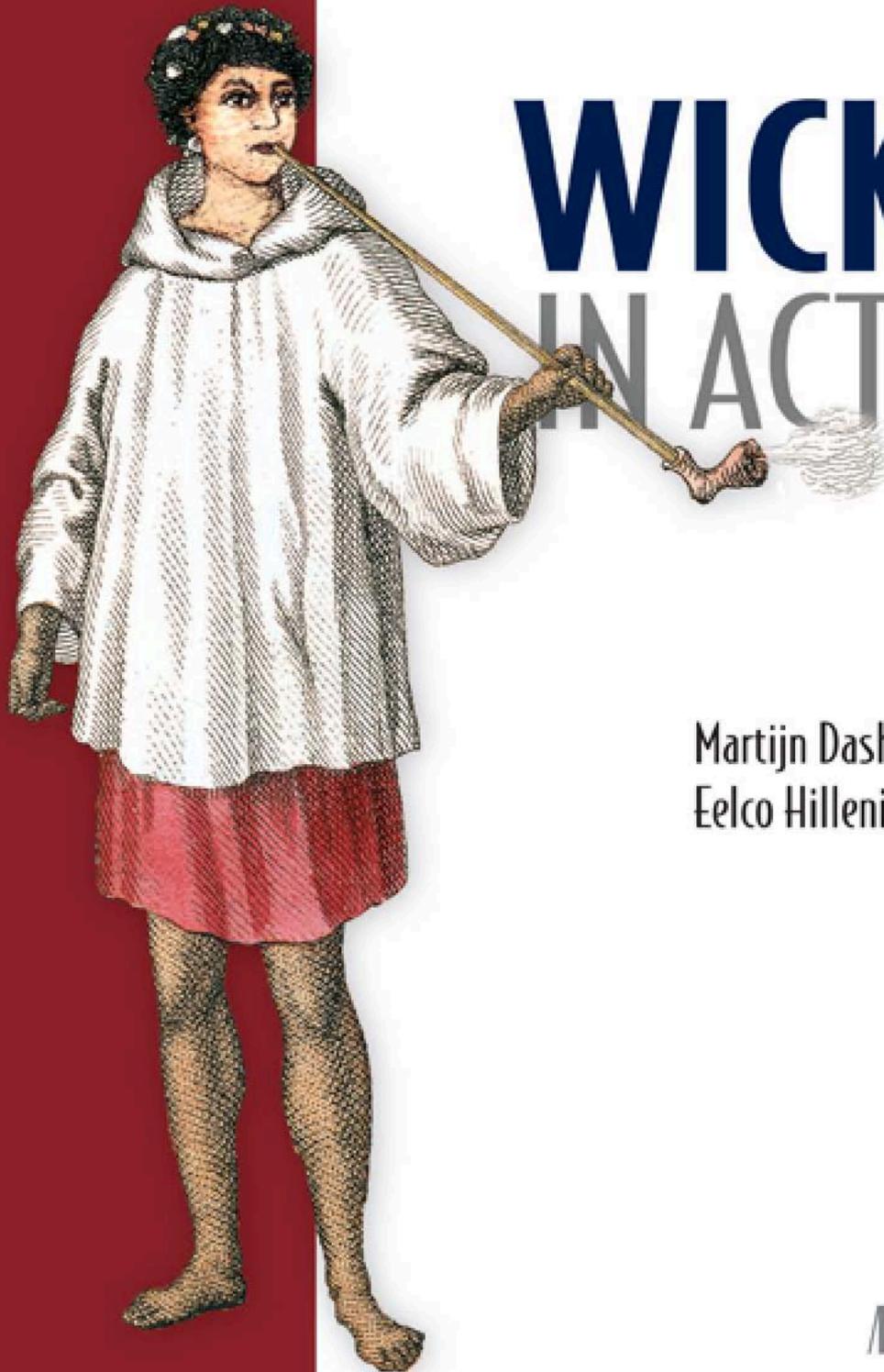# WICKET
IN ACTION

Martijn Dashorst
Eelco Hillenius

MANNING

Wicket in Action
by Martijn Dashorst and Eelco Hillenius
Bonus Chapter 15

15

*Setting up
a Wicket project*

In this chapter:

- Creating the standard web application directory structure
- Using Apache Ant and Maven to build your Wicket application
- Using the Wicket quick-start project template

Setting up a web project can be a complex (if it's your first web project) or boring (all projects after your first) ordeal. That is why many development teams have created a default project setup that they copy from one project to the next. But when you start with new technology, that default setup suddenly doesn't work anymore. For instance, when you develop Wicket applications, you don't need to compile JSPs, there is no use for tag libraries, and the number of XML configuration files drops significantly. So, the infrastructure for these actions and dependencies is no longer necessary. Often it's beneficial to consider the way things have been done and see if you can make improvements.

TIP   A lot of quick-start projects are available on the web, such as QWicket (sets up Wicket with Spring) and AppFuse Light (sets up Wicket with Spring and any number of ORM frameworks). The Apache Wicket

365

project also provides a minimal Maven-based quick-start project without dependencies on other projects. We'll look at this project later.

Let's create a skeleton web project by putting to work several Wicket concepts we introduced in chapters 1 and 2. You can then use the project as a starting point to explore the examples in this book, to try things yourself, or as a basis for your next Wicket-based project. If you're new to Java and haven't done any web development, this chapter will provide some insight into structuring your project for an efficient development process and building a web application using the tools of the trade: Ant and Maven.

We'll start by laying down the foundations, such as the correct Java development kit and creating a project directory structure. From there we'll be able to specify a project build using Ant or Maven. Finally, we'll create the classes and resources every Wicket application needs. In chapter 3, we build a full application on these fundaments. The results of this chapter can be stored in a zip archive and used for other, new projects as well. But first, let's start with the basics.

15.1 Setting up camp

To streamline the deployment of web applications, Java has the notion of web application archives (WAR) and enterprise application archives (EAR). The latter is commonly used for complex applications consisting of multiple WAR files and Enterprise Java Beans (EJBs). We'll describe a project setup to efficiently build a ready-to-deploy WAR file containing a quick-start application. First, let's look at the prerequisites to build such an application.

