Spring Boot

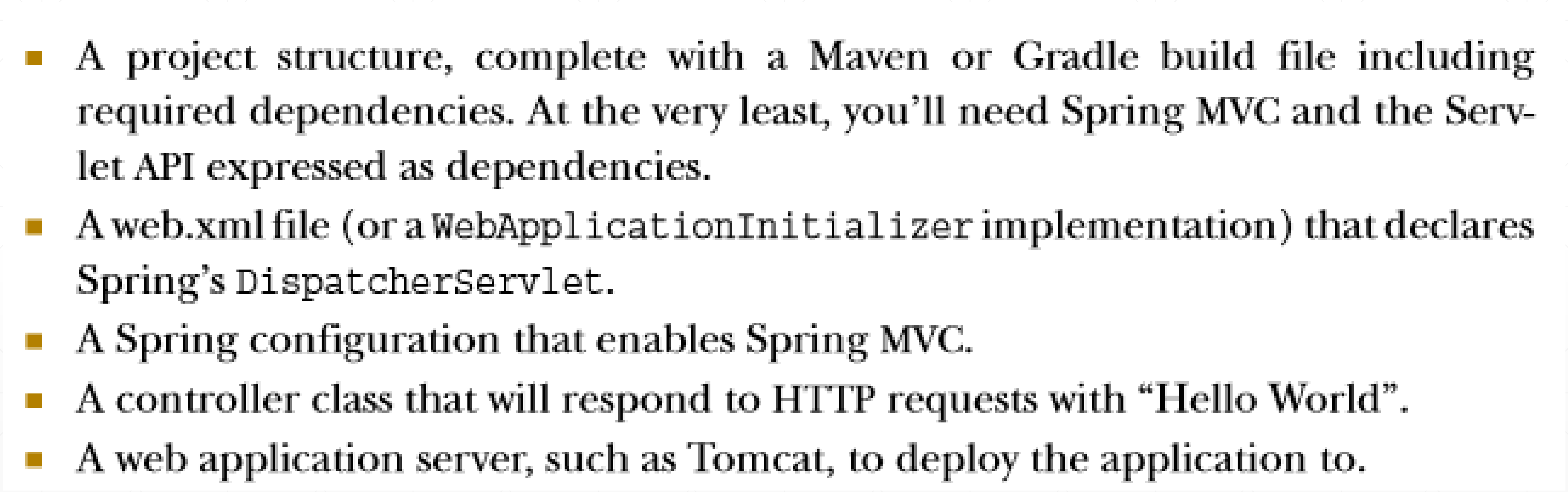
Spring Boot

- Spring was lightweight in terms of component code, but heavyweight in terms of configuration

- Spring 3.0 introduced a Java-based configuration as a type-safe and refactorable option to XML.

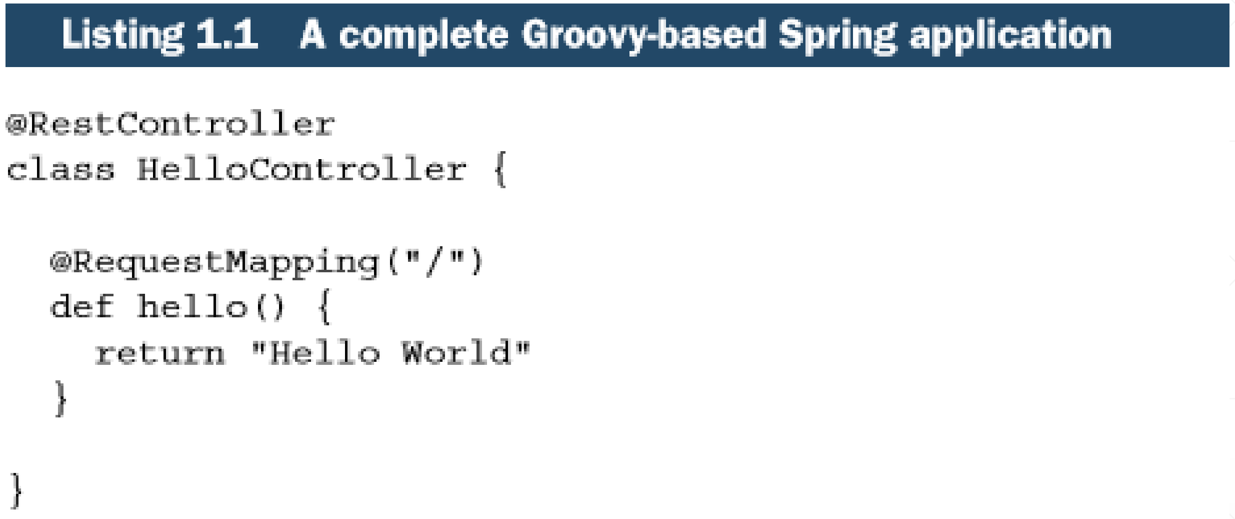
Simple Example

Developing a very simple Hello World web application with Spring



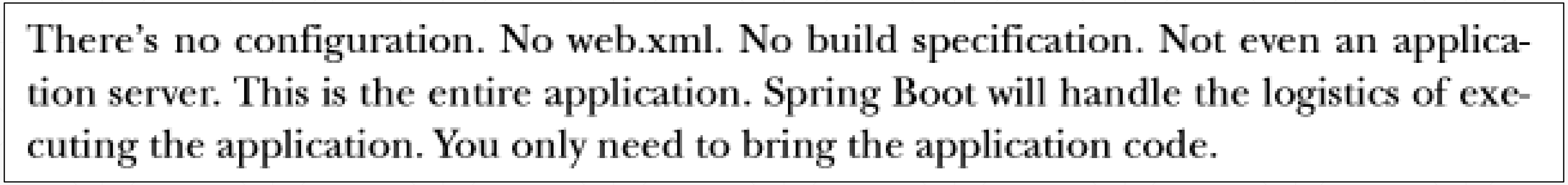
- A project structure, complete with a Maven or Gradle build file including required dependencies. At the very least, you'll need Spring MVC and the Servlet API expressed as dependencies.
- A web.xml file (or a `WebApplicationInitializer` implementation) that declares Spring's `DispatcherServlet`.
- A Spring configuration that enables Spring MVC.
- A controller class that will respond to HTTP requests with "Hello World".
- A web application server, such as Tomcat, to deploy the application to.

Simple Example in Spring Boot



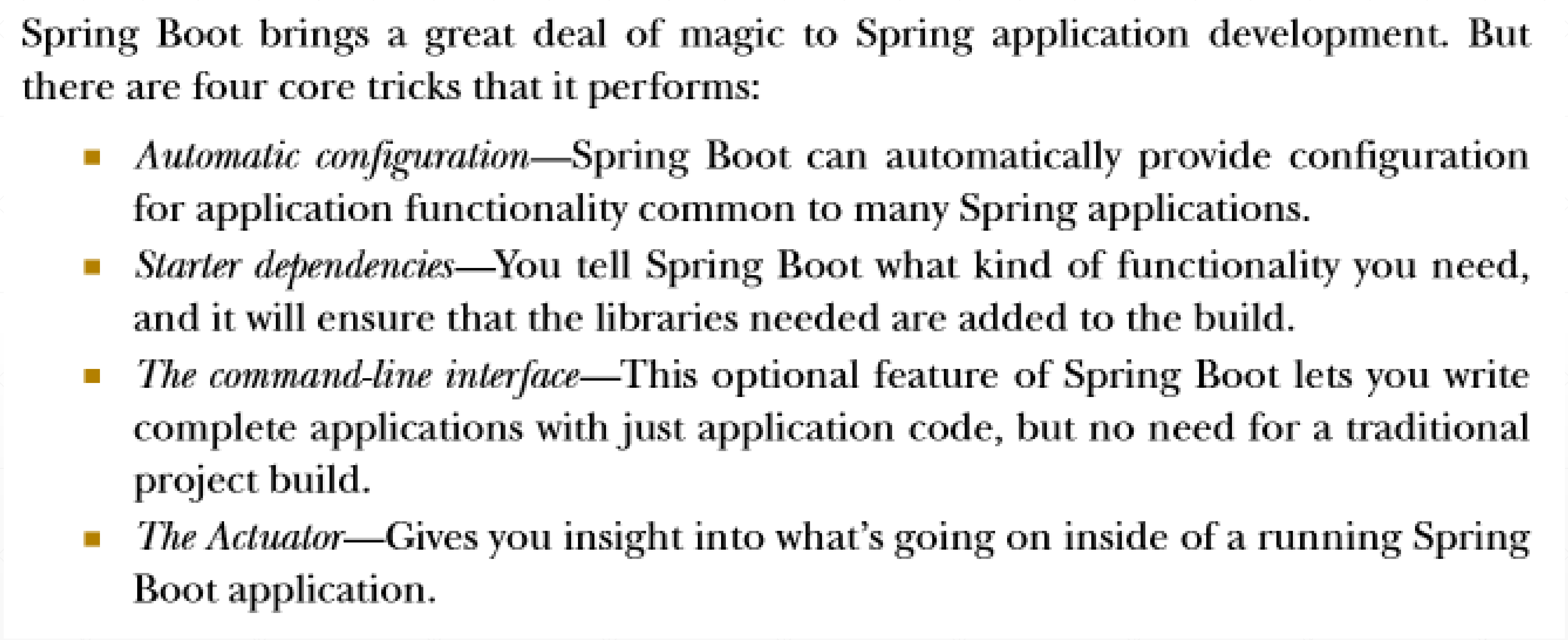
Listing 1.1 A complete Groovy-based Spring application

```
@RestController
class HelloController {

  @RequestMapping("/")
  def hello() {
    return "Hello World"
  }

}
```



There's no configuration. No web.xml. No build specification. Not even an application server. This is the entire application. Spring Boot will handle the logistics of executing the application. You only need to bring the application code.

Examining Spring Boot essentials

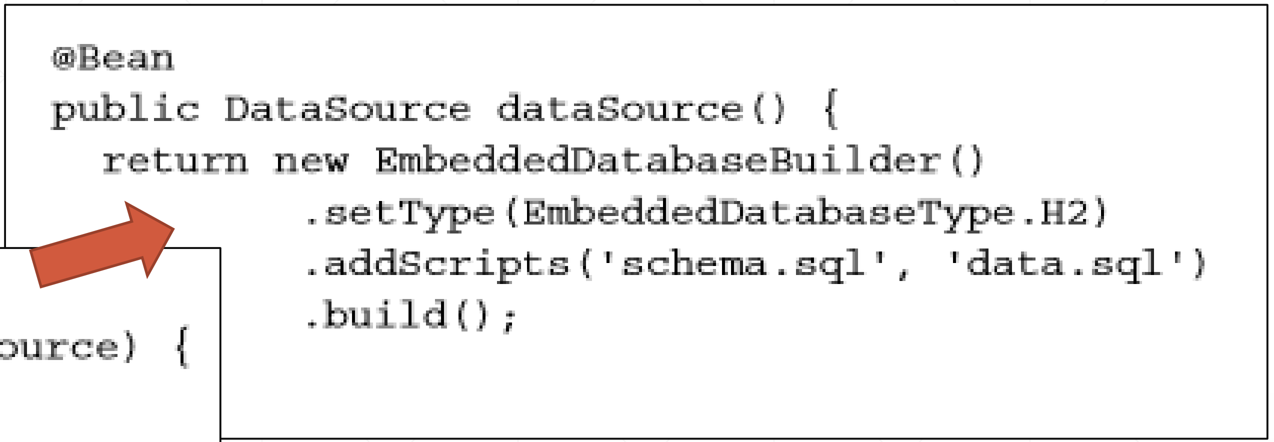


Spring Boot brings a great deal of magic to Spring application development. But there are four core tricks that it performs:

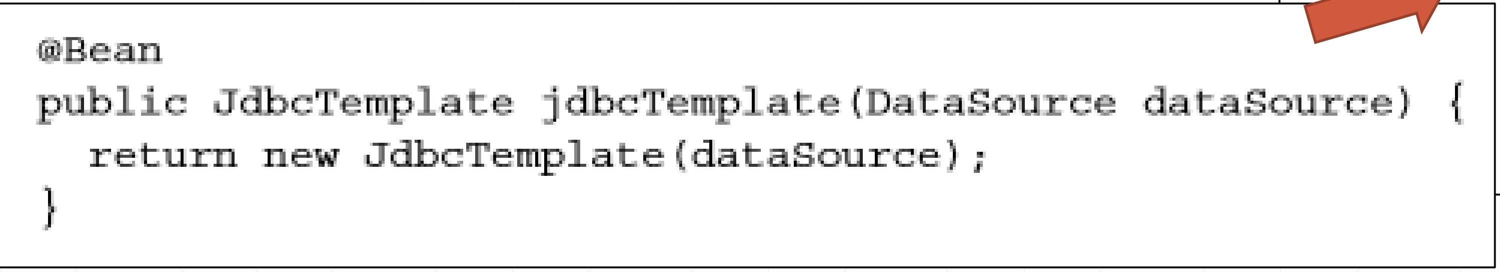
- *Automatic configuration*—Spring Boot can automatically provide configuration for application functionality common to many Spring applications.
- *Starter dependencies*—You tell Spring Boot what kind of functionality you need, and it will ensure that the libraries needed are added to the build.
- *The command-line interface*—This optional feature of Spring Boot lets you write complete applications with just application code, but no need for a traditional project build.
- *The Actuator*—Gives you insight into what's going on inside of a running Spring Boot application.

