

eso, y mostrarlo de nuevo no nos ayudaría a que la aplicación de la lista de lectura se escribiera más rápido.

En lugar de perder el tiempo hablando de la configuración de Spring, sabiendo que Spring Boot se encargará de eso por nosotros, veamos cómo aprovechar la configuración automática de Spring Boot nos mantiene enfocados en escribir el código de la aplicación. No se me ocurre mejor manera de hacerlo que empezar a escribir el código de la aplicación para la lista de lectura.

#### DEFINIENDO EL DOMINIO

El concepto de dominio central en nuestra aplicación es un libro que está en la lista de lectura de un lector. Por lo tanto, necesitaremos definir una clase de entidad que represente un libro. El Listado 2.5 muestra cómo Libro se define el tipo.

#### Listado 2.5 El **Libro** la clase representa un libro en la lista de lectura

lista de lectura de paquetes;

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

@Entidad

Libro de clase pública {

    @Identificación

    @GeneratedValue (estrategia = GenerationType.AUTO)

    identificación larga privada;

    lector de cadenas privado;

    cadena privada isbn;

    título de cadena privada;

    autor de cadena privado;

    descripción de cadena privada;

    public Long getId () {

        id de retorno;

    }

    public void setId (Long id) {

        this.id = id;

    }

    public String getReader () {

        lector de retorno;

    }

    public void setReader (lector de cadenas) {

        this.reader = reader;

    }

    public String getIsbn () {

        return isbn;

```

    }

    public void setIsbn (String isbn) {
        this.isbn = isbn;
    }

    public String getTitle () {
        titulo de devolución;
    }

    public void setTitle (titulo de cadena) {
        this.title = titulo;
    }

    public String getAuthor () {
        autor de retorno;
    }

    public void setAuthor (autor de la cadena) {
        this.author = autor;
    }

    public String getDescription () {
        descripción de devolución;
    }

    public void setDescription (Descripción de la cadena) {
        this.description = descripción;
    }
}

```

Como puede ver, el Libro class es un objeto Java simple con un puñado de propiedades que describen un libro y los métodos de acceso necesarios. Está anotado con `@ Entidad` designándolo como una entidad JPA. la identificación la propiedad está anotada con `@ Identificación` y `@ GeneratedValue` para indicar que este campo es la identidad de la entidad y que su valor se proporcionará automáticamente.

#### DEFINIENDO LA INTERFAZ DEL REPOSITORIO

A continuación, necesitamos definir el repositorio a través del cual Leyendo lista los objetos se conservarán en la base de datos. Debido a que estamos usando Spring Data JPA, esa tarea es una simple cuestión de crear una interfaz que amplíe Spring Data JPA JpaRepository interfaz:

lista de lectura de paquetes;

```

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

```

La interfaz pública ReadingListRepository extiende JpaRepository <Libro, Largo> {

```

    List <Libro> findByReader (lector de cadenas);
}

```

Extendiendo `JpaRepository`, `ReadingListRepository` hereda 18 métodos para realizar operaciones de persistencia comunes. Los `JpaRepository` La interfaz está parametrizada con dos parámetros: el tipo de dominio con el que trabajará el repositorio y el tipo de su propiedad ID. Además, agregué un `findByReader()` método a través del cual se puede buscar una lista de lectura dado el nombre de usuario de un lector.

Si se pregunta quién implementará `ReadingListRepository` y los 18 métodos que hereda, no se preocupe demasiado por eso. SpringData proporciona una magia especial propia, lo que hace posible definir un repositorio con solo una interfaz. La interfaz se implementará automáticamente en tiempo de ejecución cuando se inicie la aplicación.

#### CREANDO LA INTERFAZ WEB

Ahora que tenemos el dominio de la aplicación definido y un repositorio para los objetos persistentes desde ese dominio a la base de datos, todo lo que queda es crear el front-end web. Un controlador Spring MVC como el del listado 2.6 manejará las solicitudes HTTP para la aplicación.

Listado 2.6 Un controlador Spring MVC que encabeza la aplicación de lista de lectura

```
lista de lectura de paquetes;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Controller;

import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable; import
org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.RequestMethod;

import java.util.List;

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

    ReadingListRepository privado readingListRepository;

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

    @RequestMapping (value = "/" + {reader}", method = RequestMethod.GET) lectores de cadenas públicos Libros (

        @PathVariable ("lector") Lector de cadenas,
        Modelo modelo) {

        Lista <Libro> readingList =
            readingListRepository.findByReader (lector);
        if (readingList != null) {
            model.addAttribute ("libros", readingList);
        }
        return "readingList";
    }
}
```

```

@RequestMapping (valor = "/" + {lector}", método = RequestMethod.POST) cadena pública addToReadingList (

    @PathVariable ("lector") lector de cadenas, libro libro) {
        book.setReader (lector);
        readingListRepository.save (libro);
        return "redireccionar: / {lector}";
    }
}

```

ReadingListController está anotado con `@ Controlador` para ser recogido por el escaneo de componentes y registrado automáticamente como un bean en el contexto de la aplicación Spring. También está anotado con `@ RequestMapping` para mapear todos sus métodos de manejo a una ruta URL base de `"/"`.

El controlador tiene dos métodos:

- lectores Libros () —Administra HTTP OBTENER solicitudes de `/ {reader}` recuperando un Libro list del repositorio (que se inyectó en el constructor del controlador) para el lector especificado en la ruta. Pone la lista de Libro en el modelo bajo la clave "libros" y devuelve "readingList" como el nombre lógico de la vista para representar el modelo.
- addToReadingList () —Administra HTTP CORREO solicitudes para `/ {reader}`, vinculando los datos en el cuerpo de la solicitud a un Libro objeto. Este método establece el Libro objetos lector propiedad al nombre del lector, y luego guarda el modificado Libro a través del repositorio salvar() método. Finalmente, regresa especificando una redirección a `/ {reader}` (que será manejado por el otro método del controlador).

los lectores Libros () El método concluye devolviendo "readingList" como el nombre de la vista lógica. Por lo tanto, también debemos crear esa vista. Decidí al principio de este proyecto que usaríamos Thymeleaf para definir las vistas de la aplicación, por lo que el siguiente paso es crear un archivo llamado readingList.html en `src / main / resources / templates` con el siguiente contenido.

#### Listado 2.7 La plantilla Thymeleaf que presenta una lista de lectura

```

<html>
<cabeza>
<title> Lista de lectura </title>
<link rel = "stylesheet" th: href = "@ {/ style.css}"> </link>
</head>

<cuerpo>
<h2> Tu lista de lectura </h2>
<div th: a menos que = "${ # lists.isEmpty (libros)}">
<dl th: each = "libro: $ {libros}">
<dt class = "bookHeadline">
<span th: text = "${ book.title}"> Título </span> por
<span th: text = "${ book.author}"> Autor </span>
(ISBN: <span th: text = "${ book.isbn}"> ISBN </span>)

```

```

        </dt>
        <dd class = "bookDescription">
            <span th: if = "${book.description}"
                th: text = "${book.description}"> Descripción </span>
            <span th: if = "${book.description eq null}">
                No hay descripción disponible </span>
        </dd>
    </dl>
</div>
<div th: if = "${#lists.isEmpty(libros)}">
    <p> No tienes libros en tu lista de libros </p> </div>

<h />

<h3> Agregar un libro </h3>
<método de formulario = "POST">
    <label for = "title"> Título: </label>
    <input type = "text" name = "title" size = "50"> </input> <br/> <label for = "author"> Autor: </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"> Descripción:
</label> <br/>
    <textarea name = "description" cols = "80" rows = "5"> </textarea> <br/>

    <input type = "enviar"> </input>
</form>

</body>
</html>

```

Esta plantilla define una página HTML que se divide conceptualmente en dos partes. En la parte superior de la página hay una lista de libros que están en la lista de lectura del lector. En la parte inferior hay un formulario que el lector puede usar para agregar un nuevo libro a la lista de lectura.

Por motivos estéticos, la plantilla Thymeleaf hace referencia a una hoja de estilo denominada `style.css`. Ese archivo debe crearse en `src / main / resources / static` y verse así:

```

cuerpo {
    color de fondo: #cccccc;
    familia de fuentes: arial, helvetica, sans-serif;
}

.bookHeadline {
    tamaño de fuente: 12pt;
    font-weight: negrita;
}

.descripcion del libro {
    tamaño de fuente: 10pt;
}

etiqueta {
    font-weight: negrita;
}

```

Esta hoja de estilo es simple y no se excede para que la aplicación se vea bien. Pero sirve para nuestros propósitos y, como pronto verá, sirve para demostrar una parte de la configuración automática de Spring Boot.

Lo crea o no, es una aplicación completa. En este capítulo se le han presentado todas y cada una de las líneas. Tómese un momento, hojee las páginas anteriores y vea si puede encontrar alguna configuración. De hecho, aparte de las tres líneas de configuración en el listado 2.1 (que esencialmente activan la configuración automática), no tenía que escribir ninguna configuración de Spring.

A pesar de la falta de configuración de Spring, esta completa aplicación de Spring está lista para ejecutarse. Encendamos y veamos cómo se ve.

### 2.3.2 Ejecutando la aplicación

Hay varias formas de ejecutar una aplicación Spring Boot. Anteriormente, en la sección 2.5, discutimos cómo ejecutar la aplicación a través de Maven y Gradle, así como también cómo construir y ejecutar un JAR ejecutable. Más adelante, en el capítulo 8, también verá cómo crear un archivo WAR que se pueda implementar de manera tradicional en un servidor de aplicaciones web Java como Tomcat.

Si está desarrollando su aplicación con Spring Tool Suite, también tiene la opción de ejecutar la aplicación dentro de su IDE seleccionando el proyecto y eligiendo Ejecutar como> Aplicación Spring Boot en el menú Ejecutar, como se muestra en la figura 2.3.

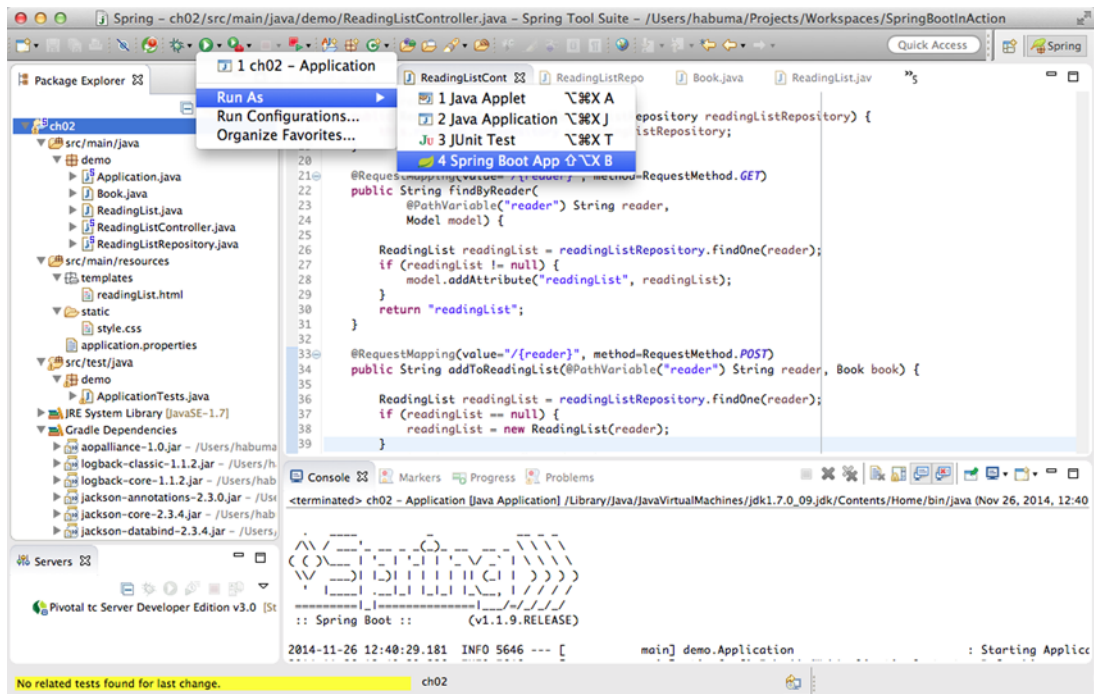


Figura 2.3 Ejecución de una aplicación Spring Boot desde Spring Tool Suite

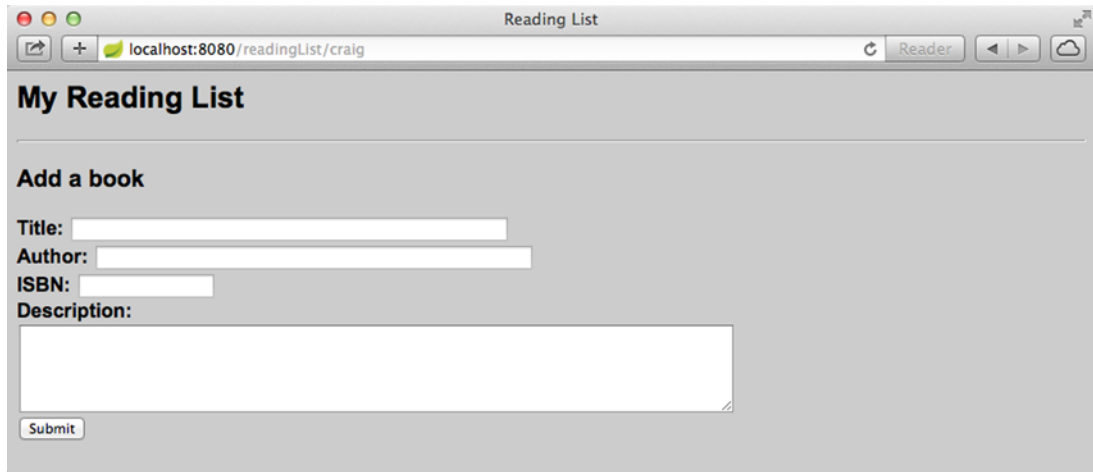


Figura 2.4 Una lista de lectura inicialmente vacía

Suponiendo que todo funciona, su navegador debería mostrarle una lista de lectura vacía junto con un formulario para agregar un nuevo libro a la lista. La figura 2.4 muestra cómo podría verse.

Ahora siga adelante y use el formulario para agregar algunos libros a su lista de lectura. Después de hacerlo, su lista podría verse como en la figura 2.5.

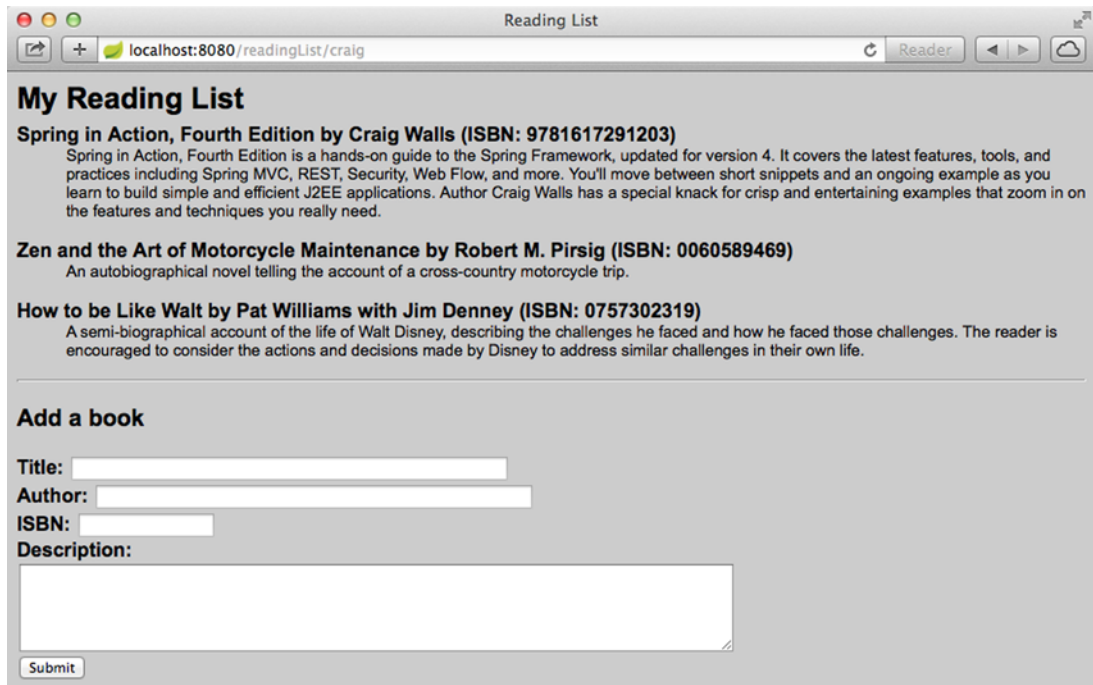


Figura 2.5 La lista de lectura después de que se hayan agregado algunos libros.