15.1.1 Before you start

In this section, we'll talk you through some requirements and recommended tools before you start developing your first Wicket application. We'll discuss all of these items in more detail later in this chapter. The following list is a minimum requirement to get started:

- A Java 5 development kit or newer
- A Java IDE
- A servlet container
- Apache Ant

As you may have figured out, you'll need a Java development kit (JDK). The version of Wicket that this book is about, Wicket 1.3, requires Java 1.4 or higher.

TIP   We advise you to use a more recent Java version—at the very least Java 5, but preferably Java 6. This gives you a jump in performance and provides you with more modern language features such as generics and annotations. You should check the life cycle of your preferred JDK to see when its support will terminate. For example, Java 1.4 support will end October 30, 2008, and Java 5 support will end one year after that (http://java.sun.com/products/archive/eol.policy.html).

JAVA IDES

We recommend that you use one of the common IDEs for developing Java. Most leading Java IDEs are freely available (Eclipse, NetBeans, and JDeveloper, to name a few), and others have trial versions that let you try them for a couple of weeks (IntelliJ IDEA, MyEclipse).

It's generally a bad idea to learn Java and Wicket without the use of a knowledgeable IDE. An IDE helps you navigate the sources and pull up class hierarchies, and it makes debugging your application easy. Determining the best IDE is a matter of taste and previous experiences (although many developers are religious about it). Because developing Wicket applications only requires editors for Java and HTML files, any of the available IDEs will work perfectly.

SERVLET CONTAINERS

To run the applications, you can choose any servlet container that is compatible with the servlet specification 2.3 or newer. All recent versions of Tomcat, Jetty, Resin, JBoss, and so forth are at the desired level. Some IDEs have an application server already embedded, so you might try that one. We'll assume you don't have an application server installed, so we'll use an embedded application server. All Wicket example projects provide a way of running the examples without installing your own application server. We'll discuss this embedded server in section 15.3.

BUILD TOOLS

Finally, you'll need a build tool for building the project. Apache Ant and Maven are both good tools with which to create web applications. In section 15.2, we'll provide the means to build applications with either one. But first, we need to settle on a directory structure that will satisfy both tools.

15.1.2 *The directory structure*

The directory structure for the project we're going to lay out is basically a standard: many projects recommend it, including the Maven project. We've used this structure in many projects, and it has helped us and new project members find their way around the project. Figure 15.1 shows the directory structure.

Now that we have a project structure to work in, let's fill in the blanks. First, we'll put the dependencies in place.

15.1.3 *The dependencies*

To build a Wicket application, you need to give the Java compiler some JAR files so it can compile your sources. To run an application in an embedded Jetty container, you introduce some dependencies that aren't needed when you don't use this way of running and testing your application. In this section we don't provide version numbers for the dependencies, because versions will have been updated between the time we write this book and the time you read it. You can find the most up-to-date list of required dependencies and their versions on the Wicket website.
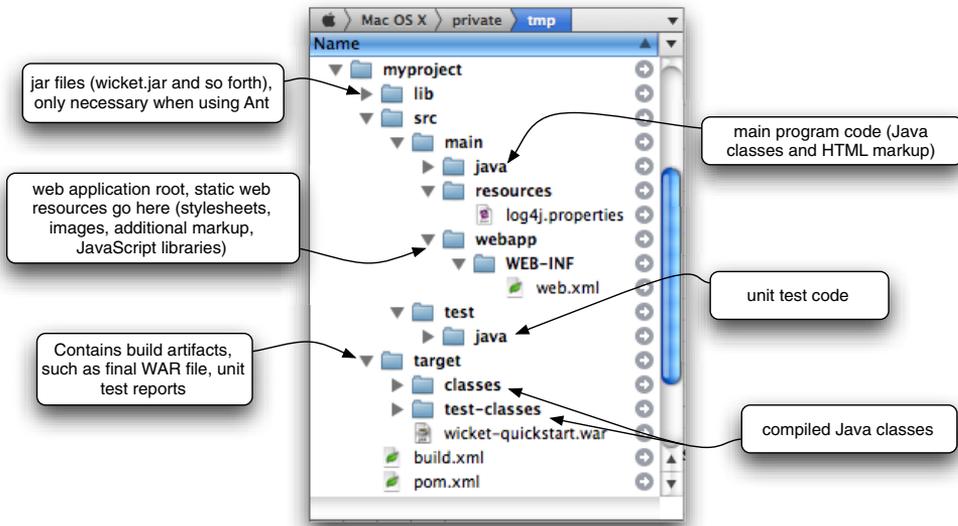
Figure 15.1   The recommended standard directory layout for our quick-start project

TIP   When you use Apache Maven as your build tool, it will automatically download the dependencies for you and store them in a local repository. All your projects can then use this single downloaded JAR file. You won't have to hunt down dependencies and download them manually when using Maven.

Here is a list of the compile-time dependencies:

- wicket.jar
- servletapi-2.3.jar
- slf4j-api.jar

With these dependencies, you can compile a web application. To run the application using an embedded Jetty servlet container, you need the following dependencies:

- jetty.jar
- jetty-util.jar

Depending on your logging preferences, you'll need additional dependencies. The simple logging facade for Java (slf4j) project allows the usage of different logging implementations, most notably the JDK provided logging and Apache's log4j logging. If you choose the latter, then you'll need the log4j JAR file and the slf4j log4j bridge, as follows:

- slf4j-log4j12.jar
- log4j.jar

All these JAR files go in the project's lib subdirectory. This is where you tell the compiler to look when compiling the application. And that brings us to the next section: configuring the build.

15.2 Configuring the build

Now that we've laid down the foundations for development, we can look at the tools with which you build an application. Building an application in a Java web context means creating a web archive (WAR) file. Although not all Java web applications require building a web archive, the servlet specification requires that the deployable application adhere to a strict structure. Some configuration files must be located in the right place in the archive, and your application's dependencies and code must be located in the correct directory in the archive. To build a web archive, follow these steps:

1 Compile the Java code.
2 Run the tests.
3 Assemble the web archive.

You can perform these steps by hand, but that would be silly. Programming has been around for many years, and building applications was one of the first thing developers automated.

Basically, you can use two approaches to build Java applications: Apache Ant and Apache Maven. The choice between the two tools is one of taste and sometimes of almost religious debate; we'll provide a setup for both tools and let you decide which one you prefer. For beginning web developers, we recommend starting with the Ant file. You already have much to learn, and jumping into Maven might be overwhelming.

This section is intended to get you building Wicket applications, not as a reference for Ant and Maven. Both tools have a lot of documentation available in the form of books and articles. First, we'll look at configuring an Ant build.

