



SpringBoot **Learn By Example**

A fast paced guide to learn SpringBoot

K. Siva Prasad Reddy



SpringBoot : Learn By Example

A fast paced guide to learn SpringBoot

K Siva Prasad Reddy

This book is for sale at <http://leanpub.com/springboot-learn-by-example>

This version was published on 2016-07-30



Leanpub

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.



This work is licensed under a [Creative Commons Attribution-NonCommercial 3.0 Unported License](#)

Tweet This Book!

Please help K Siva Prasad Reddy by spreading the word about this book on [Twitter!](#)

The suggested hashtag for this book is [#springboot_learn_by_example](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#springboot_learn_by_example

Contents

About This Book	i
Who this book is for	i
What this book covers	i
What you need to use this book	i
Source Code	ii
About Author	ii
Book Revision History	iii
Chapter 1: Introduction to SpringBoot	1
1.1. Overview of Spring framework	1
1.2. Developing Web Application using SpringMVC and JPA	4
1.3. A Quick Taste of SpringBoot	16
1.4. Summary	19
Chapter 2: Getting Started with SpringBoot	20
2.1. What is SpringBoot?	20
2.2. Our First SpringBoot Application	22
2.3. Summary	32
Chapter 3: SpringBoot Behind the scenes	33
3.1. Exploring the power of @Conditional	33
3.2. How SpringBoot AutoConfiguration Works?	40
3.3. Summary	44
Chapter 4: Creating Custom SpringBoot Starter	45
4.1. Create Parent Module spring-boot-starter-twitter4j	45
4.2. Create twitter4j-spring-boot-autoconfigure module	46
4.3. Create twitter4j-spring-boot-starter module	52
4.4. Create twitter4j-spring-boot-sample sample application	53
4.5. Summary	57
Appendix A : Resources	58

About This Book

The **SpringBoot : Lean by Example** book will help you to understand what is SpringBoot, how SpringBoot helps you to build Spring based applications quickly and easily and the inner workings of SpringBoot using easy to follow examples.

Who this book is for

This book is written for Java developers who are familiar with basics of Spring framework and want to learn SpringBoot. This book is suitable for both beginners and experienced developers as the book covers starting from the basics of SpringBoot to exploring how SpringBoot works behind the scenes.

What this book covers

The **SpringBoot : Learn By Example** book covers the following:

- What is SpringBoot?
- How SpringBoot increases developer productivity?
- How SpringBoot AutoConfiguration works behind the scenes?
- Working with Databases using JdbcTemplate, MyBatis, JOOQ, Spring Data JPA
- Working with MongoDB NoSQL database
- Developing web applications using SpringBoot and Thymeleaf
- Developing REST API using SpringBoot and Consuming from AngularJS Application
- Securing web applications using SpringBoot and SpringSecurity
- Monitoring SpringBoot applications with SpringBoot Actuator
- Testing SpringBoot applications

What you need to use this book

- [JDK 1.8](#)¹
- Your favourite IDE
 - [Spring Tool Suite](#)²

¹<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

²<https://spring.io/tools/sts>

- IntelliJ IDEA³
- NetBeans IDE⁴
- Build Tools
 - Maven⁵
 - Gradle⁶
- Database Server
 - MySQL⁷
 - PostgreSQL⁸

Source Code

You can find the source code for this book at GitHub <https://github.com/sivaprasadreddy/springboot-learn-by-example>⁹

About Author

K. Siva Prasad Reddy is a Software Developer living in Hyderabad, India with more than 10 years of experience in developing enterprise applications using Java and JavaEE technologies. Siva is a Sun Certified Java Programmer with deep expertise in building enterprise applications using server-side technologies such as **Java, JavaEE, Spring, Hibernate, MyBatis, JSF, PrimeFaces, and WebServices (SOAP/REST)**.

Siva is also the author of **Java Persistence with MyBatis 3** and **PrimeFaces Beginners Guide** Packt Publishing books. He has worked with several Fortune 500 organizations and is passionate about learning new technologies and their developments.

Siva shares the knowledge he has acquired on his blog at <http://sivalabs.in>. If you want to find out more about his work, you can follow him on Twitter [@sivalabs](https://twitter.com/sivalabs) and GitHub <https://github.com/sivaprasadreddy>.

³<https://www.jetbrains.com/idea/>

⁴<https://netbeans.org/>

⁵<https://maven.apache.org/>

⁶<http://gradle.org/>

⁷<https://www.mysql.com/downloads/>

⁸<http://www.postgresql.org/download/>

⁹<https://github.com/sivaprasadreddy/springboot-learn-by-example>

Book Revision History

- 08-July-2016 : Published First version.
- 16-July-2016 : Added following sections.
 - Added Section **9.3. Working with Multiple Databases**
 - Added Sub-section **Exposing JPA entities with bi-directional references through RESTful services** in Section **12.2. REST API using SpringMVC**
- 30-July-2016 : Added new chapter. Chapter 16: Deploying SpringBoot Applications.
 - Running SpringBoot applications in production mode
 - Deploying SpringBoot application on Heroku
 - Running SpringBoot application on Docker

Chapter 1: Introduction to SpringBoot

Spring is a very popular Java framework for building web and enterprise applications. Unlike many other frameworks which focuses on only one area(web or persistence etc), Spring framework provides a wide verity of features addressing the modern business needs via its portfolio projects.

Spring framework provides flexibility to configure beans in multiple ways such as XML, Annotations and JavaConfig. With the number of features increased the complexity also gets increased and configuring Spring applications becomes tedious and error-prone. Spring team created SpringBoot to address the complexity of configuration.

But before diving into SpringBoot, we will take a quick look at Spring framework and see what kind of problems SpringBoot is trying to address.

In this chapter we will cover:

- **Overview of Spring framework**
- **Developing Web Application using SpringMVC and JPA**
- **A Quick Taste of SpringBoot**

1.1. Overview of Spring framework

If you are a Java developer then there is a high chance that you might have heard about Spring framework and probably have used it in your projects. Spring framework was created primarily as a Dependency Injection container but it is much more than that.

Spring is very popular because of several reasons:

- Spring's dependency injection approach encourages writing testable code
- Easy to use and powerful database transaction management capabilities
- Spring simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts/JSF etc web frameworks
- State of the art Web MVC framework for building web applications

Along with Spring framework there are many other Spring portfolio projects which helps to build applications addressing modern business needs:

- **Spring Data:** Simplifies data access from relational and NoSQL data stores.
- **Spring Batch:** Provides powerful batch processing framework.

- **Spring Security:** Robust security framework to secure applications.
- **Spring Social:** Supports integration with social networking sites like Facebook, Twitter, LinkedIn, GitHub etc.
- **Spring Integration:** An implementation of Enterprise Integration Patterns to facilitate integration with other enterprise applications using lightweight messaging and declarative adapters.

There are many other interesting projects addressing various other modern application development needs. For more information take a look at <http://spring.io/projects>.

In the initial days, Spring provides XML based approach for configuring beans. Later Spring introduced XML based DSLs, Annotations and JavaConfig based approaches for configuring beans.

Let us take a quick look at how each of those configuration styles looks like.

XML based configuration

```
<bean id="userService" class="com.sivalabs.myapp.service.UserService">
  <property name="userDao" ref="userDao"/>
</bean>
```

```
<bean id="userDao" class="com.sivalabs.myapp.dao.JdbcUserDao">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/test"/>
  <property name="username" value="root"/>
  <property name="password" value="secret"/>
</bean>
```

Annotation based configuration

```
@Service
public class UserService
{
    private UserDao userDao;

    @Autowired
    public UserService(UserDao dao){
        this.userDao = dao;
    }
    ...
    ...
}
```

```
@Repository
public class JdbcUserDao
{
    private DataSource dataSource;

    @Autowired
    public JdbcUserDao(DataSource dataSource){
        this.dataSource = dataSource;
    }
    ...
    ...
}
```

JavaConfig based configuration

```
@Configuration
public class AppConfig
{
    @Bean
    public UserService userService(UserDao dao){
        return new UserService(dao);
    }

    @Bean
    public UserDao userDao(DataSource dataSource){
        return new JdbcUserDao(dataSource);
    }

    @Bean
```

```
public DataSource dataSource(){
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName("com.mysql.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/test");
    dataSource.setUsername("root");
    dataSource.setPassword("secret");
    return dataSource;
}
}
```

Spring provides many approaches for doing the same thing and we can even mix the approaches as well like you can use both JavaConfig and Annotation based configuration styles in the same application. That is a lot of flexibility and it is one way good and one way bad. **People new to Spring framework may gets confused about which approach to follow.**



As of now the Spring team is suggesting to follow JavaConfig based approach as it gives more flexibility. But there is no one-size fits all kind of solution. One has to choose the approach based on their own application needs.

OK, now that you had a glimpse of how various styles of Spring bean configurations looks like. Let us take a quick look at the configuration of a typical SpringMVC + JPA/Hibernate based web application configuration looks like.



Download Book Sample Code from GitHub

You will be seeing lots of code samples through out the book. To keep focus on the key parts of the code I may omit some snippets such as CSS styles, import statements, optional jar dependency configurations.

