

Spring Framework 4 Tutorial

Practical, Rapid, Intuitive

([Spring Framework for the Real World – Beginner to Expert](#) - Module II)

A beginner's rapid course for developing neat real world applications using the latest Spring projects in the right ways

Second Edition (Spring Boot 1.4), August 2016

Sanjay Patel

© Copyright 2016 [Sanjay Patel](#)

You are not authorized to read this book if you haven't downloaded it from [here](#) or received it directly from the author.

Contents

1	Introduction.....	9
	Why this course.....	9
	How to go through this course.....	9
	Prerequisite	10
	Help and support.....	10
	Meet the author	10
	Editions	10
2	Spring MVC: The Basics	11
	Source code so far	11
	Request Handlers	11
	DispatcherServlet.....	12
	@RequestMapping	12
	Sending The Response	12
	Rendering HTML	12
	Using JSP	13
	Source code so far	16
	@ResponseBody.....	16
	ViewController	16
	Customizing MVC Auto Configuration.....	17
	Source code so far	17
3	Adding Static Content.....	18
	Bootstrap	18
	Adding Bootstrap	19
	Bootstrapping home.jsp	19
	Adding Bootstrap widgets.....	20
	Source code so far	22

4	Some refactoring	23
	home.jsp	23
	header.jsp	23
	footer.jsp.....	25
	Source code so far	25
5	Sending data to views.....	26
	Preparing – trimming the HelloController.....	26
	Sending data to view	26
	Source code so far	27
6	Internationalization - using messages files.....	28
	Using messages files	28
	How Spring searches for a message	29
	Localizing messages in controllers	29
	Placeholders in messages	30
	Creating a getMessage utility method.....	30
	Source code so far	31
7	Receiving Request Data	32
	Java 8 parameter name discovery	33
	Optional parameters.....	33
	Source code so far	35
8	Displaying And Submitting Forms.....	36
	Handling form submission.....	38
	@RequestMapping at class level.....	40
	Linking /signup from UI	41
	Using Java beans to receive data.....	42
	Source code so far	42
9	Validation.....	43
	Using Hibernate Validator	43
	Using constraint annotations	43

Validating	44
Re-displaying the form	45
@ModelAttribute	47
Customizing error messages	47
Using the message attribute	48
Using DefaultMessageCodesResolver	50
So, which way to prefer?	51
Distinct error messages	51
Source code so far	52
10 Spring Data	53
Adding dependencies	53
Providing connection details	54
Mapping entity classes	54
Auto-creating the schema when application starts	56
Saving and retrieving data	56
Custom query methods	57
Saving users	57
User Roles	59
Source code so far	60
11 Flash Attributes	61
Showing an internationalized message	62
Coding a flash method	62
Source code so far	63
12 Custom Validation	64
Creating custom constraints	64
Extending LocalValidatorFactoryBean	68
Adding errors to BindingResult manually	68
Validation in service layer	68
Source code so far	68

13	Exception Handling	70
	More customization	71
	Source code so far	72
14	Transactions.....	73
	Transactional methods in Spring.....	73
	Transactional classes	75
	Source code so far	76
15	Running Code At Startup	77
	Using @PostConstruct	77
	Application events and listeners	78
	Source code so far	80
16	Spring Security	81
	Adding the dependency	81
	Auto-configured basic authentication	81
	Configuring Form Based Authentication.....	83
	Logging out	86
	Displaying login errors	87
	Mapping User entities to Spring Security users	88
	Getting the logged in user	92
	Source code so far	92
17	Security Tag Library	93
	Adding a Sign In link	93
	Using Spring Security tag library.....	93
	Displaying logged-in user's name	94
	Showing <i>Sign Up</i> and <i>Sign In</i> only when no one is logged in	94
	Displaying name only someone has logged in.....	95
	Source code so far	95
18	Authorization	97
	Request level authorization	97

	Method level authorization.....	99
	Source code so far	99
19	More Security	100
	Encrypting Passwords	100
	Remember me	102
	Signing in and out programmatically	103
	Source code so far	105
20	Email Verification.....	107
	Using the MockMailSender	107
	Adding a verificationCode field to the User class	107
	Updating signup service	108
	Mail subject and body	108
	The application URL	108
	sendVerificationMail.....	109
	Showing a unverified warning	110
	Verifying the user	110
	Resending verification mail.....	113
	Setting the permissions right	116
	Source code so far	117
21	Forgot password.....	118
	Forgot password button.....	119
	Forgot password form	119
	Forgot password GET handler	120
	Forgot password POST handler	121
	The service method	122
	Resetting the password.....	124
	Source code so far	130
22	Showing User Profile.....	131
	Coding the handler.....	131

	Coding the view	131
	Coding the service method	132
	Source code so far	134
23	Edit profile	135
	Edit Profile link	135
	Displaying the form	136
	Submitting the form	137
	A subtle validation problem	139
	Validation Groups.....	139
	Source code so far	141
24	AOP	142
	What is AOP	142
	Coding an Aspect.....	142
	Source code so far	144
25	Asynchronous Processing.....	145
	Sending Mails Asynchronously.....	145
	Source code so far	146
26	Scheduling	147
	Further Reading.....	147
27	Deployment.....	148
	Deployment Options.....	148
	Deploying to Pivotal Cloud Foundry	148
28	Next Steps.....	154

CONSIDERING SPRING FOR YOUR NEXT PROJECT?

We can provide all kinds of help you may need, starting from individual mentoring to full solutions. [Click here](#) for more details, and feel free to contact us by mailing to info@naturalprogrammer.com.

1 Introduction

Welcome to this rapid course on developing neat real world applications using the latest Spring projects in the right ways. This is the *Module II* of our [Spring Framework for the Real World – Beginner to Expert](#) course.

If you are new to Spring Framework and want to rapidly get up and running developing real world applications with it, this is a perfect course for you. Even if you know earlier versions of Spring, if you are new to Spring 4 and Spring Boot, this course is a perfect fit for you.

Why this course

"With so many courses on Spring existing around, why another?" You may ask.

Most existing Spring courses try to cover all Spring topics with equal pace and importance. But, Spring now has become too vast to learn in that flat manner. So, this course takes a different route. Here, we will uncover the common features of Spring in an intuitive manner, step-by-step, by taking a problem centered approach. This will keep your interest up throughout. In the process, you will learn most of the essential real-world features, and you will feel confident to explore the uncommon features on your own, when needed. This is the quickest way towards mastering a vast subject like Spring, we believe.

Also, in this course, we will be developing a real-world user module with features like signup, login, email verification, forgot password, show profile and edit profile, which can be reused in your projects, saving hours of your effort and letting you jump to coding your business functionality quickly.

For more details about this course, visit its [landing page](#).

How to go through this course

This is a complete hands on course, and after walking through it you can think and code like an experienced Spring developer. So, to get the real benefit out of it, don't just read it. Instead, remember to walk through all its the hands on exercises patiently. The exercises are very short and to the point.

Prerequisite

1. You should already be knowing the basics of *web development*, *http* and *java servlets*. If you are new to these, you can first go through our whiteboard video tutorial [Basic Concepts of Web Development, HTTP and Java Servlets](#).
2. You also should be familiar with HTML, CSS, JSP, JSTL and EL. There are numerous books and tutorials available on these.
3. This book does not start from the beginning, but instead is *Module II* of our [Spring Framework for the Real World – Beginner to Expert](#) course. So, you should already have gone through *Module I*, which is the [Spring Framework 4 Getting Started And Dependency Injection For Beginners](#) book. In fact, we have been developing a demo application there, which we are going to resume in this course. If you don't already have that, [checkout](#) or [download](#) that before continuing.

Help and support

1. Community help is available at [stackoverflow.com](#), under the **np-spring** tag. Do not forget to tag your question with **np-spring**, otherwise we'll miss it!
2. For any bug, submit an issue [here](#). But please check first that the issue isn't already reported.
3. Mentoring, training and professional help is provided by [naturalprogrammer.com](#).

Meet the author

Sanjay has about 22 years of programming and leading experience. Since 2009, he is working on the Java and Spring Framework stack full time, and is the lead developer of *Spring Lemon*.

Presently he is working as the principal technical lead of Bridgeton Research, Inc.. Prior to joining Bridgeton, he was the technical director of RAD Solutions Private Limited, doing research on open source tools, frameworks, patterns and methodologies for rapid application development. Previously, he was a project leader at Cambridge Solutions and an assistant manager at L & T Limited. He is an MCA from Osmania University and a B. Sc. (Physics) from Sambalpur University, India.

He is also an experienced teacher, with about 20K students enrolled in his video tutorials and books at [Udemy](#), [YouTube](#) and [Gumroad](#).

[Click here](#) to know more about him.

Editions

First Edition - May 2016

2 Spring MVC: The Basics

In this chapter, we will learn the basics of Spring MVC.

Let's begin by trying to understand our [HelloController](#), which was coded in *Module 1*, i.e. the [Spring Framework 4 Getting Started And Dependency Injection For Beginners](#) course. If you haven't gone through that, go through that first.

Source code so far

- [Browse online](#)
- [Download zip](#)

Request Handlers

Our *HelloController* looks as below:

```
package com.naturalprogrammer.spring.tutorial.controllers;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @Value("${app.name}")
    private String appName;

    @RequestMapping("/hello")
    public String hello() {
        return "Hello " + appName;
    }
}
```

Notice that it's annotated with `@RestController`, and its *hello* method is annotated with `@RequestMapping`. The methods that are annotated with `@RequestMapping` inside `@Controller` or `@RestController` classes, such as the *hello* method above, are known as *request handlers*, or simply *handlers*.

When a web request comes, one of these handlers are used for sending the response.

DispatcherServlet

For receiving the requests and guiding those to the handlers, Spring MVC uses a servlet named [DispatcherServlet](#). In other words, when a request comes, *DispatcherServlet* chooses one of the handlers and then calls that.

@RequestMapping

Handlers are chosen by *DispatcherServlet* based on the *@RequestMapping* parameters. For example, in the `hello()` handler above, the `"/hello"` argument tells the *DispatcherServlet* that *all the requests with URL /hello be given to this method*.

@RequestMapping can take many other kinds of parameters, e.g. *the HTTP method*, which help narrowing down to the correct handler when a request comes. We are going to cover some of these later; but for a detailed reference, refer the [documentation](#).

Beginning with Spring 4.3 (Spring Boot 1.4), we also have some specializations of *@RequestMapping*, e.g. *@GetMapping*, *@PostMapping*, which we'll discuss in later chapters.

Sending The Response

In our *hello()* handler, we have simply returned a string, which becomes the response body. That's the default behavior of handlers inside *@RestController* classes – the return value is straightly written to the response.

Handler methods can also return objects or maps, which are converted to some appropriate format before writing to the response. [Converters](#) are used for this, and when using Spring Boot, a [JSON Converter](#) is configured by default.

That means, if we return an object or a map from the *hello()* method, the response body will be in JSON format. In a Spring application, *JSON web services* are coded in this way.

Rendering HTML

Responding with Strings or JSON objects work well for web services. But, when developing traditional web applications, we often need to render HTML pages. In such cases, a view technology, like JSP or Thymeleaf, is normally used.

Spring supports many view technologies. JSP is the most popular among those, whereas Thymeleaf is a new one, which seems to be technically better than JSP in many ways. In fact, the Spring team recommends Thymeleaf over JSP. But, in this tutorial, we'll use JSP.

JSP VS THYMELEAF

Although Thymeleaf is technically superior to JSP, we preferred to stick to JSP. Here are a couple of reasons.

First, the world is gradually moving towards isomorphic or rich JavaScript clients, such as AngularJS or mobile clients, backed by APIs in the server side. Pure server side templating seems to be fading away gradually. So, we feel that spending your time in learning AngularJS or React will be more important than learning Thymeleaf, which that may never get widely adopted.

Secondly, JSP is already familiar to most and is well adopted by the enterprise world. In your company, the chances of you working on a JSP project is much more than that of Thymeleaf. Visit a job site and you'll find that JSP is much more in demand than Thymeleaf.

Using JSP

So, let's see how to use JSP views.

Getting prepared

To let our application use JSP, do the following:

1. There are some [limitations](#) when using JSP in a Spring Boot application. Hence, in *pom.xml*, change the packaging from `jar` to `war`.

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

  <groupId>com.naturalprogrammer.spring</groupId>
  <artifactId>spring-tutorial</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jarwar</packaging>

  <name>np-spring-tutorial</name>
  <description>Demo project for Spring Boot</description>
  ...
```

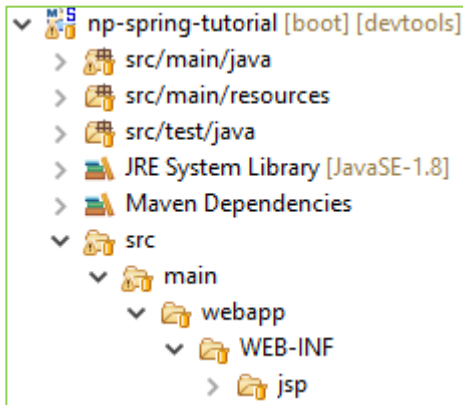
2. Add the following dependencies to *pom.xml*:

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
  <scope>provided</scope>
</dependency>
```

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
</dependency>
```

The first one is for JSP, whereas the second one is for JSTL.

3. JSP files should be kept inside `src/main/webapp/WEB-INF` folder. We prefer to keep those in a `jsp` folder there. So, create a folder hierarchy up to `src/main/webapp/WEB-INF/jsp`, looking as below:



Hello, JSP

Let's now code a "Hello, JSP" page.

Creating the JSP page

In the just created `src/main/webapp/WEB-INF/jsp` folder, create a `home.jsp`, looking as below:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Spring Framework Tutorial For Beginners</title>
</head>
<body>
  Hello, JSP!
</body>
</html>
```

Assuming you already know JSP, the above code needs no explanation.

Coding the handler

Next, let's code a handler method, where we'll use the JSP for rendering the response. It can be coded either in an existing controller class or a new one. Let's create a new class named `HomeController`, say in our `.controllers` package, looking as below:

```
package com.naturalprogrammer.spring.tutorial.controllers;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HomeController {

    @RequestMapping("/")
    public String home() {
        return "home";
    }
}
```

Note that we have used `@Controller` instead of `@RestController`. Consequently, the return value of its handler methods, e.g. `"home"`, will NOT be written to the response. Instead, it'll just help Spring find a view, which the request will be forwarded to. That view will then render the response.

Resolving views

To find out the correct view, Spring uses components called [ViewResolvers](#).

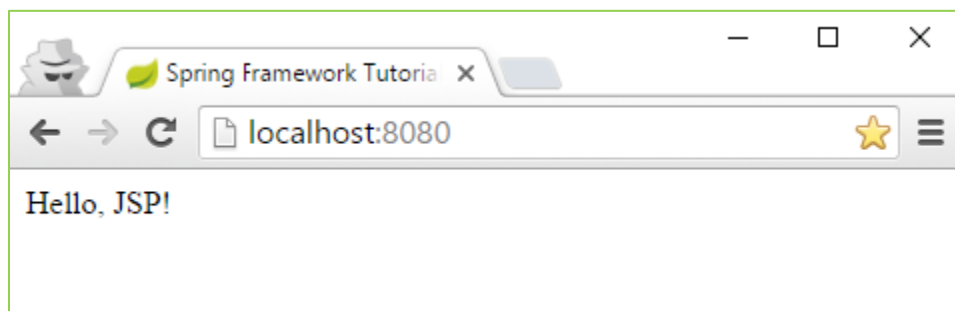
In a Spring Boot application, an `InternalResourceViewResolver` gets auto configured [by default](#). The `InternalResourceViewResolver` treats the return value of a handler, e.g. `"home"` in our case, as the path of the view file w.r.t. `src/main/webapp`. It also looks for a *prefix* and a *suffix* properties, as below:

```
spring.mvc.view.prefix: /WEB-INF/jsp/
spring.mvc.view.suffix: .jsp
```

If found, the prefix and suffix get concatenated with the return value to formulate the path.

So, add the above lines in your `application.properties`, and then your `homp.jsp` will be correctly found by Spring.

Try running the application and visit <http://localhost:8080>. You should now see "Hello JSP" as below:



Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

@ResponseBody

When you have multiple handler methods in a `@Controller` class, and you want just a few of those to write directly to the response instead of forwarding to views, annotate those methods with `@ResponseBody`. In other words, handlers annotated with `@ResponseBody` inside `@Controller` classes are equivalent to the handlers of `@RestController` classes.

For example, try annotating the `home()` method with `@ResponseBody`, as below:

```
...
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HomeController {

    @RequestMapping("/")
    @ResponseBody
    public String home() {
        return "home";
    }
}
```

If you now run the application now and visit <http://localhost:8080/>, you should see just "home."

Remove the `@ResponseBody` before we proceed.

ViewController

Notice that our `home()` handler doesn't do anything except just forwarding to the `home` view:

```
@Controller
public class HomeController {

    @RequestMapping("/")
    public String home() {
        return "home";
    }
}
```

In such cases, instead of coding the handler, you can just use [ViewControllers](#). ViewControllers should be added to a `ViewControllerRegistry` that Spring Boot auto configures along with other MVC components. Let's discuss how exactly to do it.

Customizing MVC Auto Configuration

As we know, Spring Boot auto configures various components when an application starts.

To customize the auto configuration of MVC components, we can configure beans extending from `WebMvcConfigurerAdapter`. For example, to add ViewControllers to the `ViewControllerRegistry`, have a `@Configuration` class as below, say in a new `com.naturalprogrammer.spring.tutorial.config` package:

```
package com.naturalprogrammer.spring.tutorial.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }
}
```

The above class overrides the `addViewControllers` method of `WebMvcConfigurerAdapter` and adds a `ViewController`. [WebMvcConfigurerAdapter](#) has many such overridable methods.

`HomeController` is no more needed – remove it. Then, run the application and visit <http://localhost:8080/> - you'll see "Hello, JSP."

More about configuring Spring MVC can be found [here](#).

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

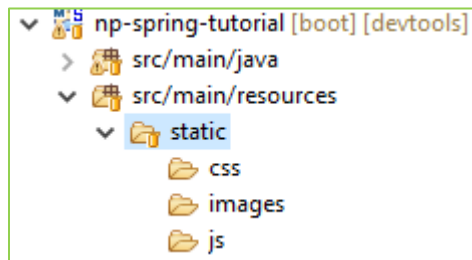
3 Adding Static Content

Static content is that which does not change, such as the CSS, images or JavaScript files.

By default, a Spring Boot application serves static content from `/static`, `/public`, `/resources` or `/META-INF/resources` folders in the classpath. So, create a `static` folder inside `src/main/resources`. That would be a good place to put our static resources.

To know more and change the default behavior, such as the file locations or cache settings, refer [Spring Boot documentation](#).

Then, create three sub-folders in that, named `css`, `js`, and `images`. We shall be putting our static content in those. Your folder structure now should look as below:



We chose the names `css`, `js` and `images`, because, by default Spring Security will make URLs of pattern `/css/**`, `/js/**`, `/images/**` and `/**/favicon.ico` as public. Of course we can override the defaults, but why deviate?

Bootstrap

[Bootstrap](#) is a popular front-end framework for developing good looking html pages. Using it, professional looking web-pages can be developed very easily. You just have to know about the widgets it provides, and from its documentation, copy the code snippets into your html or JSP pages and edit those. It has become so popular today that you'll find it in the job description of almost every web project. So, we thought to include it in this course.

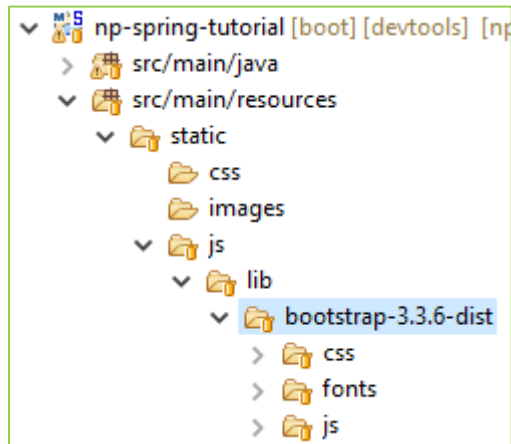
But, isn't the new [Material Design](#) by google a *Bootstrap* killer? You may ask. Looking at Bootstrap's omnipresence and ecosystem, we don't think so.

Let's add it to our application.

Adding Bootstrap

There are [many ways](#) to add Bootstrap to our application. Let's follow a simple one – just download the latest stable distribution. It'll be a zip file, named like *bootstrap-3.3.6-dist*. Unzip that. You can put the unzipped folder straight into the *static/js* folder, but let's create a *lib* folder inside that, where we'll keep all such downloaded libraries.

So, put the unzipped bootstrap folder inside *lib*. Your folder structure should now look as below:



Bootstrapping home.jsp

Time to bootstrap our *home.jsp*. To do so, we can refer to the [getting started template](#) of Bootstrap. Precisely, we need to add the links to Bootstrap and related css and js files. After doing so, our *home.jsp* should now look as below:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Spring Framework Tutorial For Beginners</title>

  <!-- Bootstrap -->
  <link href="/js/lib/bootstrap-3.3.6-dist/css/bootstrap.min.css" rel="stylesheet">

  <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries -->
  <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
  <!--[if lt IE 9]>
    <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
    <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
  <![endif]>
</head>
<body>
  Hello, JSP!
```

```

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
<!-- Include all compiled plugins (below), or include individual files as needed -->
<script src="/js/lib/bootstrap-3.3.6-dist/js/bootstrap.min.js"></script>
</body>
</html>

```

Try running the code, and you may not be able to notice much difference, because we haven't added any bootstrap widgets on the page yet.

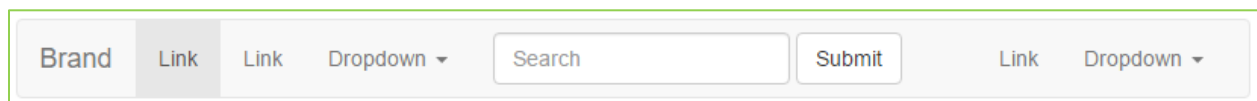
Adding Bootstrap widgets

Bootstrap widgets are html snippets that can be pasted on your page and changed as per your needs. Those are divided into three groups: [CSS](#), [Components](#) and [JavaScript](#). For example, visit [CSS](#), and you'll find snippets for *Tables*, *Forms*, *Buttons* etc.

Bootstrap Grid System

Bootstrap also includes a responsive, mobile first fluid grid system to layout your pages beautifully. Read its [documentation](#) to know more about it.

So, let's now add a *navigation bar* to our application. Navigation bars typically stick to the top of each page in a site, looking as below:



The snippet for Bootstrap's navigation bar widget can be found [here](#) in its components page. Paste that snippet just below the `<body>` tag in *home.jsp*:

```

...

<body>
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <!-- Brand and toggle get grouped for better mobile display -->
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed" data-
toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-
expanded="false">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="#">Brand</a>
      </div>

      <!-- Collect the nav links, forms, and other content for toggling --

```

```

<div class="collapse navbar-collapse" id="bs-example-navbar-
collapse-1">
  <ul class="nav navbar-nav">
    <li class="active"><a href="#">Link <span class="sr-
only">(current)</span></a></li>
    <li><a href="#">Link</a></li>
    <li class="dropdown">
      <a href="#" class="dropdown-toggle" data-toggle="dropdown"
role="button" aria-haspopup="true" aria-expanded="false">Dropdown <span
class="caret"></span></a>
      <ul class="dropdown-menu">
        <li><a href="#">Action</a></li>
        <li><a href="#">Another action</a></li>
        <li><a href="#">Something else here</a></li>
        <li role="separator" class="divider"></li>
        <li><a href="#">Separated link</a></li>
        <li role="separator" class="divider"></li>
        <li><a href="#">One more separated link</a></li>
      </ul>
    </li>
  </ul>
  <form class="navbar-form navbar-left" role="search">
    <div class="form-group">
      <input type="text" class="form-control" placeholder="Search">
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
  </form>
  <ul class="nav navbar-nav navbar-right">
    <li><a href="#">Link</a></li>
    <li class="dropdown">
      <a href="#" class="dropdown-toggle" data-toggle="dropdown"
role="button" aria-haspopup="true" aria-expanded="false">Dropdown <span
class="caret"></span></a>
      <ul class="dropdown-menu">
        <li><a href="#">Action</a></li>
        <li><a href="#">Another action</a></li>
        <li><a href="#">Something else here</a></li>
        <li role="separator" class="divider"></li>
        <li><a href="#">Separated link</a></li>
      </ul>
    </li>
  </ul>
</div><!-- /.navbar-collapse -->
</div><!-- /.container-fluid -->
</nav>

Hello, JSP!

...

```

Next, to center our content, let's also use a [container](#). So, enclose the entire visible body of *home.jsp* inside a container snippet, as below:

...

```

<body>
<div class="container">
  <nav class="navbar navbar-default">
    ...
  </nav>

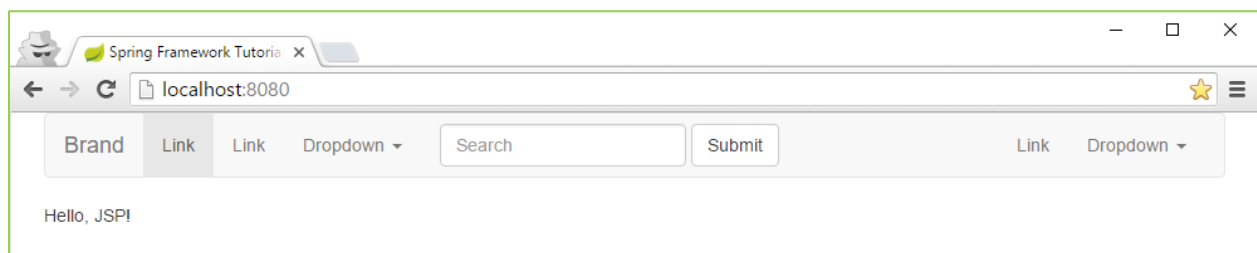
  Hello, JSP!
</div>

  <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
  ...