Siéntase libre de tomarse un momento para jugar con la aplicación. Cuando esté listo, continúe y veremos cómo Spring Boot hizo posible escribir una aplicación Spring completa sin código de configuración Spring.

### 2.3.3 ¿Lo que acaba de suceder?

Como dije, es difícil describir la configuración automática cuando no hay ninguna configuración a la que apuntar. Entonces, en lugar de dedicar tiempo a discutir lo que no tiene que hacer, esta sección se ha centrado en lo que debe hacer, es decir, escribir el código de la aplicación.

Pero ciertamente hay alguna configuración en alguna parte, ¿verdad? La configuración es un elemento central de Spring Framework, y debe haber algo que le diga a Spring cómo ejecutar su aplicación.

Cuando agrega Spring Boot a su aplicación, hay un archivo JAR llamado springboot-autoconfigure que contiene varias clases de configuración. Cada una de estas clases de configuración está disponible en el classpath de la aplicación y tiene la oportunidad de contribuir a la configuración de su aplicación. Hay configuración para Thymeleaf, configuración para Spring Data JPA, configuración para Spring MVC y configuración para docenas de otras cosas de las que podría o no querer aprovechar en su aplicación Spring.

Sin embargo, lo que hace que toda esta configuración sea especial es que aprovecha el soporte de Spring para la configuración condicional, que se introdujo en Spring 4.0. La configuración condicional permite que la configuración esté disponible en una aplicación, pero que se ignore a menos que se cumplan determinadas condiciones.

Es bastante fácil escribir sus propias condiciones en Spring. Todo lo que tienes que hacer es implementar el Condición interfaz y anular su `partidos()` método. Por ejemplo, la siguiente clase de condición simple solo pasará si `JdbcTemplate` está disponible en la ruta de clases:

```
lista de lectura de paquetes;
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext; import
org.springframework.core.type.AnnotatedTypeMetadata;

La clase pública JdbcTemplateCondition implementa Condición {
    @Anular
    coincidencias booleanas públicas (contexto ConditionContext,
                                     Metadatos AnnotatedTypeMetadata) {

        tratar {
            context.getClassLoader().loadClass (
                "org.springframework.jdbc.core.JdbcTemplate");
            devuelve verdadero;
        } captura (Excepción e) {
            falso retorno;
        }
    }
}
```



Puede utilizar esta clase de condición personalizada cuando declare beans en Java:

```
@Condicional (JdbcTemplateCondition.class)
public MyService myService () {
    ...
}
```

En este caso, el MyService bean solo se creará si el JdbcTemplateCondition pasa. Es decir que el MyService bean solo se creará si JdbcTemplate está disponible en classpath. De lo contrario, se ignorará la declaración del bean.

Aunque la condición que se muestra aquí es bastante simple, Spring Boot define varias condiciones más interesantes y las aplica a las clases de configuración que componen la configuración automática de Spring Boot. Spring Boot aplica la configuración condicional definiendo varias anotaciones condicionales especiales y usándolas en sus clases de configuración. La Tabla 2.1 enumera las anotaciones condicionales que proporciona Spring Boot.

Cuadro 2.1 Anotaciones condicionales utilizadas en la configuración automática

Anotación condicional	¿Configuración aplicada si...?
@ConditionalOnBean	... el bean especificado se ha configurado
@ConditionalOnMissingBean	... el bean especificado aún no se ha configurado ... la clase especificada
@ConditionalOnClass	está disponible en la ruta de clases ... la clase especificada no está
@ConditionalOnMissingClass	disponible en la ruta de clases
@ConditionalOnExpression	... la expresión de Spring Expression Language (SpEL) dada se evalúa como cierto
@ConditionalOnJava	... la versión de Java coincide con un valor específico o un rango de versiones
@ConditionalOnJndi	... hay un JNDI InitialContext Existen ubicaciones JNDI disponibles y proporcionadas opcionalmente
@ConditionalOnProperty	... la propiedad de configuración especificada tiene un valor específico ... el
@ConditionalOnResource	recurso especificado está disponible en la ruta de clase ... la aplicación es una
@ConditionalOnWebApplication	aplicación web
@ConditionalOnNotWebApplication	... la aplicación no es una aplicación web

En general, nunca debería necesitar mirar el código fuente de las clases de autoconfiguración de Spring Boot. Pero como una ilustración de cómo se usan las anotaciones en la tabla 2.1, considere este extracto de DataSourceAutoConfiguration ( proporcionado como parte de la biblioteca de configuración automática de Spring Boot):

```
@Configuración
@ConditionalOnClass ({DataSource.class, EmbeddedDatabaseType.class}) @EnableConfigurationProperties
(DataSourceProperties.class)
```

```

@Import ({Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class
    })
clase pública DataSourceAutoConfiguration {

    ...

}

```

Como se puede ver, `DataSourceAutoConfiguration` es un `@ Configuración`- clase anotada que (entre otras cosas) importa alguna configuración adicional de otras clases de configuración y define algunos beans propios. ¿Qué es lo más importante de notar aquí?

es eso `DataSourceAutoConfiguration` está anotado con `@ ConditionalOnClass` a requieren que ambos Fuente de datos y `EmbeddedDatabaseType` estar disponible en el classpath. Si no están disponibles, la condición falla y se proporciona cualquier configuración. por `DataSourceAutoConfiguration` será ignorado.

Dentro de `DataSourceAutoConfiguration` hay un anidado `JdbcTemplateConfiguration` clase que proporciona autoconfiguración de un `JdbcTemplate` frijol:

```

@Configuration
@Conditional (DataSourceAutoConfiguration.DataSourceAvailableCondition.class) clase estática protegida JdbcTemplateConfiguration {

    @Autowired (obligatorio = falso)
    fuente de datos privada DataSource;

    @Frijol
    @ConditionalOnMissingBean (JdbcOperations.class)
    public JdbcTemplate jdbcTemplate () {
        return new JdbcTemplate (this.dataSource);
    }

    ...

}

```

`JdbcTemplateConfiguration` es una anotación con el nivel bajo `@ Condicional` a requiere que el `DataSourceAvailableCondition` pasar, esencialmente requiriendo que un Fuente de datos bean esté disponible o se creará mediante la configuración automática. Suponiendo que un Fuente de datos bean estará disponible, el `@ Frijol`- anotado `jdbcTemplate ()` El método configura un `JdbcTemplate` frijol. Pero `jdbcTemplate ()` está anotado con `@ConditionalOnMissingBean` de modo que el bean se configurará solo si aún no hay un bean de tipo `JdbcOperations` ( la interfaz que `JdbcTemplate` implementos).

Hay mucho más para `DataSourceAutoConfiguration` ya las otras clases de autoconfiguración proporcionadas por Spring Boot que se muestran aquí. Pero esto debería darle una idea de cómo Spring Boot aprovecha la configuración condicional para implementar la configuración automática.

Como pertenece directamente a nuestro ejemplo, las siguientes decisiones de configuración las toman los condicionales en la configuración automática:

- Dado que H2 está en la ruta de clases, se creará un bean de base de datos H2 incorporado. Este frijol es de tipo `javax.sql.DataSource`, que la implementación de JPA (Hibernate) necesitará para acceder a la base de datos.
- Debido a que Hibernate Entity Manager está en el classpath (de manera transitiva a través de Spring Data JPA), la configuración automática configurará los beans necesarios para soportar el trabajo. con Hibernate, incluido Spring's `LocalContainerEntityManagerFactoryBean` y `JpaVendorAdapter`.
- Debido a que Spring Data JPA está en la ruta de clases, Spring Data JPA se configurará para crear automáticamente implementaciones de repositorio a partir de interfaces de repositorio. Debido a que Thymeleaf está en la ruta de clases, Thymeleaf se configurará como una opción de vista para Spring MVC, incluido un solucionador de plantillas, un motor de plantillas y un solucionador de vistas de Thymeleaf. El solucionador de plantillas está configurado para resolver plantillas desde `/ templates` en relación con la raíz de la ruta de clases.
- Debido a que Spring MVC está en la ruta de clases (gracias a la dependencia del iniciador web), Spring's `DispatcherServlet` se configurará y Spring MVC se habilitará.
- Debido a que esta es una aplicación web Spring MVC, se registrará un controlador de recursos para entregar contenido estático desde `/ static` en relación con la raíz de la ruta de clases. (El controlador de recursos también proporcionará contenido estático de `/ public`, `/ resources` y `/ META-INF / resources`).
- Debido a que Tomcat está en la ruta de clase (referido de manera transitiva por la dependencia del iniciador web), se iniciará un contenedor Tomcat incrustado para escuchar en el puerto 8080.

Sin embargo, la conclusión principal aquí es que la configuración automática de Spring Boot asume la carga de configurar Spring para que pueda concentrarse en escribir su aplicación.

## 2.4 *Resumen*

Aprovechando las dependencias de arranque de Spring Boot y la configuración automática, puede desarrollar aplicaciones Spring de manera más rápida y sencilla. Las dependencias iniciales lo ayudan a concentrarse en el tipo de funcionalidad que necesita su aplicación en lugar de en las bibliotecas y versiones específicas que brindan esa funcionalidad. Mientras tanto, la configuración automática lo libera de la configuración estándar que es común entre las aplicaciones Spring sin Spring Boot.

Aunque la configuración automática es una forma conveniente de trabajar con Spring, también representa un enfoque obstinado del desarrollo de Spring. ¿Qué sucede si desea o necesita configurar Spring de manera diferente? En el próximo capítulo, veremos cómo puede anular la configuración automática de Spring Boot según sea necesario para lograr los objetivos de su aplicación. También verá cómo aplicar algunas de las mismas técnicas para configurar sus propios componentes de aplicación.

# 3

## *Personalización configuración*

---

### ***Este capítulo cubre***

- Anulación de beans configurados automáticamente
- Configuración con propiedades externas
- Personalización de páginas de error

La libertad de elección es algo asombroso. Si alguna vez ordenó una pizza (¿quién no?), Entonces sabrá que tiene control total sobre los ingredientes que se colocan en el pastel. Si pide salchichas, pepperoni, pimientos verdes y queso adicional, básicamente está configurando la pizza según sus especificaciones precisas.

Por otro lado, la mayoría de las pizzerías también ofrecen una forma de configuración automática. Puede pedir la pizza para los amantes de la carne, la pizza vegetariana, la pizza italiana picante o el ejemplo definitivo de la configuración automática de la pizza, la pizza suprema. Al pedir una de estas pizzas, no es necesario que especifique explícitamente los ingredientes. El tipo de pizza ordenada implica qué ingredientes se utilizan.

Pero, ¿qué pasa si te gustan todos los ingredientes de la pizza suprema, pero también quieres jalapeños y prefieres no comer champiñones? ¿Su gusto por la comida picante y la aversión a los hongos significa que la configuración automática no es aplicable y que debe

configurar explícitamente su pizza? Absolutamente no. La mayoría de las pizzerías le permitirán personalizar su pizza, incluso si comenzó con una opción preconfigurada del menú.

Trabajar con la configuración tradicional de Spring es muy parecido a pedir una pizza y especificar explícitamente todos los ingredientes. Tiene control total sobre lo que entra en su configuración de Spring, pero declarar explícitamente todos los beans en la aplicación no es óptimo. Por otro lado, la configuración automática de Spring Boot es como pedir una pizza especial del menú. Es más fácil dejar que Spring Boot maneje los detalles que declarar todos y cada uno de los beans en el contexto de la aplicación.

Afortunadamente, la configuración automática de Spring Boot es flexible. Al igual que la pizzería que dejará los hongos y agregará jalapeños a su pizza, Spring Boot le permitirá intervenir e influir en cómo aplica la configuración automática.

En este capítulo, veremos dos formas de influir en la configuración automática: anulaciones de configuración explícitas y configuración detallada con propiedades. También veremos cómo Spring Boot le ha proporcionado ganchos para que conecte una página de error personalizada.

### 3.1 *Anulación de la configuración automática de Spring Boot*

En términos generales, si puede obtener los mismos resultados sin configuración que con una configuración explícita, ninguna configuración es una opción obvia. ¿Por qué haría un trabajo adicional, escribiendo y manteniendo un código de configuración adicional, si puede obtener lo que necesita sin él?

La mayoría de las veces, los beans configurados automáticamente son exactamente lo que desea y no es necesario anularlos. Pero hay algunos casos en los que la mejor suposición que Spring Boot puede hacer durante la configuración automática probablemente no sea lo suficientemente buena.

Un excelente ejemplo de un caso en el que la configuración automática no es lo suficientemente buena es cuando aplica seguridad a su aplicación. La seguridad no es única para todos, y hay decisiones en torno a la seguridad de las aplicaciones que Spring Boot no tiene por qué hacer por usted. Aunque Spring Boot proporciona una configuración automática básica para la seguridad, seguramente querrá anularla para cumplir con sus requisitos de seguridad específicos.

Para ver cómo anular la configuración automática con una configuración explícita, comenzaremos agregando Spring Security al ejemplo de la lista de lectura. Después de ver lo que obtiene de forma gratuita con la configuración automática, anularemos la configuración de seguridad básica para adaptarse a una situación particular.

#### 3.1.1 *Asegurar la aplicación*

La configuración automática de Spring Boot hace que asegurar una aplicación sea pan comido. Todo lo que necesita hacer es agregar el iniciador de seguridad a la compilación. Para Gradle, la siguiente dependencia servirá:

```
compile ("org.springframework.boot: spring-boot-starter-security")
```

O, si está usando Maven, agregue esto < dependencia> a tu construcción < dependencias> cuadra:

```
<dependencia>
  <groupId> org.springframework.boot </groupId>
  <artifactId> spring-boot-starter-security </artifactId>
</dependencia>
```

¡Eso es! Reconstruya su aplicación y ejecútela. ¡Ahora es una aplicación web segura! El iniciador de seguridad agrega Spring Security (entre otras cosas) a la ruta de clases de la aplicación. Con Spring Security en la ruta de clases, se activa la configuración automática y se crea una configuración de Spring Security muy básica.

Si intenta abrir la aplicación en su navegador, se encontrará inmediatamente con un cuadro de diálogo de autenticación básica HTTP. El nombre de usuario que deberá ingresar es "usuario". En cuanto a la contraseña, es un poco más complicado. La contraseña se genera aleatoriamente y se escribe en los registros cada vez que se ejecuta la aplicación. Deberá revisar los mensajes de registro (escritos en stdout por defecto) y busque una línea que se parezca a esto:

Usando la contraseña de seguridad predeterminada: d9d8abe5-42b5-4f20-a32a-76ee3df658d9

No puedo decirlo con certeza, pero supongo que esta configuración de seguridad en particular probablemente no sea ideal para usted. Primero, los cuadros de diálogo HTTP Basic son torpes y no muy fáciles de usar. Y apuesto a que no desarrolla demasiadas aplicaciones que tienen un solo usuario al que no le importa buscar su contraseña en un archivo de registro. Por lo tanto, probablemente desee realizar algunos cambios en la configuración de Spring Security. Al menos, querrá proporcionar una página de inicio de sesión atractiva y especificar un servicio de autenticación que opere contra una base de datos o un almacén de usuarios basado en LDAP.