I would strongly recommend you to download the source code for quick reference. You can find the source code for this book at GitHub <https://github.com/sivaprasadreddy/springboot-learn-by-example>¹⁰

1.2. Developing Web Application using SpringMVC and JPA

Before getting to know what is SpringBoot and what kind of features it provides, let us take a look at how a typical Spring web application configuration looks like, what are the pain points and then we will discuss about how SpringBoot addresses those problems.

¹⁰<https://github.com/sivaprasadreddy/springboot-learn-by-example>

Step 1: Configure Maven Dependencies

First thing we need to do is configure all the dependencies required in our *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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sivalabs</groupId>
  <artifactId>springmvc-jpa-demo</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>springmvc-jpa-demo</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <failOnMissingWebXml>>false</failOnMissingWebXml>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
      <version>4.2.4.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-jpa</artifactId>
      <version>1.9.2.RELEASE</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>jcl-over-slf4j</artifactId>
      <version>1.7.13</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.13</version>
    </dependency>
  </dependencies>
</project>
```

```
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.13</version>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>1.4.190</version>
</dependency>
<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>1.4</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.38</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>4.3.11.Final</version>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf-spring4</artifactId>
  <version>2.1.4.RELEASE</version>
</dependency>
```

```
</dependencies>
</project>
```

We have configured all our Maven jar dependencies to include Spring MVC, Spring Data JPA, JPA/Hibernate, Thymeleaf and Log4j.

Step 2: Configure Service/DAO layer beans using JavaConfig.

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages="com.sivalabs.demo")
@PropertySource(value = { "classpath:application.properties" })
public class AppConfig {

    @Autowired
    private Environment env;

    @Bean
    public static PropertySourcesPlaceholderConfigurer placeholderConfigurer()
    {
        return new PropertySourcesPlaceholderConfigurer();
    }

    @Value("${init-db:false}")
    private String initDatabase;

    @Bean
    public PlatformTransactionManager transactionManager()
    {
        EntityManagerFactory factory = entityManagerFactory().getObject();
        return new JpaTransactionManager(factory);
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory()
    {
        LocalContainerEntityManagerFactoryBean factory =
            new LocalContainerEntityManagerFactoryBean();
        HibernateJpaVendorAdapter vendorAdapter =
            new HibernateJpaVendorAdapter();
        vendorAdapter.setGenerateDdl(Boolean.TRUE);
        vendorAdapter.setShowSql(Boolean.TRUE);
    }
}
```

```
factory.setDataSource(dataSource());
factory.setJpaVendorAdapter(vendorAdapter);
factory.setPackagesToScan(env.getProperty("packages-to-scan"));

Properties jpaProperties = new Properties();
jpaProperties.put("hibernate.hbm2ddl.auto",
    env.getProperty("hibernate.hbm2ddl.auto"));
factory.setJpaProperties(jpaProperties);

factory.afterPropertiesSet();
factory.setLoadTimeWeaver(new InstrumentationLoadTimeWeaver());
return factory;
}

@Bean
public HibernateExceptionHandler hibernateExceptionHandler()
{
    return new HibernateExceptionHandler();
}

@Bean
public DataSource dataSource()
{
    BasicDataSource dataSource = new BasicDataSource();
    dataSource.setDriverClassName(env.getProperty("jdbc.driverClassName"));
    dataSource.setUrl(env.getProperty("jdbc.url"));
    dataSource.setUsername(env.getProperty("jdbc.username"));
    dataSource.setPassword(env.getProperty("jdbc.password"));
    return dataSource;
}

@Bean
public DataSourceInitializer dataSourceInitializer(DataSource dataSource)
{
    DataSourceInitializer dsInitializer = new DataSourceInitializer();
    dsInitializer.setDataSource(dataSource);
    ResourceDatabasePopulator dbPopulator = new ResourceDatabasePopulator();
    dbPopulator.addScript(new ClassPathResource("data.sql"));
    dsInitializer.setDatabasePopulator(dbPopulator);
    dsInitializer.setEnabled(Boolean.parseBoolean(initDatabase));
    return dsInitializer;
}
```

```
}
```

In our `AppConfig.java` configuration class we have done the following:

- Marked it as a Spring Configuration class using `@Configuration` annotation.
- Enabled Annotation based transaction management using `@EnableTransactionManagement`
- Configured `@EnableJpaRepositories` to indicate where to look for Spring Data JPA repositories
- Configured PropertyPlaceholder bean using `@PropertySource` annotation and `PropertySourcesPlaceholderConfigurer` bean definition which loads properties from `application.properties` file.
- Defined beans for `DataSource`, `JPA EntityManagerFactory`, `JpaTransactionManager`.
- Configured `DataSourceInitializer` bean to initialize the database by executing `data.sql` script on application start up.

we can configure property placeholder values in `application.properties` as follows:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/test
jdbc.username=root
jdbc.password=admin
init-db=true
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.show_sql=true
hibernate.hbm2ddl.auto=update
```

Create a simple sql script `data.sql` to populate sample data into `USER` table.

```
delete from user;

insert into user(id, name) values(1, 'Siva');
insert into user(id, name) values(2, 'Prasad');
insert into user(id, name) values(3, 'Reddy');
```

Create `log4j.properties` file with basic configuration as follows:


```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p %t %c{2}:%L - %m%n

log4j.category.com.sivalabs=DEBUG
log4j.category.org.springframework=INFO
```

Step 3: Configure Spring MVC web layer beans

We will have to configure Thymeleaf ViewResolver, static ResourceHandlers, MessageSource for i18n etc.

```
@Configuration
@ComponentScan(basePackages = { "com.sivalabs.demo" })
@EnableWebMvc
public class WebMvcConfig extends WebMvcConfigurerAdapter
{
    @Bean
    public TemplateResolver templateResolver() {
        TemplateResolver templateResolver = new ServletContextTemplateResolver();
        templateResolver.setPrefix("/WEB-INF/views/");
        templateResolver.setSuffix(".html");
        templateResolver.setTemplateMode("HTML5");
        templateResolver.setCacheable(false);
        return templateResolver;
    }

    @Bean
    public SpringTemplateEngine templateEngine() {
        SpringTemplateEngine templateEngine = new SpringTemplateEngine();
        templateEngine.setTemplateResolver(templateResolver());
        return templateEngine;
    }

    @Bean
    public ThymeleafViewResolver viewResolver() {
        ThymeleafViewResolver thymeleafViewResolver =
            new ThymeleafViewResolver();
        thymeleafViewResolver.setTemplateEngine(templateEngine());
        thymeleafViewResolver.setCharacterEncoding("UTF-8");
        return thymeleafViewResolver;
    }
}
```

```

    }

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry)
    {
        registry.addResourceHandler("/resources/**")
            .addResourceLocations("/resources/");
    }

    @Bean(name = "messageSource")
    public MessageSource configureMessageSource()
    {
        ReloadableResourceBundleMessageSource messageSource =
            new ReloadableResourceBundleMessageSource();
        messageSource.setBasename("classpath:messages");
        messageSource.setCacheSeconds(5);
        messageSource.setDefaultEncoding("UTF-8");
        return messageSource;
    }
}

```

In our `WebMvcConfig.java` configuration class we have done the following:

- Marked it as a Spring Configuration class using `@Configuration` annotation.
- Enabled Annotation based Spring MVC configuration using `@EnableWebMvc`
- Configured Thymeleaf ViewResolver by registering `TemplateResolver`, `SpringTemplateEngine`, `ThymeleafViewResolver` beans.
- Registered `ResourceHandlers` bean to indicate requests for static resources with URI `/resources/**` will be served from the location `/resources/` directory.
- Configured `MessageSource` bean to load i18n messages from ResourceBundle `messages_{country-code}.properties` from classpath.

Create `messages.properties` file in `src/main/resources` folder and add the following property.

`app.title=SpringMVC JPA Demo (Without SpringBoot)`

Step 4: Register Spring MVC FrontController servlet DispatcherServlet.

Prior to Servlet 3.x specification we have to register Servlets/Filters in `web.xml`. Since Servlet 3.x specification we can register Servlets/Filters programmatically using `ServletContainerInitializer`. Spring MVC provides a convenient class `AbstractAnnotationConfigDispatcherServletInitializer` to register `DispatcherServlet`.

```
public class SpringWebAppInitializer
    extends AbstractAnnotationConfigDispatcherServletInitializer
{
    @Override
    protected Class<?>[] getRootConfigClasses()
    {
        return new Class<?>[] { AppConfig.class};
    }

    @Override
    protected Class<?>[] getServletConfigClasses()
    {
        return new Class<?>[] { WebMvcConfig.class };
    }

    @Override
    protected String[] getServletMappings()
    {
        return new String[] { "/" };
    }

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[]{ new OpenEntityManagerInViewFilter() };
    }
}
```

In our `SpringWebAppInitializer.java` configuration class we have done the following:

- We have configured `AppConfig.class` as `RootConfirationClasses` which will become the parent `ApplicationContext` that contains bean definitions shared by all child (`DispatcherServlet`) contexts.
- We have configured `WebMvcConfig.class` as `ServletConfigClasses` which is child `ApplicationContext` that contains `WebMvc` bean definitions.
- We have configured `"/` as `ServletMapping` means all the requests will be handled by `DispatcherServlet`.
- We have registered `OpenEntityManagerInViewFilter` as a `Servlet Filter` so that we can lazy load the `JPA Entity` lazy collections while rendering the view.

Step 5: Create a JPA Entity and Spring Data JPA Repository

Create a JPA entity `User.java` and a Spring Data JPA repository for `User` entity.

