Spring MVC Cookbook

Spring MVC is a lightweight framework for web applications. Being part of the Spring Framework, it naturally extends it to offer a consistent solution for web related issues.

By designing your Maven modules, you will achieve an Enterprise-standard stateless REST application. This guide uniquely features a massive overview of practical development techniques collated from the Spring ecosystem. You will implement Hypermedia and HATEOAS to guide your customer’s stateless conversation with the product and see how a messaging service based on WebSocket can be configured. Finally, you will learn how to set up and organize different levels of automated tests.

What this book will do for you...

- Structure your project with Maven and create self-tested, domain-specific deployable web archives
- Generate templates for a responsive and powerful frontend with AngularJS and Bootstrap
- Build a high performance stateless RESTful and hypermedia application to support your multiple customer experiences
- Authenticate over REST with a BASIC authentication scheme and OAuth2
- Document and publish your REST API using Swagger and Swagger UI
- Communicate through WebSocket and STOMP messages

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions to solve problems efficiently
- Clear explanations of what you did
- Solutions that can be applied to solve real-world problems

Spring MVC Cookbook

Over 40 recipes for creating cloud-ready Java web applications with Spring MVC

Alex Bretet
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Setup Routine for an Enterprise Spring Application'
- A synopsis of the book’s content
- More information on Spring MVC Cookbook
About the Author

**Alex Bretet** is a certified Java and Spring Integration engineer. Currently working at Thunderhead, a pioneer company and global actor among SaaS providers, he has a rich developer background from different industries including energy, insurance, finance, and the Internet.

Captivated by the Internet's communication capabilities and by start-ups, he believes in skunk development outcomes (when groups of like-minded people together achieve unbelievable targets on tight deadlines).

He also defends a number of open source initiatives and indeed Spring technologies, whose pragmatism constantly disrupts the most established practices and offers valuable alternatives for the benefit of all.

You can contact him at [alex.bretet@gmail.com](mailto:alex.bretet@gmail.com) or follow him on Twitter at [@abretet](https://twitter.com/abretet)
Welcome to the singular universe of *Spring MVC Cookbook*. We hope you are ready for this journey that will take you through modern Spring web development practices. We have been building the [cloudstreetmarket.com](http://cloudstreetmarket.com), a stock exchange platform with social capabilities. We are about to take you through each step of its development process.

**What this book covers**

*Chapter 1, Setup Routine for an Enterprise Spring Application,* introduces a set of practices that are standard in the industry. From the configuration of an Eclipse IDE for optimized support of Java 8, Tomcat 8, Git and Maven, to a proper understanding of Maven as a build automation tool and as a dependency management tool, you will learn how to deploy the Spring Framework on a sustainable base in this chapter.

Whether a project is meant to become a massively profitable product or to remain a rewarding experience, it should always begin with the same Enterprise-level patterns.

This chapter is more than a guide through the first development stage of the book's application, Cloud Street Market. A bunch of standard practices are presented here as a routine for developers toward enterprise Spring applications.

*Chapter 2, Designing a Microservice Architecture with Spring MVC,* is a slightly longer chapter. It covers the core principles of Spring MVC, such as its request flow or the central role of DispatcherServlet. It is also about learning how to configure Spring MVC controllers and controller method-handlers with an extended source of information related to a controller's annotations.

In the path of a Microservice architecture, we install Spring and Spring MVC across modules and web projects to build functional units that are easy to deploy and easy to scale. In this perspective, we are going to shape our application with a web module that is responsible for serving a Twitter Bootstrap template along with another web module specialized in REST Web Services.
In this chapter, you also will learn how to transfer the model from controllers to JSP views using the JSTL and how to design a JavaScript MVC pattern with AngularJS.

Chapter 3, Working with Java Persistence and Entities, gives you a glimpse of. It is necessary at this stage to learn how persistent data can be handled in a Spring ecosystem and thus in a Spring MVC application. We will see how to configure, a JPA persistence provider (Hibernate) from dataSource and entityManagerFactory in Spring. You will learn how to build a beneficial JPA object-relational mapping from EJB3 Entities and then how to query repositories using Spring Data JPA.

Chapter 4, Building a REST API for a Stateless Architecture, provides insights into Spring MVC as a REST Web Services engine. We will see the amazing support the Framework provides, with several annotations acting as doorknobs on method-handlers to abstract web-related logic and thus only focus on the business. This principle appears with annotations for request binding (binding of parameters, URL paths, headers, and so on) and Response Marshalling and also for integrated support of Spring Data pagination.

This chapter also presents how to set up exception handling as part of Spring MVC to translate predefined exception-Types into generic error responses. You will understand how to configure the content negotiation, an important bit for REST APIs, and finally, how to expose and document REST Endpoints using Swagger and the Swagger UI.

Chapter 5, Authenticating with Spring MVC, presents how to configure an authentication on controllers and services from standard protocols such as HTTP BASIC and OAuth2. You will learn several concepts and practices related to Spring Security, such as the Filters chain, the <http> namespace, the Authentication Manager, or the management of roles and users. Our OAuth2 flow is a client implementation. We authenticate users in our application when they are first authenticated with a third-party provider: Yahoo! These Yahoo! authentications and connections are later used to pull fresh financial data from Yahoo! Finance. It will be very interesting to see how the OAuth2 flow can be entirely abstracted, in the backend, with the Spring Social library.

Chapter 6, Implementing HATEOAS, demonstrates how to take a RESTful Spring MVC API a step further. A Hypermedia-driven application provides links along with every single requested resource. These links reflect URLs of related resources. They provide to the user client (whatever type of client it may be) real-time navigation options—precious documentation that is also the actual implementation.

We will see how to build such links from JPA Entities associations or from the controller layer.

Chapter 7, Developing CRUD Operations and Validations, goes into the more advanced concepts of Spring MVC. Presenting the tools and techniques that support interactive HTTP methods (PUT, POST, and DELETE), we will lean on the HTTP1/1 specification (RFC 7231 Semantics and Content) to learn how to return the appropriate response status code or headers.

Our use of the Spring Validator together with the ValidationUtils utility class provides a compliant implementation of the validation-related JSR-303 and JSR-349 specifications.
The last recipe is the place where an internationalization (I18N) of messages and content happens. We also present a client-side implementation, with AngularJS, that relies on published internationalization web services.

Chapter 8, Communicating Through WebSockets and STOMP, focuses on the uprising WebSocket technology and on building Message-Oriented-Middleware for our application. This chapter provides a rare showcase that implements so much about WebSockets in Spring. From the use of the default embedded WebSocket message broker to a full-featured external broker (with STOMP and AMQP protocols), we will see how to broadcast messages to multiple clients and also how to defer the execution of time-consuming tasks with great scalability benefits.

You will learn how to dynamically create private queues and how to get authenticated clients to post and receive messages from these private queues.

To achieve a WebSocket authentication and an authentication of messages, we will make the API stateful. By stateful, understand that the API will use HTTP sessions to keep users authenticated between their requests. With the support of Spring Session and the highly clusterable Redis server, sessions will be shared across multiple web apps.

Chapter 9, Testing and Troubleshooting, introduces a set of tools and common practices to maintain, debug, and improve an application's state. As a way of finishing this journey, we will visit how to upgrade the database schema from one version of the application to another as part of the Maven builds with the Flyway Maven Plugin. We will also go through how to write and automate unit tests (with Maven Surefire and Mockito) and integration tests (using a set of libraries such as Cargo, Rest-assured, and Maven Failsafe).

The last recipe provides insightful guidelines in order to apply Log4j2 globally as a logging framework, as it is more than important to rely on a relevant logging solution to troubleshoot efficiently, whichever environment we may be on.
Setup Routine for an Enterprise Spring Application

The topics covered in this chapter correspond to this four-step routine:

- Installing Eclipse for JEE developers and Java SE 8
- Configuring Eclipse for Java SE 8, Maven 3, and Tomcat 8
- Defining the project structure with Maven
- Installing Spring, Spring MVC, and a web structure

Introduction

Before we dive into this routine for initializing the developments, we are going to answer, as an introduction, a couple of questions that should help you understand the routine better.

Remember that the result of this chapter will also constitute the minimal starting point for all the further chapters.