Auto configuration

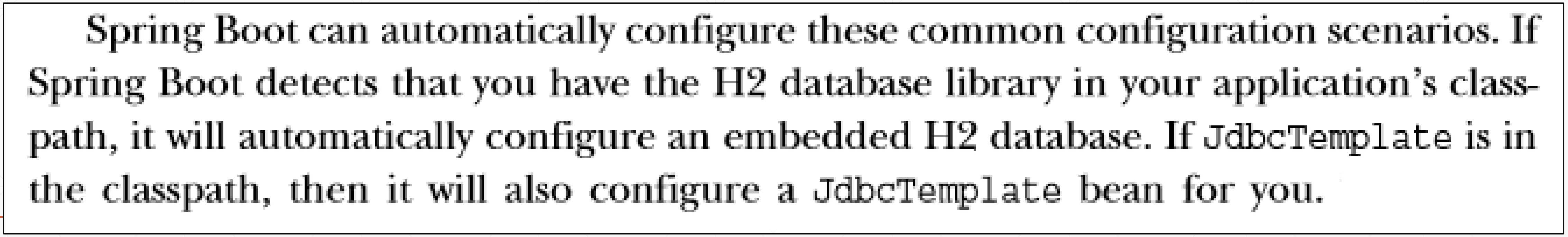
- Example java configuration in a Spring application:



```
@Bean
public DataSource dataSource() {
  return new EmbeddedDatabaseBuilder()
          .setType(EmbeddedDatabaseType.H2)
          .addScripts('schema.sql', 'data.sql')
          .build();
```



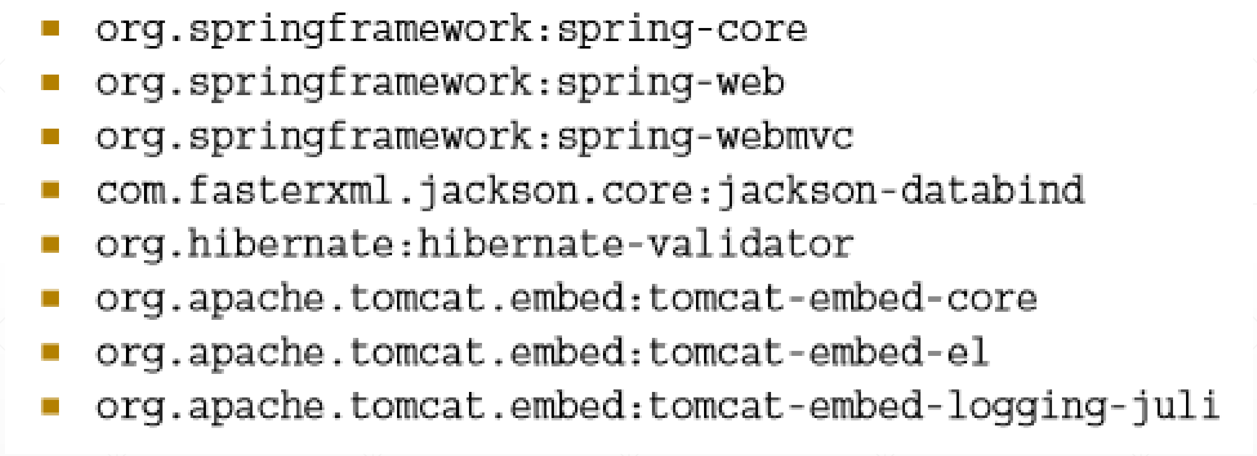
```
@Bean
public JdbcTemplate jdbcTemplate(DataSource dataSource) {
  return new JdbcTemplate(dataSource);
}
```



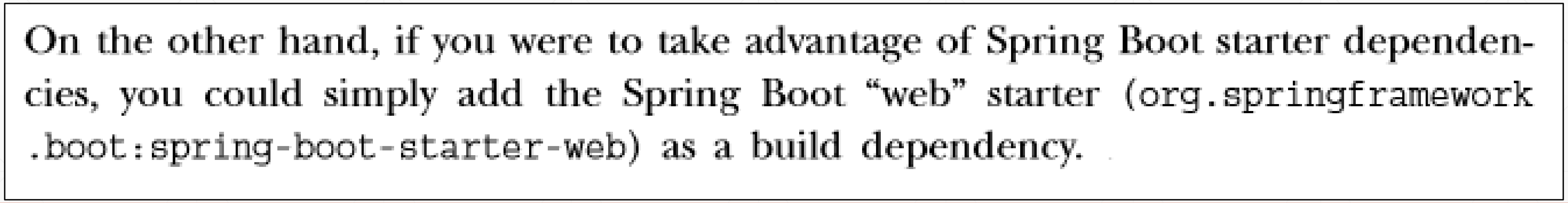
Spring Boot can automatically configure these common configuration scenarios. If Spring Boot detects that you have the H2 database library in your application's classpath, it will automatically configure an embedded H2 database. If `JdbcTemplate` is in the classpath, then it will also configure a `JdbcTemplate` bean for you.

Starter dependencies

- Example: for building a REST API with Spring MVC that works with JSON resource representations, you´ll need at least the following eight dependencies in your Maven or Gradle build:



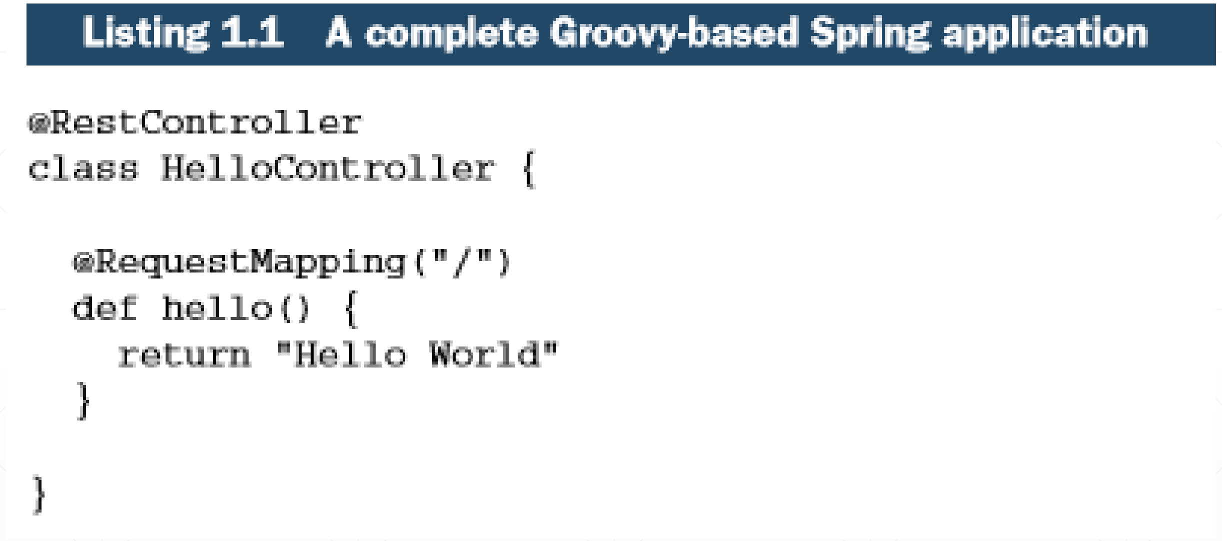
- `org.springframework:spring-core`
- `org.springframework:spring-web`
- `org.springframework:spring-webmvc`
- `com.fasterxml.jackson.core:jackson-databind`
- `org.hibernate:hibernate-validator`
- `org.apache.tomcat.embed:tomcat-embed-core`
- `org.apache.tomcat.embed:tomcat-embed-el`
- `org.apache.tomcat.embed:tomcat-embed-logging-juli`



On the other hand, if you were to take advantage of Spring Boot starter dependencies, you could simply add the Spring Boot "web" starter (`org.springframework.boot:spring-boot-starter-web`) as a build dependency.

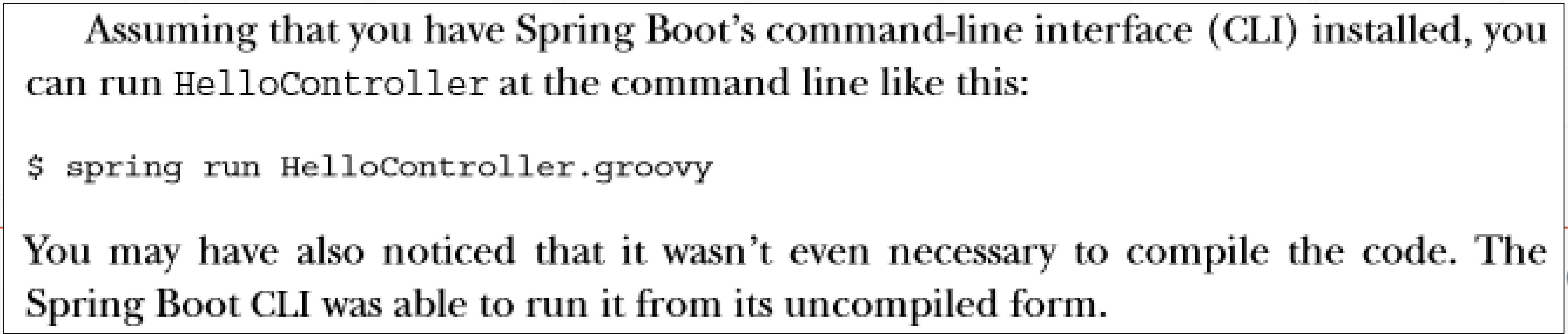
The command-line interface (CLI)

- Although it provides tremendous power and simplicity for Spring development, it also introduces a rather unconventional development model.



Listing 1.1 A complete Groovy-based Spring application

```
@RestController
class HelloController {

  @RequestMapping("/")
  def hello() {
    return "Hello World"
  }

}
```

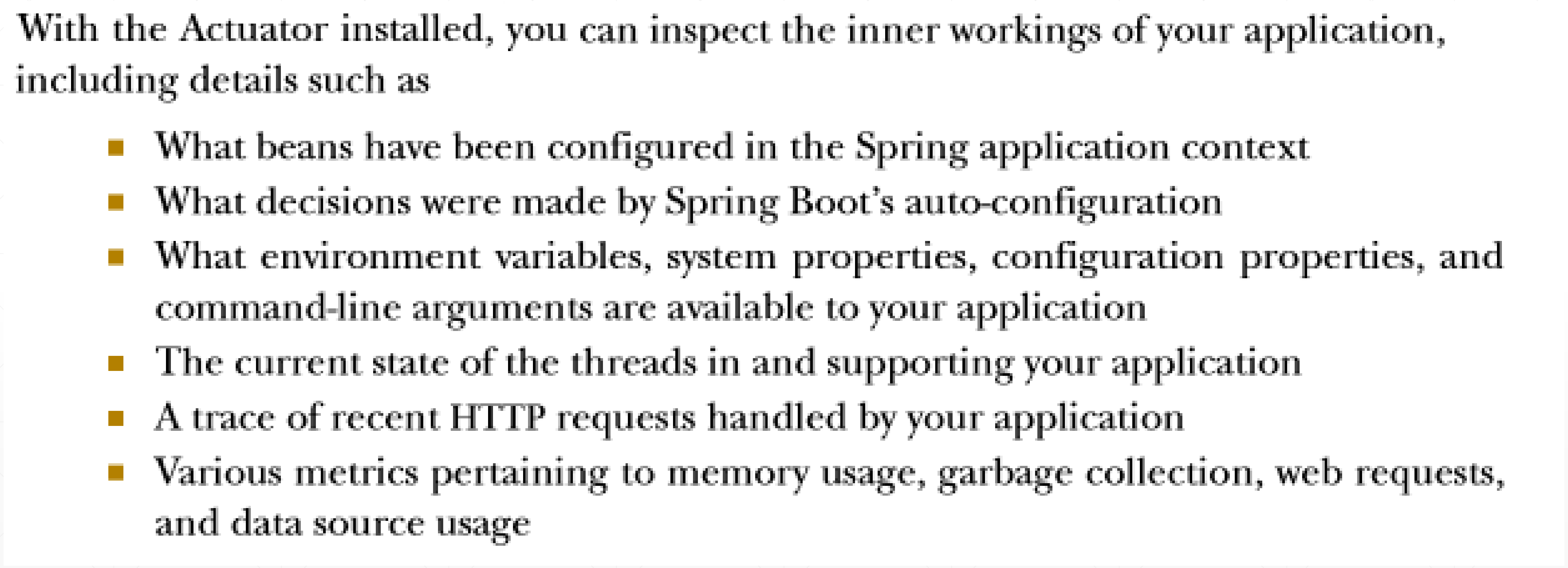


Assuming that you have Spring Boot's command-line interface (CLI) installed, you can run `HelloController` at the command line like this:

```
$ spring run HelloController.groovy
```

You may have also noticed that it wasn't even necessary to compile the code. The Spring Boot CLI was able to run it from its uncompiled form.

The actuator



With the Actuator installed, you can inspect the inner workings of your application, including details such as

- What beans have been configured in the Spring application context
- What decisions were made by Spring Boot's auto-configuration
- What environment variables, system properties, configuration properties, and command-line arguments are available to your application
- The current state of the threads in and supporting your application
- A trace of recent HTTP requests handled by your application
- Various metrics pertaining to memory usage, garbage collection, web requests, and data source usage

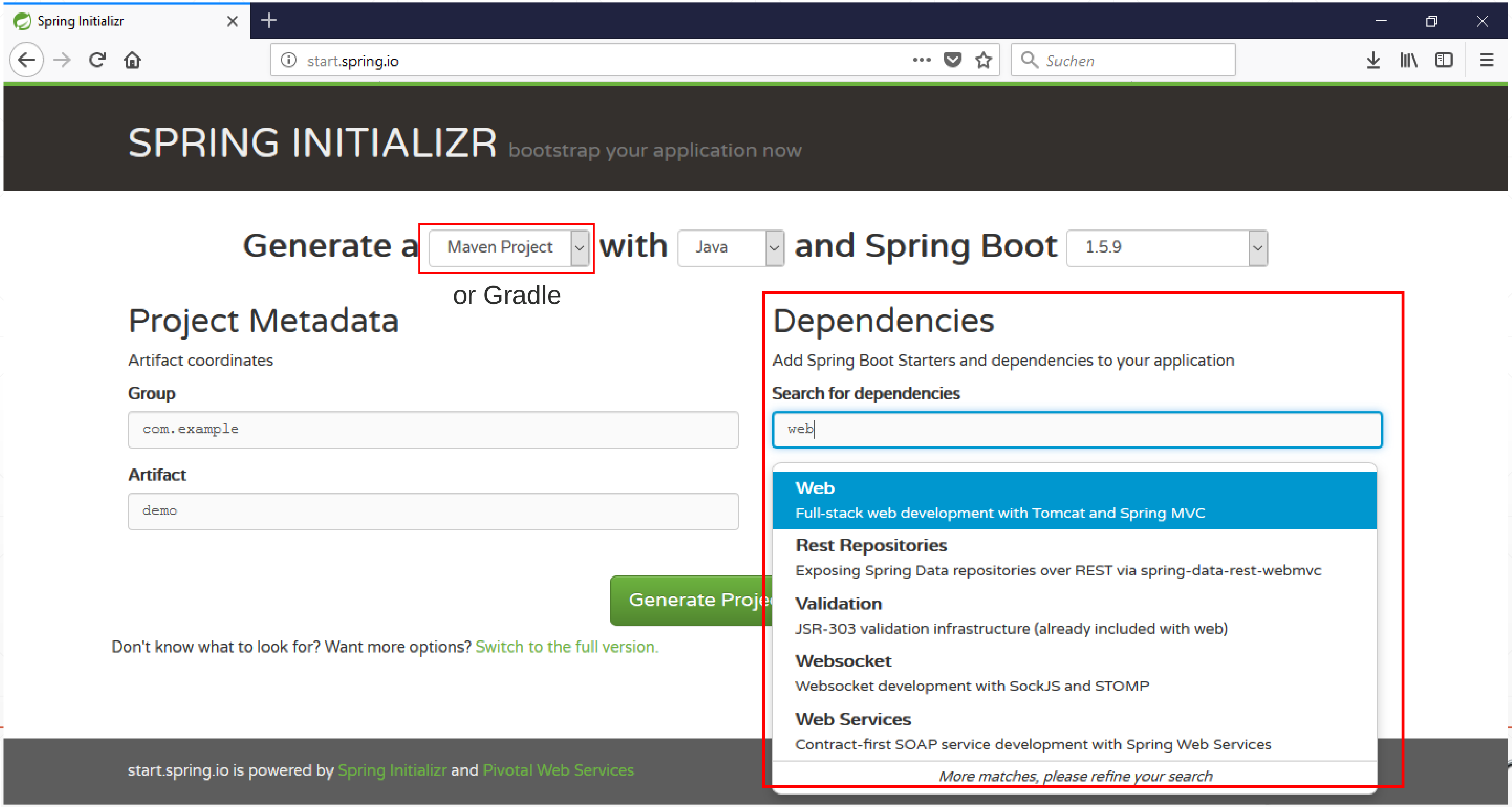
Getting started

- There are several ways to install the Spring Boot CLI:
  - From a downloaded distribution
  - Using the Groovy Environment Manager
  - With OS X Homebrew
  - As a port using MacPorts

Spring Initializr

- Is used to create the Spring Boot project structure
- Spring Initializr can be used in several ways:
  - Through a web-based interface
  - Via Spring Tool Suite
  - Via IntelliJ IDEA
  - Using the Spring Boot CL

Using spring Initializr's web interface



Spring Initializr

start.spring.io

Suchen

SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Java and Spring Boot 1.5.9

or Gradle

Project Metadata

Artifact coordinates

Group

com.example

Artifact

demo

Generate Project

Don't know what to look for? Want more options? Switch to the full version.