```
@Entity
public class User
{
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;
    private String name;

    public User()
    {
    }

    public User(Integer id, String name)
    {
        this.id = id;
        this.name = name;
    }

    //setters and getters
}

public interface UserRepository extends JpaRepository<User, Integer>
{
}
}
```

If you don't understand what is **JpaRepository** don't worry. We will learn more about **Spring Data JPA** in **Chapter 9: Working With JPA** in detail.

Step 6: Create a SpringMVC Controller

Create a SpringMVC controller to handle URL "/" and render list of users.

```
@Controller
public class HomeController
{
    @Autowired UserRepository userRepo;

    @RequestMapping("/")
    public String home(Model model)
    {
        model.addAttribute("users", userRepo.findAll());
        return "index";
    }
}
```

Step 7: Create a thymeleaf view /WEB-INF/views/index.html to render list of Users.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8"/>
<title>Home</title>
</head>
<body>
  <h2 th:text="#{app.title}">App Title</h2>
  <table>
    <thead>
      <tr>
        <th>Id</th>
        <th>Name</th>
      </tr>
    </thead>
    <tbody>
      <tr th:each="user : ${users}">
        <td th:text="${user.id}">Id</td>
        <td th:text="${user.name}">Name</td>
      </tr>
    </tbody>
  </table>
</body>
</html>
```

We are all set now to run the application. But before that we need to download and configure the server like Tomcat or Jetty or Wildfly etc in your IDE. You can download Tomcat 8 and configure in your favourite IDE, run the application and point your browser to <http://localhost:8080/springmvc-jpa-demo>. You should see the list of users details in a table.



Users List

Yay...We did it. But wait..Isn't it too much work to just show a list of user details pulled from a database table?

Let us be honest and fair. All this configuration is not just for this one use-case. This configuration is basis for rest of the application also. But again, this is too much of work to do if you want to quickly get up and running. Another problem with it is, assume you want to develop another SpringMVC application with similar technical stack? Well, you copy-paste the configuration and tweak it. Right?

But remember one thing: if you have to do the same thing again and again, you should find an automated way to do it.

Apart from writing the same configuration again and again, do you see any other problems here?

Well, let me list our what are the problems I am seeing here.

1. You need to hunt for all the compatible libraries for the specific Spring version and configure them.
2. 95% of the times we configure the DataSource, EntityManagerFactory, TransactionManager etc beans in the same way. Wouldn't it be great if Spring can do it for me automatically.
3. Similarly we configure SpringMVC beans like ViewResolver, MessageSource etc in the same way most of the times. If Spring can automatically do it for me that would be awesome.

Imagine, what if Spring is capable of configuring beans automatically? What if you can customize the automatic configuration using simple customizable properties? For example, instead of mapping DispatcherServlet url-pattern to "/" you want to map it to "/app/". Instead of putting thymeleaf views in "/WEB-INF/views" folder you may want to place them in "/WEB-INF/templates/" folder.

So basically you want Spring to do things automatically but provide the flexibility to override the default configuration in a simpler way?

Well, you are about to enter into the world of SpringBoot where your dreams come true!!!

1.3. A Quick Taste of SpringBoot

Welcome to SpringBoot!. SpringBoot do what exactly you are looking for. It will do things automatically for you but allows you to override the defaults if you want to.

Instead of explaining in theory I prefer to explain by example. So let us implement the same application that we built earlier but this time using SpringBoot.

Step 1: Create a Maven based SpringBoot Project

Create a Maven project and configure the dependencies as follows:

```
<?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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sivalabs</groupId>
  <artifactId>hello-springboot</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello-springboot</name>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.6.RELEASE</version>
  </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-data-jpa</artifactId>
    </dependency>
    <dependency>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
</dependencies>
</project>
```

Wow our **pom.xml** suddenly become so small!!.

Don't worry if any of this configuration doesn't make sense at this point of time. We have plenty of explanation on all of this in coming chapters.

Step 2: Configure datasource/JPA properties in src/main/resources/application.properties as follows.

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
spring.datasource.password=admin
spring.datasource.initialize=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

you can copy the same **data.sql** file into **src/main/resources** folder.

Step 3: Create a JPA Entity and Spring Data JPA Repository Interface for the entity.

Create **User.java**, **UserRepository.java** and **HomeController.java** same as in **springmvc-jpa-demo** application.

Step 4: Create Thymeleaf view to show list of users

Copy **/WEB-INF/views/index.html** that we created in **springmvc-jpa-demo** application into **src/main/resources/templates** folder in our new project.

Step 5: Create SpringBoot EntryPoint Class.

Create a Java class **Application.java** with main method as follows:


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

Now run **Application.java** as a Java Application and point your browser to <http://localhost:8080/>. You should see the list of users in table format.

Ok ok, I hear you are shouting “What is going on???”.

Let me explain what just happened

1. Easy dependency Management

- First thing to observe is we are using some dependencies named like **spring-boot-starter-***.
- Remember I said “95% of the times I use same configuration”. So when you add **spring-boot-starter-web** dependency by default it will pull all the commonly used libraries while developing Spring MVC applications such as **spring-webmvc**, **jackson-json**, **validation-api** and **tomcat**.
- We have added **spring-boot-starter-data-jpa** dependency. This pulls all the **spring-data-jpa** dependencies and also adds Hibernate libraries because majority of the applications use Hibernate as JPA implementation.

2. Auto Configuration

- Not only the **spring-boot-starter-web** adds all these libraries but also configures the commonly registered beans like **DispatcherServlet**, **ResourceHandlers**, **MessageSource** etc beans with sensible defaults.
- We also added **spring-boot-starter-thymeleaf** which not only adds the thymeleaf library dependencies but also configures **ThymeleafViewResolver** beans as well automatically.
- We haven’t defined any of the **DataSource**, **EntityManagerFactory**, **TransactionManager** etc beans but they are automatically gets created. How?

If we have any in-memory database drivers like H2 or HSQL in our classpath then SpringBoot will automatically creates an in-memory DataSource and then registers EntityManagerFactory, TransactionManager beans automatically with sensible defaults.

But we are using MySQL, so we need to explicitly provide MySQL connection details. We have configured those MySQL connection details in application.properties file and SpringBoot creates a DataSource using these properties.

3. Embedded Servlet Container Support

- The most important and surprising thing is we have created a simple Java class annotated with some magical annotation `@SpringApplication` having a main method and by running that main we are able to run the application and access it at `http://localhost:8080/`.
- Where is the servlet container comes from?

We have added `spring-boot-starter-web` which pull the `spring-boot-starter-tomcat` automatically and when we run the `main()` method it started tomcat as an embedded container so that we don't have to deploy our application on any externally installed tomcat server.

By the way have you observe that our packaging type in `pom.xml` is 'jar' not 'war'. Wonderful! Ok, but what if I want to use Jetty server instead of tomcat? Simple, exclude `spring-boot-starter-tomcat` from `spring-boot-starter-web` and include `spring-boot-starter-jetty`. That's it.

But, this looks all magical!!!

I can imagine what you are thinking. You are thinking like **SpringBoot looks cool and it is doing lot of things automatically for me. But still I am not fully understand how it is all really working behind the scenes.** Right?

I can understand. Watching magic show is fun normally, but not in Software Development. Don't worry, we will be looking at each of those things and explain in-detail how things are happening behind the scenes. But I don't want to overwhelm you by dumping everything on to your hear right now in this chapter.

1.4. Summary

In this chapter we had a quick overview of various Spring configuration styles and understand the complexity of configuring Spring applications. Also, we had a quick look at SpringBoot by creating a simple web application.

In next chapter we will take a more detailed look at SpringBoot and how we can create SpringBoot applications in different ways.

Chapter 2: Getting Started with SpringBoot

In the previous chapter we had a quick look at how we can develop a typical SpringMVC + JPA(Hibernate) based web application using plain SpringMVC and Spring ORM modules. We have discussed about the complexity of configuring the Spring based applications. And then we took a quick look at how to develop the same web application using SpringBoot in a much easier way. But we didn't go in-detail about SpringBoot.

In this chapter we are going to take a more detailed look into SpringBoot and what are the features SpringBoot offers.

In this chapter we will cover:

- **What is SpringBoot?**
- **Creating SpringBoot application**
 - IDE and Build Tools support
 - Fat jar using SpringBoot Maven Plugin

2.1. What is SpringBoot?

SpringBoot is an opinionated framework which helps to build Spring based applications quickly and easily. The main goal of SpringBoot is to quickly create Spring based applications without requiring the developers to write the same boilerplate configuration again and again.