</body>

```

Run the application now, and you'll now see a much beautiful home page, just as below:



Better ways to add front-end libraries

We added Bootstrap manually. But, when coding large scale applications, you may like to use [WebJars](#). Even better would be using some JavaScript builds tools like [Gulp](#) or [Grunt](#) along with [npm](#) or [bower](#).

Boggled? There's an easy way to begin – use Yeoman generators! For example, for my AngularJS applications, I use [this generator](#), and for Angular 2, I plan to use [this one](#).

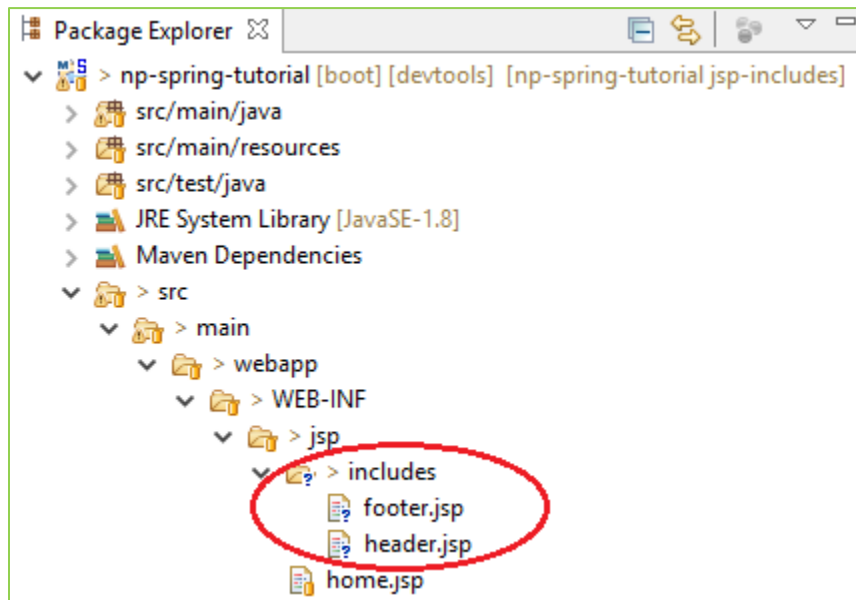
Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

4 Some refactoring

The navigation bar that we have in our home page should actually be visible on all pages that we are going to create in due course. So, let's move its code, and in fact all the common code, to separate JSP files, so that we can just include those in all our pages.

Specifically, let's create a `header.jsp` and a `footer.jsp`, and move the common code out to those. Let's place the files in a new `includes` sub-folder. The file structure should now look as below:



home.jsp

`home.jsp` should now look as below:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@include file="includes/header.jsp"%>

Hello, JSP!

<%@include file="includes/footer.jsp"%>
```

header.jsp

`header.jsp` should now look as below, which is nothing but a cut-paste from `home.jsp`:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
```

```

<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Spring Framework Tutorial For Beginners</title>
  <!-- Bootstrap -->
  <link href="/js/lib/bootstrap-3.3.6-dist/css/bootstrap.min.css" rel="stylesheet">

  <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries -->
  <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
  <!--[if lt IE 9]>
    <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
    <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
  <![endif]-->
</head>
</head>
<body>
<div class="container">
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <!-- Brand and toggle get grouped for better mobile display -->
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-
target="#bs-example-navbar-collapse-1" aria-expanded="false">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="#">Brand</a>
      </div>

      <!-- Collect the nav links, forms, and other content for toggling -->
      <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
        <ul class="nav navbar-nav">
          <li class="active"><a href="#">Link <span class="sr-
only">(current)</span></a></li>
          <li><a href="#">Link</a></li>
          <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-
haspopup="true" aria-expanded="false">Dropdown <span class="caret"></span></a>
            <ul class="dropdown-menu">
              <li><a href="#">Action</a></li>
              <li><a href="#">Another action</a></li>
              <li><a href="#">Something else here</a></li>
              <li role="separator" class="divider"></li>
              <li><a href="#">Separated link</a></li>
              <li role="separator" class="divider"></li>
              <li><a href="#">One more separated link</a></li>
            </ul>
          </li>
        </ul>
        <form class="navbar-form navbar-left" role="search">
          <div class="form-group">
            <input type="text" class="form-control" placeholder="Search">
          </div>
          <button type="submit" class="btn btn-default">Submit</button>
        </form>
        <ul class="nav navbar-nav navbar-right">
          <li><a href="#">Link</a></li>
          <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-
haspopup="true" aria-expanded="false">Dropdown <span class="caret"></span></a>
            <ul class="dropdown-menu">
              <li><a href="#">Action</a></li>
              <li><a href="#">Another action</a></li>
              <li><a href="#">Something else here</a></li>
              <li role="separator" class="divider"></li>
              <li><a href="#">Separated link</a></li>
            </ul>
          </li>
        </ul>
      </div>
    </div>
  </nav>

```



```
        </li>
      </ul>
    </div><!-- /.navbar-collapse -->
  </div><!-- /.container-fluid -->
</nav>
```

footer.jsp

footer.jsp will be another cut-paste from *home.jsp*:

```
</div>

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
<!-- Include all compiled plugins (below), or include individual files as needed -->
<script src="/js/lib/bootstrap-3.3.6-dist/js/bootstrap.min.js"></script>
</body>
</html>
```

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

5 Sending data to views

Although we now know how to forward to a JSP from a request handler method, we are yet to see how to pass data to it. That's what we are going to look at in this chapter.

Preparing – trimming the HelloController

Let's use the existing HelloController for the demo. So, first trim the HelloController as below:

```
@RestController
public class HelloController {

    @Value("${app.name}")
    private String appName;

    @RequestMapping("/hello")
    public String hello() {
        return "hello";
    }
    return "Hello " + appName;
}
```

The above is now not a `@RestController`, but a `@Controller`, just like our HomeController. As you see, its `hello()` handler will now forward to the "hello" view.

But, we don't have a `hello.jsp` yet. So, create one, just by copying the existing `home.jsp`:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@include file="includes/header.jsp"%>

Hello, JSP!

<%@include file="includes/footer.jsp"%>
```

If you did everything correctly, running the application and visiting <http://localhost:8080/hello> should now show "Hello, JSP."

Sending data to view

Data can be sent from a handler method to a view by using Spring's `Model` object. Specifically, you can

1. in a handler method, add some data into the Model object
2. in the JSP view, refer that using EL.

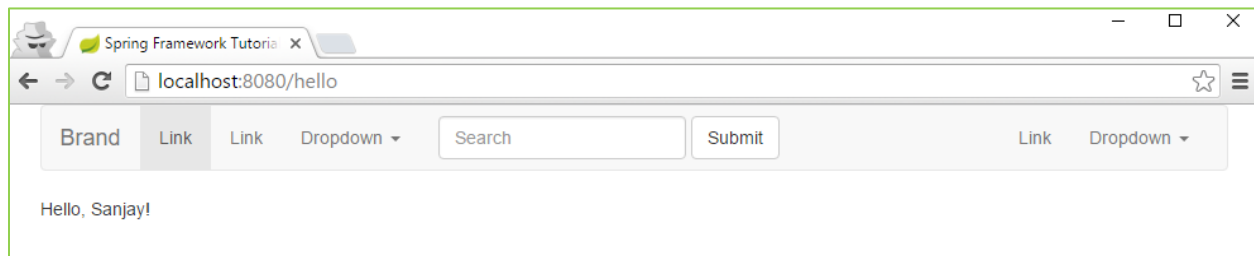
Model can be injected into the controller method as a parameter, and then data can be added to it as attributes. For example, change the hello() handler as below:

```
@RequestMapping("/hello")
public String hello(Model model) {
    model.addAttribute("name", "Sanjay");
    return "hello";
}
```

The above code adds to the Model an attribute having *name* as "name" and *value* as "Sanjay." That can then be accessed in *hello.jsp* using EL, as below:

Hello, JSP `${name}`!

Try running it now, and you should see "Hello, Sanjay!", as below:



To access the Model object, rather than injecting the Model as a parameter to the handler, we could also have used Spring's [ModelAndView](#), as below:

```
@RequestMapping("/hello")
public ModelAndView hello() {
    ModelAndView mav = new ModelAndView("hello"); // the logical view name
    mav.addObject("name", "Sanjay");
    return mav;
}
```

Whether you use *Model* or *ModelAndView* is just a matter of style. Using *Model*, as we did earlier, seems to be the current trend, though.

We'll see a few more ways to put data in the Model object in the coming lessons. Also, as you'll gain some experience, you'd like to know where exactly the attributes of the model objects are stored. They are initially stored in a Java Map and then, after the controller method finishes executing, but before the view takes over, the attributes are copied to the HTTP request object by default, or session object if you tell to do so. [Here](#) is an interesting article about it.

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

6 Internationalization - using messages files

The “Hello, Sanjay” message that our application shows when a user visits `/hello` is a hardcoded one. “Hello” is hardcoded in the JSP, whereas “Sanjay” is hardcoded in the controller.

But ideally, we should put such messages in localized properties files. That enables our application to support internationalization.

Using messages files

So, let's try to move “Hello” in `hello.jsp` to a properties file. Do the following steps for that:

1. By default, Spring Boot expects the properties files to be named as `messages*.properties`, available at the classpath root. So, create a `messages.properties` file in `src/main/resources`, and add the following line to that:

```
hello: Hi
```

2. Next, change the message in `hello.jsp` as below:

```
Hello<spring:message code="hello" />, ${name}!
```

3. The above line uses the `message` tag of [Spring tag library](#). To use Spring tag library, add the following directive at the top of your `header.jsp`:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<!DOCTYPE html>
...
```

Now, if you run the application and visit `/hello`, you'll see “Hi, Sanjay.” The “Hi” part is pulled from the messages file.

If you want to support some locale, say `fr`, you can also have a `messages_fr.properties` file, which may look as below:

```
hello: Salut
```

When a French user will visits `/hello` (technically, when Spring will conclude that `fr` locale should be used, e.g. by looking at the `Accept-Language` request header from the browser), “Salut” will then be shown rather than “Hi”.

How Spring searches for a message

When Spring searches for a message for a particular locale, it starts with the most specific *messages*.properties* file and then falls back to more generic ones. For example, if the selected locale is `en_US`, Spring will first try to find a message in *messages_en_US.properties* file. If not found, a *messages_en.properties* would be referred to, and *messages.properties* would be referred to at last.

Localizing messages in controllers

Localizing the message in JSPs was easy. We just had to use the `<spring:message>` tag.

Let's now see how to localize messages in controllers, e.g. how to localize "Sanjay" in the `HelloController`.

For fetching the messages, Spring uses a `MessageSource` bean, which is auto-configured by Spring Boot. We can use the same bean to fetch a message manually. To see it working, update our `HelloController` as below:

```
@Controller
public class HelloController {

    @Autowired
    private MessageSource messageSource;

    @RequestMapping("/hello")
    public String hello(Model model) {

        model.addAttribute("name",
            messageSource.getMessage("name", null, LocaleContextHolder.getLocale()));

        return "hello";
    }
}
```

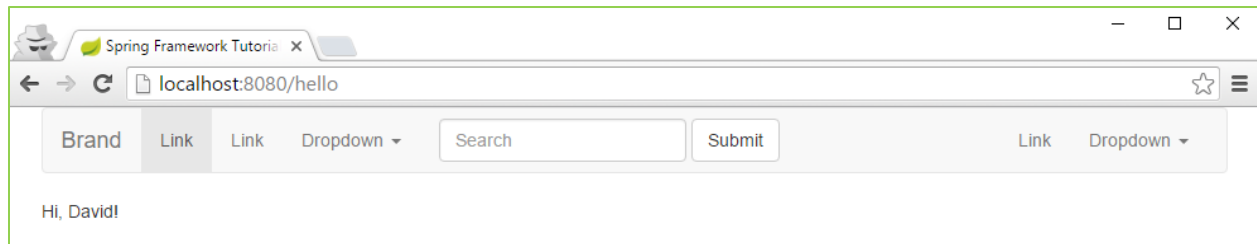
Note that we are injecting the `messageSource` bean, and then using its `getMessage` method to fetch a message.

The above `getMessage` method takes three parameters. The first one is the message code. We'll discuss about the second one a bit later. The third one is the locale. We have passed `LocaleContextHolder.getLocale()` there, which returns the locale associated with the current request, if any, or else the server's locale.

Next, add a message in our *messages.properties* file, as below:

```
hello: Hi
name: David
```

Now, visit `/hello`, and you should see “Hi, David”:



Placeholders in messages

Messages can have placeholders, as below:

```
mailBody: Visit {0} if you are above 18, or else visit {1}
```

The placeholders are filled using `getMessage`'s second parameter, which will be an array of objects. For example, the following code snippet can be used for getting a filled `mailBody` message:

```
Object[] urls = {"http://above18.example.com",
                 "http://below18.example.com"};

String mailBody = messageSource.getMessage("mailBody", urls,
                                           LocaleContextHolder.getLocale());
```

Creating a getMessage utility method

So, in every class that we plan to fetch a message, do we need to inject the `MessageSource` and call the verbose `getMessage`? Can't a simple utility method be coded for this?

Why not? Code a `MyUtil` class, say in a new `com.naturalprogrammer.spring.tutorial.util` package, looking as below:

```
package com.naturalprogrammer.spring.tutorial.util;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.context.i18n.LocaleContextHolder;
import org.springframework.stereotype.Component;

@Component
public class MyUtil {

    private static MessageSource messageSource;

    @Autowired
    public MyUtil(MessageSource messageSource) {
        MyUtil.messageSource = messageSource;
    }
}
```

```

    public static String getMessage(String messageKey, Object... args) {

        return messageSource.getMessage(messageKey, args,
            LocaleContextHolder.getLocale());
    }
}

```

Note that the class is annotated with `@Component`, and its constructor is annotated with `@Autowired`. Consequently, the auto-configured `MessageSource` will be injected into the constructor, which is then stored in a static `messageSource` variable. We have a static `getMessage` method, which uses that `messageSource` to fetch the message.

Note that the signature of our `getMessage` is simplified – it takes a variable list of arguments instead of an array, and we don't pass the locale. A call to it can look as below:

```

String mailBody = MyUtil.getMessage(
    "mailBody",
    "http://above18.example.com",
    "http://below18.example.com");

```

Before proceeding, let's update `HelloController` to use our `getMessage`, as below:

```

@Controller
public class HelloController {

    @Autowired
    private MessageSource messageSource;

    @RequestMapping("/hello")
    public String hello(Model model) {

        model.addAttribute("name", MyUtil.getMessage("name"));
        messageSource.getMessage("name", null,
            LocaleContextHolder.getLocale()));

        return "hello";
    }
}

```

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

7 Receiving Request Data

Commonly, data is sent in web requests by using *path variables* and *request parameters*. For example, see the URL below:

```
http://localhost:8080/hello/5?name=Sanjay
```

The `5` above, which is a part of the URL, is called a *path variable*. The parameters following the `"?"`, i.e. `name=Sanjay` is called a *request parameter*.

A request handler method can receive these by using `@PathVariable` and `@RequestParam` annotations. For example, let's change our hello handler to receive these:

```
...  
  
import org.springframework.web.bind.annotation.PathVariable;  
import org.springframework.web.bind.annotation.RequestParam;  
  
...  
  
@RequestMapping("/hello/{id}")  
public String hello(Model model,  
    @PathVariable("id") String id,  
    @RequestParam("name") String name) {  
  
    model.addAttribute("id", id);  
    model.addAttribute("name", name);  
  
    return "hello";  
}
```

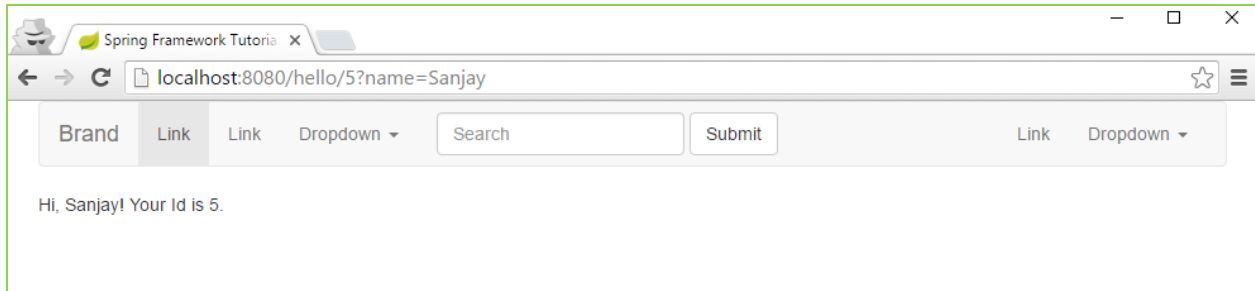
In the URL in `@RequestMapping`, note how we have used `{id}` as the placeholder. That corresponds to the `@PathVariable`'s `"id"` argument. Similarly, the `name` request parameter in the URL will correspond to the `"name"` argument of `@RequestParam`.

This way, we are receiving the request data as arguments to the handler method. Inside the method, we are adding those to the *model*.

Next, change the message in `hello.jsp` as below:

```
<spring:message code="hello" />, ${name}! Your Id is ${id}.
```

Now, run the application, and visit <http://localhost:8080/hello/5?name=Sanjay>. You'll see "Hi, Sanjay! Your Id is 5," as below.



Java 8 parameter name discovery

If we compile our code with the `-parameters` flag, Java 8 preserves the method parameter names in the JAR. Spring defaults to those if we don't pass arguments to `@PathVariable` and `@RequestParam`. That means, we can shorten our code as below:

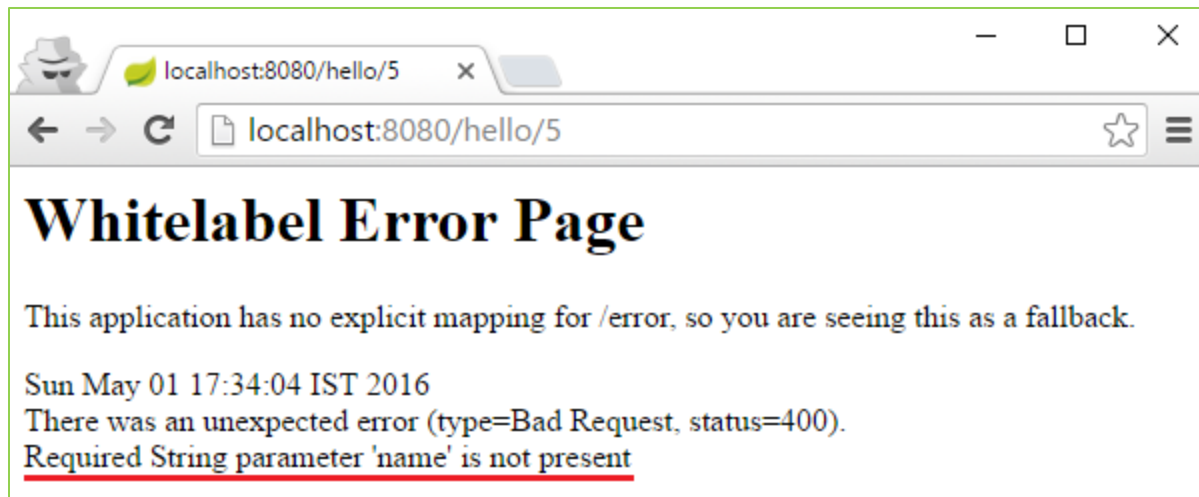
```
@PathVariable("id") String id,  
@RequestParam("name") String name) {
```

For this to work properly all the times, configure the `maven-compiler-plugin` with `-parameters` flag in your `pom.xml`. In other words, add the `maven-compiler-plugin` in the plugins section, configured as below:

```
<build>  
  <plugins>  
    <plugin>  
      <groupId>org.apache.maven.plugins</groupId>  
      <artifactId>maven-compiler-plugin</artifactId>  
      <configuration>  
        <compilerArgs>  
          <arg>-parameters</arg>  
        </compilerArgs>  
      </configuration>  
    </plugin>  
    ...  
  </plugins>  
</build>
```

Optional parameters

If you visit the URL <http://localhost:8080/hello/5>, leaving out the `name`, Spring will report an error, as below:



If you want to make *name* optional, you can use `@RequestParam(required = false)`, as below:

```
@RequestMapping("/hello/{id}")
public String hello(Model model,
    @PathVariable String id,
    @RequestParam(required=false) String name) {

    model.addAttribute("id", id);
    model.addAttribute("name",
        name == null ? "No Name" : name);

    return "hello";
}
```

Better yet, use Java 8's `Optional`:

```
...
import java.util.Optional;
...

@RequestMapping("/hello/{id}")
public String hello(Model model,
    @PathVariable String id,
    @RequestParam Optional<String> name) {

    model.addAttribute("id", id);
    model.addAttribute("name", name.orElse("No Name"));

    return "hello";
}
```

Apart from path variables and request parameters, sometimes you may also need to receive data coming by other means, e.g. *request headers*. It's no difficult – refer the [documentation](#) of Spring MVC for the details.

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

8 Displaying And Submitting Forms

In this chapter, we are going to see how to display forms. Specifically, we are going to display a *signup* form where users can register by feeding their *email id*, *name* and a *password*.

Let's first code the UI. Follow these steps for that:

1. Copy *home.jsp* to *signup.jsp*
2. Replace "Hello, JSP!" in that with Bootstrap's [form](#) snippet.
3. Delete the *divs* enclosing *file* and *checkbox* fields.
4. Add a name field, as below, between the email and password fields:

```
<div class="form-group">
  <label for="exampleInputName1">Name</label>
  <input type="text" class="form-control" id="exampleInputName1" placeholder="Name">
</div>
```

Your *signup.jsp* should now be looking as below:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@include file="includes/header.jsp"%>

<form>
  <div class="form-group">
    <label for="exampleInputEmail1">Email address</label>
    <input type="email" class="form-control" id="exampleInputEmail1"
      placeholder="Email">
  </div>
  <div class="form-group">
    <label for="exampleInputName1">Name</label>
    <input type="text" class="form-control" id="exampleInputName1"
      placeholder="Name">
  </div>
  <div class="form-group">
    <label for="exampleInputPassword1">Password</label>
    <input type="password" class="form-control" id="exampleInputPassword1"
      placeholder="Password">
  </div>

  <button type="submit" class="btn btn-default">Submit</button>
</form>

<%@include file="includes/footer.jsp"%>
```

Next, add a view controller in the *MvcConfig* class, as below:

```

@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
        registry.addViewController("/signup").setViewName("signup");
    }
}

```

If you run the application now and visit `/signup`, you should see the form, as below:

The screenshot shows a web browser window with the address bar displaying `localhost:8080/signup`. The page features a header with navigation elements: 'Brand', 'Link', 'Link', 'Dropdown', a search input, a 'Submit' button, another 'Link', and another 'Dropdown'. The main content area contains a form with three input fields: 'Email address' (with placeholder 'Email'), 'Name' (with placeholder 'Name'), and 'Password' (with placeholder 'Password'). A 'Submit' button is located below the password field.

To give it a better look, enclose the form in a Bootstrap [panel](#)-primary component, and change the button's class from `btn-default` to `btn-primary`, as below:

```

<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@include file="includes/header.jsp"%>

<div class="panel panel-primary">
  <div class="panel-heading">Please sign up</div>
  <div class="panel-body">
    <form>
      <div class="form-group">
        <label for="exampleInputEmail1">Email address</label>
        <input type="email" class="form-control" id="exampleInputEmail1"
          placeholder="Email">
      </div>
      <div class="form-group">
        <label for="exampleInputName1">Name</label>
        <input type="text" class="form-control" id="exampleInputName1"
          placeholder="Name">
      </div>
      <div class="form-group">
        <label for="exampleInputPassword1">Password</label>
        <input type="password" class="form-control" id="exampleInputPassword1"
          placeholder="Password">
      </div>
    </form>
  </div>
</div>

```

```

        <button type="submit" class="btn btn-primary">Submit</button>
    </form>
</div>
</div>

<%@include file="includes/footer.jsp"%>

```

Handling form submission

To handle the form submission, first do the following updates to *signup.jsp*:

1. Add `method="post"` to the *form* element. We want the posing URL to be the same */signup*; otherwise adding an `action` attribute would also have been needed.
2. Add `name` attributes to all input fields. The name attributes become the request parameters, as you would be knowing. So, add `name="email"`, `name="name"` and `name="password"` attributes to the *email*, *name* and *password* input fields, respectively.

Next, code a signup controller, say in the *.controllers* package, looking as below:

```

package com.naturalprogrammer.spring.tutorial.controllers;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class SignupController {

    private static final Log log = LogFactory.getLog(SignupController.class);

    @RequestMapping(path="/signup", method=RequestMethod.POST)
    public String doSignup(
        @RequestParam String email,
        @RequestParam String name,
        @RequestParam String password) {

        log.info("Email: " + email + "; Name: " + name + "; Password:" + password);

        return "redirect:/";
    }
}

```

Notice how we are using `method=RequestMethod.POST` in `@RequestMapping` to indicate that only POST requests should come to the handler. We expect the GET requests to go to the ViewController, for displaying the signup form.