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

web

Web
Full-stack web development with Tomcat and Spring MVC

Rest Repositories
Exposing Spring Data repositories over REST via spring-data-rest-webmvc

Validation
JSR-303 validation infrastructure (already included with web)

Websocket
Websocket development with SockJS and STOMP

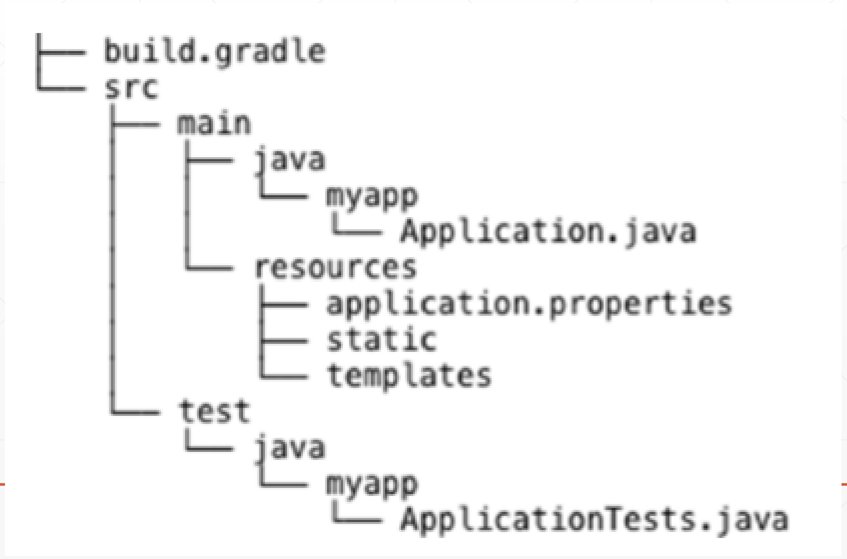
Web Services
Contract-first SOAP service development with Spring Web Services

More matches, please refine your search

start.spring.io is powered by Spring Initializr and Pivotal Web Services

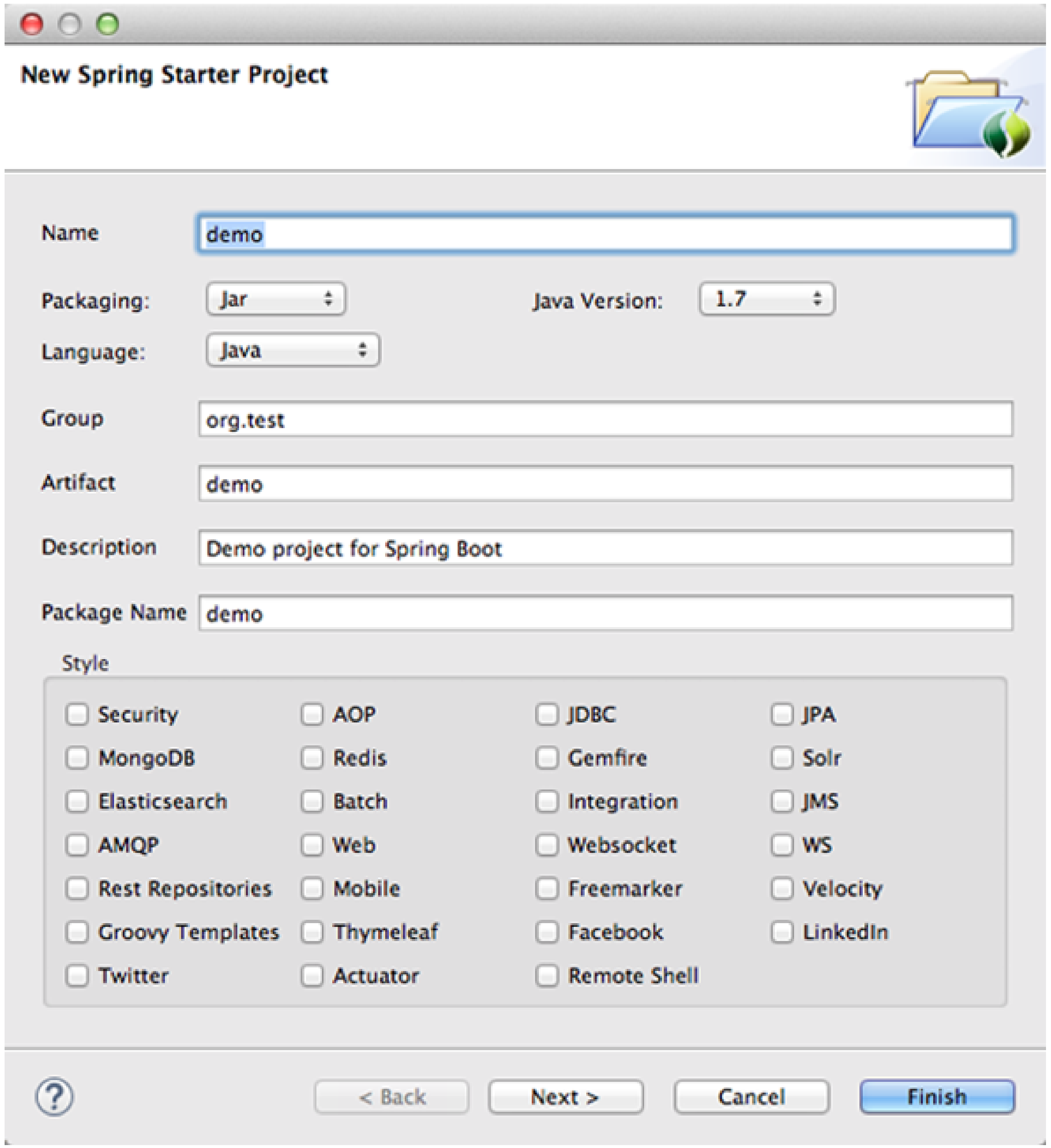
Using spring Initializr's web interface

- For example, suppose that you were to specify the following to Spring Initializr:
  - Artifact: myapp
  - Package Name: myapp
  - Type: Gradle Project
  - Dependencies: Web and JPA
- The downloaded project structure would look like the following:



```
├── build.gradle
└── src
    ├── main
    │   ├── java
    │   │   └── myapp
    │   │       └── Application.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── myapp
                └── ApplicationTests.java
```

Creating Spring Boot projects in spring tool suite



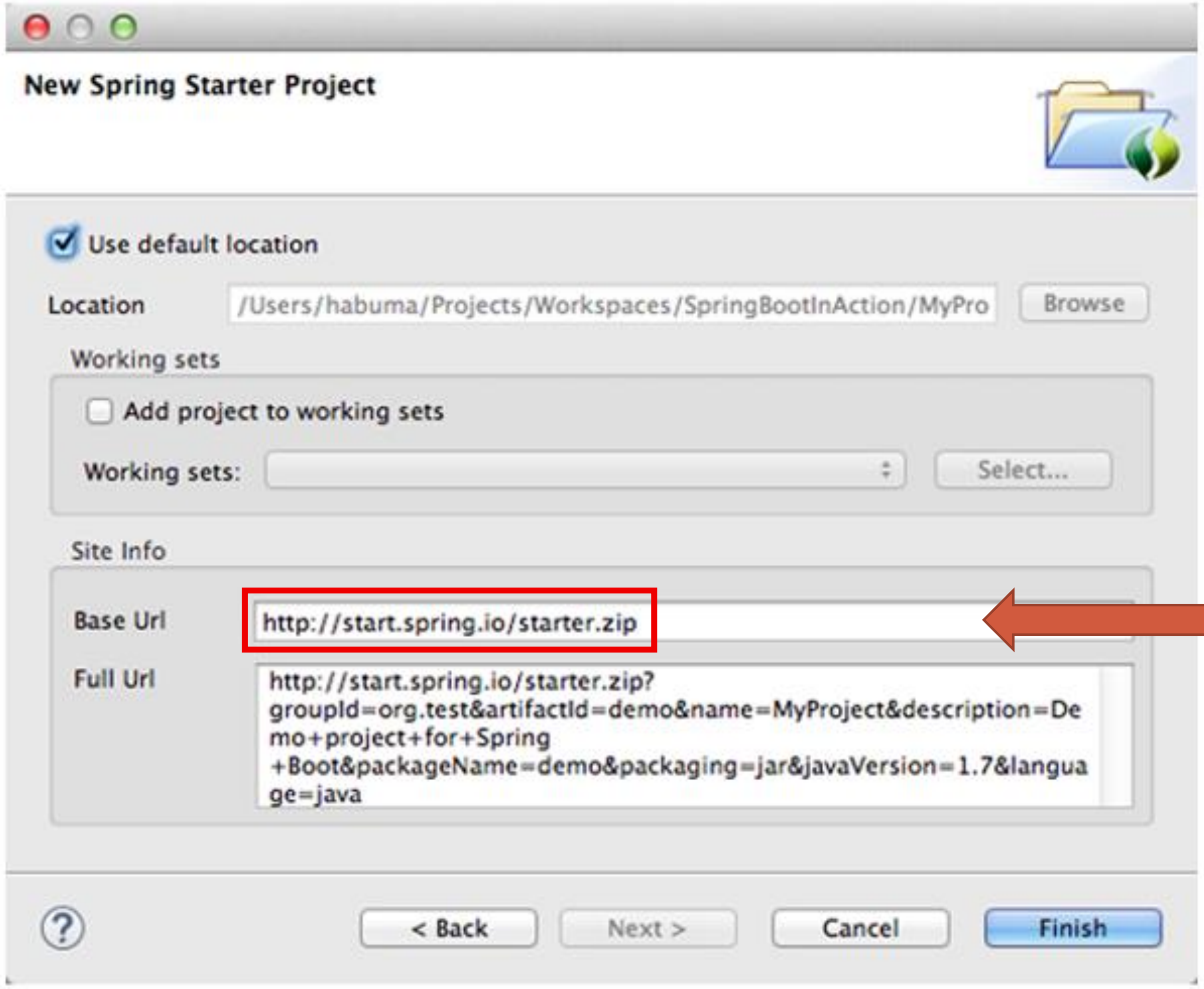
New Spring Starter Project

Name	demo
Packaging:	Jar
Java Version:	1.7
Language:	Java
Group	org.test
Artifact	demo
Description	Demo project for Spring Boot
Package Name	demo

Style

☐ Security	☐ AOP	☐ JDBC	☐ JPA
☐ MongoDB	☐ Redis	☐ Gemfire	☐ Solr
☐ Elasticsearch	☐ Batch	☐ Integration	☐ JMS
☐ AMQP	☐ Web	☐ Websocket	☐ WS
☐ Rest Repositories	☐ Mobile	☐ Freemarker	☐ Velocity
☐ Groovy Templates	☐ Thymeleaf	☐ Facebook	☐ LinkedIn
☐ Twitter	☐ Actuator	☐ Remote Shell	

Creating Spring Boot projects in Spring tool suite



New Spring Starter Project

☑ Use default location

Location: /Users/habuma/Projects/Workspaces/SpringBootInAction/MyPro

Working sets

☐ Add project to working sets

Working sets:

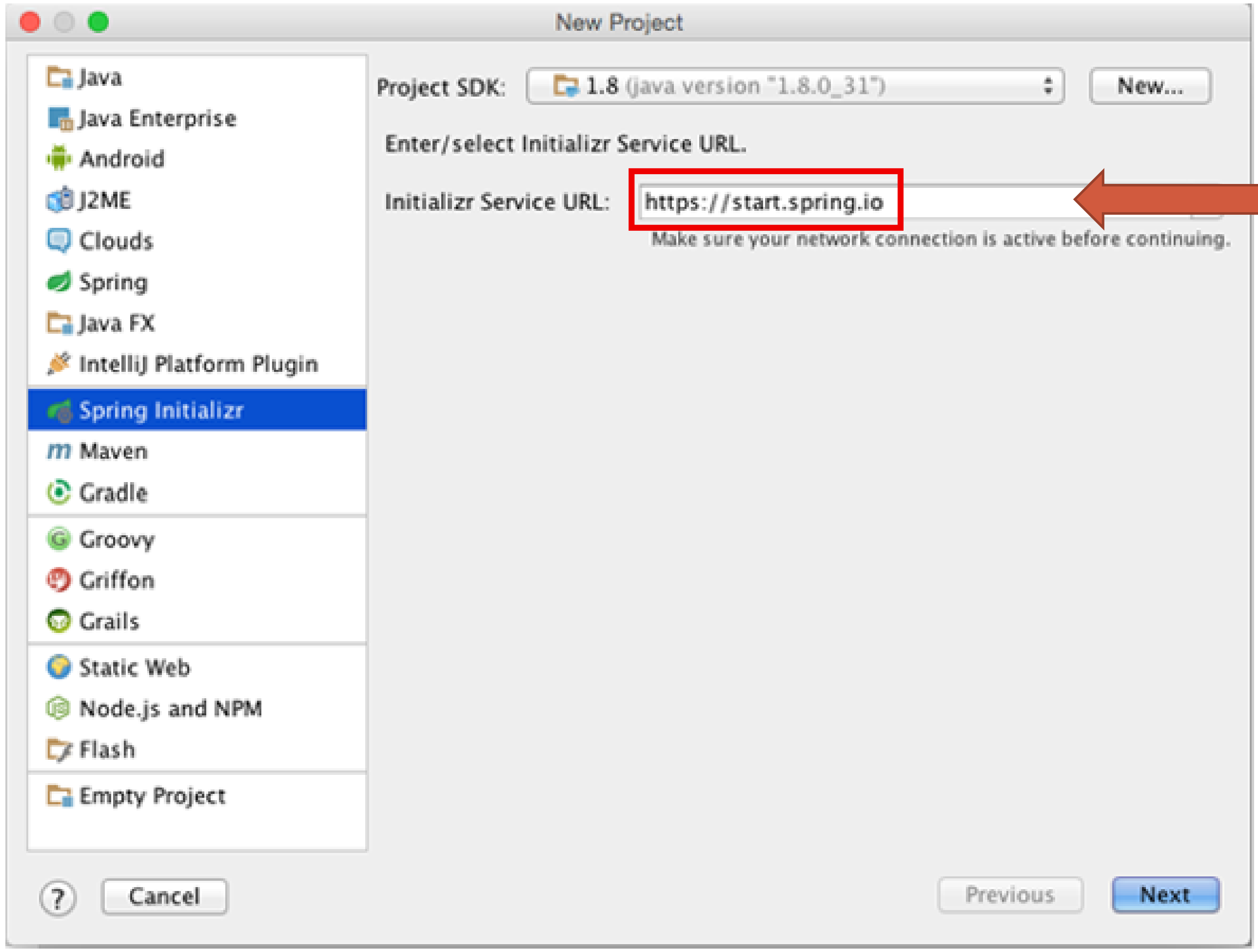
Site Info

Base Url: http://start.spring.io/starter.zip

Full Url: http://start.spring.io/starter.zip?groupId=org.test&artifactId=demo&name=MyProject&description=Demo+project+for+Spring+Boot&packageName=demo&packaging=jar&javaVersion=1.7&language=java

Spring Tool Suite integrates with Spring Initializr to create and directly import Spring Boot projects into the IDE.

