artifacts provided by Oracle, archetypes for creating projects, and Maven plug-ins provided by Oracle, for executing various build operations.

### 5.3.2 Installing 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 your shared internal repository, if you have one.

The plug-in is located in your Oracle WebLogic Server 12c home and consists of two files:

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

```
ORACLE_HOME/oracle_
common/plugins/maven/com/oracle/maven/oracle-maven-sync/12.1.2/oracle-maven-syn
c.12.1.2.pom
```

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

```
ORACLE_HOME/oracle_
common/plugins/maven/com/oracle/maven/oracle-maven-sync/12.1.2/oracle-maven-syn
c.12.1.2.jar
```

To install the plug-in into your local Maven repository, run the following command from the `ORACLE_COMMON/oracle_`  
`common/plugins/maven/com/oracle/maven/oracle-maven-sync/12.1.2` directory:

```
mvn install:install-file -DpomFile=oracle-maven-sync.12.1.2.pom
-Dfile=oracle-maven-sync.12.1.2.jar
```

The simplest way deploy the plug-in into a shared internal repository, use the web user interface provided by your Maven Repository Manager to upload the JAR file 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_COMMON/oracle_`  
`common/plugins/maven/com/oracle/maven/oracle-maven-sync/12.1.2` directory:

```
mvn deploy:deploy-file -DpomFile=oracle-maven-sync-12.1.2.pom
```

```
-Dfile=oracle-maven-sync-12.1.2.jar
-Durl=http://servername/archiva/repositories/internal -DrepositoryId=internal
```

To use the deploy plug-in as shown, you will need to define the repository in your Maven `settings.xml` file and also 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>

### 5.3.3 Running the Oracle Maven Synchronization Plug-In

The Oracle Maven Synchronization Plug-in supports two Maven goals:

- **Help**, which prints out help information
- **Push**, which is used to populate a repository

You can execute the **help** goal by running the following command:

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

The output of the **help** goal is as follows:

```
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building odm 1.0-SNAPSHOT
[INFO] -----
[INFO] --- oracle-maven-sync:12.1.2:help (default-cli) @ odm ---
[INFO] -----
[INFO] ORACLE MAVEN SYNCHRONIZATION PLUGIN - HELP
[INFO] -----
[INFO] The following goals are available:
[INFO]
[INFO] help           prints this help
[INFO]
[INFO] push           Install to the local repository and optionally deploy
[INFO]                to a remote repository from the specified oracle home
[INFO]
[INFO]                The plugin will use your current Maven settings to
[INFO]                determine the path to the local repository and,
[INFO]                optionally, a remote deployment repository. For
[INFO]                details on how to configure Maven's repository
[INFO]                settings, see the Maven settings reference:
[INFO]                http://maven.apache.org/settings.html
[INFO] parameters:
[INFO]   oracle-maven-sync.oracleHome
[INFO]     - This is the path to the Oracle Home
[INFO]   oracle-maven-sync.testOnly
[INFO]     - Must be set to 'false' or no action will be taken
[INFO]     Defaults to 'true'
[INFO]   oracle-maven-sync.failOnError
[INFO]     - If set to 'true' the plugin will stop and return an
[INFO]     error immediately upon the first failure to deploy
[INFO]     an artifact. Otherwise, the plugin will log the
[INFO]     error and attempt to complete deployment of all
[INFO]     other artifacts.
[INFO]     Defaults to 'false'
[INFO]   oracle-maven-sync.serverId [OPTIONAL]
[INFO]     - This is the ID of the server (repository) in
```

```

[INFO] your settings.xml file - where you have
[INFO] specified the remote Maven repository and its
[INFO] authentication information. The plugin will
[INFO] only install to the local repository if this
[INFO] parameter is not set.
[INFO]
[INFO] You can specify the parameters on the command line like this:
[INFO] -Doracle-maven-sync.serverId=archiva-internal
[INFO] -Doracle-maven-sync.testOnly=false
[INFO] -Doracle-maven-sync.failOnError=false
[INFO]
[INFO] To override the localRepository target used by the plugin, you
[INFO] can specify the following option on the command-line:
[INFO] -Dmaven.local.repo=/alternate/path/to/repository
[INFO]
[INFO] To supply an alternate settings.xml for purposes of this operation,
[INFO] use the --settings option. For example:
[INFO] mvn --settings /alternate/path/settings.xml ...
[INFO]
[INFO] ..or in your POM like this:
[INFO] <plugin>
[INFO]   <groupId>com.oracle.maven</groupId>
[INFO]   <artifactId>oracle-maven-sync</artifactId>
[INFO]   <version>12.1.2</version>
[INFO]   <configuration>
[INFO]     <oracleHome>/home/mark/Oracle/Middleware</oracleHome>
[INFO]     <failOnError>>false</failOnError>
[INFO]   </configuration>
[INFO] </plugin>
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 0.689s
[INFO] Finished at: Day Month Year Time Year
[INFO] Final Memory: 3M/119M
[INFO] -----

```

This output shows the parameters that are available for the plug-in's **push** goal. [Table 5–1](#) describes the parameters:

**Table 5–1 Push Goal Parameters and Description**

Parameter	Description
serverId	A pointer to 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 should provide the remote artifact repository's deployment information, such as URL, user name, and password.
oracleHome	The path to the Oracle home that you wish to populate the Maven repository from.
testingOnly	This controls whether the plug-in attempts to publish the artifacts to the repository.  If you test this to true, which is the default value, then the push goal will find all of your POM files and print out details of what it would have done if this is set to false. However, it does not publish any artifacts or make any change to the system.

**Table 5–1 (Cont.) Push Goal Parameters and Description**

Parameter	Description
failOnError	<p>If you set this property to <code>false</code> and the plug-in fails to process a resource, it continues to process all resources. Failures are logged as warnings, but the process completes successfully.</p> <p>If you set this property to <code>true</code>, when it encounters the first problem the plug-in will immediately exit with an error. This is the default.</p>

### 5.3.4 Things to Know About Replacing Artifacts

Some Maven Repository Managers have a setting that controls whether you will be able to 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. This screen is accessed by selecting **Repositories** under the Administration menu. Then, click **Edit** to change the setting of the repository you want to change.

If you are using Archiva, you must deselect the **Block Re-deployment of Released Artifacts** option in the **Managed Repository** settings.

Other Maven Repository Managers have similar settings. If you are using a different tool, consult the documentation for that tool to find out how to change this setting.

### 5.3.5 Populating Your Maven Repository

To populate your repository, you must use the **push** goal. You can specify the parameters given in [Table 5–1](#) on the command line or in your Project Object Model file.

This section contains the following topics:

- [Populating a Local Repository](#)
- [Populating a Remote Repository](#)

#### 5.3.5.1 Populating a Local Repository

If you are populating your local repository, you only need to specify `oracleHome` and `testingOnly=false`.

For example:

```
mvn com.oracle.maven:oracle-maven-sync:push
  -DoracleHome=/path/to/oracleHome
  --Doracle-maven-sync.testingOnly=false
```

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
  -Doracle-maven-sync.oracleHome=/path/to/oracleHome
  -Dmaven.repo.local=/alternate/path
```

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

```
<plugin>
  <groupId>com.oracle.maven</groupId>
  <artifactId>oracle-maven-sync</artifactId>
  <version>12.1.2</version>
  <configuration>
    <oracleHome>/path/to/oracleHome</oracleHome>
    <testOnly>>false</testOnly>
  </configuration>
</plugin>
```

After adding the plug-in, execute Maven by running the following command on the command line:

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

### 5.3.5.2 Populating a Remote Repository

If you are populating your 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` that matches the `serverId` you provide to the plug-in. If authentication is required for deployment, you must also add a server entry to your Maven `settings.xml`.

For example:

```
mvn com.oracle.maven:oracle-maven-sync:push
  -Doracle-maven-sync.oracleHome=/path/to/oracleHome
  -Doracle-maven-sync.serverId=internal
```

The corresponding Maven `settings.xml` with authentication details look like the following:

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

```

    <username>deployer</username>
    <password>welcome1</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:

[http://maven.apache.org/guides/mini/guide-encryption.html#How\\_to\\_encrypt\\_server\\_passwords](http://maven.apache.org/guides/mini/guide-encryption.html#How_to_encrypt_server_passwords)

---



---

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

```

<plugin>
  <groupId>com.oracle.maven</groupId>
  <artifactId>oracle-maven-sync</artifactId>
  <version>12.1.2</version>
  <configuration>
    <serverId>internal</serverId>
    <oracleHome>/path/to/oracleHome</oracleHome>
    <testOnly>>false</testOnly>
  </configuration>
</plugin>

```

After adding the plug-in, execute Maven by running the following command on the command line:

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

### 5.3.6 Running the Push Goal

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

- Checks the Oracle home you have provided and makes a list of all of the Maven artifacts inside that Oracle home. This is done by looking for Project Object Model 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 Project Object Model file is available in the Oracle home.
- Calculates a SHA1 checksum for the JAR file.
- Attempts to publish the JAR, Project Object Model, 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, `appc` and `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

### 5.3.7 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 Oracle Patch (OPatch) to manage application of patches to installed software in the Oracle home. When you use Oracle Patch 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 12.1.2 provides a mechanism to address this issue.

#### 5.3.7.1 Oracle's Approach to Patching

If any problems are found after a release of Oracle Fusion Middleware (for example, 12.1.2) into production, a one-off patch is created to fix that problem. Between any two releases, for example 12.1.2 and 12.1.3, 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 a great deal of flexibility and 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 arises when you have a number of environments, like test, QA, SIT, and production, which are likely to have different versions (or patches) installed.

#### 5.3.7.2 Maintain One Maven Repository for Each Environment

It is recommended that, in such a situation, you set up one Maven repository for each environment that you wish to target. For example, a Maven test repository that contains artifacts that matches 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.

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

After you patch your Oracle home, you should 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.

