**Spring Overview**

The Spring framework is Java based framework for building web and enterprise applications. Spring at its core is a dependency injection container that provides flexibility to configure beans in multiple ways, such as XML, Annotations, and Java Config.

The Spring framework was created primarily as a dependency injection container, but it is much more than that. Spring is very popular for 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 the JPA/Hibernate ORM and Struts/JSF web frameworks

• State-of-the-art Web MVC framework for building web applications

Along with the Spring framework, there are many other Spring sub-projects that help build applications that address modern business needs:

• **Spring Data:** Simplifies data access from relational and NoSQL datastores.

• **Spring Batch:** Provides a 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 we can see at http://spring.io/projects. For addressing various other modern application development needs.

**Spring Configuration Styles**

Spring initially provided an XML-based approach for configuring beans. Later Spring introduced XML-based DSLs, Annotations, and JavaConfig-based approaches for configuring beans.

**Example of XML-Based Configuration**

|  |
| --- |
| <bean id="userService" class="com.nareshit.hmspro.service.UserService">  <property name="userDao" ref="userDao"/>  </bean>  <bean id="userDao" class="com.nareshit.hmspro.dao.JdbcUserDao">  <property name="dataSource" ref="dataSource"/>  </bean>  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroymethod="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>  **<!-- DSL based configuration -->**  <beans>  <jee:jndi-lookup id="entityManagerFactory" jndi-name="persistence/defaultPU"/>  </beans> |
|  |

**Example of 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;  }  } |

**Example of a 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;  }} |

As we can see, Spring provides multiple approaches for configuring application components and we can even mix the approaches as well. For example, we can use JavaConfig- and Annotation-based configuration styles in the same application. As of now, the Spring community is suggesting us to follow the JavaConfig-based approach, as it gives us more flexibility.

**Developing Web Application Using SpringMVC and JPA without Boot**

Refer Handout:Spring-Mvc-Project-without-boot.docx

**Problems with traditional Spring Mvc application development**

1. spring consists of multiple modules like Spring Web MVC, Spring DAO, Spring ORM etc. Which we can use individually or integrate one into another. This integration configurations we always had to do either in XML approach or in annotation based approach.
2. When configuring these modules dependencies in pom.xml file, we need to ensure the jar version compatibility.
3. While integrating the above modules we need to ensure the all module specific custom components (java classes) are available for component-scan so that they can eligible for auto wiring.
4. The application developed with the above modules must need to deploy in another external container like Apache tomcat.

The above problems were addressed with the Spring Boot features.

**What Is Spring Boot?**

Spring Boot is an **opinionated framework with the convention over configuration approach** to build Spring-based applications quickly and easily. The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again.

The key Spring Boot features include:

* Easy Dependency Management
* AutoConfiguration
* Embedded servlet container support
* Spring Boot starters
* Spring Boot actuator

**Easy Dependency Management**

The first thing to note is the use of the dependencies named **spring-boot-starter-\***. Remember that “Most of the time, we use the same configuration”. So when we add the **spring-boot-starter-web** dependency, it will by default pull all the commonly used libraries while developing Spring MVC applications, such as spring-webmvc, jackson-json,validation-api, and tomcat.

We added the spring-boot-starter-data-jpa dependency. This pulls all the spring-datajpa dependencies and adds Hibernate libraries because most applications use Hibernate as a JPA implementation.

**Spring Boot AutoConfiguration**

Spring Boot addresses the problem that , the traditional Spring applications requires complex configuration by eliminating the need to manually set up the boilerplate configuration.

The spring-boot-starter-web add all these libraries but it also configures the commonly registered beans like DispatcherServlet, ResourceHandlers, MessageSource, etc. with sensible defaults.

If we have any in-memory database drivers like H2 or HSQL in the classpath, then Spring Boot will automatically create an in-memory datasource and will register the EntityManagerFactory and TransactionManager beans automatically with sensible defaults

Spring Boot uses an opinionated view of approach with Convention Over Configuration paradigm if **@EnableAutoConfiguration** or **@SpringBootApplication annotations are using** and configures various components automatically, by registering beans based on various criteria. The criteria can be:

* Availability of a particular class in a classpath
* Presence or absence of a Spring bean
* Presence of a system property
* Absence of a configuration file

For example, if we have the spring-webmvc dependency in our classpath, Spring Boot assumes we are trying to build a SpringMVC-based web application and automatically tries to register DispatcherServlet if it is not already registered.