Beginning with Spring 4.3 (Spring Boot 1.4), we can actually use `@PostMapping` instead of `@RequestMapping(method=RequestMethod.POST)`. So, change the above code as below:

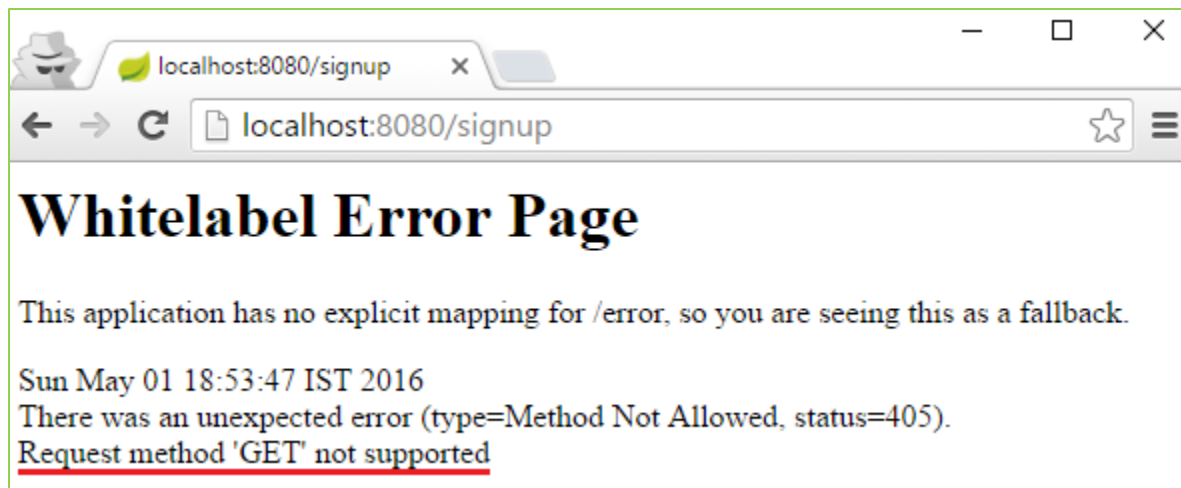
```
@RequestMapping(path="/signup", method=RequestMethod.POST)
@PostMapping("/signup")
public String doSignup(
```

Secondly, notice how we are receiving the request parameters as method arguments, using `@RequestParam`.

Lastly, look at the return statement. When we return a string starting with `"redirect:"`, a redirect response is sent instead of forwarding to a view. As a result, when the above handler finishes executing, the user'll be redirected to `"/"`.

Similarly, when a return value starts with `"forward:"`, the request is forwarded to another URL.

Time to test all this out. Visit `/signup`, and you get the following error!



That's because using the same `"/signup"` path both in a `ViewController` as well as a `Controller` won't work. So, remove the `/signup` `ViewController` entry from `MvcConfig`:

```
@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
        registry.addViewController("/signup").setViewName("signup");
    }
}
```

And, add a GET handler in `SignupController`, as below:

```
@Controller
public class SignupController {

    ...
}
```

```

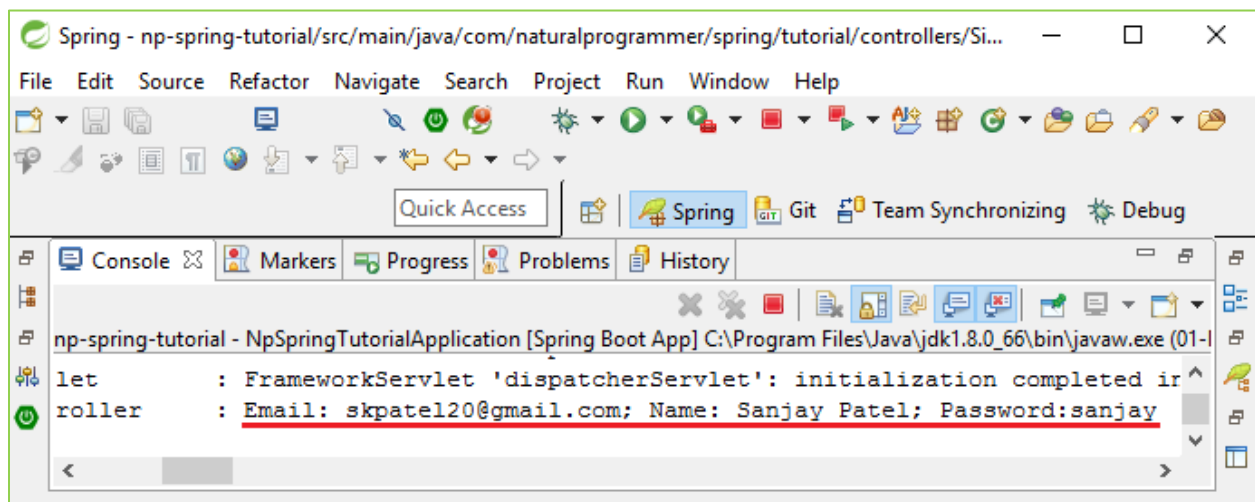
    @GetMapping("/signup")
    public String signup() {

        return "signup";
    }

    ...
}

```

Try visiting `/signup` now, and you'll see the form. Enter some data there and press *Submit*. You'll be redirected to the home page on your browser, and you'll see the expected log line on your console, as below:



@RequestMapping at class level

`@RequestMapping` can also be used at class level, in which case it applies to all handlers in the class. For example, both the handlers in our *SignupController* are now mapped to the `/signup` path. We can move that path mapping to the class level, as below:

```

@Controller
@RequestMapping("/signup")
public class SignupController {

    ...

    @GetMapping("/signup")
    public String signup() {
        ...
    }

    @PostMapping(path="/signup")
    public String doSignup(...) {
        ...
    }
}

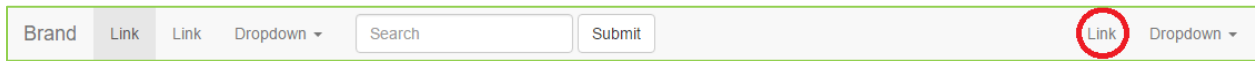
```



```
}
```

Linking /signup from UI

Before moving forward, let's provide a link to */signup* in our navigation bar. Specifically, let's replace the "Link" that you see below with a "Signup" link.

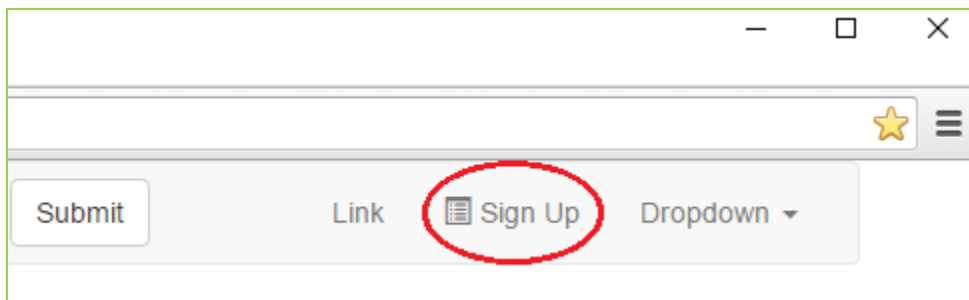


For that, change *header.jsp* as below:

```
...
    </li>
  </ul>
  <form class="navbar-form navbar-left" role="search">
    <div class="form-group">
      <input type="text" class="form-control" placeholder="Search">
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
  </form>
  <ul class="nav navbar-nav navbar-right">
    <li><a href="#">Link</a></li>
    <li><a href="<c:url value='/signup' />">
      <span class="glyphicon glyphicon-list-alt"></span> Sign Up
    </a></li>
    <li class="dropdown">
  ...
```

Notice that

1. Instead of using the raw */signup* link, we are using `<c:url>`. That makes our application more robust, as you may be knowing.
2. We have prefixed "Sign Up" with ``. That will display a Bootstrap [Glyphicon](#), as below:



Using Java beans to receive data

Our *doSignup* handler is receiving the request parameters as different method parameters. But, Spring provides a better way to receive multiple request parameters. We can use a Java bean, with field names matching the parameter names. For example, code a **User** class, say in a new *com.naturalprogrammer.spring.tutorial.domain* package, as below:

```
package com.naturalprogrammer.spring.tutorial.domain;

public class User {

    private String email;
    private String name;
    private String password;

    // Getters and Setters

    ...

}
```

Then, change *doSignup* as below:

```
...

import com.naturalprogrammer.spring.tutorial.domain.User;

@Controller
@RequestMapping("/signup")
public class SignupController {

    ...

    @PostMapping
    public String doSignup(User user) {

        log.info("Email: " + user.getEmail()
            + "; Name: " + user.getName()
            + "; Password:" + user.getPassword());

        return "redirect:/";

    }

}
```

Test it out, and it should work properly.

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

9 Validation

In this chapter, we'll see how to validate the input parameters received in our handler methods. Specifically, in our *doSignup* handler, we'd like to check for a few conditions, e.g. *the fields are not blank*. If those conditions aren't met, we'd like to redisplay the form, along with the error messages.

Using Hibernate Validator

Spring [supports](#) JSR-303 Bean Validation API. In fact, by including the *spring-boot-starter-web* dependency in *pom.xml*, we already have the [Hibernate validator](#) in our classpath, which is an implementation of the API.

So, let's see how to use Hibernate validator to validate the *doSignup* input data.

Using constraint annotations

The first step will be to annotate the fields of our User class with the desired constraint annotations, as below:

```
package com.naturalprogrammer.spring.tutorial.domain;

import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotBlank;

public class User {

    @NotBlank
    @Email
    @Size(min=4, max=250)
    private String email;

    @NotBlank
    @Size(max=100)
    private String name;

    @NotBlank
    @Size(min=6, max=40)
    private String password;

    // Getters and Setters

    ...
}
```

The annotations above specify that

1. Email shouldn't be blank, should be of the email format, and its size should be between 4 and 250 characters.
2. Name shouldn't be blank, and should be of 100 characters max.
3. Password shouldn't be blank, and its size should be between 6 and 40 characters.

[Here](#) is a full list of built-in constraints with their documentation.

Validating

Now that we have specified the validations we need, Spring will do the validation and report the errors if we change our handler as below:

```
...

import org.springframework.validation.BindingResult;
import org.springframework.validation.annotation.Validated;

...

@Controller
@RequestMapping("/signup")
public class SignupController {

    ...

    @PostMapping
    public String doSignup(@Validated User user, BindingResult result) {

        if (result.hasErrors())
            return "signup";

        ...

    }

}
```

Notice that,

1. We have annotated the User parameter with `@Validated`. That'll tell Spring to do the validations, and put any errors in the next parameter, which is a *BindingResult*. You must provide a *BindingResult* parameter following the parameter(s) being validated, as we have done.
2. In the code, we are checking if there are any errors, and returning the view name if so. That'll result in re-displaying the view to the user, in case of any validation errors. So, the user'll see the page again, where she can correct the errors and re-submit.

You can also use `@Valid` instead of `@Validated`. But `@Validated` lets you use validation groups, as we'll see [later](#).

Re-displaying the form

Test it out. When you'll feed some invalid data, the same signup page will be re-rendered. But, there are a couple of problems:

1. No error messages are being displayed, so that the user can know about the errors and rectify those.
2. The original data that we had fed is lost. That should be available in the fields, so that we do just the necessary edit.

How to address these?

If you think that we could add the existing form data and the errors as model attributes, and then render those in the JSP, you're right!

But we don't have to do it manually. Spring automatically adds the bean arguments, such as our `user` argument, and BindingResult arguments, such as our `result` argument to the model.

So, it's just a matter of using those model attributes in the JSP. Using Spring's [form tag library](#) makes it trivial. Let's see how.

First, add the following directive to the top of your `header.jsp`:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<!DOCTYPE html>
...
```

Then, change the form in `signup.jsp` as below:

```
<form:form modelAttribute="user" method="post" role="form">

  <form:errors cssClass="error" />

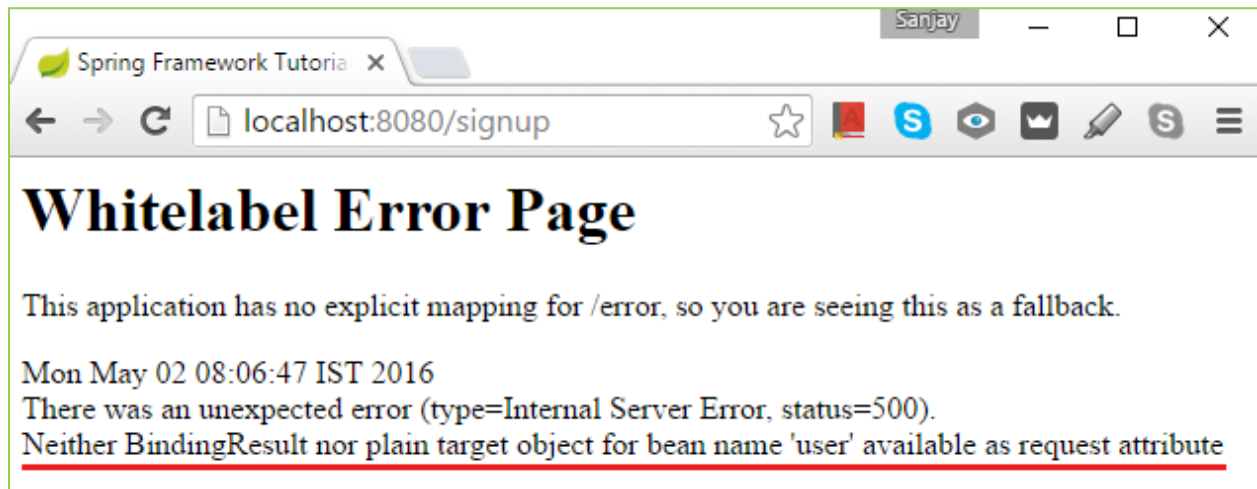
  <div class="form-group">
    <form:label path="email">Email address</form:label>
    <form:input path="email" type="email" class="form-control"
      placeholder="Enter email" />
    <form:errors path="email" cssClass="error" />
  </div>
  <div class="form-group">
    <form:label path="name">Name</form:label>
    <form:input path="name" class="form-control" placeholder="Enter name" />
    <form:errors path="name" cssClass="error" />
  </div>
  <div class="form-group">
    <form:label path="password">Password</form:label>
    <form:password path="password" class="form-control" placeholder="Password" />
    <form:errors path="password" cssClass="error" />
  </div>
  <button type="submit" class="btn btn-primary">Submit</button>
```

```
</form:form>
```

Specifically,

1. We have replaced the standard *form*, *label* and *input* tags with `<form:form>`, `<form:label>`, `<form:input>` and `<form:password>`. That'll make Spring render the form as we wanted.
2. `<form:form>` has a `modelAttribute="user"` attribute. That binds the form to the `user` attribute of the model. As we discussed earlier, the `user` argument of our `doSignup` handler would have been added to model automatically – so this'll work.
3. `<form:form>` no more needs a `method="post"` attribute, because that's now the default.
4. The `path="email"` attributes bind the associated elements to the email field of the user bean.
5. `<form:errors />` will show form level errors.
6. `<form:errors path="email" />` will show errors related to email field. Same for other fields.
7. The `cssClass` attribute in `form:errors` elements will give a css class to the error messages, which can be used to show the error distinctly, as we'll see later.
8. `id` attributes are removed from `form:label` and `form:input` elements. Those will be automatically provided by Spring.

Ready to test it out? If you now try to visit the signup page, you see an error as below:



That's because, the `<form:form modelAttribute="user">` in our `signup.jsp` is now expecting a `user` model attribute to be available. But, in our signup GET handler, which looks as below, we doesn't supply one.

```
@GetMapping
public String signup() {

    return "signup";
}
```

So, let's supply a blank user model attribute, as below:

```
@GetMapping
public String signup(Model model) {

    model.addAttribute("user", new User());
    return "signup";
}
```

The above code should be obvious to you, because you know how to add attributes to the model. However, the first argument to the `addAttribute` above is redundant when the name of the attribute, i.e. “user,” is same as the name of the class, i.e. “User,” in camelCase. So, remove the first argument:

```
model.addAttribute("user", new User());
```

Test it now, and it should work.

To see the errors, you may like to temporarily squeeze the constraint limits. For example, you can decrease the maximum size of the email field to 10, and see how the errors are shown.

@ModelAttribute

We saw how Spring automatically adds the bean arguments of handler methods to the model – that’s how the `user` argument of our `doSignup` handler got added to the model.

The name of such model attribute would be the name of the class in camelCase. That’s why we had “user” going into the model, which was then referred in the JSP as below:

```
<form:form modelAttribute="user" role="form">
```

To change the name of the model attribute, the `@ModelAttribute` annotation can be used on the parameters, as below:

```
public String doSignup(@ModelAttribute("signupForm") @Valid User user,...
```

Using `@ModelAttribute` on method parameters as above is discussed in more depth [here](#).

The `@ModelAttribute` annotation can also be used on methods inside controller classes, to add something to the model before any of the handler inside the controller class is called. For more details, refer [here](#).

Customizing error messages

The error messages shown now are too generic. For example, when you leave the email blank, you see “may not be empty.” Instead, “Please provide your email address” would definitely be better.

There are a couple of ways to customize such error messages, as we’ll see next.

Using the message attribute

Default error messages can be overridden by providing a message attribute to the constraint annotations, as below:

```
@NotBlank(message = "Please provide your email address")
@email
@Size(min=4, max=250)
private String email;
```

Internationalizing messages

Instead of passing a literal message as above, you can also pass a message code enclosed with curly braces. For example, try the following:

```
@NotBlank(message = "{blankEmail}")
@email
@Size(min=4, max=250)
private String email;
```

The message will then be picked from a `ValidationMessages.properties` file in classpath root. So, create a `ValidationMessages.properties` file at `src/main/resources`, and add the following line to it:

```
blankEmail: Please provide your email address
```

Just like `messages.properties` that we have seen earlier, localized `ValidationMessages.properties` files can be provided, named `ValidationMessages_<locale>.properties`.

Other attributes of the annotations, such as `min` and `max`, can be used in the messages. For example, provide a message code for the `@Size` annotation, as below:

```
@NotBlank(message = "{blankEmail}")
@email
@Size(min=4, max=250, message = "{emailSize}")
private String email;
```

And, add a line to `ValidationMessages.properties`, as below:

```
emailSize: Email must be between {min} and {max} chars long
```

Notice how we have used the `min` and `max` attributes in the message above.

Test it out, you'll see error messages as below:

Spring Framework Tutorial x

localhost:8080/signup

Brand Link Link Dropdown Search Submit Link Sign Up Dropdown

Please sign up

Email address

Enter email

Email must be between 4 and 250 chars long
Please provide your email address

Name

Sanjay Patel 101

Password

.....

Submit

Using messages.properties

We already maintain a set of *messages*.properties* to hold our localized messages. Why maintain another set of *ValidationMessages*.properties*, and not put our validation messages in *messages*.properties*, you may ask?

Well, that can be configured easily. For that, we need to configure a custom validator that would, instead of using *ValidationMessages*.properties*, use the existing `messageSource` bean. Remember the `messageSource` bean? That's the one which serves messages from *messages*.properties*.

A custom validator can be configured by overriding the `getValidator()` method of the `WebMvcConfigurerAdapter` in our `MvcConfig` class, just as below:

```
package com.naturalprogrammer.spring.tutorial.config;

...

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.MessageSource;
import org.springframework.validation.Validator;
import org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;

@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {

    ...

    @Autowired
    private MessageSource messageSource;

    @Override
```

```

    public Validator getValidator() {
        LocalValidatorFactoryBean factory = new LocalValidatorFactoryBean();
        factory.setValidationMessageSource(messageSource);
        return factory;
    }
}

```

In the `getValidator()` above, we are creating a new validator, injecting our `messageSource` into that, and returning that. The new validator is nothing but an instance of Spring's [LocalValidatorFactoryBean](#) class, which Spring uses for the validation.

So, paste the above code in `MvcConfig`, move the messages from `ValidationMessages.properties` to `messages.properties` and delete `ValidationMessages.properties`. Now test it out, and it should work well.

Using DefaultMessageCodesResolver

Let's now see another way to customize error messages.

Append the following line in `messages.properties`:

```

hello: Hi
name: David

blankEmail: Please provide your email address
emailSize: Email must be between {min} and {max} chars long

NotBlank.user.email: Another way: Please provide your email address

```

Now, the above error will be shown if email is left blank:

Email address
Enter email
Email must be between 4 and 250 chars long
<u>Another way: Please provide your email address</u>

So, how does this work?

Spring uses a `DefaultMessageCodesResolver` for getting the error messages, which gets auto configured when your application starts. For getting an error message, the `DefaultMessageCodesResolver` looks for message codes in certain formats. Its [Javadoc](#) explains in details how exactly it works, so we aren't covering that here.

Let's take another example. Append another line in `messages.properties`:

```

hello: Hi
name: David

```

```
blankEmail: Please provide your email address
emailSize: Email must be between {min} and {max} chars long

NotBlank.user.email: Another way: Please provide your email address
Size.email: Another way: {0} must be between {2} and {1} chars long
```

Test it out, and you'll see the above message, with the placeholders `{0}`, `{1}` and `{2}` substituted with the *field name*, and *max* and *min* attributes respectively:

Email address

Enter email

Another way: Please provide your email address

Another way: email must be between 4 and 250 chars long

So, which way to prefer?

We saw a couple of ways to customize error messages. You can choose either, or take a mixed approach. We slightly prefer the former one, because that seems a bit straightforward and allows to use placeholders by name, e.g. `{min}` and `{max}`.

So, for this tutorial, let's go ahead with the former approach. Remove the following lines from *messages.properties*:

```
NotBlank.user.email: Another way: Please provide your email address
Size.email: Another way: {0} must be between {2} and {1} chars long
```

Distinct error messages

In *signup.jsp*, look at some *form:errors* element, e.g.:

```
<form:errors path="email" cssClass="error" />
```

The `cssClass` attribute specifies a css class name to be given to the rendered error messages.

That means, we can customize the look of the error messages just by providing an "error" css class. So, let's do that. Create a *styles.css* in *src/main/resources/static/css* folder, and add an error class in that, looking as below:

```
form .error {
    color: #A94442;
}
```

Then, link that from the *header.jsp*. by adding the following line below the bootstrap css link:

...

```
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Spring Framework Tutorial For Beginners</title>
  <!-- Bootstrap -->
  <link href="/js/lib/bootstrap-3.3.6-dist/css/bootstrap.min.css" rel="stylesheet">

  <link href="/css/styles.css" rel="stylesheet">

  . . .
```

That's it – you'll now see reddish error messages, as below:

Email address

Email must be between 4 and 250 chars long

Please provide your email address

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

10 Spring Data

Spring Framework provides extensive support for working with databases. You can use JDBC with `JdbcTemplate`, or ORM technologies such as Hibernate or JPA. Or, you can use Spring Data, which provides an additional level of functionality, and is the most common and recommended way today.

Spring Data is actually a set of projects – for accessing the multiple kinds of data. For example, [Spring Data JPA](#) is for accessing relational databases, whereas [Spring Data MongoDB](#) is for accessing MongoDB.

In this chapter, we are going to see how to use Spring Data JPA in our project.

As the name implies, Spring Data JPA uses [JPA](#) for accessing databases. No problem if you don't know JPA – we are going to cover the needed basics. Nonetheless, JPA is a good thing to learn, and you can find some excellent videos for it [here](#).

So, let's use Spring Data JPA and save our users to a MySQL database.

Adding dependencies

To use Spring Data JPA and MySQL, first add the following dependencies to *pom.xml*:

```
...  
  