The key SpringBoot features include:

- SpringBoot starters
- SpringBoot AutoConfiguration
- Elegant Configuration Management
- SpringBoot Actuator
- Easy to use Embedded Servlet container support

and many more.

SpringBoot starters

SpringBoot offers many starter modules to get started quickly with many of the commonly used technologies like **SpringMVC**, **JPA**, **MongoDB**, **Spring Batch**, **SpringSecurity**, **Solr**, **ElasticSearch** etc. These starters are pre-configured with the most commonly used library dependencies so that we don't have to search for the compatible library versions and configure them manually.

For ex: **spring-boot-starter-data-jpa** starter module includes all the dependencies required to use **Spring Data JPA** along with **Hibernate** library dependencies as **Hibernate** is the most commonly used **JPA** implementation.

You can find list of all the SpringBoot starters that comes out-of-the-box in official documentation at [Starter POMs¹¹](#)

SpringBoot AutoConfiguration

SpringBoot addresses the “**Spring applications needs complex configuration**” problem by eliminating the need to manually configuring the boilerplate configuration by using **AutoConfiguration**. SpringBoot takes opinionated view of the application and configures various components automatically by registering beans based on various criteria.

The criteria can be:

- Availability of a particular class in classpath
- Presence or Absence of a Spring Bean
- Presence of a System Property
- Absence of configuration file

etc.

For example:

If you have **spring-webmvc** dependency in your classpath SpringBoot assumes you are trying to build a SpringMVC based web application and automatically tries to register **DispatcherServlet** if it is not already registered by you.

If you have any Embedded database driver in classpath like **H2** or **HSQL** and if you haven't configured any **DataSource** bean explicitly then SpringBoot will automatically register a **DataSource** bean using in-memory database settings.

We will learn more about the **AutoConfiguration** in further chapters in detail.

¹¹<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-starter-poms>

Elegant Configuration Management

Spring supports externalizing configurable properties using `@PropertySource` configuration. Spring-Boot takes it even further by using the sensible defaults and powerful type-safe property binding to bean properties. SpringBoot supports having separate configuration files for different profiles without requiring much configuration.

SpringBoot Actuator

Being able to get the various details of an application running in production is very crucial to many applications. SpringBoot Actuator provides a wide variety of such production ready features without requiring to write much code.

Some of the Spring Actuator features are:

- Can view the application bean configuration details
- Can view the application URL Mappings, Environment details and configuration parameter values
- Can view the registered Health Check Metrics

and many more.

Easy to use Embedded Servlet container support

Traditionally while building web applications we used to create `war` type modules and then deploy on external servers like `tomcat`, `wildfly` etc. But by using SpringBoot we can create `jar` type module and embed the servlet container within application itself very easily so that our application will be a self-contained deployment unit. Also, during development we can easily run the SpringBoot `jar` type module as a Java application from IDE or from command-line using build tool like `Maven` or `Gradle`.

We will learn more about these features and how to use them effectively in next chapters.

2.2. Our First SpringBoot Application

There are many ways to create a SpringBoot application. The simplest way is to use Spring Initializer <http://start.spring.io/> which is an online SpringBoot application generator.

Using Spring Initializer

You can point your browser to <http://start.spring.io/> and enter the project details as follows:

1. Click on **Switch to the full version link**
2. Select Maven Project and SpringBoot version (As of writing this book latest is 1.3.2)
3. Enter the maven project details as follows:
Group: com.sivalabs
Artifact : springboot-basic
Name: springboot-basic
Package Name: com.sivalabs.demo
Packaging: Jar
Java version: 1.8
Language: Java
4. Either you can search for the starters if you are already familiar with the names or click on **Switch to the full version link**. to see all the available starter.
You can see many starter modules organized into various categories like **Core, Web, Data** etc.
Select **Web** checkbox from **Web** category
Click on **Generate Project** button.

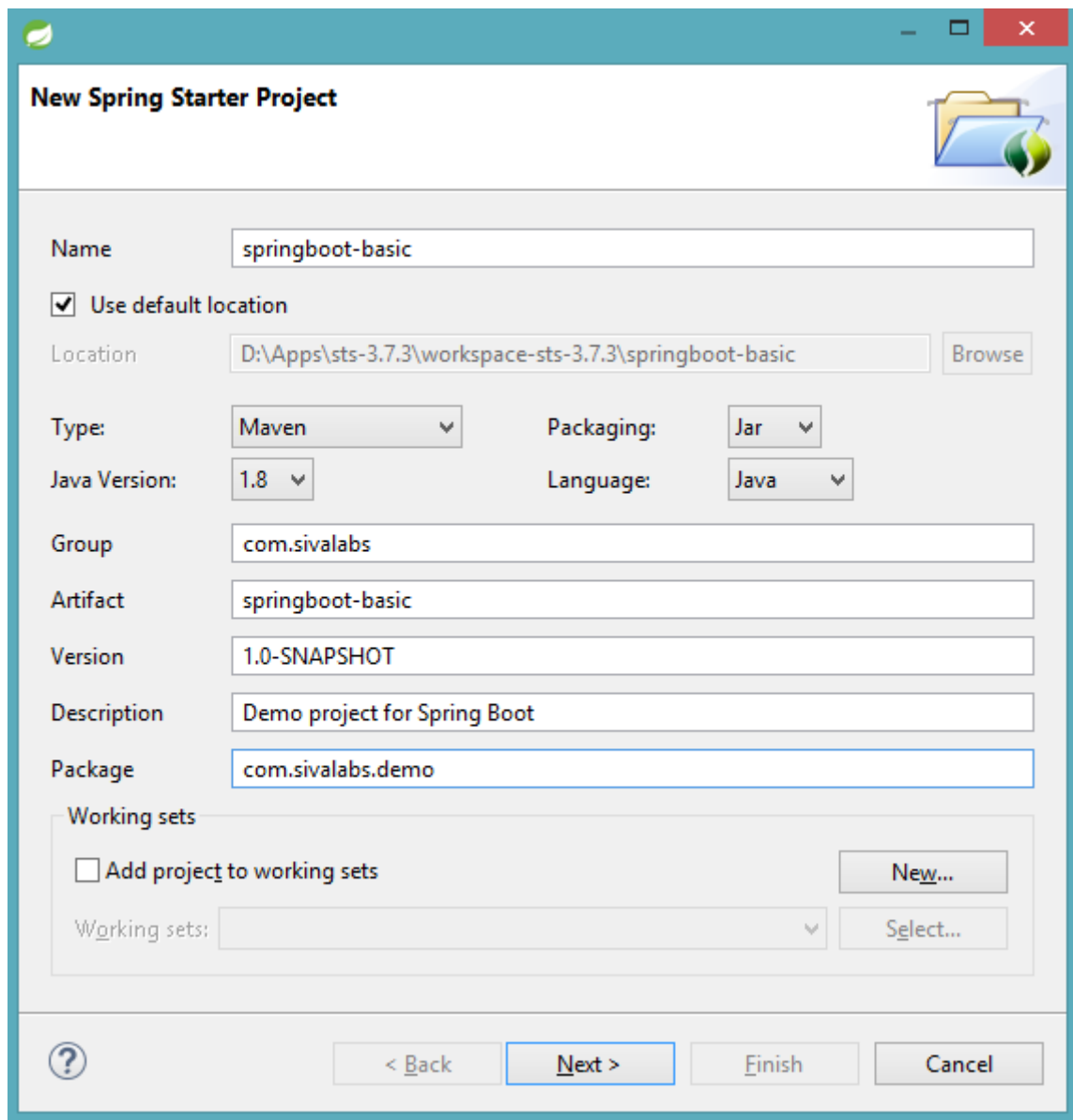
Now you can extract the downloaded zip file and import into your favorite IDE.

Using Spring Tool Suite

Spring Tool Suite (STS <https://spring.io/tools/sts>) is an extension of Eclipse IDE with lot of Spring framework related plugins.

You can easily create a SpringBoot application from STS as follows:

- **Select File -> New -> Other -> Spring -> Spring Starter Project -> Next**
- You will see the wizard which looks similar to Spring Initializer.