If we have any embedded database drivers in the classpath, such as H2 or HSQL, and if we haven’t configured a DataSource bean explicitly, then Spring Boot will automatically register a DataSource bean using in-memory database settings.

**Spring Boot Starters**

Spring Boot offers many starter modules 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 we don’t have to search for the compatible library versions and configure them manually.

For example, the **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.

**Spring Boot Actuator**

getting the various details of an application running in production is crucial to many applications. The Spring Boot actuator provides a wide variety of such **production-ready features** without requiring developers to write much code. Some of the Spring actuator features are:

* **mappings:** This lists all the HTTP request mappings
* **info:** This displays information about the application
* **health:** This displays the application's health conditions
* **metrics:** This shows different metrics collected from the application
* **dump:** This performs a thread dump and displays the result

**Embedded Servlet Container Support**

Traditionally, while building web applications, we need to create WAR type modules and then deploy them on external servers like Tomcat, WildFly, etc. But by using Spring Boot, we can create a JAR type module and embed the Servlet container in the application very easily so that the application will be a self-contained deployment unit. Also, during development, we can easily run the Spring Boot JAR type module as a Java application from the IDE or from the command-line using a build tool like Maven or Gradle.

The most important thing is that when we created spring boot application then we will get a simple Java class annotated with some magical annotation (@SpringApplication), which has a main() method. By running that main() method, we are able to run the application and access it at **http://localhost:8080/**. Where does the servlet container come from?

We added **spring-boot-starter-web**, which pulls **spring-boot-starter-tomcat** automatically. When we run the main() method, it starts tomcat as an embedded container so that we don’t have to deploy our application on any externally installed tomcat server. What if we want to use a Jetty server instead of Tomcat? We simply exclude spring-boot-starter-tomcat from spring-boot-starter-web and include spring-boot-starter-jetty.

**Creating A New Spring Boot Application**

There are many ways of creating Spring Boot-based applications:

* Using the Spring Boot CLI as a command-line tool
* Using IDEs such as STS to provide Spring Boot, which are supported out of the box
* Using the Spring Initialize project at [**http://start.spring.io**](http://start.spring.io)

**Using the Spring Boot CLI**

The Spring Boot CLI is a command-line tool to create and run the spring boot applications by perform the following steps:

1. Install the Spring Boot command-line tool by downloading the [spring-boot-cli-1.5.10.BUILD-20180130.154323-54-bin.zip](https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-cli/1.5.10.BUILD-SNAPSHOT/spring-boot-cli-1.5.10.BUILD-20180130.154323-54-bin.zip)

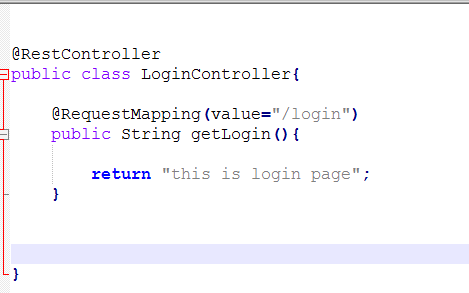
file from https://repo.spring.io/snapshot/org/springframework/boot/spring-boot-cli/1.5.10.BUILD-SNAPSHOT/

2. Unzip the file into a directory of our choice. Open a terminal window and change the terminal prompt to the bin folder. Ensure that the bin folder is added to the system path so that Spring Boot can be run from any location.

3. Verify the installation with the following command. If successful, the Spring CLI version will be printed in the console:

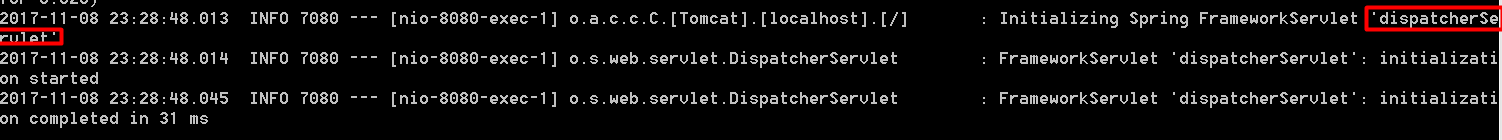
**$spring –-version**

**Spring CLI** 1.5.10.BUILD-SNAPSHOT

4. As the next step, a quick REST service will be developed, which is supported out of the box in Spring Boot. To do so, copy and paste the following code using any editor of choice and save 