### 5.3.8 Considerations for Archetype Catalogs

By running the Oracle Maven Synchronization Plug-in's `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 archetype catalog, execute a command like the following:

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

### 5.3.9 Example settings.xml

This example, `settings.xml`, provides a template for Maven integration with the rest of the continuous integration system described in this book. It provides configuration to support central Archiva repository interaction and Hudson continuous integration server integration as described in [Chapter 4](#) and [Chapter 12](#). You must change values of URLs, passwords, and so on to match your system's values:

```
<settings>
  <profiles>
    <profile>
      <id>default</id>
      <repositories>
        <repository>
          <id>dev-group</id>
          <name>Dev Group</name>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>false</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
          </snapshots>
          <url>http://SERVER:PORT/archiva/repository/dev-group</url>
          <layout>default</layout>
        </repository>
        <repository>
          <id>dev</id>
          <name>Dev</name>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>false</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
          </snapshots>
          <url>http://SERVER:PORT/archiva/repository/dev</url>
          <layout>default</layout>
        </repository>
        <repository>
          <id>prod-group</id>
          <name>Prod Group</name>
          <releases>
            <enabled>true</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
```

```

        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>http://SERVER:PORT/archiva/repository/prod-group</url>
    <layout>default</layout>
</repository>
<repository>
    <id>prod</id>
    <name>Prod</name>
    <releases>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>http://SERVER:PORT/archiva/repository/prod</url>
    <layout>default</layout>
</repository>
<repository>
    <id>qa-group</id>
    <name>QA Group</name>
    <releases>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>http://SERVER:PORT/archiva/repository/qa-group</url>
    <layout>default</layout>
</repository>
<repository>
    <id>qa</id>
    <name>QA</name>
    <releases>
        <enabled>true</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
        <enabled>false</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>http://SERVER:PORT/archiva/repository/qa</url>
    <layout>default</layout>
</repository>
<repository>
    <id>test-group</id>
    <name>Test Group</name>
    <releases>
        <enabled>true</enabled>

```

```

        <updatePolicy>always</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
        <enabled>>false</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>http://SERVER:PORT/archiva/repository/test-group</url>
    <layout>default</layout>
</repository>
<repository>
    <id>test</id>
    <name>Test</name>
    <releases>
        <enabled>>true</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
        <enabled>>false</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>http://SERVER:PORT/archiva/repository/test</url>
    <layout>default</layout>
</repository>
<repository>
    <id>archiva-snapshots</id>
    <name>Archiva Snapshots</name>
    <releases>
        <enabled>>false</enabled>
        <updatePolicy>always</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
    </releases>
    <snapshots>
        <enabled>>true</enabled>
        <updatePolicy>never</updatePolicy>
        <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <url>http://SERVER:PORT/archiva/repository/snapshots</url>
    <layout>default</layout>
</repository>
</repositories>
</profile>
</profiles>
<servers>
    <server>
        <id>dev</id>
        <username>hudson</username>
        <password>PASSWORD</password>
    </server>
    <server>
        <id>dev-group</id>
        <username>hudson</username>
        <password>PASSWORD</password>
    </server>
    <server>
        <id>archiva-snapshots</id>
        <username>hudson</username>

```

```
        <password>PASSWORD</password>
    </server>
</servers>
<mirrors>
  <mirror>
    <id>dev-mirror</id>
    <name>All else</name>
    <url>http://SERVER:PORT/archiva/repository/dev-group</url>
    <mirrorOf>*</mirrorOf>
  </mirror>
</mirrors>
<activeProfiles>
  <activeProfile>default</activeProfile>
</activeProfiles>
</settings>
```

### 5.3.10 Deploying a Single Artifact

The Maven deploy plug-in can also be used to deploy an artifact (or artifacts) and Project Object Module to the remote artifact repository.

For example, run the following command to deploy to the `archiva-releases` repository, as defined in the sample `settings.xml` file.

```
mvn deploy:deploy-file
-Dfile=/path/to/oracle-maven-sync-12.1.2.jar
-DrepositoryId=archiva-releases
-DpomFile=/path/to/oracle-maven-sync-12.1.2.pom
-Durl=http://server:port/archiva/repository/internal
```

---

# Installing and Configuring Hudson for Continuous Integration

Hudson is a popular continuous integration server product. It enables you to define build jobs and manages the execution of those jobs for you. If necessary, it has the ability to scale up to a farm of build servers.

This chapter describes how to install and configure Hudson to automate the build process and how to integrate Hudson with Maven.

This chapter contains the following sections:

- [Section 6.1, "Prerequisites for Installing and Configuring Hudson"](#)
- [Section 6.2, "Downloading Hudson"](#)
- [Section 6.3, "Installing Hudson"](#)
- [Section 6.4, "Configuring the HTTP Port"](#)
- [Section 6.5, "Starting Hudson"](#)
- [Section 6.6, "Configuring Maven After Startup"](#)
- [Section 6.7, "For More Information"](#)

## 6.1 Prerequisites for Installing and Configuring Hudson

Ensure that you have the following components of the continuous integration system configured before you begin installing:

- Subversion server configured and running, as directed in [Chapter 3](#).
- Archiva configured and running as directed in [Chapter 4](#), which implies that you have an Oracle Fusion Middleware product installed in an Oracle home and have run the Oracle Maven synchronization plug-in to populate Archiva.
- JDK 1.6 or higher installed on the Hudson host.
- Maven 3 installed on the Hudson host.

## 6.2 Downloading Hudson

The latest production version of Hudson can be downloaded directly from the following location:

<http://hudson-ci.org/>

Hudson is distributed in two distinct versions:

- WAR file, which can either run as standalone or can be added to an existing application server installation.
- Linux RPMs compiled for specific operating systems. Package management support in the form of appropriate repositories are available to install the RPM and necessary dependencies.

This document focuses on Oracle Linux and Windows installations. Details for the other operating systems may vary from these. For instructions on various types of installation, see:

<http://wiki.hudson-ci.org/display/HUDSON/Installing+Hudson>.

## 6.3 Installing Hudson

This section contains the following topics:

- [Installing Hudson on Linux](#)
- [Installing Hudson on Windows](#)

### 6.3.1 Installing Hudson on Linux

On a Linux computer supporting YUM, run the following commands on the command line:

```
sudo wget -O /etc/yum.repos.d/hudson.repo http://hudson-ci.org/redhat/hudson.repo
sudo yum check-update
sudo yum install hudson
```

This installs Hudson as a daemon and creates a Hudson user. This user is used by the server to perform build job-related activities.

### 6.3.2 Installing Hudson on Windows

You must download the Hudson WAR distribution and start it in standalone mode by running the following command on the command line:

```
java -jar hudson-2.2.1.war
```

When Hudson starts:

1. Open the following URL in a web browser:  
`http://localhost:8080`
2. Navigate to **Manage Hudson**, then **Install as Windows Service**. This enables you to configure Hudson as a standard Windows service.

For instructions, see

<http://wiki.hudson-ci.org/display/HUDSON/Installing+Hudson+as+a+Windows+service>.

## 6.4 Configuring the HTTP Port

If you are using a single host for your artifact repository and continuous integration server, you must change the HTTP port used by Hudson.

- On Linux

This value is located in the `/etc/sysconfig/hudson` directory with `HUDSON_PORT`.

- On Windows  
This value is located in the `c:\ciroot\hudson\etc\sysconfig\hudson` directory.

## 6.5 Starting Hudson

To start Hudson:

- On Linux

If you have installed Hudson as a service, you can start the application by running the following command:

```
/etc/init.d/hudson start
```

Start up can be monitored on Linux by checking the logs in the following directory:

```
/var/log/hudson/hudson.log
```

Run the following command to monitor logs:

```
tail -f /var/log/hudson/hudson.log
```

- On Windows

Start Hudson on Windows as a normal service:

1. Go to **Control Panel**.
2. Navigate to **Administrative Tools**, then **Services**.
3. Select the Hudson service and click **Start**.

Hudson logs are available in the following location:

```
HUDSON_HOME/logs
```

## 6.6 Configuring Maven After Startup

This section contains the following topics:

- [First Time Startup](#)
- [Configuring the JDK](#)
- [Specifying the Maven Home](#)
- [Setting Up Maven for Use by Hudson](#)
- [Installing Hudson Plug-Ins](#)
- [Integrating the Repository](#)
- [Monitoring Subversion](#)

### 6.6.1 First Time Startup

The first time you start Hudson, go to the home page to complete the installation:

1. Open a browser and go to `http://localhost:8080` (change the port if you modified it during installation).
2. In the list of plugins that is presented, scroll down to find the **Subversion**, **Maven 3** and **Maven 3 SNAPSHOT Monitor** options and select these options.

3. Scroll down to the bottom and click **Install**.
4. Click **Finish** to move to the Hudson home page.

The rest of the configuration in this section is continued from the Hudson home page.

## 6.6.2 Configuring the JDK

You must configure the JDK you intend to use it for direct Java build configurations. To configure:

1. Log in to Hudson:  
`http://HOSTNAME:HUDSON_PORT`
2. Navigate to **Manage Hudson**, then **Configure System**.
3. In the **Configure System** screen, scroll down to the JDK section and click **Add JDK**, deselect the option **Install automatically** and then enter a name, for example, `jdk1.7.0` and add the complete path of your installed JDK.  
For example: `/ciroot/product_binaries/jdk1.7.0_21`.
4. Scroll down to the bottom of the page and click **Save**.

## 6.6.3 Specifying the Maven Home

You must specify the Maven 3 location so that Hudson knows where Maven is located. To do so:

1. Log in to Hudson:  
`http://HOSTNAME:HUDSON_PORT`
2. Navigate to **Manage Hudson**, then **Configure System**.
3. On the **Configure System** screen, scroll down to the section **Maven 3**.
4. Click **Add Maven**, then deselect the option **Install automatically** and enter a name and the fill path to the Maven installation, as shown in the following image:

**Maven 3**

Maven 3 installations

Name	MAVEN_HOME	Install automatically
Maven 3.0.4	/ciroot/apache-maven-3.0.4	<input type="checkbox"/>

List of Maven 3 installations on this system

5. Scroll down to the bottom of the page and click **Save**.

## 6.6.4 Setting Up Maven for Use by Hudson

To utilize Maven settings from Hudson, embed the `settings.xml` content into a settings object in Hudson's global Maven configuration:

1. Log in to Hudson:  
`http://HOSTNAME:HUDSON_PORT`



2. Go to **Manage Hudson**, then **Maven 3 Configuration**.
3. Click **Add**.
4. For Type, select **SETTINGS**.

The screenshot shows the 'Documents' interface in Hudson. At the top, there are buttons for 'Refresh', 'Add', and 'Remove'. Below is a table with columns 'ID', 'Type', and 'Name'. A single document is listed with ID 'd3d95231-81a1-4232-8494-041c3b394c30', Type 'SETTINGS', and Name 'Global'. Below the table is a form for editing the document. The 'ID' field is populated with the same ID. The 'Type' dropdown is set to 'SETTINGS'. The 'Name' field is 'Global'. There is a large empty text area for 'Description'. Below the form is a table of attributes:

Attributes	Value
CREATED	2012-08-23T13:36:07.394-0700
CREATED_BY	unknown
UPDATED	2012-08-23T13:42:26.801-0700

At the bottom, there is a large text area containing XML configuration for Maven settings:

```
<settings>
<profiles>
  <profile>
    <id>default</id>
    <repositories>
      <repository>
        <id>archiva-releases</id>
        <name>Archiva Releases</name>
        <releases>
          <enabled>true</enabled>
          <updatePolicy>always</updatePolicy>
          <checksumPolicy>warn</checksumPolicy>
        </releases>
        <snapshots>
          <enabled>false</enabled>
          <updatePolicy>never</updatePolicy>
          <checksumPolicy>fail</checksumPolicy>
        </snapshots>
        <url>http://localhost:8092/archiva/repository/internal</url>
      </repository>
    </repositories>
  </profile>
</profiles>
<layout>default</layout>
</settings>
```

At the bottom left of the XML area are buttons for 'Update' and 'Revert'.