New Spring Starter Project

Name:

Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

Description:

Package:

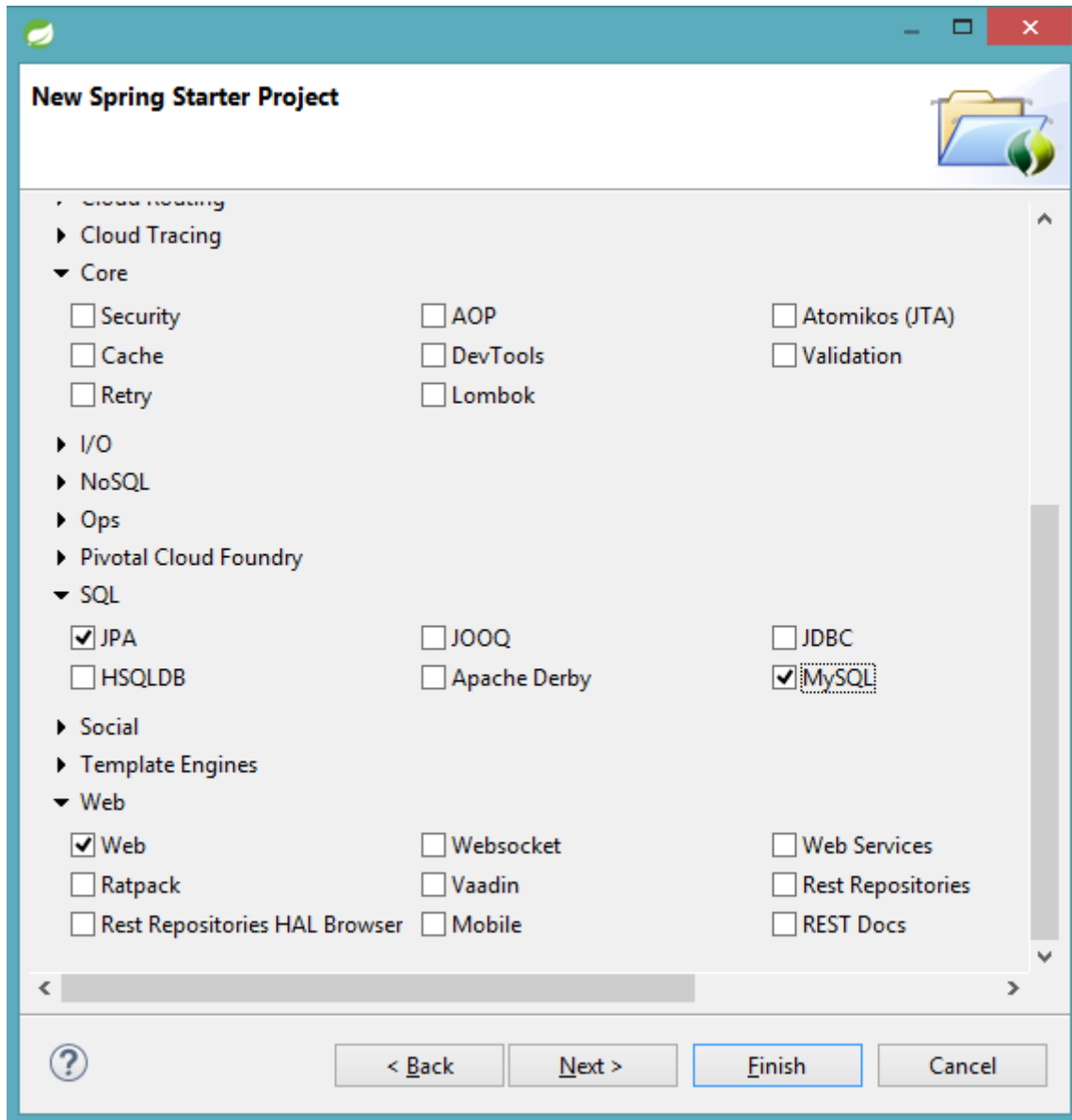
Working sets

Add project to working sets

Working sets:

Spring Starter Project

Enter the project details, click Next and select the starters and click Finish.



Spring Boot Starters

The SpringBoot project will be created and automatically imported into STS IDE as well.

Using IntelliJ IDEA

IntelliJ IDEA is a powerful commercial IDE with great features including support for SpringBoot. You can create a SpringBoot project from IntelliJ IDEA as follows:

- Select File -> New -> Project -> Spring Initializer -> Next

- Enter the project details, click Next and select the starters, click Next and enter Project Name and click Finish.



Spring support comes out-of-the-box only in commercial IntelliJ IDEA Ultimate Edition, but not in Community Edition which is free. But if you want to use IntelliJ IDEA Community Edition then you can generate the project using Spring Initializer and import into IntelliJ IDEA as Maven/Gradle project.

Using NetBeans IDE

NetBeans IDE is another popular IDE for developing Java applications. As of now there is no out-of-the-box support for creating SpringBoot project in NetBeans IDE but the community built [NB SpringBoot Plugin](#)¹² which supports creating SpringBoot applications directly from IDE itself.



There are some other options to quickly start using SpringBoot by using Spring Boot CLI and SDKMAN. You can find more details at [Installing Spring Boot](#)¹³.

Now you have created a SpringBoot Maven based project with web starter.

Let us explore what is contained in the generated application.

Step 1: First, let us take a look at pom.xml file.

```
<?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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sivalabs</groupId>
  <artifactId>springboot-basic</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>springboot-basic</name>

  <parent>
    <groupId>org.springframework.boot</groupId>
```

¹²<https://github.com/AlexFalappa/nb-springboot>

¹³<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-installing-spring-boot>

```
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.6.RELEASE</version>
</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>
```

1. First thing to note here is our **springboot-basic** maven module is inheriting from **spring-boot-starter-parent** module.

By inheriting from **spring-boot-starter-parent** module we will automatically get the following benefits:

- We only need to specify the springboot version once in parent module configuration. We don't need to specify the version for all the starter dependencies and other supporting libraries.

To see all the list of supporting libraries see the **pom.xml** of **org.springframework.boot:spring-boot-dependencies:1.3.6.RELEASE** maven module.

- The parent module **spring-boot-starter-parent** already included the most commonly used

plugin such as **maven-jar-plugin**, **maven-surefire-plugin**, **maven-war-plugin**, **exec-maven-plugin**, **maven-resources-plugin** with sensible defaults.

- In addition to the above mentioned plugins, module **spring-boot-starter-parent** module also configured the **spring-boot-maven-plugin** which will be used to build **fat jar**.

We will look into **spring-boot-maven-plugin** in more details in further sections in this chapter.

2. We have selected only **web** starter but **test** starter also be included by default.
3. We have selected **1.8** as the Java Version, hence the property `<java.version>1.8</java.version>` is included. This `java.version` value will be used to configure the JDK version for maven compiler in **spring-boot-starter-parent** module.

```
<maven.compiler.source>${java.version}</maven.compiler.source>
```

```
<maven.compiler.target>${java.version}</maven.compiler.target>
```

4. Application EntryPoint class **SpringbootBasicApplication.java**

A SpringBoot jar type module will have an application entry point Java class with **public static void main(String[] args)** method which you can run to start the application.

```
package com.sivalabs.demo;

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

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

Here **SpringbootBasicApplication** class is annotated with **@SpringBootApplication** annotation which is a composed annotation.

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {
    ....
    ....
}

```

Let me explain the meaning of these annotations.

- **@Configuration** annotation indicates that this class is a Spring configuration class.
- **@ComponentScan** annotation enables component scanning for Spring beans in the package in which the current class is defined
- **@EnableAutoConfiguration** annotation is the one which triggers all the SpringBoot's auto configuration mechanism.

We are bootstrapping the application by calling **SpringApplication.run(SpringbootBasicApplication.class, args)** in our **main()** method. We can pass one or more Spring Configuration classes to **SpringApplication.run()** method. But if we have our application entry-point class in **root package** then it is sufficient to pass our application entry class only which takes care of scanning other Spring Configuration classes with in all the sub-packages.

Step 2: Now create a simple SpringMVC controller, HomeController, as follows:

```

package com.sivalabs.demo;

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

@Controller
public class HomeController
{
    @RequestMapping("/")
    public String home(Model model) {
        return "index.html";
    }
}

```

This is a simple SpringMVC controller with one request handler method for URL `"/` which returns the view name `"index.html"`.

Step 3: Create a HTML view `index.html`

By default SpringBoot serves the static content from `src/main/public/` and `src/main/static/` directories. So create `index.html` in `src/main/public/` as follows:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8"/>
<title>Home</title>
</head>
<body>
    <h2>Hello World!!</h2>
</body>
</html>
```

Now from your IDE run the `SpringbootBasicApplication.java` `main()` method as stand-alone Java class which will start the embedded tomcat server on port **8080** and point your browser to `http://localhost:8080/`.

You should be able to see the response **Hello World!!**.

You can also run the SpringBoot application using `spring-boot-maven-plugin` goal as follows:

```
mvn spring-boot:run
```

Fat jar using SpringBoot Maven Plugin

We can run our application directly from our IDE or using Maven goal `spring-boot:run` but ultimately we need to create a deployment unit which can be run in our production environment without any IDE support.

We can use `spring-boot-maven-plugin` to create a single deployment unit (fat jar) by executing the following Maven goals.