<dependencies>  
    ...  
    <dependency>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-data-jpa</artifactId>  
    </dependency>  
    <dependency>  
        <groupId>mysql</groupId>  
        <artifactId>mysql-connector-java</artifactId>  
        <scope>runtime</scope>  
    </dependency>  
    </dependencies>  
    ...
```

The second one is for the MySQL driver. Note that its scope is set to *runtime*. That's because we don't want to use the driver directly in our code.

Providing connection details

Next, install MySQL if it's not already installed on your machine. On Windows machines, I'd install [XAMPP](#), which includes MySQL.

Then, create a MySQL database, and specify the connection details in *application.properties*, as below:

```
spring.datasource.url: jdbc:mysql://localhost:3306/dbname
spring.datasource.username: username
spring.datasource.password: password
```

Note that the connection details, e.g. *dbname*, *username* and *password*, might be different in your case.

When Spring Boot sees the above properties, it auto-configures a [DataSource](#), which is then used by Spring Data JPA. To fine tune the configuration, you can provide more properties. See the [Spring Boot documentation](#) for details.

Mapping entity classes

JPA needs us to map our domain classes to relational tables. For example, our *User* class would map to a user table in the database.

The mapping can be done through either XML, or by using annotations. Let's see how to map our *User* class using annotations.

@Entity and @Table

Annotating a class with `@Entity` maps it to a table, and all its fields become columns. So, annotate our *User* class as below:

```
@Entity
public class User {
```

All JPA annotations, e.g. the `@Entity` above, would be in the `javax.persistence` package.

By default, the class name in camelCase becomes the table name. That means, our *User* class will be mapped to a "user" table in the database.

To change the default table name, we can use the `@Table` annotation. In fact, it's a good idea not use "user" as a table name, because it's a SQL reserved word. So, let's change the table name to "usr":

```
@Entity
@Table(name="usr")
```

```
public class User {
```

@Id

A table in a database should have a single or multi column *primary key*, which will uniquely identify a row. Single-column primary keys are more common than multi-column primary keys.

If a field of an @Entity class is annotated with @Id, that field becomes the single-column primary key for the table.

So, let's create a new `id` field in our User class, and annotate it with @Id:

```
@Id
@GeneratedValue
private long id;
```

Note that we also have annotated the field with @GeneratedValue, and the type of the field is *long*. Consequently, when a new row will be inserted, the id will be auto generated by the database, incrementing 1 each time.

Don't forget to code getter and setter for the field.

@Column

Although all the fields of an @Entity class are automatically mapped to columns, for applying certain constraints, we can annotate those with @Column. For example, annotate the fields of User with @Column, as below:

```
@NotBlank(message = "{blankEmail}")
>Email
@Size(min=4, max=250, message = "{emailSize}")
@Column(nullable = false, length = 250)
private String email;

@NotBlank
@Size(max=100)
@Column(nullable = false, length = 100)
private String name;

@NotBlank
@Size(min=6, max=40)
@Column(nullable = false) // no length because it will be encrypted
private String password;
```

`nullable = false` means that the column can't have null values. `length` specifies the maximum allowed number of characters.

Instead of hardcoding and repeating the *length* attribute both in @Size and @Column, you can define those as constants.

Auto-creating the schema when application starts

For JPA to automatically create the database schema to match the mappings when your application starts, add a property as below to your *application.properties*:

```
spring.jpa.hibernate.ddl-auto: create
```

This property can take one of the four values:

- *validate*: validate the schema, makes no changes to the database
- *update*: update the schema
- *create*: creates the schema, destroying previous data
- *create-drop*: drop the schema at the end of the session

We have specified *create*, which means that, when the application restarts, any previous data will be destroyed, and the schema will be re-created. Obviously, it's only for the development environment. When you are checking out the application and want your fed data to survive across restarts, use *update*, which will update the schema, but won't delete any existing data. For production environments, we can use *update*, but if you want to maintain your schema manually rather than allowing hibernate doing that, use *validate*.

Now, if you start your database service, run the application and then peep into the database using some tool like *MySQL Workbench*, you'll find the *usr* table created, with all the metadata such as primary key and column constraints.

Saving and retrieving data

To save and retrieve data, Spring Data can create repository objects. A repository object for an entity will be automatically created if you just code its interface. For example, create a *UserRepository* interface as below, say in a new *com.naturalprogrammer.spring.tutorial.repositories* package, for accessing our *User* entity:

```
package com.naturalprogrammer.spring.tutorial.repositories;

import org.springframework.data.jpa.repository.JpaRepository;
import com.naturalprogrammer.spring.tutorial.domain.User;

public interface UserRepository extends JpaRepository<User, Long> {

}
```

Notice that the *UserRepository* extends the *JpaRepository* interface. The type of entity and ID it is supposed to work with, i.e. *User* and *Long*, are specified as the generic parameters of the *JpaRepository*.

By extending [JpaRepository](#), our *UserRepository* inherits several methods for accessing User data, including methods for saving, deleting, and finding User entities. We are going to use some of these later.

Custom query methods

Spring Data also allows us to define custom query methods in the repository interface by simply declaring their method signature. For example, to fetch a user by email, just declare a `findByEmail` method as below:

```
...  
  
import java.util.Optional;  
  
...  
  
public interface UserRepository extends JpaRepository<User, Long> {  
    Optional<User> findByEmail(String email);  
}
```

When Spring Data creates a *UserRepository* object, it also creates the implementation of such methods. By parsing the method name, its query builder mechanism tries to build the query. The query builder mechanism is quite smart – see its [documentation](#) for more details.

But, there may be cases when you need to specify the query manually, either in JPA or native SQL form. The [@Query](#) annotation can be used in such cases.

Spring Data JPA is very powerful – it can do much more than we just discussed. See its [reference material](#) for more details. We'll be using some more of its features, like auditing and optimistic locking, in [Module III](#).

Saving users

In our *SignupController*, we can now inject the *UserRepository* and use its `save` method to save the user.

But, controller layers should be thin, and we should actually use a service layer for such operations. So, first create a `UserService` interface, say in a new *com.naturalprogrammer.spring.tutorial.services* package, as below:

```
package com.naturalprogrammer.spring.tutorial.services;  
  
import com.naturalprogrammer.spring.tutorial.domain.User;  
  
public interface UserService {  
    void signup(User user);  
}
```

```
}
```

Then, create its implementation as below:

```
package com.naturalprogrammer.spring.tutorial.services;

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

import com.naturalprogrammer.spring.tutorial.domain.User;
import com.naturalprogrammer.spring.tutorial.repositories.UserRepository;

@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserRepository userRepository;

    @Override
    public void signup(User user) {

        userRepository.save(user);
    }
}
```

`userRepository.save` will persist the user object to the database. Before persisting, JPA will validate the fields of the user objects that are annotated with JSR-303 constraints such as `@Size`. But, those are already validated in the controller, and so, to prevent revalidation, add this line in `application.properties`:

```
spring.jpa.properties.java.persistence.validation.mode: none
```

Note that we have annotated `ServiceImpl` with `@Service`. As a result, Spring will create its bean, which can be injected in the `SignupController`.

So, inject `UserService` in `SignupController`, and use that to save the user. Your `SignupController` should now be looking as below:

```
...

@Controller
@RequestMapping("/signup")
public class SignupController {

    private static final Log log = LoggerFactory.getLogger(SignupController.class);

    @Autowired
    private UserService userService;

    ...

    @PostMapping
```

```

    public String doSignup(@Validated User user, BindingResult result) {

        if (result.hasErrors())
            return "signup";

        log.info("Email: " + user.getEmail() +
            +", Name: " + user.getName() +
            +", Password: " + user.getPassword());

        userService.signup(user);

        return "redirect:/";
    }
}

```

Start MySQL if it's not already running, and try running the application now. Then, try a signup. You will be able to see the record in the database.

User Roles

Most real world applications need to categorize their users by some roles. In our application, let's plan to have three kinds of users:

1. UNVERIFIED – A new users whose mail isn't verified yet.
2. BLOCKED – A bad user, one who is blocked.
3. ADMIN – An administrator, who is able to do the admin operations.

So, create a **Role** enum in the User class, as below:

```

public class User implements UserDetails {

    public static enum Role {
        UNVERIFIED, BLOCKED, ADMIN
    }

    ...
}

```

Then, add a **roles** collection in the User class, which will contain all the roles of the user:

```

...

import java.util.Collection;
import java.util.HashSet;

...

public class User implements UserDetails {

    ...

    @ElementCollection(fetch = FetchType.EAGER)
    private Collection<Role> roles = new HashSet<Role>();
}

```

```

    public Collection<Role> getRoles() {
        return roles;
    }
    public void setRoles(Collection<Role> roles) {
        this.roles = roles;
    }
    ...
}

```

Finally, to add UNVERIFIED role to a newly signed up user, change the signup method of *UserServiceImpl* as below:

```

@Service
public class UserServiceImpl implements UserService {

    ...

    @Override
    public void signup(User user) {

        user.getRoles().add(Role.UNVERIFIED);
        userRepository.save(user);
    }
}

```

Test it out.

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

11 Flash Attributes

After doing a sign up, the user currently gets redirected to the home page. But we don't show her any message saying that the signup succeeded.

To show a message following such a redirect, we'll need to pass the message to the redirected page. Spring's [flash attributes](#) come handy in such situations.

Technically speaking, flash attributes provide a way for one request to store attributes intended for use in another. If we add a parameter of type `RedirectAttributes` to a handler method, whatever attributes we add to that parameter will be available as model attributes in the redirected page.

To see it working, update our *doSignup* handler as below:

```
...

import org.springframework.web.servlet.mvc.support.RedirectAttributes;

...

@PostMapping
public String doSignup(@Valid User user,
    BindingResult result,
    RedirectAttributes redirectAttributes) {

    if (result.hasErrors())
        return "signup";

    userService.signup(user);

    redirectAttributes.addFlashAttribute("flashMessage", "Signup succeeded!");
    redirectAttributes.addFlashAttribute("flashKind", "success");

    return "redirect:/";
}
```

Notice that, in addition to the message, we also have added a `flashKind` attribute. That can take values like "success", "danger", etc., suggesting the css class of the displayed message.

Next, add a Bootstrap [alert component](#) to the bottom of *header.jsp*, as below:

```
...

        </ul>
    </div><!-- /.navbar-collapse -->
</div><!-- /.container-fluid -->
</nav>

<c:if test="${not empty flashMessage}">
```

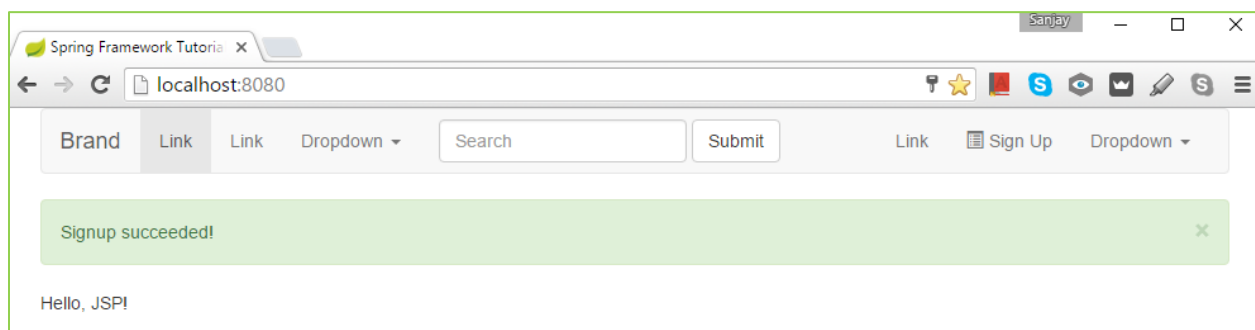
```

<div class="alert alert-${flashKind} alert-dismissible"
      role="alert">
  <button type="button" class="close" data-dismiss="alert"
    aria-label="Close">
    <span aria-hidden="true">&times;</span>
  </button>
  ${flashMessage}
</div>
</c:if>

```

It'll show a flash message, if present, in a Bootstrap alert component. Notice how the *flashMessage* and *flashKind* model attributes are used.

Test signing up now, and you'll see a beautiful success message, as below:



Showing an internationalized message

To show i18n messages, we can replace the hardcoded message with our [getMessage](#) utility method, as below:

```

redirectAttributes.addFlashAttribute("flashMessage",
    MyUtil.getMessage("signupSuccess"));

```

Of course, it'll need us to add the message in *messages.properties*:

```

signupSuccess: Signup succeeded!

```

Coding a flash method

Why not move the flash code to a utility method? Create a utility method as below, in our *MyUtil* class:

```

public static void flash(RedirectAttributes redirectAttributes,
    String kind, String messageKey) {

    redirectAttributes.addFlashAttribute("flashKind", kind);
    redirectAttributes.addFlashAttribute("flashMessage",

```

```
        MyUtil.getMessage(messageKey));  
    }
```

Now, change the *doSignup* handler just to call our *flash* method:

```
@PostMapping  
public String doSignup(@Validated User user,  
    BindingResult result,  
    RedirectAttributes redirectAttributes) {  
  
    if (result.hasErrors())  
        return "signup";  
  
    userService.signup(user);  
  
    redirectAttributes.addFlashAttribute("flashMessage",  
        MyUtil.getMessage("signupSuccess"));  
    redirectAttributes.addFlashAttribute("flashKind", "success");  
    MyUtil.flash(redirectAttributes, "success", "signupSuccess");  
  
    return "redirect:/";  
}
```

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

12 Custom Validation

Although Hibernate validator has a rich set of [built-in constraints](#), we are going to need validations beyond that. For example, when signing up, we need to check the uniqueness of the email id.

Spring supports multiple ways for doing such custom validations. Creating custom constraints is one of those, which we'll discuss next.

Creating custom constraints

Similar to the built-in constraints like `@NotBlank`, we can easily create custom constraints. Those can just be a composition of a few existing constraints, or can contain actual custom logic.

Constraint composition

In our User class, the password field is annotated as below:

```
@NotBlank
@Size(min=6, max=40)
private String password;
```

If you later code a *change password* form, probably you are going to copy/paste these annotations there. That's not desirable, because, if your CEO one day feels that passwords should be at least 15 characters long, you'll end up updating your code at multiple places.

That's why you should create higher level constraints, even if they would just be a composition of other constraints.

To create such a composed constraint, simply annotate the constraint declaration with its comprising constraints. For example, code a `@Password` annotation as below, say in a new *com.naturalprogrammer.spring.tutorial.validation* package:

```
package com.naturalprogrammer.spring.tutorial.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
```



```
import javax.validation.Payload;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.NotBlank;

@Constraint(validatedBy = { })
@NotBlank(message="{blankPassword}")
@Size(min=6, max=40, message="{passwordSize}")
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
public @interface Password {

    String message() default "";

    Class<?>[] groups() default { };

    Class<? extends Payload>[] payload() default { };

}
```

The `@Constraint` annotation that you see above makes *Password* a custom constraint. It takes a `validatedBy` attribute, which would be a list of classes containing the validation logic. It's an empty list in our case, because we don't have any validation logic. Instead, we want *Password* to be a composition of `@NotBlank` and `@Size`, and so we have used those annotations, as you can see above.

Other annotations, e.g. `@Target`, `@Retention` and `@Documented` are standard ones, used for [creating Java annotations](#).

Next, add the used messages in *messages.properties*:

```
blankPassword: Please provide a password
passwordSize: Password must be between {min} and {max} chars long
```

The new `@Password` annotation can now be used in the *User* class:

```
@NotBlank
@Size(min=6, max=40)
@Password
@Column(nullable = false) // no length because it will be encrypted
private String password;
```

Test it out.

Refer Hibernate validator's [documentation](#) to know more about how to create composed constraints.

Coding custom validation logic

Let's now code a `@UniqueEmail` constraint, which can be used for checking the uniqueness of the email id when a user signs up. Code it as below:

```

package com.naturalprogrammer.spring.tutorial.validation;

import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.constraints.Size;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotBlank;

@NotBlank(message = "{blankEmail}")
@Size(min=4, max=250, message = "{emailSize}")
@Constraint(validatedBy=UniqueEmailValidator.class)
>Email
@Documented
@Retention(RUNTIME)
@Target({METHOD, FIELD, CONSTRUCTOR, PARAMETER, ANNOTATION_TYPE})
public @interface UniqueEmail {

    String message() default "{duplicateEmail}";

    Class[] groups() default {};

    Class[] payload() default {};
}

```

This time, it'll not just be a composition of *@NotBlank* and *@Size*, but also will have some custom logic. That's why *@Constraint* now takes an argument, *UniqueEmailValidator.class*, which will contain the custom validation logic.

So, code a *UniqueEmailValidator* class, in the same package, as below:

```

package com.naturalprogrammer.spring.tutorial.validation;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

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

import com.naturalprogrammer.spring.tutorial.repositories.UserRepository;

@Component
public class UniqueEmailValidator
implements ConstraintValidator<UniqueEmail, String> {

```

```

@Autowired
private UserRepository userRepository;

@Override
public void initialize(UniqueEmail constraintAnnotation) {
    // initialization code
}

@Override
public boolean isValid(String email, ConstraintValidatorContext context) {

    return !userRepository.findByEmail(email).isPresent();
}
}

```

Notice above that:

1. The class is annotated with *@Component*, so that Spring will create a bean for it.
2. The type of the annotation and the field, *UniqueEmail* and *String*, are specified as the generic parameters of *ConstraintValidator*.
3. The *initialize* method can have initialization code. Refer the [documentation](#) for more details.
4. *isValid* method contains the validation logic. It should return *true* if the input is valid, *false* otherwise. In our case above, we are returning *true* if the email is not already present in the database, *false* otherwise. To access the database, we are using the [findByEmail](#) method that we had coded earlier.

Constraints can be created not only for primitive fields, but also for classes. Class level constraints are useful when more than one fields participate in the validation. We'll see an example of it [later](#).

To know more details on creating custom constraints, refer its [documentation](#).

Creating a database index

For *userRepository.findByEmail* to work faster, as well as for ensuring the uniqueness of emails in the database level, we should create a unique database index for email. To do so, alter the *User* class declaration as below:

```

...

import javax.persistence.Index;

...

@Entity
@Table(name="usr", indexes = {
    @Index(columnList = "email", unique=true)
})
public class User {

    ...
}

```

In case your *usr* table already has duplicate emails, delete those rows manually, and then restart the application now. This should automatically create the index, which you can verify from MySQL Workbench or your favourite MySQL client.

Using the constraint

You can now update the email field declaration as below.

```
@NotBlank(message = "{blankEmail}")  
@Email  
@Size(min=4, max=250, message = "{emailSize}")  
@UniqueEmail  
@Column(nullable = false, length = 250)  
private String email;
```

Also, let's not forget to add a `duplicateEmail` message in *messages.properties*:

```
duplicateEmail: Email id already used
```

Time to test it out!

Extending LocalValidatorFactoryBean

Another way for doing custom validation in Spring is to code custom validation classes by extending [LocalValidatorFactoryBean](#). But, that does not seem to be as flexible and popular as creating the custom constraints that we just discussed. So, we thought not to cover that in this book.

Adding errors to BindingResult manually

Still another way to do custom validation will be to manually checking for the errors and inserting those to the BindingResult. We'll try it out [later](#).

Validation in service layer

We have been doing the validation in the controller layer, by using the `@Validated` annotation on handler parameters. But, validation is business logic, after all. So, many people argue that the service layer is more suited for it. We agree, and in fact, have covered it in depth in [Module III](#).

Source code so far

- [Browse online](#)
- [Download zip](#)

- [See changes](#)

13 Exception Handling

By default, a Spring Boot web application shows a terse error page when an exception is thrown while processing a request. To see how that default error page looks, just throw an exception, say from our *signup* GET handler, as below:

```
@Controller
@RequestMapping("/signup")
public class SignupController {

    ...
    @GetMapping
    public String signup(Model model) {

        model.addAttribute(new User());
        throw new RuntimeException("An error!");
        //return "signup";
    }

    ...
}
```

When you now visit */signup*, you'll see the following page:



Looks very dry, doesn't it? Spring Boot allows us to replace this view with a custom one. To do so, first, disable the default view by adding the following line in *application.properties*:

```
server.error.whitelabel.enabled: false
```

Then, add an *error.jsp* inside *src/main/webapp/WEB-INF/jsp*, looking as below:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
```

```

<%@include file="includes/header.jsp"%>

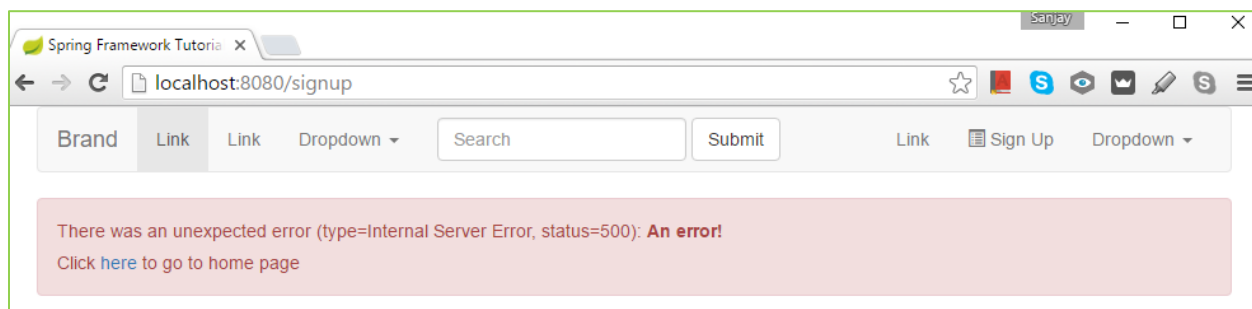
    <div class="alert alert-danger">
        <p>
            There was an unexpected error
            (type=${error}, status=${status}): <strong>${message}</strong>
        </p>
        <p>Click <a href="/">here</a> to go to home page</a></p>
    </div>

<%@include file="includes/footer.jsp"%>

```

Leaving aside the Bootstrap stuff, notice how we are using the `error`, `status` and `message` model attributes. These come from the [BasicErrorController](#), which Spring Boot uses for displaying exceptions.

Visit <http://localhost:8080/signup> now, and you'll see a nice looking error page, as below:



More customization

We just saw how to replace the error view. Spring Boot also allows more customization - you can replace its *BasicErrorController* with your own implementation. Or, you can code a *ControllerAdvice*. Refer Spring Boot [documentation](#) to know more about it, and for an exhaustive pattern for error handling, refer [Module III](#).

Before we move forward, don't forget to set right the *signup* code:

```

@Controller
@RequestMapping("/signup")
public class SignupController {

    ...
    @GetMapping
    public String signup(Model model) {

        model.addAttribute(new User());
        throw new RuntimeException("An error!");
        return "signup";
    }

    ...
}

```

```
}
```

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

14 Transactions

If you haven't heard of database transactions, those are sequence of operations performed as single logical units of work.

A classic example of a transaction is transferring money from one account to another in a bank. It would involve two operations – deducting money from the source account, and adding it to the target account. If the first operation succeeds but the second fails, say due to a hardware failure, money will be deducted from the source account, but will not be added to the target.

This is undesirable. So, when something in the middle of a transaction fails, all previous operations must be rolled back, so that the database comes back to the earlier stage.

More details about transactions can be found [here](#).

Most relational databases support transaction management. You can begin a transaction, execute multiple statements, and then end the transaction. If something wrong happens before the transaction ends, the transaction is *rolled back*, i.e. the result of previous operations are not saved to the database. On the other hand, when the transaction ends successfully, it's *committed*, i.e. the results of the operations are saved to the database.

Transactional methods in Spring

Writing transactional code in Spring is simple. If you just annotate a method with `@Transactional`, Spring begins a transaction before the method is executed, and ends the transaction after the method finishes executing.

Let's see it working. First of all, let's test whether our signup operation is now transactional. Add a throw statement in the `signup` service method in `UserServiceImpl`:

```
@Override
public void signup(User user) {
    ...
    userRepository.save(user);
    throw new RuntimeException();
}
```

If you try signing up now, the above exception will be thrown after `save` is called but before the signup method completes.

Look at the database now, and the signed up user will be available there. So, the save operation was committed. Had the above `signup` method been transactional, the save operation would have been rolled back instead.

Now, annotate the method with `@Transactional`, as below:

```

...

import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

...

@Service
public class UserServiceImpl implements UserService {

    ...

    @Override
    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
    public void signup(User user) {

        ...

        userRepository.save(user);
        throw new RuntimeException();
    }
}

```

Now, when the exception will be thrown inside the method, all previous operations will be rolled back. Consequently, the user will not be saved.

Test it out.

Transaction propagation

@Transactional annotation takes a **propagation** attribute, as you see above. We have set it as REQUIRED above. Consequently, when the signup method will be called, Spring will begin a transaction only if not already inside another transaction. In other words, when the signup method will be called from another *@Transactional* method, a new transaction will not be begun. Instead, its operations will be a part of the earlier transaction.

Another frequently used value for propagation is SUPPORTS. It's typically used for non-sensitive, read only operations. It states that, if already a transaction is begun, the operations will be a part of that. If not, the operations will execute non-transactional.

For other types of propagations, refer its [Javadoc](#).

readOnly

@Transactional also takes a **readOnly** attribute. If it's true, insert, update and delete operations will not be allowed inside the transaction. Some transaction managers may not support read only transactions, in which case this flag will be ignored.

Transactional classes

If `@Transactional` is used on a class, all the methods of the class become transactional. In fact, a common pattern is to

1. Annotate all service classes as below:

```
@Service
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class UserServiceImpl implements UserService {
```

2. Annotate all service methods that write to the database as below:

```
@Override
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void signup(User user) {
```

Whereas we have already annotated the `signup` method, annotate `ServiceImpl` as shown above. Also, remove throwing the exception, so that your code becomes ready to move to the next chapter.

Precisely, the `ServiceImpl` should now be looking as below:

```
package com.naturalprogrammer.spring.tutorial.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import com.naturalprogrammer.spring.tutorial.domain.User;
import com.naturalprogrammer.spring.tutorial.domain.User.Role;
import com.naturalprogrammer.spring.tutorial.repositories.UserRepository;

@Service
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class UserServiceImpl implements UserService {

    @Autowired
    private UserRepository userRepository;

    @Override
    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
    public void signup(User user) {

        user.getRoles().add(Role.UNVERIFIED);
        userRepository.save(user);
    }
}
```

What we saw in this chapter was using the default transaction manager of Spring Boot when using Spring Data JPA. If you want to have distributed transactions or use some other transaction manager, see Spring Boot's [reference material](#).

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

15 Running Code At Startup

It's often necessary to run some startup code when an application starts. For example, in a test environment, you can add some test records. Or, you can add an admin user on new installations.

There are a couple of ways to this.

Using @PostConstruct

Spring provides various mechanisms to control the [lifecycle](#) of components. For example, you can annotate component methods with `@PostConstruct` or `@PreDestroy`, and those will be called by Spring after the component is constructed (i.e. all the injections in the component have been done) or before it is destroyed.

To see `@PostConstruct` working, add the following code in `UserServiceImpl`:

```
package com.naturalprogrammer.spring.tutorial.services;

import javax.annotation.PostConstruct;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

...

@Service
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class UserServiceImpl implements UserService {

    private static final Log log = LogFactory.getLog(UserServiceImpl.class);

    @PostConstruct
    public void postConstruct() {
        log.info("UserServiceImpl constructed");
    }

    ...

}
```

Run the application now, and you'll see the log message in your console.

So, this would be a way to run some code at startup. But a better way, which we'll discuss next, would be coding a listener, listening to an application-ready event.

Application events and listeners

For executing code at various application [events](#), Spring allows us to write *listener methods*. For example, if we write a listener for the `ApplicationReadyEvent`, that will get executed after the application is ready.

As an example, let's see how to code a listener to add an ADMIN user on new installations.

Listener methods can be coded inside any component. So, let's code our listener inside `UserServiceImpl`. For having it inside `UserServiceImpl`, Spring requires its signature to be coded in one of the interfaces that `UserServiceImpl` implements. So, first code its signature in our `UserService` interface:

```
package com.naturalprogrammer.spring.tutorial.services;

import org.springframework.boot.context.event.ApplicationReadyEvent;

import com.naturalprogrammer.spring.tutorial.domain.User;

public interface UserService {

    void signup(User user);
    void afterApplicationReady(ApplicationReadyEvent event);
}
```

Then, code it in `UserServiceImpl`, looking as below:

```
package com.naturalprogrammer.spring.tutorial.services;

...

import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.context.event.EventListener;

...

@Service
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class UserServiceImpl implements UserService {

    ...

    @Autowired
    private UserRepository userRepository;

    @Override
    @EventListener
    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
    public void afterApplicationReady(ApplicationReadyEvent event) {

        if (!userRepository.findByEmail("admin@example.com").isPresent()) {

            User user = new User();
```

```

        user.setEmail("admin@example.com");
        user.setName("First Admin");
        user.setPassword("password");

        user.getRoles().add(Role.ADMIN);

        userRepository.save(user);
    }

    ...
}

```

Notice that:

1. Event listeners are annotated with `@EventListener`.
2. They take one parameter – the event they are listening to. It's *ApplicationReadyEvent* in our case.
3. Inside the method, we check if the database already has a user with email *admin@example.com*. *userRepository.findByEmail* returns a Java 8 Optional, and its *isPresent* method is used for the checking.
4. If no user with email *admin@example.com* is found, we create one.