5. Provide a name and optional description.
6. Find the `settings.xml` on your file system that you have configured in [Chapter 5](#) and copy the contents into the large text field at the bottom of the page. It is located in the `/$HOME/.m2/settings.xml` directory.
7. Click **Save**.

---

**Note:** Oracle recommends that you replace any localhost URL references in the `settings.xml` with fully qualified domain names or IP addresses because Hudson builds can eventually occur on non-local build hosts.

---

## 6.6.5 Installing Hudson Plug-Ins

Hudson jobs may require job-specific customizations of environment variables. Because Hudson does not support this by default, you must install an additional plug-in. To install the plug-in:

1. Log in to Hudson:  
`http://HOSTNAME:HUDSON_PORT`
2. Go to **Manage Hudson**, then **Manage Plugins**.
3. Select **Available**.
4. Select **Hudson Setenv Plugin**.
5. Click **Install**.
6. After the installation completes, use the restart option in Hudson to enable the plug-in.

## 6.6.6 Integrating the Repository

To configure automatic builds when changes are checked in, you must configure Hudson to monitor the artifact repository for SNAPSHOT deployment changes. Such changes trigger builds of affected components that have dependencies on the changed artifacts. To configure Hudson to monitor the artifact repository:

1. Log in to Hudson:  
`http://HOSTNAME:HUDSON_PORT`
2. Navigate to **Manage Hudson**, then **System Configuration**.
3. In the main system configuration panel under System Configuration, select **Maven 3 SNAPSHOT Monitor**. In the Archiva configuration instructions, you must have created a continuous integration specific user for continuous integration server access to the repository. Enter the path to Maven repository.

Maven 3 SNAPSHOT Monitor	
Maven Repository URL	<input type="text" value="http://localhost:8092/archiva/internal"/>
User	<input type="text" value="hudson"/>
Password	<input type="password" value="*****"/>

4. Set the **User** and **Password** for the continuous integration user.

## 6.6.7 Monitoring Subversion

In addition to monitoring the artifact repository for updated dependencies, the continuous integration server must constantly check the source control system for updates and trigger project builds accordingly. Unlike repository monitoring, software configuration management monitoring must be uniquely configured per build configuration. As you create new build configurations, you must set the Subversion location information for the related project. For more information, see [Section 12.3](#).

Subversion support comes with the base Hudson distribution. Other source control systems may require separate Hudson plug-in installation.

## 6.7 For More Information

You can find the primary source of the official documentation on Hudson in the following location:

<http://wiki.hudson-ci.org/display/HUDSON/Use+Hudson>

For new users, you can find introductory guides to Hudson in the following location:

[http://wiki.eclipse.org/Hudson-ci/Using\\_Hudson](http://wiki.eclipse.org/Hudson-ci/Using_Hudson)



---

---

# Understanding Maven Version Numbers

In a Maven environment, it is very important to understand the use of version numbers. A well thought out strategy can greatly simplify your dependency management workload. This chapter presents important concepts about how version numbers work in Maven in general, and also some specific details of how the Oracle-supplied artifacts use version numbers and how you should use them when referring to Oracle artifacts.

This chapter includes the following sections:

- [Section 7.1, "How Version Numbers Work in Maven"](#)
- [Section 7.2, "The SNAPSHOT Qualifier"](#)
- [Section 7.3, "Version Range References"](#)
- [Section 7.4, "Understanding Maven Version Numbers in Oracle Provided Artifacts"](#)

## 7.1 How Version Numbers Work in Maven

Maven's versioning scheme uses the following standards:

- MajorVersion
- MinorVersion
- IncrementalVersion
- BuildNumber
- Qualifier

For example:

- MajorVersion: 1.2.1
- MinorVersion: 2.0
- IncrementalVersion: 1.2-SNAPSHOT
- BuildNumber: 1.4.2-12
- Qualifier: 1.2-beta-2

All versions with a qualifier are older than the same version without a qualifier (release version).

For example:

1.2-beta-2 is older than 1.2.

Identical versions with different qualifier fields are compared by using basic string comparison.

For example:

1.2-beta-2 is newer than 1.2-alpha-6.

If you do not follow Maven versioning standards in your project versioning scheme, then for version comparison, Maven interprets the entire version as a simple string. Maven and its core plug-ins use version comparison for a number of tasks, most importantly, the release process.

If you use a nonstandard versioning scheme, Maven release and version plug-in goals might not yield the expected results. Because basic string comparison is performed on nonstandard versions, version comparison calculates the order of versions incorrectly in some cases.

For example, Maven arranges the version list in the following manner:

```
1.0.1.0
1.0.10.1
1.0.10.2
1.0.9.3
```

Version 1.0.9.3 should come before 1.0.10.1 and 1.0.10.2, but the unexpected fourth field (.3) forced Maven to evaluate the version as a string.

An example of this effect on Maven is found in the Maven Versions plug-in. The Maven Versions plug-in provides goals to check your project dependencies for currency in a different ways. One useful goal is `versions:dependency-updates-report`. The `versions:dependency-updates-report` goal examines a project's dependency hierarchy and reports which ones have newer releases available. When you are coordinating a large release, this goal can help you to find stale references in dependency configuration. If Maven incorrectly identifies a newer release, then it is also reported incorrectly in the plug-in. Given the preceding example sequence, if your current reference was 1.0.10.2, then the plug-in would report 1.0.9.3 as a newer release.

Version resolution is also very important if you intend to use version ranges in your dependency references. See [Section 7.3](#) for information about version changes.

## 7.2 The SNAPSHOT Qualifier

Maven treats the SNAPSHOT qualifier differently from all others. If a version number is followed by -SNAPSHOT, then Maven considers it the "as-yet-unreleased" version of the associated MajorVersion, MinorVersion, or IncrementalVersion.

In a continuous integration environment, the SNAPSHOT version plays a vital role in keeping the integration build up-to-date while minimizing the amount of rebuilding that is required for each integration step.

SNAPSHOT version references enable Maven to fetch the most recently deployed instance of the SNAPSHOT dependency at a dependent project build time. Note that the SNAPSHOT changes constantly. Whenever an agent deploys the artifact, it is updated in the shared repository. The SNAPSHOT dependency is refetched, on a developer's machine or it is updated in every build. This ensures that dependencies are updated and integrated with the latest changes without the need for changes to the project dependency reference configuration.

Usually, only the most recently deployed SNAPSHOT, for a particular version of an artifact is kept in the artifact repository. Although the repository can be configured to maintain a rolling archive with a number of the most recent deployments of a given artifact, the older instances are typically used only for troubleshooting purposes and do not play a role in integration.

Continuous build servers that include the ability to define and execute a job based on a Maven project, such as Hudson, can be configured to recognize when a SNAPSHOT artifact is updated and then rebuild projects that have a dependency on the updated artifact.

For example, a Hudson build configuration that maps to a Maven Project Object Model has a SNAPSHOT dependency. Hudson periodically checks the artifact repository for SNAPSHOT updates. When it detects the update of the project's dependency, it triggers a new build of the project to ensure that integration is performed with the most recent version of the dependency. If other projects have a dependency on this project, they too are rebuilt with updated dependencies.

## 7.3 Version Range References

Maven enables you to specify a range of versions that are acceptable to use as dependencies. [Table 7-1](#) shows a range of version specifications:

**Table 7-1** Version Range References

Range	Meaning
(,1.0]	$x \leq 1.0$
1.0	It generally means 1.0 or a later version, if 1.0 is not available. Various Maven plug-ins may interpret this differently, so it is safer to use one of the other, more specific options.
[1.0]	Exactly 1.0
[1.2,1.3]	$1.2 \leq x \leq 1.3$
[1.0,2.0)	$1.0 \leq x < 2.0$
[1.5,)	$x \geq 1.5$
(,1.0],[1.2,)	$x \leq 1.0$ or $x \geq 1.2$ . Multiple sets are separated by a comma.
(,1.1),(1.1,)	This excludes 1.1 if it is known not to work in combination with the library.

When Maven encounters multiple matches for a version reference, it uses the highest matching version. Generally, version references should be only as specific as required so that Maven is free to choose a new version of dependencies where appropriate, but knows when a specific version must be used. This enables Maven to choose the most appropriate version in cases where a dependency is specified at different points in the transitive dependency graph, with different versions. When a conflict like this occurs, Maven chooses the highest version from all references.

Given the option to use version ranges, you may wonder if there is still utility in using SNAPSHOT versions. Although you can achieve some of the same results by using a version range expression, a SNAPSHOT works better in a continuous build system for the following reasons:

- Maven artifact repository managers deal with SNAPSHOTs more efficiently than next version ranges. Because a single artifact can be deployed multiple times in a

day, the number of unique instances maintained by the repository can increase very rapidly.

- Non-SNAPSHOT release versions are meant to be maintained indefinitely. If you are constantly releasing a new version and incrementing the build number or version, the storage requirements can quickly become unmanageable. Repository managers are designed to discard older SNAPSHOTs to make room for new instances so the amount of storage required stays constant.
- SNAPSHOTs are also recognized by Maven and Maven's release process, which affords you some benefits when performing a release build.

## 7.4 Understanding Maven Version Numbers in Oracle Provided Artifacts

The two important scenarios where Maven version numbers are used in Oracle provided artifacts are as follows:

- In the Maven coordinates of the artifact, that is, in the `project.version` of the artifact's POM
- In the dependency section of POMs to refer to other artifacts

This section provides details on how version numbers are defined for Oracle artifacts in both the scenarios.

### 7.4.1 Version Numbers in Maven Coordinates

The version number of the artifact defined in the POM file is the same as the version number of the released product, for example 12.1.2.0.0, expressed using five digits, as described in the following:

In `x.x.x-y-z`:

- `x.x.x` is the release version number, for example 12.1.2.
- `y` is the `PatchSet` number, for example 0, 1, 2, 3, ... with no leading zeros.
- `z` is the `Bundle Patch` number, for example 0, 1, 2, 3, ... with no leading zeros.
- The periods and hyphens are literals.

---



---

**Note:** The version numbers of artifacts (as specified in `project.version` in the POM) use a different format than version number ranges used in dependencies (as specified in `project.dependencies.dependency.version` in the POM).