Creating spring boot projects in IntelliJ idea



Project SDK: 1.8 (java version "1.8.0_31")

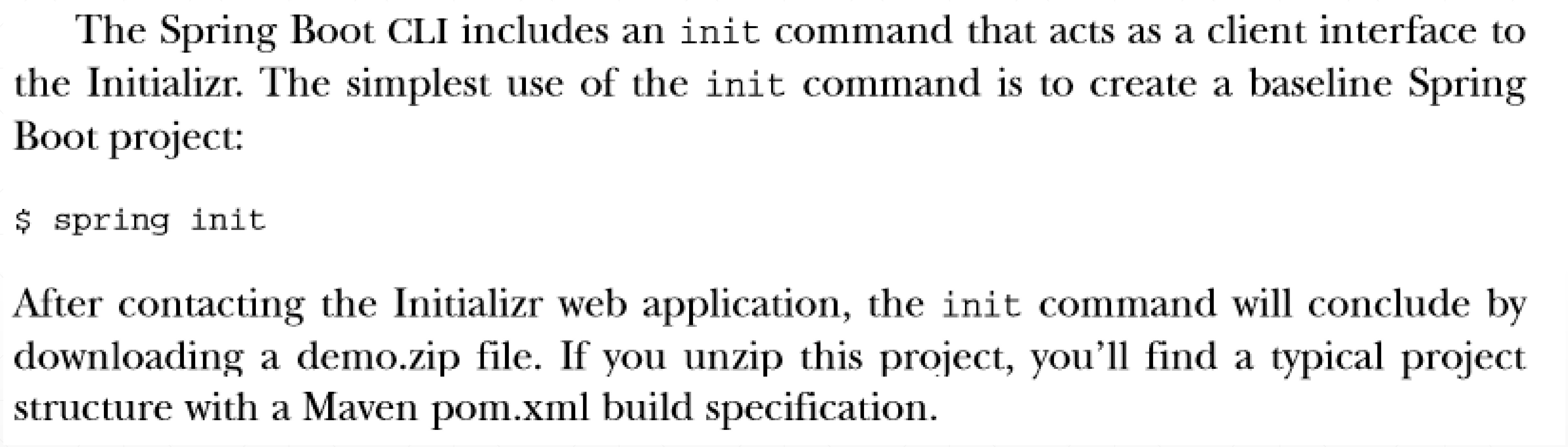
Enter/select Initializr Service URL.

Initializr Service URL: https://start.spring.io

Make sure your network connection is active before continuing.

Same here

Using the Initializr from the Spring Boot CLI

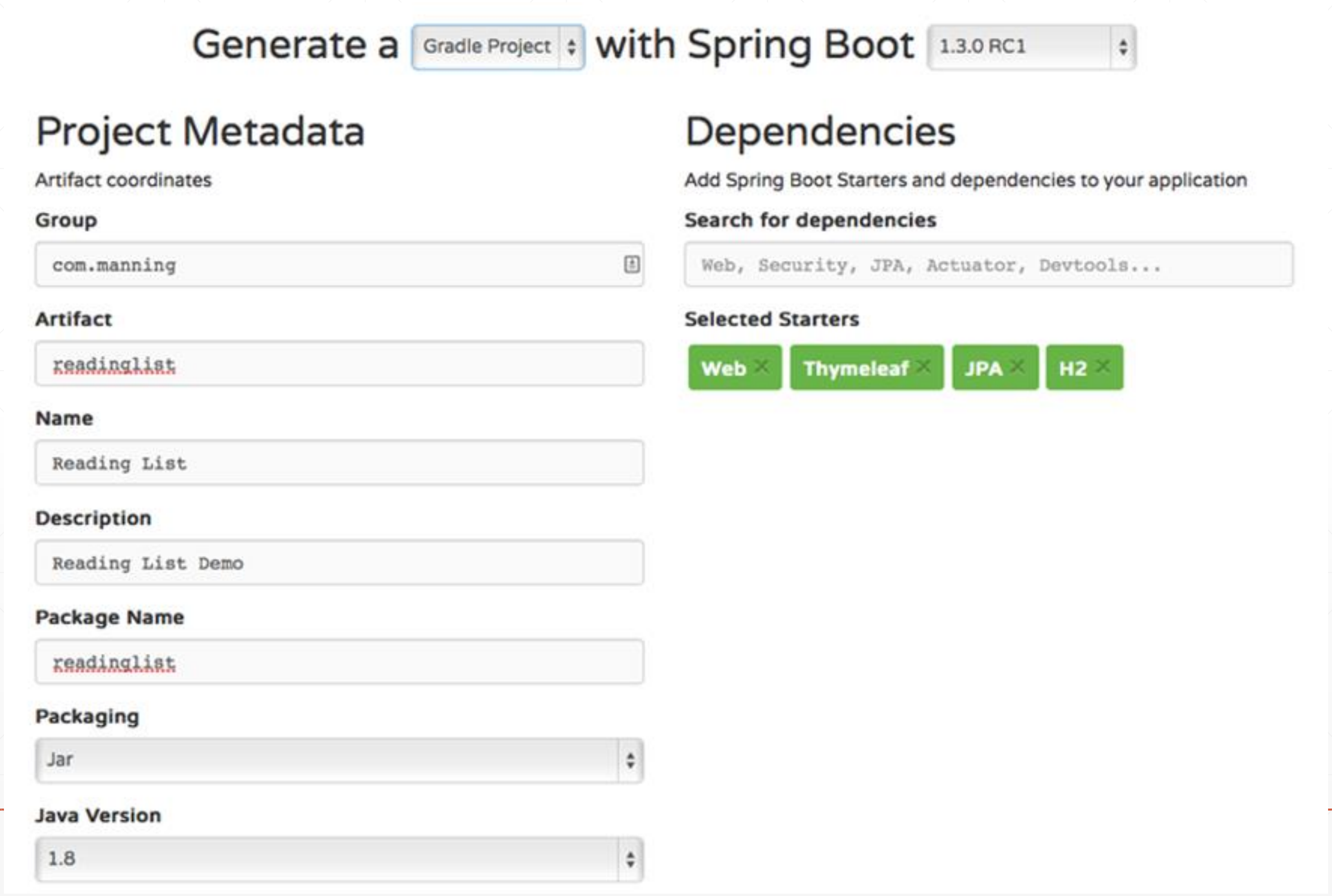


The Spring Boot CLI includes an `init` command that acts as a client interface to the Initializr. The simplest use of the `init` command is to create a baseline Spring Boot project:

```
$ spring init
```

After contacting the Initializr web application, the `init` command will conclude by downloading a demo.zip file. If you unzip this project, you'll find a typical project structure with a Maven pom.xml build specification.

Putting Spring Boot to work



Generate a Gradle Project with Spring Boot 1.3.0 RC1

Project Metadata

Artifact coordinates

Group	com.manning
Artifact	readinglist
Name	Reading List
Description	Reading List Demo
Package Name	readinglist
Packaging	Jar
Java Version	1.8

Dependencies

Add Spring Boot Starters and dependencies to your application

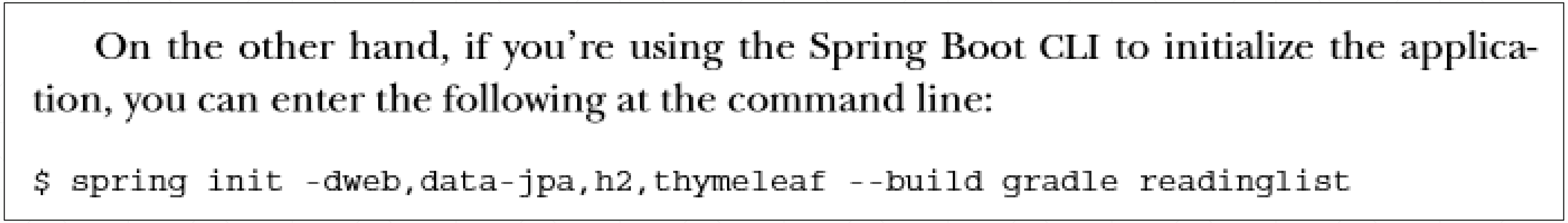
Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Starters

Web ✕ Thymeleaf ✕ JPA ✕ H2 ✕

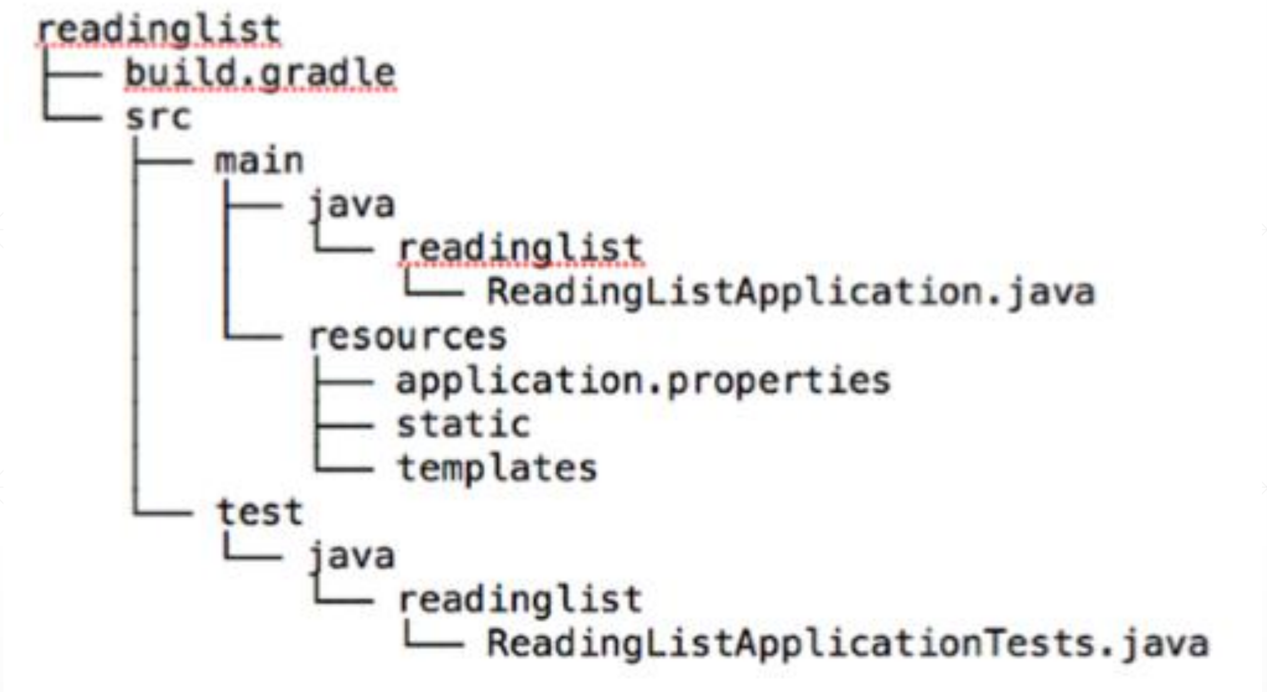
Putting Spring Boot to work



On the other hand, if you're using the Spring Boot CLI to initialize the application, you can enter the following at the command line:

```
$ spring init -dweb,data-jpa,h2,thymeleaf --build gradle readinglist
```

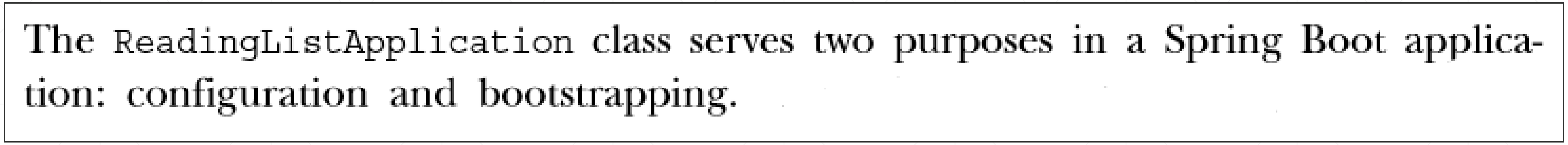
Putting Spring Boot to work



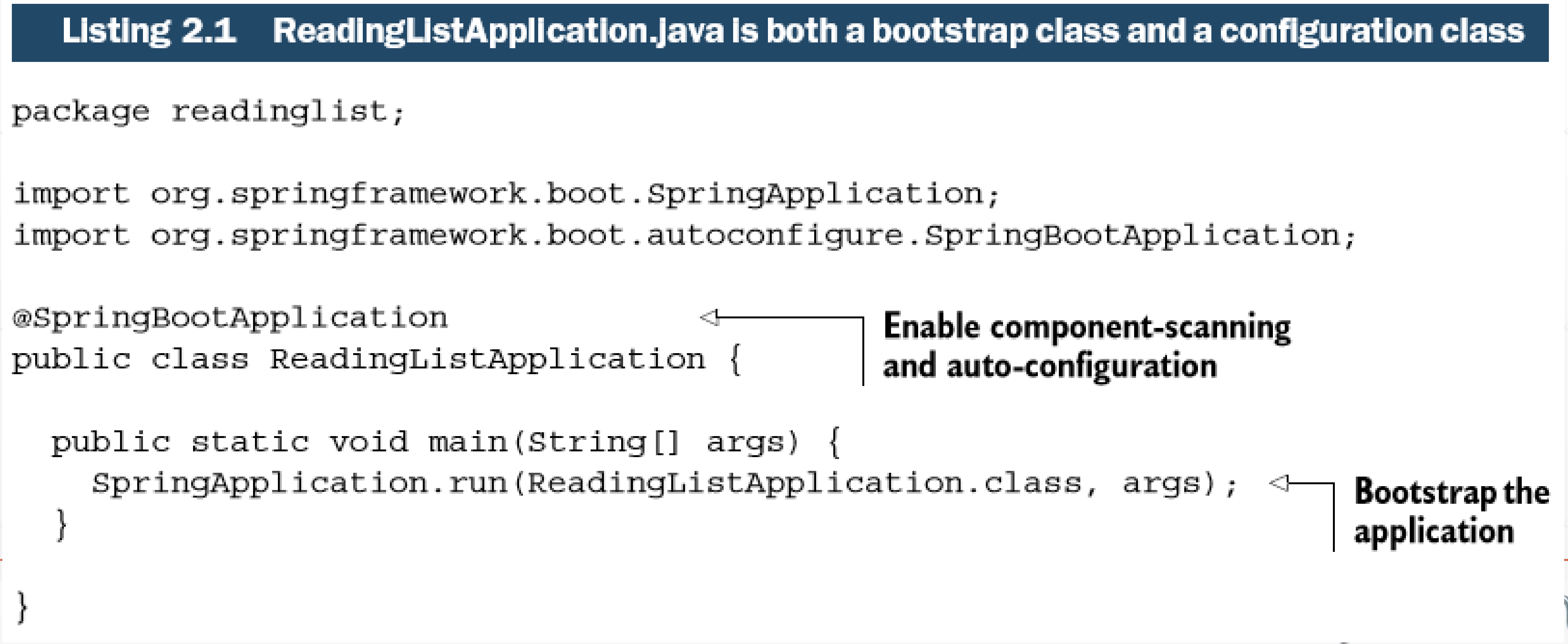
```
readinglist
├── build.gradle
└── src
    ├── main
    │   ├── java
    │   │   └── readinglist
    │   │       └── ReadingListApplication.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── readinglist
                └── ReadingListApplicationTests.java
```

- build.gradle—The Gradle build specification
- ReadingListApplication.java—The application's bootstrap class and primary Spring configuration class
- application.properties—A place to configure application and Spring Boot properties
- ReadingListApplicationTests.java—A basic integration test class

Bootstrapping Spring



The ReadingListApplication class serves two purposes in a Spring Boot application: configuration and bootstrapping.