Veamos cómo hacerlo escribiendo alguna configuración de Spring Security explícita para anular el esquema de seguridad configurado automáticamente.

### 3.1.2 *Crear una configuración de seguridad personalizada*

Anular la configuración automática es una simple cuestión de escribir explícitamente la configuración como si la configuración automática no existiera. Esta configuración explícita puede tomar cualquier forma que admita Spring, incluida la configuración XML y la configuración basada en Groovy.

Para nuestros propósitos, nos centraremos en la configuración de Java al escribir una configuración explícita. En el caso de Spring Security, esto significa escribir una clase de configuración que se extiende `WebSecurityConfigurerAdapter`. `SecurityConfig` en el listado 3.1 es el clase de configuración que usaremos.

Listado 3.1 Configuración explícita para anular la seguridad configurada automáticamente

lista de lectura de paquetes;

```
import org.springframework.beans.factory.annotation.Autowired; import
org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication
builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders
HttpSecurity;
```

```

importar    org.springframework.security.config.annotation.web.configuration.
                                                    EnableWebSecurity;

importar    org.springframework.security.config.annotation.web.configuration.
                                                    WebSecurityConfigurerAdapter;

importar    org.springframework.security.core.userdetails.UserDetails; org.springframework.security.core.userdetails.UserDetailsService;
importar    org.springframework.security.core.userdetails.
importar
                                                    UsernameNotFoundException;

@Configuración
@EnableWebSecurity
SecurityConfig de clase pública extiende WebSecurityConfigurerAdapter {

    @Autowired
    ReaderRepository privado readerRepository;

    @Anular
    La configuración vacía protegida (HttpSecurity http) arroja una excepción {
        http
            . authorizeRequests ()
                . antMatchers ("/"). access ("hasRole ('READER')")
                . antMatchers ("/ ***"). permitAll ()

            . y()

            . formLogin ()
                . loginPage ("/ login")
                . failureUrl ("/ login? error = true");
    }

    @Anular
    configurar vacío protegido
        AuthenticationManagerBuilder auth) arroja Exception {
        auth
            . userDetailsService (new UserDetailsService () {
                @Anular
                public UserDetails loadUserByUsername (cadena de nombre de usuario)
                    lanza UsernameNotFoundException {
                        return readerRepository.findOne (nombre de usuario);
                    }
            });
    }
}

```

Requerir LECTOR acceso

Establecer inicio de sesión camino de la forma

Definir personalizado UserDetailsService

SecurityConfig es una configuración de Spring Security muy básica. Aun así, hace mucho de lo que necesitamos para personalizar la seguridad de la aplicación de lista de lectura. Al proporcionar esta clase de configuración de seguridad personalizada, le pedimos a Spring Boot que omita la configuración automática de seguridad y utilice nuestra configuración de seguridad en su lugar.

Clases de configuración que se extienden WebSecurityConfigurerAdapter puede anular dos diferentes configurar () métodos. En SecurityConfig, el primero configurar () El método especifica que las solicitudes de "/" (que ReadingListController Los métodos están asignados a) requieren un usuario autenticado con el rol de LECTOR. Todas las demás solicitudes

Las rutas están configuradas para acceso abierto a todos los usuarios. También designa / login como la ruta para la página de inicio de sesión, así como la página de error de inicio de sesión (junto con un error atributo).

Spring Security ofrece varias opciones de autenticación, incluida la autenticación contra almacenes de usuarios respaldados por JDBC, almacenes de usuarios respaldados por LDAP y almacenes de usuarios en memoria. Para nuestra aplicación, autenticaremos a los usuarios en la base de datos a través de JPA. El segundo configurar () El método configura esto configurando un servicio de detalles de usuario personalizado. Este servicio puede ser cualquier clase que implemente UserDetailsServiceImpl y se utiliza para buscar los detalles del usuario dado un nombre de usuario. La siguiente lista le ha dado una implementación anónima de clase interna que simplemente llama al Encuentra uno() método en un inyectado ReaderRepository ( que es una interfaz de repositorio Spring Data JPA).

### Listado 3.2 Una interfaz de repositorio para lectores persistentes

```
lista de lectura de paquetes;
import org.springframework.data.jpa.repository.JpaRepository;
```

```
ReaderRepository de interfaz pública
    extiende JpaRepository <Reader, String> {
}
}
```

← Lectores persistentes  
a través de JPA

Al igual que con BookRepository, no es necesario escribir una implementación de ReaderRepository. Porque se extiende JpaRepository, Spring Data JPA creará automáticamente una implementación en tiempo de ejecución. Esto le brinda 18 métodos para trabajar con Lector entidades.

Hablando de Lector entidades, el Lector clase (que se muestra en el listado 3.3) es la pieza final del rompecabezas. Es un tipo de entidad JPA simple con algunos campos para capturar el nombre de usuario, la contraseña y el nombre completo del usuario.

### Listado 3.3 Una entidad JPA que define un Lector

```
lista de lectura de paquetes;
import java.util.Arrays;
import java.util.Collection;
import javax.persistence.Entity;
import javax.persistence.Id;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority; import org.springframework.security.core.userdetails.UserDetails;
```

```
@Entidad
Lector de clase pública implementa UserDetails {
```

```
    private static final long serialVersionUID = 1L;
```

```
@Identificación
    nombre de usuario de cadena privada;
    Private String nombre completo;
    contraseña de cadena privada;
```

Campos de lector



```

public String getUsername () {
    devolver el nombre de usuario;
}

public void setUsername (String username) {
    this.username = nombre de usuario;
}

public String getFullname () {
    return fullname;
}

public void setFullname (String fullname) {
    this.fullname = fullname;
}

public String getPassword () {
    devolver contraseña;
}

public void setPassword (contraseña de cadena) {
    this.password = contraseña;
}

// Métodos UserDetails

@Anular
Colección pública <? extiende GrantedAuthority> getAuthorities () {
    return Arrays.asList (new SimpleGrantedAuthority ("LECTOR"));
}

@Anular
public boolean isAccountNonExpired () {
    devuelve verdadero;
}

@Anular
public boolean isAccountNonLocked () {
    devuelve verdadero;
}

@Anular
public boolean isCredentialsNonExpired () {
    devuelve verdadero;
}

@Anular
public boolean isEnabled () {
    devuelve verdadero;
}
}

```

Conceder  
 LECTOR  
 privilegio

No caduque,  
 bloquear o deshabilitar

Como se puede ver, Lector está anotado con @Entidad para convertirlo en una entidad JPA. Además, su nombre de usuario el campo está anotado con @Identificación para designarlo como el ID de la entidad. Esto parecía una elección natural, ya que el nombre de usuario debe identificar de forma única el Lector.

También notarás que Lector implementa el Detalles de usuario interfaz y varios de sus métodos. Esto hace posible utilizar un Lector objeto para representar a un usuario en Spring Security. El método `getAuthorities()` se anula para otorgar siempre a los usuarios `READER` autoridad. Los `isAccountNonExpired()`, `isAccountNonLocked()`, `isCredentials-` `No caducado()`, y `está habilitado()` todos los métodos están implementados para volver cierto para que la cuenta del lector nunca caduque, bloquee o revoque.

Reconstruya y reinicie la aplicación y debería poder iniciar sesión en la aplicación como uno de los lectores.

**MANTENERLO SIMPLE** En una aplicación más grande, las autorizaciones otorgadas a un usuario podrían ser entidades y mantenerse en una tabla de base de datos separada. Asimismo, el booleano Los valores que indican si una cuenta no está vencida, no está bloqueada y está habilitada pueden ser campos extraídos de la base de datos. Para nuestros propósitos, sin embargo, he decidido mantener estos detalles simples para no distraernos de lo que realmente estamos discutiendo... es decir, anular la configuración automática de Spring Boot.

Podemos hacer mucho más con respecto a la configuración de seguridad,<sup>1</sup> pero esto es todo lo que necesitamos aquí, y demuestra cómo anular la configuración automática de seguridad proporcionada por Spring Boot.

Nuevamente, todo lo que necesita hacer para anular la configuración automática de Spring Boot es escribir la configuración explícita. Spring Boot verá su configuración, dará un paso atrás y permitirá que su configuración tenga prioridad. Para entender cómo funciona esto, echemos un vistazo bajo las cubiertas de la configuración automática de Spring Boot para ver cómo funciona y cómo se permite anularlo.

### 3.1.3 Echando otro vistazo bajo las sábanas de la configuración automática

Como discutimos en la sección 2.3.3, la configuración automática de Spring Boot viene con varias clases de configuración, cualquiera de las cuales se puede aplicar en su aplicación. Toda esta configuración utiliza el soporte de configuración condicional de Spring 4.0 para tomar decisiones en tiempo de ejecución sobre si se debe usar o ignorar la configuración de Spring Boot.

En su mayor parte, el `@ConditionalOnMissingBean` La anotación descrita en la tabla 2.1 es lo que hace posible anular la configuración automática. El `JdbcTemplate` bean definido en Spring Boot's `DataSourceAutoConfiguration` es un examen muy simple de cómo `@ConditionalOnMissingBean` trabaja:

```
@Frijol
@ConditionalOnMissingBean(JdbcOperations.class)
public JdbcTemplate jdbcTemplate() {
    return new JdbcTemplate(this.dataSource);
}
```

---

<sup>1</sup>Para una inmersión más profunda en Spring Security, eche un vistazo a los capítulos 9 y 14 de mi *Primavera en acción, cuarta edición* (Manning, 2014).

los `JdbcTemplate()` el método está anotado con `@Frijol` y está listo para configurar un `JdbcTemplate` frijol si es necesario. Pero también está anotado con `@ConditionalOnMissingBean`, que requiere que no exista ya un bean de tipo `JdbcOperations` (la interfaz que `JdbcTemplate` implementa). Si ya hay un `JdbcOperations` bean, entonces la condición fallará y el `JdbcTemplate()` no se utilizará el método bean.

¿Qué circunstancias darían lugar a que ya hubiera un `JdbcOperation` ¿frijol? Spring Boot está diseñado para cargar la configuración a nivel de aplicación antes de considerar sus clases de configuración automática. Por lo tanto, si ya ha configurado un `JdbcTemplate` bean, entonces habrá un bean de tipo `JdbcOperaciones` en el momento en que tiene lugar la autoconfiguración, y la autoconfiguración `JdbcTemplate` bean será ignorado.

En lo que respecta a Spring Security, hay varias clases de configuración consideradas durante la configuración automática. No sería práctico repasar cada uno de ellos en detalle aquí, pero el más significativo al permitirnos anular la configuración de seguridad autoconfigurada de Spring Boot es `SpringBootWebSecurityConfiguration`. Aquí hay un extracto de esa clase de configuración:

```
@Configuración
@EnableConfigurationProperties
@ConditionalOnClass ({EnableWebSecurity.class})
@ConditionalOnMissingBean (WebSecurityConfiguration.class) @ConditionalOnWebApplication

clase pública SpringBootWebSecurityConfiguration {

...

}
```

Como se puede ver, `SpringBootWebSecurityConfiguration` está anotado con algunos anotaciones condicionales. Por la `@ConditionalOnClass` anotación, la `@EnableWebSecurity` la anotación debe estar disponible en la ruta de clases. Y por `@ConditionalOnWebApplication`, la aplicación debe ser una aplicación web. Pero es el `@ConditionalOnMissingBean` anotación que hace posible que se utilice nuestra clase de configuración de seguridad en lugar de `SpringBootWebSecurityConfiguration`.

Los `@ConditionalOnMissingBean` requiere que no haya un bean de tipo `WebSecurityConfiguration`. Aunque puede que no sea evidente en la superficie, al anotar nuestro `SecurityConfig` clase con `@EnableWebSecurity`, indirectamente estamos creando un bean de tipo `WebSecurityConfiguration`. Por lo tanto, para cuando se lleve a cabo la autoconfiguración, ya habrá un bean de tipo `WebSecurityConfiguration`, la

`@ConditionalOnMissingBean` la condición fallará, y cualquier configuración ofrecida por `SpringBootWebSecurityConfiguration` se omitirá.

Aunque la configuración automática de Spring Boot y `@ConditionalOnMissingBean` le permite anular explícitamente cualquiera de los beans que de otro modo se autoconfigurarían, no siempre es necesario llegar a ese extremo. Veamos cómo puede establecer algunas propiedades de configuración simples para ajustar los componentes configurados automáticamente.

## 3.2 Externalizar la configuración con propiedades

Cuando se trata de la seguridad de las aplicaciones, es casi seguro que querrá hacerse cargo de la configuración. Pero sería una pena renunciar a la configuración automática solo para modificar un pequeño detalle, como el número de puerto del servidor o el nivel de registro. Si necesita establecer una URL de base de datos, ¿no sería más fácil establecer una propiedad en algún lugar que declarar completamente un bean de fuente de datos?

Resulta que los beans que Spring Boot configura automáticamente ofrecen más de 300 propiedades para realizar ajustes. Cuando necesite ajustar la configuración, puede especificar estas propiedades a través de variables de entorno, propiedades del sistema Java, JNDI, argumentos de línea de comandos o archivos de propiedades.

Para comenzar con estas propiedades, veamos un ejemplo muy simple. Es posible que haya notado que Spring Boot emite un banner ascii-art cuando ejecuta la aplicación de lista de lectura desde la línea de comandos. Si desea deshabilitar el banner, puede hacerlo configurando una propiedad llamada `spring.main.show-banner` a falso. Una forma de hacerlo es especificar la propiedad como un parámetro de línea de comandos cuando ejecuta la aplicación:

```
$ java -jar lista-de-lectura-0.0.1-SNAPSHOT.jar --spring.main.show-banner=false
```

Otra forma es crear un archivo llamado `application.properties` que incluya la siguiente línea:

```
spring.main.show-banner=false
```

O, si lo prefiere, cree un archivo YAML llamado `application.yml` que tenga este aspecto:

```
primavera:
  principal:
    mostrar-banner: falso
```

También puede establecer la propiedad como una variable de entorno. Por ejemplo, si está utilizando el shell `bash` o `zsh`, puede configurarlo con el exportar mando:

```
$ exportación spring_main_show_banner=false
```

Tenga en cuenta el uso de guiones bajos en lugar de puntos y guiones, según sea necesario para los nombres de las variables de entorno.

De hecho, existen varias formas de establecer propiedades para una aplicación Spring Boot. Spring Boot extraerá propiedades de varias fuentes de propiedades, incluidas las siguientes:

- 1 Argumentos de la línea de comandos
- 2 Atributos JNDI de java: propiedades del sistema
- 3 JVM `comp / env`
- 4 Variables de entorno del sistema operativo
- 5 Valores generados aleatoriamente para propiedades con el prefijo aleatorio.\* (referenciado al establecer otras propiedades, como ``$` { random.long}`)
- 6 Un archivo `application.properties` o `application.yml` fuera de la aplicación

- 7 Un archivo `application.properties` o `application.yml` empaquetado dentro de la aplicación
- 8 Fuentes de propiedad especificadas por `@PropertySource`
- 9 Propiedades predeterminadas

Esta lista está en orden de precedencia. Es decir, cualquier propiedad establecida de una fuente superior en la lista anulará la misma propiedad establecida en una fuente inferior en la lista. Los argumentos de la línea de comandos, por ejemplo, anulan las propiedades de cualquier otra fuente de propiedad.

En cuanto a los archivos `application.properties` y `application.yml`, pueden residir en cualquiera de las cuatro ubicaciones:

- 1 Externamente, en un subdirectorio / `config` del directorio desde el que se ejecuta la aplicación
- 2 Externamente, en el directorio desde el que se ejecuta la aplicación Internamente, en
- 3 un paquete llamado "config"
- 4 Internamente, en la raíz del classpath

Nuevamente, esta lista está en orden de precedencia. Es decir, un archivo `application.properties` en un subdirectorio / `config` anulará las mismas propiedades establecidas en un archivo `application.properties` en la ruta de clases de la aplicación.

