Spring Framework 4 And Dependency Injection For Beginners

Spring Framework Getting Started And Dependency Injection Fundamentals For Real World Application Development

Spring Framework for the Real World – Beginner to Expert, Module I

Second Edition, May 2016

Sanjay Patel

www.naturalprogrammer.com

Contents

1 Introduction....6

What is Spring Framework....6

What is Spring Boot....6

Why this book....6

What's covered....7

How to go through this book....7

Prerequisites....7

Help and support....8

Meet the author....8

Editions....8

2 Hello, world....9

Creating a new Spring project....9

Project metadata....10

Source code so far....12

Glancing at the source code....13

Running the application....16

Coding a Hello Controller....16

Source code so far....18

3 DI – Setting The Stage....19

The MailSender interface....19

MockMailSender....19

SmtpMailSender....20

Using the service....20

Source code so far....21

4 DI – The Basics....22

The Problem....22

The Spring way 22
So, what's happening? 23
Summary 23
Source code so far 24
5 Injecting Beans 25
Dependency injection by name and type 25
Multiple beans – how to inject the right one? 26
Source code so far 27
Setter Injection 27
@Autowired 27
Bean qualifiers 28
Source code so far 28
@Inject 29
@Resource, @Autowired or @Inject? 29
Property, setter or constructor injection? 29
Conclusion 29
6 Configuring Beans 30
Component annotations 30
Component scanning 30
Bean names 31
Configuration classes 31
Source code so far 32
XML Configuration 32
7 Application properties 33
Deploying to a different environment 34
Source code so far 35
8 Spring profiles 36
Source code so far 37
9 Conditional annotations 38

A bit of caution 39
Spring Boot Developer Tools 39
Source code so far 41
10 Sending SMTP Mails 42
Configuring a JavaMailSender 42
Coding SmtpMailSender 42
Source code so far 45
11 Configuring Beans contd 46
@Bean in @Components 48
Source code so far 48
12 Next Steps 49

1 Introduction

PREFER VIDEO VERSION?

If you prefer videos over books, visit here. However, we recommend our books over videos, because they will be more precise, will have links that you can explore, code snippets that you can copy, etc. Also, we update our books more frequently than our videos.

What is Spring Framework

Spring is one of the most popular frameworks for developing web and enterprise applications. The core strength of Spring framework is its powerful *dependency injection* capabilities, which makes it a component based framework. This enables us to easily assemble an application by using some built-in, third party and self-coded components.

In addition to the core framework, Spring has many other projects, such as Spring Data and Spring Security. Many of these again have multiple modules. For example, Spring Data has many modules such as Spring Data JPA and Spring Data MongoDB. All these projects and modules come with rich sets of built-in components, which can be assembled along with your own components containing your business logic. Great applications can be developed this way.

What is Spring Boot

But, assembling a Spring project is tricky, because, you need to handpick the right JARs and components, and configure and assemble those properly. That's why Spring came up with a project called Spring Boot, which does it automatically, based on some factors like what JARs and application properties are available.

That said, Spring Boot doesn't take the control away from you – it does allow you to manually configure a piece, overriding its automation.

In summary, Spring Boot makes a developer's life very easy, and is highly recommended nowadays for almost all kinds of projects. In fact, the sample project in this book uses Spring Boot.

Why this book

As discussed above, Dependency Injection (DI), or Inversion of Control (IoC) is the core function of Spring Framework. When learning Spring, it therefore becomes important to grasp its DI features first.

But, Spring DI is very rich, and so learning it becomes a challenge. Spring's documentation, although great for reference, doesn't serve as a step-by-step learning material. On the other hand, most books and tutorials tend to rush towards teaching the modules of practical utility, such as Spring Boot or MVC, rather than focusing enough on the DI concepts first. That too, most of these books, tutorials or blog posts still use XML configuration, rather than the recommended Java and annotation based configuration.

Hence, we came up with this lucid beginners' book and the video tutorial. After going through this book, you should have a solid grasp of Spring's dependency injection concepts, and how to use Java and annotation based configuration effectively. You can then feel confident to pursue learning other Spring modules.

In summary, if you are new to Spring Framework, this is an excellent place to start learning it.

To know more about why this book and the big picture, visit the landing page of our unique course Spring Framework for the Real World – Beginner to Expert. This book serves as *Module I* of that course.

What's covered

In this book, we are first going to learn how to create a new Spring project, and code some hello world stuff. Then, we are going to learn the basics of dependency injection and the commonly used dependency injection features of Spring framework, using Java configuration.

We are also going to cover the key features useful in real world development, like *Spring profiles*, *application properties*, *conditional annotations* etc., which are very useful in real world development.

As a bonus, you will also learn how to send SMTP mails from Spring applications.

How to go through this book

This book is a complete hands on course. 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.

Prerequisites

We expect you to already know core Java and its object oriented features like *classes* and *interfaces*. Knowledge of web development will help but isn't mandatory.

To follow the practical steps in the chapters, you also should have JDK 8 and some IDE installed. I use Oracle JDK 8 and Spring Tool Suite on a Windows 10 machine.

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 – February 2016

Second Edition – May 2016

2 Hello, world

So, let's begin by creating a "Hello, world" application!

Creating a new Spring project

There are multiple ways to create a new Spring project. The common ways are discussed below.

Creating from scratch

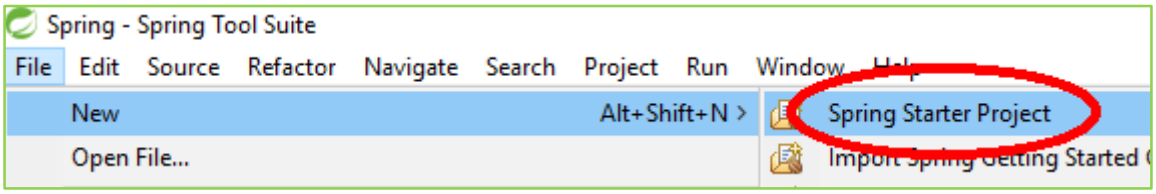
Because Spring projects are standard Java projects, you can just create those manually. In fact, the Spring guides like this one show how to do it that way. We won't recommend it though, because there are easier ways available.