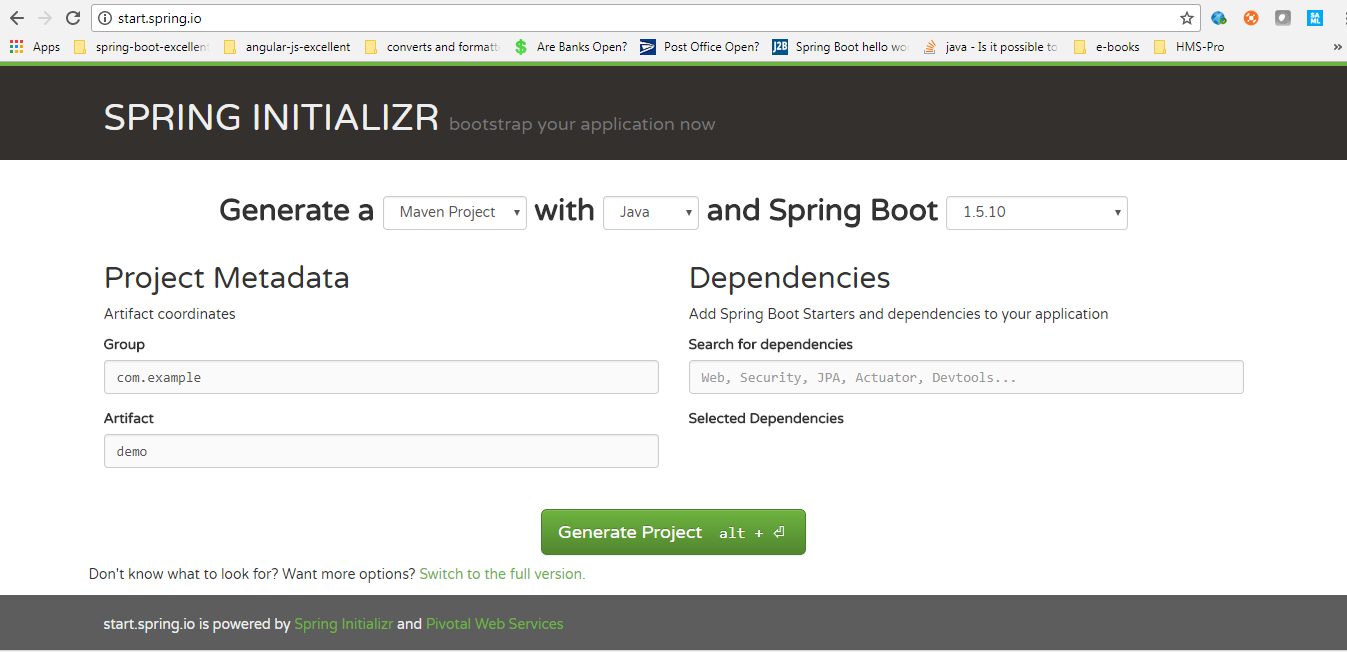
In order to run this Java application, go to the folder where LoginController.java is saved and execute the following command. The last few lines of the server start-up log will be similar to the following:

**$spring run LoginController.java**



**Using Spring Initializer**

We can point our browser to **http://start.spring.io/** and see the project details, as shown in Figure:



1. Select Maven Project and Spring Boot version

2.Enter the Maven project details as follows:

•Group: com.capgemini

•Artifact: springboot-sample

•Name: springboot-sample

•Package Name: com.capgemini.sample

•Packaging: JAR

•Java version: 1.8

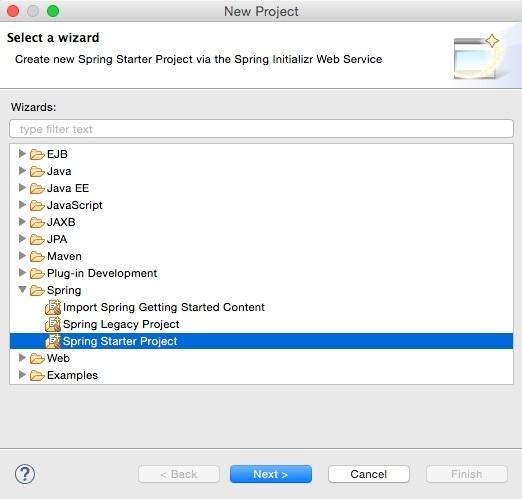
•Language: Java

3. we can search for the starters if we are already familiar with their names or click on the Switch to the Full Version link to see all the available starters. We’ll see many starter modules organized into various categories, like Core, Web, Data, etc. Select the Web check box from the Web category.