15.2.1 An Ant build.xml file

Ant is the definitive build tool for Java development. Many books have been written on the subject, including the excellent *Ant in Action* (Manning, 2007). Instead of trying to out-write Erik Hatcher and Steve Loughran, we'll provide a basic description of what is done in a build file.

Building a web application can be a complex process, but usually it boils down to the steps outlined in figure 15.2.

Figure 15.2   Steps for building and running a (web) application. This is a general process for any build of any program. Using Ant or Maven as our build system automates most of these steps.

You use these steps to create a deployable web application when using an Ant build:

1 Set up some default properties, such as the directories and Java version.
2 Start with a clean slate by removing all temporary files.
3 Initialize the build by creating the build directories.

4 Compile the Java sources.

5 Run the unit tests.

6 Package the application (including the runtime dependencies and resources) into the WAR archive.

7 Run the application using the embedded Jetty server.

The Ant file that performs these steps (build.xml) file is shown in listing 15.1 The build.xml file should be located in the project's root folder (quickstart/build.xml).

Listing 15.1   Ant build file used to build a web archive and run an application

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="war" name="wicket-quickstart" basedir=".">
    <property name="final.name" value="wicket-quickstart"/>
    <property name="src.main.dir" value="src/main/java"/>
    <property name="src.test.dir" value="src/test/java"/>
    <property name="src.web.dir" value="src/main/webapp"/>
    <property name="lib.dir" value="lib"/>
    <property name="build.dir" value="target"/>
    <property name="build.main.classes" value="${build.dir}/classes"/>
    <property name="build.test.classes"
            value="${build.dir}/test-classes"/>
    <property name="build.test.reports"
            value="${build.dir}/test-reports"/>
    <property name="build.reports.dir"
            value="${build.dir}/reports"/>
    <path id="build.classpath">
        <fileset dir="${lib.dir}">
            <include name="**/*.jar"/>
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${build.dir}" failonerror="false"/>
        <delete file="${final.name}.war" failonerror="false"/>
    </target>

    <target name="init">
        <mkdir dir="${build.dir}"/>
    </target>

    <target name="compile" depends="init">
        <mkdir dir="${build.main.classes}"/>
        <javac destdir="${build.main.classes}" target="1.5" source="1.5"
            srcdir="${src.main.dir}" classpathref="build.classpath"/>

        <copy todir="${build.main.classes}">
            <fileset dir="${src.main.dir}">
                <include name="**/*.*"/>
                <exclude name="**/*.java"/>
            </fileset>
        </copy>
    </target>

    <target name="test-compile" depends="compile">
        <mkdir dir="${build.test.classes}"/>
```

❶ Define build settings

❷ Copy markup files to class path

```xml
        <javac destdir="${build.test.classes}"
             target="1.5" source="1.5" srcdir="${src.test.dir}">
            <classpath>
                <path refid="build.classpath"/>
                <pathelement path="${build.main.classes}"/>
            </classpath>
        </javac>

        <copy todir="${build.test.classes}">
            <fileset dir="${src.test.dir}">
                <include name="**/*.*"/>
                <exclude name="**/*.java"/>
            </fileset>
        </copy>
</target>

<target name="test" depends="test-compile">
    <mkdir dir="${build.test.reports}"/>
    <junit dir="./" failureproperty="test.failure"
            printSummary="yes" fork="true" haltonerror="true">
        <sysproperty key="basedir" value="."/>
        <formatter type="xml"/>
        <classpath>
            <path refid="build.classpath"/>
            <pathelement path="${build.main.classes}"/>
            <pathelement path="${build.test.classes}"/>
        </classpath>
        <batchtest todir="${build.test.reports}">
            <fileset dir="${src.test.dir}">
                <include name="**/*Test*.java"/>
            </fileset>
        </batchtest>
    </junit>

    <mkdir dir="${build.reports.dir}"/>
    <junitreport todir="${build.reports.dir}">          Generate test  ❸
        <fileset dir="${build.test.reports}">               reports
            <include name="TEST-*.xml"/>
        </fileset>
        <report format="frames" todir="${build.reports.dir}"/>
    </junitreport>
</target>

<target name="war" depends="test">
    <war destfile="${build.dir}/${final.name}.war"
         webxml="${src.web.dir}/WEB-INF/web.xml">
        <lib dir="lib">
            <include name="wicket*.jar"/>
            <include name="slf4j*.jar"/>          ❹  Add dependencies
            <include name="log4j*.jar"/>
        </lib>
        <classes dir="${build.main.classes}"/>
        <fileset dir="${src.web.dir}">
            <include name="**/*"/>                ❺  Copy external
            <exclude name="**/web.xml"/>              resources
        </fileset>
    </war>
</target>
```

```
        <target name="run" depends="test-compile">
            <java classname="com.mycompany.Start" fork="true">
                <classpath>
                    <path refid="build.classpath"/>
                    <pathelement path="${build.main.classes}"/>
                    <pathelement path="${build.test.classes}"/>
                </classpath>
            </java>
        </target>
    </project>
```

❻ **Run application**

We define several properties for the build process ❶. It's considered a good habit to use properties instead of hard-coded literals scattered across your build file: when you need to reference the directory where your sources are, use the property ${src.main.dir} and keep the exact location of the source files in one place instead of all over the build file.

By default, Wicket tries to locate the markup files that belong to your components and pages on the class path ❷. This Ant target ensures that all the resources located in your source folders are copied to the build directory alongside your classes. If you didn't execute this step, Wicket won't find them at all and would show an exception page that it couldn't find the associated markup file.

Before you can successfully build a package, it's customary to run unit tests. The test runner needs both your compile and test classpaths. You also have to add the compiled test cases to the classpath. Here we generate HTML reports from the gathered test results ❸.

For quick development, we've put some extra JAR files in our lib folder that may be problematic when you deploy your application as a WAR file into a standalone servlet container. The servlet-api.jar file is provided by your servlet container, as are the Jetty JAR files. By including only the JARs we need ❹, we ensure that our WAR file doesn't contain unwanted, unnecessary, or even conflicting JAR files.

When creating the WAR file ❺, you should copy the external resources, such as CSS and JavaScript files, into the archive. Because Ant wants control over the web.xml file, you can't copy the file as a resource: you must specify it as a parameter in the WAR task. Ant will complain about already having a web.xml file present in the resulting WAR file and will fail the build if you don't exclude it from the resource copy.

When you start working on your application, it's highly beneficial to see the results immediately. In our build file, we include a task for Ant to run our application using the embedded Jetty server ❻. This enables us to quickly see if our application works. Performing the following on the command line is enough to start our application:

```
    ant run
```

This build file has everything we need to create a web application: it knows where to find the sources, dependencies, and markup files; how to compile the Java code; and what to package in the resulting WAR file. Now, let's see how you accomplish the same build using Maven.

15.2.2 *A Maven project file*

One of the major improvements in building software recently has been the development of Maven. Maven is a comprehensive project-management system. Using a project description in the form of a project object model, Maven can manage a project's dependencies, build, reporting, and documentation.

Choosing between Maven and Ant is like a religious choice for some developers. Instead of repeating those debates here, we leave the choice to you. The Wicket project is built using Maven, so it may be beneficial to read this section if you want to build Wicket from source. If you don't like to work with Maven, you can skip this section and continue setting up the project. In this section, we'll just scratch the surface of Maven; if you want more information, please refer to the Maven website or one of the available Maven books.

DESCRIBING YOUR PROJECT

Maven works with a project descriptor. In this file, you describe your project, how it's structured, what dependencies are needed, and what artifacts are the result of the build. This is essential to using Maven: you don't tell it how to work, you just tell it what needs to be done. You can achieve the same using Ant, but it takes some work to provide the same rich functionality offered by Maven.

A project is described in a project object model (POM) file. Maven expects this file to have the name pom.xml. First, create a file named pom.xml in the root of the project. The following snippet shows the minimal contents of a POM:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <name>Quickstart</name>
    <groupId>com.mycompany.quickstart</groupId>
    <artifactId>quickstart</artifactId>
    <packaging>war</packaging>
    <version>1.0</version>
</project>
```

This POM starts with the model version. This is the version Maven needs to be able to parse your project descriptor. At the moment of writing, the latest version is 4.0.0, which is the minimal version for Maven 2 to work.

The `name` element is a descriptive name for the project. Maven uses this name when generating the documentation. This name is also shown when you run Maven commands on your project.

The group identifier (`groupId`) usually contains the name of your company and/or project. It's the grouping under which all the artifacts are stored in your local repository. The local repository is where all the dependencies you use end up; typically, you can find it in your home directory under .m2/repository (Windows users should look in C:\Documents and Settings\username\.m2\repository). We'll go into the details of the local repository in the next section when we discuss Maven's dependency-management features.

The artifact identifier tells Maven how to name your project's artifacts. Using it together with the version and the packaging, Maven will create in our example a WAR

archive named quickstart-1.0.war. Table 15.1 shows some combinations using the Wicket projects as an example.

Table 15.1 Example combinations for Maven dependencies. A dependency is identified by its group id, artifact id, version, and packaging type. The group id is converted to a directory path in the Maven repository, and the artifact id together with the version and packaging result in the filename.

Group id	Artifact id	Version	Packaging	Artifact
org.apache.wicket	wicket	134	JAR	wicket-1.3.4.jar
org.apache.wicket	wicket-examples	134	WAR	wicket-examples-2.0.war
org.apache.wicket	wicket-spring	134	JAR	wicket-spring-2.0.jar

Before you can start building an application, you need to define your dependencies.

MANAGING YOUR DEPENDENCIES

Maven does more than compile and package your Java code; it also manages your project's dependencies. It does so by specifying which artifacts your project depends on using the group and artifact identifiers and the version. Maven then tries to automatically download the dependency from a central repository and copy the file into your local repository for future reference.

The Maven central repository is a huge collection of (mainly) open source Java libraries. Many projects, including Wicket, submit their artifacts to this repository for the benefit of all Maven users. Every time you use a specific dependency in a build, Maven points the Java compiler or packaging tool to the dependency in your local repository. This way, the dependency can be stored on your disk just once. Figure 15.3 shown some an example of the Maven local repository.
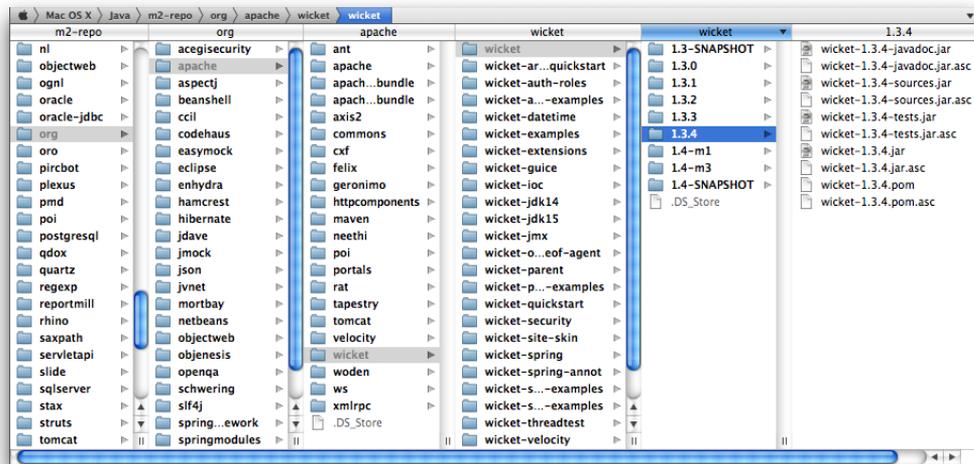

Figure 15.3 An example of the Maven local repository organized by group id and version. This repository illustrates a benefit of using Maven: most open source projects publish their sources in the Maven repository so you can attach them in your IDE to the binary JAR file, making the JavaDoc and sources available in the editor.

For our Wicket project, we need to define that we'll depend on the Wicket JAR. This is shown in listing 15.2.