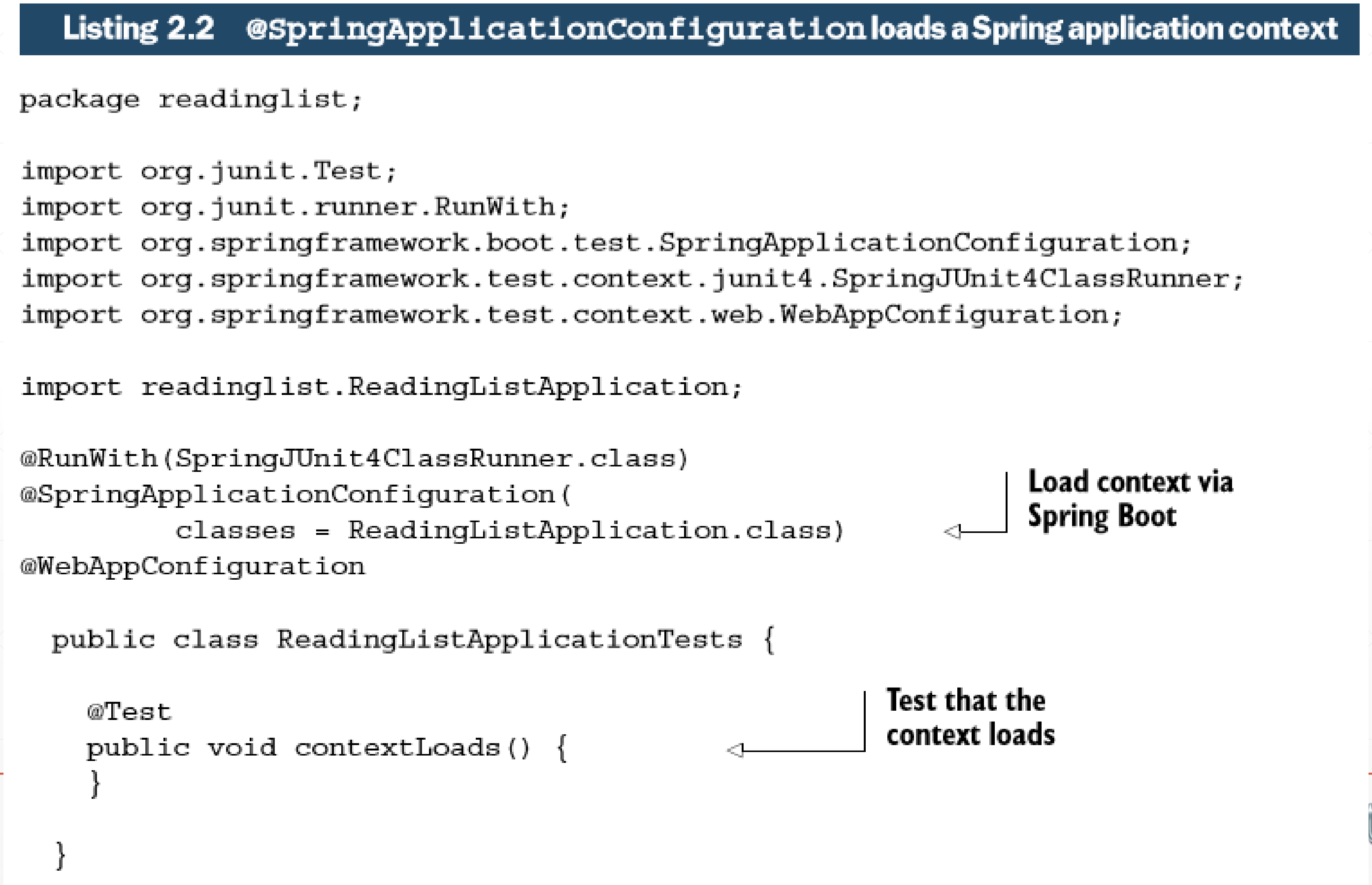
Listing 2.1 ReadingListApplication.java is both a bootstrap class and a configuration class

```
package readinglist;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class ReadingListApplication {

  public static void main(String[] args) {
    SpringApplication.run(ReadingListApplication.class, args);
  }

}
```

Enable component-scanning and auto-configuration

Bootstrap the application

Testing Spring Boot applications



Listing 2.2 `@SpringApplicationConfiguration` loads a Spring application context

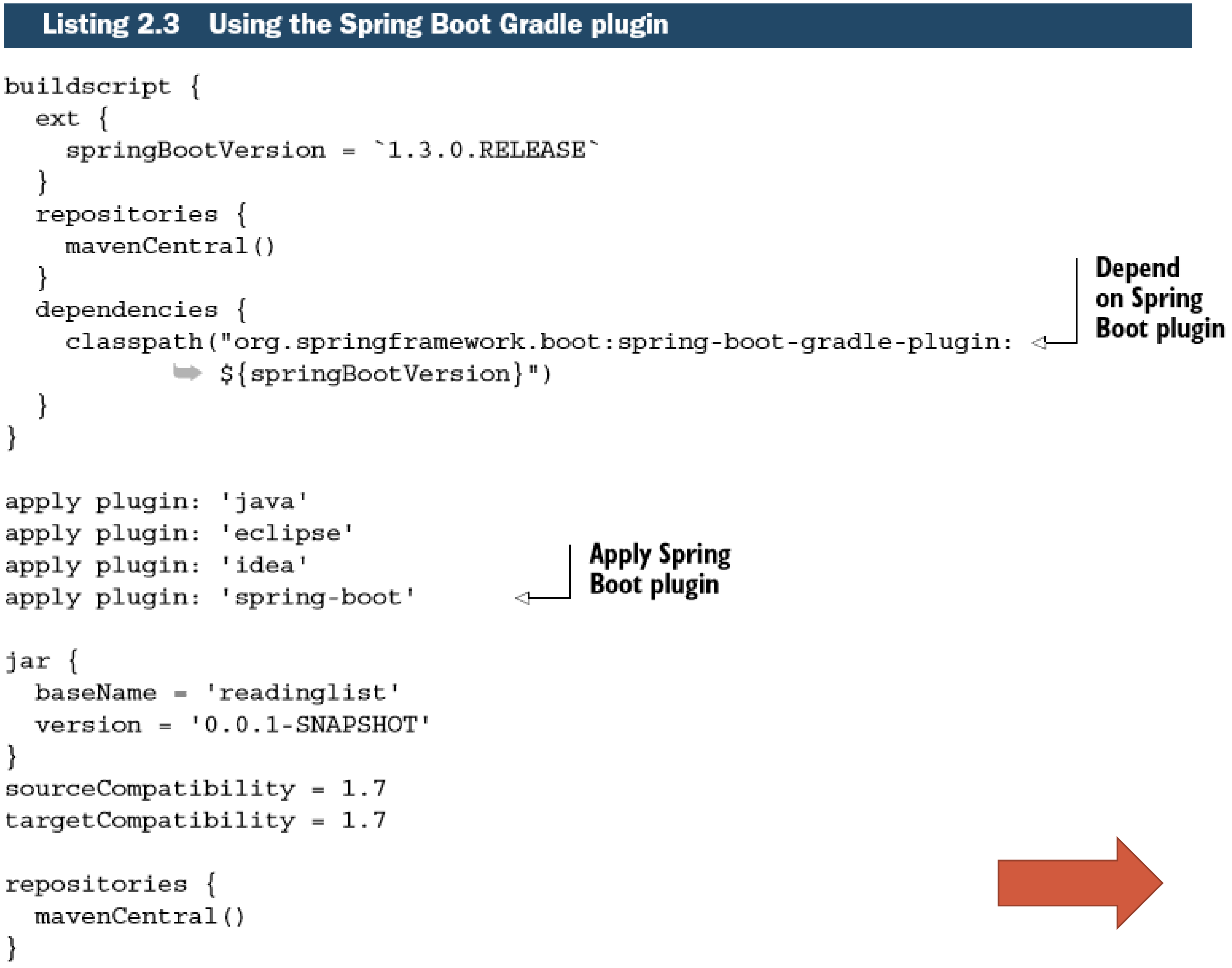
```
package readinglist;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.web.WebAppConfiguration;

import readinglist.ReadingListApplication;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(
         classes = ReadingListApplication.class)
@WebAppConfiguration

  public class ReadingListApplicationTests {

    @Test
    public void contextLoads() {
    }

  }
```

Load context via Spring Boot

Test that the context loads

Dissecting a Spring Boot project build

Using Gradle plugin



Listing 2.3 Using the Spring Boot Gradle plugin

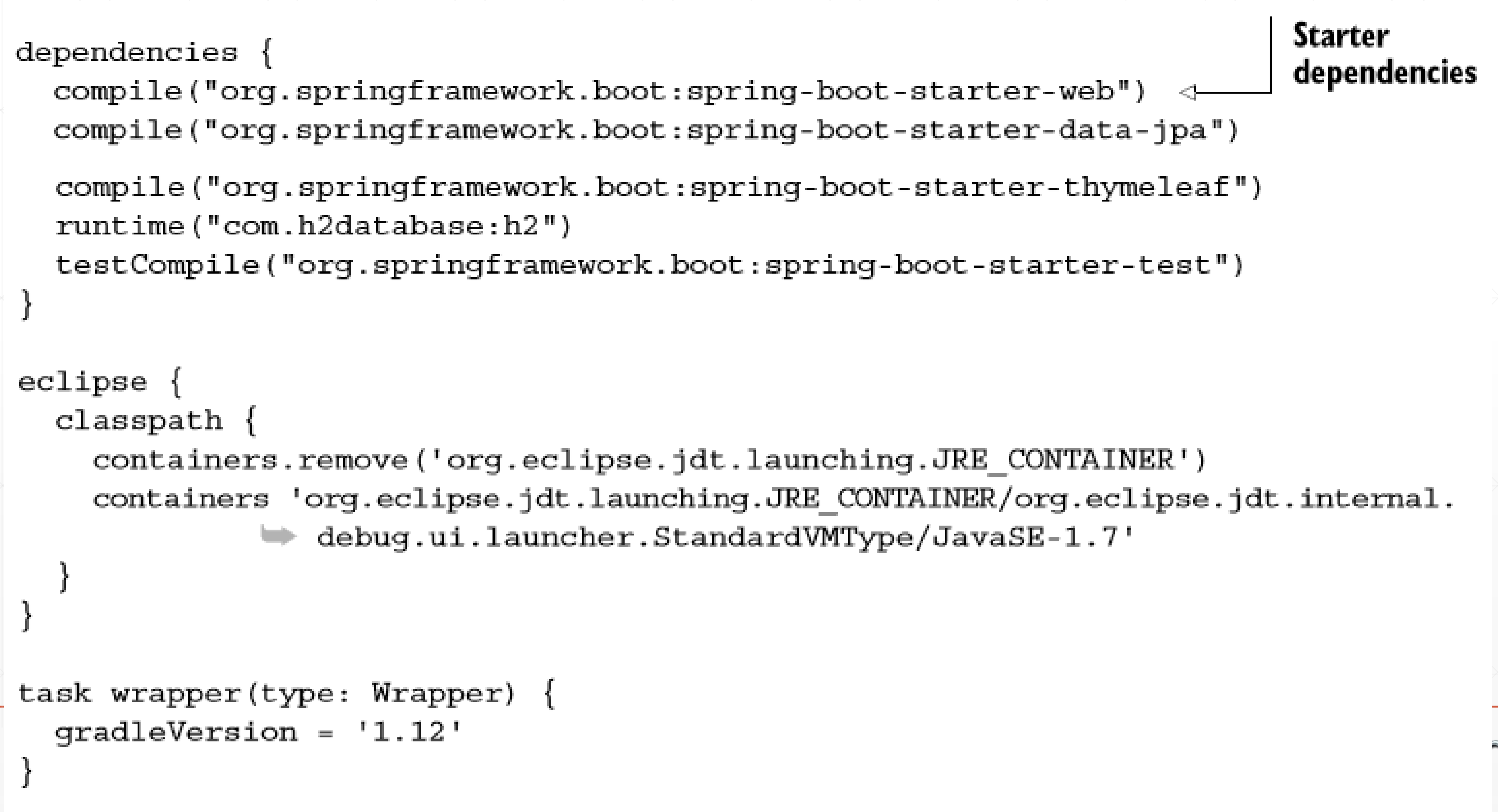
```
buildscript {
  ext {
    springBootVersion = `1.3.0.RELEASE`
  }
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:
           ➥ ${springBootVersion}")
  }
}

apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'idea'
apply plugin: 'spring-boot'

jar {
  baseName = 'readinglist'
  version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.7
targetCompatibility = 1.7

repositories {
  mavenCentral()
}
```