Using start.spring.io

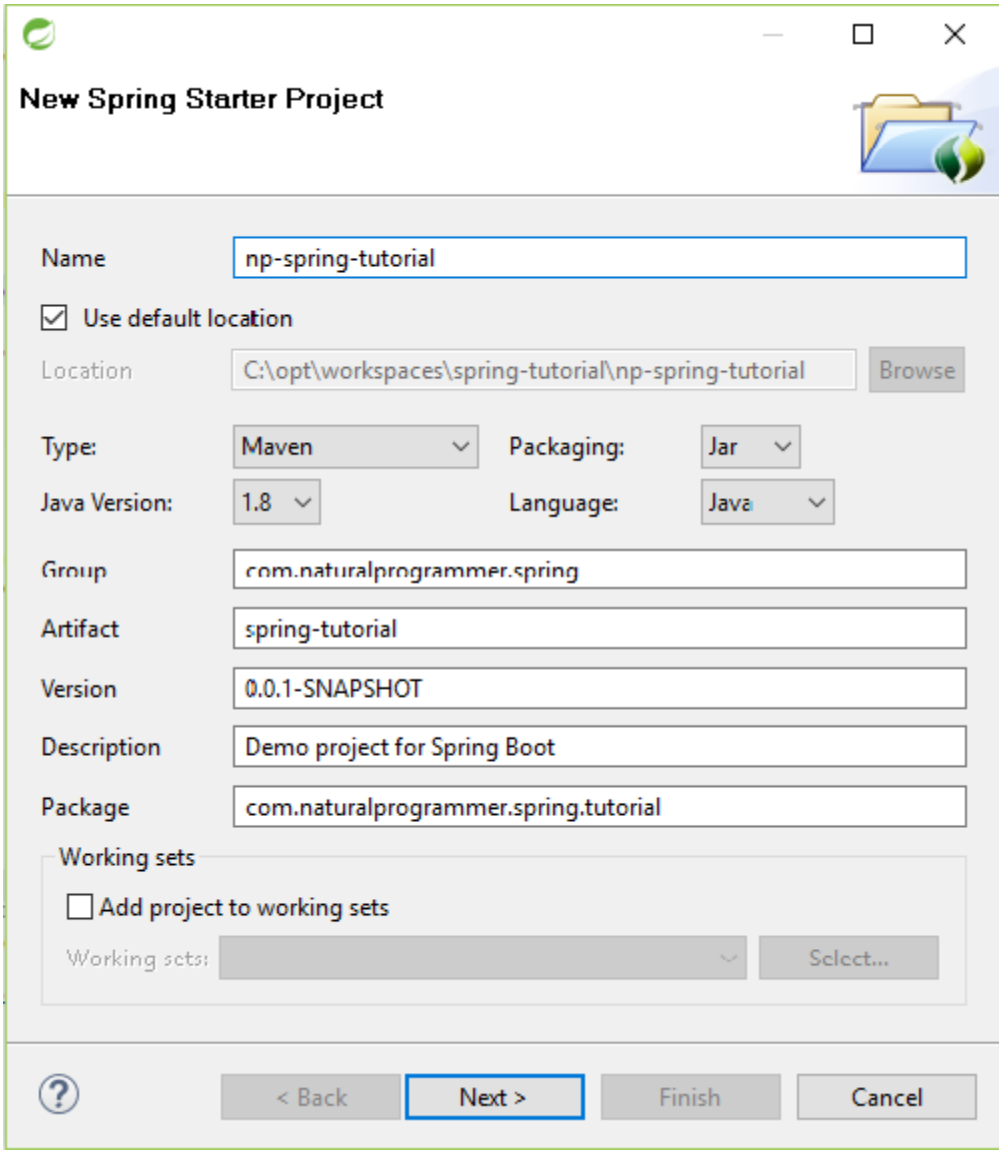
For easily creating new projects, Spring provides a wizard at start.spring.io. The wizard creates a zip file that you can download, unzip and import in your IDE.

Using an IDE

IDEs go one step further. For example, Spring Tool Suite provides a *Spring Starter Project* wizard, which you can access as shown below.



The wizard looks as below:



It actually connects to *start.spring.io* under the hood, and creates a project in the workspace.

In fact, this is our preferred way.

Project metadata

So, let's go ahead creating a new project. You can use either *start.spring.io* or an IDE. Whatever you do, you'll need to feed the following metadata.

Name

For the name of the project, we recommend using small alphanumeric characters and dashes only, e.g. *np-spring-tutorial*.

Type

It's the build tool to use. If you don't know about build tools, they help us compile and package our projects. You don't need to be a master of these tools to follow this book – we are going to discuss whatever is needed. So, if you don't know about these, no problem.

You'll find here a couple of options: *Maven* and *Gradle*. Maven is much more popular than Gradle, whereas Gradle is a new one, which was introduced to address some of the things which Maven couldn't do.

Unless there is a specific need for Gradle, we prefer to go with Maven, because it's widely adopted.

Packaging

You'll find two options for packaging – *JAR* and *WAR*.

The executable of our application is going to be a *fat* JAR or WAR, with tomcat and all the dependent JARs embedded in it. That JAR or WAR can run standalone, with the standard

```
java -jar xyz.jar
```

command. This seems to be the preferable way of packaging web applications nowadays, because that eases deploying the applications, e.g. at cloud services like Pivotal Cloud Foundry.

Let's choose JAR packaging. Using JSP needs WAR packaging, but we'll talk about that later.

Java version

Of course we will like to use Java 1.8, although Spring still supports 1.7.

Group and Artifact

Group and *Artifact* uniquely identify a project. Group uniquely identifies a group to which the project belongs, whereas Artifact uniquely identifies the project inside the group.

A common practice to name a group is to begin it with a domain name owned by us, in reverse order. For example, as we own *naturalprogrammer.com*, we use a group name `com.naturalprogrammer.spring` for all our spring projects. No problem if you don't own a domain – just use some name which you feel will be unique. Unless the project is going to be used along with some other clashing projects, it will not be a problem.

For the artifact, we prefer to use a few alphanumeric words separated by dashes, e.g. `spring-tutorial`.

Version

This will be the version of the project. It you are unsure what's this, just leave it untouched.

Description

A one line description of the project.

Package

This will be the name of the root package of your application. The common practice for naming it is to join the group and the artifact in some way, e.g. `com.naturalprogrammer.spring.tutorial`. Remember that "-" isn't allowed here. You should either omit those, replace those with underscores, or do something about those.

If you are using *Spring Tool Suite*, now press next to see the rest of the fields, which are discussed below.

Boot version

This is the version of Spring Boot to use. Normally we'd choose the latest stable release.

Dependencies

This section asks us to choose the dependencies, i.e. the Spring and other projects that will be used in our project. Whatever we choose here will go into the configuration file of the build tool. The build tool will then automatically fetch the needed JARs and put those in the classpath.

So, choose just *Web* for now. More dependencies can easily be added into the configuration file later.

Press *Next*, and you'll see a message indicating that the service at *start.spring.io* is going to be used. Pressing *Finish* will then create the project in your workspace. If you are doing this first time, it will take some time to fetch all the JARs to your local system.

> **TIME TO PUT YOUR PROJECT UNDER VERSION CONTROL!**
>
> If you are coding a serious project, this is the right time to put your source code under version control.
>
> Git being the most popular version control system nowadays, that should be your first choice, unless you have some other factors influencing your decision. If you are new to Git, go through our Rapid Git Tutorial to learn how to use Git in an Eclipse based IDE, like STS.

Source code so far

Click here to browse the complete source code of the project so far.

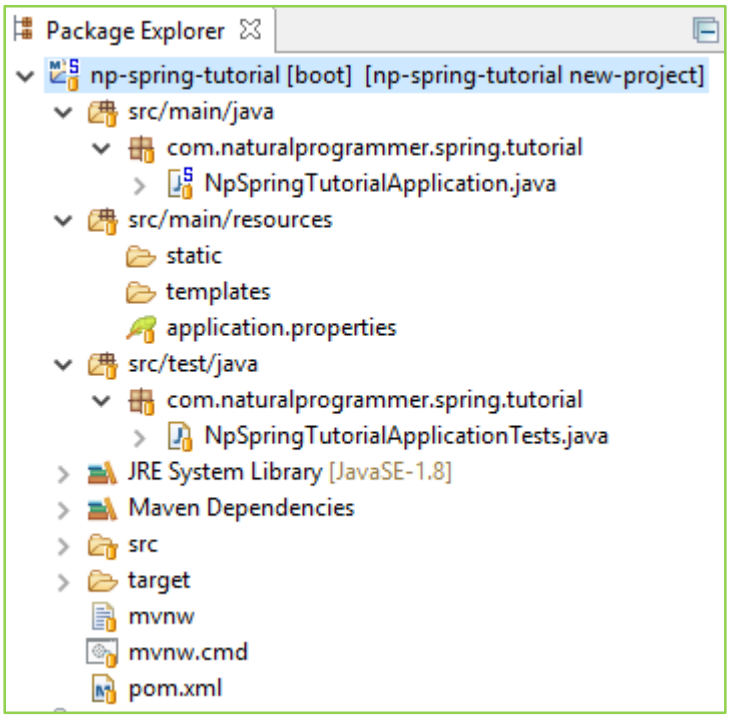
If you want to download it as a zip, click here. The zip can be extracted in some folder, and then imported into STS by using *File – Import – Maven – Existing Maven Project*.