Listing 15.2 Maven project descriptor for our quick-start application

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <name>Quickstart</name>
    <groupId>com.example</groupId>
    <artifactId>quickstart</artifactId>
    <version>1.0</version>
    <packaging>war</packaging>
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>servlet-api</artifactId>
            <version>2.3</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.wicket</groupId>
            <artifactId>wicket</artifactId>
            <version>1.3.4</version>
            <scope>compile</scope>
        </dependency>
        <dependency>
            <groupId>org.mortbay.jetty</groupId>
            <artifactId>jetty</artifactId>
            <version>6.1.10</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.mortbay.jetty</groupId>
            <artifactId>jetty-util</artifactId>
            <version>6.1.1</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.4.2</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.14</version>
        </dependency>
    </dependencies>
</project>
```

1 Provided by container

2 Wicket dependency

3 Embed Jetty server

4 Default log4j

In this example, we add five dependencies to the project. Because we're building a web application, we need the servlet specification APIs: the JAR contains all the necessary interfaces that web applications need to handle requests. The implementations of these interfaces are provided by the application server. This explains the scope

element of the dependency ❶: it's provided at runtime. Specifying the scope pro-vided tells Maven to include the dependency while compiling but to omit it when packaging the WAR file.

Our second dependency ❷ is the Wicket JAR file. In this case, the scope is compile time. This is the broadest scope and the default, so we could have omitted the scope definition in this case. Making a dependency's scope compile ensures that the dependency takes part in all phases of the build: compilation, testing, packaging, and so forth.

With the Wicket dependency, one of the major strengths of using Maven comes into play: transitive dependency handling. When projects available from the Maven repositories also specify their dependencies in a POM, Maven can infer their dependencies and include them into your project. In this case, Wicket is dependent on slf4j. As a user of the Wicket artifact, you don't have to specify those yourself: Maven automatically includes the Wicket dependencies in your project. At the moment of writing, dependencies marked provided aren't taken into account when resolving the classpath, which is why we have to specify the servlet API dependency ourselves.

The other dependencies are used for embedding the Jetty server later ❸ and letting slf4j use the log4j library ❹. You can safely ignore these dependencies if you wish to use a different logging implementation, but you must provide a slf4j logging implementation.

Maven has many tricks up its sleeve, and one of them is a life saver: the IDE plug-ins. Using the dependencies defined in your project, Maven can generate the project files for your favorite IDE. At the moment of writing, both Eclipse and IDEA are supported.

NOTE   NetBeans and IntelliJ IDEA have direct support for Maven project descriptors, so you don't have to generate project files for those IDEs. Support for Maven POM files in Eclipse is an ongoing development effort at the moment.

When we want to generate the project files for our IDE of choice, we can issue one of the commands shown in the following example:

```
mvn eclipse:eclipse
mvn idea:idea
```

This command instructs the Maven IDE plug-in to generate the necessary project and classpath files. See the plug-in's documentation to configure it such that it also downloads and installs available source archives for your dependencies: it's an indispensable benefit of using open source software and will help you tremendously when debugging.

We're almost ready for our Maven project. We need to do one thing: configure the Maven WAR plug-in.

BUILDING A WAR

When building a WAR archive using Maven, you usually don't have to specify what it needs to contain. Maven knows where your classes went when they were compiled, and Maven knows which libraries your project depends on and that it needs to put in the WEB-INF/lib folder. Unfortunately, Maven doesn't know what types of resources

need to be copied when compiling the sources. We can tell Maven to copy resources using the part of the POM from listing 15.3.

Listing 15.3   Copying resources from our source directory to the classpath

```
<project>
    ...
    <dependencies>...</dependencies>
    <build>
        <resources>
            <resource>
                <filtering>false</filtering>
                <directory>src/main/java</directory>
                <includes>                                 Copy
                    <include>**</include>                  everything...
                </includes>
                <excludes>                                 ...except Java
                    <exclude>**/*.java</exclude>           sources
                </excludes>
            </resource>
        </resources>
    </build>
</project>
```

In this snippet from our POM, we provide Maven with the location of our resources and the resources to include and exclude. Because we used the default Maven directory layout, we have nothing more to declare. We can now build our WAR file using Maven.

Maven packs a lot of functionality and can be overwhelming. Table 15.2 shows the Maven commands that are commonly used when building an application.

Table 15.2   Commonly used Maven commands

Command line	Description
mvn clean	Cleans up the build directories
mvn compile	Compiles the Java code into targets/classes
mvn test	Compiles the test code into targets/test-classes and runs the (unit) tests
mvn package	Builds the final artifact (JAR, WAR, EAR)
mvn install	Installs the final artifact in your local repository
mvn eclipse:eclipse mvn idea:idea	Generates IDE project files based on the dependencies in your POM (Eclipse and IntelliJ IDEA are supported)
mvn jetty:run	Runs your application using the Maven Jetty plug-in

When you want to build the WAR archive, usually you only run the `package` command, because that command performs the `compile` and `test` commands as well. To start

with a clean slate—the preferred state when building a WAR archive for deployment—you can add the `clean` argument.

> ### Be bad: disable running the unit tests
>
> Maven always tries to run the tests when you package your project. If you have some failing tests, but you still want the artifact to be created, you can add a command-line parameter that tells Maven to skip the tests:
>
> `mvn -Dmaven.test.skip=true clean package`
>
> All bets are off; but when you're trying to build something and you need it now, this is a good compromise.

When you build a package, it will look similar to the following example:

```
mvn clean package
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------
[INFO] Building Quickstart
[INFO]    task-segment: [clean, package]
[INFO] ------------------------------------------------------------
[INFO] [clean:clean]
[INFO] Deleting directory quickstart/target
[INFO] Deleting directory quickstart/target/classes
[INFO] Deleting directory quickstart/target/test-classes
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] No sources to compile
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [surefire:test]
[INFO] No tests to run.
[INFO] [war:war]
[INFO] Exploding webapp...
[INFO] Assembling webapp quickstart in quickstart/target/quickstart-1.0
[INFO] Copy webapp webResources to quickstart/target/quickstart-1.0
[INFO] Generating war quickstart/target/quickstart-1.0.war
[INFO] ------------------------------------------------------------
[ERROR] BUILD ERROR
[INFO] ------------------------------------------------------------
[INFO] Error assembling WAR: Deployment descriptor: quickstart/target/
   quickstart-1.0/WEB-INF/web.xml does not exist.
```

As you can see, Maven complains that we don't have a web.xml file yet. This is something we'll pick up shortly.

Now that you can build a WAR archive for deployment on your test server, you can begin building your application. Let's first create some things that need to go into every application; you only have to do this once.

15.3 *Creating the application infrastructure*

All people are created equal, and so are Wicket applications. All applications have some things in common. By putting these things into a base project structure, you can jump-start your application development and quickly build proof-of-concept applications to isolate a bug or try a new component or train of thought. Each Wicket application contains these elements:

- An `Application` object
- A home page
- A web.xml file

In this section, we'll build these elements. As a bonus, we'll make developing web applications easier by embedding a servlet engine in our application, so that we can instantaneously run and debug our application without having to jump through the hoops of building and deploying the entire application. Let's start with the application.

15.3.1 *Creating the Application object*

As we mentioned in chapter 2, every Wicket application requires an `Application` object. This object provides Wicket with the configuration parameters and establishes the first page users see when they arrive at your application if they don't provide a special URL. To create the `WicketApplication` object, we put the following code in QuickstartApplication.java in the project's src/main/java/com/mycompany subdirectory:

```
package com.mycompany;

import org.apache.wicket.protocol.http.WebApplication;

public class WicketApplication extends WebApplication {
    public WicketApplication() {
    }
    public Class getHomePage() {
        return null;
    }
}
```

As you can see, nothing much has been done here (yet). The class is empty; it contains only the things required to make the class compile. We had to implement the `getHome-Page` method so Wicket knows where to direct users when they get to the application.

Now that we have an application, we need to tell it which page is the starting page. Let's create the page first and work from there. Create a file called HomePage.html in the same directory as the `WicketApplication` Java class:

```
<html>
<head>
    <title>Quickstart</title>
</head>
<body>
    <h1>Quickstart</h1>
    <p>This is your first Wicket application!</p>
</body>
</html>
```

This should suffice for a first run of the application. Later, we'll add more markup and components to make it more interesting.

For this to work, we also need a Java class: `HomePage`. Create a new file named HomePage.java in the same directory, as follows:

```
package com.mycompany;

import org.apache.wicket.markup.html.WebPage;

public class HomePage extends WebPage {
    public HomePage() {
    }
}
```

This page is similar to the first incarnation of the Hello, World page we showed in section 1.3.1. It doesn't have any dynamic behavior yet, but we'll get to that shortly.

Now that we have a home page, we can implement the `getHomePage` method on our application class:

```
public Class getHomePage() {
    return Index.class;
}
```

This method tells Wicket to return a new instance of the `HomePage` class whenever the page is requested.

While we're on the subject, how does Wicket know that the page is requested? The servlet container needs to tell Wicket that a request has arrived. So, we need to make the servlet container aware that a Wicket application is waiting to handle requests. Enter the web.xml configuration file.

15.3.2 *Configuring the web.xml file*

Each web application must define its connections to the servlet container in a file called web.xml. The web.xml file needs to be present in the WEB-INF root folder in the WAR file when you're going to deploy an application in a servlet container.

The web.xml file contains several sections, each of which has a specific place. The best way to edit a web.xml file is to use a DTD-aware XML editor, which most IDEs have by now. You may have to download and install a plug-in, depending on your IDE of choice.

Save the web.xml file in the project's src/main/webapp/WEB-INF folder. To make the editor aware of your DTD, begin the web.xml file with the following declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

You can use the 2.4 servlet specification or newer, which is more relaxed about the order of the elements requirements. For now, we'll stick to the 2.3 specification. Listing 15.4 shows the contents of the file.

Listing 15.4 web.xml deployment descriptor for the quick-start application

```
<web-app>
    <display-name>myproject</display-name>
    <context-param>                                          ❶  Set Wicket
        <param-name>configuration</param-name>                  mode
        <param-value>development</param-value>
    </context-param>
    <filter>
        <filter-name>wicket.myproject</filter-name>          ❷  Define
        <filter-class>                                          Wicket
            org.apache.wicket.protocol.http.WicketFilter        filter
        </filter-class>
        <init-param>
            <param-name>applicationClassName</param-name>    ❸  Set
            <param-value>                                       application
                com.mycompany.WicketApplication                 class name
            </param-value>
        </init-param>
    </filter>

    <filter-mapping>                                          ❹  Map all
        <filter-name>wicket.myproject</filter-name>             requests to
        <url-pattern>/*</url-pattern>                           Wicket filter
    </filter-mapping>
</web-app>
```

In this mapping, we tell Wicket to start in development mode ❶. This configures Wicket to allocate resources to streamline the development process. In development mode, Wicket shows stack traces when an exception is caught, reports coding errors with line-precise messages, reloads changes in markup, and shows the debug inspector. The price for all these features is slightly less performance and increased memory usage.

Conversely, when you set the configuration to deployment mode, Wicket reports no stack traces (the average Joe user won't know what to do with them anyway) but shows a bit friendlier error page (if you didn't provide one yourself). Wicket stops scanning for changes to markup and resource files, and other settings are configured to maximize performance and scalability. Section 14.3 goes into more detail about these settings and configuring your application for maximum performance in production.

We define the Wicket filter ❷ and provide it with our application's class name ❸. We define the filter mapping in such as way that it maps all requests behind the URL pattern /* to go through the Wicket filter and hence our application ❹. Wicket identifies whether a request is targeted for your application or is a request for a context resource (such as a stylesheet or JavaScript file in the src/main/webapp directory).

Now that you've told the servlet container and Wicket how to create the application, you can prepare to run the application. If you have an IDE with application server support, you can skip the following section and try the application. For those without an application server, we'll make sure you're able to run the application using the Jetty embedded application server.

15.3.3 *Embedding a Jetty server*

In section 15.2.3, we put the dependencies for the Jetty embedded server into our project. Now it's time to put those JAR files to good use. Jetty is a full-featured, high-performance, open source servlet engine. Installing it is easy: you download the current stable distribution, unzip it to a folder of your liking, copy your WAR file to the webapps subdirectory of the Jetty installation, and start Jetty.

The ease of installation is one aspect that appeals to us; the other is that Jetty isn't limited to running standalone. You can embed Jetty in a Java application and have it serve your web application. We'll use this feature in this section. First, we'll create a Java class that gives us the entry point to start Jetty and our web application. In the com.mycompany package in the src/test/java directory, we create the Start class. We put the class in the src/test/java directory because we don't want to deploy the Jetty servlet container with our web application. A servlet container should be provided by our production server. Listing 15.5 provides the code for our embedded server.

Listing 15.5   Embedding the Jetty servlet container in our application