Let's do it with fun!

Throughout this book, we will be acting on behalf of the ZipCloud company. ZipCloud aims to build different products in the social and financial industry. We are going to build the first product of the company: cloudstreetmarket.com which will be a wonderful stock exchange platform with social capabilities. This project must be an optimal beginning for this little ZipCloud start-up!
Why such a routine?

Whatever your initial objectives may be, it is necessary to make sure that the design will not suffer from early stage failures. This routine should cover you against this risk.

The idea beyond the routine itself is to share a bootstrap methodology to kick off the project base that you need now and that will support your needs tomorrow. The routine is also a key to drive your product thoughts toward a sustainable architecture which will be easy to refactor and to maintain.

Setting up a new project for an enterprise-level architecture will not kill the excitement and creativity!

Why making use of the Eclipse IDE?

There is competition in the domain, but Eclipse is popular among the Java community for being an active open source solution; it is consequently accessible online to anyone with no restrictions. It also provides, among other usages, a very good support for web implementations and particularly to MVC web implementations.

Why making use of Maven?

Maven is a software project management and comprehension tool. It is an open source project supported by the Apache community and the Apache Software Foundation. For nearly 10 years, Maven has given massive benefits. It has also shaped a standard structure for Java projects. With its Project Object Model (POM) approach, it provides, to anyone and potentially to any third-party software, a uniform and radical way of understanding and building a Java project hierarchy with all its dependencies.

In early stage architectures, it is critical to consider the following decisions:

- Opening the project definition to potentially different development environments and continuous integration tools
- Monitoring the dependencies and maybe securing their access
- Imposing a uniform directory structure within the project hierarchy
- Building a self-tested software with self-tested components

Choosing Maven secures these points and fulfills our project's need to make our project reusable, secure, and testable (under automation).
What does the Spring Framework bring?

The Spring Framework and its community have contributed to pulling forward the Java platform for more than a decade. Presenting the whole framework in detail would require us to write more than a book. However, the core functionality based on the principles of Inversion of Control (IOC) and Dependency Injection (DI) through a performant access to the bean repository allows considerable reusability. Staying lightweight, it secures great scaling capabilities and could probably suit all modern architectures.

Installing Eclipse for JEE Developers and Java SE 8

The following recipe is about downloading and installing the Eclipse IDE for JEE developers and downloading and installing JDK 8 Oracle Hotspot.

Getting ready

This first recipe could appear redundant or unnecessary in regard to your education or experience. However, having a uniform configuration all along this book will provide you with many benefits.

For instance, you will certainly avoid unidentified bugs (integration or development). You will also experience the same interfaces as seen in the presented screenshots. Also, because the third-party products are living, you will not have the surprise of encountering unexpected screens or windows.

How to do it...

The whole first chapter in general requires a step by step cooperation. From the next chapter, we will be using GIT and your active involvement will be lightened.

1. Download a distribution of the Eclipse IDE for Java EE developers:

   - We will be using an Eclipse Luna distribution in this book. We recommend that you install this version in order to match our guidelines and screenshots completely. Download a Luna distribution for the OS and environment of your choice from [https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunasr1](https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunasr1).

     The product to download is not a compiled installer but a zip archive.

   - If you feel confident enough to use another version (more recent) of the Eclipse IDE for Java EE Developers, all of them can be found at [https://www.eclipse.org/downloads](https://www.eclipse.org/downloads).
For upcoming installations, on Windows, a few target locations are suggested to be in the root directory (C:\). To avoid permission-related issues, it would be better if your Windows user is configured to be a Local Administrator. If you can’t be part of this group, feel free to target installation directories you have write access to.

2. Extract the downloaded archive into an eclipse directory from the steps:
   - C:\Users\{system.username}\eclipse: Extract it here if you are on Windows
   - /home/usr/{system.username}/eclipse: Extract it here if you are on Linux
   - /Users/{system.username}/eclipse: Extract it here if you are on Mac OS X

3. Select and download JDK 8:
   - We suggest that you download the Oracle Hotspot JDK. Hotspot is a performant JVM implementation originally built by Sun Microsystems. Now owned by Oracle, the Hotspot JRE and JDK are downloadable for free.
   - Then, choose the product corresponding to your machine through the Oracle website's link http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

   To avoid a compatibility issue later on, do stay consistent with the architecture choice (32 or 64 bits) that you have made earlier for the Eclipse archive.

4. Install JDK 8 on the operating system of your choice using the following instructions:
   - On Windows, this is a monitored installation initiated with an executable file:
     1. Execute the downloaded file and wait until you reach the next installation step
     2. On the installation-step window, pay attention to the destination directory and change it to C:\java\jdk1.8.X_XX (X_XX refers to the latest current version here. We will be using jdk1.8.0_25 in this book. Also, it won't be necessary to install an external JRE, so uncheck the public JRE feature.)

   On Linux/Mac, perform the following steps:
     1. Download the tar.gz archive corresponding to your environment
     2. Change the current directory to where you want to install Java. For easier instructions, let's agree on the /usr/java directory
     3. Move the downloaded tar.gz archive to this current directory
4. Unpack the archive with the following command line targeting the name of your archive: `tar zxvf jdk-8u25-linux-i586.tar.gz` (this example is for a binary archive corresponding to a Linux x86 machine)

You must end up with the `/usr/java/jdk1.8.0_25` directory structure that contains the `/bin, /db, /jre, /include` subdirectories.

**How it works…**

In this section we are going to provide more insights about the version of Eclipse we used and about how we chose this specific version of JVM.

**Eclipse for Java EE developers**

We have successfully installed the Eclipse IDE for Java EE developers here. Comparatively to Eclipse IDE for Java Developers, there are some additional packages coming along such as Java EE Developer Tools, Data Tools Platform, and JavaScript Development Tools. This version is appreciated for its ability to manage development servers as part of the IDE itself, its capability to customize project facets, and its ability to support JPA. The Luna version is officially Java SE 8 compatible; this has been a decisive factor at the time of writing.

**Choosing a JVM**

The choice of the JVM implementation could be discussed over performance, memory management, garbage collection, and optimization capabilities.

There are lots of different JVM implementations, including couple of open source solutions such as OpenJDK and IcedTea (RedHat). The choice of JVM really depends on the application’s requirements. We have chosen Oracle Hotspot from experience and from reference implementations deployed in production; this JVM implementation can be trusted for a wide range of generic purposes. Hotspot also behaves very well if you have to run Java UI applications. Eclipse is one of them.

**Java SE 8**

If you haven’t already played with Scala or Clojure, it is time that you took the functional programming train with Java! With Java SE 8, Lambda expressions reduce the amount of code dramatically providing improved readability and maintainability. We won’t implement this Java 8 feature, but since it is probably the most popular, it must be highlighted as it has given a massive credit to the paradigm change. It is important, nowadays, to be familiar with these patterns.

**Configuring Eclipse for Java 8, Maven 3, and Tomcat 8**

This recipe entails configuration technics to develop efficiently on Eclipse with Java, Maven, and Tomcat.
Getting ready

Once the different products are installed, there are a couple of steps that we need to follow, mainly to make Eclipse work properly with Java SE 8, Maven 3, and Tomcat 8. In this recipe, we will also look at how to customize the Eclipse configuration file (Eclipse.ini) in order to make the most of the platform that runs Java and to make sure that it will cope with any significant growth of the application.

How to do it...