Además, descubrí que si tiene `application.properties` y `application.yml` en paralelo en el mismo nivel de precedencia, las propiedades en `application.yml` anularán las de `application.properties`.

Deshabilitar un banner `ascii-art` es solo un pequeño ejemplo de cómo usar las propiedades. Veamos algunas formas más comunes de modificar los beans configurados automáticamente.

### 3.2.1 *Autoconfiguración de ajuste fino*

Como dije, hay más de 300 propiedades que puede configurar para modificar y ajustar los beans en una aplicación Spring Boot. El Apéndice C ofrece una lista exhaustiva de estas propiedades, pero sería imposible repasar todas y cada una de ellas aquí. En su lugar, examinemos algunas de las propiedades más útiles expuestas por Spring Boot.

#### DESHABILITAR EL CACHING DE PLANTILLAS

Si ha estado jugando mucho con la aplicación de lista de lectura, es posible que haya notado que los cambios en cualquiera de las plantillas de Thymeleaf no se aplican a menos que reinicie la aplicación. Eso es porque las plantillas de Thymeleaf se almacenan en caché de forma predeterminada. Esto mejora el rendimiento de la aplicación porque solo compila las plantillas una vez, pero es difícil hacer cambios sobre la marcha durante el desarrollo.

Puede deshabilitar el almacenamiento en caché de la plantilla Thymeleaf configurando `spring.thymeleaf.cache` a falso. Puede hacer esto cuando ejecuta la aplicación desde la línea de comando configurándola como un argumento de línea de comando:

```
$ java -jar lista-de-lectura-0.0.1-SNAPSHOT.jar --spring.thymeleaf.cache=false
```

O, si prefiere desactivar el almacenamiento en caché cada vez que ejecuta la aplicación, puede crear un archivo `application.yml` con las siguientes líneas:

```
primavera:
  tomillo:
    caché: falso
```

Querrá asegurarse de que este archivo `application.yml` no siga a la aplicación en producción, o de lo contrario su aplicación de producción no se dará cuenta de los beneficios de rendimiento del almacenamiento en caché de plantillas.

Como desarrollador, puede que le resulte conveniente tener desactivado el almacenamiento en caché de plantillas todo el tiempo mientras realiza cambios en las plantillas. En ese caso, puede desactivar el almacenamiento en caché de Thymeleaf a través de una variable de entorno:

```
$ export spring_thymeleaf_cache = falso
```

Aunque estamos usando Thymeleaf para las vistas de nuestra aplicación, el almacenamiento en caché de la plantilla se puede desactivar para las otras opciones de plantilla compatibles de Spring Boot configurando estas propiedades:

- `spring.freemarker.cache` (Freemarker)
- `spring.groovy.template.cache` (Plantillas maravillosas)
- `spring.velocity.cache` (Velocidad)

Por defecto, todas estas propiedades son cierto, lo que significa que las plantillas se almacenan en caché. Poniéndolos a falso deshabilita el almacenamiento en caché.

#### CONFIGURACIÓN DEL SERVIDOR INCORPORADO

Cuando ejecuta una aplicación Spring Boot desde la línea de comandos (o mediante Spring Tool Suite), la aplicación inicia un servidor integrado (Tomcat, por defecto) escuchando en el puerto 8080. Esto está bien en la mayoría de los casos, pero puede volverse problemático si se encuentra en la necesidad de ejecutar varias aplicaciones simultáneamente. Si todas las aplicaciones intentan iniciar un servidor Tomcat en el mismo puerto, habrá colisiones de puertos a partir de la segunda aplicación.

Si, por alguna razón, prefiere que el servidor escuche en un puerto diferente, entonces todo lo que necesita hacer es configurar el Puerto de servicio propiedad. Si se trata de un cambio único, es bastante fácil hacerlo como un argumento de línea de comandos:

```
$ java -jar lista-de-lectura-0.0.1-SNAPSHOT.jar --server.port = 8000
```

Pero si desea que el cambio de puerto sea más permanente, puede configurar Puerto de servicio en una de las otras ubicaciones admitidas. Por ejemplo, puede configurarlo en un archivo `application.yml` en la raíz de la ruta de clase de la aplicación:

```
servidor:
  puerto: 8000
```

Además de ajustar el puerto del servidor, es posible que también deba habilitar el servidor para que sirva de forma segura a través de HTTPS. Lo primero que deberá hacer es crear un almacén de claves utilizando el JDK herramienta clave utilidad:

```
$ keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

Se le harán varias preguntas sobre su nombre y organización, la mayoría de las cuales son irrelevantes. Pero cuando se le solicite una contraseña, asegúrese de recordar lo que elija. Por el bien de este ejemplo, elegí "letmein" como contraseña.

Ahora solo necesita establecer algunas propiedades para habilitar HTTPS en el servidor integrado. Podrías especificarlos todos en la línea de comandos, pero eso sería terriblemente inconveniente. En su lugar, probablemente los establecerá en `application.properties` o `application.yml`. En `application.yml`, podrían verse así:

```
servidor:
  puerto: 8443
  ssl:
    almacén de claves: file: ///path/to/mykeys.jks
    clave-almacén-contraseña: letmein
    clave-contraseña: letmein
```

Aquí el Puerto de servicio La propiedad se establece en 8443, una opción común para el desarrollo de servidores HTTPS. Los `server.ssl.key-store` La propiedad debe establecerse en la ruta donde se creó el archivo de almacén de claves. Aquí se muestra con un archivo: `//` URL para cargarlo desde el sistema de archivos, pero si lo empaqueta dentro del archivo JAR de la aplicación, debe usar un `classpath:` URL para hacer referencia a él. Y tanto el `server.ssl.key-store-password` y `server.ssl.key-password` las propiedades se establecen en la contraseña que se proporcionó al crear el almacén de claves.

Con estas propiedades en su lugar, su aplicación debería estar escuchando solicitudes HTTPS en el puerto 8443. (Dependiendo del navegador que esté usando, puede encontrar una advertencia acerca de que el servidor no puede verificar su identidad. Esto no es nada de qué preocuparse al servir desde `localhost` durante el desarrollo).

#### CONFIGURAR EL REGISTRO

La mayoría de las aplicaciones proporcionan alguna forma de registro. E incluso si su aplicación no registra nada directamente, las bibliotecas que utiliza su aplicación ciertamente registrarán su actividad.

De forma predeterminada, Spring Boot configura el registro a través de Logback ( <http://logback.qos.ch> ) a iniciar sesión en la consola en el nivel INFO. Probablemente ya haya visto muchos registros de nivel INFO al ejecutar la aplicación y otros ejemplos.

#### Cambio de Logback por otra implementación de registro

En términos generales, nunca debería necesitar cambiar las implementaciones de registro; Logback debería ser adecuado para usted. Sin embargo, si decide que prefiere usar Log4j o Log4j2, deberá cambiar sus dependencias para incluir el iniciador apropiado para la implementación de registro que desea usar y excluir Logback.

*(continuado)*

Para las compilaciones de Maven, puede excluir Logback excluyendo el iniciador de registro predeterminado resuelto transitivamente por la dependencia del iniciador raíz:

```
<dependencia>
  <groupId> org.springframework.boot </groupId>
  <artifactId> arrancador-de-arranque-primavera </artifactId>
  <exclusiones>
    <exclusión>
      <groupId> org.springframework.boot </groupId>
      <artifactId> registro de arranque de arranque de primavera </artifactId>
    </exclusión>
  </exclusiones>
</dependencia>
```

En Gradle, es más fácil colocar la exclusión debajo de la configuraciones sección:

```
configuraciones {
    all * .exclude grupo: 'org.springframework.boot',
        módulo: 'spring-boot-starter-logging'
}
```

Con el iniciador de registro predeterminado excluido, ahora puede incluir el iniciador para la implementación de registro que prefiere usar. Con una compilación de Maven, puede agregar Log4j de esta manera:

```
<dependencia>
  <groupId> org.springframework.boot </groupId>
  <artifactId> spring-boot-starter-log4j </artifactId>
</dependencia>
```

En una compilación de Gradle, puede agregar Log4j de esta manera:

```
compilar ("org.springframework.boot: spring-boot-starter-log4j")
```

Si prefiere usar Log4j2, cambie el artefacto de "spring-boot-starter-log4j" a "spring-boot-starter-log4j2".

Para tener un control total sobre la configuración de registro, puede crear un archivo logback.xml en la raíz de la ruta de clases (en src / main / resources). A continuación, se muestra un ejemplo de un archivo logback.xml simple que puede utilizar:

```
<configuración>
  <appender name = "STDOUT" class = "ch.qos.logback.core.ConsoleAppender">
    <codificador>
      <patrón>
        % d {HH: mm: ss.SSS} [% subproceso] % -5level% logger {36} -% msg% n </patrón>
      </encoder>
    </appender>

  <logger name = "root" level = "INFO" />
```



```

<root level = "INFO">
  <appender-ref ref = "STDOUT" />
</root>
</configuration>

```

Aparte del patrón utilizado para el registro, esta configuración de Logback es más o menos equivalente a la predeterminada que obtendrá si no tiene un archivo logback.xml. Pero al editar logback.xml puede obtener un control total sobre los archivos de registro de su aplicación. Los detalles de lo que puede incluirse en logback.xml están fuera del alcance de este libro, así que consulte la documentación de Logback para obtener más información.

Aun así, los cambios más comunes que realizará en una configuración de registro son cambiar los niveles de registro y quizás especificar un archivo donde se deben escribir los registros. Con las propiedades de configuración de Spring Boot, puede realizar esos cambios sin tener que crear un archivo logback.xml.

Para establecer los niveles de registro, cree propiedades con el prefijo logging.level, seguido del nombre del registrador para el que desea establecer el nivel de registro. Por ejemplo, suponga que le gustaría establecer el nivel de registro raíz en WARN, pero registrar los registros de Spring Security en el nivel DEBUG. Las siguientes entradas en application.yml se encargarán de ello por usted:

```

Inicio sesión:
  nivel:
    raíz: WARN
    org:
      marco de resorte:
        seguridad: DEBUG

```

Opcionalmente, puede contraer el nombre del paquete Spring Security en una sola línea:

```

Inicio sesión:
  nivel:
    raíz: WARN
    org.springframework.security: DEBUG

```

Ahora suponga que desea escribir las entradas del registro en un archivo llamado BookWorm.log en / var / logs /. los logging.path y archivo de registro las propiedades pueden ayudar con eso:

```

Inicio sesión:
  ruta: / var / logs /
  archivo: BookWorm.log
  nivel:
    raíz: WARN
    org:
      marco de resorte:
        seguridad: DEBUG

```

Suponiendo que la aplicación tenga permisos de escritura en / var / logs /, las entradas del registro se escribirán en /var/logs/BookWorm.log. De forma predeterminada, los archivos de registro rotarán una vez que alcancen los 10 megabytes de tamaño.

De manera similar, todas estas propiedades se pueden configurar en application.properties como esta:

```
logging.path = / var / logs /
logging.file = BookWorm.log
logging.level.root = WARN
logging.level.root.org.springframework.security = DEPURAR
```

Si aún necesita el control total de la configuración de registro, pero prefiere nombrar el archivo de configuración de Logback con otro nombre que no sea logback.xml, puede especificar un nombre personalizado configurando el logging.config propiedad:

```
Inicio sesión:
config:
    classpath: logging-config.xml
```

Aunque normalmente no necesitará cambiar el nombre del archivo de configuración, puede resultar útil si desea utilizar dos configuraciones de registro diferentes para diferentes perfiles de tiempo de ejecución (consulte la sección 3.2.3).

#### CONFIGURAR UNA FUENTE DE DATOS

En este punto, todavía estamos desarrollando nuestra aplicación de lista de lectura. Como tal, la base de datos H2 integrada que estamos usando es perfecta para nuestras necesidades. Pero una vez que llevemos la aplicación a producción, es posible que deseemos considerar una solución de base de datos más permanente.

Aunque puede configurar explícitamente su propio Fuente de datos frijol, generalmente no es necesario. En su lugar, simplemente configure la URL y las credenciales para su base de datos a través de propiedades. Por ejemplo, si está utilizando una base de datos MySQL, su archivo application.yml podría verse así:

```
primavera:
  fuente de datos:
    url: jdbc: mysql: // localhost / readlist
    nombre de usuario: dbuser
    contraseña: dbpass
```

Por lo general, no necesitará especificar el controlador JDBC; Spring Boot puede averiguarlo a partir de la URL de la base de datos. Pero si hay un problema, puede intentar configurar el spring.datasource

. nombre-clase-conductor propiedad:

```
primavera:
  fuente de datos:
    url: jdbc: mysql: // localhost / readlist
    nombre de usuario: dbuser
    contraseña: dbpass
    nombre-clase-controlador: com.mysql.jdbc.Driver
```

Spring Boot utilizará estos datos de conexión al configurar automáticamente el Fuente de datos frijol. los Fuente de datos bean se agrupará, utilizando la agrupación de Tomcat Fuente de datos si es

disponible en el classpath. De lo contrario, buscará y utilizará una de estas otras implementaciones del grupo de conexiones en la ruta de clases:

- HikariCP
- Commons DBCP
- Commons DBCP 2

Aunque estas son las únicas opciones de pool de conexiones disponibles a través de la configuración automática, siempre puede configurar explícitamente un Fuente de datos bean para usar cualquier implementación de grupo de conexiones que desee.

También puede optar por buscar el Fuente de datos desde JNDI estableciendo el `spring.datasource.jndi-name` propiedad:

primavera:

```
fuelle de datos:
    nombre-jndi: java: / comp / env / jdbc / readingListDS
```

Si configura el `spring.datasource.jndi-name` propiedad, las otras propiedades de conexión de la fuente de datos (si están configuradas) serán ignoradas.

Hay muchas formas de influir en los componentes que Spring Boot configura automáticamente con solo establecer una propiedad o dos. Pero este estilo de configuración externalizada no se limita a los beans configurados por Spring Boot. Veamos cómo puede utilizar el mismo mecanismo de configuración de propiedades para ajustar los componentes de su propia aplicación.

### 3.2.2 *Configuración externa de beans de aplicación*

Supongamos que queremos mostrar no solo el título de un libro en la lista de lectura de alguien, sino también proporcionar un enlace al libro en [Amazon.com](https://www.amazon.com). Y no solo queremos proporcionar un enlace al libro, sino que también queremos etiquetar el libro para aprovechar el programa de asociados de Amazon, de modo que si alguien compra un libro a través de uno de los enlaces de nuestra aplicación, recibamos un pequeño pago por la referencia.

Esto es bastante simple de hacer cambiando la plantilla Thymeleaf para representar el título de cada libro como un enlace:

```
<a th: href = "http://www.amazon.com/gp/product/
    + $ {book.isbn}
    + '/ etiqueta = habuma-20' "
    th: text = "$ {book.title}"> Título </a>
```

Esto funcionará perfectamente. Ahora, si alguien hace clic en el enlace y compra el libro, /obtendrá crédito por la referencia. Eso es porque "habuma-20" es *mi* ID de asociado de Amazon. Si prefiere recibir crédito, puede cambiar fácilmente el valor del etiqueta atributo a su ID de asociado de Amazon en la plantilla Thymeleaf.

Aunque es bastante fácil cambiar el ID de asociado de Amazon en la plantilla, todavía está codificado. Solo estamos vinculando a Amazon desde esta plantilla, pero luego podemos agregar funciones a la aplicación donde vinculamos a Amazon desde varias páginas. En ese caso, los cambios en el ID de asociado de Amazon requerirían cambios en varios lugares.

en el código de la aplicación. Es por eso que los detalles como este a menudo se mantienen mejor fuera del código para que se puedan administrar en un solo lugar.

En lugar de codificar el ID de asociado de Amazon en la plantilla, podemos referirnos a él como un valor en el modelo:

```
<a th: href = "http://www.amazon.com/gp/product/"
      + ${book.isbn}
      + '/' tag = ' + ${amazonID} "
  th: text = "${book.title}"> Título </a>
```

Además, ReadingListController deberá completar el modelo en la clave "amazonID" para contener el ID de asociado de Amazon. Nuevamente, no deberíamos codificarlo, sino referirnos a una variable de instancia. Y esa variable de instancia debe completarse desde la configuración de la propiedad. El Listado 3.4 muestra el nuevo ReadingListController,

que completa el modelo a partir de un ID de asociado de Amazon injectado.

#### Listado 3.4 ReadingListController modificado para aceptar una ID de Amazon

lista de lectura de paquetes;

```
import java.util.List;
```

```
import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.boot.context.properties.ConfigurationProperties; import org.springframework.stereotype.Controller;
```

```
import org.springframework.ui.Model;
```

```
import org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.RequestMethod;
```

```
@Controlador
```

```
@RequestMapping ("/")
```

```
@ConfigurationProperties (prefijo = "amazon")
```

```
clase pública ReadingListController {
```

```
    private String AssociateId;
```

```
    ReadingListRepository privado readingListRepository;
```

```
    @Autowired
```

```
    Public ReadingListController (
```

```
        ReadingListRepository readingListRepository) {
```

```
        this.readingListRepository = readingListRepository;
```

```
    }
```

```
    public void setAssociateId (String AssociateId) {
```

```
        this.associateId = AssociateId;
```

```
    }
```

```
    @RequestMapping (método = RequestMethod.GET)
```

```
    lectores de cadenas públicos Libros (lector de lectores, modelo modelo) {
```

```
        Lista <Libro> readingList =
```

← Inyectar con  
propiedades


← Método de setter  
para AssociateId

```

        readingListRepository.findByReader (lector);
    if (readingList! = null) {
        model.addAttribute ("libros", readingList);
        model.addAttribute ("lector", lector);
        model.addAttribute ("amazonID", AssociateId);
    }
    return "readingList";
}

@RequestMapping (método = RequestMethod.POST)
public String addToReadingList (lector lector, libro libro) {
    book.setReader (lector);
    readingListRepository.save (libro);
    return "redireccionar: /";
}
}

```


**Poner AssociateId  
en modelo**

Como puede ver, el `ReadingListController` ahora tiene un `AssociateId` propiedad y un correspondiente `setAssociateId()` método mediante el cual se puede establecer la propiedad. Y `lectores Libros()` ahora agrega el valor de `AssociateId` al modelo bajo la clave "amazonID".

¡Perfecto! Ahora la única pregunta es dónde `AssociateId` obtiene su valor.

Darse cuenta de `ReadingListController` ahora está anotado con `@Configurable` Propiedades. Esto especifica que este bean debe tener sus propiedades inyectadas (a través de métodos `setter`) con valores de propiedades de configuración. Más específicamente, el prefijo atributo especifica que el `ReadingListController` bean se inyectará con propiedades con un prefijo "amazon".

Poniendo todo esto junto, hemos especificado que `ReadingListController` deben tener sus propiedades inyectadas desde las propiedades de configuración con el prefijo "amazon". `ReadingListController` tiene una sola propiedad con un método de establecimiento: el `AssociateId` propiedad. Por lo tanto, todo lo que tenemos que hacer para especificar el ID de asociado de Amazon es agregar un `amazon.associateId` propiedad en una de las ubicaciones de origen de propiedad admitidas.

Por ejemplo, podríamos establecer esa propiedad en `application.properties`:

```
amazon.associateId = habuma-20
```

O en `application.yml`:

```
Amazonas:
  AssociateId: habuma-20
```

O podríamos establecerlo como una variable de entorno, especificarlo como un argumento de línea de comandos o agregarlo en cualquiera de los otros lugares donde se pueden establecer las propiedades de configuración.

**HABILITAR PROPIEDADES DE CONFIGURACIÓN** Técnicamente, el `@Configurable` Propiedades la anotación no funcionará a menos que la haya habilitado agregando clases. Sin embargo, `@EnableConfigurationProperties` en una de sus configuraciones de Spring esto a menudo es innecesario porque toda la configuración

las clases detrás de la configuración automática de Spring Boot ya están anotadas con `@EnableConfigurationProperties`. Por lo tanto, a menos que no esté aprovechando la configuración automática en absoluto (¿y por qué sucedería eso?), No debería necesitar usar explícitamente `@ EnableConfigurationProperties`.

También vale la pena señalar que el solucionador de propiedades de Spring Boot es lo suficientemente inteligente como para tratar las propiedades en formato camel como intercambiables con propiedades con nombres similares con guiones o guiones bajos. En otras palabras, una propiedad llamada `amazon.associateld` es equivalente a ambos `amazon.associate_id` y `amazon.associate-id`. Siéntete libre de usar la convención de nomenclatura que más le convenga.

#### RECOGIDA DE PROPIEDADES EN UNA CLASE

Aunque anotando `ReadingListController` con `@ Propiedades de configuración` funciona bien, puede que no sea lo ideal. ¿No parece un poco extraño que el prefijo de propiedad sea "amazon" cuando, de hecho, `ReadingListController` ¿Tiene poco que ver con Amazon? Además, las mejoras futuras pueden presentar la necesidad de configurar propiedades no relacionadas a Amazon en `ReadingListController`.

En lugar de capturar las propiedades de configuración en `ReadingListController`, puede ser mejor anotar un bean separado con `@ Propiedades de configuración` y deje que ese bean recopile todas las propiedades de configuración. `AmazonProperties` en el listado 3.5, por ejemplo, captura las propiedades de configuración específicas de Amazon.

#### Listado 3.5 Capturando propiedades de configuración en un bean

lista de lectura de paquetes;

importar org.springframework.boot.context.properties.

Propiedades de configuración;

import org.springframework.stereotype.Component;

`@Componente`

`@ConfigurationProperties ("amazon")`

`AmazonProperties` de clase pública {

`private String Associateld;`

`public void setAssociateld (String Associateld) {`

`this.associateld = Associateld;`

`}`

`public String getAssociateld () {`

`return Associateld;`

`}`

`}`

← Inyectar con "amazon" -  
propiedades prefijadas

← Associateld  
método setter

Con `AmazonProperties` capturando el `amazon.associateld` propiedad de configuración, podemos cambiar `ReadingListController` ( como se muestra en el listado 3.6) para extraer el ID de asociado de Amazon de un `AmazonProperties`.

### Listado 3.6 ReadingListController inyectado con AmazonProperties

lista de lectura de paquetes;

```
import java.util.List;
```

```
importar org.springframework.beans.factory.annotation.Autowired; importar org.springframework.stereotype.Controller;
```

```
import org.springframework.ui.Model;
```

```
import org.springframework.web.bind.annotation.RequestMapping; import
org.springframework.web.bind.annotation.RequestMethod;
```

```
@Controlador
```

```
@RequestMapping ("/")
```

```
clase pública ReadingListController {
```

```
    ReadingListRepository privado readingListRepository;
```

```
    AmazonProperties privadas amazonProperties;
```

```
    @Autowired
```

```
    Public ReadingListController (
```

```
        ReadingListRepository readingListRepository,
```

```
        AmazonProperties amazonProperties) {
```

```
        this.readingListRepository = readingListRepository;
```

```
        this.amazonProperties = amazonProperties;
```

```
    }
```

```
    @RequestMapping (método = RequestMethod.GET)
```

```
    lectores de cadenas públicos Libros (lector de lectores, modelo modelo) {
```

```
        Lista <Libro> readingList =
```

```
            readingListRepository.findByReader (lector);
```

```
        if (readingList! = null) {
```

```
            model.addAttribute ("libros", readingList);
```

```
            model.addAttribute ("lector", lector);
```

```
            model.addAttribute ("amazonID", amazonProperties.getAssociateId ());
```

```
        }
```

```
        return "readingList";
```

```
    }
```

```
    @RequestMapping (método = RequestMethod.POST)
```

```
    public String addToReadingList (lector lector, libro libro) {
```

```
        book.setReader (lector);
```

```
        readingListRepository.save (libro);
```

```
        return "redireccionar: /";
```

```
    }
```

```
}
```

← Inyectar  
AmazonProperties

← Agregar ID de asociado  
modelar

ReadingListController ya no es el destinatario directo de las propiedades de configuración. En cambio, obtiene la información que necesita de la inyección AmazonProperties frijol.

Como hemos visto, las propiedades de configuración son útiles para ajustar tanto los componentes configurados automáticamente como los detalles inyectados en nuestros propios beans de aplicación. Pero que si

¿Necesitamos configurar diferentes propiedades para diferentes entornos de implementación? Echemos un vistazo a cómo usar los perfiles de Spring para configurar la configuración específica del entorno.

### 3.2.3 Configurar con perfiles

Cuando las aplicaciones se implementan en diferentes entornos de ejecución, generalmente hay algunos detalles de configuración que serán diferentes. Los detalles de una conexión de base de datos, por ejemplo, probablemente sean diferentes en un entorno de desarrollo que en un entorno de garantía de calidad, y aún diferentes en un entorno de producción. Spring Framework introdujo soporte para la configuración basada en perfiles en Spring 3.1. Los perfiles son un tipo de configuración condicional en la que se utilizan o ignoran diferentes beans o clases de configuración en función de los perfiles que están activos en tiempo de ejecución.

Por ejemplo, suponga que la configuración de seguridad que creamos en el listado 3.1 es para fines de producción, pero la configuración de seguridad configurada automáticamente está bien para el desarrollo. En ese caso, podemos anotar SecurityConfig con @Perfil como esto:

```
@Profile ("producción")
@Configuration
@EnableWebSecurity
SecurityConfig de clase pública extiende WebSecurityConfigurerAdapter {

    ...

}
```

Los @ Perfil La anotación utilizada aquí requiere que el perfil de "producción" esté activo en tiempo de ejecución para que se aplique esta configuración. Si el perfil de "producción" no está activo, esta configuración se ignorará y, a falta de otra configuración de seguridad primordial, se aplicará la configuración de seguridad configurada automáticamente.

Los perfiles se pueden activar configurando el spring.profiles.active propiedad utilizando cualquiera de los medios disponibles para establecer cualquier otra propiedad de configuración. Por ejemplo, puede activar el perfil de "producción" ejecutando la aplicación en la línea de comando de esta manera:

```
$ java -jar lista de lectura-0.0.1-SNAPSHOT.jar -
    spring.profiles.active = producción
```

O puede agregar el spring.profiles.active propiedad a application.yml:

```
primavera:
  perfiles:
    activo: producción
```

O puede establecer una variable de entorno y ponerla en application.properties o usar cualquiera de las otras opciones mencionadas al principio de la sección 3.2.

Pero debido a que Spring Boot se autoconfigura mucho para usted, sería muy inconveniente escribir una configuración explícita solo para que pueda tener un lugar para poner @ Perfil.



Afortunadamente, Spring Boot admite perfiles para propiedades establecidas en `application.properties` y `application.yml`.

Para demostrar las propiedades perfiladas, suponga que desea una configuración de registro diferente en producción que en desarrollo. En producción, solo le interesan las entradas de registro en el nivel WARN o superior, y desea escribir las entradas de registro en un archivo de registro. En desarrollo, sin embargo, solo desea que las cosas se registren en la consola y en el nivel DEBUG o superior.

Todo lo que necesita hacer es crear configuraciones independientes para cada entorno. Sin embargo, la forma de hacerlo depende de si está utilizando una configuración de archivo de propiedades o una configuración YAML.

#### TRABAJAR CON ARCHIVOS DE PROPIEDADES ESPECÍFICAS DE PERFIL

Si está utilizando `application.properties` para expresar las propiedades de configuración, puede proporcionar propiedades específicas del perfil creando archivos de propiedades adicionales nombrados con el patrón “`application- {profile} .properties`”.

Para el escenario de registro, la configuración de desarrollo estaría en un archivo llamado `application-development.properties` y contendría propiedades para el registro detallado escrito en consola:

```
logging.level.root = DEBUG
```

Pero para la producción, `application-production.properties` configuraría el registro para que esté en el nivel WARN y superior y para escribir en un archivo de registro:

```
logging.path = / var / logs /
logging.file = BookWorm.log
logging.level.root = WARN
```

Mientras tanto, cualquier propiedad que no sea específica de ningún perfil o que sirva como predeterminada (en caso de que una configuración específica del perfil no especifique lo contrario) puede continuar expresándose en `application.properties`:

```
amazon.associateId = habuma-20
logging.level.root = INFO
```

#### CONFIGURACIÓN CON ARCHIVOS YAML MULTIPROFIL

Si está utilizando YAML para las propiedades de configuración, puede seguir una convención de nomenclatura similar a la de los archivos de propiedades. Es decir, puede crear archivos YAML cuyos nombres sigan un patrón de “`aplicación- {perfil} .yml`” y continuar colocando propiedades sin perfil en `application.yml`.

Pero con YAML, también tiene la opción de expresar las propiedades de configuración para todos los perfiles en un solo archivo `application.yml`. Por ejemplo, la configuración de registro que queremos se puede declarar en `application.yml` así:

```
Inicio sesión:
  nivel:
    raíz: INFO
```

```

---

primavera:
  perfiles: desarrollo

Inicio sesión:
  nivel:
    raíz: DEBUG

---

primavera:
  perfiles: producción

Inicio sesión:
  ruta: / tmp /
  archivo: BookWorm.log
  nivel:
    raíz: WARN

```

Como puede ver, este archivo `application.yml` está dividido en tres secciones por un conjunto de guiones triples (`---`). La segunda y tercera sección especifican cada una un valor para primavera

. `perfiles`. Esta propiedad indica a qué perfil se aplican las propiedades de cada sección. Las propiedades definidas en la sección central se aplican al desarrollo porque establece

`Spring.profiles` al desarrollo". Del mismo modo, la última sección tiene `Spring.profiles` establecido en "producción", haciéndolo aplicable cuando el perfil de "producción" está activo.

La primera sección, por otro lado, no especifica un valor para `Spring.profiles`.

Por lo tanto, sus propiedades son comunes a todos los perfiles o son predeterminadas si el perfil activo no tiene las propiedades establecidas.

Además de la configuración automática y las propiedades de configuración externa, Spring Boot tiene otro truco bajo la manga para simplificar una tarea de desarrollo común: configura automáticamente una página para que se muestre cuando una aplicación encuentra algún error. Para concluir este capítulo, echaremos un vistazo a la página de error de Spring Boot y veremos cómo personalizarla para que se ajuste a nuestra aplicación.

### 3.3 Personalizar las páginas de error de la aplicación

Los errores ocurren. Incluso algunas de las aplicaciones más sólidas que se ejecutan en producción ocasionalmente tienen problemas. Aunque es importante reducir la posibilidad de que un usuario encuentre un error, también es importante que su aplicación aún se presente bien al mostrar una página de error.

En los últimos años, las páginas de errores creativos se han convertido en una forma de arte. Si alguna vez has visto la página de error inspirada en Star Wars en [GitHub.com](https://github.com) o [DropBox.com](https://dropbox.com) página de error similar a Escher, tienes una idea de lo que estoy hablando.

No sé si ha encontrado algún error al probar la aplicación de lista de lectura, pero si es así, probablemente haya visto una página de error muy parecida a la de la figura 3.1.

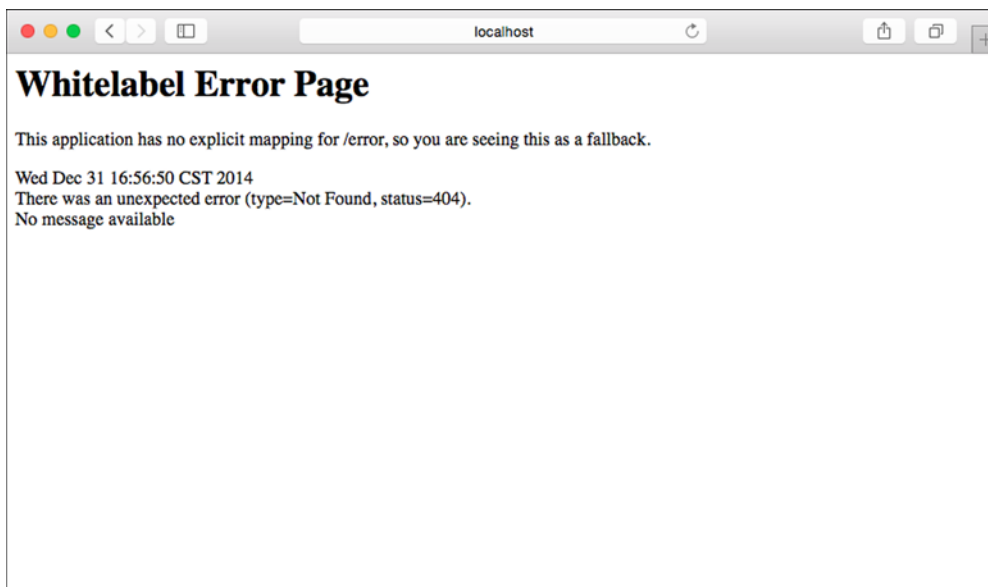


Figura 3.1 Página de error de etiqueta blanca predeterminada de Spring Boot.

Spring Boot ofrece esta página de error de "etiqueta blanca" de forma predeterminada como parte de la configuración automática. Aunque es un poco más atractivo que un seguimiento de pila, no se compara con algunas de las grandes obras de arte de errores disponibles en Internet. Con el fin de presentar los fallos de su aplicación como obras maestras, probablemente desee crear una página de error personalizada para sus aplicaciones.

El controlador de errores predeterminado que Spring Boot configura automáticamente busca una vista cuyo nombre sea "error". Si no puede encontrar uno, utiliza su vista de error de etiqueta blanca predeterminada que se muestra en la figura 3.1. Por lo tanto, la forma más fácil de personalizar la página de error es crear una vista personalizada que se resuelva para una vista llamada "error".

En última instancia, esto depende de los resolutores de vista que estén en su lugar cuando se resuelva la vista de error. Esto incluye

- Cualquier frijol que implemente Spring's Vista interfaz y tiene un ID de bean de "error" (resuelto por Spring's BeanNameViewResolver)
- Una plantilla Thymeleaf llamada "error.html" si Thymeleaf está configurada Una plantilla
- FreeMarker llamada "error.ftl" si FreeMarker está configurado Una plantilla Velocity llamada
- "error.vm" si Velocity está configurada Una plantilla JSP llamada "error.jsp" si se usa Vistas JSP
- 

Debido a que estamos usando Thymeleaf para la aplicación de lista de lectura, todo lo que debemos hacer para personalizar la página de error es crear un archivo llamado "error.html" y colocarlo en la carpeta de plantillas junto con nuestras otras plantillas de aplicación. El Listado 3.7 muestra un reemplazo simple pero efectivo para la página de error de etiqueta blanca predeterminada.

Listado 3.7 Página de error personalizada para la aplicación de lista de lectura

```

<html>
  <cabeza>
    <title> ¡Vaya! </title>
    <link rel = "stylesheet" th: href = "@ {/ style.css}"> </link>
  </head>

  <html>
    <div class = "errorPage">
      <span class = "oops"> ¡Vaya! </span> <br/>
      <img th: src = "@ {/ MissingPage.png}"> </img>
      <p> Parece haber un problema con la página que solicitaste.
        (<span th: text = "$ {path}"> </span>). </p>

      <p th: text = "$ {Detalles:' + mensaje}"> </p> </div>

    </html>
  </html>

```

mostrar  
solicitado  
camino

Mostrar error  
detalles

Esta plantilla de error personalizada debe llamarse "error.html" y colocarse en el directorio de plantillas para que la encuentre el solucionador de plantillas de Thymeleaf. Para una compilación típica de Maven o Gradle, eso significa ponerlo en `src / main / resources / templates` para que esté en la raíz de la ruta de clases durante el tiempo de ejecución.

En su mayor parte, esta es una plantilla simple de Thymeleaf que muestra una imagen y algún texto de error. Hay dos piezas específicas de información que también presenta: la ruta de solicitud del error y el mensaje de excepción. Sin embargo, estos no son los únicos detalles disponibles para una página de error. De forma predeterminada, Spring Boot hace que los siguientes atributos de error estén disponibles para la vista de errores:

- **marca de tiempo** —La hora en que ocurrió el error
- **estado** —El código de estado HTTP
- **error** —La razón del error
- **excepción** —El nombre de clase de la excepción
- **mensaje** —El mensaje de excepción (si el error fue causado por una excepción)
- **errores** —Cualquier error de un `BindingResult` excepción (si el error fue causado por una excepción)
- **rastros** —El seguimiento de la pila de excepciones (si el error fue causado por una excepción)
- **camino** : La ruta de URL solicitada cuando se produjo el error

Algunos de estos atributos, como `camino`, son útiles para comunicar el problema al usuario. Otros, como `rastros`, debe usarse con moderación, ocultarse o usarse inteligentemente en la página de error para mantener la página de error lo más fácil de usar posible.

También notará que la plantilla hace referencia a una imagen llamada `MissingPage.png`. El contenido real de la imagen no es importante, así que siéntete libre de flexionar tus músculos de diseño gráfico y crear una imagen que se adapte a ti. Pero asegúrese de ponerlo en `src / main / resources / static` o `src / main / resources / public` para que pueda ser servido cuando la aplicación se esté ejecutando.



Figura 3.2 Una página de error personalizada muestra estilo ante el fracaso

La figura 3.2 muestra lo que verá el usuario cuando se produzca un error. Puede que no sea una obra de arte, pero creo que eleva un poco la estética de la página de error de la aplicación.

### 3.4 *Resumen*

Spring Boot elimina gran parte de la configuración estándar que a menudo se requiere en las aplicaciones Spring. Pero al permitir que Spring Boot haga toda la configuración, confía en él para configurar los componentes de la manera que se adapte a su aplicación. Cuando la configuración automática no se ajusta a sus necesidades, Spring Boot le permite anular y ajustar la configuración que proporciona.

Anular la configuración automática es una simple cuestión de escribir una configuración Spring explícita como lo haría en ausencia de Spring Boot. La configuración automática de Spring Boot está diseñada para favorecer la configuración proporcionada por la aplicación sobre su propia configuración automática.

Incluso cuando la configuración automática sea adecuada, es posible que deba ajustar algunos detalles. Spring Boot habilita varios solucionadores de propiedades que le permiten modificar la configuración estableciendo propiedades como variables de entorno, en archivos de propiedades, en archivos YAML y de varias otras formas. Este mismo modelo de configuración basado en propiedades puede incluso aplicarse a componentes definidos por la aplicación, lo que permite la inyección de valor en las propiedades del bean desde fuentes de configuración externas.

Spring Boot también configura automáticamente una página de error de etiqueta blanca simple. Aunque es más fácil de usar que una excepción y un seguimiento de pila, la página de error de etiqueta blanca todavía deja mucho que desear estéticamente. Afortunadamente, Spring Boot ofrece varias opciones para personalizar o reemplazar completamente la página de error de etiqueta blanca para que se adapte al estilo específico de una aplicación.

Ahora que hemos escrito una aplicación completa con Spring Boot, deberíamos verificar que realmente hace lo que esperamos que haga. Es decir, en lugar de pincharlo en el navegador web manualmente, deberíamos escribir algunas pruebas automatizadas y repetibles que ejerciten la aplicación y demuestren que está funcionando correctamente. Eso es exactamente lo que haremos en el próximo capítulo.

# Prueba con Spring Boot



## *Este capítulo cubre*

- Pruebas de integración
- Prueba de aplicaciones en un servidor
- Utilidades de prueba de Spring Boot

Se ha dicho que si no sabes a dónde vas, cualquier camino te llevará allí. Pero con el desarrollo de software, si no sabe a dónde va, probablemente terminará con una aplicación con errores que nadie puede usar.

La mejor manera de saber con certeza hacia dónde se dirige al escribir aplicaciones es escribir pruebas que afirmen el comportamiento deseado de una aplicación. Si esas pruebas fallan, sabrá que tiene trabajo por hacer. Si pasan, entonces ha llegado (al menos hasta que piense en algunas pruebas más que pueda escribir).

Ya sea que escriba pruebas primero o después de que el código ya se haya escrito, es importante que escriba pruebas no solo para verificar la precisión de su código, sino también para asegurarse de que hace todo lo que espera. Las pruebas también son una gran protección para asegurarse de que las cosas no se rompan a medida que su aplicación continúa evolucionando.

Cuando se trata de escribir pruebas unitarias, Spring generalmente está fuera de escena. El acoplamiento flexible y el diseño impulsado por la interfaz, que Spring fomenta, hace que sea realmente fácil escribir pruebas unitarias. Pero Spring no está necesariamente involucrado en esas pruebas unitarias.

Las pruebas de integración, por otro lado, requieren algo de ayuda de Spring. Si Spring es responsable de configurar y conectar los componentes en su aplicación de producción, Spring también debería ser responsable de configurar y conectar esos componentes en sus pruebas.

Muelles SpringJUnit4ClassRunner ayuda a cargar un contexto de aplicación Spring en pruebas de aplicación basadas en JUnit. Spring Boot se basa en el soporte de pruebas de integración de Spring al permitir la configuración automática y el inicio del servidor web al probar las aplicaciones Spring Boot. También ofrece un puñado de utilidades de prueba útiles.

En este capítulo, veremos todas las formas en que Spring Boot admite las pruebas de integración. Comenzaremos viendo cómo realizar pruebas con un contexto de aplicación totalmente habilitado para Spring Boot.

## 4.1 Configuración automática de pruebas de integración

En el centro de todo lo que hace Spring Framework, su tarea más esencial es conectar todos los componentes que componen una aplicación. Lo hace leyendo una especificación de cableado (ya sea XML, basada en Java, basada en Groovy o de otro tipo), instanciando beans en un contexto de aplicación e inyectando beans en otros beans que dependen de ellos.

Cuando se prueba la integración de una aplicación Spring, es importante dejar que Spring conecte los beans que son el objetivo de la prueba de la misma manera que conecta esos beans cuando la aplicación se ejecuta en producción. Claro, es posible que pueda instanciar manualmente los componentes e inyectarlos entre sí, pero para cualquier aplicación sustancialmente grande, eso puede ser una tarea ardua. Además, Spring ofrece instalaciones adicionales como escaneo de componentes, cableado automático y aspectos declarativos como almacenamiento en caché, transacciones y seguridad. Dado todo lo que se necesitaría para recrear lo que hace Spring, generalmente es mejor dejar que Spring haga el trabajo pesado, incluso en una prueba de integración.

Spring ha ofrecido un excelente soporte para las pruebas de integración desde la versión 1.1.1. Desde Spring 2.5, se ha ofrecido soporte para pruebas de integración en forma de SpringJUnit4ClassRunner, un corredor de clase JUnit que carga un contexto de aplicación Spring para usar en una prueba JUnit y habilita el cableado automático de beans en la clase de prueba.

Por ejemplo, considere la siguiente lista, que muestra una prueba de integración Spring muy básica.

### Listado 4.1 Prueba de integración Spring con SpringJUnit4ClassRunner

```
@RunWith (SpringJUnit4ClassRunner.class)
@ContextConfiguration (
    clases = AddressBookConfiguration.class)
AddressServiceTests de clase pública {

    @Autowired
    addressService privado addressService;

    @Prueba
```

← Aplicación de cargas  
contexto

← Inyecta dirección  
Servicio



```

public void testService () {
    Dirección address = addressService.findByLastName ("Sherman"); assertEquals ("P", dirección.getFirstName ());

    assertEquals ("Sherman", address.getLastName ());
    assertEquals ("42 Wallaby Way", address.getAddressLine1 ()); assertEquals ("Sydney", address.getCity ());

    assertEquals ("Nueva Gales del Sur", address.getState ()); assertEquals ("2000",
    address.getPostCode ());
}
}

```

← Dirección de pruebas  
Servicio

Como se puede ver, `AddressServiceTests` está anotado con `@Corre` con y `@ContextConfiguration`. `@Corre` con es dado `SpringJUnit4ClassRunner.class` para habilitar Spring pruebas de integración. <sup>1</sup> Mientras tanto, `@ContextoConfiguración` especifica cómo cargar el contexto de la aplicación. Aquí le pedimos que cargue el contexto de la aplicación Spring dada la especificación definida en `AddressBookConfiguration`.

Además de cargar el contexto de la aplicación, `SpringJUnit4ClassRunner` también permite inyectar beans desde el contexto de la aplicación en la propia prueba a través del cableado automático. Debido a que esta prueba está dirigida a un `AddressService` bean, se conecta automáticamente a la prueba. Finalmente, el `testService ()` El método realiza llamadas al servicio de direcciones y verifica los resultados.

Aunque `@ContextoConfiguración` hace un gran trabajo al cargar el contexto de la aplicación Spring, no lo carga con el tratamiento completo de Spring Boot. Las aplicaciones de Spring Boot son finalmente cargadas por `SpringApplication`, ya sea explícitamente (como en el listado 2.1) o usando `SpringBootTestInitializer` ( que veremos en el capítulo 8). `SpringApplication` no solo carga el contexto de la aplicación, sino que también habilita el registro, la carga de propiedades externas (`application.properties` o `application.yml`) y otras características de Spring Boot. Si está usando `@ContextConfiguration`, no obtendrá esas funciones.

Para recuperar esas funciones en sus pruebas de integración, puede cambiar `@ContextoConfiguración` para Spring Boot's `@SpringApplicationConfiguration`:

```

@RunWith (SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration (
    clases = AddressBookConfiguration.class)
AddressServiceTests de clase pública {
    ...
}

```

El uso de `@SpringApplicationConfiguration` es en gran parte idéntico a `@ContextConfiguration`. Pero a diferencia `@ContextConfiguration` `@SpringApplicationConfigu-` racionar carga el contexto de la aplicación Spring usando `SpringApplication` de la misma forma y con el mismo trato que recibiría si se estuviera cargando en una aplicación de producción. Esto incluye la carga de propiedades externas y el registro de Spring Boot.

<sup>1</sup> A partir de Spring 4.2, puede usar opcionalmente `SpringClassRule` y `SpringMethodRule` como JUnit basado en reglas alternativas a `SpringJUnit4ClassRunner`.

Baste decir que, en su mayor parte, `@ SpringApplicationConfiguration` reemplaza a `@ ContextoConfiguración` al escribir pruebas para aplicaciones Spring Boot. Ciertamente usaremos `@ SpringApplicationConfiguration` a lo largo de este capítulo a medida que escribimos pruebas para nuestra aplicación Spring Boot, incluidas las pruebas dirigidas al front-end web de la aplicación.

Hablando de pruebas web, eso es lo que haremos a continuación.

## 4.2 Prueba de aplicaciones web

Una de las cosas buenas de Spring MVC es que promueve un modelo de programación en torno a objetos Java simples y antiguos (POJO) que están anotados para declarar cómo deben procesar las solicitudes web. Este modelo de programación no solo es simple, sino que le permite tratar los controladores como lo haría con cualquier otro componente de su aplicación. Incluso podría tener la tentación de escribir pruebas en su controlador que las prueben como POJO.

Por ejemplo, considere el `addToReadingList ()` método de Leyendo lista-  
Controlador:

```
@RequestMapping (método = RequestMethod.POST)
public String addToReadingList (libro libro) {
    book.setReader (lector);
    readingListRepository.save (libro);
    return "redireccionar: / readList";
}
```

Si ignorara la `@ RequestMapping` método, se quedaría con un método Java bastante básico. No se necesitaría mucho para imaginar una prueba que proporcione un simulacro implementación de `ReadingListRepository`, llamadas `addToReadingList ()` directamente, y afirma el valor de retorno y verifica la llamada al repositorio `salvar()` método.

El problema con una prueba de este tipo es que solo prueba el método en sí. Si bien eso es mejor que ninguna prueba, falla al probar que el método maneja una CORREO solicitud a `/ readingList`. Tampoco prueba que los campos de formulario estén correctamente vinculados a la Libro parámetro. Y aunque se podría afirmar que el regresó Cuerda contiene un cierto valor, sería imposible probar definitivamente que la solicitud es, de hecho, redirigida a `/ readingList` una vez finalizado el método.

Para probar correctamente una aplicación web, necesita una forma de enviarle solicitudes HTTP reales y afirmar que procesa esas solicitudes correctamente. Afortunadamente, hay dos opciones disponibles para los desarrolladores de aplicaciones Spring Boot que hacen posible ese tipo de pruebas:

- *Spring Mock MVC*—Permite que los controladores se prueben en una aproximación simulada de un contenedor de servlets sin realmente iniciar un servidor de aplicaciones
- *Pruebas de integración web*—Actualmente inicia la aplicación en un contenedor de servlet integrado (como Tomcat o Jetty), lo que permite realizar pruebas que ejercitan la aplicación en un servidor de aplicaciones real

Cada uno de este tipo de pruebas tiene sus pros y sus contras. Obviamente, iniciar un servidor resultará en una prueba más lenta que burlarse de un contenedor de servlets. Pero no hay duda de que

Las pruebas basadas en servidor están más cerca del entorno del mundo real en el que se ejecutarán cuando se implementen en producción.

Comenzaremos por ver cómo puede probar una aplicación web utilizando el marco de prueba Mock MVC de Spring. Luego, en la sección 4.3, verá cómo escribir pruebas en una aplicación que se está ejecutando en un servidor de aplicaciones.

#### 4.2.1 *Burlándose de Spring MVC*

Desde Spring 3.2, Spring Framework ha tenido una instalación muy útil para probar aplicaciones web burlándose de Spring MVC. Esto hace posible realizar solicitudes HTTP contra un controlador sin ejecutar el controlador dentro de un contenedor de servlet real. En cambio, el framework Mock MVC de Spring se burla lo suficiente de Spring MVC para que sea casi como si la aplicación se estuviera ejecutando dentro de un contenedor de servlets... pero no es así.

Para configurar un Mock MVC en su prueba, puede usar `MockMvcBuilders`. Esta clase ofrece dos métodos estáticos:

- `standaloneSetup ()` —Construye un `MockMvc` para servir a uno o más controladores creados y configurados manualmente
- `webApplicationContextSetup ()` —Construye un MVC simulado usando un contexto de aplicación Spring, que presumiblemente incluye uno o más controladores configurados

La principal diferencia entre estas dos opciones es que `standaloneSetup ()` espera que cree una instancia e inyecte manualmente los controladores que desea probar, mientras que `webApplicationContextSetup ()` funciona a partir de una instancia de `WebApplicationContext`, cuales probablemente fue cargado por Spring. El primero es un poco más parecido a una prueba unitaria en el sentido de que probablemente solo lo usará para pruebas muy enfocadas en un solo controlador. Sin embargo, este último permite que Spring cargue sus controladores y sus dependencias para una prueba de integración completa.

Para nuestros propósitos, usaremos `webApplicationContextSetup ()` para que podamos probar el `ReadingListController` ya que se ha creado una instancia e inyectado desde el contexto de la aplicación que Spring Boot ha configurado automáticamente.

los `webApplicationContextSetup ()` toma una `WebApplicationContext` como argumento. Por lo tanto, necesitaremos anotar la clase de prueba con `@ WebAppConfiguration` y use `@Autowired` para inyectar el `WebApplicationContext` en la prueba como una variable de instancia. La siguiente lista muestra el punto de partida para nuestras pruebas Mock MVC.

Listado 4.2 Creación de un MVC simulado para controladores de pruebas de integración

```
@RunWith (SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration (
    clases = ReadingListApplication.class)
@WebAppConfiguration
public class MockMvcWebTests {
```

← Habilita la web  
prueba de contexto

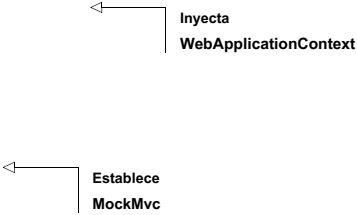
```

@Autowired
WebApplicationContext webContext privado;

MockMvc mockMvc privado;

@Before
setupMockMvc público vacío () {
    mockMvc = MockMvcBuilders
        .webAppContextSetup (webContext)
        .construir();
}
}

```



Los `@WebAppConfiguration` anotación declara que el contexto de la aplicación creado por `SpringJUnit4ClassRunner` debería ser un `WebApplicationContext` (a diferencia de un básico no web `ApplicationContext`).

los `setupMockMvc ()` El método está anotado con JUnit's `@Before`, indicando que debe ejecutarse antes que cualquier método de prueba. Pasa lo inyectado `WebApplicationContext` en el `webAppContextSetup ()` método y luego llama `construir()` para producir un

`MockMvc` instancia, que se asigna a una variable de instancia para que la utilicen los métodos de prueba.

Ahora que tenemos un `MockMvc`, estamos listos para escribir algunos métodos de prueba. Comencemos con un método de prueba simple que realiza un HTTP OBTENER request against `/readingList` y afirma que el modelo y la vista cumplen con nuestras expectativas. El seguimiento página principal()

El método de prueba hace lo que necesitamos:

```

@Prueba
public void homePage () lanza Exception {
    mockMvc.perform (MockMvcRequestBuilders.get ("/ readingList"))
        .andExpect (MockMvcResultMatchers.status ().isOk ())
        .andExpect (MockMvcResultMatchers.view ().name ("readingList"))
        .andExpect (MockMvcResultMatchers.model ().attributeExists ("libros"))
        .andExpect (MockMvcResultMatchers.model ().atributo ("libros",
            Matchers.is (Matchers.empty ()))));
}

```

Como puede ver, en este método de prueba se utilizan muchos métodos estáticos, incluidos métodos estáticos de `Spring MockMvcRequestBuilders` y `MockMvcResultMatchers`,

así como de la biblioteca de Hamcrest `Matchers`. Antes de profundizar en los detalles de este método de prueba, agreguemos algunas importaciones estáticas para que el código sea más fácil de leer:

```

import org.hamcrest.Matchers estáticos. *;
import org.springframework.test.web.servlet.request estático.
    ↳ MockMvcRequestBuilders. *;
import org.springframework.test.web.servlet.result estático.
    ↳ MockMvcResultMatchers. *;

```

Con esas importaciones estáticas en su lugar, el método de prueba se puede reescribir así:

```

@Prueba
public void homePage () lanza Exception {
    mockMvc.perform (get ("/ readingList"))
        .andExpect (status (). isOk ())
        .andExpect (view (). name ("readingList"))
        .andExpect (modelo (). attributeExists ("libros"))
        .andExpect (modelo (). atributo ("libros", es (vacío ()))));
}

```

Ahora, el método de prueba se lee casi de forma natural. Primero realiza una OBTENER solicitud contra / readingList. Entonces espera que la solicitud sea exitosa ( isOk () afirma un código de respuesta HTTP 200) y que la vista tiene un nombre lógico de leyendo lista. También afirma que el modelo contiene un atributo llamado libros, pero ese atributo es una colección vacía. Todo es muy sencillo.

Lo principal a tener en cuenta aquí es que en ningún momento se implementa la aplicación en un servidor web. En cambio, se ejecuta dentro de un Spring MVC simulado, lo suficientemente capaz de manejar las solicitudes HTTP que le lanzamos a través del MockMvc ejemplo.

Bastante bien, ¿eh?

Probemos un método de prueba más. Esta vez lo haremos un poco más interesante enviando un HTTP CORREO solicitud para publicar un nuevo libro. Deberíamos esperar que después de la CORREO se maneja la solicitud, la solicitud será redirigida de nuevo a / readingList y que el libros El atributo en el modelo contendrá el libro recién agregado. La siguiente lista muestra cómo podemos usar Spring's Mock MVC para hacer este tipo de prueba.

Listado 4.3 Prueba de la publicación de un libro nuevo

```

@Prueba
public void postBook () arroja una excepción {mockMvc.perform (post ("/
readingList")

    . contentType (MediaType.APPLICATION_FORM_URLENCODED)
    . param ("titulo", "TÍTULO DEL LIBRO")
    . param ("autor", "AUTOR DEL LIBRO")
    . param ("isbn", "1234567890")
    . param ("descripción", "DESCRIPCIÓN"))
    . andExpect (estado (). is3xxRedirection ())
    . andExpect (header (). string ("Ubicación", "/" readList")));

Libro esperado Libro = nuevo Libro (); WaitBook.setld (1L);

libro esperado.setReader ("craig");
WaitBook.setTitle ("TÍTULO DEL LIBRO");
ExpertoBook.setAuthor ("AUTOR DEL LIBRO");
WaitBook.setIsbn ("1234567890");
WaitBook.setDescription ("DESCRIPCIÓN");

mockMvc.perform (get ("/ readingList"))
    . andExpect (status (). isOk ())
    . andExpect (view (). name ("readingList"))

```

**Realiza Solicitud POST**

**Establece libro esperado**

**Realiza GET petición**

```

        .andExpect (modelo (). attributeExists ("libros"))
        .andExpect (modelo (). atributo ("libros", hasSize (1)))
        .y Esperar (modelo (). atributo ("libros",
            contiene (samePropertyValuesAs (libro esperado))));
    }

```

Obviamente, la prueba del listado 4.3 es un poco más complicada. En realidad, son dos pruebas en un método. La primera parte publica el libro y afirma los resultados de esa solicitud. La segunda parte realiza una nueva OBTENER solicitud en la página de inicio y afirma que el libro recién creado está en el modelo.

Al publicar el libro, debemos asegurarnos de establecer el tipo de contenido en "aplicación / x-www-form-urlencoded" (con `MediaType.APPLICATION_FORM_URLENCODED`) como eso será el tipo de contenido que enviará un navegador cuando el libro se publique en la aplicación en ejecución. Luego usamos el `MockMvcRequestBuilders` 's `param ()` método para establecer los campos que simulan el formulario que se envía. Una vez que se ha realizado la solicitud, afirmamos que la respuesta es una redirección a `/ readingList`.

Suponiendo que gran parte del método de prueba pasa, pasamos a la segunda parte. Primero, configuramos un `Libro` objeto que contiene los valores esperados. Usaremos esto para comparar con el valor que está en el modelo después de obtener la página de inicio.

Luego realizamos un OBTENER solicitud de `/ readingList`. En su mayor parte, esto no es diferente de cómo probamos la página de inicio antes, excepto que en lugar de una colección vacía en el modelo, estamos verificando que tenga un elemento y que el elemento sea el mismo que el esperado. Libro creamos. Si es así, entonces nuestro controlador parece estar haciendo su trabajo de guardar un libro cuando se le envía uno.

Hasta ahora, estas pruebas han asumido una aplicación no segura, muy parecida a la que escribimos en el capítulo 2. ¿Pero qué pasa si queremos probar una aplicación segura, como la del capítulo 3?

#### 4.2.2 Prueba de seguridad web

Spring Security ofrece soporte para probar aplicaciones web seguras fácilmente. Para aprovecharlo, debe agregar el módulo de prueba de Spring Security a su compilación. El seguimiento `testCompile` la dependencia en Gradle es todo lo que necesita:

```
testCompile ("org.springframework.security:spring-security-test")
```

O si está usando Maven, agregue lo siguiente <dependencia> a tu construcción:

```

<dependencia>
  <groupId> org.springframework.security </groupId>
  <artifactId> prueba-de-seguridad-primavera </artifactId>
  <scope> probar </scope>
</dependencia>

```

Con el módulo de prueba de Spring Security en la ruta de clase de su aplicación, solo necesita aplicar el configurador de Spring Security al crear el `MockMvc` ejemplo:

```

@Antes
configuración de vacío públicoMockMvc () {mockMvc
= MockMvcBuilders
    . webAppContextSetup (webContext)
    . aplicar (springSecurity ())
    . construir();
}

```

los `springSecurity ()` El método devuelve un configurador Mock MVC que habilita Spring Security para Mock MVC. Simplemente aplicándolo como se muestra aquí, Spring Security estará en juego en todas las solicitudes realizadas a través de MockMvc. La configuración de seguridad específica dependerá de cómo haya configurado Spring Security (o cómo Spring Boot haya configurado automáticamente Spring Security). En el caso de la aplicación de lista de lectura, es la misma configuración de seguridad que creamos en `SecurityConfig.java` en el capítulo 3.

EL MÉTODO `SPRINGSECURITY () springSecurity ()` es un método estático de `SecurityMockMvcConfigurers`, que he importado estáticamente por motivos de legibilidad.

Con Spring Security habilitado, ya no podemos simplemente solicitar la página de inicio y esperar una respuesta HTTP 200. Si la solicitud no está autenticada, deberíamos esperar una redirección a la página de inicio de sesión:

```

@Prueba
public void homePage_unauthenticatedUser () lanza Exception {mockMvc.perform (get ("/"))

    . andExpect (estado (). is3xxRedirection ())
    . andExpect (header (). string ("Ubicación",

                                                "http: // localhost / login"));
}

```

Pero, ¿cómo podemos realizar una solicitud autenticada? Spring Security ofrece dos anotaciones que pueden ayudar:

- **@ConMockUser** : Carga el contexto de seguridad con un Detalles de usuario utilizando el nombre de usuario, la contraseña y la autorización proporcionados
- **@ConDetallesDeUsuario** : Carga el contexto de seguridad buscando un Detalles de usuario objeto para el nombre de usuario dado

En ambos casos, el contexto de seguridad de Spring Security está cargado con un Detalles de usuario objeto que se utilizará durante la duración del método de prueba anotado. Los **@ WithMockUser**

La anotación es la más básica de las dos. Le permite declarar explícitamente un Detalles de usuario para cargarse en el contexto de seguridad:

```

@Prueba
@WithMockUser (nombre de usuario = "craig",
               contraseña = "contraseña",
               roles = "LECTOR")
public void homePage_authenticatedUser () lanza Exception {
    ...
}

```

Como se puede ver, `@ WithMockUser` omite la búsqueda normal de un Detalles de usuario objeto y en su lugar crea uno con los valores especificados. Para pruebas simples, esto puede estar bien. Pero para nuestra prueba, necesitamos un Lector ( que implementa Detalles de usuario) en lugar del genérico

Detalles de usuario ese `@ WithMockUser` crea. Para eso, necesitaremos `@ WithUserDetails`.

Los `@ WithUserDetails` anotación utiliza el configurado `UserDetailsService` a carga el Detalles de usuario objeto. Como recordará del capítulo 3, configuramos un `UserDetailsService` bean que mira hacia arriba y devuelve un Lector objeto para un nombre de usuario determinado. ¡Eso es perfecto! Así que anotaremos nuestro método de prueba con `@ WithUserDetails`, como se muestra en la siguiente lista.

#### Listado 4.4 Prueba de un método seguro con autenticación de usuario

```
@Prueba
@WithUserDetails ("craig")
public void homePage_authenticatedUser () lanza Exception {

    Lector esperado Lector = nuevo Lector ();
    esperabaReader.setUsername ("craig");
    esperabaReader.setPassword ("contraseña");
    esperabaReader.setFullname ("Muros de Craig");

    mockMvc.perform (obtener ("/")
        .andExpect (status ().isOk ())
        .andExpect (view ().name ("readingList"))
        .y Esperar (modelo (). atributo ("lector",
            samePropertyValuesAs (esperabaReader)))
        .andExpect (modelo (). atributo ("libros", hasSize (0))))