---



---

The release version number of Oracle-owned components do not change by a one-off patch. The release version number changes with a release and always matches the release, even if the component has not changed from the previous release.

The `PatchSet` (fourth position) changes when you apply a `PatchSet`. The `Bundle Patch` (fifth position) changes when you apply a `Bundle Patch`, `Patch set Update`, or equivalent (the name of this type of patch varies from product to product).

Following are the examples of valid version numbers:

```
12.1.2-0-0    12.1.2-1-0    12.1.2-2-0
12.1.2-0-1    12.1.2-1-1    12.1.2-2-1
              ...
12.1.2-0-10   12.1.2-1-1    12.1.2-2-1
```



## 7.4.2 Version Number Ranges in Dependencies

The two important scenarios where dependencies on Oracle-provided Maven artifacts are specified are as following:

- Inside the POM files of artifacts that are part of the Oracle product
- Inside POM files that you include in your own projects

The version number range should be specified in both the scenarios. This section describes how version number ranges are specified in Oracle-provided artifacts and when you are declaring a dependency on an Oracle-provided artifact.

When specifying dependencies on other artifacts, the most specific correct syntax should be used to ensure that the definition does not allow an incorrect or unsuitable version of the dependency to be used.

In `[x.x.x,y.y.y)`:

- `x.x.x` is the release version number, for example 12.1.2
- `y.y.y` is the next possible release version number, for example, 12.1.3
- Brackets, periods, commands and parenthesis are literals

An example of the correct way to specify a dependency is as follows:

```
[12.1.2,12.1.3)
```

As [Table 7–1](#) shows, the previous example means that the latest available version is 12.1.2 or greater, but less than 12.1.3.

The version number scheme used by Oracle-provided artifacts ensures correct sorting of version numbers, for example, Maven will resolve the following versions in the order shown (from oldest to newest):

```
12.1.2-0-0, 12.1.2-0-1, 12.1.2-0-2, 12.1.2-0-10, 12.1.2-1-0, 12.1.2-1-1, 12.1.2-1-2, 12.1.2-1-10,
12.1.2-0-0, 12.1.3-0-0
```

If it is necessary to specify a dependency which relies on a certain PatchSet or Bundle Patch, for example, when a new API is introduced, you must include the fourth or fourth and fifth digits respectively.

For example:

```
[12.1.2-2,12.1.3)      depends on 12.1.2 with PatchSet 2
[12.1.2-2-5,12.1.3)   depends on 12.1.2 with PatchSet 2 and Bundle Patch 5
```



---

## 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 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.

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

```
com.oracle.maven:oracle-common:12.1.2-0-0
- com.oracle.weblogic:wls-common:12.1.2-0-0
  - com.oracle.weblogic.archetype:basic-webapp:12.1.2-0-0
  - com.oracle.weblogic.archetype:basic-webapp-ejb:12.1.2-0-0
  - com.oracle.weblogic.archetype:basic-webservice:12.1.2-0-0
  - com.oracle.weblogic.archetype:basic-mdb:12.1.2-0-0
- com.oracle.coherence:gar-common:12.1.2-0-0
  - com.oracle.coherence:maven-gar-archetype:12.1.2-0-0
```

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:wls-common:12.1.2-0-0`, all projects associated with this parent will be 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.

This enables you to minimize the number of settings needed in each project POM. For example, if you are going to deploy all of 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:wls-maven-plugin:12.1.2-0-0` 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>12.1.2-0-0</version>
```

---

```

<packaging>pom</packaging>
<name>wls-common</name>
<parent>
  <groupId>com.oracle.maven</groupId>
  <artifactId>oracle-common</artifactId>
  <version>12.1.2-0-0</version>
</parent>
<build>
  <plugins>
    <plugin>
      <groupId>com.oracle.weblogic</groupId>
      <artifactId>wls-maven-plugin</artifactId>
      <version>12.1.2-0-0</version>
      <executions>
        <execution>
          <phase>pre-integration-test</phase>
          <goals>
            <goal>deploy</goal>
          </goals>
          <configuration>
            <user>weblogic</user>
            <password>welcome1</password>
            <verbose>true</verbose>
          </configuration>
        </execution>
      </executions>
      <configuration>
        <middlewareHome>/home/oracle/fmwhome</middlewareHome>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

Similarly, if you want to affect all projects targeted at any Oracle Fusion Middleware runtime, then you should place your customizations in `com.oracle.maven:oracle-common:12.1.2-0-0`.

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. For more information, see:

<http://www.sonatype.com/books/mvnref-book/reference/profiles.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.

---



---

## Building Java EE Projects for WebLogic Server with Maven

This chapter provides details on how to use the WebLogic Maven archetypes to create, build, and deploy WebLogic Server Java EE applications.

This chapter contains the following sections:

- [Section 9.1, "Introduction to Building Java EE Project with Maven"](#)
- [Section 9.2, "Using the Basic WebApp Maven Archetype"](#)
- [Section 9.3, "Using the Basic WebApp with EJB Maven Archetype"](#)
- [Section 9.4, "Using the Basic WebService Maven Archetype"](#)
- [Section 9.5, "Using the Basic MDB Maven Archetype"](#)

### 9.1 Introduction to Building Java EE Project with Maven

A Maven plug-in and four archetypes are provided for Oracle WebLogic Server. The Maven coordinates are described in [Table 9–1](#).

**Table 9–1** *Maven Coordinates with WebLogic Server*

Artifact	groupid	artifactId	version
WebLogic Server plug-in	com.oracle.weblogic	weblogic-maven-plugin	12.1.2-0-0
Basic WebApp archetype	com.oracle.weblogic.archetype	basic-webapp	12.1.2-0-0
WebApp with EJB archetype	com.oracle.weblogic.archetype	basic-webapp-ejb	12.1.2-0-0
Basic MDB archetype	com.oracle.weblogic.archetype	basic-mdb	12.1.2-0-0
Basic WebServices archetype	com.oracle.weblogic.archetype	basic-webservice	12.1.2-0-0

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

### 9.2 Using the Basic WebApp Maven Archetype

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=12.1.2-0-0
-DgroupId=org.mycompany
-DartifactId=my-basic-webapp-project
-Dversion=1.0-SNAPSHOT

```

This runs Maven's `archetype:generate` goal which enables you to create a new project from an archetype. The parameters are described in [Table 9-2](#).

**Table 9-2 Parameters for the Basic WebApp Project**

Parameter	Purpose
<code>archetypeGroupId</code>	Identifies the <code>groupId</code> of the archetype that you wish to use to create the new project. This must be <code>com.oracle.weblogic</code> as shown in the preceding example.
<code>archetypeArtifactId</code>	Identifies the <code>artifactId</code> of the archetype that you wish to use to create the new project. This must be <code>basic-webapp</code> as shown in the preceding example.
<code>archetypeVersion</code>	Identifies the version of the archetype that you wish to use to create the new project. This must be <code>12.1.2-0-0</code> as shown in the preceding example.
<code>groupId</code>	Identifies the <code>groupId</code> for your new project. This would normally start with your organization's domain name in reverse format.
<code>artifactId</code>	Identifies the <code>artifactId</code> for your new project. This would normally be an identifier for this project.
<code>version</code>	Identifies the version number for your new project. This would normally be <code>1.0-SNAPSHOT</code> 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 will limit the number of archetypes that you are shown and make the command execute much faster.

After creating your project, it will contain the following files:

```

|-- pom.xml
|-- src
|   |-- main
|       |-- java
|           |-- org
|               |-- mycompany
|                   |-- AccountBean.java
|       |-- webapp
|           |-- WEB-INF
|               |-- beans.xml
|               |-- web.xml
|               |-- weblogic.xml
|           |-- css
|               |-- bootstrap.css
|           |-- index.xhtml
|           |-- template.xhtml

```

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.

There are a number of files included in the project, as described in [Table 9-3](#).

**Table 9-3** 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, and 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.
All other files	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.

This section contains the following topics:

- [Customizing the Project Object Model File to Suit Your Environment](#)
- [Compiling Your Project](#)
- [Packaging Your Project](#)
- [Deploying Your Project to the WebLogic Server Using Maven](#)
- [Deploying Your Project to the WebLogic Server Using Different Options](#)
- [Testing Your Basic WebApp Project](#)

## 9.2.1 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. You should 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 Archiva, you should add a `pluginRepository` to the POM file. The following is an example, you can modify it to suit your environment:

```
<pluginRepositories>
  <pluginRepository>
    <id>archiva-internal</id>
    <name>Archiva Managed Internal Repository</name>
    <url>http://localhost:8081/archiva/repository/internal/</url>
    <releases>
      <enabled>true</enabled>
    </releases>
    <snapshots>
      <enabled>>false</enabled>
    </snapshots>
  </pluginRepository>
</pluginRepositories>
```

## 9.2.2 Compiling Your Project

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

```
mvn compile
```

This 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.

## 9.2.3 Packaging Your Project

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

```
mvn package
```

Again, this 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 of the goals up to and including the goal you name. This is very similar to a standard Java EE application, except that if you have some 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.

## 9.2.4 Deploying Your Project to the WebLogic Server Using Maven

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

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

This executes the `deploy` goal in the WebLogic Maven Plug-in. This goal supports all standard types of deployment archives.

## 9.2.5 Deploying Your Project to the WebLogic Server Using Different Options

After you have packaged your project, you can also deploy it to the WebLogic Server using any of the other existing (non-Maven) mechanisms. For example, the WebLogic Administration Console, or an ANT or WLST script.



## 9.2.6 Testing Your Basic WebApp Project

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

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

The user interface for the Basic WebApp looks like this:

### Basic Webapp

This project shows a basic example of a web application working with JSF and CDI to simulate the deposit functionality of bank system.

© Company 2012

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

## 9.3 Using the Basic WebApp with EJB Maven Archetype

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

1. 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-ejb
-DarchetypeVersion=12.1.2-0-0
-DgroupId=org.mycompany
-DartifactId=my-basic-webapp-ejb-project
-Dversion=1.0-SNAPSHOT
```

This runs Maven's `archetype:generate` goal which enables you to create a new project from an archetype. See [Table 9-4](#) for a description of the parameters.

**Table 9-4 Parameters for the Basic WebApp with EJB Project**

Parameter	Purpose
<code>archetypeGroupId</code>	Identifies the <code>groupId</code> of the archetype that you wish to use to create the new project. This must be <code>com.oracle.weblogic</code> as shown in the preceding example.
<code>archetypeArtifactId</code>	Identifies the <code>artifactId</code> of the archetype that you wish to use to create the new project. This must be <code>basic-webapp-ejb</code> as shown in the preceding example.
<code>archetypeVersion</code>	Identifies the version of the archetype that you wish to use to create the new project. This must be <code>12.1.2-0-0</code> as shown in the preceding example.