Depend on Spring Boot plugin

Apply Spring Boot plugin

Dissecting a Spring Boot project build

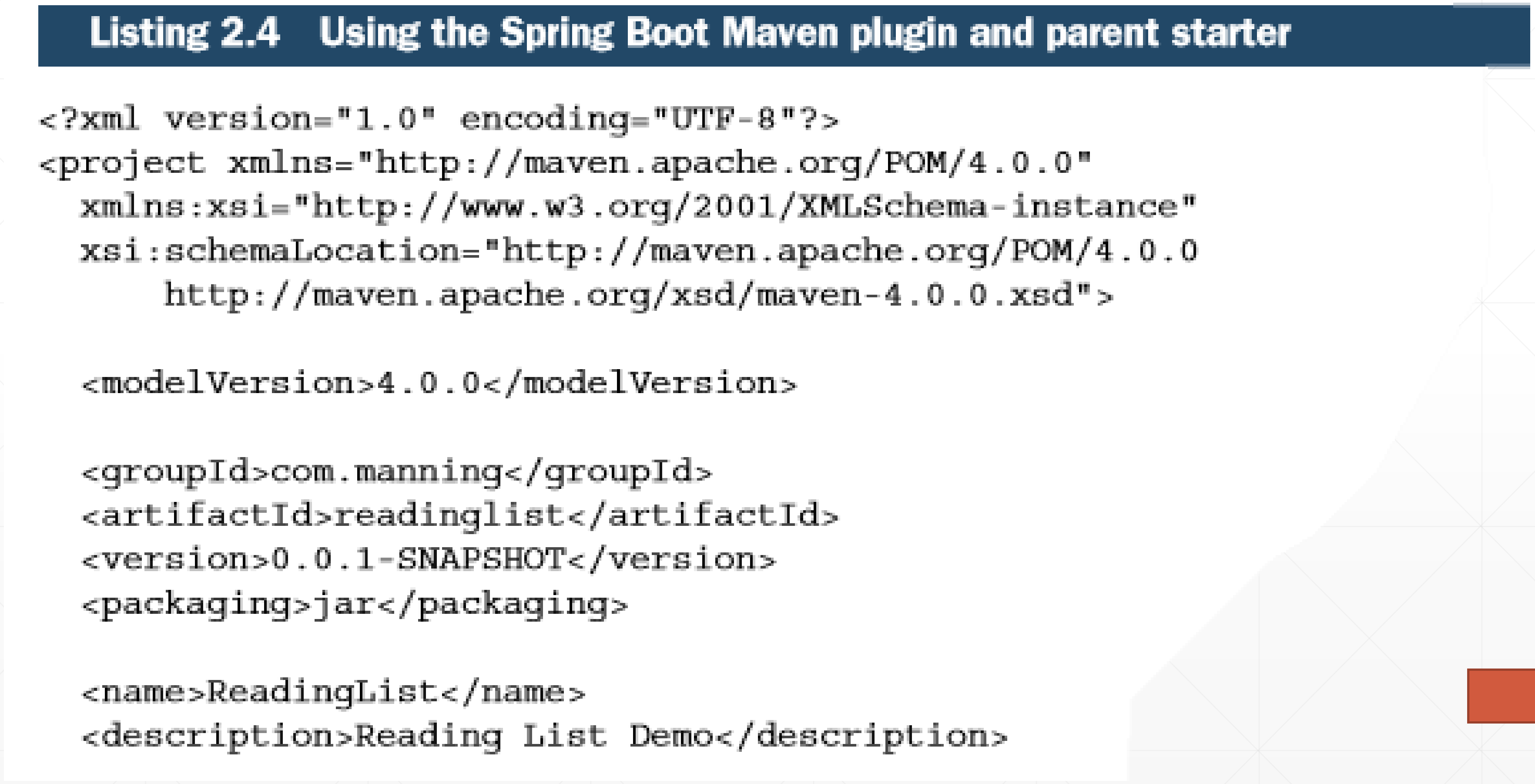
Using Gradle plugin



```
dependencies {
  compile("org.springframework.boot:spring-boot-starter-web")  ◁
  compile("org.springframework.boot:spring-boot-starter-data-jpa")

  compile("org.springframework.boot:spring-boot-starter-thymeleaf")
  runtime("com.h2database:h2")
  testCompile("org.springframework.boot:spring-boot-starter-test")
}

eclipse {
  classpath {
    containers.remove('org.eclipse.jdt.launching.JRE_CONTAINER')
    containers 'org.eclipse.jdt.launching.JRE_CONTAINER/org.eclipse.jdt.internal.
            ➥ debug.ui.launcher.StandardVMType/JavaSE-1.7'
  }
}

task wrapper(type: Wrapper) {
  gradleVersion = '1.12'
}
```

Dissecting a Spring Boot project build

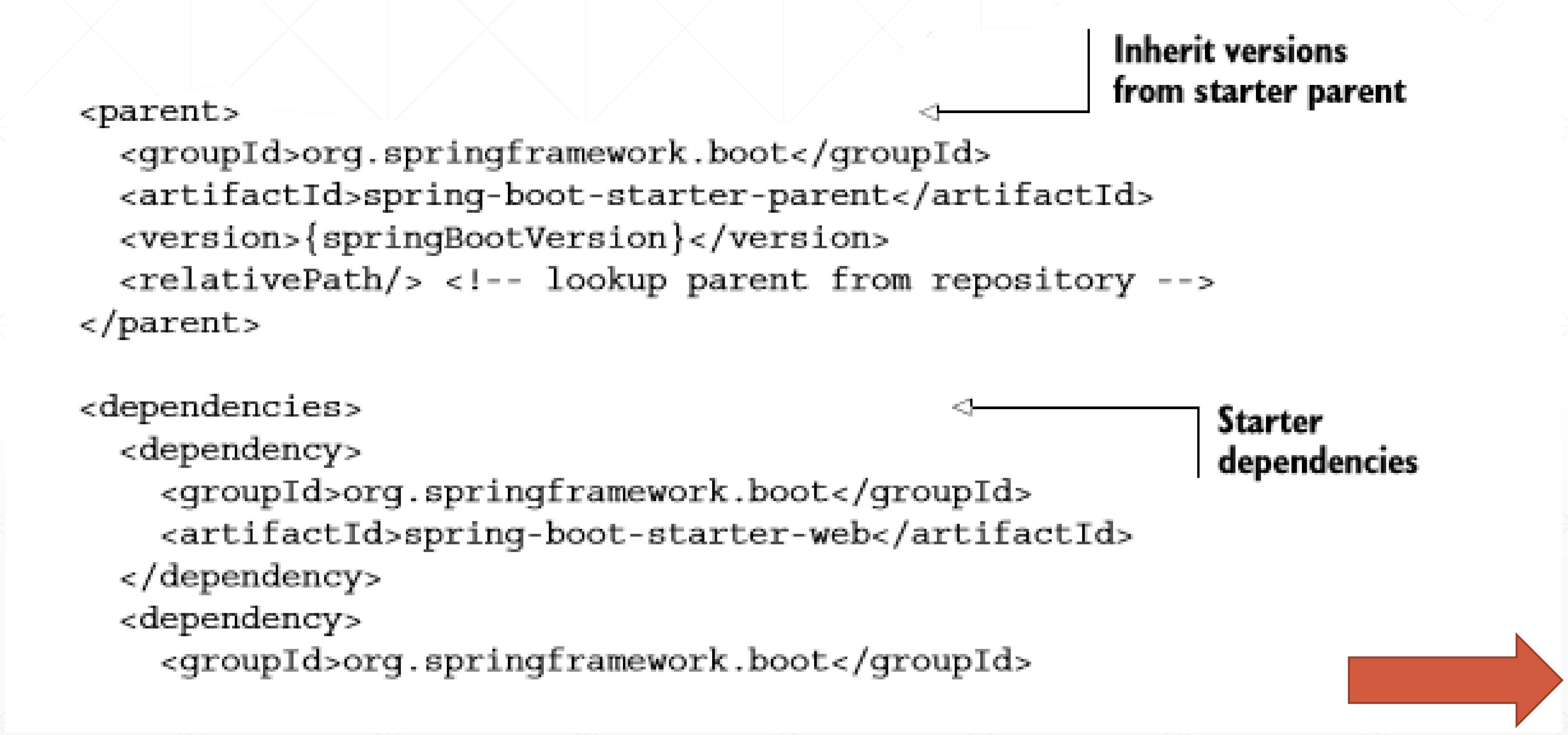
Using Maven plugin:



Listing 2.4 Using the Spring Boot Maven plugin and parent starter

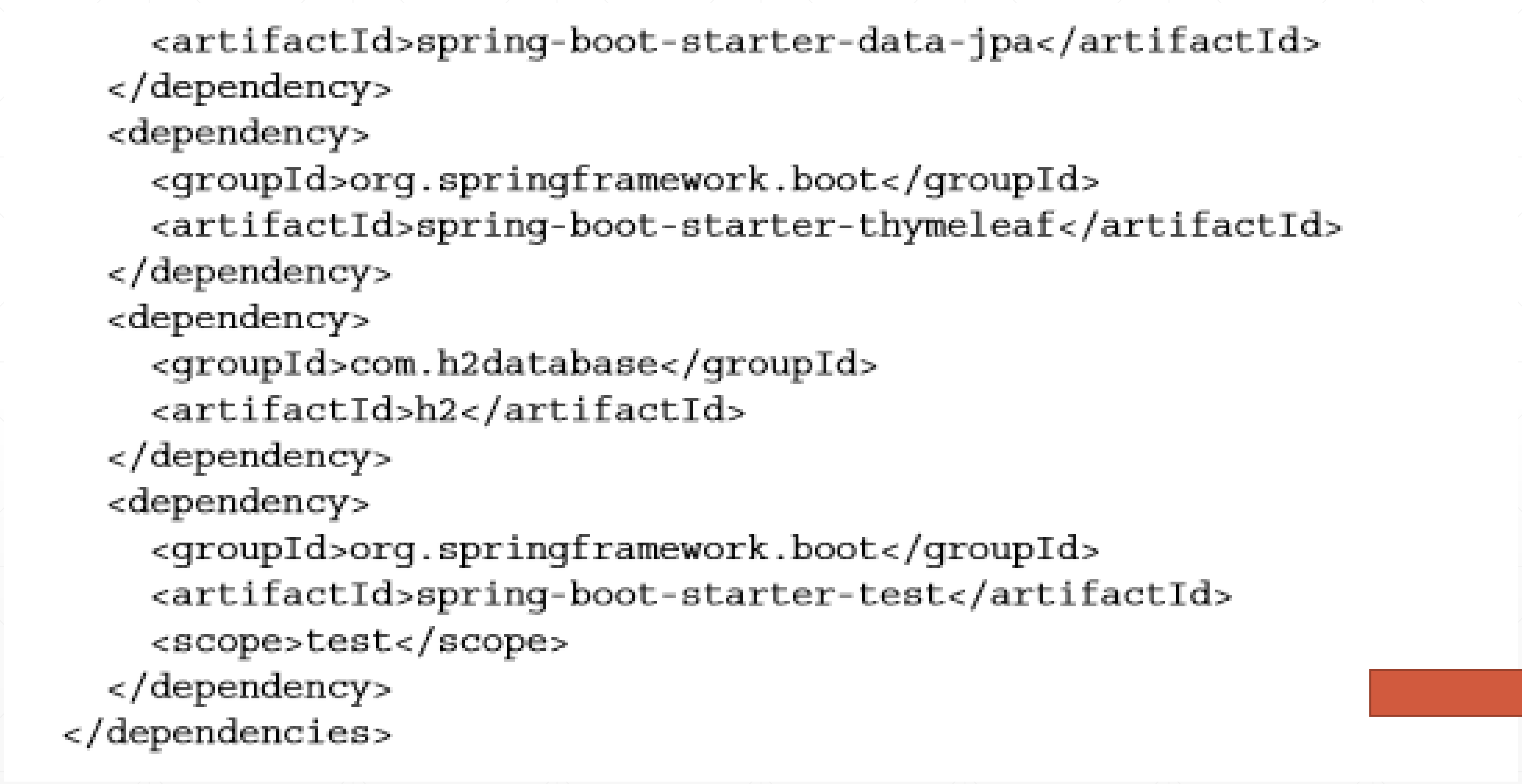
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
      http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.manning</groupId>
  <artifactId>readinglist</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>ReadingList</name>
  <description>Reading List Demo</description>
```

Dissecting a Spring Boot project build


Inherit versions from starter parent
Starter dependencies

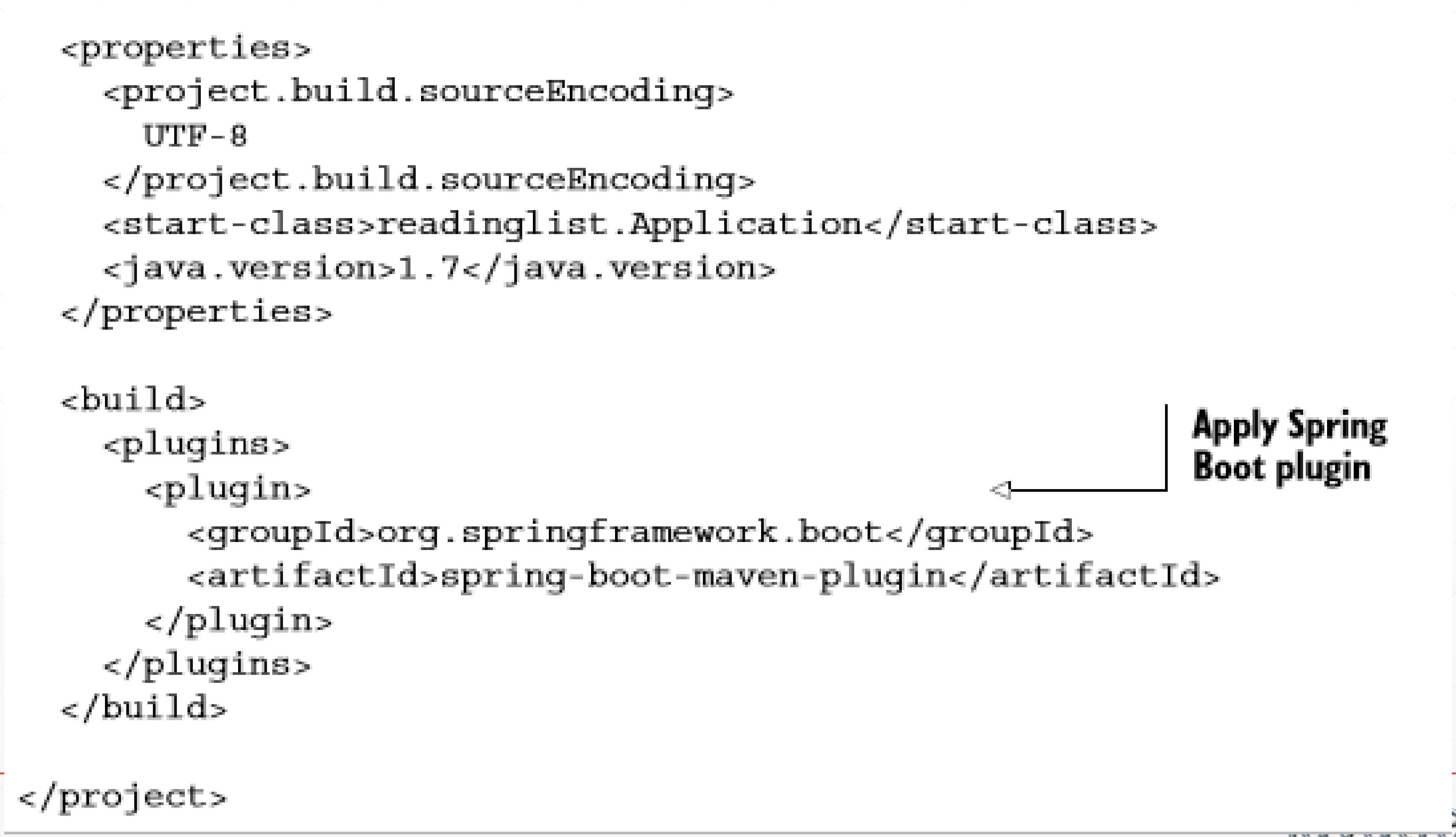
```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>{springBootVersion}</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
```

Dissecting a Spring Boot project build



```
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Dissecting a Spring Boot project build