Let’s take a look at the following steps to configure Eclipse on your desktop:

1. You can start by creating a shortcut on your desktop to point to the Eclipse executable:
   - On Windows, the executable file is `Eclipse.exe` and is located at the `eclipse` directory root
   - On Linux/Mac, the file is named `Eclipse` and is also located at the `eclipse` directory root

2. Then, we need to customize the `eclipse.ini` file:
   In the Eclipse directory, where you have previously extracted the Eclipse archive, you can find the `eclipse.ini` file. It is a text file that contains a few command-line options in order to control the Eclipse startup.
   - The Eclipse community recommends to specify the path to our JVM here. Hence, depending on your system, add the following two lines at the top of the file:
     For Windows, add the following:
     ```
     -vm 
     C:\java\jdk1.8.0_25\jre\bin\server\jvm.dll
     ```
     For Linux/Mac, add this:
     ```
     -vm 
     /usr/java/jdk1.8.0_25/jre/lib/your.architecture/server/libjvm.so
     ```
   The following is an optional setting that you can consider:
   - If your development machine has at least 2 GB of RAM, you can enter the following options to make Eclipse run faster than the default settings. This section is optional because Eclipse’s default settings are already optimized to suit most users’ environment:
     ```
     -vmargs
     -Xms128m
     ```
-Xmx512m
-Xverify:none
-Dosgi.requiredJavaVersion=1.6
-XX:MaxGCPauseMillis=10
-XX:MaxHeapFreeRatio=70
-XX:+UseConcMarkSweepGC
-XX:+CMSIncrementalMode
-XX:+CMSIncrementalPacing

If your machine has less than 2 GB of RAM, you can still enter this set of options without overriding the default –Xms and –Xmx arguments.

All the options under -vmargs are arguments that will be passed to the JVM at startup. It is important not to mess up the Eclipse options (the top part of the file) with the VM arguments (the bottom part).

3. After this we will go through the following steps to start Eclipse and set the workspace:

Launch the executable described in the Step 2.
- For our project, specify the path: <home-directory>/workspace

This path is different for each Operating System:
- C:\Users\{system.username}\workspace: This is the path on Windows
- /home/usr/{system.username}/workspace: This is on Linux
- /Users/{system.username}/workspace: This is on Mac OS
- Click on OK and let the Eclipse program start

The workspace is the place from where you manage your Java projects. It can be specific to one application, but not necessarily.

4. Then, we need to check the JRE definitions:

Here, a couple of settings need to be verified in Eclipse:

1. Open the Preferences menu under Window (on Mac OS X the Preference menu is under the Eclipse menu).
2. In the navigation panel on the left-hand side, open the Java hierarchy and click on Installed JREs under Java.
3. On the central screen, remove any existing JREs that you may already have.
4. Click on the Add... button to add a standard JVM.
5. Enter C:\java\jdk1.8.0_25 (or /usr/java/...) as JRE home.
6. And enter jdk1.8.0_25 as JRE name.

We tell Eclipse to use the Java Runtime Environment of JDK 8.

After completing these steps, you should end up with the following configuration:

![Preferences settings](image)

5. Now, we will check the compiler compliance level:
   1. In the navigation panel, click on Compiler under Java.
   2. Check that the Compiler compliance level is set to 1.8 in the drop-down list.

6. After this, we need to check the Maven configuration:
   1. Still in the navigation panel of the Preferences menu, open the Maven hierarchy and navigate to Maven | Installations.
   2. We will specify here which Maven installation we plan to use. For the purpose of this book, the embedded Maven will be perfect.
   3. Back in the navigation panel, go to Maven | User Settings.
   4. Set the local repository to <home-directory>/.m2/repository.

   ![Maven settings](image)

   In this local repository, our local cached versions of the required artefacts will reside. It will prevent our environment from having to download them on each build.

   5. For the User Settings field, create a settings.xml file in the .m2 directory: <home-directory>/.m2/settings.xml.
6. Edit the settings.xml file and add the following block:

   (You can also copy/paste it from the chapter_1/source_code/.m2 directory):