**Table 9–4 (Cont.) Parameters for the Basic WebApp with EJB Project**

Parameter	Purpose
groupId	Identifies the groupId for your new project. This would normally start with your organization's domain name in reverse format.
artifactId	Identifies the artifactId for your new project. This would normally be an identifier for this project.
version	Identifies the version number for your new project. This would normally be <b>1.0-SNAPSHOT</b> 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 will contain the following files:

```
my-basic-webapp-ejb-project/
|-- pom.xml
|-- src
|   |-- main
|       |-- java
|           |-- org
|               |-- mycompany
|                   |-- entity
|                       |-- Account.java
|                       |-- interceptor
|                           |-- LogInterceptor.java
|                           |-- OnDeposit.java
|                       |-- service
|                           |-- AccountBean.java
|                           |-- AccountManager.java
|                           |-- AccountManagerImpl.java
|       |-- resources
|           |-- META-INF
|               |-- persistence.xml
|       |-- scripts
|       |-- webapp
|           |-- WEB-INF
|               |-- beans.xml
|               |-- web.xml
|               |-- weblogic.xml
|           |-- css
|               |-- bootstrap.css
|       |-- index.xhtml
|       |-- template.xhtml
```

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.

There are a number of files included in the project, the purpose of each file is described in [Table 9–5](#).

**Table 9–5 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, and 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.
All other files	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 [Section 9.2.1](#).
4. Compile your Basic WebApp with EJB Project. See [Section 9.2.2](#).
5. Package your Basic WebApp with EJB Project. See [Section 9.2.3](#).
6. Deploy your Basic WebApp with EJB Project. For information about deploying it using Maven, see [Section 9.2.4](#). For information about deploying it using other options, see [Section 9.2.5](#).
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 where you deployed it:

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

The user interface for the Basic WebApp with EJB looks like this:

### Basic Webapp And EJB

This project shows a basic example of a web application working with JSF, EJB and JPA to simulate the deposit functionality of bank system.

Please Enter Your Account Name and Amount

Account Name

Amount

© Company 2012

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

## 9.4 Using the Basic WebService Maven Archetype

To use the Basic WebService project using the Maven Archetype:

1. Create a new Basic WebService project using the Maven archetype, you must issue a command similar to the following:

```

mvn archetype:generate
  -DarchetypeGroupId=com.oracle.weblogic.archetype
  -DarchetypeArtifactId=basic-webservice
  -DarchetypeVersion=12.1.2-0-0
  -DgroupId=org.mycompany
  -DartifactId=my-basic-webservice-project
  -Dversion=1.0-SNAPSHOT

```

This runs Maven's `archetype:generate` goal which enables you to create a new project from an archetype. See [Table 9–6](#) for the parameters and description.

**Table 9–6 Parameters for the Basic WebService Project**

Parameter	Purpose
<code>archetypeGroupId</code>	Identifies the <code>groupId</code> of the archetype that you wish to use to create the new project. This must be <code>com.oracle.weblogic</code> as shown in the preceding example.
<code>archetypeArtifactId</code>	Identifies the <code>artifactId</code> of the archetype that you wish to use to create the new project. This must be <code>basic-webservice</code> as shown in the preceding example.
<code>archetypeVersion</code>	Identifies the version of the archetype that you wish to use to create the new project. This must be <code>12.1.2-0-0</code> as shown in the preceding example.
<code>groupId</code>	Identifies the <code>groupId</code> for your new project. This would normally start with your organization's domain name in reverse format.
<code>artifactId</code>	Identifies the <code>artifactId</code> for your new project. This would normally be an identifier for this project.
<code>version</code>	Identifies the version number for your new project. This would normally be <b>1.0-SNAPSHOT</b> 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 will contain 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.

There are a number of files included in the project, see [Table 9–7](#) for the purpose of each file.

**Table 9–7 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 [Section 9.2.1](#).
4. Compile your Basic WebService Project. See [Section 9.2.2](#).
5. Package your Basic WebService Project. See [Section 9.2.3](#).
6. Deploy your Basic WebService Project. For information about deploying it using Maven, see [Section 9.2.4](#). For information about deploying it using other options, see [Section 9.2.5](#).
7. Test your Basic WebService Project.

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

```
http://servername:7001/basicWebservice/SayHello
```

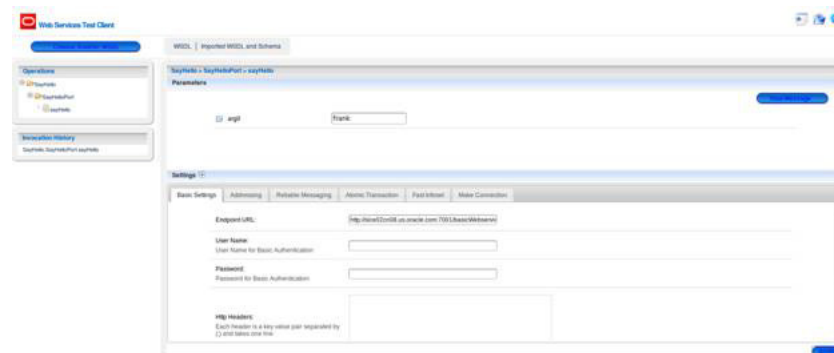
The user interface for the Basic WebService looks like the following:

## Web Services

Endpoint	Information
Service Name: {http://example.org}SayHello	Address: <a href="http://host/mycompany.com:7001/basicWebservice/Sayhello">http://host/mycompany.com:7001/basicWebservice/Sayhello</a>
Port Name: {http://example.org}SayHelloPort	WSDL: <a href="http://host/mycompany.com:7001/basicWebservice/SayHello?wsdl">http://host/mycompany.com:7001/basicWebservice/SayHello?wsdl</a> <a href="#">Test</a>
	Implementation class: org.mycompany.SayHello

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.

To test the web service, select **SayHello** operation in the left hand pane, then enter a value for **arg0** as shown in the following example, and select **Invoke**.



Scroll down to see the test results, as shown in the following example:



## 9.5 Using the Basic MDB Maven Archetype

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=12.1.2-0-0
-DgroupId=org.mycompany
-DartifactId=my-basic-mdb-project
-Dversion=1.0-SNAPSHOT
```

This runs Maven's `archetype:generate` goal which enables you to create a new project from an archetype. See [Table 9–8](#) for the parameters and description.

**Table 9–8 Parameters for the Basic MDB Project**

Parameter	Purpose
<code>archetypeGroupId</code>	Identifies the <code>groupId</code> of the archetype that you wish to use to create the new project. This must be <code>com.oracle.weblogic</code> as shown in the preceding example.
<code>archetypeArtifactId</code>	Identifies the <code>artifactId</code> of the archetype that you wish to use to create the new project. This must be <code>basic-mdb</code> as shown in the preceding example.
<code>archetypeVersion</code>	Identifies the version of the archetype that you wish to use to create the new project. This must be <code>12.1.2-0-0</code> as shown in the preceding example.
<code>groupId</code>	Identifies the <code>groupId</code> for your new project. This would normally start with your organization's domain name in reverse format.
<code>artifactId</code>	Identifies the <code>artifactId</code> for your new project. This would normally be an identifier for this project.
<code>version</code>	Identifies the version number for your new project. This would normally be <code>1.0-SNAPSHOT</code> 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 will contain the following files:

```

my-basic-mdb-project/
|-- pom.xml
|-- src
|   |-- main
|       |-- java
|           |-- org
|               |-- mycompany
|                   |-- jms
|                       |-- destination
|                           |-- QueueMDB.java
|                   |-- jsf
|                       |-- AccountBean.java
|       |-- scripts
|           |-- configure_resources.py
|       |-- webapp
|           |-- WEB-INF
|               |-- beans.xml
|               |-- web.xml
|               |-- weblogic.xml
|           |-- css
|               |-- bootstrap.css
|           |-- index.xhtml
|           |-- template.xhtml

```

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.

There are a number of files included in the project; see [Table 9–9](#) for the purpose of each file.

**Table 9–9 Files Created for the Basic MDB Project**

File	Purpose
<code>pom.xml</code>	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.
Files under <code>src/main/java</code>	An example Message Driven Bean that is used by the web application to store data.
All other files	HTML 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.
- Customize the POM to suit your environment. See [Section 9.2.1](#).
- Compile your Basic MDB Project. See [Section 9.2.2](#).
- Package your Basic MDB Project. See [Section 9.2.3](#).
- Deploy your Basic MDB Project. For information about deploying it using Maven, see [Section 9.2.4](#). For information about deploying it using other options, see [Section 9.2.5](#).
- Test your Basic MDB Project.

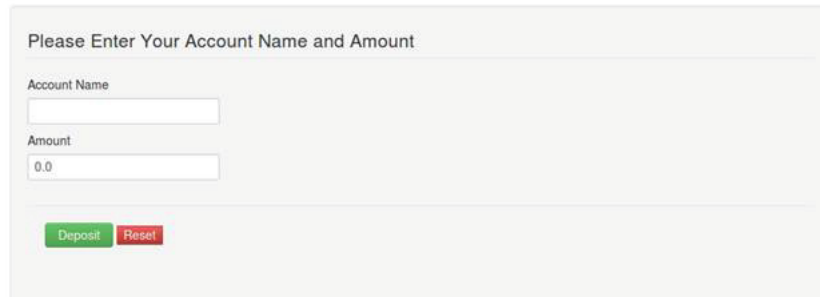
You can test the Basic MDB by visiting the following URL on the WebLogic Server where you deployed it:

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

The user interface for the Basic MDB looks like the following:

### Basic MDB

This project shows a basic example of a web application working with JSF, CDI, JMS and MDB to simulate the deposit functionality of bank system.



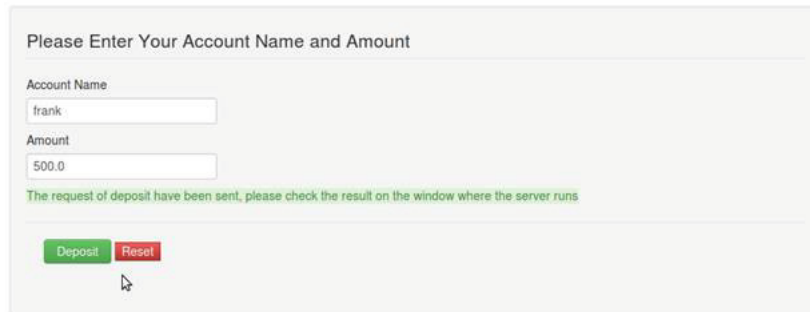
The screenshot shows a web form titled "Please Enter Your Account Name and Amount". It contains two input fields: "Account Name" and "Amount". The "Amount" field has the value "0.0". Below the fields are two buttons: "Deposit" (green) and "Reset" (red). The form is set against a light gray background.

© Company 2012

Provide the **Account Name** and **Amount**, then select **Deposit**:

### Basic MDB

This project shows a basic example of a web application working with JSF, CDI, JMS and MDB to simulate the deposit functionality of bank system.



The screenshot shows the same web form as above, but now the "Account Name" field contains the text "frank" and the "Amount" field contains "500.0". Below the fields, a green message box displays the text: "The request of deposit have been sent, please check the result on the window where the server runs". The "Deposit" and "Reset" buttons are still present. A mouse cursor is visible over the "Deposit" button.

© Company 2012

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



---

## Building Oracle Coherence Projects with Maven

This chapter provides details on how to use the Oracle Coherence archetypes to create, build, and deploy an Oracle Coherence project.

It includes the following topics:

- [Section 10.1, "Introduction to Building Oracle Coherence Projects with Maven"](#)
- [Section 10.2, "Creating a Project from a Maven Archetype"](#)
- [Section 10.3, "Building Your Project with Maven"](#)
- [Section 10.4, "Deploying Your Project to the WebLogic Server Coherence Container with Maven"](#)
- [Section 10.5, "Building a More Complete Example"](#)

### 10.1 Introduction to Building Oracle Coherence Projects with Maven

A Maven plug-in and an archetype is provided for Oracle Coherence Grid Archive (GAR) projects. [Table 10–1](#) describes the Maven coordinates.

**Table 10–1** *Maven Coordinates for Coherence*

artifacts	groupid	artifactId	version
Coherence plugin	com.oracle.coherence	maven-gar-plugin	12.1.2-0-0
Coherence archetype	com.oracle.coherence	maven-gar-archetype	12.1.2-0-0

[Table 10–2](#) describes the goals supported by the Oracle Coherence plug-in.

**Table 10–2** *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.
repackage	Repackages the packaged JAR archive with optional metadata and GAR extension.

## 10.2 Creating a Project from a Maven Archetype

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