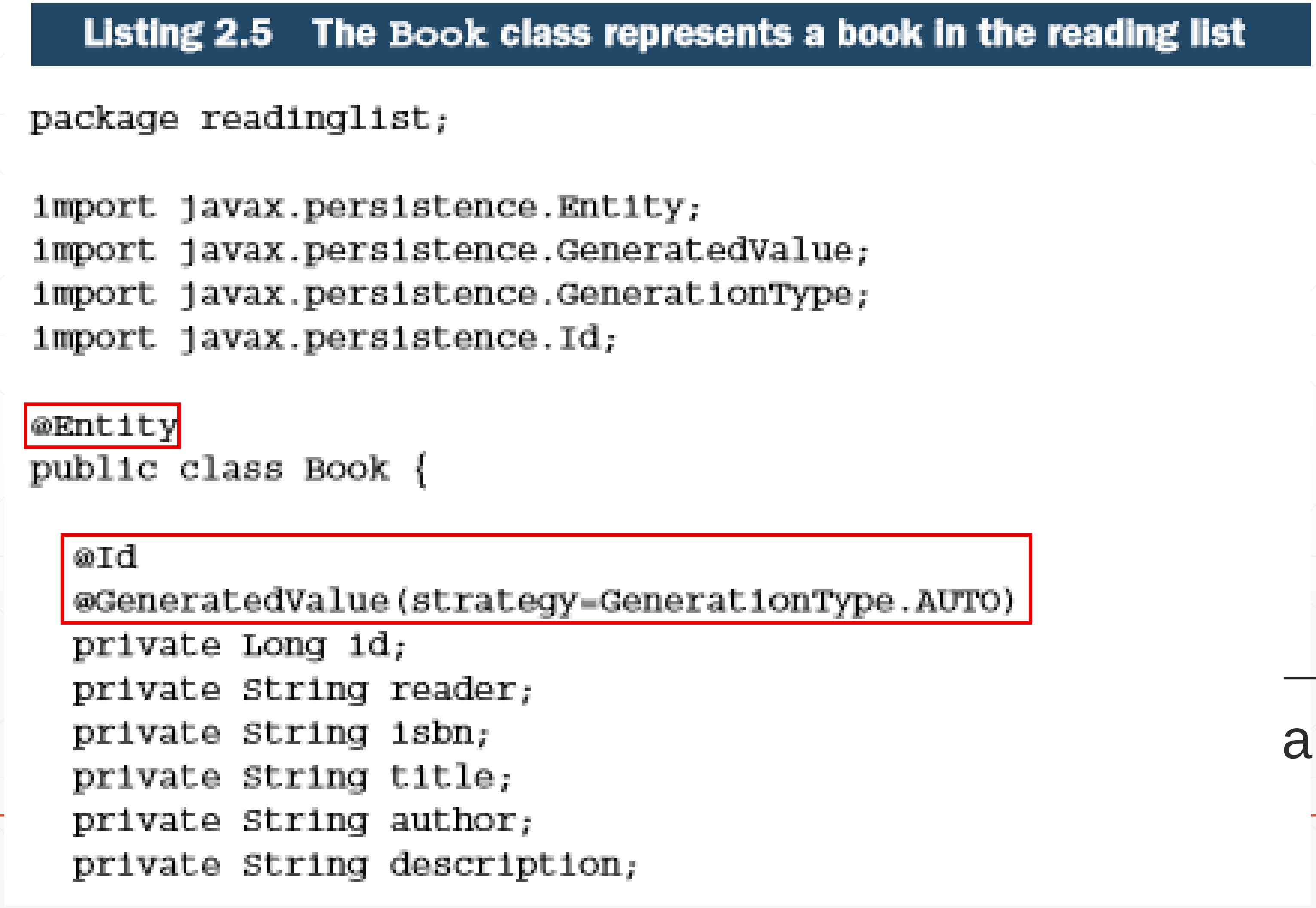


```
<properties>
  <project.build.sourceEncoding>
    UTF-8
  </project.build.sourceEncoding>
  <start-class>readinglist.Application</start-class>
  <java.version>1.7</java.version>
</properties>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

</project>
```

Apply Spring Boot plugin

First project

- Define an entity class that represents a book:

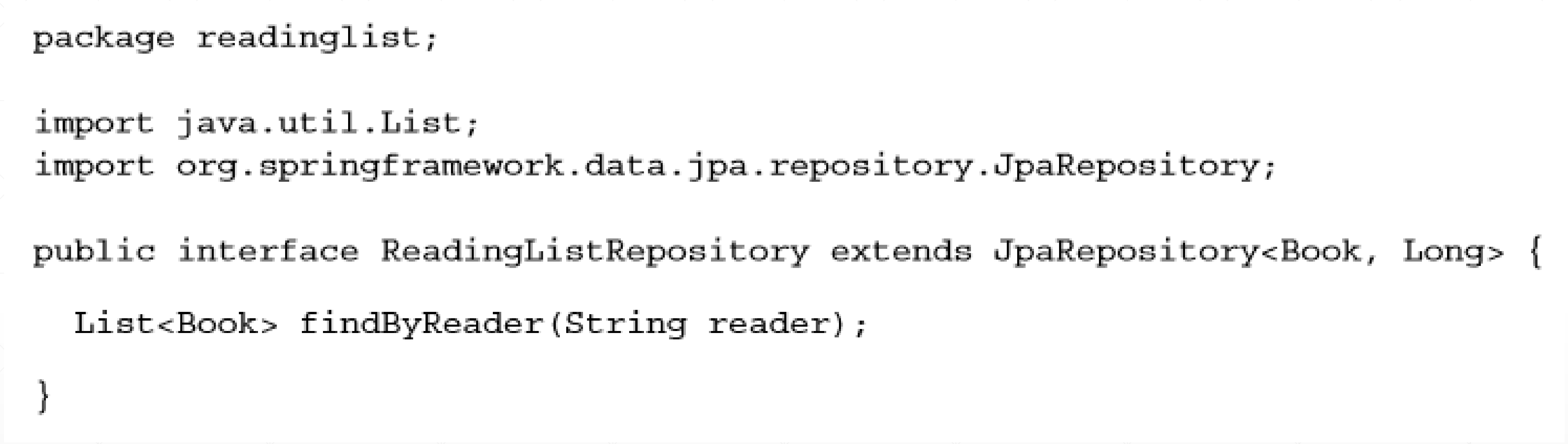


Listing 2.5 The Book class represents a book in the reading list

```
package readinglist;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Book {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;
  private String reader;
  private String isbn;
  private String title;
  private String author;
  private String description;
```

→ Also create Getters and Setters

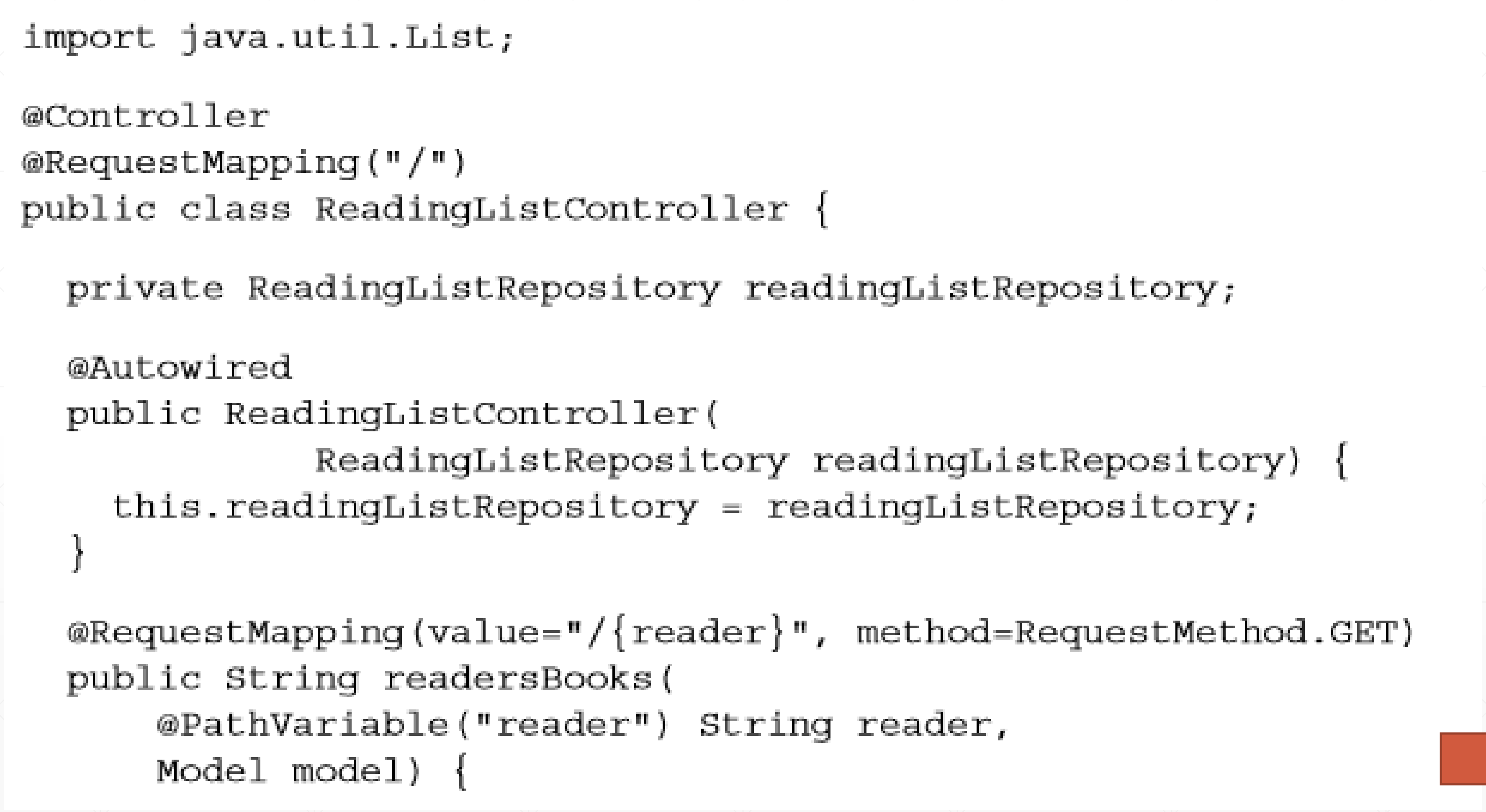
Defining the repository interface



```java
package readinglist;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ReadingListRepository extends JpaRepository<Book, Long> {

  List<Book> findByReader(String reader);

}
```

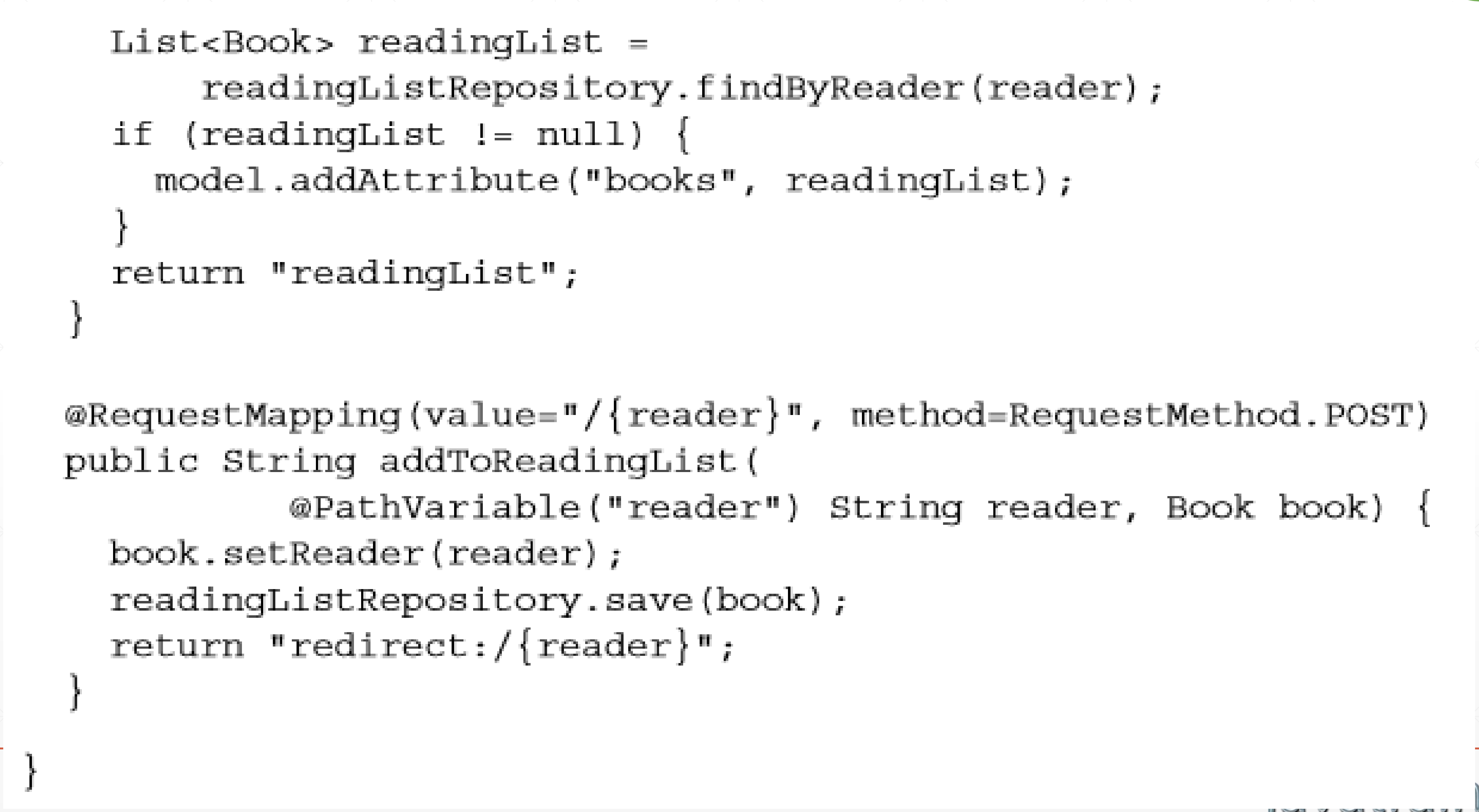
Spring Data provides a special magic of its own, making it possible to define a repository with just an interface. The interface will be implemented automatically at runtime when the application is started.