```
package com.mycompany;

import org.mortbay.jetty.Connector;
import org.mortbay.jetty.Server;
import org.mortbay.jetty.bio.SocketConnector;
import org.mortbay.jetty.webapp.WebAppContext;

public class Start {
    public static void main(String[] args) throws Exception {
        Server server = new Server();
        SocketConnector connector = new SocketConnector();
        connector.setMaxIdleTime(1000 * 60 * 60);
        connector.setSoLingerTime(-1);
        connector.setPort(8080);                                     ❶ Listen to
        server.setConnectors(new Connector[] { connector });           port 8080

        WebAppContext bb = new WebAppContext();
        bb.setServer(server);
        bb.setContextPath("/");            ❸ Define        ❷ Publish
        bb.setWar("src/main/webapp");        document root    application
        server.addHandler(bb);

        try {
            System.out.println(">>> STARTING EMBEDDED JETTY SERVER,"
                    + " PRESS ANY KEY TO STOP");
            server.start();                                      ❹ Start
            while (System.in.available() == 0) {    Wait until     Jetty
                Thread.sleep(1000);                 ready
            }
            server.stop();
            server.join();
        } catch (Exception e) {
            e.printStackTrace();
```

```
            System.exit(100);
        }
    }
}
```

Here we configure the Jetty server and start it. We configure a connector to listen to port 8080 ❶. Next, we register our application with Jetty ❷. We map our application context to the root context (/) and point the web application root to be our project subdirectory src/main/webapp ❸. All files in this directory are served by the Jetty server. This directory is an excellent place to store our CSS and JavaScript files.

   Now that you've created a launcher class for the application and configured the Jetty server, you can run the application for the first time ❹.

15.3.4 *Running the application for the first time*

After all this hard work, you finally get your reward: the chance to run your application for the first time. To recap what you've accomplished so far, here are the steps you've taken:

   1 Create a project directory structure.
   2 Create an Ant build file or a Maven POM.
   3 Write the WicketApplication class.
   4 Write the first Wicket page: HomePage.
   5 Configure the web.xml file.
   6 Create a launcher for Jetty, and configure the Jetty server.

If you haven't compiled the source code, now is a good time to do so. Either use your IDE's capabilities, run the Ant script to compile the sources, and run the application; or use the Maven Jetty plug-in.

RUNNING THE APPLICATION USING ANT

Using Ant to run an application is as simple as invoking the following command line, provided you followed the steps outlined in section 15.2.5:

```
ant run
Buildfile: build.xml

init:

compile:
    [javac] Compiling 3 source files to target/classes
     [copy] Copying 1 files to target/classes

run:
[java] *************************************************************
[java] *** WARNING: Wicket is running in DEVELOPMENT mode.       ***
[java] ***                            ^^^^^^^^^^^                ***
[java] *** Do NOT deploy to your live server(s) without changing ***
[java] *** this. See Application#getConfigurationType() for more  ***
[java] *** information.                                          ***
[java] *************************************************************
```

The application will start when no compilation problems have been reported. You can now start your browser and look at the application.

Note the warning in the application log that we're running in development mode. As the warning clearly states, don't use development mode for anything other than development. If you did, your users would be able to use your application, but they would see too much information and might be able to use Wicket's debugger to see how your application works. That is a hacker's gold mine waiting to be exploited. If you need a feature that is enabled in development mode but not in deployment mode, you can selectively enable that feature in the Application's `init` method.

RUNNING THE APPLICATION USING THE MAVEN JETTY PLUG-IN

If you ever want to try a web application quickly, and it's built using Maven, you can use the Maven Jetty plug-in to run the application directly without having to deploy the application to a server and (re)start that server. To enable this plug-in, add the following XML snippet to the `build` section of the POM:

```
<plugins>
    <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <version>6.1.10</version>
        <configuration>
            <scanIntervalSeconds>60</scanIntervalSeconds>      ⟵ Check for updates
            <contextPath>/</contextPath>
        </configuration>
    </plugin>
</plugins>
```

This configures Maven to be able to find the plug-in (it isn't an out-of-the-box plug-in, but is provided by the Jetty maintainers) and supplies some configuration for scanning the application for updates. Starting the application is now as simple as running the following command line:

```
mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO] ----------------------------------------------------------------
[INFO] Building Quickstart
[INFO]    task-segment: [jetty:run]
[INFO] ----------------------------------------------------------------
... SNIP ...
[INFO] Starting jetty 6.1.10 ...
[INFO] ****************************************************************
[INFO] *** WARNING: Wicket is running in DEVELOPMENT mode.        ***
[INFO] ***                        ^^^^^^^^^^^                     ***
[INFO] *** Do NOT deploy to your live server(s) without changing  ***
[INFO] *** this. See Application#getConfigurationType() for more   ***
[INFO] *** information.                                           ***
[INFO] ****************************************************************
[INFO] Started SelectChannelConnector @ 0.0.0.0:8080
[INFO] Started Jetty Server
[INFO] Starting scanner at interval of 60 seconds.
```

If everything is configured correctly, Maven will compile and test your code, and start the Jetty server. When the last line is visible, you can start your browser to check out the application.

RUNNING THE APPLICATION IN THE BROWSER
Let's see what our application looks like. Open your browser, and type the following URL in the address bar:

    http://localhost:8080

You'll see the page shown in figure 15.4, served to you by the Jetty server.

Quickstart

This is your first Wicket application!

Figure 15.4 The first page of our application, served by Jetty. That is our first web page, built from scratch!

15.4 Using Wicket's quick-start project generator

The Wicket website has a link to a quick-start project generator. This generator assumes you'll be using Maven as your build system or at least to set things up. You'll need at least Maven 2.0.4 to make this work—downloading and installing Maven is trivial, and OS X comes with Maven preinstalled since version 10.5.

The generator needs three items to generate the project skeleton:

- Your group id
- Your artifact id
- The Wicket version

Figure 15.5 shows the quick-start wizard in action.