Instead of hardcoding the email, name and password of the first admin, why not have these as properties? To do so, update *UserServiceImpl* code as below:

```

...

import org.springframework.beans.factory.annotation.Value;

@Service
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class UserServiceImpl implements UserService {

    ...

    @Value("${app.admin.email:admin@example.com}")
    private String adminEmail;

    @Value("${app.admin.name:First Admin}")
    private String adminName;

    @Value("${app.admin.password:password}")
    private String adminPassword;

    @Override
    @EventListener
    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
    public void afterApplicationReady(ApplicationReadyEvent event) {

        if (!userRepository.findByEmail(adminEmail).isPresent()) {

            User user = new User();

```

```
        user.setEmail(adminEmail);
        user.setName(adminName);
        user.setPassword(adminPassword);

        user.getRoles().add(Role.ADMIN);

        userRepository.save(user);
    }
}

...
}
```

The **highlighted** strings, after the colon in *@Value*, supply default values in case any property isn't supplied. Consequently, even if you don't provide the properties in *application.properties*, if you run the application now, you'll have the admin user created.

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

16 Spring Security

Security, as you know, is broadly divided into two areas: *Authentication* and *Authorization*. In layman terms, *Authentication* means logging the users in, whereas *Authorization* means restricting access to parts of the application based on the rights of the users.

Let's see how to use Spring Security to set up authentication and authorization in our application.

Adding the dependency

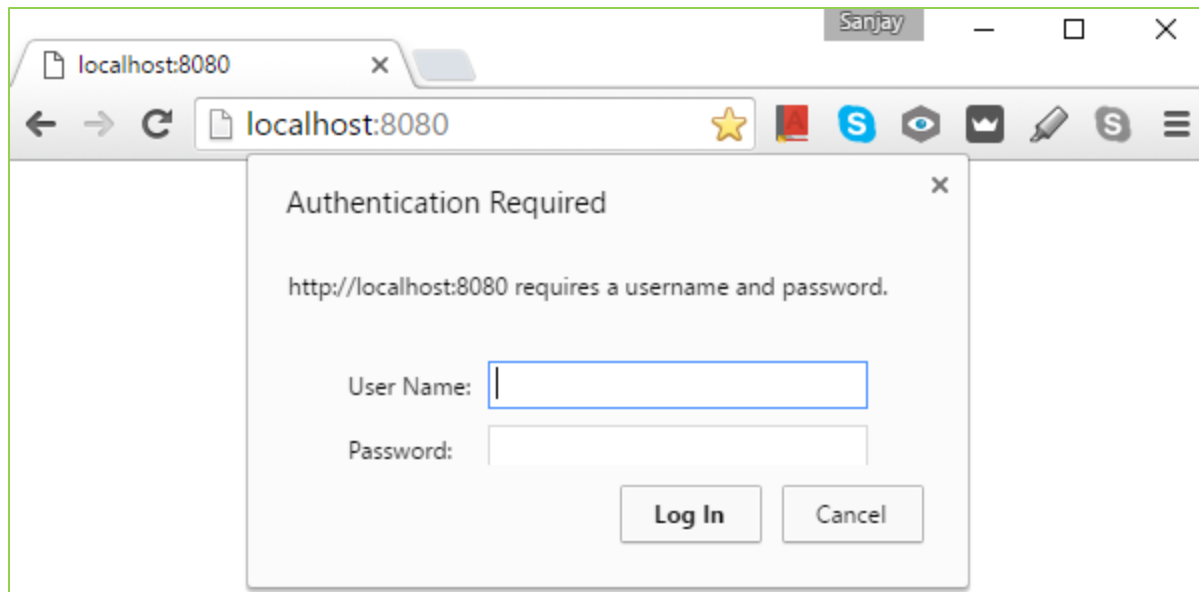
First, let's add the following dependency in *pom.xml*:

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

It will transitively put all the Spring Security JARs in our classpath.

Auto-configured basic authentication

If Spring Security is on the classpath, as we have now, Spring Boot will automatically secure all HTTP endpoints with [basic authentication](#). So, if you run the application and visit any page, you'll be asked to login, as below:



Spring Boot will also automatically set up an in-memory user store with a single user having username `user` and a random password. The password is written to console when the application starts, as below:

```
Console  [Markers] [Progress] [Problems] [History]
np-spring-tutorial - NpSpringTutorialApplication [Spring Boot App] C:\Program Files\Java\jre1.8.0_73\bin\ja
2016-03-12 13:12:09.830 INFO 10928 --- [ost-startStop-1] .e.Delegating
2016-03-12 13:12:09.830 INFO 10928 --- [ost-startStop-1] o.s.b.c.e.Ser
2016-03-12 13:12:10.144 INFO 10928 --- [ost-startStop-1] b.a.s.Authent

Using default security password: 8b5dadd-d724-4b8d-b1e9-0a7a2654969f

2016-03-12 13:12:10.224 INFO 10928 --- [ost-startStop-1] o.s.s.web.Def
```

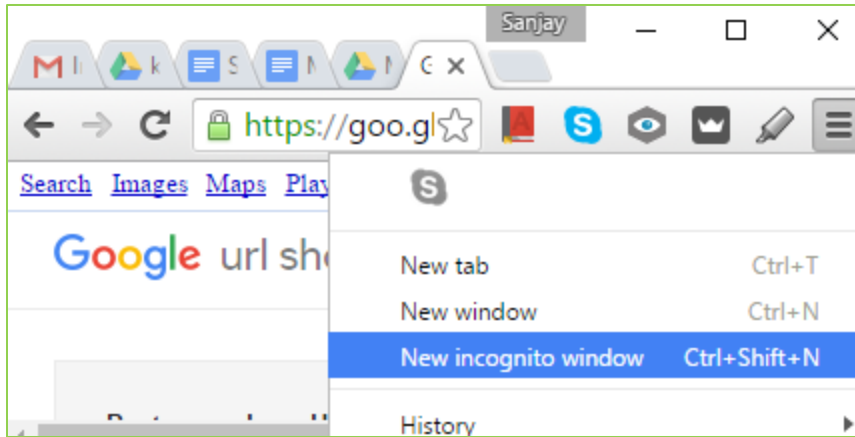
In other words, to access the application, you now have to login with username `user` and the password that is written to console.

To change these default username and password, just add the following lines in `application.properties`:

```
security.user.name: user@example.com
security.user.password: password
```

`user@example.com` and `password` can then be used for logging in.

To repeatedly test logging in, you may need logging out. But we don't have logout functionality yet. In such case, you can use a *new incognito window* of Chrome for each test case. A new incognito window can be opened by using the following menu item of Chrome:



Configuring Form Based Authentication

Although basic authentication gets auto-configured just by adding Spring Security, it doesn't seem adequate for real world web applications. Instead, [form based authentication](#) is the most popularly used authentication type.

To use form based authentication, we need to customize Spring Security's auto-configuration. It can be done by providing beans extending from `WebMvcConfigurerAdapter`. So, let's have a configuration class, say `SecurityConfig` in our `.config` package, as below:

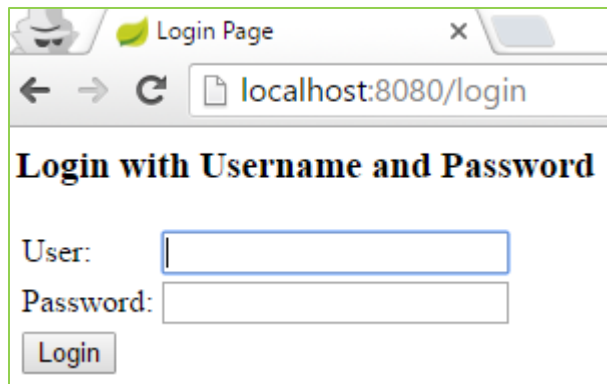
```
package com.naturalprogrammer.spring.tutorial.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

}
```

The `WebSecurityConfigurerAdapter` superclass automatically configures form based login. So, try running the application now, and you'll see a login form, as below:



A custom login form

The login page that you see above is auto generated by Spring, which doesn't look nice. Let's replace it with a custom login page.

The custom login page should contain a form posting to `/login` with `username` and `password` parameters, because Spring Security, by default, expects login requests to be posted to `/login`, with `username` and `password` parameters.

Hence, let's code a `login.jsp`, in our `src/main/webapp/WEB-INF/jsp` folder, looking as below. We'll make it available at `/login`.

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@include file="includes/header.jsp"%>

<div class="panel panel-primary">

    <div class="panel-heading">
        <h3 class="panel-title">Please sign in</h3>
    </div>

    <div class="panel-body">

        <form:form role="form">

            <div class="form-group">
                <label for="username">Email address</label>
                <input id="username" name="username"
                    type="email" class="form-control"
                    placeholder="Enter email" />
                <p class="help-block">Enter your email address.</p>
            </div>

            <div class="form-group">
                <label for="password">Password</label>
                <input id="password" name="password"
                    type="password" class="form-control"
                    placeholder="Password" />
            </div>

            <button type="submit" class="btn btn-primary">
```

```

        Sign In
    </button>
</form:form>
</div>
</div>

<%@include file="includes/footer.jsp"%>

```

Although Bootstrap makes the form look a bit scary, it's just a simple form with the *username* and *password* fields.

To make it available at */login*, add a *ViewController* in our *MvcConfig* class:

```

@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
        registry.addViewController("/login").setViewName("login");
    }

    ...
}

```

Next, we'll need to tell Spring to use our login page. For that, override the `configure(HttpSecurity http)` method of *WebSecurityConfigurerAdapter* as below:

```

...

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http
            .authorizeRequests()
                .anyRequest().authenticated()
            .and()
                .formLogin()
                    .loginPage("/login").permitAll();
    }
}

```

Points to note in the above code:

1. `authorizeRequests().anyRequest().authenticated()` tells Spring Security not to allow anonymous access to any request. So, your entire website becomes protected.
2. `.formLogin()` configures form based authentication, and returns a [FormLoginConfigurer](#), which is used for further customization.

3. `.loginPage("/login").permitAll()` tells Spring Security to use our custom login page available at `/login`, and permit all users, including anonymous ones, to access that page.

Try running the application now, and you'll see our custom login page working properly.

Logging out

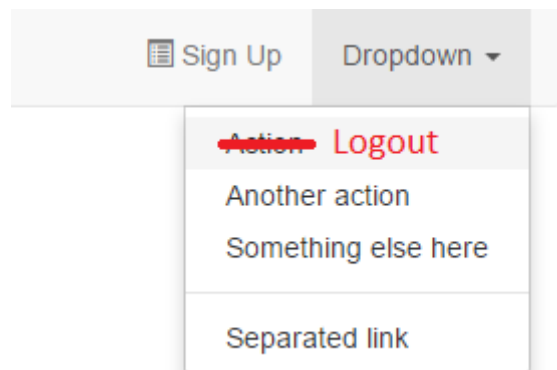
So, how to log out?

If we call `http.logout()` in the `configure` method above, Spring will give us a `/logout` endpoint. Posting to that endpoint will log a user out.

Precisely, first add `.and().logout().permitAll()` in `configure`, as below:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .anyRequest().authenticated()
        .and()
            .formLogin()
                .loginPage("/login").permitAll()
        .and()
            .logout().permitAll();
}
```

Then, we need to give a link, say somewhere on the navigation bar, clicking on which we should send a POST request to `/logout`. So, let's plan to just replace the *Action* link of the second dropdown with *Logout*, as below:



Note that there are two Dropdowns in the navigation bar. Don't update the first one, and then keep wondering why the second one didn't update in the UI.

Links do not send POST requests, and so we can't just set the `href` of the *Logout* link to `/logout`. Instead, we need to have a form with action as `/logout`, and then POST that form via JavaScript. Precisely, replace the *Action* link in `header.jsp` as follows:

```
<li><a href="#">Action</a></li>
```

```

<li>
    <c:url var="logoutUrl" value="/logout" />
    <form:form id="logoutForm" action="${logoutUrl}">
    </form:form>
    <a href="#" onclick="document.getElementById('logoutForm').submit()">
        <span class="glyphicon glyphicon-log-out"></span> Sign out
    </a>
</li>

```

We used curl to build the `/logout` link – just a matter of good practice.

Logging out should now work - test it out.

Displaying a logout message

Notice that, on logging out, Spring redirects the user to `/login?logout` page. The `?logout` parameter helps us display a logout message. So, add the following snippet as the first element inside the panel body of `login.jsp`:

```

<div class="panel-body">

    <c:if test="${param.logout != null}">
        <div class="alert alert-success">
            You have been logged out
        </div>
    </c:if>

    ...

```

On logging out, you'll now see "You have been logged out."

Displaying login errors

Errors do occur while logging in. For example, users often provide wrong username or password. In such cases, Spring saves the exception in a `SPRING_SECURITY_LAST_EXCEPTION` session attribute, and redirects the user to `/login?error`.

That `?error` parameter, along with `SPRING_SECURITY_LAST_EXCEPTION`, can help displaying an error message to the user. So, add the following snippet in `login.jsp`, below the logout message:

```

<div class="panel-body">

    <c:if test="${param.logout != null}">
        <div class="alert alert-success">
            You have been logged out
        </div>
    </c:if>

    <c:if test="${param.error != null}">
        <div class="alert alert-danger">
            Failed to login.
        </div>
    </c:if>

```

```

        <c:if test="${SPRING_SECURITY_LAST_EXCEPTION != null}">
            Reason: <c:out value="${SPRING_SECURITY_LAST_EXCEPTION.message}" />
        </c:if>
    </div>
</c:if>

```

Try providing wrong credentials now, and you'll see the error message.

If you want Spring to redirect to a different error URL, or handle login errors in a different way, you can easily do so using [FormLoginConfigurer](#) methods, viz. *failureUrl* or *failureHandler*. For example, when coding REST APIs, you would like to respond login failures with a 403 error instead of a redirection. [Module III](#) covers it in details.

Mapping User entities to Spring Security users

As we saw, Spring Boot auto-configures an in memory user store by default. But, real world applications will like to use persistent user stores. A common practice actually is to use JPA entities for storing the users.

So, let's see how to use our User entities as Spring Security users.

Implementing UserDetails

To hold the data about a logged-in user, Spring Security uses a [UserDetails](#) object. So, let's have our *User* class implement *UserDetails*:

```

...

import org.springframework.security.core.userdetails.UserDetails;

@Entity
@Table(name="usr", indexes = {
    @Index(columnList = "email", unique=true)
})
public class User implements UserDetails {

```

This'll need us to implement *getUsername*, *getPassword*, *getAuthorities* and a few more methods of *UserDetails*, as elaborated below.

getUsername

getUsername should return the login id of the user. Let's use our *email* field as the login id. So, code *getUsername* as below:

```

@Override
public String getUsername() {
    return email;
}

```



```
}
```

getPassword

This would be nothing but the getter of our *password* property, which we already have. Just annotate that with `@Override`:

```
@Override
public String getPassword() {
    return password;
}
```

getAuthorities

Spring Security calls `getAuthorities` to know about the roles of the logged in user, so that access to various parts of the application can be restricted based on those roles.

It should return a collection of [GrantedAuthorities](#). Let's override it, and return a collection of [SimpleGrantedAuthorities](#) made out of our roles:

```
...

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;

...

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {

    Collection<GrantedAuthority> authorities =
        new HashSet<GrantedAuthority>(roles.size());

    for (Role role: roles)
        authorities.add(new SimpleGrantedAuthority(
            "ROLE_" + role.name()));

    return authorities;
}
```

SimpleGrantedAuthority is an implementation of *GrantedAuthority*, provided by Spring Security. Its constructor expects a role as a string. When creating *SimpleGrantedAuthority*s, we are prepending our roles with "ROLE_", because Spring Security wants its role names to begin with "ROLE_".

Although by calling *getAuthorities* Spring Security can know the roles of the logged in user, we are yet to see how to restrict access to parts of our application using that.

Other methods

We also need to implement `isAccountNonExpired`, `isAccountNonLocked`, `isCredentialsNonExpired` and `isEnabled`. Let's just return `true` from all of these:

```

@Override
public boolean isAccountNonExpired() {
    return true;
}
@Override
public boolean isAccountNonLocked() {
    return true;
}
@Override
public boolean isCredentialsNonExpired() {
    return true;
}
@Override
public boolean isEnabled() {
    return true;
}

```

Refer to `UserDetails`' [Javadoc](#) to know more about these.

Also, by implementing `UserDetails`, our `User` class has now become serializable. Consequently, you may be seeing a warning saying “The serializable class `User` does not declare a static final `serialVersionUID` field of type `long`.” To fix it, if you are using STS, you can choose the “Add generated serial version ID” context menu option, which will add a generated `serialVersionUID` to the `User` class.

Configuring a UserDetailsService

When a user logs in, Spring fetches the corresponding user entity by using a [UserDetailsService](#) that we are supposed to configure. So, create a new `@Service` class in the `.services` folder, say `UserDetailsServiceImpl`, implementing `UserDetailsService`:

```

package com.naturalprogrammer.spring.tutorial.services;

import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.stereotype.Service;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

}

```

The compiler will ask you to implement its `loadUserByUsername` method, which should return a `UserDetails`, given a `username`. In our case, we have mapped `UserDetails` to our `User` entity and `username` to `email`, and so, our `loadUserByUsername` should return a `User`, given an email, as below:

```

@Autowired
private UserRepository userRepository;

@Override
public UserDetails loadUserByUsername(String username)
    throws UsernameNotFoundException {

```

```

    User user = userRepository.findByEmail(username)
        .orElseThrow(() -> new UsernameNotFoundException(username));

    return user;
}

```

To find the user, note that we are using the `findByEmail` method of the `userRepository` that we have coded [earlier](#). We throw a `UsernameNotFoundException` if the user is not found.

Configuring AuthenticationManager

When a user logs in, Spring uses an [AuthenticationManager](#) to verify the authentication details, e.g. username and password. It's the `AuthenticationManager` that will use our `UserDetailsService` for fetching the user data.

Spring Security 4.1 (Spring Boot 1.4) onwards, if Spring Security finds a `UserDetailsService` in the application context, it automatically wires that to its `AuthenticationManager`. So, because our `UserDetailsServiceImpl` implements `UserDetailsService`, we should now be able to login using the *email/password* of users stored in our `usr` table without any further configuration.

Test it out. Specifically, try logging in with *admin@example.com/password* – that's the *email/password* of the user that [we create](#) when the application starts.

Note: Prior to Spring Boot 1.4, we had to manually wire our `UserDetailsService` into the `AuthenticationManager` by adding the following code in our `SecurityConfig`:

```

...

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.config.annotation.authentication.builders.Authen
ticationManagerBuilder;
import org.springframework.security.core.userdetails.UserDetailsService;

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private UserDetailsService userDetailsService;

    @Override
    protected void configure(AuthenticationManagerBuilder builder)
        throws Exception {

        builder.userDetailsService(userDetailsService);
    }

    ...

```

To know more about `AuthenticationManager` and other Spring Security technical details, refer its [documentation](#). But, we'd suggest not to get too details into it now – it may boggle you if you are a beginner.

Getting the logged in user

When a user logs in, Spring Security builds an [Authentication](#) object. The Authentication object, apart from containing the credentials, authorities etc., also contains the UserDetails as its **principal** property. In other words, we can get the logged in user by accessing the *principal* property of Spring Security's authentication object.

Spring Security's authentication object can be accessed in Java using:

```
SecurityContextHolder.getContext().getAuthentication()
```

So, to access the currently logged in user, let's code a static utility method, say in *MyUtil* class, as below:

```
package com.naturalprogrammer.spring.tutorial.util;

...

import org.springframework.security.core.Authentication;
import org.springframework.security.core.context.SecurityContextHolder;

import com.naturalprogrammer.spring.tutorial.domain.User;
...

@Component
public class MyUtil {

    ...

    public static User getUser() {

        // get the authentication object
        Authentication auth = SecurityContextHolder
            .getContext().getAuthentication();

        // get the user from the authentication object
        if (auth != null) {
            Object principal = auth.getPrincipal();
            if (principal instanceof User) {
                return (User) principal;
            }
        }
        return null;
    }
}
```

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

17 Security Tag Library

We now have the *login* and *logout* functionality, but our UI needs some improvements. Precisely, let's do the following:

1. Beside the Sign Up link, let's add a *Sign In* link, linked to */login*.
2. When someone is logged in, her name should be visible in the navigation bar. Specifically, let's replace the "Dropdown" in the top-right corner with the name of the logged in user.
3. Let *Sign Up* and *Sign In* be visible only when no one has logged in. Similarly, let the name of the user be visible when someone has logged in.

Adding a Sign In link

Adding a *Sign In* link is simple – in *header.jsp*, just duplicate the *Sign Up* snippet and do some fixes, as highlighted below:

```
...  
<ul class="nav navbar-nav navbar-right">  
  <li>  
    <a href="      <span class="glyphicon glyphicon-list-alt"></span> Sign Up  
    </a>  
  </li>  
  <li>  
    <a href="      <span class="glyphicon glyphicon-log-in"></span> Login  
    </a>  
  </li>  
</ul>  
...
```

Using Spring Security tag library

Whereas adding a Sign In link was simple, for the other requirements, Spring's Security's JSP [tag library](#) will come in handy.

To use it, first add the following dependency to *pom.xml*:

```
<dependency>  
  <groupId>org.springframework.security</groupId>  
  <artifactId>spring-security-taglibs</artifactId>  
</dependency>
```

Then, declare the taglib in *header.jsp*:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags" %>
<!DOCTYPE html>
...
```

Displaying logged-in user's name

Let's now replace "Dropdown" with the name of the logged in user.

To access the logged in user data, our [getUser\(\)](#) utility method can be used. But, in JSPs, using the `authentication` tag of Spring Security tag library is easier. It allows access to the current Authentication object. Thus, let's update our *header.jsp* as below:

```
...
<li class="dropdown">
  <a href="#" class="dropdown-toggle" data-toggle="dropdown"
    role="button" aria-haspopup="true" aria-expanded="false">
    Dropdown
    <span class="glyphicon glyphicon-user"></span>
    <span class="glyphicon glyphicon-user" property="principal.name" />
    <span class="caret"></span>
  </a>
  <ul class="dropdown-menu">
    <li>
      <c:url var="logoutUrl" value="/logout" />
    </li>
  </ul>
</li>
...
```

As you see, we have replaced "Dropdown" with a couple of lines. Whereas the first line is just for showing a Bootstrap icon, the highlighted line uses the `authentication` tag to access the Authentication object. As you know, the principal property of the Authentication object will be the logged in user, and so *principal.name* will be the name of the user that we want to show.

Showing *Sign Up* and *Sign In* only when no one is logged in

Spring Security tag library has an `authorize` tag, which can be used to conditionally render content based on some security conditions. It has an `access` attribute, which takes a [web-security expression](#) that is used to determine whether its contents should be rendered. For example, enclose our *Sign Up* and *Sign In* links with an `authorize` tag, as below:

```
...
```

```

<sec:authorize access="isAnonymous()">
  <li>
    <a href="

```

Notice that here we have set the value of the `access` attribute to `isAnonymous()`. Consequently, *Sign Up* and *Sign In* will be rendered only when no one is logged in.

The web security expression that `access` can take should be a [Spring EL](#) expression, composed of the methods listed [here](#).

Displaying name only someone has logged in

To display some content only when someone has logged in, we can use the same `authorize` tag. So, let's enclose the entire second dropdown menu with an `authorize` tag, so that we can use that menu for profile related functionality, to be displayed only when someone has logged in:

```

<sec:authorize access="isAuthenticated()">
  <li class="dropdown">
    <a href="#" class="dropdown-toggle" data-toggle="dropdown"
      role="button" aria-haspopup="true" aria-expanded="false">
      Dropdown
      <span class="glyphicon glyphicon-user"></span>
      <sec:authentication property="principal.name" />
      <span class="caret"></span>
    </a>
  </li>
  ...
</sec:authorize>

```

Time to check the application out. Try logging in with `admin@example.com/password`, and then logout, and see how the navigation bar changes.

But, did you notice that, when not logged in, if you try to visit the home or `/signup` page, just the `/login` page appears? We'll fix that in the next chapter.

Source code so far

- [Browse online](#)

- [Download zip](#)
- [See changes](#)

18 Authorization

Whereas *authentication* means logging the users in, *authorization* means checking the rights of the users before granting access to parts of the application.

Spring enables you to enforce authorization both at request level and method level.

Request level authorization

Spring Security can be configured to restrict access to the URLs based on who has logged in. For this purpose, `authorizeRequests()` in the `configure(HttpSecurity http)` method of our `SecurityConfig` can be used. Let's change that snippet as below:

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http
        .authorizeRequests()
            .1mvcMatchers(HttpMethod.GET,
                "/").permitAll()
            .2mvcMatchers(
                "/signup",
                "/forgot-password",
                "/reset-password/*").permitAll()
            .3mvcMatchers(HttpMethod.GET, "/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated()

        ...
}
```

Note that we are calling three `.mvcMatchers`, followed by `.anyRequest().authenticated()`. Each `.mvcMatchers` define some access rules for some URL patterns. The URL patterns follow [Spring MVC request mapping patterns](#), which are similar to [ANT path patterns](#).

Whereas the code above looks mostly self-explanatory, notice the following:

1. The first *mvcMatcher* takes `HttpMethod.GET` as the first parameter. That means, it's applicable only to GET requests, and not to any other, e.g. POST requests.
2. The third *mvcMatcher* restricts access to all URLs beginning with `/admin` only to users who have ADMIN roles.
3. Like `permitAll` or `hasRole` used above, Spring provides many more methods – [here](#) is the complete list.