```
mvn clean package
```

Now you can see two interesting files in target directory, `springboot-basic-1.0-SNAPSHOT.jar` and `springboot-basic-1.0-SNAPSHOT.jar.original`.

The `springboot-basic-1.0-SNAPSHOT.jar.original` will contain only the compiled classes and classpath resources.

But if you look into `springboot-basic-1.0-SNAPSHOT.jar` you can find the following:

- Compiled classes of our own source code in **src/main/java**
- Static resources from **src/main/resources**
- All the dependent jars in **lib** directory
- Classes in **org.springframework.boot.loader** package which does SpringBoot magic of running SpringBoot application.



We can create self contained deployment unit for jar type modules using plugins like **maven-shade-plugin** which packages all the dependent jars classes into single jar file.

But Spring Boot follows a different approach which allows to nest jars directly with in our SpringBoot application jar file. You can read more about it here [The executable jar format¹⁴](#)

Now you can run the application using the following command:

```
java -jar springboot-basic-1.0-SNAPSHOT.jar
```

SpringBoot using Gradle

Gradle is another popular build tool based on Groovy DSL. You can use Gradle instead of Maven to build SpringBoot applications. Gradle also follows the similar project structure as Maven like main java source resides in **src/main/java**, main resources resides in **src/main/resources** etc.

You can create a gradle based SpringBoot project with **build.gradle** as follows:

```
buildscript {
    repositories {
        jcenter()
    }

    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.3.6.RELEASE")
    }
}

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

repositories {
    jcenter()
}
```

¹⁴<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar>

```
}  
  
dependencies {  
    compile("org.springframework.boot:spring-boot-starter-web")  
    testCompile("org.springframework.boot:spring-boot-starter-test")  
}
```

Now you can run the application using **gradle bootRun** or you can use **gradle build** command which generates the fat jar in **build/libs** directory.



Maven or Gradle?

In Java world **Maven** and **Gradle** are two most popular build tools. Maven was released in 2004 and is being used widely by lot of developers. Gradle was released in 2012 which is more powerful and easy to customize for the modern projects complex build process needs.

As Maven is still the most commonly used build tool **we will be using Maven throughout the book**. However you can find the Gradle build scripts in the [book sample code](#)¹⁵ for each of the modules.

So, you can use either **Maven** or **Gradle**. Choice is yours!!

2.3. Summary

Now that you know how to create a simple SpringBoot application and run it. But you want to understand the magic behind the SpringBoot's AutoConfiguration. But before that you should know about Spring's **@Conditional** feature based on which all the SpringBoot's AutoConfiguration magic depends.

¹⁵<https://github.com/sivaprasadreddy/springboot-learn-by-example>

Chapter 3: SpringBoot Behind the scenes

In the previous chapter we have learned how to create a SpringBoot application in various ways. SpringBoot makes developing Spring based applications very easily using its powerful AutoConfiguration mechanism. But it may not be still clear how SpringBoot is actually working behind the scenes.

In this chapter we will cover:

- Exploring the power of @Conditional
- How SpringBoot AutoConfiguration Works?

3.1. Exploring the power of @Conditional

While developing Spring based applications we may come across of a need to register beans conditionally.

For example, you may want to register a DataSource bean pointing to DEV Database while running application locally and point to a different PRODUCTION Database while running in production.

You can externalize the database connection parameters into properties files and use the file appropriate for the environment. But you need to change the configuration whenever you need to point to a different environment and build the application.

To address this problem **Spring 3.1** introduced the concept of **Profiles**. You can register multiple beans of the same type and associate them to one or more profiles. When you run the application you can activate the desired profiles and beans associated with the activated profiles only will be registered.


```
@Configuration
public class AppConfig
{
    @Bean
    @Profile("DEV")
    public DataSource devDataSource() {
        ...
    }

    @Bean
    @Profile("PROD")
    public DataSource prodDataSource() {
        ...
    }
}
```

Then you can specify the active profile using System Property `-Dspring.profiles.active=DEV`

This approach works for simple cases like enable or disable bean registrations based on activated profiles. But if you want to register beans based on some conditional logic then the Profiles approach itself is not sufficient.

To provide much more flexibility for registering Spring beans conditionally, **Spring 4** introduced the concept of **@Conditional**. By using **@Conditional** approach you can register a bean conditionally based on any arbitrary condition.

For example, you may want to register a bean when:

- A specific class is present in classpath
- A Spring bean of certain type doesn't already registered in ApplicationContext
- A specific file exists on a location
- A specific property value is configured in a configuration file
- A specific system property is present/absent

These are just a few examples only and you can have any condition you want.

Let us take a look at how Spring's **@Conditional** works.

Using @Conditional based on System Property

Suppose we have a **UserDAO** interface with methods to get data from a data store. We have two implements of **UserDAO** interface namely **JdbcUserDAO** which talks to **MySQL** database and **MongoUserDAO** which talks to **MongoDB**.

We may want to enable only one of **JdbcUserDAO** and **MongoUserDAO** based on a System Property say **dbType**.

If the application is started using `java -jar myapp.jar -DdbType=MySQL` then we want to enable **JdbcUserDAO**, otherwise if the application is started using `java -jar myapp.jar -DdbType=MONGO` we want to enable **MongoUserDAO**.

Suppose we have **UserDAO** interface and **JdbcUserDAO**, **MongoUserDAO** implementations as follows:

```
public interface UserDAO
{
    List<String> getAllUserNames();
}

public class JdbcUserDAO implements UserDAO
{
    @Override
    public List<String> getAllUserNames()
    {
        System.out.println("**** Getting usernames from RDBMS ****");
        return Arrays.asList("Siva", "Prasad", "Reddy");
    }
}

public class MongoUserDAO implements UserDAO
{
    @Override
    public List<String> getAllUserNames()
    {
        System.out.println("**** Getting usernames from MongoDB ****");
        return Arrays.asList("Bond", "James", "Bond");
    }
}
```

We can implement the Condition **MySQLDatabaseTypeCondition** to check whether the System Property **dbType** is “**MYSQL**” as follows:

```

public class MySQLDatabaseTypeCondition implements Condition
{
    @Override
    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata metadata)
    {
        String enabledDBType = System.getProperty("dbType");
        return (enabledDBType != null &&
            enabledDBType.equalsIgnoreCase("MYSQL"));
    }
}

```

We can implement the Condition **MongoDBDatabaseTypeCondition** to check whether the System Property **dbType** is “MONGODB” as follows:

```

public class MongoDBDatabaseTypeCondition implements Condition
{
    @Override
    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata metadata)
    {
        String enabledDBType = System.getProperty("dbType");
        return (enabledDBType != null
            && enabledDBType.equalsIgnoreCase("MONGODB"));
    }
}

```

Now we can configure both **JdbcUserDAO** and **MongoUserDAO** beans conditionally using **@Conditional** as follows:

```

@Configuration
public class AppConfig
{
    @Bean
    @Conditional(MySQLDatabaseTypeCondition.class)
    public UserDAO jdbcUserDAO(){
        return new JdbcUserDAO();
    }

    @Bean
    @Conditional(MongoDBDatabaseTypeCondition.class)
    public UserDAO mongoUserDAO(){

```

```

        return new MongoUserDAO();
    }
}

```

Now if we run the application like `java -jar myapp.jar -DdbType=MYSQL` then only `JdbcUserDAO` bean will be registered.

But if you set the System property like `-DdbType=MONGODB` then only `MongoUserDAO` bean will be registered.

Now that we have seen how to conditionally register a bean based on System Property.

Using @Conditional based on Presence/Absence of a Java Class

Suppose we want to register `MongoUserDAO` bean only when MongoDB java driver class `com.mongodb.Server` is available on classpath, if not we want to register `JdbcUserDAO` bean.

To accomplish that we can create Conditions to check the presence or absence of MongoDB driver class `com.mongodb.Server` as follows:

```

public class MongoDriverPresentsCondition implements Condition
{
    @Override
    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata metadata)
    {
        try {
            Class.forName("com.mongodb.Server");
            return true;
        } catch (ClassNotFoundException e) {
            return false;
        }
    }
}

```

```

public class MongoDriverNotPresentsCondition implements Condition
{
    @Override
    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata metadata)
    {
        try {
            Class.forName("com.mongodb.Server");
            return false;
        }
    }
}

```

```

        } catch (ClassNotFoundException e) {
            return true;
        }
    }
}

```

We just have seen how to register beans conditionally based on presence/absence of a class in classpath.

Using @Conditional based on Configured Spring Beans

What if we want to register `MongoUserDAO` bean only if no other Spring bean of type `UserDAO` is already registered.

We can create a Condition to check if there is any existing bean of a certain type as follows:

```

public class UserDAOBeanNotPresentsCondition implements Condition
{
    @Override
    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata metadata)
    {
        UserDAO userDAO = conditionContext.getBeanFactory().getBean(UserDAO.class);
        return (userDAO == null);
    }
}

```

Using @Conditional based on Properties Configuration

What if we want to register `MongoUserDAO` bean only if property `app.dbType=MONGO` is set in properties placeholder configuration file?

We can implement that Condition as follows:

```

public class MongoDBTypePropertyCondition implements Condition
{
    @Override
    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata metadata)
    {
        String dbType = conditionContext.getEnvironment()
            .getProperty("app.dbType");
        return "MONGO".equalsIgnoreCase(dbType);
    }
}

```

We have just seen how to implement various types of Conditions.

But there is even more elegant way to implement Conditions using Annotations.

Instead of creating a Condition implementation for both **MYSQL** and **MongoDB**, we can create a **DatabaseType** annotation as follows:

```

@Target({ ElementType.TYPE, ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Conditional(DatabaseTypeCondition.class)
public @interface DatabaseType
{
    String value();
}

```

Then we can implement **DatabaseTypeCondition** to use the **DatabaseType** value to determine whether to enable or disable bean registration as follows:

```

public class DatabaseTypeCondition implements Condition
{
    @Override
    public boolean matches(ConditionContext conditionContext,
        AnnotatedTypeMetadata metadata)
    {
        Map<String, Object> attributes = metadata
            .getAnnotationAttributes(DatabaseType.class.getName());
        String type = (String) attributes.get("value");
        String enabledDbType = System.getProperty("dbType", "MYSQL");
        return (enabledDbType != null && type != null
            && enabledDbType.equalsIgnoreCase(type));
    }
}

```

Now we can use the `@DatabaseType` annotation on our bean definitions as follows:

```
@Configuration
@ComponentScan
public class AppConfig
{
    @DatabaseType("MYSQL")
    public UserDao jdbcUserDAO(){
        return new JdbcUserDAO();
    }

    @Bean
    @DatabaseType("MONGO")
    public UserDao mongoUserDAO(){
        return new MongoUserDAO();
    }
}
```

Here we are getting the metadata from `DatabaseType` annotation and checking against the System Property `dbType` value to determine whether to enable or disable the bean registration.

We have seen good number of examples to understand how we can register beans conditionally using `@Conditional` annotation.

SpringBoot extensively uses `@Conditional` feature to register beans conditionally based on various criteria. You can find various Condition implementations that SpringBoot uses in `org.springframework.boot.autoconfigure` package of `spring-boot-autoconfigure-{version}.jar`.

Now that we come to know about how SpringBoot uses `@Conditional` feature to conditionally check whether to register a bean or not.

But what exactly triggers the auto-configuration mechanism? This is what we are going to look at in next section.

3.2. How SpringBoot AutoConfiguration Works?

The key to the SpringBoot's auto-configuration magic is `@EnableAutoConfiguration` annotation. Typically we annotate our Application entry point class with either `@SpringBootApplication` or if we want to customize the defaults we can use the following annotations:

```

@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application
{

}

```

The `@EnableAutoConfiguration` annotation enables the auto-configuration of Spring Application-Context by scanning the classpath components and registers the beans that are matching various Conditions.

SpringBoot provides various AutoConfiguration classes in `spring-boot-autoconfigure-{version}.jar` which are responsible for registering various components.

Typically AutoConfiguration classes are annotated with `@Configuration` to mark it as a Spring configuration class and annotated with `@EnableConfigurationProperties` to bind the customization properties and one or more Conditional bean registration methods.

For example consider `org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration` class.

```

@Configuration
@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ Registrar.class, DataSourcePoolMetadataProvidersConfiguration.class })
public class DataSourceAutoConfiguration {
    ...
    ...

    @Conditional(DataSourceAutoConfiguration.EmbeddedDataSourceCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import(EmbeddedDataSourceConfiguration.class)
    protected static class EmbeddedConfiguration {

    }

    @Configuration
    @ConditionalOnMissingBean(DataSourceInitializer.class)
    protected static class DataSourceInitializerConfiguration {
        @Bean
        public DataSourceInitializer dataSourceInitializer() {
            return new DataSourceInitializer();
        }
    }
}

```



```

    }

    @Conditional(DataSourceAutoConfiguration.NonEmbeddedDataSourceCondition.class\
s)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    protected static class NonEmbeddedConfiguration {
        @Autowired
        private DataSourceProperties properties;

        @Bean
        @ConfigurationProperties(prefix = DataSourceProperties.PREFIX)
        public DataSource dataSource() {
            DataSourceBuilder factory = DataSourceBuilder
                .create(this.properties.getClassLoader())
                .driverClassName(this.properties.getDriverClassName())
                .url(this.properties.getUrl()).username(this.properties.getU\
servername())
                .password(this.properties.getPassword());
            if (this.properties.getType() != null) {
                factory.type(this.properties.getType());
            }
            return factory.build();
        }
    }
    ...
    ...

    @Configuration
    @ConditionalOnProperty(prefix = "spring.datasource", name = "jmx-enabled")
    @ConditionalOnClass(name = "org.apache.tomcat.jdbc.pool.DataSourceProxy")
    @Conditional(DataSourceAutoConfiguration.DataSourceAvailableCondition.class)
    @ConditionalOnMissingBean(name = "dataSourceMBean")
    protected static class TomcatDataSourceJmxConfiguration {
        @Bean
        public Object dataSourceMBean(DataSource dataSource) {
            ....
            ....
        }
    }
    ...
    ...
}

```

Here `DataSourceAutoConfiguration` is annotated with `@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })` which means that Auto Configuration of beans within `DataSourceAutoConfiguration` will be considered only if `DataSource.class` and `EmbeddedDatabaseType.class` classes are available on classpath.

The class is also annotated with `@EnableConfigurationProperties(DataSourceProperties.class)` which enables binding the properties in `application.properties` to the properties of `DataSourceProperties` class automatically.

```
@ConfigurationProperties(prefix = DataSourceProperties.PREFIX)
public class DataSourceProperties
    implements BeanClassLoaderAware, EnvironmentAware, InitializingBean {

    public static final String PREFIX = "spring.datasource";
    ...
    ...
    private String driverClassName;
    private String url;
    private String username;
    private String password;
    ...
    //setters and getters
}
```

With this configuration all the properties that starts with `spring.datasource.*` will be automatically binds to `DataSourceProperties` object.

```
spring.datasource.url=jdbc:mysql://localhost:3306/test
spring.datasource.username=root
spring.datasource.password=secret
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

You can also see some inner classes and bean definition methods that are annotated with Spring-Boot's Conditional annotations such as `@ConditionalOnMissingBean`, `@ConditionalOnClass` and `@ConditionalOnProperty` etc. These bean definitions will be registered in `ApplicationContext` only if those conditions are matched.

You can also explore many other AutoConfiguration classes in `spring-boot-autoconfigure-{version}.jar` such as

- `org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration`
- `org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration`

- `org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration`
- `org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration`

etc.

3.3. Summary

Now that we have an understanding of how SpringBoot auto-configuration works by using various AutoConfiguration classes along with **@Conditional** features.

In next chapter, we will look into how to create our own Custom SpringBoot Starter.

Chapter 4: Creating Custom SpringBoot Starter

SpringBoot provides lot of starter modules to get up and running quickly. SpringBoot's auto-configure mechanism takes care of configuring SpringBeans on our behalf based on various criteria.

In addition to the springboot starters that comes out-of-the-box provided by Core Spring Team, we can also create our own starter modules.

In this post we will look into how to create a custom SpringBoot starter. To demonstrate it we are going to create `twitter4j-spring-boot-starter` which will auto-configure `Twitter4J` beans.

To accomplish this, we are going to create:

1. `twitter4j-spring-boot-autoconfigure` module which contains `Twitter4J` `AutoConfiguration` bean definitions
2. `twitter4j-spring-boot-starter` module which pulls in `twitter4j-spring-boot-autoconfigure` and `twitter4j-core` dependencies
3. Sample application which uses `twitter4j-spring-boot-starter`

4.1. Create Parent Module `spring-boot-starter-twitter4j`

First we are going to create a parent pom type module to define dependency versions and sub-modules.

```
<?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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.sivalabs</groupId>
  <artifactId>spring-boot-starter-twitter4j</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
```

```

<name>spring-boot-starter-twitter4j</name>

<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <twitter4j.version>4.0.3</twitter4j.version>
  <spring-boot.version>1.3.6.RELEASE</spring-boot.version>
</properties>

<modules>
  <module>twitter4j-spring-boot-autoconfigure</module>
  <module>twitter4j-spring-boot-starter</module>
  <module>twitter4j-spring-boot-sample</module>
</modules>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>${spring-boot.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>

    <dependency>
      <groupId>org.twitter4j</groupId>
      <artifactId>twitter4j-core</artifactId>
      <version>${twitter4j.version}</version>
    </dependency>
  </dependencies>
</dependencyManagement>

</project>

```

In this `pom.xml` we are defining the SpringBoot and Twitter4j versions in section so that we don't need to specify versions all over the places.

4.2. Create twitter4j-spring-boot-autoconfigure module

Create a child module with name `twitter4j-spring-boot-autoconfigure` in our parent maven module `spring-boot-starter-twitter4j`.

Add the maven dependencies such as spring-boot, spring-boot-autoconfigure, twitter4j-core and junit as follows:

```
<?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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sivalabs</groupId>
  <artifactId>twitter4j-spring-boot-autoconfigure</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>

  <parent>
    <groupId>com.sivalabs</groupId>
    <artifactId>spring-boot-starter-twitter4j</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-autoconfigure</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-configuration-processor</artifactId>
      <optional>>true</optional>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

```

    <dependency>
      <groupId>org.twitter4j</groupId>
      <artifactId>twitter4j-core</artifactId>
      <optional>true</optional>
    </dependency>
  </dependencies>
</project>

```

Note that we have specified `twitter4j-core` as optional dependency because `twitter4j-core` should be added to the project only when `twitter4j-spring-boot-starter` is added to the project.

Create Twitter4jProperties to hold the Twitter4J config parameters

Create `Twitter4jProperties.java` to hold the Twitter4J OAuth config parameters.