> If you know Git, you will have figured out that the project is hosted at GitHub. Branches are created corresponding to the topics of this book. `new-project` is the branch name for the source code so far, as you would have figured out by looking at the links above.

If you know Git, you should prefer to check out the project rather than downloading the zips.

Glancing at the source code

Browsing the source code of the generated project will tell you that it's just a java project, with a standard maven directory layout:



Pom.xml

At the root of the project, you'll find a file named `pom.xml`, which is the configuration file that maven uses to know about the project. It will be looking as below:

```
<?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>jar</packaging>

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

```
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>1.3.2.RELEASE</version>
		<relativePath/> <!-- lookup parent from repository -->
	</parent>

	<properties>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<java.version>1.8</java.version>
	</properties>

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

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
		</dependency>
	</dependencies>

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

</project>
```

Notice that all the metadata that we fed when creating the project, except the root package, are found here. Also notice the `dependencies` section:

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

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

This section tells Maven what all other projects this project depends on, so that Maven will pull the required JARs and put those in the classpath. Maven does this recursively, i.e., the JARs on which the dependencies depend on will also be pulled and put onto the classpath, and so on.

Below the dependencies is a *build* section with the `spring-boot-maven-plugin`, as below:

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

This plugin is used for packaging the application as a fat JAR or WAR, which will have tomcat and all the dependent JARs embedded. That fat JAR can then be run just by using `java -jar`, as below:

```
java -jar spring-tutorial-0.0.1-SNAPSHOT.jar
```

Java classes

The wizard will also have created a couple of Java classes, as shown below.

NpSpringTutorialApplication.java

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

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

@SpringBootApplication
public class NpSpringTutorialApplication {

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

`SpringApplication.run` that you see above configures and runs the application.

NpSpringTutorialApplicationTests.java

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

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

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = NpSpringTutorialApplication.class)
```

```
@WebAppConfiguration
public class NpSpringTutorialApplicationTests {

	@Test
	public void contextLoads() {
	}
}
```

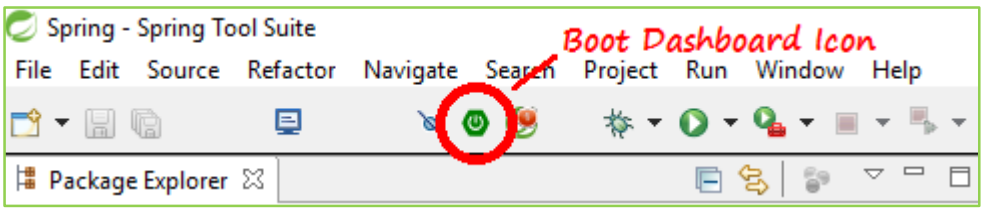
This is just an empty test class that you can fill later.

Other files

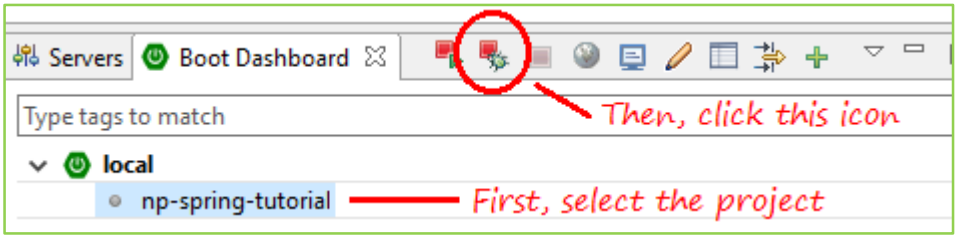
You'll also find a couple of empty folders, and an empty `application.properties` file in *src/main/resources*. We'll discuss about these at appropriate time.

Running the application

In STS, there will be multiple ways to run the application. Using the *Boot Dashboard* seems to be the best among those. It can be opened by clicking on the icon shown below:



Boot Dashboard will look as below:



There, you can select the project that you want to run, and then click on the debug (or run) button as shown above.

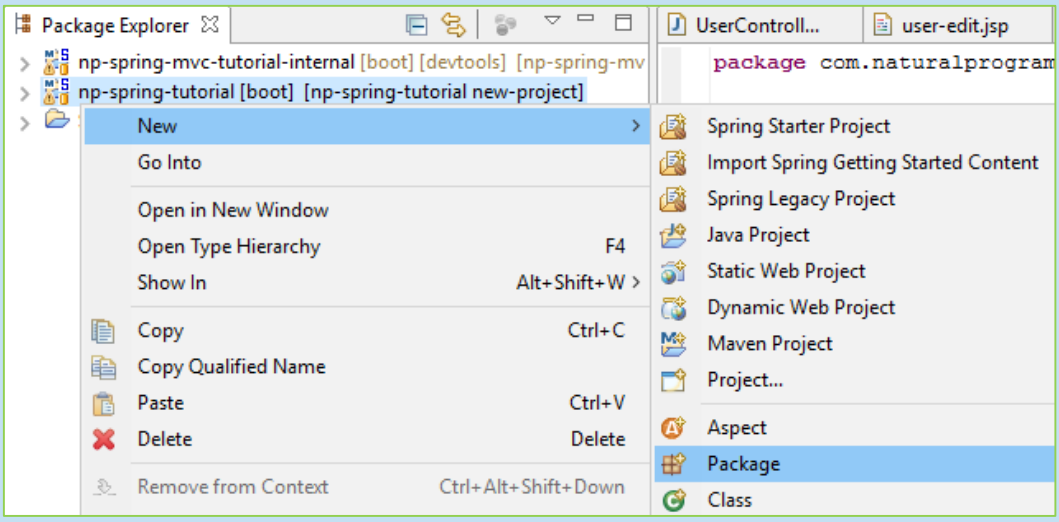
Other ways to run the application in STS will be to select either the project or the main class, click on the *Debug* or *Run* button on the toolbar, and then to select either *Java Application* or *Spring Boot App*. You'll get the same options if you right click on the project or the main class.

Coding a Hello Controller

Running the application now will do nothing but displaying a few lines scrolling on the console. To see some action, let's code a *controller* class and a *request handler* method in that.

So, create a new package, say `com.naturalprogrammer.spring.tutorial.controllers`, for putting the controller classes.