In summary, the above configuration will allow access to home, `/signup`, `/forgot-password`, and `/reset-password/*` pages to anyone, pages beginning with `/admin` only to administrators, and all other pages only to logged in users.

An ADMIN page

The third *MvcMatcher* restricts access to all URLs beginning with */admin* only to users who have ADMIN roles. To test it out, let's create a dummy *admin.jsp*, in *src/main/webapp/WEB-INF/jsp*, as below:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@include file="includes/header.jsp"%>

You are an ADMIN!

<%@include file="includes/footer.jsp"%>
```

Then, add a ViewController for */admin*, as below:

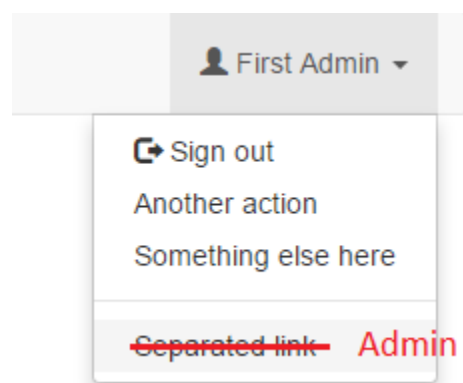
```
@Configuration
public class MvcConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        ...
        registry.addViewController("/admin").setViewName("admin");
    }
    ...
}
```

Test the application out, and you'll see that */admin* can be visited only if you login as admin. You can use the credentials of the [first admin user](#) for the testing.

Showing an Admin link

Let's now add an *Admin* link to our second dropdown. Specifically, let's change our UI as below:



To do so, update *header.jsp* as below:

```
...
<li role="separator" class="divider"></li>
<li><a href="#">Separated link</a></li>
<li><a href="/admin">Admin</a></li>
```

```

    </ul>
  </li>
</sec:authorize>
...

```

How about letting the link visible only when an ADMIN has logged in? That's easy with the *authorize* tag. So, enclose the *Admin* link, along with the divider, with an *authorize* tag, as below:

```

<sec:authorize access="hasRole('ADMIN') ">
  <li role="separator" class="divider"></li>
  <li><a href="/admin">Admin</a></li>
</sec:authorize>

```

Test it out.

Method level authorization

Apart from restricting URLs, we can also restrict access to our bean methods. To enable this with *pre/post* annotations, we need to annotate some configuration class, say the *SecurityConfig* class, with *@EnableGlobalMethodSecurity* as below:

```

@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

```

For restricting access to a bean method, we can then annotate that with pre/post [annotations](#), like *@PreAuthorize*. See this example:

```

@PreAuthorize("isAuthenticated()")
public void someMethod(String verificationCode) {

```

@PreAuthotize takes a *Spring EL* expression, which is *isAuthenticated()* in the above case.

[Here](#) is a list of methods that can be used in the expression.

We are not going to use method level authorization in this book, but [Module III](#) uses it extensively.

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

19 More Security

In this chapter, we'll discuss some more real-world security functionalities.

Encrypting Passwords

We're now storing passwords as plaintext in the *usr* table. To store those encrypted, we need to do the following:

1. When a user signs up or changes password, encrypt the password before storing it.
2. When someone signs in, match the given password against the encrypted one (using some algorithm).

For this, we need a [PasswordEncoder](#) object. Spring provides a few concrete implementations of `PasswordEncoder`, among which `BCryptPasswordEncoder` is the most preferred one. So, let's create a `BCryptPasswordEncoder` bean, say in our `SecurityConfig` class, as below:

```
package com.naturalprogrammer.spring.tutorial.config;

...
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    ...
}
```

Then, let's update our signup code in `UserServiceImpl` as below:

```
...

import org.springframework.security.crypto.password.PasswordEncoder;

@Service
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class UserServiceImpl implements UserService {

    ...

    @Autowired
    private PasswordEncoder passwordEncoder;
```

```

...

@Override
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void signup(User user) {

    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.getRoles().add(Role.UNVERIFIED);
    userRepository.save(user);
}
}

```

We just injected our passwordEncoder and used that for encrypting the password before saving to database.

Also update the code for creating the first admin as below:

```

@Service
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class UserServiceImpl implements UserService {

    ...

    public void afterApplicationReady(ApplicationReadyEvent event) {

        ...
        user.setPassword(adminPassword);
        user.setPassword(passwordEncoder.encode(adminPassword));
        ...
    }
}

```

When someone logs in, Spring Security's authentication manager, on finding a PasswordEncoder bean in the application context, would use that to match the given password against the encrypted one in the database.

So, test it out now. Before testing, if you haven't set [spring.jpa.hibernate.ddl-auto](#) as `create`, remember to manually delete the old user records, which will already have plain passwords stored.

Matching a plain password against an encrypted one

For matching a plain password against an encrypted one, passwordEncoder's `matches` method can be used, as below:

```

if (passwordEncoder.matches(plainPassword, encryptedPassword))
    ...

```

Remember me

You must be seeing the *Remember Me* check box on almost every login form on the web. Clicking on it, the user is remembered by the browser across sessions. This is typically accomplished by sending a cookie to the browser, with the cookie being detected during future sessions and causing automated login to take place.

Spring Security provides the necessary hooks for these operations to take place, and has two concrete remember-me implementations. Between these, [TokenBasedRememberMeServices](#) is more popular and simple, which hashes login data as a cookie-based token.

To tell our application to use *TokenBasedRememberMeServices*, add the following to *SecurityConfig*:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Value("${rememberMeKey:topSecret}")
    private String rememberMeKey;

    @Autowired
    private UserDetailsService userDetailsService;

    ...

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http

            ...

            .logout()
                .permitAll()
            .and()
                .rememberMe()
                    .key(rememberMeKey)
                    .rememberMeServices(new
TokenBasedRememberMeServices(rememberMeKey, userDetailsService));
    }
}
```

As you see, it needs a secret key, which we have named as `rememberMeKey`. It's used for encrypting the token. You can define that in *application.properties*. It'll default to `topSecret` if not found in application properties, as you see above.

It also needs a reference to the *UserDetailsService* bean, which we have autowired.

With the above configuration in place, if a *remember-me* parameter with `value true, yes, on` or `1` is sent along with a login request, the user is remembered. That means, we just need to add a check box with name *remember-me* in our *login.jsp*:

```

...
    <div class="form-group">
        <div class="checkbox">
            <label>
                <input name="remember-me" type="checkbox">
                Remember me
            </label>
        </div>
    </div>

    <button type="submit" class="btn btn-primary">
        Sign In
    </button>
</form:form>
</div>
</div>

<%@include file="includes/footer.jsp"%>

```

Test it now, and it should work. I prefer using Firefox rather than Chrome for testing *remember-me*, because Chrome is sometimes smart enough to remember the user even if you don't check the checkbox.

By default, users will be remembered for two weeks. You can customize this and many other attributes just by changing various properties of [TokenBasedRememberMeServices](#).

Signing in and out programmatically

Sometimes, we need to sign in or out a user programmatically. For example, when a user signs up, we may like to sign her in as well.

That can be done by simply replacing the authentication token of the security context.

Specifically, a `login` utility method would look as below:

```

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

...

public static void login(User user) {
    // make the authentication object
    Authentication authentication =
        new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());

    // put that in the security context
    SecurityContextHolder.getContext().setAuthentication(authentication);
}

```

The first statement builds a *UsernamePasswordAuthenticationToken* out of the given user, and the second one puts that in the security context.

Logging out is simpler – just set the authentication as `null`:

```
public static void logout() {  
    SecurityContextHolder.getContext().setAuthentication(null);  
}
```

Put both these methods in *MyUtil*; they will be useful.

So, to sign in a user after she signs up, add a line to our signup method in *UserServiceImpl*:

```
@Service  
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)  
public class UserServiceImpl implements UserService {  
  
    ...  
  
    @Override  
    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)  
    public void signup(User user) {  
  
        user.setPassword(passwordEncoder.encode(user.getPassword()));  
        user.getRoles().add(Role.UNVERIFIED);  
        userRepository.save(user);  
        MyUtil.login(user);  
    }  
}
```

Test it out, it should work.

But wait – there is a subtle issue!

Because the *signup* method is transactional, the user will be saved only after the method finishes executing. And, if something goes wrong while saving, the transaction will be rolled back. Consequently, the user will not be saved, although she will already have been logged in!

To prevent such issues, we can tell Spring to execute the login code only after the commit succeeds. That can be done this way:

```
import org.springframework.transaction.support.TransactionSynchronizationAdapter;  
import org.springframework.transaction.support.TransactionSynchronizationManager;  
  
...  
  
@Override  
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)  
public void signup(User user) {  
  
    user.setPassword(passwordEncoder.encode(user.getPassword()));  
    user.getRoles().add(Role.UNVERIFIED);  
    userRepository.save(user);  
  
    TransactionSynchronizationManager.registerSynchronization(  
        new TransactionSynchronizationAdapter() {  
            @Override  
            public void afterCommit() {  
                MyUtil.login(user);  
            }  
        }  
    );  
}
```



```

TransactionSynchronizationManager.registerSynchronization(
    new TransactionSynchronizationAdapter() {
        @Override
        public void afterCommit() {
            MyUtil.login(user);
        }
    }
);
}

```

We basically have wrapped the login code in a `TransactionSynchronizationAdapter`, and registered that in the `TransactionSynchronizationManager`. `TransactionSynchronizationAdapter` has many more override-able methods – give a look at its [Javadoc](#).

Too much boilerplate code, right? How about we move the boilerplate code to a utility method, and just call that with the our code wrapped in a [Runnable](#)?

Specifically, create the following method in *MyUtil*:

```

public static void afterCommit(Runnable runnable) {
    TransactionSynchronizationManager.registerSynchronization(
        new TransactionSynchronizationAdapter() {
            @Override
            public void afterCommit() {
                runnable.run();
            }
        }
    );
}

```

signup method now becomes simpler:

```

@Override
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void signup(User user) {

    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.getRoles().add(Role.UNVERIFIED);
    userRepository.save(user);

    MyUtil.afterCommit(() -> {
        MyUtil.login(user);
    });
}

```

Source code so far

- [Browse online](#)
- [Download zip](#)

- [See changes](#)

20 Email Verification

On signup, a verification link should be mailed to the user. Clicking on the link, the user should get verified. We're going to code that in this chapter.

Specifically, we are going to do the following:

1. On signup, we'll generate a *unique verification code* and store that in the *User* entity – in a `verificationCode` field. Then, we'll send a mail to the user with a link as below:

```
http://application-url/users/unique-verification-code/verify
```

2. When the user will click on the link, we'll match the verification code against her database record, and remove her UNVERIFIED role.

Using the MockMailSender

For sending mails, our *MailSender* service can be used. Let's plan to use the *MockMailSender* rather than the *SmtplibMailSender* in this development phase. Commenting the `spring.mail.host` property in *application.properties* will configure *MockMailSender*, as you know. So, do that:

```
...  
# spring.mail.host = smtp.gmail.com  
spring.mail.username = your_gmail_id@gmail.com  
spring.mail.password = an_application_password  
...
```

Adding a verificationCode field to the User class

To hold the unique verification code that we are going to generate, create a `verificationCode` field in the *User* class, as below:

```
@Column(length = 36, unique=true)  
private String verificationCode;  
  
public String getVerificationCode() {  
    return verificationCode;  
}  
  
public void setVerificationCode(String verificationCode) {  
    this.verificationCode = verificationCode;  
}
```

We have set the length of the column to 36, because we plan to store UUIDs in it.

Updating signup service

Next, let's update the *signup* service. Specifically, add two lines as below:

```
@Override
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void signup(User user) {

    user.setPassword(passwordEncoder.encode(user.getPassword()));
    user.getRoles().add(Role.UNVERIFIED);
    1user.setVerificationCode(UUID.randomUUID().toString());
    userRepository.save(user);

    MyUtil.afterCommit(() -> {

        MyUtil.login(user);
        2sendVerificationMail(user);
    });
}
```

The first line assigns a random *java.util.UUID* code to the *verificationCode* field, whereas the second sends the verification mail upon successful commit.

As you see, for sending the mail, we have used a helper method `sendVerificationMail(user)`, which we'll code in a moment.

Mail subject and body

Before coding `sendVerificationMail(user)`, first let's add two lines in *messages.properties*, for the *subject* and the *body* of the mail, as below:

```
verifySubject: Please verify your email id
verifyEmail: Hi,<br/><br/>Your email id at XYZ is unverified. \
Please click the link below to get verified:<br/><br/>{0}<br/><br/>
```

Notice that, in *verifyEmail*, we have used the placeholder `{0}` for the link. Also notice how we have used backslash to break a long message into two lines. Backslash must be the last character in such case – even whitespaces aren't allowed after it.

Also, enhance the *signupSuccess* message as below:

```
signupSuccess: Signup succeeded! Please check your mail to verify yourself.
```

The application URL

Let the verification link be of the format:

```
http://application-url/users/verification-code/verify
```

There are two variables here, viz. `application-url` and `verification-code`. Whereas *verification code* is available in the user entity, where to get the *application URL*?

You may argue that the *application URL* can be obtained from the HTTP request. But, what if you decide to use a mobo-client in future? Or you need to send some mails from a background process? So, in practice, it's wise to define the *application URL* as an application property. Hence, add the following line to *application.properties*:

```
application.url: http://localhost:8080
```

We are now ready to code the *sendVerificationMail* helper method.

sendVerificationMail

The *sendVerificationMail* helper method can be coded in *UserServiceImpl* as below:

```
import javax.mail.MessagingException;
import org.apache.commons.lang3.exception.ExceptionUtils;
import com.naturalprogrammer.spring.tutorial.mail.MailSender;

...

@Value("${application.url}")
private String applicationUrl;

@Autowired
private MailSender mailSender;

...

private void sendVerificationMail(User user) {
    try {

        // make the link
        String verifyLink = applicationUrl
            + "/users/" + user.getVerificationCode() + "/verify";

        // send the mail
        mailSender.send(user.getEmail(),
            MyUtil.getMessage("verifySubject"),
            MyUtil.getMessage("verifyEmail", verifyLink));

    } catch (MessagingException e) {
        // In case of exception, just log the error and keep silent
        log.error(ExceptionUtils.getStackTrace(e));
    }
}
```

Notice that, in case of *MessagingException*, we are logging the stack trace by using `ExceptionUtils`. That's a utility class from the [Apache Commons Lang](#) library. So, include that dependency in your *pom.xml*:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>RELEASE</version>
</dependency>
```

Showing a unverified warning

How about we keep showing a unverified warning box when a unverified user has logged in? To do so, add the following snippet to *header.jsp*, just above the flash message:

```
...
<sec:authorize access="hasRole('UNVERIFIED') ">
  <div class="alert alert-warning alert-dismissible">
    <button type="button" class="close" data-dismiss="alert" aria-label="Close">
      <span aria-hidden="true">&times;</span>
    </button>
    <spring:message code="unverified" />
  </div>
</sec:authorize>

<c:if test="${not empty flashMessage}">
  ...
```

The above code checks if the user has UNVERIFIED code, and if so, displays a Bootstrap warning alert. The message will be picked up from *messages.properties*, having key `unverified`. So, in *messages.properties*, add the following message:

```
unverified: Your email id is unverified
```

Verifying the user

To verify a user when she clicks on the verification link, the first step will be to code a handler method to handle requests of the format

```
/users/verification-code/verify
```

Coding the controller

Let's create a new controller class, say `UserController`, and put our handler method in that, as below:

```
package com.naturalprogrammer.spring.tutorial.controllers;
```

```

...

@Controller
@RequestMapping("/users")
public class UserController {

    @GetMapping("/{verificationCode}/verify")
    public String verify(@PathVariable String verificationCode) {
        ...
    }
}

```

Notice how we have provided the *path arguments* to both the `@RequestMapping` at class level and `@GetMapping` and method level. The final path will be a concatenation of both, i.e. `/users/{verificationCode}/verify`.

As a security measure, let's say a user must be logged in to get verified. In fact, Spring Security will automatically do that for us as a result of our [authorization configuration](#). If a user isn't logged in, Spring Security will first redirect her to the login page.

Populate the handler method as below:

```

@Controller
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("/{verificationCode}/verify")
    public String verify(@PathVariable String verificationCode,
        RedirectAttributes redirectAttributes) {

        userService.verify(verificationCode);
        MyUtil.flash(redirectAttributes,
            "success", "verificationSuccess");

        return "redirect:/";
    }
}

```

After calling the service method that we are yet to code, it flashes a success message to the user, and redirects her to the home page. So, define a `verificationSuccess` message in `messages.properties`:

```
verificationSuccess: Verification succeeded!
```

Coding the service method

Let's now code the service method. First, put its signature in `UserService`, as below:

```
public interface UserService {
```

```

    void afterApplicationReady(ApplicationReadyEvent event);
    void signup(User user);
    void verify(String verificationCode);
}

```

Then, code the method in *UserServiceImpl*, as below:

```

@Override
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void verify(String verificationCode) {

    // get the current-user from the session
    User currentUser = MyUtil.getUser();

    // fetch a fresh copy from the database
    User user = userRepository.findOne(currentUser.getId());

    // ensure that the user is unverified
    MyUtil.validate(user.getRoles().contains(Role.UNVERIFIED),
        "alreadyVerified");

    // ensure that the verification code of the user matches
    // with the given one
    MyUtil.validate(verificationCode.equals(user.getVerificationCode()),
        "wrongVerificationCode");

    makeVerified(user); // make him verified
    userRepository.save(user);

    // after successful commit,
    MyUtil.afterCommit(() -> {

        // Re-login the user, so that the UNVERIFIED role is removed
        MyUtil.login(user);
    });
}

private void makeVerified(User user) {
    user.getRoles().remove(Role.UNVERIFIED);
    user.setVerificationCode(null);
}

```

Looks like a lot of code, but it's actually simple.

We are first getting the currently logged in user, and then fetching a fresh copy from the database. The `findOne` method of *userRepository*, inherited from Spring Data's [CrudRepository](#), is used for fetching an entity by its primary key.

Then, we are doing a couple of validations. The `validate` method, which we are yet to code, will check for the given condition, and throw a `RuntimeException` if the condition is not met. The message of the exception will be picked from *messages.properties*, using the given message key.

After the validations succeed, we have used the `makeVerified` method to remove the UNVERIFIED role from the user and reset the verification code field.

We are then saving the user, and on successful commit, calling `MyUtil.login` to replace the session user with the updated one.

Coding the validate utility

So, code a validate method in the `MyUtil` class, as below:

```
public static void validate(boolean valid,
    String messageKey, Object... messageArguments) {

    if (!valid)
        throw new RuntimeException(getMessage(messageKey, messageArguments));
}
```

As we told, it just throws a `RuntimeException` if the given condition is invalid. The message of the exception is picked from `messages.properties` by using the `getMessage` utility method.

Put the following messages in `messages.properties`, and coding the verification process is complete!

```
alreadyVerified: Already verified
wrongVerificationCode: Wrong verification code
```

Resending verification mail

Users should be able to request for resending the verification mail if they miss the first one. So, let's add a link to the unverified message in `header.jsp`, clicking on which, the user should get the verification mail again. Let's have the link of the following format:

```
/users/{id}/resend-verification-mail
```

Modifying unverified message

The first step will be to modify `unverified` message in `messages.properties`, as below:

```
unverified: Your email id is unverified. \
<a href="/users/{0}/resend-verification-mail">Click here</a> \
to get the verification mail again.
```

The placeholder `{0}` above should be replaced with the id of the logged in user. Hence, modify `header.jsp` as below:

```
<sec:authorize access="hasRole('UNVERIFIED') ">
    <div class="alert alert-warning alert-dismissible">
        <button type="button" class="close" data-dismiss="alert" aria-label="Close">
```

```

        <span aria-hidden="true">&times;</span>
    </button>
    <sec:authentication property="principal.id" var="currentUserId" />
    <spring:message code="unverified" arguments="{currentUserId}" />
</div>
</sec:authorize>

<c:if test="${not empty flashMessage}">

```

See how the *authentication* tag of Spring Security tag library is used for putting the current user id into a JSP variable. Then, that variable is used as the argument to the *message* tag of the Spring tag library.

Coding the handler

The handler for resending verification mail can be coded, in *UserController*, as below:

```

@GetMapping("/users/{id}/resend-verification-mail")
public void resendVerificationMail(@PathVariable long userId) {

    ...

}

```

As you see, the ID of the user is being received as a path variable. It can be used to fetch the user from the database, using the *userRepository.findOne(id)*, as you know.

But there is a more concise way to fetch an entity, given an *ID* as a path variable. Annotate some configuration class, say *NpSpringTutorialApplication*, with *@EnableSpringDataWebSupport*:

```

@SpringBootApplication
@EnableSpringDataWebSupport
public class NpSpringTutorialApplication {

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

}

```

We can now map a path variable straight to a domain entity, as below:

```

@GetMapping("/users/{id}/resend-verification-mail")
public void resendVerificationMail(@PathVariable("id") User user) {

    ...

}

```

Spring MVC will then automatically fetch the entity using the path variable, by using a *DomainClassConverter*. See the [documentation](#) of *@EnableSpringDataWebSupport* to know more about it.

The body of the handler can just be calling a service method, followed by a flash and then a redirect. The complete code will look as below:

```

@GetMapping("/{id}/resend-verification-mail")
public String resendVerificationMail(@PathVariable("id") User user,
    RedirectAttributes redirectAttributes) {

    userService.resendVerificationMail(user);

    MyUtil.flash(redirectAttributes,
        "success", "verificationMailSent");

    return "redirect:/";
}

```

Before moving on to the service method, add the `verificationMailSent` message to `messages.properties`:

```
verificationMailSent: Verification mail sent
```

Coding the service

Let's begin coding the service method by adding its signature to `UserService`:

```

public interface UserService {

    void afterApplicationReady(ApplicationReadyEvent event);
    void signup(User user);
    void verify(String verificationCode);
    void resendVerificationMail(User user);
}

```

Then, code the method, in `UserServiceImpl`, as below:

```

@Override
public void resendVerificationMail(User user) {

    // The user must exist
    MyUtil.validate(user != null, "userNotFound");

    // must be unverified
    MyUtil.validate(user.getRoles().contains(Role.UNVERIFIED),
        "alreadyVerified");

    // send the verification mail
    sendVerificationMail(user);
}

```

Just a couple of validations, followed by sending the verification mail using the private method that we wrote earlier.

Add the `userNotFound` validation message to `messages.properties`:

```
userNotFound: User not found
```

Setting the permissions right

If you didn't observe, there is a security loophole in our code. Presently, any authenticated user can visit a resend-verification-mail link meant for another user. Let's restrict that just to the same user who has logged in, and to ADMINS.

So, add a validate call in our service method, as below:

```
@Override
public void resendVerificationMail(User user) {

    // The user must exist
    MyUtil.validate(user != null, "userNotFound");

    // Only self or ADMINS allowed
    MyUtil.validate(user.isAdminOrSelfLoggedIn(), "notPermitted");

    // must be unverified
    MyUtil.validate(user.getRoles().contains(Role.UNVERIFIED),
        "alreadyVerified");

    // send the verification mail
    sendVerificationMail(user);
}
```

Code the `isAdminOrSelfLoggedIn` method in `User` class, as below:

```
public boolean isAdminOrSelfLoggedIn() {

    User loggedIn = MyUtil.getUser();

    if (loggedIn == null) // nobody logged in
        return false;

    if (loggedIn.isAdmin()) // an Admin has logged in
        return true;

    if (loggedIn.getId() == id) // Same user has logged in
        return true;

    return false; // some other user has logged in
}

public boolean isAdmin() {
    return roles.contains(Role.ADMIN);
}
```

A cleaner approach to check such permissions will be to use `@PreAuthorize` with `hasPermission`, which is discussed in detail in our e-book [Spring Framework REST API Development – A Complete Blueprint](#).

Finally, add the `notPermitted` message in the `messages.properties`, as below:

```
notPermitted: Not permitted
```

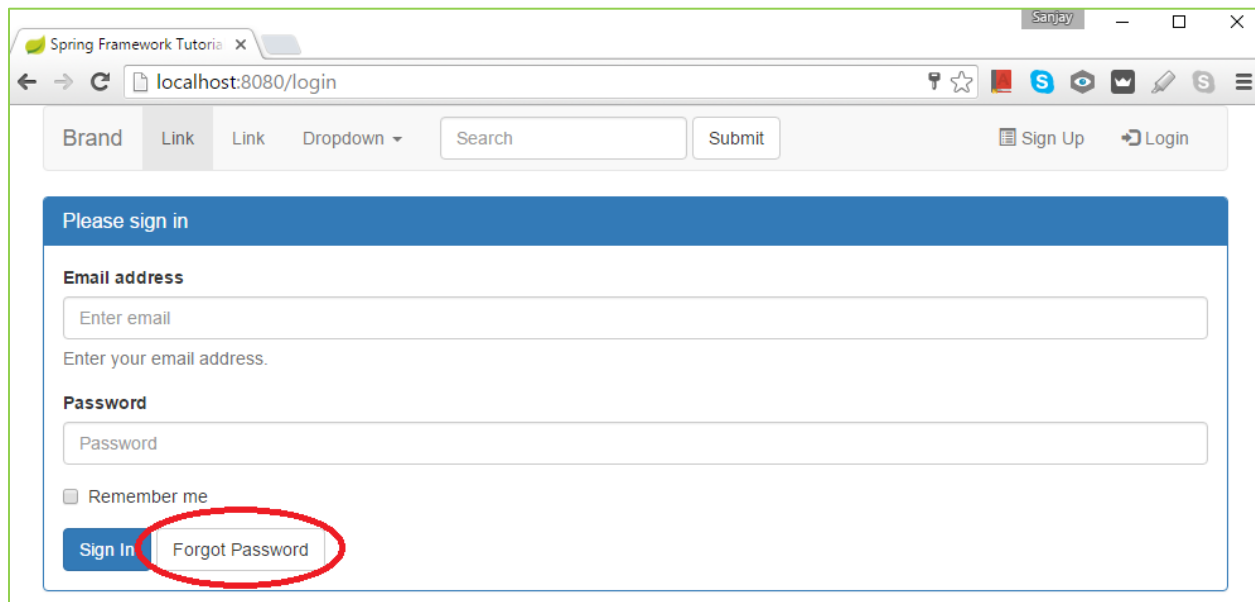
This completes our user verification functionality! Test it out.

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

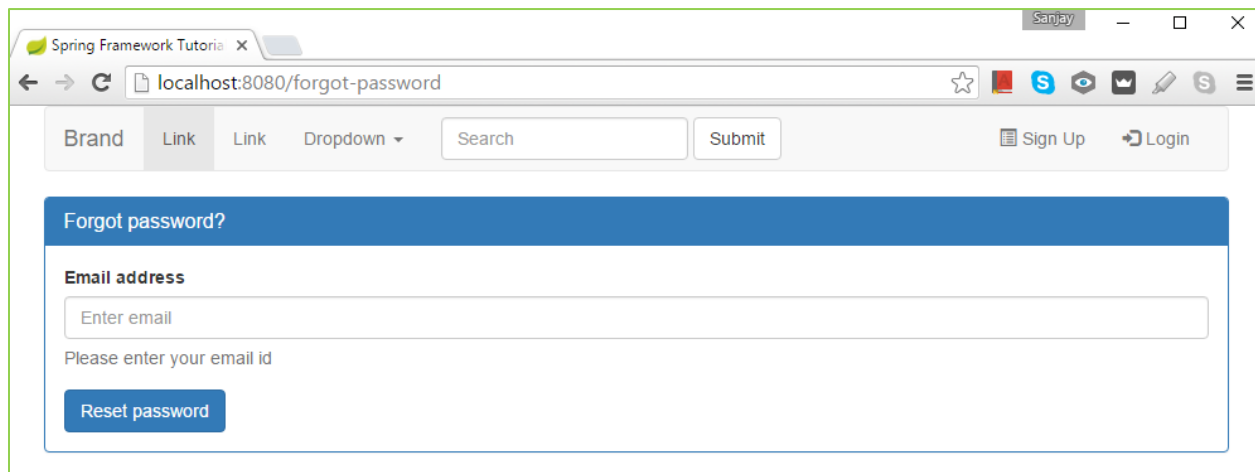
21 Forgot password

Users often forget their passwords. So, let's provide a *forgot password* button somewhere, say in the *Sign In* form, as below:



A screenshot of a web browser window showing a login form. The browser's address bar displays 'localhost:8080/login'. The page has a header with a navigation bar containing 'Brand', 'Link', 'Link', and a 'Dropdown' menu, along with a 'Search' input and a 'Submit' button. On the right side of the header are links for 'Sign Up' and 'Login'. The main content area is titled 'Please sign in' and contains two input fields: 'Email address' with the placeholder 'Enter email' and 'Password' with the placeholder 'Password'. Below these fields is a checkbox labeled 'Remember me'. At the bottom of the form are two buttons: 'Sign In' and 'Forgot Password'. The 'Forgot Password' button is circled in red.

When a user will click on the button, a *forgot password form* will be shown, as below:



A screenshot of a web browser window showing the 'Forgot password?' form. The browser's address bar displays 'localhost:8080/forgot-password'. The page layout is consistent with the previous screenshot, featuring the same header and navigation elements. The main content area is titled 'Forgot password?' and contains an 'Email address' input field with the placeholder 'Enter email'. Below the input field is the text 'Please enter your email id'. At the bottom of the form is a single button labeled 'Reset password'.

As you see, the form will just have an *email* field. On entering the registered email id, the user will get a mail containing a reset-password link. Clicking on the link will bring her to a *reset password form* on our site, as below:

The screenshot shows a web browser window with the title 'Spring Framework Tutorial'. The address bar displays 'localhost:8080/reset-password/ef6eabe7-ac4e-458e-b936-a33c03e21197'. The page features a navigation bar with 'Brand', 'Link', 'Link', and 'Dropdown' menus, a search bar, and a 'Submit' button. On the right, there are 'Sign Up' and 'Login' links. The main content area is titled 'Reset your password' and contains two input fields: 'Type new password' and 'Retype new password'. A blue 'Reset password' button is located at the bottom of the form.

She can set a new password there.

Forgot password button

So, let's begin the coding. For adding the *forgot password button*, add the following snippet to *login.jsp*:

```
...
<button type="submit" class="btn btn-primary">
    Sign In
</button>

<a class="btn btn-default" href="/forgot-password">
    Forgot Password
</a>

</form:form>
...
```

Note that it's actually a link to */forgot-password*, although it will look like a button because of the bootstrap styles.

Forgot password form

Next, let's code the forgot password form, which 'll ask for the email id of the user.

ForgotPasswordForm class

Let's first create a **ForgotPasswordForm** class that will be used for receiving the form data. Let's create a new package, say *com.naturalprogrammer.spring.tutorial.dto*, for such classes, and create *ForgotPasswordForm* in that, as below:

```

package com.naturalprogrammer.spring.tutorial.dto;

import com.naturalprogrammer.spring.tutorial.validation.ExistingEmail;

public class ForgotPasswordForm {

    @ExistingEmail
    private String email;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}

```

Looks simple, but what's the `@ExistingEmail` annotation? That's a custom constraint we are yet to code, similar to our `@UniqueEmail`, to check that such an email exists in our database.

@ExistingEmail constraint

Ready for an exercise? Code the `@ExistingEmail` constraint yourself.

Hint: You just need to copy `@UniqueEmail` and `@UniqueEmailValidator`, and do a few tiny changes. Refer to the finished source code of our [ExistingEmail](#) and [ExistingEmailValidator](#) if you find it difficult. Note that you will also need to add the error message, e.g. `emailNotFound: Email not found`, in [messages.properties](#).

Forgot password GET handler

Next, let's code the GET handler for `/forgot-password`, which will display the form. So, create a new `ForgotPasswordController` class with a handler method, looking as below:

```

package com.naturalprogrammer.spring.tutorial.controllers;

...

@Controller
@RequestMapping("/forgot-password")
public class ForgotPasswordController {

    @GetMapping
    public String forgotPassword(Model model) {

        model.addAttribute(new ForgotPasswordForm());
        return "forgot-password";
    }
}

```


The handler just adds a new *ForgotPasswordForm* to the model, and then forwards to `forgot-password.jsp`, which you can code as below:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@ include file="includes/header.jsp" %>

<div class="panel panel-primary">

    <div class="panel-heading">
        <h3 class="panel-title">Forgot password?</h3>
    </div>

    <div class="panel-body">

        <form:form modelAttribute="forgotPasswordForm" role="form">

            <form:errors />

            <div class="form-group">
                <form:label path="email">Email address</form:label>
                <form:input path="email" type="email"
                    class="form-control" placeholder="Enter email" />
                <form:errors cssClass="error" path="email" />
                <p class="help-block">Please enter your email id</p>
            </div>

            <button type="submit" class="btn btn-primary">Reset password</button>

        </form:form>
    </div>
</div>

<%@ include file="includes/footer.jsp" %>
```

As you see, it's just a form with an email field and a submit button. It should be obvious to you, as we have already covered displaying and submitting forms in details in earlier chapters.

Next, let's code the POST handler, which will be called when the user will submit the form.

Forgot password POST handler

The POST handler can be coded in *ForgotPasswordController*, as below:

```
@Controller
@RequestMapping("/forgot-password")
public class ForgotPasswordController {

    @Autowired
    private UserService userService;

    ...

    @PostMapping
```

```

public String forgotPassword(
    @Validated ForgotPasswordForm forgotPasswordForm,
    BindingResult result,
    RedirectAttributes redirectAttributes) {

    if (result.hasErrors())
        return "forgot-password";

    userService.forgotPassword(forgotPasswordForm);
    MyUtil.flash(redirectAttributes, "info",
        "checkMailResetPassword");

    return "redirect:/";
}

```

Much of the above code needs no explanation – it's quite similar to the [signup POST handler](#). We call the service layer to do the job, and then flash a message. Add the following message line to *messages.properties*:

```
checkMailResetPassword: Please check your mail to reset your password.
```

Next, we'll code the service layer.

The service method

First, add a *forgotPassword* method to the *UserService* interface:

```

public interface UserService {

    ...

    void forgotPassword(ForgotPasswordForm forgotPasswordForm);

}

```

Then, code the method in *UserServiceImpl*, as below:

```

@Override
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void forgotPassword(ForgotPasswordForm forgotPasswordForm) {

    // fetch the user record from database
    User user = userRepository.findByEmail(forgotPasswordForm.getEmail()).get();

    // set a reset password code
    user.setResetPasswordCode(UUID.randomUUID().toString());
    userRepository.save(user);

    // after successful commit, mail him a link to reset his password
    MyUtil.afterCommit(() -> mailResetPasswordLink(user));

}

```

Here we are fetching the user from the database, and then storing a unique UUID code in it, in a new `resetPasswordCode` field that we are yet to code. That code will be mailed to the user, and then matched against her record later, as we'll see.

We are then saving the user, and on successful commit, sending her a mail having a link with the `resetPasswordCode`. We're using the `mailResetPasswordLink` private method for sending the mail, which we are yet to code.

So, let's first add the `resetPasswordCode` field to `User` class:

```
public class User implements UserDetails {  
  
    ...  
  
    @Column(length = 36, unique=true)  
    private String resetPasswordCode;  
  
    public String getResetPasswordCode() {  
        return resetPasswordCode;  
    }  
    public void setResetPasswordCode(String resetPasswordCode) {  
        this.resetPasswordCode = resetPasswordCode;  
    }  
  
    ...  
}
```

Then, let's code the `mailResetPasswordLink` private method, in `UserServiceImpl`, as below:

```
private void mailResetPasswordLink(User user) {  
  
    try {  
  
        // make the link  
        String resetPasswordLink = applicationUrl  
            + "/reset-password/" + user.getResetPasswordCode();  
  
        // send the mail  
        mailSender.send(user.getEmail(),  
            MyUtil.getMessage("resetPasswordSubject"),  
            MyUtil.getMessage("resetPasswordEmail",  
                resetPasswordLink));  
  
    } catch (MessagingException e) {  
        // In case of exception, just log the error and keep silent  
  
        log.error(ExceptionUtils.getStackTrace(e));  
    }  
}
```

Needs no explanation - quite similar to [sendVerificationMail](#) that we have coded earlier. Put the used messages in `messages.properties`, as below:

```
resetPasswordSubject: Forgot password?
```

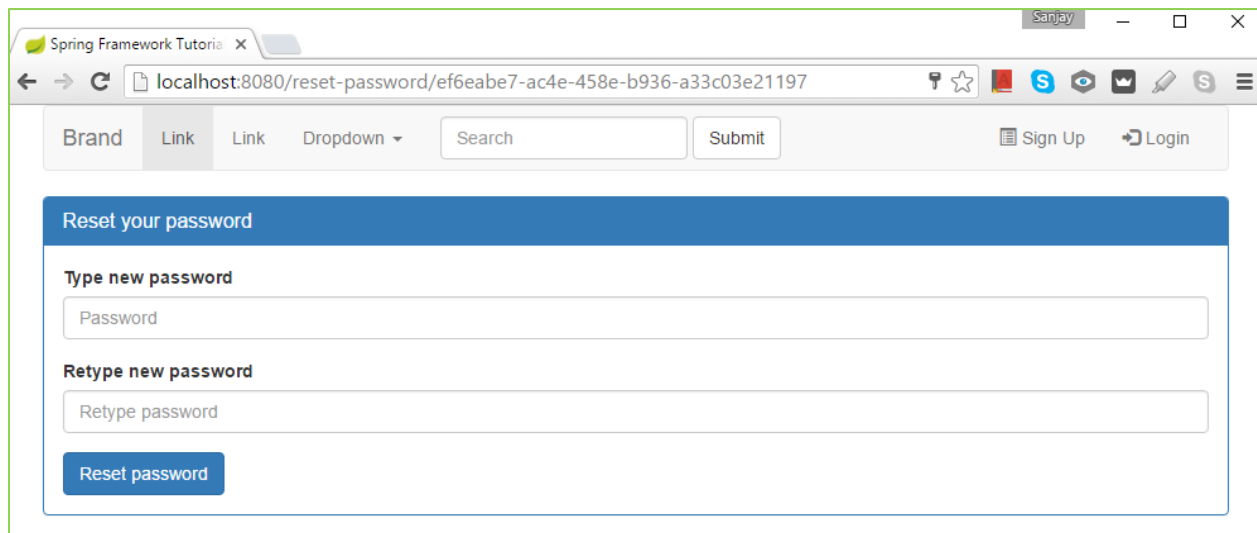
```
resetPasswordEmail: Hi,<br/><br/>\nPlease click the link below to reset your password:\n<br/><br/>{0}<br/><br/>
```

Resetting the password

Notice that the reset-password link that we are sending to the user is of the format

```
/reset-password/unique-reset-password-code
```

Clicking on it, the user should see a form as below:

A screenshot of a web browser window. The address bar shows 'localhost:8080/reset-password/ef6eabe7-ac4e-458e-b936-a33c03e21197'. The page has a header with 'Brand', 'Link', 'Link', 'Dropdown', a search bar, and a 'Submit' button. There are also 'Sign Up' and 'Login' links. The main content area is titled 'Reset your password' and contains two input fields: 'Type new password' and 'Retype new password'. Below these fields is a 'Reset password' button.

The form will ask for a new password, as you see above. So, let's code the form.

ResetPasswordForm class

Let's first create a `ResetPasswordForm` class that will be used for receiving the form data, i.e. `password` and `retypePassword`. So, create a `ResetPasswordForm` in the `.dto` package, as below:

```
ResetPasswordForm
public class ResetPasswordForm {

    @Password
    private String password;

    @Password
    private String retypePassword;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

```

    }

    public String getRetypePassword() {
        return retypePassword;
    }

    public void setRetypePassword(String retypePassword) {
        this.retypePassword = retypePassword;
    }
}

```

Looks simple, but notice the `@RetypePassword` class level annotation. It's a custom constraint that we are yet to code, to ensure that both the entered passwords are same. Unlike `@UniqueEmail`, it had to be a class level constraint, because it needs access to both the fields.

Let's see how to code it.

Coding @RetypePassword annotation

Code the annotation, say in the `.validation` package, as below:

```

package com.naturalprogrammer.spring.tutorial.validation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Constraint(validatedBy=RetypePasswordValidator.class)
public @interface RetypePassword {

    String message() default "{passwordsDoNotMatch}";

    Class[] groups() default {};

    Class[] payload() default {};
}

```

Looks quite similar to the custom annotations we have coded earlier, except that its `@Target` is limited just to `TYPE` and `ANNOTATION_TYPE`, because it's a class level annotation.

Add the `passwordsDoNotMatch` message to `messages.properties`:

```
passwordsDoNotMatch: Passwords do not match
```

RetypePasswordValidator

Next, let's code the validator class, i.e. `RetypePasswordValidator`, in the `.validation` package, as below:

```
package com.naturalprogrammer.spring.tutorial.validation;

import java.util.Objects;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

import org.springframework.stereotype.Component;

import com.naturalprogrammer.spring.tutorial.dto.ResetPasswordForm;

@Component
public class RetypePasswordValidator
    implements ConstraintValidator<RetypePassword, ResetPasswordForm> {

    @Override
    public boolean isValid(ResetPasswordForm retypePasswordForm,
        ConstraintValidatorContext context) {

        if (!Objects.equals(retypePasswordForm.getPassword(),
            retypePasswordForm.getRetypePassword())) {

            // Moving the error from form-level to
            // field-level property: retypePassword
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate(
                "{passwordsDoNotMatch}")
                .addPropertyNode("retypePassword").addConstraintViolation();

            return false;
        }

        return true;
    }

    @Override
    public void initialize(RetypePassword constraintAnnotation) {
        // initialization code
    }
}
```

The code looks familiar, but notice how we have used the `context`, for associating the error to the `retypePassword` property.

This will ensure that the error message will be displayed beside the `retypePassword` field in our form. Refer to the [documentation](#) of Hibernate Validator for more details.

Next, let's code the JSP for the form.

Coding the JSP

The JSP can be coded just as below. Name it as `reset-password.jsp`, and place it at `src/main/webapp/WEB-INF/jsp`.

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@ include file="includes/header.jsp" %>

<div class="panel panel-primary">

    <div class="panel-heading">
        <h3 class="panel-title">Reset your password</h3>
    </div>

    <div class="panel-body">

        <form:form modelAttribute="resetPasswordForm" role="form">

            <form:errors cssClass="error" />

            <div class="form-group">
                <form:label path="password">Type new password</form:label>
                <form:password path="password" class="form-control"
                    placeholder="Password" />
                <form:errors cssClass="error" path="password" />
            </div>

            <div class="form-group">
                <form:label path="retypePassword">Retype new password</form:label>
                <form:password path="retypePassword" class="form-control"
                    placeholder="Retype password" />
                <form:errors cssClass="error" path="retypePassword" />
            </div>

            <button type="submit" class="btn btn-primary">Reset password</button>

        </form:form>
    </div>
</div>

<%@include file="includes/footer.jsp"%>
```

Quite self-explanatory; nothing new to talk about.

Coding the GET handler

Now we can code the controller and service. Code a new `ResetPasswordController` in the `.controllers` package, and put the GET handler in that, as below:

```
@Controller
@RequestMapping("/reset-password/{resetPasswordCode}")
public class ResetPasswordController {

    @GetMapping
    public String forgotPassword(Model model) {

        model.addAttribute(new ResetPasswordForm());
    }
}
```

```

        return "reset-password";
    }
}

```

It will render the form by using the JSP that we last coded.

If you feel that we are coding too many controller classes, you can very well put the handlers in any existing controller – up to you.

Coding the POST handler

Next, let's code the form submission handler, which 'll be called when the user presses the submit button. Code it in the *ResetPasswordController*, as below:

```

@Controller
@RequestMapping("/reset-password/{resetPasswordCode}")
public class ResetPasswordController {

    @Autowired
    private UserService userService;

    ...

    @PostMapping
    public String resetPassword(
        @PathVariable String resetPasswordCode,
        @Validated ResetPasswordForm resetPasswordForm,
        BindingResult result,
        RedirectAttributes redirectAttributes) {

        if (result.hasErrors())
            return "reset-password";

        try {
            userService.resetPassword(resetPasswordCode,
                                     resetPasswordForm.getPassword());
            MyUtil.flash(redirectAttributes,
                        "success", "passwordChanged");
            return "redirect:/login";
        } catch (NoSuchElementException e) {
            result.reject("invalidUrl");
            return "reset-password";
        }
    }
}

```

The code looks somewhat new, and in fact, exhibits an important validation pattern. Notice that we are calling the `resetPassword` method of *UserService*, which we are yet to code. That method is going to throw a *NoSuchElementException* in case a user with the given *resetPasswordCode* isn't found in the database. We are catching that exception, and adding a validation error in such case, by calling `result.reject("invalidUrl")`.

The `reject` method adds a global, form level error. For adding a field level error, `rejectValue` can be used. For more details, see the [documentation](#) of *Errors*, which is a super interface of *BindingResult*.

So, we just saw a convenient way to add validation errors – manually. In this particularly case, we actually didn't need this pattern. Just throwing an exception, which would have resulted in showing our error page, was better. But, the objective here was just to show you how you can add validation errors manually.

The argument `"invalidUrl"` above is a message key. So add a line to *messages.properties* as below:

```
invalidUrl: The URL is no more valid. Probably it's already used.
```

Also, for the flash message, add another line:

```
passwordChanged: Password Changed.
```

Coding the service

For coding the service method, first add a *resetPassword* method to the *UserService* interface:

```
public interface UserService {  
  
    ...  
  
    void resetPassword(String resetPasswordCode, String password);  
}
```

Then, code the method in *UserServiceImpl*, as below:

```
@Override  
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)  
public void resetPassword(String resetPasswordCode,  
    String password) {  
  
    User user = userRepository.findByResetPasswordCode(resetPasswordCode).get();  
    user.setPassword(passwordEncoder.encode(password));  
    user.setResetPasswordCode(null);  
    userRepository.save(user);  
}
```

The highlighted line will return an optional user with the given *resetPasswordCode*. If no such user is found, *NoSuchElementException* will be thrown, which will be caught in the controller method, as we already have discussed.

Rest all is self-explanatory.

We are yet to code *findByResetPasswordCode* in *UserRepository*. Do that now, as below:

```
public interface UserRepository extends JpaRepository<User, Long> {
```

```
Optional<User> findByEmail(String email);  
Optional<User> findByResetPasswordCode(String resetPasswordCode);  
}
```

This completes the forgot password functionality! Test it out.

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

22 Showing User Profile

In this chapter, we're going to code a user profile page, to be available at `/users/{userId}`.

Coding the handler

Let's begin by coding the handler, in the *UserController*, looking as below:

```
@Controller
@RequestMapping("/users")
public class UserController {

    ...

    @GetMapping("/{userId}")
    public String getById(@PathVariable long userId, Model model) {

        model.addAttribute(userService.findById(userId));
        return "user";
    }
}
```

As you see, to fetch the user with the given id, we are using `findById` of *UserService*, which we are yet to code. After getting the user, we are adding it to the model and then forwarding to `user.jsp`, which we'll code next.

Coding the view

Code `user.jsp`, in `src/main/webapp/WEB-INF/jsp`, as below:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@include file="includes/header.jsp"%>

<div class="panel panel-primary">

    <div class="panel-heading">
        <h3 class="panel-title">Profile</h3>
    </div>

    <div class="panel-body">
        <dl class="dl-horizontal">
            <dt>Name</dt>
            <dd><c:out value="${user.name}" /></dd>
            <dt>Email</dt>
            <dd><c:out value="${user.email}" /></dd>
            <dt>Roles</dt>
        </dl>
    </div>
</div>
```

```

        <dd><c:out value="\${user.roles}" /></dd>
    </dl>
</div>

</div>

<%@include file="includes/footer.jsp"%>

```

The code is pretty simple – it just shows the *name*, *email* and *roles* of the *user* in some bootstrap widgets.

Coding the service method

Next, let's code the `findById` method. Add it to the `UserService` interface, as below:

```

public interface UserService {

    ...

    User findById(long userId);

}

```

Then, add it to `UserServiceImpl`, as below:

```

@Override
public User findById(long userId) {

    User user = userRepository.findOne(userId);
    MyUtil.validate(user != null, "userNotFound");

    if (!user.isAdminOrSelfLoggedIn())
        user.setEmail("Confidential");

    return user;

}

```

As you see, the first couple of lines fetch the user from the database and validate that such a user indeed exists. Following that, we hide the email field if the logged in user does not have right to view it. For checking the right, we use the `user.isAdminOrSelfLoggedIn()` method, which returns true only if the same user or an admin will have logged in.

Finally, we will want to allow even anonymous users to visit the profile of others. So, we need to tell Spring Security to permit access to URLs of the format `/users/*` to all users. To do so, add a line to `SecurityConfig`, as below:

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    ...

    @Override

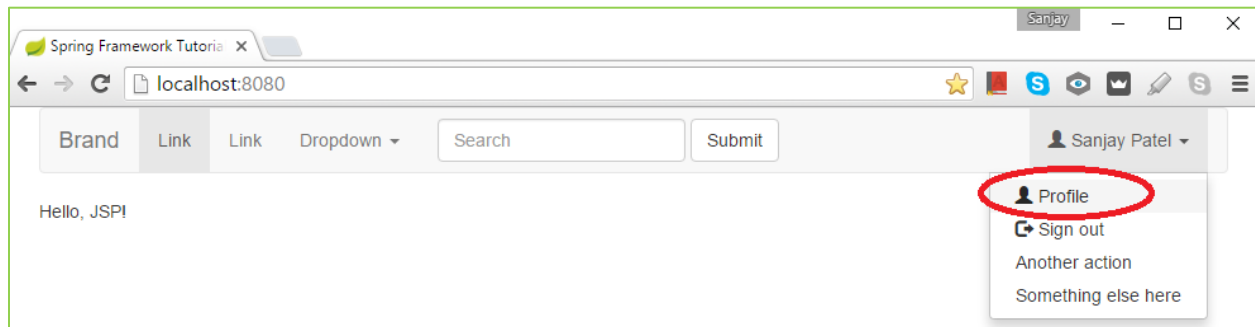
```

```

protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .mvcMatchers(HttpMethod.GET,
                "/users/*",
                "/").permitAll()
            ...
}

```

This completes our profile feature. Before we test it out, let's add a link to the profile page of the logged in user in our navigation bar, as below:



For that, add the following snippet to *header.jsp*:

```

...

<sec:authorize access="isAuthenticated()">
    <li class="dropdown">
        <a href="#" class="dropdown-toggle" data-toggle="dropdown"
            role="button" aria-haspopup="true" aria-expanded="false">
            <span class="glyphicon glyphicon-user"></span>
            <sec:authentication property="principal.name" />
            <span class="caret"></span>
        </a>
        <ul class="dropdown-menu">
            <li><a href="/users/<sec:authentication property='principal.id' />">
                <span class="glyphicon glyphicon-user"></span> Profile</a>
            </li>
            <li>
                <c:url var="logoutUrl" value="/logout" />
                <form:form id="logoutForm" action="${logoutUrl}">
                </form:form>
            </li>
        </ul>
    </li>
</sec:authorize>
...

```

Notice how we have used the `<sec:authentication>` tag for getting the id of the logged in user. We had already discussed about this tag [earlier](#), if you remember.

So, test it out!

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

23 Edit profile

In this chapter, we'll code editing the profile. Precisely, we'll provide a form at `/users/n/edit`, where the data of user with id *n* can be edited.