}
```

En el listado 4.4, usamos `@ WithUserDetails` para declarar que el usuario "craig" debe cargarse en el contexto de seguridad durante la duración de este método de prueba. Sabiendo que el Lector se colocará en el modelo, el método comienza creando un Lector objeto que pueda comparar con el modelo más adelante en la prueba. Luego realiza el OBTENER solicita y afirma el nombre de la vista y el contenido del modelo, incluido el atributo del modelo con el nombre "lector".

Una vez más, no se inicia ningún contenedor de servlets para ejecutar estas pruebas. Spring's Mock MVC ocupa el lugar de un contenedor de servlets real. El beneficio de este enfoque es que los métodos de prueba se ejecutan más rápido porque no tienen que esperar a que se inicie el servidor. Además, no es necesario iniciar un navegador web para publicar el formulario, por lo que la prueba es más simple y rápida.

Por otro lado, no es una prueba completa. Es mejor que simplemente llamar a los métodos del controlador directamente, pero realmente no ejercita la aplicación en un navegador web y verifica la vista renderizada. Para hacer eso, necesitaremos iniciar un servidor web real y usarlo con un navegador web real. Veamos cómo Spring Boot puede ayudarnos a iniciar un servidor web real para nuestras pruebas.



### 4.3 Probar una aplicación en ejecución

Cuando se trata de probar aplicaciones web, nada supera a la realidad. Iniciar la aplicación en un servidor real y usar un navegador web real es mucho más indicativo de cómo se comportará en manos de los usuarios que tocarla con un motor de prueba simulado.

Pero las pruebas reales en servidores reales con navegadores web reales pueden ser complicadas. Aunque existen complementos de tiempo de compilación para implementar aplicaciones en Tomcat o Jetty, su configuración es complicada. Además, es casi imposible ejecutar cualquiera de un conjunto de muchas pruebas de este tipo de forma aislada o sin iniciar su herramienta de compilación.

Spring Boot, sin embargo, tiene una solución. Debido a que Spring Boot ya admite la ejecución de contenedores de servlets integrados como Tomcat o Jetty como parte de la aplicación en ejecución, es lógico que se pueda usar el mismo mecanismo para iniciar la aplicación junto con su contenedor de servlets integrados durante la duración de una prueba.

Eso es exactamente lo que Spring Boot's `@WebIntegrationTest` la anotación lo hace. Anotando una clase de prueba con `@WebIntegrationTest`, declara que desea que Spring Boot no solo cree un contexto de aplicación para su prueba, sino que también inicie un contenedor de servlets incrustado. Una vez que la aplicación se ejecuta junto con el contenedor integrado, puede emitir solicitudes HTTP reales en su contra y hacer afirmaciones contra los resultados.

Por ejemplo, considere la prueba web simple en el listado 4.5, que usa `@WebIntegrationTest` para iniciar la aplicación junto con un servidor y usa Spring's `RestTemplate` para realizar solicitudes HTTP contra la aplicación.

Listado 4.5 Prueba de una aplicación web en el servidor

```
@RunWith (SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration (
    clases = ReadingListApplication.class)
@WebIntegrationTest
public class SimpleWebTest {

    @Test (esperado = HttpClientErrorException.class)
    public void pageNotFound () {
        tratar {
            RestTemplate rest = new RestTemplate (); rest.getForObject (

                "http: // localhost: 8080 / bogusPage", String.class);
            fail ("Debería resultar en HTTP 404");
        } catch (HttpClientErrorException e) {
            asertEquals (HttpStatus.NOT_FOUND, e.getStatusCode ());
            tirar e;
        }
    }
}
```

Prueba de ejecuciones en el servidor

Realiza GET petición

Afirma HTTP 404 No encontrado) respuesta

Aunque se trata de una prueba muy sencilla, demuestra suficientemente cómo utilizar el `@WebIntegrationTest` para iniciar la aplicación con un servidor. El servidor real que es

El inicio se determinará de la misma manera que lo estaría si estuviéramos ejecutando la aplicación en la línea de comandos. De forma predeterminada, inicia la escucha de Tomcat en el puerto 8080. Opcionalmente, sin embargo, podría iniciar Jetty o Undertow si alguno de ellos está en la ruta de clases.

El cuerpo del método de prueba se escribe asumiendo que la aplicación se está ejecutando y escuchando en el puerto 8080. Utiliza Spring's RestTemplate para realizar una solicitud de una página inexistente y afirma que la respuesta del servidor es un HTTP 404 (no encontrado). La prueba fallará si se devuelve cualquier otra respuesta.

#### 4.3.1 Iniciar el servidor en un puerto aleatorio

Como se mencionó, el comportamiento predeterminado es iniciar el servidor escuchando en el puerto 8080. Está bien para ejecutar una sola prueba a la vez en una máquina donde ningún otro servidor ya está escuchando en el puerto 8080. Pero si usted es como yo, usted ' probablemente he *siempre* tengo algo escuchando en el puerto 8080 en su máquina local. En ese caso, la prueba fallaría porque el servidor no se iniciaría debido a la colisión de puertos. Tiene que haber una mejor manera.

Afortunadamente, es bastante fácil pedirle a Spring Boot que inicie el servidor en un puerto seleccionado al azar. Una forma es configurar el Puerto de servicio propiedad a 0 para pedirle a Spring Boot que seleccione un puerto disponible aleatorio.

@ Prueba de integración web acepta una variedad de

Cuerda por su valor atributo. Se espera que cada entrada en la matriz sea un par de nombre / valor, en la forma nombre = valor, para establecer propiedades para su uso en la prueba. Para configurar Puerto de servicio

puedes usar @ Prueba de integración web como esto:

```
@WebIntegrationTest (value = {"server.port = 0"})
```

O, debido a que solo se establece una propiedad, puede tomar una forma más simple:

```
@WebIntegrationTest ("server.port = 0")
```

Configuración de propiedades a través del valor atributo es útil en el sentido general, pero @ Prueba de integración web también ofrece un randomPort atributo para una forma más expresiva de pedir al servidor que se inicie en un puerto aleatorio. Puede solicitar un puerto aleatorio configurando

randomPort a cierto:

```
@WebIntegrationTest (puerto aleatorio = verdadero)
```

Ahora que tenemos el servidor comenzando en un puerto aleatorio, debemos asegurarnos de usar el puerto correcto al realizar solicitudes web. Por el momento, el getForObject () El método está codificado con el puerto 8080 en su URL. Si el puerto se elige al azar, ¿cómo podemos construir la solicitud para usar el puerto correcto?

Primero, necesitaremos inyectar el puerto elegido como una variable de instancia. Para que esto sea conveniente, Spring Boot establece una propiedad con el nombre puerto.servidor.local al valor del puerto elegido. Todo lo que tenemos que hacer es usar Spring's @ Valor para inyectar esa propiedad:

```
@Value ("${local.server.port}")
puerto int privado;
```

Ahora que tenemos el puerto, solo necesitamos hacer un ligero cambio en el `getForObject()`

llamar para usarlo:

```
rest.getForObject (
    "http://localhost:{puerto}/bogusPage", String.class, puerto);
```

Aquí hemos cambiado el 8080 codificado por un `{ Puerto }` marcador de posición en la URL. Pasando el Puerto propiedad como el último parámetro en el `getForObject()` llamada, podemos estar seguros de que el marcador de posición se reemplazará con cualquier valor que se haya inyectado en Puerto.

#### 4.3.2 *Prueba de páginas HTML con Selenium*

RestTemplate está bien para solicitudes simples y es perfecto para probar puntos finales REST. Pero aunque se puede usar para realizar solicitudes contra URL que devuelven páginas HTML, no es muy conveniente para afirmar el contenido de la página o realizar operaciones en la propia página. En el mejor de los casos, podrá afirmar el contenido preciso del HTML resultante (lo que dará como resultado pruebas frágiles). Pero no podrá afirmar fácilmente el contenido seleccionado en la página o realizar operaciones como hacer clic en enlaces o enviar formularios.

Una mejor opción para probar aplicaciones HTML es Selenium ([www.seleniumhq.org](http://www.seleniumhq.org)).

Selenium hace más que solo realizar solicitudes y obtener los resultados para que usted los verifique. En realidad, Selenium enciende un navegador web y ejecuta su prueba dentro del contexto del navegador. Es lo más parecido posible a realizar las pruebas manualmente con sus propias manos. Pero a diferencia de las pruebas manuales, las pruebas de selenium son automatizadas y repetibles.

Para probar nuestra aplicación de lista de lectura usando Selenium, escribamos una prueba que obtenga la página de inicio, complete el formulario para un nuevo libro, publique el formulario y finalmente afirme que la página de destino incluye el libro recién agregado.

Primero, necesitaremos agregar Selenium a la compilación como una dependencia de prueba:

```
testCompile ("org.seleniumhq.selenium: selenium-java: 2.45.0")
```

Ahora podemos escribir la clase de prueba. La siguiente lista muestra una plantilla básica para una prueba de Selenium que usa Spring Boot's `@ WebIntegrationTest`.

#### Listado 4.6 Una plantilla para pruebas de selenium con Spring Boot

```
@RunWith (SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration (
    clases = ReadingListApplication.class)
@WebIntegrationTest (puerto aleatorio = verdadero)
ServerWebTests de clase pública {
```

```
    navegador privado estático FirefoxDriver;
```

```
    @Value ("${local.server.port}")
    puerto int privado;
```

Comienza en un  
puerto aleatorio

Inyecta  
el puerto

```

@Antes de clase
public static void openBrowser () {
    navegador = nuevo FirefoxDriver (); browser.manage ().
    timeouts ()
        . implicitamenteEspere (10, TimeUnit.SECONDS);
}

@Después de clases
public static void closeBrowser () {
    browser.quit ();
}
}

```

Configura Firefox conductor

Apaga navegador

Al igual que con la prueba web más simple que escribimos anteriormente, esta clase está anotada con `@Prueba` de integración web y conjuntos `randomPort` a cierto para que la aplicación se inicie y se ejecute con un servidor escuchando en un puerto aleatorio. Y, como antes, ese puerto se inyecta en el `Puerto` propiedad para que podamos usarla para construir URL para la aplicación en ejecución.

La estática `navegador abierto()` El método crea una nueva instancia de `FirefoxDriver`, que abrirá un navegador Firefox (deberá instalarse en la máquina que ejecuta la prueba). Cuando escribimos nuestro método de prueba, realizaremos operaciones del navegador a través del `FirefoxDriver` ejemplo. los `FirefoxDriver` también está configurado para esperar hasta 10 segundos cuando se buscan elementos en la página (en caso de que esos elementos tarden en cargarse).

Una vez finalizada la prueba, tendremos que apagar el navegador Firefox. Por lo tanto, `closeBrowser ()` llamadas `renunciar()` sobre el `FirefoxDriver` instancia para derribarlo.

ELIGE TU NAVEGADOR Aunque estamos probando con Firefox, Selenium también proporciona controladores para varios otros navegadores, incluidos Internet Explorer, Chrome de Google y Safari de Apple. No solo puede usar otros navegadores, sino que probablemente sea una buena idea escribir sus pruebas para usar todos y cada uno de los navegadores que desee admitir.

Ahora podemos escribir nuestro método de prueba. Como recordatorio, queremos cargar la página de inicio, completar y enviar el formulario, y luego afirmar que llegamos a una página que incluye nuestro libro recién agregado en la lista. La siguiente lista muestra cómo hacer esto con Selenium.

#### Listado 4.7 Prueba de la aplicación de lista de lectura con Selenium

```

@Prueba
public void addBookToEmptyList () {
    String baseUrl = "http: // localhost:" + puerto;

    browser.get (baseUrl);

    asertEquals ("No tienes libros en tu lista de libros",
        browser.findElementByTagName ("div"). getText ());

    browser.findElementByName ("titulo")
}

```

Obtiene el pagina de inicio

Afirma un vacío lista de libros

```

        .sendKeys("TÍTULO DEL LIBRO");
browser.findElementByName("autor")
        .sendKeys("AUTOR DEL LIBRO");
browser.findElementByName("isbn")
        .sendKeys("1234567890");
browser.findElementByName("descripcion")
        .sendKeys("DESCRIPCIÓN");
browser.findElementByTagName("formulario")
        .enviar();

WebElement dl =
    browser.findElementByCssSelector("dt.bookHeadline");
assertEquals("TÍTULO DEL LIBRO por AUTOR DEL LIBRO (ISBN: 1234567890)",
    dl.getText());

WebElement dt =
    browser.findElementByCssSelector("dd.bookDescription"); assertEquals("DESCRIPCIÓN", dt.getText());
}

```

← Rellena y envía formulario

← Afirma nuevo reservar en lista

Lo primero que hace el método de prueba es utilizar el `FirefoxDriver` para realizar un `OBTENER` solicitud de la página de inicio de la lista de lectura. Luego busca un `<div>` elemento en la página y afirma que su texto indica que no hay libros en la lista.

Las siguientes líneas buscan los campos en el formulario y usan el controlador `sendKeys()` método para simular eventos de pulsaciones de teclas en esos elementos de campo (esencialmente completando esos campos con los valores dados). Finalmente, busca el `<formulario>` elemento y lo envía.

Una vez que se procesa el envío del formulario, el navegador debe aterrizar en una página con el nuevo libro en la lista. Entonces, las últimas líneas buscan el `<dt>` y `<dd>` elementos en esa lista y afirman que contienen los datos que la prueba envió en el formulario.

Cuando ejecute esta prueba, verá que el navegador aparece y carga la aplicación de lista de lectura. Si prestas mucha atención, verás el formulario completado, como por un fantasma. Pero no es un espectro usar su aplicación, es la prueba.

Lo principal a notar sobre esta prueba es que @ Prueba de integración web fue capaz de iniciar la aplicación y el servidor para que Selenium pudiera comenzar a tocarlo con un navegador web. Pero lo que es especialmente interesante acerca de cómo funciona esto es que puede usar las instalaciones de prueba de su IDE para ejecutar tantas o tan pocas de estas pruebas como desee, sin tener que depender de algún complemento en la compilación de su aplicación para iniciar un servidor por usted.

Si la prueba con Selenium es algo que cree que le resultará útil, debería consultar *Selenium WebDriver en la práctica* por Yujun Liang y Alex Collins ( <http://manning.com/liang/> ), que entra en muchos más detalles sobre las pruebas con Selenium.

## 4.4 Resumen

Las pruebas son una parte importante del desarrollo de software de calidad. Sin un buen conjunto de pruebas, nunca sabrá con certeza si su aplicación está haciendo lo que se espera que haga.

Para las pruebas unitarias, que se centran en un solo componente o en un método de un componente, Spring no es realmente necesario. Los beneficios y las técnicas promovidas por Spring — suelta

el acoplamiento, la inyección de dependencias y el diseño basado en interfaces: facilitan la escritura de pruebas unitarias. Pero Spring no necesita estar involucrado directamente en las pruebas unitarias.

Sin embargo, la prueba de integración de múltiples componentes pide ayuda a Spring. De hecho, si Spring es responsable de conectar esos componentes en tiempo de ejecución, Spring también debería ser responsable de conectarlos en las pruebas de integración.

Spring Framework proporciona soporte para pruebas de integración en forma de un corredor de clase JUnit que carga un contexto de aplicación Spring y permite que los beans del contexto se inyecten en una prueba. Spring Boot se basa en el soporte de pruebas de integración de Spring con un cargador de configuración que carga el contexto de la aplicación de la misma manera que el propio Spring Boot, incluido el soporte para propiedades externalizadas y el registro de Spring Boot.

Spring Boot también permite las pruebas en el contenedor de aplicaciones web, lo que hace posible iniciar su aplicación para que sea servida por el mismo contenedor que será servida cuando se ejecute en producción. Esto le da a sus pruebas lo más parecido a un entorno del mundo real para verificar el comportamiento de la aplicación.

En este punto, hemos creado una aplicación bastante completa (aunque simple) que aprovecha los arrancadores de Spring Boot y la configuración automática para manejar el trabajo pesado para que podamos concentrarnos en escribir nuestra aplicación. Y también hemos visto cómo aprovechar el soporte de Spring Boot para probar la aplicación. En los próximos dos capítulos, tomaremos una tangente ligeramente diferente y exploraremos las formas en que Groovy puede hacer que el desarrollo de aplicaciones Spring Boot sea aún más fácil. Comenzaremos en el próximo capítulo observando algunas características del marco de Grails que se han abierto camino en Spring Boot.