```
GroupId:      com.mycompany                    (?)
ArtifactId:   myproject                        (?)
Version:      1.3.4                      ▼      (?)
Command Line: mvn archetype:create –DarchetypeGroupId=org.apache.wicket –
              DarchetypeArtifactId=wicket–archetype–quickstart –
              DarchetypeVersion=1.3.4 –DgroupId=com.mycompany –
              DartifactId=myproject
```

Figure 15.5 The Wicket quick-start project generation wizard. Fill in three values, and get a project skeleton for free!

Once you've filled in the necessary data, you can select the text in the Command Line box and copy it to the clipboard. This command line instructs Maven to generate the project skeleton. It uses an *archetype* for the Wicket project, which is a template project with hooks to generate the correct packages and Java classes based on the group and artifact identifiers you've provided.

> ### Reporting Wicket bugs the right way
>
> When you encounter a bug in Wicket (yes, even Wicket isn't perfect), the best way to provide a bug report is to reproduce the bug in a quick start. By this we mean a project that you generated with the quick-start wizards on the Wicket website and that exhibits your problem. You can zip it (call `mvn clean` first to remove all generated artifacts and minimize the zip-file size) and attach it to a bug report. This process will save the Wicket developers a lot of time trying to fix your bug.

Open a DOS box (or terminal window), and change the current directory to the place where you want to work on your new project. For example:

```
cd /Users/dashorst/workspace
```

Now, all you need to do is paste in the command line generated with the quick-start generator. The following example shows the output of one such run:

```
mvn archetype:create -DarchetypeGroupId=org.apache.wicket
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:create] (aggregator-style)
[INFO] ------------------------------------------------------------
...
[INFO] [archetype:create]
[INFO] Defaulting package to group ID: com.mycompany
[INFO] ------------------------------------------------------------
[INFO] Using following parameters for creating Archetype: wicket-arc
[INFO] ------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.mycompany
[INFO] Parameter: packageName, Value: com.mycompany
[INFO] Parameter: package, Value: com.mycompany
[INFO] Parameter: artifactId, Value: myproject
[INFO] Parameter: basedir, Value: /Workspace
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] ******************** End of debug info from resources from g
[INFO] Archetype created in dir: /Workspace/myproject
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------
[INFO] Total time: 4 seconds
[INFO] Finished at: Mon Aug 04 01:15:59 CEST 2008
[INFO] Final Memory: 5M/506M
[INFO] ------------------------------------------------------------
```

If it's your first time working with this Maven plug-in (the skeleton is generated by the archetype plug-in), a lot of internet downloads will occur. The command should run much more quickly the next time.

The generated project comes with a working setup to get you started immediately. For instance, change into the generated project directory, and start your application using the Maven Jetty plug-in, as shown in the following snippet:

```
cd myproject
mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO] ------------------------------------------------------------
[INFO] Building quickstart
[INFO]    task-segment: [jetty:run]
[INFO] ------------------------------------------------------------
[INFO] Preparing jetty:run
...
INFO  - WebApplication          - [WicketApplication] Started Wicket
**********************************************************************
*** WARNING: Wicket is running in DEVELOPMENT mode.          ***
***                          ^^^^^^^^^^^                     ***
*** Do NOT deploy to your live server(s) without changing this.  ***
*** See Application#getConfigurationType() for more information. ***
**********************************************************************
2008-08-04 01:28:00.336::INFO:  Started SelectChannelConnector@0.0.0.0:
[INFO] Started Jetty Server
```

We've left out a lot of logging because it isn't interesting. Now that the Jetty server is running, you can instruct your browser to open the home page at http://localhost: 8080 (you might need to navigate to http:// localhost:8080/myproject, depending on the Jetty plug-in configuration). The result is shown in figure 15.6.

Wicket Quickstart Archetype Homepage

If you see this message wicket is properly configured and running

Figure 15.6   The home page of the Wicket quick-start archetype project

Using this quick start is a real time saver for quickly starting a project. With a proper installation of Maven, you can get your first Wicket application running within minutes.

15.5 *Summary*

Now that you have a working application and an Ant file with which to build the application, you can proceed and implement your business functionality. To get to this point required quite a bit of work: you had to create a directory structure; download several JAR files; create build files for Ant or Maven; and create an application class, a home page, and a web.xml file. As a bonus, you embedded a servlet engine to run the application directly from the IDE.

Fortunately, you don't have to do this work often. You can store this as a template and work your way up from there, or use the ready-to-go Wicket quick-start project generator.

Use the template we developed in this chapter to your advantage. Revisit the examples from chapter 1, and try them yourself. It's one thing to read the code, but it's much more fun and educational to get hands-on experience.
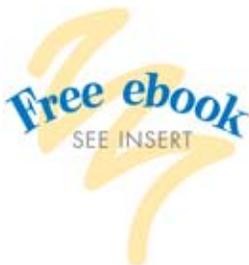
WICKET IN ACTION

Free ebook
SEE INSERT

Martijn Dashorst and Eelco Hillenius
FOREWORD BY Jonathan Locke

Wicket bridges the mismatch between the web's stateless protocol and Java's OO model. The component-based Wicket framework shields you from the HTTP under a web app so you can concentrate on business problems instead of the plumbing code. In Wicket, you use logic-free HTML templates for layout and standard Java for an application's behavior. The result? Coding a web app with Wicket feels more like regular Java programming.

Wicket in Action is a comprehensive guide for Java developers building Wicket-based web applications. It introduces Wicket's structure and components, and moves quickly into examples of Wicket at work. Written by core committers, this book shows you the "how-to" and the "why" of Wicket. You'll learn to use and customize Wicket components, to interact with Spring and Hibernate, and to implement rich Ajax-driven features.

What's Inside

- All of Wicket's basic concepts and components
- Security, localization, and internationalization
- Creating custom reusable components
- Wicket's Ajax and JavaScript capabilities
- Working with databases and resources
- Prepare your application for production

About the Authors

Martijn Dashorst has been actively involved in Wicket since it was open-sourced, and speaks regularly at conferences, including JavaOne and JavaPolis. **Eelco Hillenius** has been part of Wicket's core team almost from the start. He works for Teachscape where he is helping to build the next e-learning platform.

For online access to the authors, code samples, and (for owners of this book) a free ebook, go to www.manning.com/WicketinAction

"This is the complete and authoritative guide to Wicket, written and reviewed by the core members of the Apache Wicket team. If there's anything you want to know about Wicket, you are sure to find it in this book."

—From the Foreword by Jonathan Locke
Founder and Architect of Apache Wicket

"With this book, Wicket will become the greatest territory the Dutch have settled since Manhattan."

—Nathan Hamblen
Senior Software Developer
Teachscape, Inc.

"Loved the sample application —it tied everything together."

—Phil Hanna
Senior Software Developer
SAS Institute

"The essential guide for learning and using Wicket."

—Erik van Oosten
Lead Programmer and
Project Manager, JTeam

MANNING $44.99 / Can $44.99 [INCLUDING EBOOK]