   ```xml
   <settings xmlns="http://maven.apache.org/SETTINGS/1.1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0
   http://maven.apache.org/xsd/settings-1.1.0.xsd">
   <profiles>
     <profile>
       <id>compiler</id>
       <properties>
         <JAVA_HOME>C:\java\jdk1.8.0_25</JAVA_HOME>
       </properties>
     </profile>
   </profiles>
   <activeProfiles>
     <activeProfile>compiler</activeProfile>
   </activeProfiles>
   </settings>

   If you are not on a Windows machine, change JAVA_HOME in this file to your JDK installation directory (/usr/java/jdk1.8.0_25).

7. Go back to the navigation panel and click on Maven. Follow the configuration given in this screenshot:

8. Click on OK to save these configuration changes.
Setup Routine for an Enterprise Spring Application

7. Now we will install Tomcat 8 in the Eclipse IDE. For this, go through these steps:
   1. Download a ZIP archive for the latest Core version of Tomcat8 from the Tomcat website: http://tomcat.apache.org/download-80.cgi.
   2. Extract the downloaded archive to the following directory:
      - On Windows, extract the archive at C:\tomcat8
      - On Linux, extract the archive at /home/usr/{system.username}/tomcat8
      - On Mac OS X, extract the archive at /Users/{system.username}/tomcat8
      Depending on your system, you must be able to access the bin directory from the hierarchy: C:\tomcat8\bin, /home/usr/{system.username}/tomcat8/bin or /Users/{system.username}/tomcat8/bin.
   3. In Eclipse, select the Preferences menu under Windows, and in the navigation panel on the left-hand side, open the Server hierarchy and then select Runtime Environments.
   4. On the central window, click on the Add... button.
   5. In the next step (the New Server environment window), navigate to Apache | Apache Tomcat v8.0.
   6. Also, check this option: Create a New Local Server.
   7. Click on the Next button.
   8. Fill in the details in the window as shown in the following screenshot:
If you are on Linux (or Mac OS X), replace C:\tomcat8 with your Tomcat installation directory.

How it works...

We are going to review in this section the different elements and concepts that this recipe took us through.

The eclipse.ini file

As we've already seen, the eclipse.ini file controls the Eclipse startup. It is an extra component that makes the Eclipse platform very configurable. You can find the list of command-line arguments that can be used in their documentation at http://help.eclipse.org/luna/topic/org.eclipse.platform.doc.isv/reference/misc/runtime-options.html

It is important to acknowledge the following warnings that this documentation mentions:

- All lines after -vmargs are passed as arguments to the JVM; all arguments and options for Eclipse must be specified before -vmargs (just like when you use arguments on the command line)

This explains why we have inserted the -vm option at the top of the file.
Setup Routine for an Enterprise Spring Application

- Any use of -vmargs on the command line replaces all -vmargs settings in the .ini file unless --launcher.appendVmargs is specified either in the .ini file or on the command line.

**Setting the –vm option**

Setting the –vm option allows us to be sure of the JVM implementation on which Eclipse runs as a program. You might have noticed that we've targeted the JVM as a library (*.dll / *.so). It has better performance on startup and also identifies the program process as the Eclipse executable and not just as the Java executable.

If you wonder which JVM Eclipse uses when a –vm option is not set, be aware that Eclipse DOES NOT consult the JAVA_HOME environment variable. (Eclipse wiki).

Instead, Eclipse executes the Java command that parses your path environment variable.

**Customizing JVM arguments**

The suggested JVM argument list comes from Piotr Gabryanczyk's work on the Java memory management model. Initially, for JetBRAINS IntelliJ settings, this configuration is also useful for an Eclipse environment. It helps in the following tasks:

- Preventing the garbage collector from pausing the application for more than 10 ms (-XX:MaxGCPauseMillis=10)
- Lowering the level from which the garbage collector starts to 30% of the occupied memory (-XX:MaxHeapFreeRatio=70)
- Imposing the garbage collector to run as a parallel thread, lowering its interference with the application (-XX:+UseConcMarkSweepGC)
- Choosing the incremental pacing mode for the garbage collector, which generates breaks in the GC job so that the application can definitely stop freezing (-XX:+CMSIncrementalPacing)

The instantiated objects throughout the program's life cycle are stored in the Heap memory. The suggested parameters define a JVM startup Heap space of 128 mb (-Xms) and an overall 512 mb maximum heap space (-Xmx). The heap is divided in two subspaces, which are as follows:

- **Young generation**: New objects are stored in this area. For the leading Hotspot or OpenJDK JVMs, the young memory space is divided in two:
  - Eden: New objects are stored in this subdivision area. Objects with short lives will be deallocated from here.
  - Survivor: This is a buffer between the young and old generation. The survivor space is smaller than the Eden and it is also divided in two (the FROM and TO areas). You can adjust the ratio between Eden and Survivor objects with -XX:SurvivorRatio (here, -XX:SurvivorRatio=10 means YOUNG = 12, EDEN = 10, FROM = 1 and TO =1).
The minimum size of the young area can be adjusted with -XX:NewSize. The maximum size can be adjusted with -XX:MaxNewSize.

- **Old generation**: When objects in Eden or Survivor spaces are still referenced after enough garbage collections, they are moved here. It is possible to set the Young area size as a ratio of the Old area size with -XX:NewRatio. (That is, -XX:NewRatio=2 means HEAP = 3, YOUNG = 1 and OLD =2).

The maximum size for the new generation space -XX:MaxNewSize must always remain smaller than half the heap space (-Xmx/2) because the garbage collector may move all the Young space to the Old space.

With Hotspot or OpenJDK, the permanent generation space was used to store information related to the classes’ definition (structure, fields, methods, and so on.). You may have already encountered a PermGen space OutOfMemoryError exception when the loaded structure becomes too big. In this situation, the solution is to increase the -XX:MaxPermSize argument. It is no longer necessary with JDK8.

For this purpose, the Permanent Generation (PermGen) space has been replaced by a metadata space that is not part of the heap but of the native memory. The default maximum size of this space is unlimited. However, we can still restrict it with -XX:MetaspaceSize or -XX:MaxMetaspaceSize.

**Changing the JDK compliance level**

Downgrading a compliance level allows us to run a lower version of a Java compiler than the one the JDK is natively identified to. It impacts the Eclipse builds, errors, and warnings and also the JavaDocs. It is obviously not possible to set a higher compilation version than the native version of a compiler.

**Configuring Maven**

Inside Eclipse, most of the Maven configuration comes from the m2eclipse plugin (also called Maven integration for Eclipse). This plugin is included, by default, in Eclipse Luna. It is then not necessary to download it manually. After the Maven configuration that we went through, m2eclipse is also very helpful to trigger Maven operations from the IDE context and to provide assistance to create Java Maven projects. You will learn more about m2eclipse in the next section.

We then installed a basic settings.xml file. This file is used to configure Maven without being bound directly to any projects. The most common uses of settings.xml are probably profile definition and credential storage to access the repository manager(s).
Setup Routine for an Enterprise Spring Application

With Maven profiles, you have the possibility to run a build for a specific environment and to match a specific configuration (variable values, set of dependencies, and so on.). Maven profiles can be cumulated with each other. They can be activated through a command line, declaratively in the Maven settings or from the environment configuration such as files being present or missing on the filesystem, the used JDK, and so on.

In our settings.xml file, we have defined a compiler profile with its own JAVA_HOME property. The compiler profile is activated by default to be declaratively defined in the <activeProfiles> section. Maven will consult the settings.xml file before looking up the system variables.

A repository manager

A repository manager is a third-party application that manages all the required binaries and dependencies that a developed application may need. Acting as a buffering proxy between development environments and public repositories, a repository manager provides control of critical parameters such as build time, availability of dependencies, visibility and access restriction, and so on.

Famous solutions include Apache Archiva, Artifactory, Sonatype Nexus. In the context of our application, we won’t make use of a repository manager.

Tomcat 8 inside Eclipse

Eclipse for JEE developers allows the integration of Tomcat with other application servers within the development environment. This is made possible through the provided Web Tools Platform (WTP) plugins that can manage web artefacts, their compilation, and their deployment into the web server.

In the servers tab (made visible earlier), double-clicking on the created Tomcat v8.0 server, opens a configuration window and enables the possibility of setting up parameters that are normally defined in the server.xml Tomcat file, which is located in the tomcat8\conf directory.

By default, WTP abstracts this configuration and doesn't impact the genuine server.xml file. This behavior can be changed by activating the Publish module contexts to separate XML files option in the Server configuration window.

There's more...

- Find out more about the Eclipse installation at http://wiki.eclipse.org/Eclipse/Installation
- Learn more about the Eclipse.ini file at http://wiki.eclipse.org/Eclipse.ini
Defining the project structure with Maven

In this recipe, we will focus on defining, with Maven, the project structure we need for our application.

Getting ready

We will initially create two Eclipse projects: one for the application and one for the components that ZipCloud as a company could share later on with other projects. Take a look at the following image which presents the project components that we are going to build:

The application project cloudstreetmarket-parent will have three modules. Two of them will be packaged as web archives (war): the main web application and the REST API. One of them will be packaged as a jar dependency (cloudstreetmarket-core).
The company-specific project **zipcloud-parent** will have only one submodule—**zipcloud-core**, which will be packaged as **jar**.

**How to do it...**

The following steps will help us create a Maven parent project:

1. From Eclipse, navigate to **File | New | Other**.
2. A **New** wizard opens up wherein you can select the type of project within a hierarchy. Then, open the **Maven** category, select **Maven Project**, and click on **Next**.

   The New Maven Project wizard opens as shown in the following screenshot:

   ![New Maven Project Wizard Screenshot](image)

3. Make sure to check the **Create a simple project** option. Click on **Next**.
4. Fill up the next wizard as follows:

   - `edu.zipcloud.cloudstreetmarket` as **Group Id**
   - `cloudstreetmarket-parent` as **Artifact Id**
   - `0.0.1-SNAPSHOT` as **Version**
   - `pom` as **Packaging**
   - `CloudStreetMarket Parent` as **Name**

   Then, click on the **Finish** button

   The parent project must appear in the package explorer on the left-hand side of the dashboard.

   ![Package Explorer Screenshot](image)
We now have to tell m2eclipse which Java compiler version you plan to use in this project so that it automatically adds the right JRE system library to the submodules we are about to create. This is done through the `pom.xml` file.

5. Edit `pom.xml` file to specify the Java compiler version:
   - Double-click on the `pom.xml` file. The m2eclipse Overview tab shows up by default. You have to click on the last tab, `pom.xml`, to access the full XML definition.
   - In this definition, add the following block at the end but still as part of the `<project>` node. (You can also copy/paste this piece of code from the `cloudstreetmarket-parent`'s `pom.xml` of the `chapter_1` source code):

   ```xml
   <build>
   <plugins>
   <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.1</version>
   <configuration>
   <source>1.8</source>
   <target>1.8</target>
   <verbose>true</verbose>
   <fork>true</fork>
   <executable>${JAVA_HOME}/bin/javac</executable>
   <compilerVersion>1.8</compilerVersion>
   </configuration>
   </plugin>
   <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <version>2.4.2</version>
   <configuration>
   <jvm>${JAVA_HOME}/bin/java</jvm>
   <forkMode>once</forkMode>
   </configuration>
   </plugin>
   </plugins>
   </build>
   
   You have probably noticed the `maven-surefire-plugin` declaration as well. We will review it soon; it allows us to run unit tests during the build.
6. Now, we will create submodules:

As submodules of the Parent project, we have seen that we needed one web module to handle and render the site's screens, one web module for the REST API, and one other module that will be used to package all the business logic (services, data access, and so on.) specific to the first product cloudstreetmarket.com:

1. From the main Webapp module in Eclipse, navigate to File | New | Other. A New wizard opens up through which you can select the type of project within a hierarchy. Open the Maven category, select Maven Module, and click on Next.

2. The New Maven Module wizard opens up after this; fill it up as follows:

   - Check Create a simple project.
   - Enter cloudstreetmarket-webapp as Module Name.
   - Enter cloudstreetmarket-parent as Parent project.

3. Click on the Next button after which the next step shows up. Enter the following entries in that new window:

   - Enter edu.zipcloud.cloudstreetmarket as Group Id.
   - Enter 0.0.1-SNAPSHOT as Version.
   - Select war as Packaging.
   - Enter CloudStreetMarket Webapp as Name.

Then click on the Finish button.
7. Now we will go ahead to create and REST API module:

We are going to repeat the previous operation with different parameters.

1. From Eclipse, navigate to File | New | Other. The selection wizard pops up when you go there. After this, open the Maven category, select Maven Module, and click on Next:

2. In the New Maven Module wizard, enter the following entries:
   
   - Check the Create a simple project option.
   - Enter cloudstreetmarket-api as Module Name.
   - Enter cloudstreetmarket-parent as Parent project.

3. Click on the Next button to proceed to the next step. Enter the following entries in that window:
   
   - Enter edu.zipcloud.cloudstreetmarket as Group Id.
   - Enter 0.0.1-SNAPSHOT as Version.
   - Select war as Packaging.
   - Enter CloudStreetMarket API as Name.
   