4. Click on the Generate Project button.

Now we can extract the downloaded ZIP file and import it into our favorite IDE.

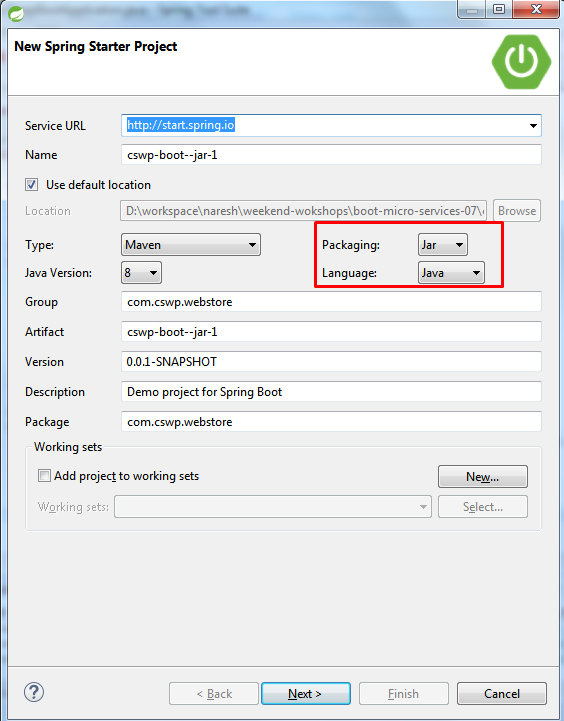
**Creating a new Spring Boot Project using the STS**

1. Open STS, right-click within the **Project Explorer** window, navigate to **New** | **Project**, and select **Spring Starter Project**, as shown in the following screenshot, and click on **Next**:

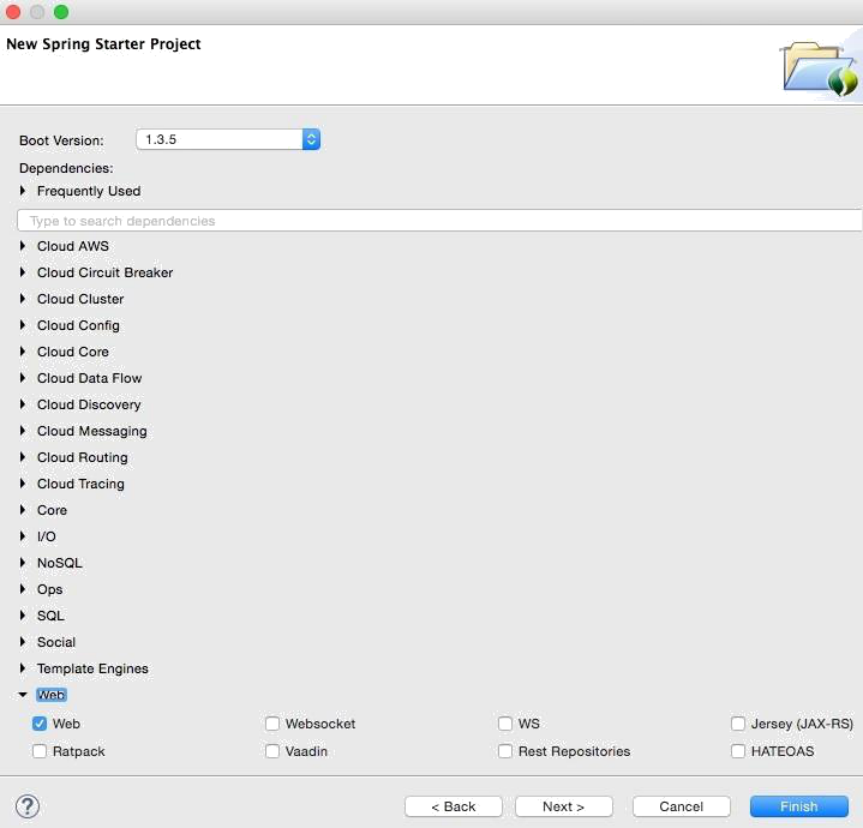
Spring Starter Project is a basic template wizard that provides a number of other starter libraries to select from.

2.Type the project name as chapter2.bootrest or any other name of your choice. It is important to choose the packaging as JAR. In traditional web applications, a war file is created and then deployed to a servlet container, whereas Spring Boot packages all the dependencies to a self-contained, autonomous JAR file with an embedded HTTP listener.