Creating the web interface



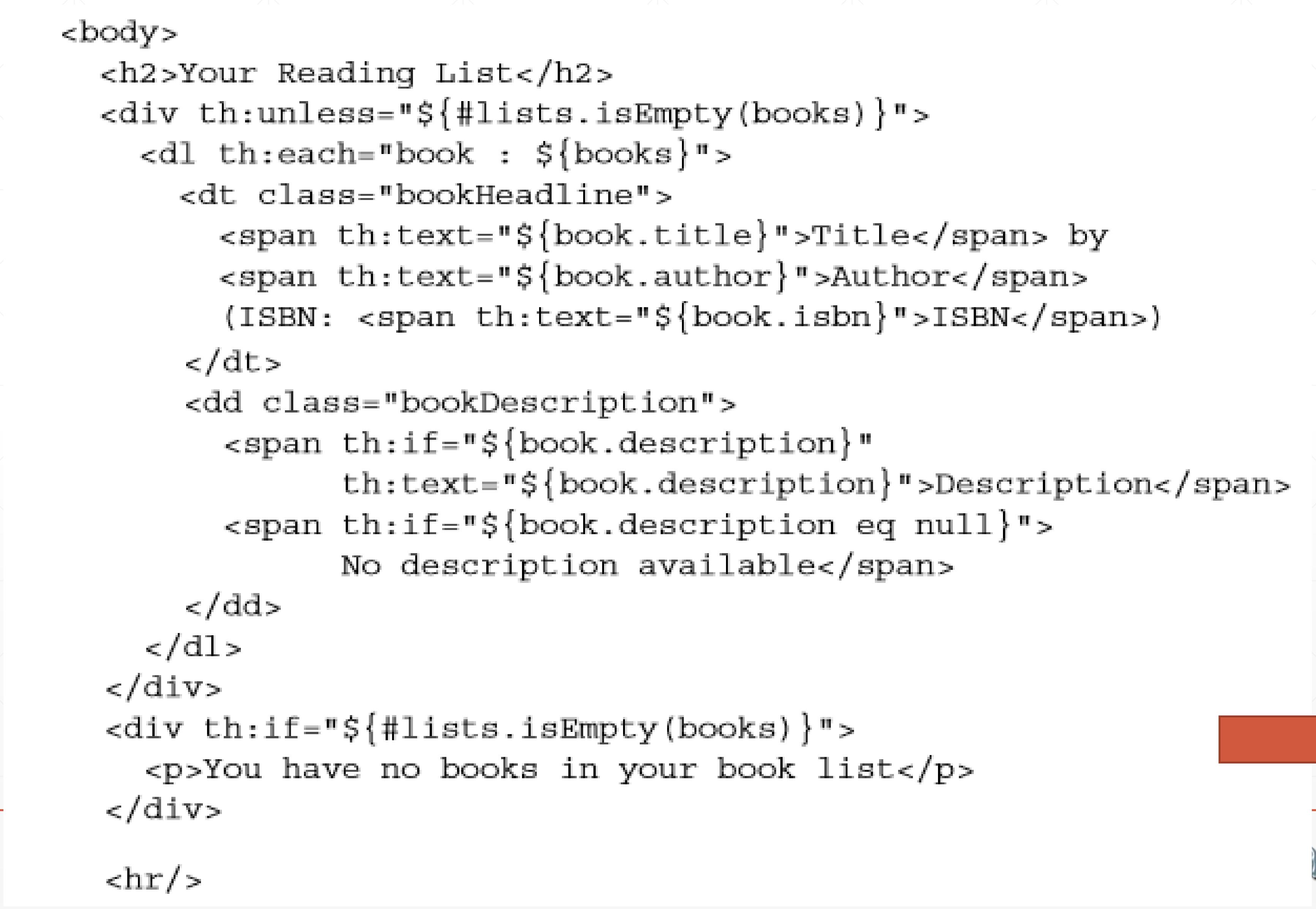
```
import java.util.List;

@Controller
@RequestMapping("/")
public class ReadingListController {

  private ReadingListRepository readingListRepository;

  @Autowired
  public ReadingListController(
             ReadingListRepository readingListRepository) {
    this.readingListRepository = readingListRepository;
  }

  @RequestMapping(value="/{reader}", method=RequestMethod.GET)
  public String readersBooks(
      @PathVariable("reader") String reader,
      Model model) {
```

Creating the web interface



```
    List<Book> readingList =
        readingListRepository.findByReader(reader);
    if (readingList != null) {
      model.addAttribute("books", readingList);
    }
    return "readingList";
  }

  @RequestMapping(value="/{reader}", method=RequestMethod.POST)
  public String addToReadingList(
            @PathVariable("reader") String reader, Book book) {
    book.setReader(reader);
    readingListRepository.save(book);
    return "redirect:/{reader}";
  }

}
```

Creating the web interface

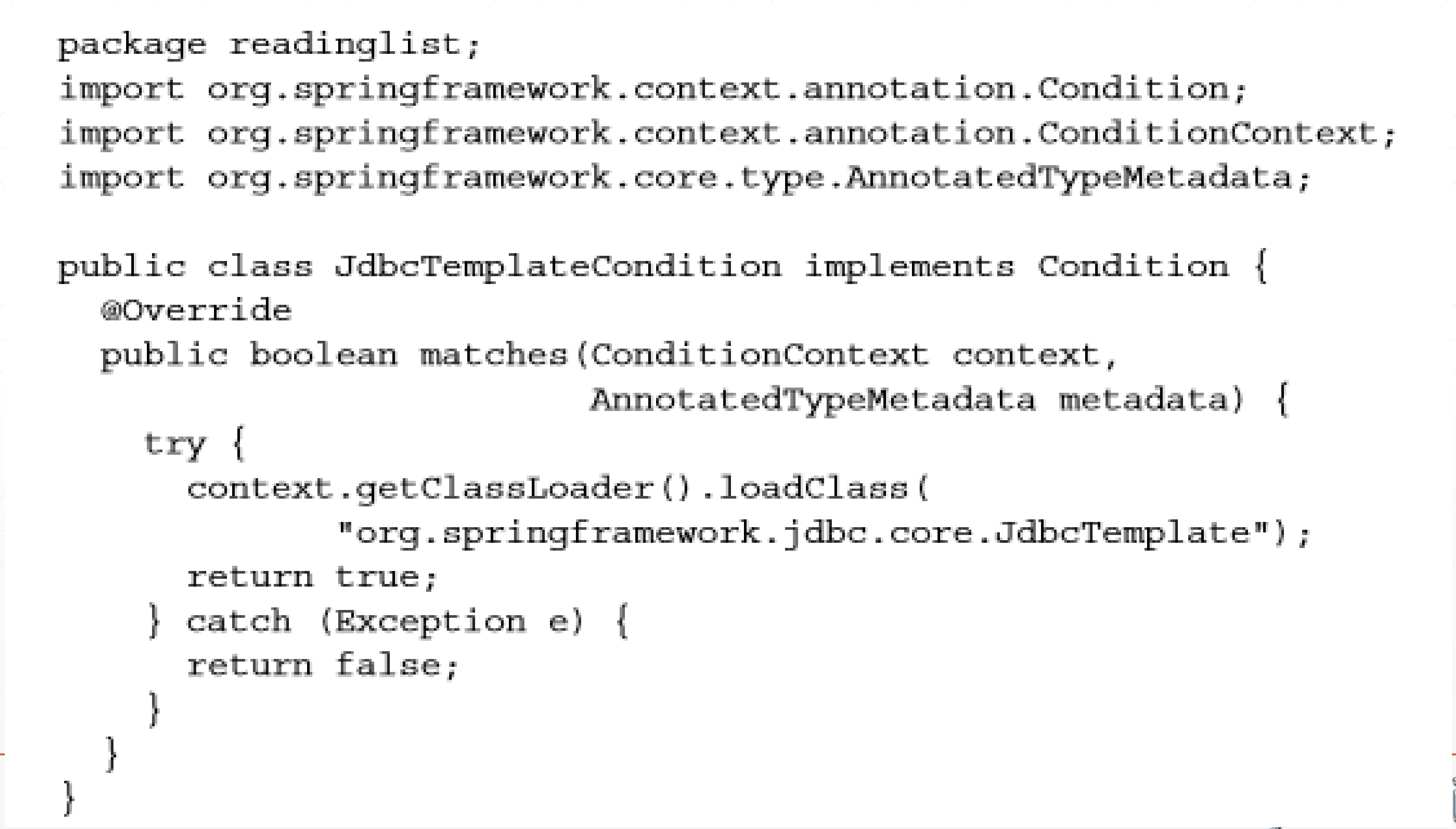


```
<body>
  <h2>Your Reading List</h2>
  <div th:unless="${#lists.isEmpty(books)}">
    <dl th:each="book : ${books}">
      <dt class="bookHeadline">
        <span th:text="${book.title}">Title</span> by
        <span th:text="${book.author}">Author</span>
        (ISBN: <span th:text="${book.isbn}">ISBN</span>)
      </dt>
      <dd class="bookDescription">
        <span th:if="${book.description}"
              th:text="${book.description}">Description</span>
        <span th:if="${book.description eq null}">
              No description available</span>
      </dd>
    </dl>
  </div>
  <div th:if="${#lists.isEmpty(books)}">
    <p>You have no books in your book list</p>
  </div>

  <hr/>
```

Creating the web interface

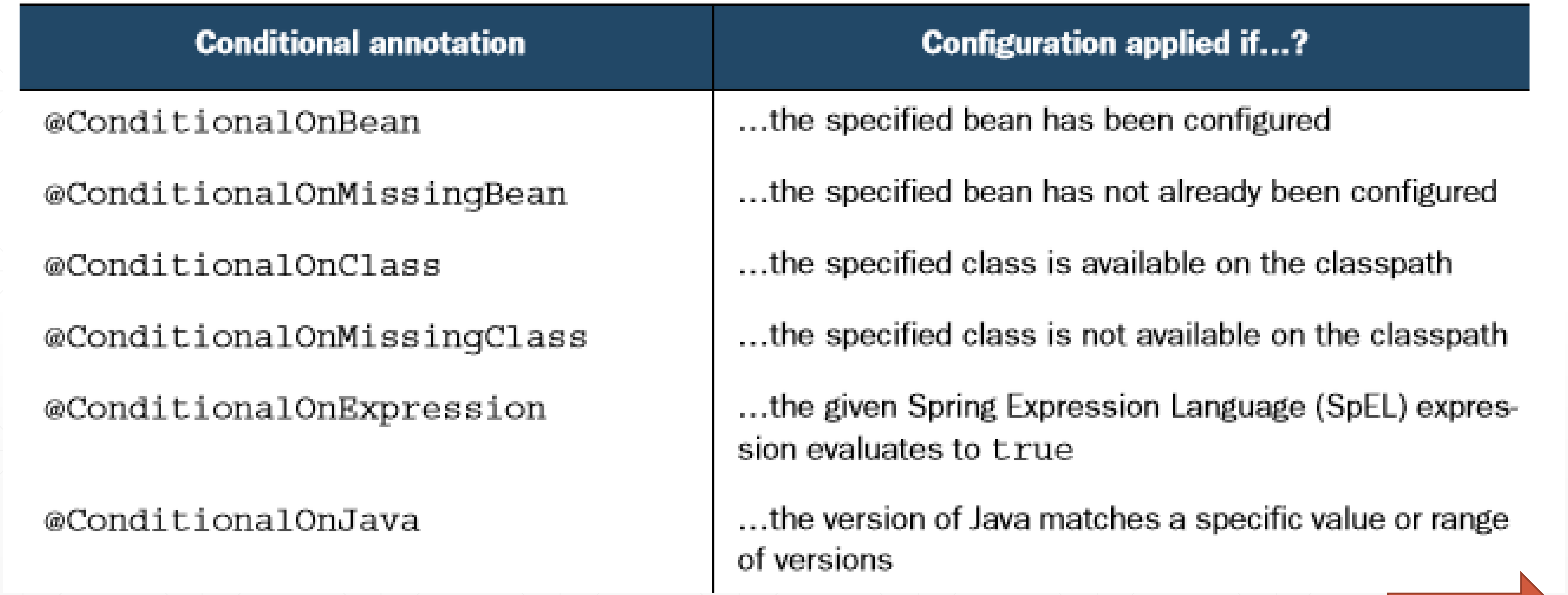
```
  <h3>Add a book</h3>
  <form method="POST">
    <label for="title">Title:</label>
      <input type="text" name="title" size="50"></input><br/>
    <label for="author">Author:</label>
      <input type="text" name="author" size="50"></input><br/>
    <label for="isbn">ISBN:</label>
      <input type="text" name="isbn" size="15"></input><br/>
    <label for="description">Description:</label><br/>
      <textarea name="description" cols="80" rows="5">
      </textarea><br/>
    <input type="submit"></input>
  </form>

</body>
```

Write your own conditions in Spring



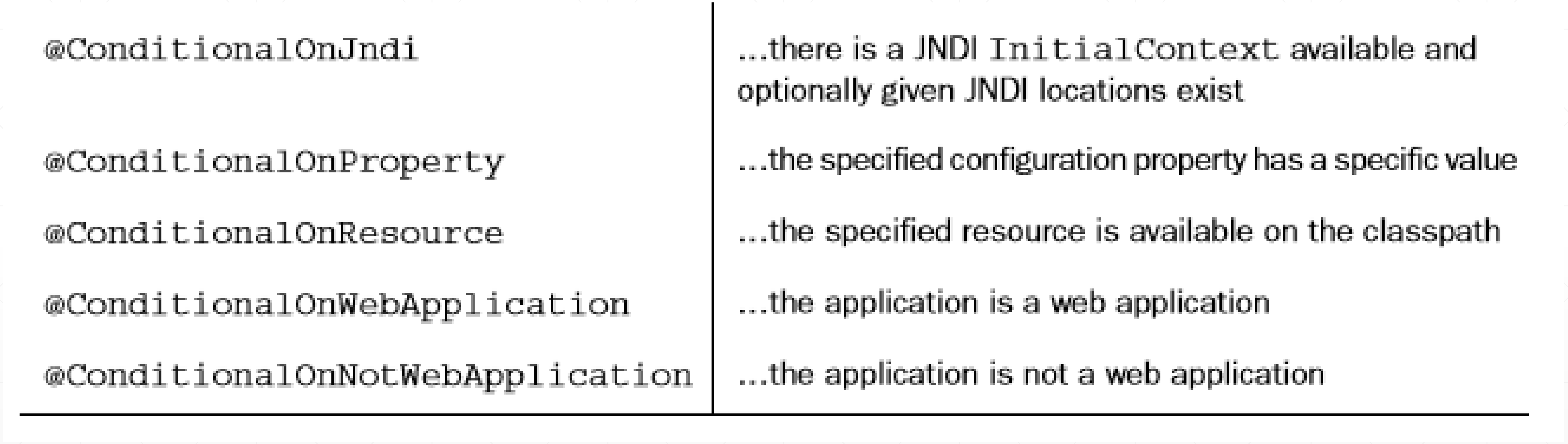
```
package readinglist;
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

public class JdbcTemplateCondition implements Condition {
  @Override
  public boolean matches(ConditionContext context,
                         AnnotatedTypeMetadata metadata) {
    try {
      context.getClassLoader().loadClass(
             "org.springframework.jdbc.core.JdbcTemplate");
      return true;
    } catch (Exception e) {
      return false;
    }
  }
}
```

Conditional annotations used in auto-configuration



Conditional annotation	Configuration applied if...?
`@ConditionalOnBean`	...the specified bean has been configured
`@ConditionalOnMissingBean`	...the specified bean has not already been configured
`@ConditionalOnClass`	...the specified class is available on the classpath
`@ConditionalOnMissingClass`	...the specified class is not available on the classpath
`@ConditionalOnExpression`	...the given Spring Expression Language (SpEL) expression evaluates to `true`
`@ConditionalOnJava`	...the version of Java matches a specific value or range of versions

Conditional annotations used in auto-configuration



`@ConditionalOnJndi`	...there is a JNDI `InitialContext` available and optionally given JNDI locations exist
`@ConditionalOnProperty`	...the specified configuration property has a specific value
`@ConditionalOnResource`	...the specified resource is available on the classpath
`@ConditionalOnWebApplication`	...the application is a web application
`@ConditionalOnNotWebApplication`	...the application is not a web application