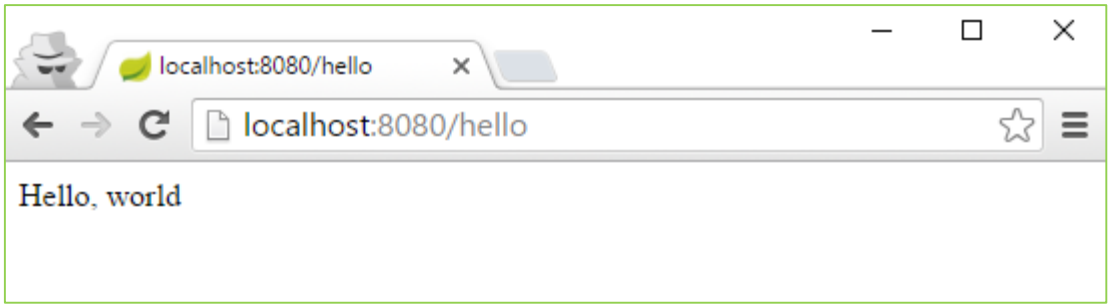
Don't know how to create a package in STS? Right click on the project, go to *New* and then *Package*, as shown below:



Inside that package, create a new class, say `HelloController`, looking as below:

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

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

	@RequestMapping("/hello")
	public String hello() {
		return "Hello, world";
	}
}
```

Now, press the Debug button on the Boot Dashboard to rerun the application. Then, visit http://localhost:8080/hello on a browser and you'll see *Hello, world.*



So, how does this work?

Not yet the right time to discuss about that. Instead, next chapter onwards, we'll first explore Spring's DI features.

Source code so far

- Browse online
- Download zip
- See changes

3 DI – Setting The Stage

In this chapter, we're going to code a simple service for sending emails, which will help us understand dependency injection concepts in later chapters.

Specifically, we are going to code an interface, namely *MailSender*, and a couple of its implementations, namely *SmtpMailSender* and *MockMailSender*. SmtpMailSender will be used for sending SMTP mails, whereas MockMailSender will just write the mail content to the console.

Let's start by creating a new package, say `com.naturalprogrammer.spring.tutorial.mail`, to put these classes in. Note that the new package must be a sub-package of the root package, i.e. `com.naturalprogrammer.spring.tutorial`. Why so is discussed in a later chapter.

The MailSender interface

MailSender will be a simple interface, with just a `send` method. Code it as below:

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

public interface MailSender {

	void send(String to, String subject, String body);
}
```

As you see, the *send* method takes three parameters:

1. `to`: the email id to which the mail will be sent
2. `subject`: the subject of the mail
3. `body`: the body of the mail

MockMailSender

Code the *MockMailSender* class as below:

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

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

public class MockMailSender implements MailSender {

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

	@Override
	public void send(String to, String subject, String body) {
		log.info("Sending mail to " + to);
```

```
            log.info("Subject: " + subject);
            log.info("Body: " + body);
      }
}
```

As you see, it just writes the mail onto the log.

> **LOGGING IN SPRING BOOT**
>
> Time to talk a bit about logging in Spring Boot. In MockMailSender above, we've used *commons logging* for the logging. But it's actually not commons logging. Instead, it's a wrapper around *SLF4J*, and SLF4J in turn will be using *Logback* by default.
>
> The log, by default is written to the console, but it can be configured the way you want.
>
> To know more about logging in Spring Boot, see its documentation here.

SmtpMailSender

Although SmtpMailSender will eventually send actual, SMTP mails, for now, let it also just write something to the log. So, code it as below:

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

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

public class SmtpMailSender implements MailSender {

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

      @Override
      public void send(String to, String subject, String body) {
            log.info("Sending SMTP mail to " + to);
            log.info("Subject: " + subject);
            log.info("Body: " + body);
      }
}
```

Notice that the only difference between it and the *MockMailSender* is word SMTP.

Using the service

For trying out the service, let's now code a *MailController* class, in the *.controllers* package. Code it as below:

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

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

import com.naturalprogrammer.spring.tutorial.mail.MailSender;
import com.naturalprogrammer.spring.tutorial.mail.MockMailSender;

@RestController
public class MailController {

	private MailSender mailSender = new MockMailSender();

	@RequestMapping("/mail")
	public String sendMail() {

		mailSender.send("abc@example.com", "Some subject", "the content");

		return "Mail sent";
	}
}
```

Notice that the instance variable *mailSender* is instantiated with a new *MockMailSender*. It's then used in the request handler method to send the mail.

If you now run the application and visit http://localhost:8080/mail, you should see "Mail sent," and in the console you should see the log lines.

Source code so far

- Browse online
- Download zip
- See changes

4 DI – The Basics

There is some evil code in our *MailController* class that we coded in the last chapter. Let's see what's that, and how to use dependency injection to fix that.

The Problem

Notice that, in *MailController*, we have instantiated a *MailSender* this way:

```
	private MailSender mailSender = new MockMailSender();
```

Hardcoding the concrete class MockMailSender above tightly coupled it to our MailController class. In other words, MailController is now depending on the concrete class MockMailSender. That means, if we ever change our mind to use SmtpMailSender instead of MockMailSender, we'll need to alter the source code, this way:

```
	private MailSender mailSender = new SmtpMailSenderMockMailSender();
```

This kind of tight coupling with concrete classes is evil. Also, it won't let us switch the implementation based on environments. For example, we'll not be able to use MockMailSender in development environment, whereas SmtpMailSender in production.

This is where dependency injection comes in.

The Spring way

Using dependency injection of Spring, here is a solution to the above problem:

1. Annotate MockMailSender class definition with `@Component`:

```
...

import org.springframework.stereotype.Component;

@Component
public class MockMailSender implements MailSender {
	...
}
```

2. Change instantiating MockMailSender as below:

```
...

import javax.annotation.Resource;
```

```
...

@RestController
public class MailController {

	@Resource
	private MailSender mailSender = new MockMailSender();

	...
}
```

If you now run the application and visit http://localhost:8080/mail, you should see it working.

So, what's happening?

This is how it works:

1. By annotating MockMailSender with *@Component*, we told Spring to create an instance of MockMailSender and keep it at *some place* when the application starts.
2. By annotating mailSender variable with *@Resource*, we told Spring to go to *that place*, search for an object of type MailSender, and assign a reference of that to this variable.

That place where Spring kept the MockMailSender instance is called *Application Context*.

So, to summarize, Spring maintains an Application Context where it keeps objects, which are also called *Beans*, instantiated. When a Spring application starts, Spring looks at some metadata, like the @Component annotation, for building the application context. The @RestController annotation is also like the @Component annotation – Spring also creates instances of the classes annotated with @RestController. That's why when our application started, Spring created an object of our MailController class as well. That time, it saw that the mailSender field was annotated with the @Resource annotation, and it then assigned mailSender a reference to the MockMailSender object that it had already stored in the application context.

Summary

So, when developing a Spring application, you basically have to know two things:

1. How to tell Spring to put objects into the application context. i.e. what kind of metadata, like the @Component annotation to use.
2. How to tell Spring to inject in your code references to the objects in the application context, i.e. what kind of metadata, like the @Resource annotation to use.

There are multiple ways to do these – Spring's DI is very rich. Next chapters onwards, we are going to go more deep into these.

Source code so far

- Browse online
- Download zip
- See changes

5 Injecting Beans

In the last chapter, we saw how to use the @Resource annotation to get a reference to a bean. In this chapter, we'll discuss that in more details.

Dependency injection by name and type

The documentation of @Resource says that it can take a name attribute, as below:

```
@Resource(name="abc")
private MailSender mailSender;
```

Let's tell you now that every bean in the application context is given a *name*. By default, the name of the bean will be its class name in *camelCase*. That means, the MockMailSender instance, which will be put in the application context as a result of being annotated with @Component, will be named as `mockMailSender`. To change the default name, we can supply a *value* attribute to @Component, as below:

```
@Component("abc")
public class MockMailSender implements MailSender {
	...
}
```

Let's now get back to the discussion on @Resource. It takes a *name* attribute, and Spring then tries to find a bean with that name and inject that to the variable. This behavior is called dependency injection *by name*.

In the absence of the name attribute, as below, the variable name becomes the name:

```
@Resource
private MailSender mailSender;
```

So, above, the name will be assumed as *mailSender*, i.e. the name of the variable. Spring will then try to search for a bean with that name.

But the search will fail, because the MockMailSender instance will have been named as *mockMailSender*.

Spring will then fall back to dependency injection *by type*. That means, a bean of the same type, i.e. MailSender, will be searched for. This time the MockMailSender instance will be found, because it's a subtype of MailSender.

Multiple beans – how to inject the right one?

Let's now see what happens if more than one bean become eligible for an injection. To illustrate this, annotate SmtpMailSender as well with @Component:

```
@Component
public class SmtpMailSender implements MailSender {
	...
}
```

When you now try to run the application, on the console you'll see an error. It's because Spring, when trying to find a suitable bean for injecting to the mailSender variable, finds two beans – the instances of both MockMailSender and SmtpMailSender.

There are multiple ways to tackle this. Let's discuss a few.

Specifying a name in @Resource

You can provide to @Resource a name attribute having value `smtpMailSender`:

```
@Resource(name="smtpMailSender")
private MailSender mailSender;
```

Spring will then be able to inject the exact bean without any error.

Matching a bean name to the variable name

Another alternative will be to give a name to the bean to match the resource name. For example, remove the name attribute from @Resource above, so that `mailSender` (the name of the variable) becomes the name by default. Then, provide the name `mailSender` to the SmtpMailSender bean, as below:

```
@Component("mailSender")
public class SmtpMailSender implements MailSender {
	...
}
```

Using @Primary

The `@Primary` annotation can be used to tell Spring to choose a bean when multiple beans become eligible for an injection. To try it out, annotate SmtpMailSender with @Primary, as below:

```
@Primary
@Component~~("mailSender")~~
public class SmtpMailSender implements MailSender {
	...
}
```

Run the application now, and it'll work well.

Source code so far

- Browse online
- Download zip
- See changes

Setter Injection

@Resource works not only on instance variables, but also on setter methods. For example, alter the injection code as below:

```
@Resource
private MailSender mailSender;

@Resource
public void setMailSender(MailSender mailSender) {
	this.mailSender = mailSender;
}
```

When a component, e.g. the MailController, is instantiated, all its setter methods annotated with @Resource are called, with the parameters injected.

@Autowired

Similar to @Resource, we also have `@Autowired`. It's more powerful – it not only works on setters, but on any method, including the constructors. For example, using @Autowired, we can replace our setter injection with *constructor injection*, this way:

```
private MailSender mailSender;

@Resource
public void setMailSender(MailSender mailSender) {
	this.mailSender = mailSender;
}

@Autowired
public MailController(MailSender mailSender) {
	this.mailSender = mailSender;
}
```

Try it running, and it'll work.

However, @Autowired does not take the *name* parameter. Instead, we need to use @Qualifier along with @Autowired.

So, what's *qualifier*?

Bean qualifiers

Beans in the application context, apart from having a name, also have another attribute called *qualifier*. Annotating a bean definition with `@Qualifier` sets its qualifier. To test it out, remove the @Primary annotation from SmtpMailSender, and provide a qualifier to it, as below:

```
@Primary
@Component
@Qualifier("smtp")
public class SmtpMailSender implements MailSender {
	...
}
```

This will set the qualifier of the smtpMailSender bean to *smtp*.

Next, annotate the MailSender parameter of the MailController constructor with @Qualifier, as below:

```
@Autowired
public MailController(@Qualifier("smtp") MailSender mailSender) {
	this.mailSender = mailSender;
}
```

Spring will then be able to find the bean with qualifier *smtp*, and the application will work well.

Default qualifier

By default, the qualifier of a bean will be same as its name. To test it out, remove the qualifier annotation from SmtpMailSender, as below:

```
@Component
@Qualifier("smtp")
public class SmtpMailSender implements MailSender {
	...
}
```

The qualifier of *smtpMailSender* will now take the default value – `smtpMailSender`. Now, change the qualifier in MailSender constructor to match that:

```
@Autowired
public MailController(@Qualifier("smtpMailSender") MailSender mailSender) {
	this.mailSender = mailSender;
}
```

Run the application now, and it'll work.

Source code so far

- Browse online

- Download zip
- See changes

@Inject

Spring also has `@Inject`, which is synonymous to @Autowired. It's part of the Java CDI JSR-299 standard, and so Spring thought to support it. It's not used that commonly, though.

@Resource, @Autowired or @Inject?

So, @Resource, @Autowired or @Inject? Which one to prefer?

In case of the multiple-bean problem, @Resource, when supplied with a name, seems cleaner than using @Autowired or @Inject along with @Qualifier. But, @Autowired or @Inject is more powerful. Probably that's why people seem to be preferring @Autowired nowadays. In fact, the Spring team seem to be recommending @Autowired over the others.

Functionally, there are some subtle differences between these, which may not matter in most cases though. For example, as we discussed earlier, @Resource first tries to match *by name*, and then falls back to *by type*. In contrast, @Autowired and @Inject first try to match *by type*. David Kessler has written an interesting article on it.

Property, setter or constructor injection?

This is another question often asked. Ideally, setter or constructor injection would be preferred over property injection, because they might ease creation of mock objects when testing. Constructor injection would be slightly preferred over setter injection, because it ensures that no injection is accidentally left out.

In fact, the Spring team seem to be recommending constructor injection, as you might already have noticed in the link that was provided above.

But, the code look cleanest when doing property injection. Hence, I personally prefer to start with property injection, and refactor the code later when needed.

Conclusion

We now have a basic knowledge of how to inject beans from the application context in to our code, i.e. how to use the @Resource annotation, the @Autowired annotation, etc. Next chapter onwards, we will be focusing on the other side of the story, i.e., how to put beans into the application context.

6 Configuring Beans

This chapter onwards, we'll focus on how to configure the beans, i.e. how to put objects into the application context.

Component annotations

Spring creates beans for classes annotated with @Component or one of its specializations, namely:

- @RestController
- @Controller
- @Service
- @Repository
- @Configuration
- @SpringBootApplication

In fact, we have already seen @Component and @RestController working.

Component scanning

The application context is created when the application starts, by calling `SpringApplication.run` in the main method, which you can find in the *NpSpringTutorialApplication* class.

Notice that *NpSpringTutorialApplication* is annotated with `@SpringBootApplication`. This tells Spring to scan for classes that are annotated with @Component, or any of its specialization such as @RestController, and create beans for those.

By default, classes will be scanned for only inside the package of the class which is annotated with @SpringBootApplication, including all sub-packages. If you want some other packages to be scanned as well, there are a few ways to do so. For example, you could pass a `scanBasePackageClasses` attribute to @SpringBootApplication, as below:

```
@SpringBootApplication(scanBasePackageClasses = {
      NpSpringTutorialApplication.class,
      SomeClassInTheOtherPackage.class})
```

The above will make Spring scan for the packages of all the given classes, including their sub-packages.

Bean names

As discussed earlier, beans have names. By default, it'll be the class name in camel case. It can be explicitly set by passing a *value* attribute to @Component or any of its specialization, as below:

```
@Component("mailSender")
public class SmtpMailSender implements MailSender {
	...
}
```

Configuration classes

If the mail sender classes were not written by us, but were part of some third party JAR, we'll not be able to annotate those with @Component, because we'll not have access to their source code.

In such case, we could use Java configuration classes. To illustrate, let's remove @Component and any other annotations from SmtpMailSender and MockMailSender. Let them be plain classes, without being aware that they are going to be used by a spring application:

```
@Component
public class SmtpMailSender implements MailSender {
	...
}
```

```
@Component
public class MockMailSender implements MailSender {
	...
}
```

Let's now have a class, say `MailConfig`, looking as below. It can reside in any package that will be scanned by Spring; so put it in the *mail* package.