```
mvn archetype:generate
  -DarchetypeGroupId=com.oracle.coherence
  -DarchetypeArtifactId=maven-gar-archetype
  -DarchetypeVersion=12.1.2-0-0
  -DgroupId=org.mycompany
  -DartifactId=my-gar-project
  -Dversion=1.0-SNAPSHOT
```

This will run Maven's `archetype:generate` goal which lets you create a new project from an archetype. [Table 10–3](#) describes the parameters:

**Table 10–3 Parameters for the Coherence Projects**

Parameter	Purpose
<code>archetypeGroupId</code>	Identifies the group ID of the archetype that you wish to use to create the new project. This must be <code>com.oracle.coherence</code> .
<code>archetypeArtifactId</code>	Identifies the artifact ID of the archetype that you wish to use to create the new project. This must be <code>maven-gar-archetype</code> .
<code>archetypeVersion</code>	Identifies the version of the archetype that you wish to use to create the new project. This must be <code>12.1.2-0-0</code> .
<code>groupId</code>	Identifies the group ID for your new project. This usually starts with your organization's domain name in reverse format.
<code>artifactId</code>	Identifies the artifact ID for your new project. This is usually an identifier for this project.
<code>version</code>	Identifies the version for your new project. This is usually <code>1.0-SNAPSHOT</code> 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-gar-project/
|-- pom.xml
  |-- src
    |-- main
      |-- java
      |-- resources
      |-- META-INF
        |-- cache-config.xml
        |-- coherence-application.xml
        |-- pof-config.xml
```

There are a number of files included in the project, as described in [Table 10–4](#).

**Table 10–4 Files in 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 it also includes 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 <i>generatePof</i> is set to <i>true</i> . By default, POF configuration metadata will not be generated.

If you are using POF in your project, 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 <i>true</i> . The configuration file is generated by scanning all classes in the GAR's classpath annotated with the class <code>com.tangosol.io.pof.annotation.Portable</code> . By default, POF configuration metadata is not generated.

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

```
<build>
<plugins>
...
  <plugin>
    <groupId>com.oracle.coherence</groupId>
    <artifactId>maven-gar-plugin</artifactId>
    <version>12.1.2-0-0</version>
    <extensions>true</extensions>
    <configuration>
      <generatePof>true</generatePof>
    </configuration>
  </plugin>
...
</plugins>
</build>
```

## 10.3 Building Your Project with Maven

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

To compile the source code in your project, run this command:

```
mvn compile
```

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

```
mvn package
```

## 10.4 Deploying Your 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 to deploy the GAR. This is done by adding 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>12.1.2-0-0</version>
  <executions>
    <!--Deploy the application to the server-->
    <execution>
      <phase>pre-integration-test</phase>
      <goals>
        <goal>deploy</goal>
      </goals>
      <configuration>
        <adminurl>t3://localhost:7001</adminurl>
        <user>weblogic</user>
        <password>welcome1</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>

```

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

```
mvn verify
```

## 10.5 Building a More Complete 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 [Chapter 11, "Building a Real Application with Maven"](#) to see an example that includes a Coherence GAR and a web application (WAR) that interacts with it.

---

# 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.

This chapter contains the following sections:

- [Section 11.1, "Introducing the Example"](#)
- [Section 11.2, "Multi-Module Maven Projects"](#)
- [Section 11.3, "Getting Started Building a Maven Project"](#)
- [Section 11.4, "Creating the GAR Project"](#)
- [Section 11.5, "Creating the WAR project"](#)
- [Section 11.6, "Creating the EAR project"](#)
- [Section 11.7, "Creating the Top-Level POM"](#)
- [Section 11.8, "Building the Application Using Maven"](#)

## 11.1 Introducing the Example

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 which you need to build into a Portable Object, a utility class to access the cache, and Coherence cache definitions.
- A Java 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 our 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 more information, refer to [Chapter 9](#) and [Chapter 10](#).

## 11.2 Multi-Module Maven Projects

Maven lets you to 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 to 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 as follows:

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:

`pom.xml`

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

`my-real-app-war/pom.xml`  
`my-real-app-war/src/main/webapp/WEB-INF/web.xml`  
`my-real-app-war/src/main/java/org/mycompany/servlets/MyServlet.java`

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

At the highest level, the 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.

## 11.3 Getting Started Building a Maven Project

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

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

Throughout the rest of this chapter, paths relative to this directory are given.

## 11.4 Creating the GAR Project

You can create the GAR project either using an archetype as described in [Section 10.2](#), or you can create the directories and files manually.

To use the archetype, run the following command:

```
mvn archetype:generate
  -DarchetypeGroupId=com.oracle.coherence
  -DarchetypeArtifactId=maven-gar-archetype
  -DarchetypeVersion=12.1.2-0-0
  -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
```

This section includes the following topics:

- [Section 11.4.1, "The POM File"](#)
- [Section 11.4.2, "Creating or Modifying the Coherence Configuration Files"](#)
- [Section 11.4.3, "Creating the Portable Objects"](#)
- [Section 11.4.4, "Creating a Wrapper Class to Access the Cache"](#)

### 11.4.1 The POM File

If you use the archetype, you will already have a POM file. You should modify that file to match the following example. If you create the project manually, you should 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>
  <groupId>org.mycompany</groupId>
  <artifactId>my-real-app-gar</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>gar</packaging>
  <parent>
    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <dependencies>
    <dependency>
      <groupId>com.oracle.coherence</groupId>
      <artifactId>coherence</artifactId>
      <version>12.1.2-0-0</version>
      <scope>provided</scope>
    </dependency>
```

```

</dependencies>
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>com.oracle.coherence</groupId>
        <artifactId>maven-gar-plugin</artifactId>
        <version>${coherence.version}</version>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </pluginManagement>
  <plugins>
    <plugin>
      <groupId>com.oracle.coherence</groupId>
      <artifactId>maven-gar-plugin</artifactId>
      <extensions>true</extensions>
      <configuration>
        <generatePof>true</generatePof>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

Examine the POM file to understand what each part stands for. The Maven coordinates for this project are set:

```

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

```

Notice that the packaging is `gar` because we are going to use the Coherence Maven plug-in to build this project into a Coherence GAR file.

The coordinates of the parent project are set. These coordinates point back to the top-level project. You need to create the POM for the top-level project in a later step.

```

<parent>
  <groupId>org.mycompany</groupId>
  <artifactId>my-real-app</artifactId>
  <version>1.0-SNAPSHOT</version>
</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:12.1.2-0-0`. 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>
    <version>12.1.2-0-0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

```



The `pluginManagement` section tells Maven to enable extensions for this plug-in. This is necessary to allow Maven to recognize GAR files as a target artefact type.

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

The `plug-ins` section includes any information that you must pass to the Coherence GAR plug-in. In this case, you must set `generatePof` to `true` so that the plug-in looks for POJOs with POF annotations and generate the necessary artifacts.

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

## 11.4.2 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, you should create these files in the locations indicated:

```
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 are 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>
  <!-- by default just include coherence POF user types -->
  <include>coherence-pof-config.xml</include>
</user-type-list>
</pof-config>
```

The following are 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>
```

Both of these files require little or no modification if you created them 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 [Chapter 10](#).

```
<?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>
```

### 11.4.3 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. 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 [Chapter 10](#).

### 11.4.4 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 org.mycompany.Person;
import com.tangosol.net.CacheFactory;
import java.util.Set;

public class CacheWrapper {
    private static CacheWrapper INSTANCE;

    public Set getPeople() {
        return CacheFactory.getCache("People").entrySet();
    }

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

    public static final CacheWrapper getInstance() {
        if(INSTANCE == null) INSTANCE = new CacheWrapper();
        return INSTANCE;
    }
}

```

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

## 11.5 Creating the WAR project

You can create the WAR project either using an archetype as described in [Chapter 9](#), or you can create the directories and files manually.

To use the archetype, run the following command:

```

mvn archetype:generate
    -DarchetypeArtifactId=basic-webapp
    -DarchetypeVersion=12.1.2-0-0
    -DgroupId=org.mycompany
    -DartifactId=my-real-app-war
    -Dversion=1.0-SNAPSHOT

```

If you use the archetype, you must remove any unnecessary files included in the project.

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
```

This section includes the following topics:

- [Section 11.5.1, "Creating or Modifying the POM File"](#)
- [Section 11.5.2, "Creating the Deployment Descriptor"](#)
- [Section 11.5.3, "Creating the Servlet"](#)

## 11.5.1 Creating or Modifying the POM File

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

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.mycompany</groupId>
  <artifactId>my-real-app-war</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>
  <parent>
    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <name>my-real-app-war</name>
  <dependencies>
    <dependency>
      <groupId>org.mycompany</groupId>
      <artifactId>my-real-app-gar</artifactId>
      <version>1.0-SNAPSHOT</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>javax.servlet-api</artifactId>
      <version>3.0.1</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

Let's review what is included in the POM file. First, you must set the coordinates for this project. Notice that the packaging for this project is `war`.