3.Select 1.8 under Java Version. Java 1.8 is recommended for Spring 4 applications. Change the other Maven properties such as Group, Artifact, and Package, as shown in the following screenshot:

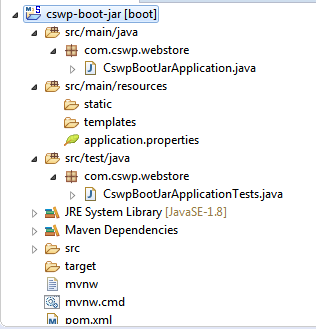


1. Once completed, click on **Next**.
2. The wizard will show the library options. In this case, as the REST service is developed, select **Web** under **Web**. This is an interesting step that tells Spring Boot that a Spring MVC webapplication is being developed so that Spring Boot can include the necessary libraries, including Tomcat as the HTTP listener and other configurations, as required:



1. Click on **Finish**.

This will generate a project named chapter2.bootrest in **Project Explorer** in STS:



7. Take a moment to examine the generated application. Files that are of interest are:

* pom.xml
* Application.java
* Application.properties
* ApplicationTests.java

**Examining the POM file**

The parent element is one of the interesting aspects in the pom.xml file. Take a look at the following:

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>**spring-boot-starter-parent**</artifactId>  <version>1.3.4.RELEASE</version>  </parent> |

The **spring-boot-starter-parent** pattern is a bill of materials (BOM), a pattern used by Maven's

dependency management. BOM is a special kind of POM file used to manage different library versions required for a project.

The cswp-boot-jar Maven module is inheriting from the spring-boot-starter-parent module. By inheriting from spring-boot-starter-parent, this new module will automatically have the following benefits:

• we only need to specify the Spring Boot version once in the parent module configuration. We don’t need to specify the version for all the starter dependencies and other supporting libraries. To see the list of supporting libraries, check out the pom.xml file of the **org.springframework.boot:springboot-dependencies:{version}** Maven module.

• The parent module **spring-boot-starter-parent** already includes the most commonly used plugins, such as maven-jar-plugin, maven-surefire-plugin, maven-war-plugin, and maven-resources-plugin, with sensible defaults.

• In addition to the previously mentioned plugins, the **spring-boot-starter-parent** module also configures the **spring-boot-maven-plugin**, which will be used to build fat JARs.

We selects only web starter, but test starter is also included by default. We 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 the Maven compiler in the spring-boot-starter-parent module.

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

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

The <dependency> element on **spring-boot-starter-web** tells Spring Boot that the application is a web application and lets Spring Boot form its opinions accordingly.

Based on this starter, Spring Boot has formed the following opinions:

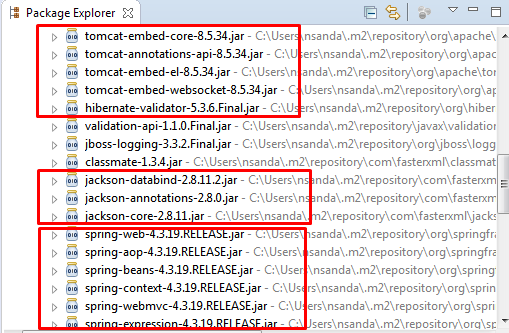
• Spring MVC for the REST framework

• Apache Jackson for the JSON binding

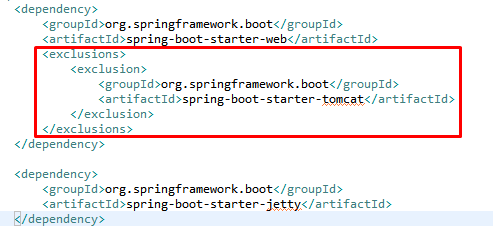
• Tomcat embedded web server container

• Hibernate for object-relational mapping

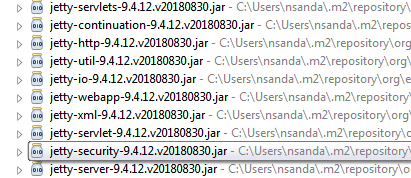
After Spring Boot forms an opinion about the type of application we intend to build, Spring Boot delivers a set of Maven dependencies based on the POM contents and starter specified for the cswp-boot-jar application.