```
@Configuration
public class MailConfig {

	@Bean
	public MailSender mockMailSender() {
		return new MockMailSender();
	}

	@Bean
	public MailSender smtpMailSender() {
		return new SmtpMailSender();
	}
}
```

Try running the application, and it should work. How?

The `@Configuration` annotation tells Spring that it's a configuration class. When the application context is built, the methods that are annotated with `@Bean` inside such configuration classes are called, and their return values are stored as beans. The name of the methods become the names of the beans by default.

Source code so far

- Browse online
- Download zip
- See changes

XML Configuration

Like Java configuration classes, we can also use xml files to configure beans. In fact, that was the only way to configure beans in early days of Spring. But nowadays people seem to be avoiding XML configuration. There is ample of material available on the Internet if you want to learn XML configuration – so we'll not discuss it in this book.

7 Application properties

Before moving forward on dependency injection, this'll be a good time to discuss about *application properties*.

Every non-trivial application needs to have some properties which differ from deployment to deployment. For example, the database connection details will differ in production environment, test environment, and development environment.

Spring has multiple ways to handle this. When using Spring Boot, a good way to maintain properties is to use the *application.properties* file in the *src/main/resources* folder.

To have a taste of it, let's change the *HelloController* to show a variable message, say like this:

```
@RestController
public class HelloController {

	private String appName;

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

Let's now set `appName` from application properties. To do so, first add a line in *application.properties* in the *src/main/resources* folder, as below:

```
app.name = Development Env
```

> The '=' above can also be a ':'.

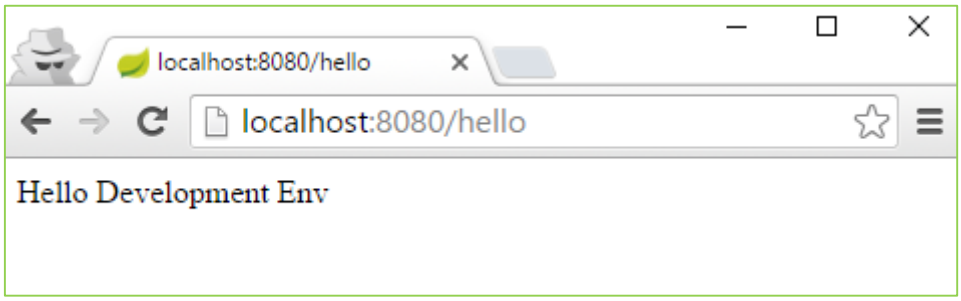
Then, annotate the *appName* variable in HelloController with a `@Value("${app.name}")` annotation, as below:

```
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;
	}
}
```

This will inject *app.name* property to the *appName* variable.

Now run the application and visit http://localhost:8080/hello. You should see the following message:



Deploying to a different environment

So, how to make the property value different in a different environment?

There are many ways to do it. A simple way, when deploying the application JAR, is to place a different *application.properties* in the same folder where you put the JAR. Property values in that external *application.properties* will then override that in *src/main/resources/application.properties*.

To see it working, let's package the application first. A simple way to do it in STS is to right click on the project in the package explorer and do *Debug As -> Maven install*. This will create the fat application JAR inside the target folder, named like *spring-tutorial-0.0.1-SNAPSHOT.jar*.

Copy that JAR to some other location, and put a new *application.properties* file in that location, looking as below:

```
app.name: Development Test Env
```

Then, open a console, cd to that folder and run the application using a command as below:

```
java -jar spring-tutorial-0.0.1-SNAPSHOT.jar
```

Now, visit http://localhost:8080/hello. You should see the overridden hello message.

So, this was a very simple example of setting property values externally. Spring Boot is very powerful in this area, providing lots of ways, suiting all kinds of scenarios. Spring Boot reference material has exhaustive documentation on it.

An elegant pattern on externalizing configuration in real-world applications is discussed in the *Module III* of our course Spring Framework for the Real World – Beginner to Expert.

Source code so far

- Browse online
- Download zip
- See changes

8 Spring profiles

A Spring application can be configured to behave differently in different environments using different *profiles*.

Let's say that you are developing an application for two kinds of businesses – *book shops* and *grocery shops*. Also, your application runs on different environments, say *development*, *test* and *production*. Under this scenario, you can plan to have *five* profiles:

- book
- grocery
- dev
- test
- prod

You can then configure a deployment to run using zero or more profiles, by providing a `spring.profiles.active` property. For example, to run the application using *book* and *dev* profile, add the following line in *application.properties*:

```
spring.profiles.active: book,dev
```

So, how to use this profile information?

You can configure your beans based on the active profiles. Let's say we are going to use the *MockMailSender* only in development environment, i.e. when the *dev* is in the active profiles list. Otherwise, we are going to use the *SmtpMailSender*. To configure so, annotate the bean methods with `@Profile`, as below:

```
@Configuration
public class MailConfig {

	@Bean
	@Profile("dev")
	public MailSender mockMailSender() {
		return new MockMailSender();
	}

	@Bean
	@Profile("!dev")
	public MailSender smtpMailSender() {
		return new SmtpMailSender();
	}
}
```

Now, remove the *@Qualifier* annotation in *MailController* constructor, because the above code will ensure that only one *MailSender* will be available in the application context:

```
@Autowired
public MailController(@Qualifier("smtpMailSender") MailSender mailSender) {
	this.mailSender = mailSender;
}
```

Test the application now by altering *spring.profiles.active*, and you'll see how the configured MailSender changes.

@Profile makes a Spring application highly configurable, and can be used not only on individual bean methods, but also on entire configuration classes.

Source code so far

- Browse online
- Download zip
- See changes

9 Conditional annotations

The *@Profile* annotation, as we saw, can be used to include or exclude beans based on the active profiles. Spring provides another, more generic, `@Conditional` annotation, which can be used to include or exclude beans based on arbitrary conditions that we can provide.

Spring Boot uses it vigorously to automatically configure the application context based on conditions like what classes are on our classpath, what properties are available in application.properties, what beans are already provided by the developer, etc. In fact, Spring Boot has got some variations of *@Conditional*, such as `@ConditionalOnClass`, `@ConditionalOnProperty` and `@ConditionalOnMissingBean`. See its reference material for more details.

Let's try an example. Let's say that we will use our *SmtpMailSender*, if in our application there is a `spring.mail.host` property pointing to some SMTP server, as below:

```
spring.mail.host = smtp.gmail.com
```

Otherwise, we'll use the MockMailSender.

To have so, change *MailConfig* as below:

```
@Configuration
public class MailConfig {

	@Bean
	@Profile("dev")
	@ConditionalOnProperty(name="spring.mail.host",
	                       havingValue="foo",
	                       matchIfMissing=true)
	public MailSender mockMailSender() {
		return new MockMailSender();
	}

	@Bean
	@Profile("!dev")
	@ConditionalOnProperty("spring.mail.host")
	public MailSender smtpMailSender() {
		return new SmtpMailSender();
	}
}
```

The first use of *@ConditionalOnProperty* above will configure a mockMailSender only if there is a property *spring.mail.host* with value `foo`, or there is no such property. When we will plan to use the mockMailSender, we will provide no such property, and so it will work.