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

Then, 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>
</parent>

```

Finally, 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. This sets the display-name for the web application.

```

<dependencies>
  <dependency>
    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app-gar</artifactId>
    <version>1.0-SNAPSHOT</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

```

## 11.5.2 Creating the Deployment Descriptor

The web application has a simple Java EE deployment descriptor, located at this location:

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

The following are the contents of this file:

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">

  <display-name>my-real-app-war</display-name>

</web-app>

```

This sets the display-name for the web application.

## 11.5.3 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 Java EE web applications, hence the use of a simplistic servlet.

The following is the content for the servlet class:

```
package org.mycompany.servlets;
```

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

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

@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 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) {
            Iterator i = people.iterator();
            while (i.hasNext()) {
                Map.Entry entry = (Map.Entry)i.next();
```

```

        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, 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.

## 11.6 Creating the EAR project

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

This section includes the following topics:

- [Section 11.6.1, "The POM File"](#)
- [Section 11.6.2, "Deployment Descriptor"](#)

### 11.6.1 The POM File

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>
  <groupId>org.mycompany</groupId>
  <artifactId>my-real-app-ear</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>ear</packaging>
  <parent>
    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <name>ear assembly</name>
  <dependencies>
    <dependency>

```

```

    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app-gar</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>gar</type>
    <scope>optional</scope>
</dependency>
<dependency>
    <groupId>org.mycompany</groupId>
    <artifactId>my-real-app-war</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>war</type>
</dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <artifactId>maven-ear-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
          </manifest>
        </archive>
        <artifactTypeMappings>
          <artifactTypeMapping type="gar" mapping="jar" />
        </artifactTypeMappings>
      </configuration>
    </plugin>
    <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>org.mycompany</groupId>
                <artifactId>my-real-app-gar</artifactId>
                <version>1.0-SNAPSHOT</version>
                <type>gar</type>
              </artifactItem>
            </artifactItems>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
  <plugin>
    <groupId>com.oracle.weblogic</groupId>
    <artifactId>weblogic-maven-plugin</artifactId>
    <version>12.1.2-0</version>
    <executions>
      <!--Deploy the application to the server-->
      <execution>
        <phase>pre-integration-test</phase>
        <goals>

```



```

        <goal>deploy</goal>
    </goals>
    <configuration>
        <adminurl>t3://127.0.0.1:7001</adminurl>
        <user>weblogic</user>
        <password>welcome1</password>
        <middlewareHome>/home/mark/space/maven/wls030213</middlewareHome>
        <!--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>
</plugins>
</build>
</project>

```

Set the Maven coordinates for this project, and point to the parent:

```

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

```

Next, there are the dependencies on the WAR and GAR projects:

```

<dependencies>
    <dependency>
        <groupId>org.mycompany</groupId>
        <artifactId>my-real-app-gar</artifactId>
        <version>1.0-SNAPSHOT</version>
        <type>gar</type>
        <scope>optional</scope>
    </dependency>
    <dependency>
        <groupId>org.mycompany</groupId>
        <artifactId>my-real-app-war</artifactId>
        <version>1.0-SNAPSHOT</version>
        <type>war</type>
    </dependency>
</dependencies>

```

There are three separate plug-in configurations. The first of these 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>
    <artifactId>maven-ear-plugin</artifactId>
    <configuration>
        <archive>

```

```

        <manifest>
            <addClasspath>true</addClasspath>
        </manifest>
    </archive>
    <artifactTypeMappings>
        <artifactTypeMapping type="gar" mapping="jar" />
    </artifactTypeMappings>
</configuration>
</plugin>

```

Configure the `maven-dependency-plugin` 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>org.mycompany</groupId>
                        <artifactId>my-real-app-gar</artifactId>
                        <version>1.0-SNAPSHOT</version>
                        <type>gar</type>
                    </artifactItem>
                </artifactItems>
            </configuration>
        </execution>
    </executions>
</plugin>

```

And finally, tells the `weblogic-maven-plugin` how to deploy the resulting EAR file. In this section, you must update the `adminurl`, `user`, `password`, and `target` parameters to match your environment. For details on these parameters, see [Table 9-1](#).

```

<plugin>
    <groupId>com.oracle.weblogic</groupId>
    <artifactId>weblogic-maven-plugin</artifactId>
    <version>12.1.2-0</version>
    <executions>
        <!--Deploy the application to the server-->
        <execution>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>deploy</goal>
            </goals>
            <configuration>
                <adminurl>t3://127.0.0.1:7001</adminurl>
                <user>weblogic</user>
                <password>welcome1</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-->
            </configuration>
        </execution>
    </executions>

```

```

        <targets>AdminServer</targets>
        <verbose>>true</verbose>
        <name>${project.build.finalName}</name>
    </configuration>
</execution>
</executions>
</plugin>

```

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

## 11.6.2 Deployment Descriptor

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.

## 11.7 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>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.mycompany</groupId>
  <artifactId>my-real-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>my-real-app</name>
  <modules>
    <module>my-real-app-war</module>
    <module>my-real-app-gar</module>
    <module>my-real-app-ear</module>
  </modules>
  <properties>
    <coherence.version>12.1.2-0-0</coherence.version>
  </properties>
</project>

```

Set the coordinates for the project. You will notice that these match the parent coordinates you specified in each of the three projects. Note that the packaging is `pom`. This tells Maven that this project is an assembly of a set of sub-projects, as named in the `modules` section.

There are one `module` entry for each of the three sub-projects.

Since this POM is the parent of the other three, and since POM's inherit from their parents, you can add any common properties to this POM and it will be available in all the three sub-projects. In this case, you are adding the property `coherence.version`.

## 11.8 Building the Application Using Maven

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
```

Maven executes all of the phases up to the one named. These commands have the following effect:

Command	Details
<code>mvn compile</code>	Compile the Java source into target class files.
<code>mvn package</code>	<ol style="list-style-type: none"><li>1. Compile the Java source into target class files.</li><li>2. Create the archive (WAR, GAR, and so on) containing compiled files and resources (deployment descriptors, configuration files, and so on).</li></ol>
<code>mvn verify</code>	<ol style="list-style-type: none"><li>1. Compile the Java source into target Class files.</li><li>2. Create the archive (WAR, GAR, and so on) containing compiled files and resources (deployment descriptors, configuration files, and so on).</li><li>3. Deploy the EAR file to the WebLogic Server environment.</li></ol>

---

## From Build Automation to Continuous Integration

This chapter provides a quick overview of some of the important considerations that you will need to think about when you move from a simple build automation to a continuous integration environment.

If you have been following the examples in this book, you would have seen how to use Maven to manage the build process for projects which are targeted for deployment on Oracle Fusion Middleware environments.

The next logical step is to move towards a continuous integration approach, so that the builds of all of your projects can be triggered, managed and monitored in one central place.

The advantage of continuous integration comes from componentization of an application and constant integration of those components as they are independently modified, often by different groups of developers. Maven projects are used to represent these components and their relationships to each other. Since Hudson understands Maven's project relationship model, it can automatically rebuild and retest affected components in the order that they should be built in. When Hudson detects the changes to the code-base, the affected components are built and reintegrated in correct order to ensure proper function of the entire application.

This chapter includes some of the important things to consider while moving to a continuous integration environment with Hudson. This chapter includes the following sections:

- [Section 12.1, "Dependency Management"](#)
- [Section 12.2, "Maven Configuration to Support Continuous Integration Deployment"](#)
- [Section 12.3, "Automating the Build with Hudson"](#)
- [Section 12.4, "Monitoring the Build"](#)

### 12.1 Dependency Management

Dependency management is a key feature of Maven and something that distinguishes it from other build automation technologies, like ANT, which Fusion Middleware has supported for some time. The section explores some important dependency management topics.

This section includes the following topics:

- [Section 12.1.1, "Using SNAPSHOT"](#)

- [Section 12.1.2, "Dependency Transitivity"](#)
- [Section 12.1.3, "Dependency Scope"](#)
- [Section 12.1.4, "Multiple Module Support"](#)

### 12.1.1 Using SNAPSHOT

Snapshot versioning is covered more extensively in [Chapter 7](#). Using snapshots for components that are under development is required for the automated continuous integration system to work properly. Note that a fixed version, non-snapshot versioned artifact should not be modified and replaced. The best practice is that you should not update artifacts after they are released. This is a core assumption of the Maven approach. However, it is worth noting that often this assumption is not correct in enterprise software development, where vendors and end users do sometimes update "finished" artifacts without changing the version number, for example through patching them in place. Even though it is possible to violate this rule, every attempt should be made to comply to ensure integration stability.

### 12.1.2 Dependency Transitivity

Most projects have dependencies on other artifacts. At build time, Maven obtains these artifacts from the configured artifact repositories and use them to resolve compilation, runtime and test dependencies.

Dependencies explicitly listed in the POM may also have dependencies of their own. These are commonly referred to as transitive dependencies. Based on dependency attributes such as scope and version, Maven uses rules to determine which dependencies the build should utilize. An important part of this resolution process has to do with version conflicts. It is possible that a project may have transitive dependencies on multiple versions of the same artifact (identical `groupId` and `artifactId`). In such a case, Maven uses the **nearest definition** which means that it uses the version of the closest dependency to your project in the tree of dependencies. You can always guarantee a particular version by declaring it explicitly in your project's POM.

---

---

**Note:** If two dependency versions are at the same depth in the dependency tree, until Maven 2.0.8 it was not defined which one would win, but since Maven 2.0.9 it is the order in the declaration that counts. Hence, the first declaration wins.

---

---

### 12.1.3 Dependency Scope

Dependencies may optionally specify a scope. In addition to determining whether or not a dependency is made available to the classpath during a particular build phase, scope affects how transitive dependency is propagated to the classpath.

There are six scopes available.

- **Compile:** This is the default scope, used if no scope is specified. Compile dependencies are available in all classpaths of a project. Furthermore, these dependencies are propagated to dependent projects.
- **Provided:** This is much like compile, but indicates you expect the JDK or a container to provide the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you can set the dependency on the servlet API and related Java EE APIs to scope provided because the web container

provides those classes. This scope is only available on the compilation and test classpath, and is not transitive.

- **Runtime:** This scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.
- **Test:** This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases.
- **System:** This scope is similar to Provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.
- **Import:** (*only available in Maven 2.0.9 or higher*) This scope is only used on a dependency of type POM. It indicates that the specified POM should be replaced with the dependencies in that POMs. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

### 12.1.4 Multiple Module Support

A series of interdependent projects, such as an application, can be aggregated by a multi-module POM. This should not be confused with a parent POM which provides inherited configuration. A multi-module POM may also be an inheritance parent to sub-module projects. When a Maven build is executed upon a multi module POM, Maven examines the tree of sub-projects and calculates the correct order of dependency to build the modules.

Multiple module POMs can be useful for organizing multiple component builds in Hudson.

## 12.2 Maven Configuration to Support Continuous Integration Deployment

This section describes some aspects of Maven that you should consider while moving to a continuous integration environment.

This section contains the following topics:

- [Section 12.2.1, "Distribution Management"](#)
- [Section 12.2.2, "Snapshot Repository Settings"](#)

### 12.2.1 Distribution Management

Every project that is part of continuous integration must specify a `distributionManagement` section in its POM. This section tells Maven where the artifacts are going to be deployed at the end of the build process, that is, which repository (local or remote). The examples used in this book use the Archiva repository. Deploying artifacts to a repository makes them available for other projects to use as dependencies.

You must define a `distributionManagement` section that describes which repository to deploy snapshots and releases to. It is recommended that the `distributionManagement` configuration be placed at a common inherited POM that is shared among all projects such as the oracle-common POM, as described in [Chapter 8](#).

The following shows an example of a `distributionManagement` configuration:

```
<distributionManagement>
```

```

<repository>
  <uniqueVersion>false</uniqueVersion>
  <id>releases</id>
  <name>Releases</name>
  <url>http://server:port/archiva/repository/releases/</url>
  <layout>default</layout>
</repository>
<snapshotRepository>
  <uniqueVersion>true</uniqueVersion>
  <id>snapshots</id>
  <name>Snapshots</name>
  <url>http://server:port/archiva/repository/snapshots</url>
  <layout>default</layout>
</snapshotRepository>
</distributionManagement>

```

## 12.2.2 Snapshot Repository Settings

There are some important settings that govern how and when Maven will access repositories:

**Update Policy:** This controls how often Maven will check with a remote repository for updates to an artifact that it already has in its local repository. Configure your snapshot repository in your `settings.xml` in Hudson to use **updatePolicy** as `always`. The effect of **updatePolicy** is on your development systems. The default value is `daily`. If you want to integrate the changes as they occur in Hudson, you should change their **updatePolicy** accordingly. Dependencies may change suddenly and without warning. While the continuous integration system should have sufficient tests in place to reduce the occurrence of regressions, you can still run into issues depending on up-to-the-minute snapshots while developing. One such example is the API changes.

You should get all project snapshot dependencies up-to-date so that their local build reflects the current state of the deployed code-base prior to check-in.

**Server credentials:** This tells Maven the credentials that are needed to access a remote repository; typically Maven repositories will require you to authenticate before you are allowed to make changes to the repository, for example, publishing a new artifact). Unless you have given the Archiva guest user global upload privileges, which is not recommended, you must specify correct credentials for the snapshot repository in the servers section. You should have a unique Hudson user with snapshot repository upload permissions. See [Chapter 4](#) for details about user and role configuration.

Use Maven's password encryption for the password value. The Maven guide to password encryption can be found here:

<http://maven.apache.org/guides/mini/guide-encryption.html>.

The following shows a sample `settings.xml` configuration for the Hudson user:

```

<settings>
...
  <servers>
...
    <server>
      <id>snapshots</id>
      <username>hudson</username>
      <password>{COQLCE6DU6GtcS5P=}</password>
    </server>
...

```



```
</servers>
...
</settings>
```

## 12.3 Automating the Build with Hudson

This section discusses how to set up your build jobs in Hudson. There are various options available to build Maven projects. This section describes the approach recommended by Oracle.

Before proceeding, ensure that you have configured Hudson, as described in [Chapter 6](#).

This section contains the following topics:

- [Section 12.3.1, "Creating a Hudson Job to Build a Maven Project"](#)
- [Section 12.3.2, "Triggering Hudson Builds"](#)
- [Section 12.3.3, "Managing a Multi-Module Maven Build with Hudson"](#)

### 12.3.1 Creating a Hudson Job to Build a Maven Project

To create a basic Maven Hudson job:

1. Open the Hudson web interface and log in, if necessary.
2. Create a new job:
  - a. Select **New Job** from the right-hand menu.
  - b. Provide a unique name and select **Build a free-style software project**.
  - c. Click **OK**.
3. Configure the source code management

Ensure that you complete configuring the Subversion server, including the SSH public and private key configuration.

- a. Under **Source Code Management**, select **Subversion**.
- b. Provide the repository URL for your project directory. For example, `svn+ssh://subversion-server/ciroot/subversion/repository/trunk/projects/my-project`

---



---

**Note:** In this example we are using a `svn+ssh` URL, which accesses Subversion using SSH. If you are using a different protocol, then the steps that are necessary to configure it may vary slightly.

Hudson attempts to verify the URL and may respond with an error message like the following:

```
Unable to access
svn+ssh://hostname/ciroot/subversion/repository/trunk :
svn: E200015: authentication cancelled(show details)
(Maybe you need to enter credential?)
```

If you get this error message, do the following:

1. From the message, click **enter credentials**.
  2. Select **SSH public key authentication (svn+ssh)**.
  3. Enter the user name.
  4. Enter the SSH private-key passphrase if required.
  5. Select the private-key file from the Hudson file system. It should be in a `~/.ssh/id_rsa` format.
- 
- 

4. Add a Maven build step
  - a. Under the Build section, select **Invoke Maven 3** from the **Add Build Step** drop-down menu.
  - b. Select **Maven 3 home**. Add necessary goals and properties in the appropriate text fields.
  - c. If you have a SNAPSHOT continuous integration build environment, then configure the goals to perform a **clean deploy**.
  - d. If necessary, open the **Advanced** settings and ensure that the Settings entry points to the Maven settings that you created in the Hudson web interface, while configuring Hudson.
5. Save the configuration
 

Click **Save** at the bottom of the page.

## 12.3.2 Triggering Hudson Builds

Hudson provides number of ways to manage a continuous integration build's triggers in Hudson. These include manual and automated triggers. The option to manually start a build is always available for any job. When choosing an automated trigger, you may consider factors like the structure of the project, the location of the source code, the presence of any branches, and so on.

This section contains the following topics:

- [Section 12.3.2.1, "Manual Build Triggering"](#)
- [Section 12.3.2.2, "Subversion Repository Triggering"](#)
- [Section 12.3.2.3, "Schedule Based Triggering"](#)
- [Section 12.3.2.4, "Trigger on Hudson Dependency Changes"](#)
- [Section 12.3.2.5, "Maven SNAPSHOT Changes"](#)

Regardless of how the build is triggered, the job is added to the pending job queue and completed when the resources become available.

### 12.3.2.1 Manual Build Triggering

All jobs can be started from the user interface with the **Build Now** link.

### 12.3.2.2 Subversion Repository Triggering

This type of build trigger is vital to establishing a healthy continuous integration build. As changes are committed to project source, Hudson triggers builds of the associated Hudson jobs. The trigger does this by periodically checking the associated Subversion URL for changes.

To enable this trigger, select the **Poll SCM** option. You must then provide a cron expression to determine the schedule Hudson uses to poll the repository.

### 12.3.2.3 Schedule Based Triggering

For some job types, you can trigger them on a schedule. Long running system integration tests are an example of a build that you might want to run periodically as opposed to every time the test source is modified.

Schedule based triggers are configured with cron expressions exactly like the Poll SCM trigger.

### 12.3.2.4 Trigger on Hudson Dependency Changes

Trivial projects may contain multiple builds that produce unique artifacts that have dependencies on each other. If Hudson rebuilds an artifact as the result of any trigger type, it must also build and test dependent artifacts to ensure integration is still valid. When dependencies that are also built on this Hudson server are successfully completed, Hudson recognizes these relationships automatically and triggers the build. In order for this trigger to work, the dependencies must also enable the post-build action **Notify that Maven dependencies have been updated by Maven 3 integration**.

### 12.3.2.5 Maven SNAPSHOT Changes

If there are dependencies that are undergoing concurrent development and being managed as snapshots in your common Maven repository, then they should be managed by your Hudson instance. If this is not practical, then you can use the SNAPSHOT dependency trigger to monitor the Maven repository for changes in such dependencies. When an updated SNAPSHOT dependency has been detected, the build will trigger and download the new dependency for integration.

This trigger also uses a cron expression to configure the polling interval.

## 12.3.3 Managing a Multi-Module Maven Build with Hudson

To manage your build dependencies correctly, you may add each project as a separate Hudson build and configure the dependency triggers manually, or you can configure a multi-module Maven POM as a parent Hudson job. The multi-module solution reduces the possibility of making mistakes wiring the dependencies manually. It also automatically stays up-to-date as the dependencies are changed in the Maven configuration.

Configuration of a multi-module build is identical to configuration for regular projects. To examine the results of component project builds, Hudson provides a tab in the build results page in the Maven 3 Build Information link. At the top of the page, the Modules tab summarizes the component build results.

To examine the log for any of the sub-project builds, use the Console Log link. All project and sub-project builds are logged in their sequence of execution.

## 12.4 Monitoring the Build

Hudson should be configured to send notifications to the correct parties when the build breaks. The continuous integration system must ensure that their changes do not break the build and test process. If this does happen, they need to be notified of the breakage and must address the issue as soon as possible. Hudson should have each user registered as a unique user. The Hudson user name must match the Subversion user name that they ordinarily commit under. Hudson relies on this name to look up the proper contact email to send notification to.

General email notification configuration can be found under **Manage Hudson -> Configure System -> E-mail Notification**.

You must also make sure some form of user management is enabled. You can do this by selecting **Enable Security** from the **Configure System** panel. There are a number of choices for user management and additional third-party plug-ins to support most other popular solutions, such as LDAP. The best option is to use **Hudson's own user database**. Select this choice from the **Access Control** section. There are additional options for limiting permissions to particular users and groups.

To add a new user, the user simply needs to follow the **sign up** link at the top of the Hudson home page and fill out the necessary information. The user name must match the corresponding Subversion user name.

### 12.4.1 Following Up on the Triggered Builds

Normally, automated notification is sufficient to ensure the continuous build system is kept healthy and produces effective results. However, there are conditions that can require additional monitoring and coordination to get the system back to an operational state. It is a good policy to designate a build coordinator to track down such problems, coordinate solutions for broad problems and perform troubleshooting when the system itself is suspect.