   Then click on the Finish button.

8. Now, we will create the core module:

For this, navigate to File | New | Other. The selection wizard pops up when you do so. Open the Maven category, select Maven Module, and click on Next.

1. In the New Maven Module wizard, enter the following entries:
   
   - Check the Create a simple project option.
   - Enter cloudstreetmarket-core as Module Name.
   - Enter cloudstreetmarket-parent as Parent project.

2. Click on the Next button to go to the next step. Fill in the fields with the following:
   
   - Enter edu.zipcloud.cloudstreetmarket as Group Id.
   - Enter 0.0.1-SNAPSHOT as Version.
   - This time, select jar as Packaging.
   - Enter CloudStreetMarket Core as Name.
   
   Then click on the Finish button.
If you have the Java perspective activated (in the top-right corner), you should see the overall created structure matching the screenshot here:

9. Now, we will create a company-specific project and its module(s):

Let’s assume that many different categories of dependencies (core, messaging, reporting, and so on...) will be part of the company-business project later.

1. We need a parent project, so from Eclipse, navigate to File | New | Other. The selection wizard pops up. Open the Maven category, select Maven Project, and click on Next.

2. In the first step of the New Maven Project wizard, as for the Parent project we created earlier, only check the Create a simple Project and Use default workspace location options.

3. Click on the Next button and fill in the next wizard as follows:
   - Enter edu.zipcloud as Group Id.
   - Enter zipcloud-parent as Artifact Id.
   - Enter 0.0.1-SNAPSHOT as Version.
   - Select pom as Packaging.
   - Enter ZipCloud Factory Business Parent as Name.
Chapter 1

Again, in the created `pom.xml` file, add the following block inside the `<project>` node to create the underlying modules properly and to enable automatic test execution. (You can also copy/paste this piece of code from the zipcloud-parent’s `pom.xml` file of the chapter_1 source code):

```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <verbose>true</verbose>
        <fork>true</fork>
        <executable>${JAVA_HOME}/bin/javac</executable>
        <compilerVersion>1.8</compilerVersion>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.4.2</version>
      <configuration>
        <jvm>${JAVA_HOME}/bin/java</jvm>
        <forkMode>once</forkMode>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Now, we are going to create one company-business core module, which will be a sub module of the parent project we just created.

For this, navigate to File | New | Other. The selection wizard pops up. Open the Maven category, select Maven Module, and click on Next.

1. In the New Maven Module wizard, enter the following details:
   - Check the Create a simple project option.
   - Enter zipcloud-core as Module Name.
   - Enter zipcloud-parent as Parent project.
2. Click on the Next button and go to the next step. Here, enter the following details:
   Enter edu.zipcloud as Group Id.
   Enter 0.0.1-SNAPSHOT as Version.
   Select jar as Packaging.
   Select ZipCloud Factory Core Business as Name.

10. Now, build the two projects:
   If the structure is correct, the following Maven command could be successfully run:

   mvn clean install

   This command can be launched in the terminal if Maven is installed on the development machine.

   In our study case, we will, for now, launch it using the m2eclipse modified Run As menu: Right click on the zipcloud-parent project and click on Run As | Maven Clean.

   In the Maven console, you should now see this beautiful line at the bottom:
   [INFO] BUILD SUCCESS

   Now, repeat the operation for the install build phase. You should now see the following output in the console:

   [INFO] ZipCloud Parent .......................SUCCESS [ 0.313 s]
   [INFO] ZipCloud Core .........................SUCCESS [ 1.100 s]
   [INFO] Build SUCCESS
   [INFO] BUILD SUCCESS

   Ok, now you should be able to build cloudstreetmarket-parent as well.
   For this, right-click on the cloudstreetmarket -parent project and click on Run As | Maven Clean. The Maven console should print the following after this step:

   [INFO] BUILD SUCCESS

   Again, right-click on the cloudstreetmarket -parent project and click on Run As | Maven Install. The Maven console should now print the following:

   [INFO] CloudStreetMarket Parent ...............SUCCESS [ 0.313 s]
   [INFO] CloudStreetMarket Webapp ...............SUCCESS [ 6.129 s]
[INFO] CloudStreetMarket Core .................SUCCESS [  0.922 s]
[INFO] CloudStreetMarket API ..................SUCCESS [  7.163 s]
[INFO] ----------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ----------------------------------------------------------

Scrolling up a bit should display the following trace:

-----------------------------------------------
TESTS
-----------------------------------------------

There are no tests to run.
Results :
Tests run: 0, Failures: 0, Errors: 0, Skipped: 0

Maven here, with the help of the maven-surefire-plugin, which we manually added, parses all the classes encountered in the src/test/java directories. Again, this path can be customized.

In the detected test classes, Maven will also run the methods annotated with the JUnit @Test annotation. A JUnit dependency is required in the project.

How it works...

In this section, we are going through quite a few concepts about Maven so that you can better understand its standards

New Maven project, new Maven module

The project creation screens we just went through also come from the m2eclipse plugin. These screens are used to initialize a Java project with a preconfigured pom.xml file and a standard directory structure.

The m2eclipse plugin also provides a set of shortcuts to run Maven build phases and some handy tabs (already seen) to manage project dependencies and visualize the pom.xml configuration.

The standard project hierarchy

Navigating through the created projects, you should be able to notice a recurring hierarchy made of the following directories: src/main/java, src/main/resource, src/test/java, and src/test/resource. This structure is the default structure that Maven drives us through. This model has become a standard nowadays. But, we can still override it (in the pom.xml files) and create our own hierarchy.
If you remember the maven-compiler-plugin definition added in the pom.xml files of the parent projects, there were the following four lines of code that we used:

```xml
<verbose>true</verbose>
<fork>true</fork>
<executable>${JAVA_HOME}/bin/javac</executable>
<compilerVersion>1.8</compilerVersion>
```

These lines allow Maven to use an external JDK for the compiler. It is better to have control over which compiler Maven uses, especially when managing different environments.

Also, there were the following two lines that might look like an over configuration:

```xml
<source>1.8</source>
<target>1.8</target>
```

From a strict Maven point of view, these lines are optional when an external JDK is defined with a specified compilerVersion. Initially, with these two lines, we can control which Java version we want the default code to be compiled in. When maintaining older systems, the existing code might still compile in a previous version of Java.

Actually, m2eclipse specifically expects these two lines in order to add JRE System Library [JavaSE-1.8] to the build path of the jar and war modules. Now, with these lines, Eclipse compiles these projects in the same way Maven does: in Java SE 8.

If this dependency still shows up as a different version of Java, you may need to right-click on the module and then navigate to Maven | Update Project.

The project's structure in the IDE

About the parent projects in the Eclipse project hierarchy; did you notice that the created submodules seem duplicated as standalone projects and as direct children of the parent? This is due to the fact that Eclipse doesn't handle hierarchies of projects yet in Luna. For this reason, the modules appear as separated projects. It might be slightly confusing because the source code appears to be located beside the parent projects. This is not the case in reality, it is only the way they are rendered, so we can have all the tools normally bound to the project level.

At this time, JetBRAINS IntelliJ IDEA already supports visual hierarchies of the projects.

Finally, if you open a parent project's pom.xml file, you should see the <modules> node populated with the created submodules. This has been done automatically as well by m2eclipse. We recommend that you keep an eye on this feature because m2eclipse doesn't always update these <modules> nodes depending on which way you alter the project hierarchy.
Maven's build life cycles

A build life cycle in Maven is a specific sequence (and a group) of predefined operations called phases. There are three existing life cycles in Maven: default, clean, and site.

Let's have a look at all the phases that include the default and clean life cycles (probably the life cycles the most commonly used by developers).

The clean life cycle

The Maven clean phase plays a central role. It resets a project build from Maven's perspective. It is usually about deleting the target directory that is created by Maven during the build process. Here are some details about the phases included in the clean life cycle. These details come from the Maven documentation:

<table>
<thead>
<tr>
<th>Phases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre-clean</td>
<td>This executes processes that are needed prior to the actual project cleaning</td>
</tr>
<tr>
<td>clean</td>
<td>This removes all files generated by the previous build</td>
</tr>
<tr>
<td>post-clean</td>
<td>This executes processes that are needed to finalize the project cleaning</td>
</tr>
</tbody>
</table>

The default life cycle

In the default life cycle, you can find the most interesting build phases that deal with source generation, compilation, resource handling, tests, integration tests, and artefact deployment. Here are some details about the phases included in the default life cycle:

<table>
<thead>
<tr>
<th>Phases</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>validate</td>
<td>This validates whether the project is correct and all necessary information is available.</td>
</tr>
<tr>
<td>initialize</td>
<td>This initializes the build state, for example, setting properties or creating directories.</td>
</tr>
<tr>
<td>generate-sources</td>
<td>This generates source code for inclusion in compilation.</td>
</tr>
<tr>
<td>process-sources</td>
<td>This processes the source code, for example, to filter any values.</td>
</tr>
<tr>
<td>generate-resources</td>
<td>This generates resources to be included in the package.</td>
</tr>
<tr>
<td>process-resources</td>
<td>This copies and processes the resources into the destination directory, which is ready for packaging.</td>
</tr>
<tr>
<td>compile</td>
<td>This compiles the source code of the project.</td>
</tr>
<tr>
<td>process-classes</td>
<td>This post processes the generated files from compilation, for example, to perform bytecode enhancement on Java classes.</td>
</tr>
<tr>
<td>generate-test-sources</td>
<td>This generates any test source code to be included in compilation.</td>
</tr>
</tbody>
</table>
**Phases** | **Descriptions**
---|---
process-test-sources | This processes the test source code, for example, to filter any values.
generate-test-resources | This creates resources for testing.
process-test-resources | This copies and processes the resources into the test destination directory.
test-compile | This compiles the test source code into the test destination directory.
process-test-classes | This post processes the generated files from test compilation, for example, to perform bytecode enhancement on Java classes. For Maven 2.0.5 and above.
test | This runs tests using a suitable unit testing framework. These tests should not require the code to be packaged or deployed.
prepare-package | This performs the operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. (Maven 2.1 and above)
package | This takes the compiled code and packages it in its distributable format, such as JAR.
pre-integration-test | This performs actions required before integration tests are executed. This may involve things such as setting up the required environment.
integration-test | This processes and deploys the package if necessary into an environment where integration tests can be run.
post-integration-test | This performs the actions required after integration tests have been executed. This may include cleaning up the environment.
verify | This runs checks to verify that the package is valid and meets the quality criteria.
install | This installs the package into the local repository to be used as a dependency in other projects locally.
deploy | This copies the final package to the remote repository to share it with other developers and projects (done in an integration or release environment).

**Plugin goals**
With the concept of plugins, Maven acquires a much wider dimension. Maven natively provides built-in plugins, but external plugins can be introduced just as other dependencies (identified by groupIds and artefactIds).

Each build phase can be attached to zero, one, or more plugin goals. A goal represents a specific and concrete task responsible for building or handling a project in a certain manner. Some phases have goals bound to them, by default, through native plugins.
Built-in life cycle bindings

Now that we have seen the purpose of each phase in the presented two life cycles, we must say that, for the default life cycle, depending upon which module packaging type we are choosing, only some of these phases are potentially activated for goal execution.

Let's see the phases that we skipped in the default life cycle for different packaging types:

<table>
<thead>
<tr>
<th>Default life cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packaging type</td>
</tr>
<tr>
<td>Activated phases</td>
</tr>
<tr>
<td>package</td>
</tr>
<tr>
<td>install</td>
</tr>
<tr>
<td>deploy</td>
</tr>
</tbody>
</table>

In Chapter 9, *Testing and Troubleshooting*, we will practically bind external plugins goals to identified build phases.

In summary, calling: `mvn clean install` on a jar packaged-module will result in executing the following phases: clean, process-resources, compile, process-test-resources, test-compile, test, package, and install.

About Maven commands

When Maven is told to execute one or more phases targeting a specific project's `pom.xml` file, it will execute the requested phase(s) for each of its modules.

Then, for every single requested phase, Maven will do the following:

- Identify which life cycle the phase belongs to
- Look for the packaging of the current module and identify the right life cycle binding
- Execute all the phases in the hierarchy of the identified life cycle bindings, which are located before the requested phase in the hierarchy
By the term execute all the phases, we mean execute all the underlying detected and attached plugin goals (native plugins or not).

In summary, calling `mvn clean install` on a jar packaged module will execute the following phases: `clean`, `process-resources`, `compile`, `process-test-resources`, `test-compile`, `test`, `package`, and `install`.

There's more...

You may wonder why we have created these projects and modules in regard to our application.

**How did we choose the jar module’s name?**

About the Maven structure, the best names for nondeployable modules often emphasize a functional purpose, a specific concept created by the business, or are driven by the product (cloudstreetmarket-chat, cloudstreetmarket-reporting, cloudstreetmarket-user-management, and so on.). This strategy makes the dependency management easier because we can infer whether a new module requires another module or not. Thinking about controllers, services, and DAO layers at a macro scale doesn't really make sense at this stage, and it could lead to design interference or circular dependencies. These technical subcomponents (service, DAO, and so on) will be present or not, as needed, in each functional module as Java packages but not as JAR-packaged dependencies.

**How did we choose the names for deployable modules?**

Choosing a name for a deployable module (war) is a bit different from choosing a name for a JAR-packaged module. The deployable archive must be thought of as scalable and potentially load balanced. It is fair to assume that the requests that will target the application to retrieve HTML contents can be distinguished from the ones that will return REST contents.

With this assumption, in our case it has been our wish to split the war into two. Doing so may raise the question of how the web sessions are maintained between the two webapps. We will answer this point later on.

**Why did we create core modules?**

We created the core modules, firstly, because it is certain that, in the cloudstreetmarket application and also in the company-shared project, we will have POJOs, exceptions, constants, enums, and some services that will be used horizontally by almost all the modules or applications. If a concept is specific to a created functional module, it must not be part of core modules.
Then, it is probably better to start big grained to refine later rather than thinking about modules that may be implemented differently or even not implemented at all. In our case, we are a start-up, and it is not silly to say that the 5 to 10 features we are going to implement can constitute the core business of this application.

See also...

- We also recommend that you install Code Style Formatters. Triggered from the Save Event, we have, with these formatters, the ability to restyle our code automatically with a uniform predefinition. Having such formatters in a team is much appreciated since it guarantees the same rendering while comparing two files with a versioning tool.

## Installing Spring, Spring MVC, and a web structure

In this recipe, we will add third-party dependencies to our `pom.xml` files using inheritance. We will load Spring application contexts and create the first controller of our application. Finally, we will deploy and start the web app in Tomcat.

### Getting ready

Now that we have Eclipse ready and Maven configured properly, the fun can begin. We need to specify all the necessary Spring dependencies in our `pom.xml` files, and we need to set up Spring so that it loads and retrieves its context for every module.

We also need to organize and optionally expose web resources, such as JSPs, JavaScript files, CSS files, and so on. If you've completed this configuration, we should end up with a static welcome page provided by the Tomcat server, started without exceptions!

### How to do it...

Our first set of changes relate to the parent projects:

1. We will define dependencies and build options for those parent projects. Let's do it with the following steps:

   1. Open the cloudstreetmarket-parent `pom.xml` from the `chapter_1` source code directory and select the `pom.xml` tab (underneath the main window).

      Copy and paste into the cloudstreetmarket-parent's `pom.xml` file the `<properties>`, `<dependencyManagement>`, and `<build>` blocks.

      Now, repeat the operation for zipcloud-parent.

   2. Open the zipcloud-parent's `pom.xml` file from the `chapter_1` source code and click on the `pom.xml` tab.
3. Copy and paste into your zipcloud-parent's pom.xml the <properties> and <dependencyManagement> blocks. You should already have copied over the <build> section in the third recipe.

2. Now, we will define dependencies and build options for web modules:

   1. Open the cloudstreetmarket-api's pom.xml from the chapter_1 source code and select the pom.xml tab.
   2. Copy and paste into your cloudstreetmarket-api's pom.xml the <build> and <dependencies> blocks.
   3. Now, repeat the operation for cloudstreetmarket-webapp.
   4. Open the cloudstreetmarket-webapp's pom.xml from the chapter_1 source code directory and click on the pom.xml tab.
   5. Copy and paste into your cloudstreetmarket-webapp's pom.xml file the <build> and <dependencies> blocks.

3. After this, we define dependencies for jar modules:

   1. Open the cloudstreetmarket-core's pom.xml from the chapter_1 source code and click on the pom.xml tab.
   2. Copy and paste into your cloudstreetmarket-core's pom.xml the entire <dependencies> block.

4. Then, we place the web resources:

   1. From the chapter_1 source code, copy and paste the entire src/main/webapp/* directory into your cloudstreetmarket-webapp project. You need to end up with the same webapp directory structure as the chapter_1 source code:

```
> src
  > main
    > java
    > resources
  > webapp
    > static
      > index.html
  > WEB-INF
    > jsp
      > index.jsp
      > dispatcher-context.xml
      > web.xml
```
2. Now, perform the same operation for `cloudstreetmarket-api`. Copy and paste from the `chapter_1` source code the entire `src/main/webapp/*` branch into your `cloudstreetmarket-api` project. You need to end up with the same webapp node and children as the `chapter_1` source code:

```
src
  main
    java
    resources
  webapp
    WEB-INF
      dispatcher-context.xml
      web.xml
  test
```

5. Now, we target a runtime for the web modules:
   1. In Eclipse, right-click on the `cloudmarket-api` project.
   2. Select the `Properties` menu.
   3. On the navigation panel, select `Targeted Runtimes`.
   4. On the central window, check the `Server Apache Tomcat v8.0` option.
   5. Click on `OK` and repeat the fifth operation on `cloudstreetmarket-webapp`.

A few Eclipse warnings in the `index.jsp` files must have disappeared after this.

If you still have Warnings in the project, your Eclipse Maven configuration may be out of synchronization with the local repository

6. This step should clean your existing project warnings (if any):
Setup Routine for an Enterprise Spring Application

In this case, perform the following steps:

1. Select all the projects in the project hierarchy, except the servers, as follows:

   ![Projects in Project Hierarchy]

   - cloudstreetmarket-api
   - cloudstreetmarket-core
   - cloudstreetmarket-parent
   - cloudstreetmarket-webapp
   - Servers
   - zipcloud-core
   - zipcloud-parent

2. Right-click somewhere in the selection and click on Update Project under Maven. The Warnings window at this stage should disappear!

7. Let's deploy the wars and start Tomcat:

   Add the servers view in Eclipse. To do so, perform the following operations:

   1. Navigate to Window | Show view | Other.
   2. Open the Server directory and select servers. You should see the following tab created on your dashboard:

   ![Server Dashboard]

   - Tomcat v8.0 Server at localhost [Stopped]

8. To deploy the web archives, go through the following operations:

   1. Inside the view we just created, right-click on the Tomcat v8.0 Server at localhost server and select Add and Remove....
   2. In the next step, which is the Add and Remove window, select the two available archives and click on Add and then on Finish.

9. To start the application in Tomcat, we need to complete these steps:

   1. In the Servers view, right-click on the Tomcat v8.0 Server at localhost server and click on Start.
   2. In the Console view, you should have the following at the end:

      INFO: Starting ProtocolHandler [*http-nio-8080*]
      INFO: Starting ProtocolHandler [*ajp-nio-8009*]
      INFO: Server startup in 6898 ms
Finally, if you try to reach http://localhost:8080/portal/index.html with your browser, you should receive the following HTML content:

![Image of a browser window displaying a message: Well done! You've completed the first chapter :)](image)

A static access to an HTML page remains a modest visual achievement for this chapter. All along this book, you will discover that we haven't diminished the importance of the environment and the context Spring MVC acts in.

**How it works...**

Through this recipe, we have been moving across web resources and Maven dependencies related to Spring, Spring MVC, and the web environment. Now, we will go through the way that Maven dependency and plugin management are performed. We will then talk about the Spring web application context and finally about the organization and packaging of web resources.

**Inheritance of Maven dependencies**

There are two strategies concerning the inheritance of dependencies between parent projects and children modules. They both are implemented from the parent project. On the one hand, we can choose to define these dependencies directly from the `<dependencies>` node, shaping a basic inheritance in this way. On the other hand, to set up a managed inheritance, we can define the `<dependencies>` node as a child node of `<dependencyManagement>`. Let's have a look at the differences between the two.
Setup Routine for an Enterprise Spring Application

**Basic inheritance**

With a basic inheritance, all the dependencies specified in the parent's `pom.xml` file are automatically inherited into the child module with the same attributes (scope, version, packaging type, and so on) unless you override them (redefining these dependencies with the same couple `groupId/artifactId`).

On the one hand, it provides the option of using the versions of the dependencies we want in the modules we want. On the other hand, we can end up with a very complex dependencies schema and huge `pom.xml` files in the children modules. Also, managing version conflicts with external transitive dependencies can be a pain.

A transitive dependency is a required dependency with the needed dependency. Transitive dependencies have been automatically imported since Maven 2.0.

There are no standards in this inheritance type for external dependencies.

**Managed inheritance**

With the `<dependencyManagement>` mechanism, dependencies defined in the parent `pom.xml` are not automatically inherited in children modules. However, the dependency attributes (scope, version, packaging type, and so on) are pulled from the parent dependency's definition, and therefore, the redefinition of these attributes is made optional.

This process drives us towards a centralized dependency definition where all the children modules use the same versions of dependencies unless a specific dependency requires a custom one.

**Including third-party dependencies**

Among the dependencies copied over, you might have noticed a few Spring modules, some test, web, logging, and utility dependencies.

The idea has been to start with a basic web development tool box, which is enhanced with all the Spring modules. We will visit most of the dependencies actually included when we face a particular situation.
The Spring Framework dependency model
As presented in the following diagram taken from the spring.io website, these days, the Spring Framework is currently made of 20 modules that are grouped in different areas:

These modules have been included in the parent POMs as managed dependencies. This will allow us, later on, to quickly cherry-pick the needed ones, narrowing down a selection for our wars.

The Spring MVC dependency
The Spring MVC module is self-contained in the spring-webmvc jar. Spring MVC in a web application is a fundamental element, as it handles incoming client requests and smoothly monitors the business operations from controllers. It finally offers a number of tools and interfaces capable of preparing responses in the format the clients expect them in.

All this workflow comes along with the spring-webmvc jar output HTML content or web services.

Spring MVC is entirely integrated in the Spring Framework, and all its components are standard with regard to the Spring architecture choices.

Using Maven properties
In each parent pom.xml file, we have defined a <properties> block as part of the <project> section. These properties are user-defined properties bound to a project, but we can also define such properties within a Maven Profile option. Like variables, properties are referenced in the POMs with their name surrounded by $[...].
Setup Routine for an Enterprise Spring Application

There is a standard on defining property names using periods as word separators. More than a standard, it is a uniform notation to access both user-defined variables and attributes of objects that constitute the Maven model. The Maven model is the public interface of Maven and starts from the project level.

The POM XML Schema Definition (xsd) is generated from this Maven model. It can sound abstract but in the end, the Maven model is only a set of POJOs with getters and setters. Have a look at the JavaDoc of the Maven model from the URL below, to identify concepts, specific to pom.xml files (Build, Dependency, Plugin, and so on.):


To summarize, we can retrieve a node value defined in a POM and navigate the Maven model hierarchy using a period-based expression language that targets the getters.

For example, ${project.name} references the current project.getName(), ${project.parent.groupId}, the current project.getParent().getGroupId(), and so on.

Defining user properties that match an existing path of the Maven model is a way of overriding its value. That's what we have done for project.build.sourceEncoding.

Maven also offers the possibility to reach properties defined in the settings.xml files such as ${settings.localRepository}; but also environment variables such as ${env.JAVA_HOME}; and Java System properties such as ${java.class.path},${java.version},${user.home},${user.name}.

The web resources

If you remember, we copied/pasted the entire src/main/webapp directory from the chapter_1 source code. The webapp directory name is a Maven standard. The webapp folder in Eclipse doesn't need to be tagged as a source folder for the build path, as it would create a complex and useless package hierarchy for static files. Preferably, it appears as a plain directory tree.

The webapp directory must be seen as the document root of the application and positioned at the root level of the WAR. The public static web resources under webapp, such as HTML files, Javascript, CSS, and image files, can be placed in the subdirectories and structure of our choice. However, as described in the Servlet 3.0 Specification, the WEB-INF directory is a special directory within the application hierarchy. All its contents can never be reached from outside the application; its content is accessible from the servlet code calling for getResource or getResourceAsStream on ServletContext. The specification also tells us that the content of a WEB-INF directory is made up of the following:

- The /WEB-INF/web.xml deployment descriptor.
- The /WEB-INF/classes/ directory for servlet and utility classes. The classes in this directory must be available to the application class loader.
The /WEB-INF/lib/*.jar area for Java ARchive files. These files contain servlets, beans, static resources, and JSPs packaged in a JAR file and other utility classes useful to the web application. The web application class loader must be able to load classes from any of these archive files.

It is good practice to create a jsp directory inside the WEB-INF folder so that the jsp files cannot be directly targeted without passing through an explicitly defined controller.

JSP applications do exist, and by definition, they will not follow this practice. These type of applications may be suited to certain needs, but they also don't specifically promote the use of an MVC pattern nor a great separation of concerns.

To use JSPs in a web application, the feature must be enabled in web.xml with the definition of a servlet of the org.apache.jasper.servlet.JspServlet type mapped to the JSP files location.

**The target runtime environment**

We have experienced warnings in the index.jsp files. We have sorted them out by adding a target runtime to our projects. We also saw that Tomcat comes with the Eclipse Compilator for Java as a JAR library. To perform the JSP compilation, the tomcat8\lib directory must include the following JAR libraries: jsp-api, servlet-api and el-api, and so on. Specifying a target runtime for a project in Eclipse emulates and anticipates situation where the application will be run from an external Tomcat container (setup with those libraries). This also explains why the jsp-api and el-api dependencies are defined in the parent POMs with a provided scope.

**The Spring web application context**

In the web.xml files, we defined a special type of Servlet, the Spring MVC DispatcherServlet, and we named it spring. This servlet covers the widest /* URL pattern. We will revisit the DispatcherServlet in the next chapter.

A DispatcherServlet has its own discovery algorithm that builds up WebApplicationContext. An optional contextConfigLocation initialization parameter is provided that points to a dispatcher-context.xml file. This parameter overrides the default expected filename and path (/WEB-INF/{servletName}-servlet.xml) for the WebApplicationContext defined in the DispatcherServlet discovery logic.

With the load-on-startup attribute set to 1, as soon as the servlet container gets ready, a new WebApplicationContext gets loaded and scoped only for the starting servlet. Now, we don't wait for the first client request to load WebApplicationContext.

A Spring WebApplicationContext file usually defines or overrides the configuration and beans that Spring MVC offers to the web application.

Still in the web.xml file, an org.sfw.web.context.ContextLoaderListener listener is set up. The purpose of this listener is to start and shut down another Spring ApplicationContext, which will be the root one following the container's life cycle.
To load more than one spring context file easily, the trick here is to use the classpath notation (which is relative) and the star (*) character in the resource path:

```xml
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath*:META-INF/spring/*-config.xml</param-value>
</context-param>
```

Doing so allows us to **load all the context files encountered in the classpath that match a standard notation and location**. This approach is appreciated for the consistency it imposes but also for the way it targets context files in underlying jars.

The aggregation of all the matching context files creates an `ApplicationContext` root with a much broader scope, and the `WebApplicationContext` inherits it. The beans we define in the root context become visible to the `WebApplicationContext` context. We can override them if needed. However, the `DispatcherServlet` context's beans are not visible to the root context.

**Plugins**

Maven is, above all, a plugin's execution framework. Every task run by Maven corresponds to a plugin. A plugin has one or more goals that are associated individually to life cycle phases. Like the dependencies, the plugins are also identified by a `groupId`, an `artifactId`, and a version. When Maven encounters a plugin that is not in the local repository, it downloads it. Also, a specific version of Maven targets, by default, a number of plugins that match the life cycle phases. These plugins are frozen on fixed versions and therefore on a defined behavior—you need to override their definition to get a more recent version or to alter their default behavior.

**The Maven compiler plugin**

The `maven-compiler-plugin` is a Maven core plugin. The core plugins are named as such because their goals are triggered on Maven core phases (clean, compile, test, and so on.). Noncore plugins relate to packaging, reporting, utilities, and so on. It is good practice to redefine the maven-compiler-plugin to control which version of the compiler is to be used or to trigger some external tools' actions (the m2eclipse project management tool, actually).

As its name suggests, the maven compiler plugin compiles the Java sources. For that, it uses the `javax.tools.JavaCompiler` class and has two goals: `compiler:compile` (triggered as part of the compile phase to compile the `java/main` source classes) and `compiler:testCompile` (triggered as part of the test-compile phase to compile the `java/test` source classes).

**The Maven surefire plugin**

The `maven-surefire-plugin` is also a Maven core plugin that has only one goal: `surefire:test`. This is invoked as part of the default life cycle (the test phase) to run unit tests defined in the application. It generates reports (*.txt or *.xml), by default, under the `${basedir}/target/surefire-reports` location.
The Maven enforcer plugin

The maven-enforcer-plugin is very useful to define environmental conditions as critical for the project. It has two goals: `enforcer:enforce` (bound, by default, to the validate phase, where it executes each defined rule once per module) and `enforcer:display-info` (it displays the detected information on execution of the rules).

The most interesting standard rule is probably `DependencyConvergence`: it analyzes all the used dependencies (direct and transitive) for us. In case of divergence of a version, it highlights it and stops the build. When we face this kind of conflict, it is amazingly easy to decide between the following:

- Excluding the lowest version from the classpath
- Not upgrading the dependency

We also quickly talked about the `<pluginManagement>` section, which was associated to the maven-enforcer-plugin. In this case, this is because m2eclipse doesn't support this plugin. Thus, to avoid a warning in Eclipse, it is necessary to add this section so that m2eclipse skips the enforce goal.

The Maven war plugin

Using the maven-war-plugin, we redefined in our web POMs. We have again overridden the default behavior of this plugin that is used to package web modules. This is definitely necessary if you have a non-Maven standard project structure.

We may want to package our web resources in a different way that how it is organized in our IDE. We may need, for some reason, to exclude some resources from the war packaging or we may even want to give a name to the built war so that it can be used by the servlet container that matches a specific context path in the application URLs (`/api`, `/app`, and so on). Filtering, moving web resources around, and managing the generated war is the purpose of this plugin.

By default, the web resources are copied to the WAR root. To override the default destination directory, specify the target path `*`.

There's more...

This has been quite a broad overview about concepts that naturally require deeper interest:

- About the way Maven manages its dependencies, we would suggest you to go through the Maven documentation on this topic at:
  
  http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html
Setup Routine for an Enterprise Spring Application

- The Maven model API documentation can again be found at: http://maven.apache.org/ref/3.0.3/maven-model/apidocs/index.html
- Concerning the servlet 3.0 specification that we mentioned earlier, more information can be found about the web.xml file definition and about the structure of a WebArchive at: http://download.oracle.com/otn-pub/jcp/servlet-3.0-fr-eval-oth-JSpec/servlet-3_0-final-spec.pdf
- Finally, for more information about Maven plugins; we absolutely recommend you visit the Maven listing at http://maven.apache.org/plugins

See also

- The spring.io website from Pivotal, and especially the Spring Framework overview page, can also refresh, or introduce a few key concepts. Follow the address: http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/overview.html

The Maven checkstyle plugin

One other interesting enough plugin which could also be highlighted here is the maven-checkstyle-plugin. When a team is growing, we sometimes need to guarantee the maintenance of certain development practices or we may need to maintain specific security-related coding practices. Like the maven-enforcer-plugin, the maven-checkstyle-plugin makes our builds assertive against this type of violation.

Find out more about this plugin, again in the Maven documentation, at: http://maven.apache.org/plugins/maven-checkstyle-plugin.
Where to buy this book

You can buy Spring MVC Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.