The second use of *@ConditionalOnProperty* will configure an smtpMailSender when *spring.mail.host* is provided.

But note that, if you provide a property just as below, our application will fail!

```
spring.mail.host = foo
```

It's because both the conditions will be satisfied, and Spring will create both the beans. Then, the injection in the MailController constructor will find multiple beans, and will scream.

However, practically no host will have been named as foo, and so the above code will work.

Test the application out, and the program should work as expected. Refer the Javadoc of @ConditionalOnProperty to know more about it.

A bit of caution

A bit of caution before you dive in to explore the conditional annotations: Don't use *@ConditionalOnBean* and *@ConditionalOnMissingBean* unless you are sure of the bean creation order. For more details, see this stackoverflow post.

> Spring Boot's auto configuration controls the bean creation order, and so Spring Boot uses these without any problem, though.
>
> It's an advanced topic, and we don't want to boggle your mind with it now.

Spring Boot Developer Tools

Before finishing this chapter, let's tell you about *Spring Boot Developer Tools*, which can make our development process a bit easier.

When developing an application, we often keep altering the code, followed by restarting the app. Developer Tools automates this restart, after a code change.

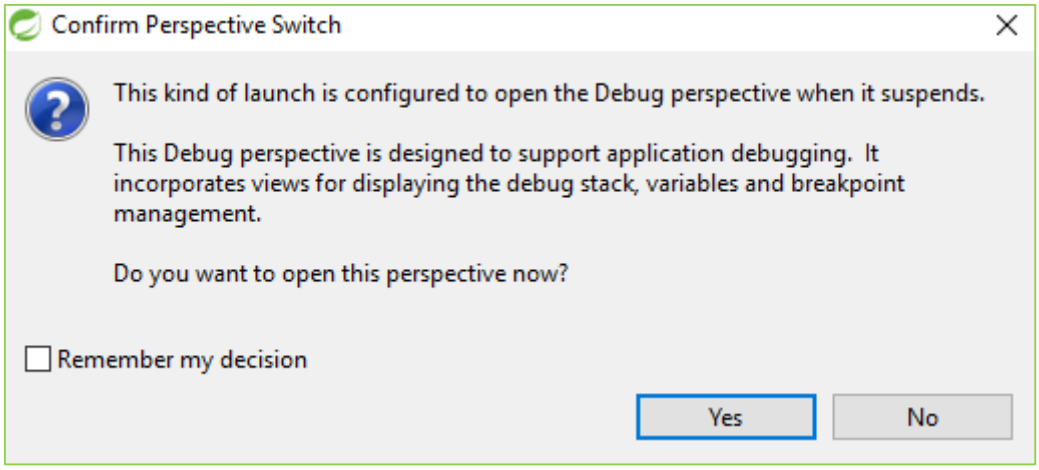
To have it in your project, just add the following dependency to *pom.xml*:

```
<?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">

      ...

      <dependencies>
```

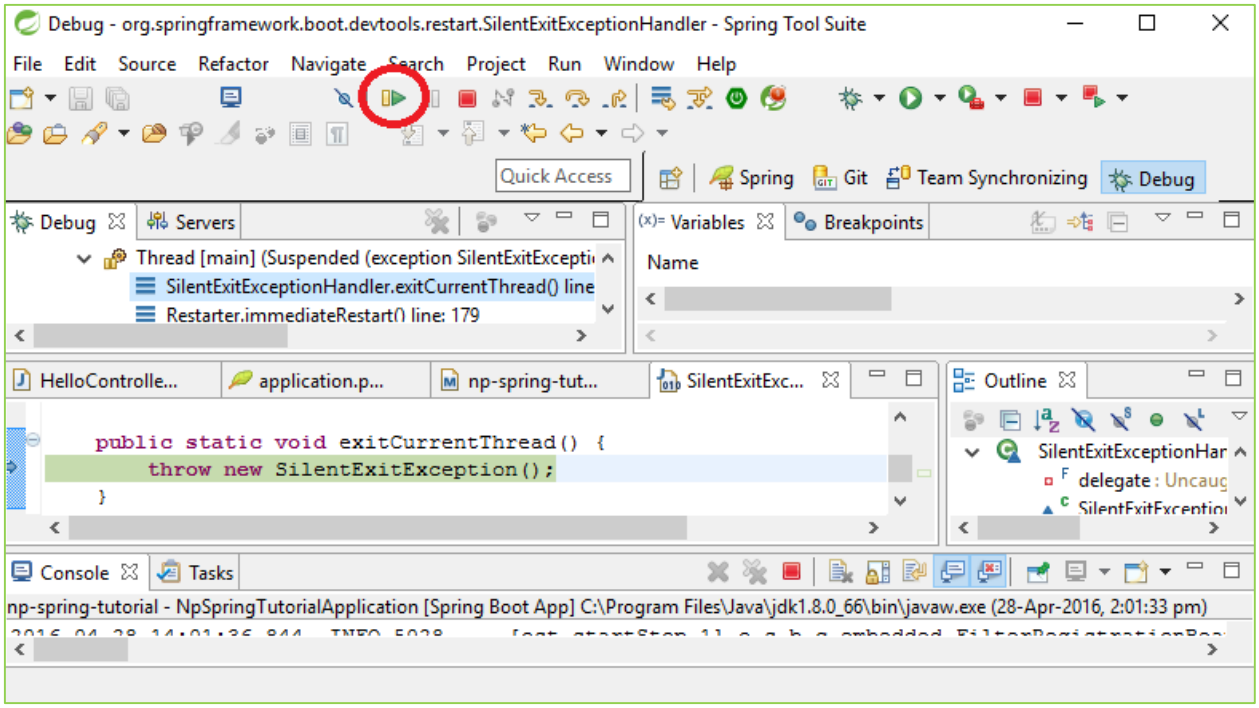
```
            ...

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <optional>true</optional>
        </dependency>

    </dependencies>

    ...

</project>
```

When you change some code now using STS, you'll see that the application will automatically restart.

However, you may now see the following dialog appearing when starting the application in debug mode:



On clicking yes, you'll see the following screen, in the *Debug* perspective:



It's no problem – just press *F8* or the *Resume* button as encircled above, and switch back to the *Spring* perspective.

Developer Tools can do many more things, like auto browser reload. Refer its documentation for more details.

Source code so far

- Browse online
- Download zip
- See changes

10 Sending SMTP Mails

In this chapter, we'll see how to send SMTP mails from Spring. We'll also learn some more dependency injection concepts of Spring in the process.

Configuring a JavaMailSender

Spring lets us send SMTP mails by using a JavaMailSender. For Spring Boot to configure it automatically , first add the following *spring-boot-starter-mail* dependency to your pom.xml.

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

Then, you need to specify a few properties in *application.properties*. Looking at MailProperties.java will give you a hint of what properties to specify. For example, for using Gmail service, you need to specify the following.

```
spring.mail.host = smtp.gmail.com
spring.mail.username = from_gmail_id@gmail.com
spring.mail.password = an_application_password

spring.mail.properties.mail.smtp.auth = true
spring.mail.properties.mail.smtp.socketFactory.port = 465
spring.mail.properties.mail.smtp.socketFactory.class = javax.net.ssl.SSLSocketFactory
spring.mail.properties.mail.smtp.socketFactory.fallback = false
spring.mail.properties.mail.smtp.ssl.enable = true
```

from_gmail_id@gmail.com above is the ID from which the mails will be sent. Replace it with an ID that you own, and an_application_password with one of your application passwords.