```

package com.sivalabs.spring.boot.autoconfigure;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.NestedConfigurationProperty;

@ConfigurationProperties(prefix= Twitter4jProperties.TWITTER4J_PREFIX)
public class Twitter4jProperties {

    public static final String TWITTER4J_PREFIX = "twitter4j";

    private Boolean debug = false;

    @NestedConfigurationProperty
    private OAuth oauth = new OAuth();

    public Boolean getDebug() {
        return debug;
    }

    public void setDebug(Boolean debug) {
        this.debug = debug;
    }

    public OAuth getOAuth() {
        return oauth;
    }
}

```

```
public void setOAuth(OAuth oauth) {
    this.oauth = oauth;
}

public static class OAuth {

    private String consumerKey;
    private String consumerSecret;
    private String accessToken;
    private String accessTokenSecret;

    public String getConsumerKey() {
        return consumerKey;
    }
    public void setConsumerKey(String consumerKey) {
        this.consumerKey = consumerKey;
    }
    public String getConsumerSecret() {
        return consumerSecret;
    }
    public void setConsumerSecret(String consumerSecret) {
        this.consumerSecret = consumerSecret;
    }
    public String getAccessToken() {
        return accessToken;
    }
    public void setAccessToken(String accessToken) {
        this.accessToken = accessToken;
    }
    public String getAccessTokenSecret() {
        return accessTokenSecret;
    }
    public void setAccessTokenSecret(String accessTokenSecret) {
        this.accessTokenSecret = accessTokenSecret;
    }
}
}
```

The `@ConfigurationProperties` annotation allows us to bind a set of properties with a common prefix to a Java bean properties. With this configuration object we can configure the twitter4j properties in `application.properties` as follows:


```
twitter4j.debug=true
twitter4j.oauth.consumer-key=your-consumer-key-here
twitter4j.oauth.consumer-secret=your-consumer-secret-here
twitter4j.oauth.access-token=your-access-token-here
twitter4j.oauth.access-token-secret=your-access-token-secret-here
```

Create Twitter4jAutoConfiguration to auto-configure Twitter4j

Here comes the key part of our starter. `Twitter4jAutoConfiguration` configuration class contains the bean definitions that will be automatically configured based on some criteria.

What is that criteria?

- If `twitter4j.TwitterFactory.class` is on classpath
- If `TwitterFactory` bean is not already defined explicitly

So, the `Twitter4jAutoConfiguration` goes like this.

```
package com.sivalabs.spring.boot.autoconfigure;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import twitter4j.Twitter;
import twitter4j.TwitterFactory;
import twitter4j.conf.ConfigurationBuilder;

@Configuration
@ConditionalOnClass({ TwitterFactory.class})
@EnableConfigurationProperties(Twitter4jProperties.class)
public class Twitter4jAutoConfiguration {

    private static Log log = LogFactory.getLog(Twitter4jAutoConfiguration.class);

    @Autowired
    private Twitter4jProperties properties;
```

```

@Bean
@ConditionalOnMissingBean
public TwitterFactory twitterFactory(){

    if (this.properties.getOauth().getConsumerKey() == null
        || this.properties.getOauth().getConsumerSecret() == null
        || this.properties.getOauth().getAccessToken() == null
        || this.properties.getOauth().getAccessTokenSecret() == null)
    {
        String msg = "Twitter4j properties not configured properly." +
            " Please check twitter4j.* properties settings in confi\
uration file.";
        log.error(msg);
        throw new RuntimeException(msg);
    }

    ConfigurationBuilder cb = new ConfigurationBuilder();
    cb.setDebugEnabled(properties.getDebug())
        .setOAuthConsumerKey(properties.getOauth().getConsumerKey())
        .setOAuthConsumerSecret(properties.getOauth().getConsumerSecret())
        .setOAuthAccessToken(properties.getOauth().getAccessToken())
        .setOAuthAccessTokenSecret(properties.getOauth().getAccessTokenSecret(\
));

    TwitterFactory tf = new TwitterFactory(cb.build());
    return tf;
}

@Bean
@ConditionalOnMissingBean
public Twitter twitter(TwitterFactory twitterFactory){
    return twitterFactory.getInstance();
}
}

```

We have used `@ConditionalOnClass({ TwitterFactory.class})` to specify that this auto configuration should take place only when `TwitterFactory.class` class is present.

We have also used `@ConditionalOnMissingBean` on bean definition methods to specify consider this bean definition only if `TwitterFactory/Twitter` beans are not already defined explicitly.

Also note that we have annotated with `@EnableConfigurationProperties(Twitter4jProperties.class)` to enable support for `ConfigurationProperties` and injected `Twitter4jProperties` bean.

Now we need to configure our custom `Twitter4jAutoConfiguration` in `src/main/resources/META-INF/spring.factories` file as follows:

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.sivalabs.spring.boot.autoconfigure
```

4.3. Create twitter4j-spring-boot-starter module

Create a child module with name `twitter4j-spring-boot-starter` in our parent maven module `spring-boot-starter-twitter4j`.

```
<?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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sivalabs</groupId>
  <artifactId>twitter4j-spring-boot-starter</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>

  <parent>
    <groupId>com.sivalabs</groupId>
    <artifactId>spring-boot-starter-twitter4j</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
      <groupId>com.sivalabs</groupId>
      <artifactId>twitter4j-spring-boot-autoconfigure</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>
</project>
```

```

    <dependency>
      <groupId>org.twitter4j</groupId>
      <artifactId>twitter4j-core</artifactId>
    </dependency>

  </dependencies>

</project>

```

Note that in this maven module we are actually pulling in **twitter4j-core** dependency.

We don't need to add any code in this module, but optionally we can specify what are the dependencies we are going to provide through this starter in **src/main/resources/META-INF/spring.provides** file as follows:

```
provides: twitter4j-core
```

That's all for our starter.

Let us create a sample using our brand new starter **twitter4j-spring-boot-starter**.

4.4. Create twitter4j-spring-boot-sample sample application

Let us create a simple SpringBoot application and add our **twitter4j-spring-boot-starter** dependency.

```

<?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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.sivalabs</groupId>
  <artifactId>twitter4j-spring-boot-sample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.6.RELEASE</version>

```

```
</parent>

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

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

<dependencies>

  <dependency>
    <groupId>com.sivalabs</groupId>
    <artifactId>twitter4j-spring-boot-starter</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

</project>
```

Create the entry-point class **SpringbootTwitter4jDemoApplication** as follows:

```
package com.sivalabs.demo;

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

@SpringBootApplication
public class SpringbootTwitter4jDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootTwitter4jDemoApplication.class, args);
    }
}
```

Create TweetService as follows:

```
package com.sivalabs.demo;

import java.util.ArrayList;
import java.util.List;

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

import twitter4j.ResponseList;
import twitter4j.Status;
import twitter4j.Twitter;
import twitter4j.TwitterException;

@Service
public class TweetService {

    @Autowired
    private Twitter twitter;

    public List<String> getLatestTweets(){
        List<String> tweets = new ArrayList<>();
        try {
            ResponseList<Status> homeTimeline = twitter.getHomeTimeline();
            for (Status status : homeTimeline) {
                tweets.add(status.getText());
            }
        } catch (TwitterException e) {
```

```

        throw new RuntimeException(e);
    }
    return tweets;
}
}

```

Now create a Test to verify our Twitter4j AutoConfiguration.

Before that make sure you have set your twitter4j oauth configuration parameter to your actual values. You can get them from <https://apps.twitter.com/>

```

package com.sivalabs.demo;

import java.util.List;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import twitter4j.TwitterException;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(SpringbootTwitter4jDemoApplication.class)
public class SpringbootTwitter4jDemoApplicationTest {

    @Autowired
    private TweetService tweetService;

    @Test
    public void testGetTweets() throws TwitterException {
        List<String> tweets = tweetService.getLatestTweets();
        for (String tweet : tweets) {
            System.err.println(tweet);
        }
    }
}

```

Now when you run this JUnit test you should be able to see the latest tweets on your console output.

To learn more about creating custom AutoConfiguration classes or starters take a look at [Creating your own auto-configuration](#)¹⁶

4.5. Summary

In this chapter, we learned about how to create our own AutoConfiguration classes and our own SpringBoot Starter. In next chapter, we will look into various SpringBoot features which we will be using commonly in Spring based applications.

¹⁶<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-developing-auto-configuration>

Appendix A : Resources

You can find the useful information about SpringBoot at the following URLs:

- [Spring project site](#)¹⁷
- [SpringBoot project site](#)¹⁸
- [SpringBoot official documentation](#)¹⁹
- [SpringBoot quick start guides](#)²⁰
- [SpringBoot GitHub repository](#)²¹

¹⁷<http://spring.io/>

¹⁸<http://projects.spring.io/spring-boot/>

¹⁹<http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>

²⁰<http://spring.io/guides>

²¹<https://github.com/spring-projects/spring-boot>