Edit Profile link

The profile page seems to be the best place to provide an *edit profile* link. We can actually place a number links there, e.g. *edit profile*, *change password* and *change email*. So, add the following snippet to *user.jsp*:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@include file="includes/header.jsp"%>

<div class="panel panel-primary">

    <div class="panel-heading">
        <h3 class="panel-title">Profile</h3>
    </div>

    <div class="panel-body">
        <dl class="dl-horizontal">
            <dt>Name</dt>
            <dd><c:out value="${user.name}" /></dd>
            <dt>Email</dt>
            <dd><c:out value="${user.email}" /></dd>
            <dt>Roles</dt>
            <dd><c:out value="${user.roles}" /></dd>
        </dl>
    </div>

    <c:if test="${user.adminOrSelfLoggedIn}" >
        <div class="panel-footer">

            <a class="btn btn-link" href="/users/${user.id}/edit">Edit</a>
            <a class="btn btn-link" href="/users/${user.id}/change-password">
                Change password</a>
            <a class="btn btn-link" href="/users/${user.id}/change-email">
                Change email id</a>
        </div>
    </c:if>

</div>

<%@include file="includes/footer.jsp"%>
```

Notice that we have enclosed the links in a `<c:if>` tag, so that they will be rendered only if an admin or the user herself has logged in.

Displaying the form

Next, let's code displaying the form.

Coding the handler

Code the handler in *UserController*, as below:

```
@Controller
@RequestMapping("/users")
public class UserController {

    ...

    @GetMapping("/{id}/edit")
    public String edit(@PathVariable("id") User user, Model model) {

        model.addAttribute(user);
        return "user-edit";
    }
}
```

As you see, the user entity is automatically fetched from database – a Spring Data feature that we have already seen [earlier](#). We are then adding the user object to the model, and forwarding to the view – *user-edit.jsp*.

Coding the view

Code *user-edit.jsp*, in *src/main/webapp/WEB-INF/jsp*, as below:

```
<%@ page language="java" contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<%@include file="includes/header.jsp"%>

<div class="panel panel-primary">

    <div class="panel-heading">
        <h3 class="panel-title">Profile</h3>
    </div>

    <div class="panel-body">

        <form:form modelAttribute="user" class="form-horizontal" role="form">

            <div class="form-group">
                <form:label path="name" class="col-lg-2 control-label">Name</form:label>
                <div class="col-lg-10">
                    <form:input path="name" class="form-control" placeholder="Name" />
                    <form:errors cssClass="error" path="name" />
                    <p class="help-block">Enter your display name.</p>
                </div>
            </div>

            <sec:authorize access="hasRole('ROLE_ADMIN')">
                <div class="form-group">
                    <form:label path="roles" class="col-lg-2 control-label">
```



```

        Roles
    </form:label>
    <div class="col-lg-10">
        <form:input path="roles" class="form-control" placeholder="Roles" />
        <form:errors cssClass="error" path="roles" />
        <p class="help-block">
            Enter the roles of the user, separated by commas
        </p>
    </div>
</div>
</sec:authorize>

<div class="form-group">
    <div class="col-lg-offset-2 col-lg-10">
        <button type="submit" class="btn btn-primary">Update</button>
    </div>
</div>

</form:form>

</div>
</div>
<%@include file="includes/footer.jsp"%>

```

It's basically a form with two fields, *name* and *roles*. *roles* is visible only when an admin has logged in.

An interesting thing to note here is that *roles* collection is mapped to a text input. Consequently, Spring will automatically convert the collection to a comma separated string, to and fro. This conversion is taken care by Spring through a couple of ways – [property editors](#) and [conversion services](#).

Spring provides many built-in property editors and conversion services. That's why we did not have to write one for converting the roles collection to/from string. Of course you can write your own conversion services and property editors, and register them with Spring. But, you will not need that most of the times, because Spring already provides a rich set of these.

Submitting the form

Let's now code the form submission.

Coding the handler

Code the handler, in the *UserController*, as below:

```

@Controller
@RequestMapping("/users")
public class UserController {

    ...

    @PostMapping("/{id}/edit")

```

```

    public String edit(@PathVariable("id") User user,
                      @Validated User updatedData,
                      BindingResult result,
                      RedirectAttributes redirectAttributes) {

        if (result.hasErrors())
            return "user-edit";

        userService.update(user, updatedData);
        MyUtil.flash(redirectAttributes, "success", "editSuccess");

        return "redirect:/";
    }
}

```

Quite self-explanatory. We are calling `userService.update` for the doing the actual update, which we'll code next. But before that, add the `editSuccess` message in `messages.properties`:

```
editSuccess: Edit succeeded!
```

Coding the service

Next, add the update method the `UserService` interface, as below:

```

public interface UserService {

    ...

    void update(User user, User updatedData);
}

```

Then, add it to `UserServiceImpl`, as below:

```

@Override
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void update(User user, User updatedData) {

    // Ensure that a user with the given id exists
    MyUtil.validate(user != null, "userNotFound");

    // Only self or an admin can edit the profile data
    MyUtil.validate(user.isAdminOrSelfLoggedIn(), "notPermitted");

    // Update the name
    user.setName(updatedData.getName());

    User loggedIn = MyUtil.getUser();

    // Only an admin can edit roles
    if (loggedIn.isAdmin())
        user.setRoles(updatedData.getRoles());

    // save the updates
    userRepository.save(user);
}

```

```

MyUtil.afterCommit(() -> {

    // If the logged in user is editing his own profile,
    // log her in again, so that Spring Security principal
    // gets updated
    if (loggedIn.getId() == user.getId())
        MyUtil.login(user);
});
}

```

Looking at the inline comments will reveal everything about the code. In summary, the first couple of lines ensures that a user entity with the given id exists, and the logged in user has permissions to edit it. Then, we update the name of the user. If the logged in user is an admin, we update the roles as well.

We then save the updated user, and upon successful commit, if the user that was updated was actually the logged-in user, we re-login her. This is to update the principal object of Spring Security.

A subtle validation problem

Time for testing. Try logging in and edit your name, and you'll see that nothing seems to happen when you press the "Update" button.

So, what's the problem?

To receive the form data, we are using the *User* class, whose *email* and *password* fields are annotated with the *@UniqueEmail* and *@Password* validation constraints. Because the form data does not contain such fields, these constraints are not letting the validation pass.

What's the solution?

One approach will be not to use the *User* class, and code a new class, quite similar to our *ForgotPasswordForm* and *ResetPasswordForm*. You already know how to do it.

A second approach will be using [validation groups](#). Let's try that out.

Validation Groups

Our *User* class is being used for receiving input data at two places – when signing up and when updating profile. Both have different validation requirements. When signing up, we want to validate all the three fields, viz. *email*, *name* and *password*. Whereas, when updating the profile, we want just to validate the *name*, and not the *email* or *password*.

In such case, we can use validation groups. Going ahead, first create a couple of marker interfaces, say in the *User* class:

```

public class User implements UserDetails {

```

```

    private static final long serialVersionUID = 3361118059928147222L;

    // Validation groups
    public static interface SignUpValidation {}
    public static interface UpdateValidation {}

    ...
}

```

Next, in the validation annotations of the fields, specify validation groups as below:

```

public class User implements UserDetails {

    ...

    @UniqueEmail(groups = SignUpValidation.class)
    @Column(nullable = false, length = 250)
    private String email;

    @NotBlank(groups = {SignUpValidation.class, UpdateValidation.class})
    @Size(max=100, groups = {SignUpValidation.class, UpdateValidation.class})
    @Column(nullable = false, length = 100)
    private String name;

    @Password(groups = SignUpValidation.class)
    @Column(nullable = false) // no length because it will be encrypted
    private String password;

    ...
}

```

Finally, pass the interfaces as parameters to the @Validated annotation in the request handlers. Specifically update *SignupController* as below:

```

@Controller
@RequestMapping("/signup")
public class SignupController {

    ...

    @PostMapping
    public String doSignup(
        @Validated(User.SignUpValidation.class) User user,
        BindingResult result,
        RedirectAttributes redirectAttributes) {

        ...
    }
}

```

And, the *UserController* as below:

```

@Controller
@RequestMapping("/users")
public class UserController {

```

```
...

@PostMapping("/{id}/edit")
public String edit(@PathVariable("id") User user,
                  @Validated(User.UpdateValidation.class) User updatedData,
                  BindingResult result,
                  RedirectAttributes redirectAttributes) {
    ...
}
```

Test it now, and things should work well.

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

24 AOP

In this chapter, we're going to learn how to do aspect oriented programming in Spring.

What is AOP

When we develop an application, we write code not only for the business functionality, but also for many other operational features, e.g. logging and transaction management. These features, which are not limited to one particular functionality, but are applicable across the entire application, are called *aspects* or *crosscutting concerns*.

Speaking more technically, when we code a method, apart from coding the business logic, we also write operational code. For example, a method could need a *begin transaction* statement at the beginning, and an *end transaction* statement at the end. Or, you could end a method by writing a log line, indicating that the function was completed.

These operational aspects can be useful not only for one particular method, but for multiple methods. We have already seen how the *@Transactional* annotation encloses a method inside a transaction. That's a classic example of aspect oriented programming in Spring.

Spring's [documentation](#) covers the AOP concepts and its terminology very well. So, without repeating that here, let's jump to some hands-on.

Coding an Aspect

Let's code an aspect that'll write a couple of log lines, one before and one after a controller method is executed.

So, create a new class, say *RequestMonitor*, which will be the aspect. Put it somewhere, say in a new *com.naturalprogrammer.spring.tutorial.aop* package, and code it as below:

```
package com.naturalprogrammer.spring.tutorial.aop;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class RequestMonitor {
```

```

private static final Log log = LoggerFactory.getLog(RequestMonitor.class);

@Around("within(com.naturalprogrammer.spring.tutorial.controllers.*)")
public Object wrap(ProceedingJoinPoint pjp) throws Throwable {

    log.info("Before handler '"
        + pjp.getSignature().getName() + "'. Thread "
        + Thread.currentThread().getName()); // toString gives more info

    Object retVal = pjp.proceed();

    log.info("Handler '"
        + pjp.getSignature().getName()
        + "' execution successful");

    return retVal;
}
}

```

Looks like a tiny class, but it has a lot to understand. Specifically, notice the following

1. The class is annotated with `@Component`, so that an instance of it will go to the application context.
2. Components in the application context that are annotated with `@Aspect` will be recognized by Spring as aspects. Hence, we needed to annotate the class with `@Aspect`.
3. There is a method in the class, named as `wrap`, which is annotated with `@Around`. Consequently, it becomes an *around advice*. An advice is associated with a [pointcut](#) expression, and runs before, after, or around method executions matched by the pointcut.
4. In our case, the pointcut expression is `within(com.naturalprogrammer.spring.tutorial.controllers.*)`, i.e. the argument of the `@Around` annotation. It tells Spring AOP to apply our wrap advice to all the methods of all the classes in the `com.naturalprogrammer.spring.tutorial.controllers` package. Refer Spring's [documentation](#) for details on how to write point-cuts.
5. Methods around which advices run are called *join points*. In our case, all the methods inside our controllers will be join points.
6. The argument passed to the wrap advice, named as `pjp` in our case, represents the join point, i.e. the controller method around which the advice will run.
7. The first log line executes before the controller method is executed.
8. `pjp.proceed()` inside the wrap method executes the actual controller method.
9. The second log line executes after the controller method is executed.
10. The return value of `pjp.proceed()` should be returned from around advices, as we have returned.

Try running our application now, and visit some URL, like `/signup`. In the log, you'll see the log lines appear, as below:

```

al.aop.RequestMonitor : Before handler 'signup'. Thread http-nio-8080-exec-7
al.aop.RequestMonitor : Handler 'signup' execution successful

```

So, this was a core introduction to Spring AOP. It's quite powerful - Read its [documentation](#) for more details.

Everything we've covered in this chapter is pure Spring AOP. But, you can also use the AspectJ compiler/weaver instead of, or in addition to, Spring AOP if your needs go beyond the facilities offered by Spring AOP alone. For more details, refer [here](#).

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

25 Asynchronous Processing

Asynchronous processing lets you run some logic in a different thread, in parallel, so that your current thread isn't blocked.

To illustrate, let's take our sign up process. When a user fills our signup form and presses submit, in our server side, we save the data to the database, and then send a verification mail. Sending the mail involves an external mail server, and so will often take some time. The browser of the user will remain stalled for all that period.

To avoid the stalling, the mail can be sent *asynchronously*. This way, the response will not wait for the mail to be sent.

Sending Mails Asynchronously

So, let's enhance our application to send all mails asynchronously. It's actually very simple in Spring, as you'll see.

To enable asynchronous processing in a Spring application, we first need to annotate one of its configuration classes with `@EnableAsync`. So, annotate `NpSpringTutorialApplication` with `@EnableAsync`:

```
@SpringBootApplication
@EnableSpringDataWebSupport
@EnableAsync
public class NpSpringTutorialApplication {

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

Now, just annotating any of your component methods with `@Async` will have it executed asynchronously. So, annotate the `send` method of `SmtplibMailSender` with `@Async`:

```
public class SmtplibMailSender implements MailSender {

    ...

    @Override
    @Async
    public void send(String to, String subject, String body)
```

```

        throws MessagingException {

        ...

    }

```

That's all. You can now test it out by configuring the *SmtpMailSender* as our mail sender. Refer our e-book [Spring Framework And Dependency Injection For Beginners](#) if you have forgotten how to do so.

To know whether the method indeed runs in a different thread, you can add a log line to it, as below:

```

@Override
@Async
public void send(String to, String subject, String body) throws
MessagingException {

    log.info("Sending SMTP mail from thread " +
        Thread.currentThread().getName()); // toString gives more info

    MimeMessage message = javaMailSender.createMimeMessage();
    MimeMessageHelper helper;

    helper = new MimeMessageHelper(message, true); // true indicates
                                                    // multipart message
    helper.setSubject(subject);
    helper.setTo(to);
    helper.setText(body, true); // true indicates html

    // continue using helper for more functionalities
    // like adding attachments, etc.

    javaMailSender.send(message);
}

```

Now, test it out, and you'll see in the log that the name of the thread will be different from the name of the thread of the request. Specifically, you'll see log lines as below:

```

Before handler 'doSignup'. Thread http-nio-8080-exec-1
Handler 'doSignup' execution successful
Sending SMTP mail from thread SimpleAsyncTaskExecutor-1

```

different

Source code so far

- [Browse online](#)
- [Download zip](#)
- [See changes](#)

26Scheduling

Sometimes you may need to execute some tasks in schedules, or periodically. For example, you may like send some usage reports all your users every week. Or, a business application may like to calculate incentives for its employees at the end of every month.

Running scheduled jobs in Spring applications is quite easy. You just have to annotate a configuration class with [@EnableScheduling](#), and then use the [@Scheduled](#) annotation on the method that you want to run in schedules.

The `@Scheduled` annotation takes some trigger metadata, which defines when and how the method will run. For example, the following method will execute every five seconds, measured between the successive start times of each invocation:

```
@Scheduled(fixedRate=5000)
public void doSomething() {
    // something that should execute periodically
}
```

Refer [here](#) for all the trigger options.

Spring has an excellent [getting started](#) guide on scheduling, so we thought not to reinvent the wheel.

Further Reading

Scheduling and asynchronous processing with Spring is a lot more configurable and powerful than we just discussed. For more details, refer to the [Task Execution and Scheduling](#) documentation of Spring.

27 Deployment

In this chapter, we'll touch upon various deployment options, and see how to deploy our application to Pivotal Cloud Foundry.

Deployment Options

When deploying an application, you will have multiple choices:

1. *Hiring physical or virtual servers at some data center, like SoftLayer.* You'll need to install the OS and other necessary software to run your application, using remote access. You may use multiple servers and other operational stuff like load balancer for scaling up and 24x7 availability.
2. *Hosting on some cloud, like AWS (Amazon Web Service).* Similar as above, except that, the time required to obtain and boot new server instances gets reduced to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change.
3. *Hosting on some PaaS (Platform-as-a-Service), like Pivotal Cloud Foundry or Heroku.* When using a PaaS, you hire the entire platform, instead of servers. It relieves you of installing the OS and other software, and you can straightly deploy your application to the platform. In fact, it's getting rapidly popular, not only because it is the most hassle free solution, but also the TCO (Total Cost of Ownership) would be the least most of the times.

Deploying to Pivotal Cloud Foundry

Among the PaaS solutions, *Pivotal Web Services* from *Pivotal Cloud Foundry* seems like a default choice for deploying Spring applications. Both Spring Framework and Pivotal Cloud Foundry are from the same company, and they are working hard to make the deployment procedure as easy as possible.

So, let's give it a try!

Signing Up

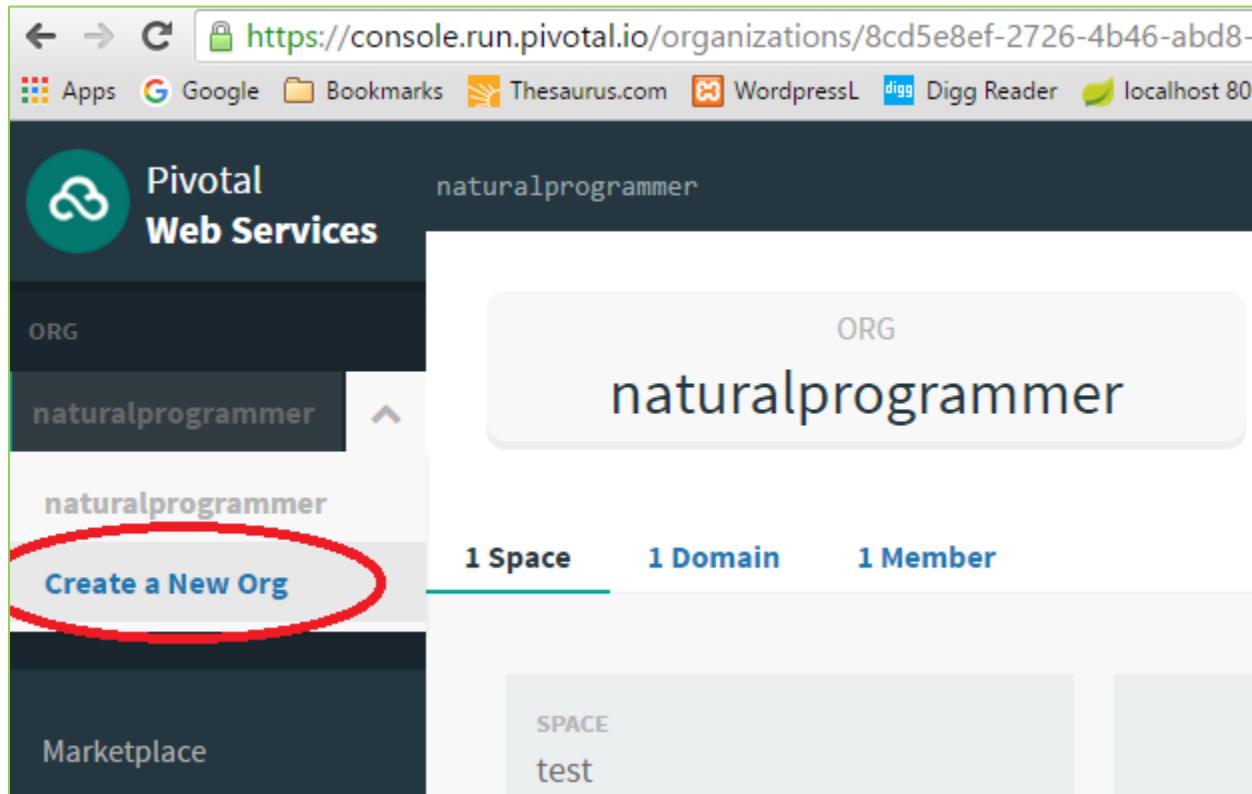
To use Pivotal Web Services, you will first need to sign up at <https://run.pivotal.io>.

When signing up, you will need to provide a mobile number where you will receive a verification code. Some mobile numbers in some countries like India fail to receive messages from Pivotal. So, use your best mobile number, and then if you do not receive the code, just raise a support request. We had the same issue, and they were very prompt in helping us out.

Org, Spaces and Apps

At Cloud Foundry, **apps** are hosted in **spaces**, and a *space* belongs to an **org**.

So, after signing up, the first thing would be to create a new *org*. Do that using the menu on the left panel at the web console, as shown below:



Although multiple *orgs* can be created under one account, just one should suffice for individuals or small organizations. You may like to name that after your or your company's name. For example, we have named ours as *naturalprogrammer*.

Creating an *org* will automatically create a *space*, named *development*, which you can rename if you like. For *naturalprogrammer*, we renamed it to *test*. You may also like to create more spaces, e.g. *production*, and assign correct rights to your team members to access correct spaces.

Naming apps

Every *app* in Cloud Foundry will have a unique name. Names are made of alphanumeric characters without any space. For example, for our application, we may prefer to have the names *spring-tutorial-dev*, *spring-tutorial-test*, and *spring-tutorial* for development, test and production spaces respectively.

As you will see, dashes and underscores are allowed in app names, although the documentation is silent about it.

Deploying

For deploying an application to Pivotal Cloud Foundry, its [Command Line Interface](#) (CLI) seems to be the primary tool. Another alternative is the [Boot Dashboard](#) of STS, but it's yet to catch up with the CLI.

So, let's see how to deploy our application using the CLI.

Installing the CLI

Installing the CLI is easy, just follow its well written [documentation](#).

Packaging the application

To push our application using the CLI, we first need to package it. For packaging, the maven command `maven clean package` can be used. But, using STS, we find it easier to package an application by right clicking on the project and then using the *Debug -> Maven clean* followed by *Debug -> Maven install* context menu options. This would make our WAR, e.g. `spring-tutorial-0.0.1-SNAPSHOT.war`, inside the *target* folder of the project.

Creating a MySQL service

Our application uses a MySQL database service. Our local MySQL installation was being used for this when running locally. When deploying to PWS, we can hire such services from its marketplace. So, login to PWS web console, go to the space where the application will be deployed, and then create a service by using the *Add Service* button as shown in the screenshot below.

The screenshot shows the Pivotal Web Services console for the 'test' space. The left sidebar contains navigation links: Marketplace, Docs, Support, Tools, Blog, and Status. The main content area has tabs for 'Overview' and 'Settings'. Under 'Overview', there are two sections: 'Apps' and 'Services'. The 'Apps' section contains a table with columns: NAME, INSTAN..., MEMORY, LAST P..., and ROUTE. It lists three applications: 's2-social-demo', 'spring-sample-app', and 'spring-tutorial-test', each with 1 instance and 1024MB memory. The 'Services' section contains a table with columns: SERVICE, NAME, BOUND APPS, and PLAN. It shows a 'ClearDB MySQL Da' service named 'spring-soc...' with 1 instance and a 'free - (MONTHLY)' plan. The 'Add Service' button is circled in red.

Name the service, as, say `spring-tutorial-test-db`. This name is going to be used later.

Creating a manifest yml file

When deploying the application, we need to specify some parameters. Those can be specified either in the command line, or in a *manifest* yml file. So, create a *manifest-test.yml* file in some folder, as below:

```
---
applications:
- name: np-spring-tutorial
  path: C:\opt\workspaces\spring-tutorial-v2\np-spring-tutorial\target\spring-
tutorial-0.0.1-SNAPSHOT.war
  env:
    application_url: http://np-spring-tutorial.cfapps.io
  services:
    - np-spring-tutorial-db
```

A manifest file can contain many more attributes - refer to its [documentation](#) for details. In our one above, notice the following:

1. We have given the [name](#) of the application as *np-spring-tutorial*. Be careful to choose a unique name.
2. As we used the name *np-spring-tutorial*, by default, the application will be available at <http://np-spring-tutorial.cfapps.io>. To alter the route, you can specify the [host](#) and [domain](#) attributes.
3. The [path](#) attribute specifies the location of the JAR or WAR to push. It can either be an absolute path or a path relative to where the manifest file is stored.
4. You can specify a list of environment variables through the [env](#) block. Why do we need to define environment variables? To override the application properties. For example, if you look at the *application.properties* file in our resources folder, you'll find many properties, such as `application.url` and `spring.profiles.active`, that'll differ in different environments. To override those, we can define those as environment variables. Because periods aren't allowed in environment variables, Spring allows us to replace those with underscore. That's why we have an *application_url* environment variable defined in the manifest above. To know more about externalized configuration, refer Spring Boot's [reference material](#). The [Module III](#) of this course has a detailed pattern on how to properly organize application properties.
5. The list of services that we are going to use should be specified in the [services](#) block. As you see in the manifest above, it contains our database service that we had created earlier in this chapter.

There can be multiple manifest files, one for each environment. Also, they can contain sensitive information, e.g. the application password for the mail sending service. Hence, those should not be stored in the project where developers can see them. Instead, those should be stored externally, with proper restrictions.

In my PC, let me store the file at *C:\opt\manifest-test.yml* for now.

Pushing the application

We are now ready to push our WAR using CLI. For that, you first need to login using the command

```
cf login -a https://api.run.pivotal.io
```

If you have more than one org or space, you can use the options `-o ORG -s SPACE`.

This command will ask you for your PWS credentials, and return to the command prompt. Then, do the deployment using our manifest file, as below

```
cf push -f c:\opt\manifest-test.yml
```

It should take some time, and upon successful push, the application should be running at <http://np-spring-tutorial.cfapps.io>. To stop it, either use the web console or the `cf stop np-spring-tutorial` command.

The cf command is quite powerful, and in fact Cloud Foundry is quite feature rich. We just explored the minimum details to get you started. Refer to its [documentation](#) for greater details.

Also, the web console is quite intuitive, and you can learn a lot just by exploring that. Do that.

28 Next Steps

So, we have come to the end of this book, i.e. the Module II of our [Spring Framework for the Real World – Beginner to Expert](#) course.

You should now have a good grasp of the essential features and patterns of Spring for real-world application development. Throughout this book, we have provided links for further exploration, which should be of good use to you when required.

In the next module, i.e. [Module III](#), we'll cover many more common real-world use cases, useful particularly while coding REST APIs or JSON Web Services. In today's world of rich client based applications, e.g. *Mobo* or *JavaScript* based applications, coding APIs is vital, and so, don't skip the next module!

[Click here](#) to view its content and introduction, and [visit here](#) to buy it.