> Remember - for using Gmail service, you must have activated 2-step verification, and the password given above should be an application password.

Coding SmtpMailSender

Let's now update the code of *SmtpMailSender* to send actual mails.

We will be using the auto-configured *JavaMailSender*. So, we first will need to inject it to *SmtpMailSender*. As you know, had *SmtpMailSender* been annotated with @Component, we could

have used an instance variable annotated with *@Autowired* or *@Resource* for the injection. Instead, we are creating the bean in *MailConfig*. So, in *SmtpMailSender*, just add an instance variable and a setter method, as below:

```
...

import org.springframework.mail.javamail.JavaMailSender;

...

public class SmtpMailSender implements MailSender {

	...

	private JavaMailSender javaMailSender;

	public void setJavaMailSender(JavaMailSender javaMailSender) {
		this.javaMailSender = javaMailSender;
	}

	...
}
```

Alternatively, a constructor with a *JavaMailSender* parameter would also have worked.

Updating MailConfig

Then, in *MailConfig*, inject the *javaMailSender* explicitly to the *smtpMailSender*, as below:

```
@Configuration
public class MailConfig {

	...

	@Autowired
	private JavaMailSender javaMailSender;

	@Bean
	@ConditionalOnProperty("spring.mail.host")
	public MailSender smtpMailSender() {
		SmtpMailSender mailSender = new SmtpMailSender();
		mailSender.setJavaMailSender(javaMailSender);
		return mailSender;
	}
}
```

See above how we have obtained a reference to *JavaMailSender* using *@Autowired*, and then injected that explicitly to the *smtpMailSender* bean.

Instead of using *@Autowired* to obtain a reference to the configured *JavaMailSender*, if we just have a *JavaMailSender* parameter in the bean method, as below, Spring will inject it as an argument. That means, the following code will also work, and in fact will look cleaner:

```
@Configuration
public class MailConfig {

	...

	@Bean
	@ConditionalOnProperty("spring.mail.host")
	public MailSender smtpMailSender(JavaMailSender javaMailSender) {
		SmtpMailSender mailSender = new SmtpMailSender();
		mailSender.setJavaMailSender(javaMailSender);
		return mailSender;
	}
}
```

Sending email

Let's now replace the snippet for sending email with the actual code. It will look as below:

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

import javax.mail.MessagingException;
import javax.mail.internet.MimeMessage;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;

public class SmtpMailSender implements MailSender {

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

	private JavaMailSender javaMailSender;

	public void setJavaMailSender(JavaMailSender javaMailSender) {
		this.javaMailSender = javaMailSender;
	}

	@Override
	public void send(String to, String subject, String body)
			throws MessagingException {

		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);
        }
}
```

We are first creating a MimeMessage and then a MimeMessageHelper. Using the helper, we are then setting the *subject*, *to,* etc. Finally, we are using the *javaMailSender* for sending the mail.

Because the above send method has now a *throws* declaration, that needs to be added to the MailSender interface, as well as the MailController:

```
public interface MailSender {

        void send(String to, String subject, String body) throws MessagingException;
}
```

```
@RestController
public class MailController {

        ...

        @RequestMapping("/mail")
        public String sendMail() throws MessagingException {

                mailSender.send("abc@example.com", "Some subject", "the content");

                return "Mail sent";
        }
}
```

You should now be able to send real, SMTP mails. Test it out!

When testing it, you will notice a lag when you visit http://localhost:8080/mail. That's because the response is being sent from our application only after the mail is sent. To avoid such lags, you should send mails asynchronously. How to do it is discussed in our Spring Framework 4 Tutorial: Practical, Rapid, Intuitive.

Source code so far

- Browse online
- Download zip
- See changes

11 Configuring Beans contd.

In this chapter, we'll explore some interesting behaviour of configuration classes. Let's first create a simple *DemoBean* class, say in the same *.mail* package:

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

public class DemoBean {

	public String foo() {
		return "something";
	}
}
```

Then, let's configure it in *MailConfig*:

```
@Configuration
public class MailConfig {

	...

	@Bean
	public DemoBean demoBean() {
		return new DemoBean();
	}

	...
}
```

So, when the application starts, a *demoBean* bean will be created.

Now, if you need to access that bean in another bean creation method, say inside the *smtpMailSender* method in *MailConfig*, what will you do?

You already know a couple of ways to do so. Just like injecting any other bean, you can use *@Autowired* to inject it as an instance variable. Or, you can have it as a parameter of the *smtpMailSender* method. Having it as a parameter will look like this:

```
@Configuration
public class MailConfig {

	...

	@Bean
	public DemoBean demoBean() {
```

```
            return new DemoBean();
      }

      @Bean
      @ConditionalOnProperty("spring.mail.host")
      public MailSender smtpMailSender(JavaMailSender javaMailSender,
                                       DemoBean demoBean) {
            demoBean.foo();
            SmtpMailSender mailSender = new SmtpMailSender();
            mailSender.setJavaMailSender(javaMailSender);
            return mailSender;
      }
}
```

Let's now see another way. You can just call the *demoBean()* method, as below!

```
@Configuration
public class MailConfig {

      ...

      @Bean
      public DemoBean demoBean() {
            return new DemoBean();
      }

      @Bean
      @ConditionalOnProperty("spring.mail.host")
      public MailSender smtpMailSender(JavaMailSender javaMailSender) {
            demoBean().foo();
            SmtpMailSender mailSender = new SmtpMailSender();
            mailSender.setJavaMailSender(javaMailSender);
            return mailSender;
      }
}
```

You may argue that it will cause the *demoBean()* method to run twice -- once when Spring will call it to create the bean, and a second time when we will call it inside the *smtpMailSender* method.

In contrast, bean creation methods run only once. Spring caches their return values, and return that in subsequent calls. So, *demoBean()* will execute only once. To check it out, let's create a default constructor in DemoBean and put a log line there.

```
public class DemoBean {

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

      public DemoBean () {
            log.info("DemoBean created!");
      }

      ...
}
```

Run the application now, and you'll see the log line only once.

@Bean in @Components

The @Bean annotation works not only in the classes annotated with @Configuration, but in any component class. For example, in MailConfig.java, replace @Configuration with @Component, and the application will still work. But, if you see the log, now you 'll notice that the *DemoBean* is being created twice. That's because Spring caches the return values of only those bean creation methods that are inside classes annotated with *@Configuration*.

Although @Bean methods can be placed in any class annotated with @Component or any of its specialization, @Configuration classes are the best place for those. So, avoid placing @Bean methods in classes other than those annotated with @Configuration.

Source code so far

- Browse online
- Download zip
- See changes

12 Next Steps

We have come to the end of this book. You should now have a solid understanding of Spring's dependency injection essentials.

Of course there is much more – Spring's DI is very rich. For example, we covered only singleton beans, but beans in application context can have narrower scopes as well, e.g. *Request* or *Session* scope.

If you now go ahead to learn those advanced DI features, you'll soon get bored and lose interest. We have already covered well enough to give you a solid foundation to start coding good Spring applications and explore Spring documentation as and when needed.

So, following our lean and rapid method, it's time to move forward to *Module II* of our Spring Framework for the Real World – Beginner to Expert course, venturing into other Spring components like Spring MVC, Spring Data, Spring Security etc.!

Click here to know more about the course.