We can see in that Tomcat is the default embedded web server container. Suppose we want to use Jetty instead of Tomcat; all we need to do is change the <dependencies> section in the POM, as shown below

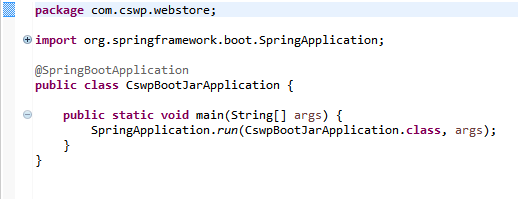


We can see below that the Maven dependencies for Tomcat are replaced with dependencies for Jetty.

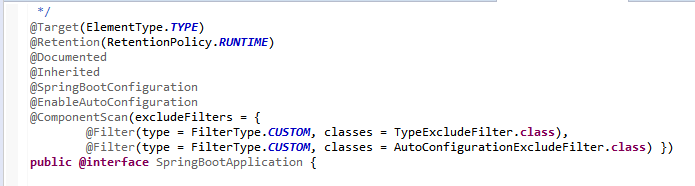


**The Application Entry Point Class**

The generated Spring Boot JAR type module will have an application entry point Java class called **CswpBootJarApplication**.**java** with the public static void main(String[] args) method, which we can run to start the application.



Here the **CswpBootJarApplication.java** class is annotated with the @SpringBootApplication annotation, which is a composed annotation



The @**SpringBootConfiguration** is another composed annotation with the @Configuration annotation.



Here are the meanings of these annotations:

• @EnableAutoConfiguration triggers Spring Boot’s autoconfiguration mechanisms.

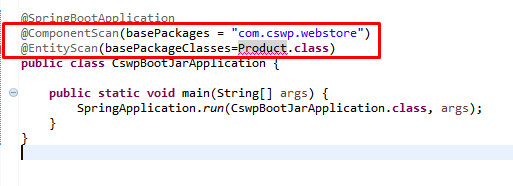
• @Configuration indicates that this class is a Spring configuration class.

• @ComponentScan enables component scanning for Spring beans in the package in which the current class is defined.

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

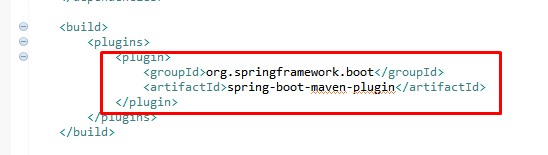
It is highly recommended that we put the main entry point class in the root package, say in com.cswp.webstore, so that the @EnableAutoConfiguration and @ComponentScan annotations will scan for Spring beans, JPA entities, etc., in the root and all of its sub-packages automatically.

If we have an entry point class in a nested package, we might need to specify the basePackages to scan or Spring components explicitly as shown below

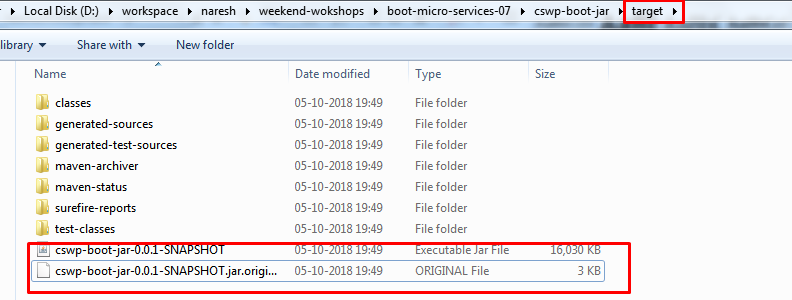


**Fat JAR Using the Spring Boot Maven Plugin**

We can run our application directly from the IDE or use Maven spring-boot:run during development, but ultimately we need to create a deployment unit that can be run in the production environment without any IDE support. We can use **spring-boot-maven-plugin** to create a single deployment unit (a fat JAR) by executing the **mvn clean package** Maven goals.

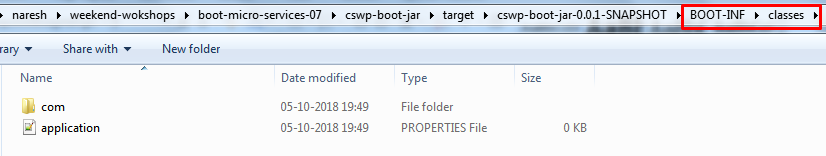


Now there are two interesting files in the target directory—**cswp-boot-jar-0.0.1-SNAPSHOT** and **cswp-boot-jar-0.0.1-SNAPSHOT.jar.original** as shown below:

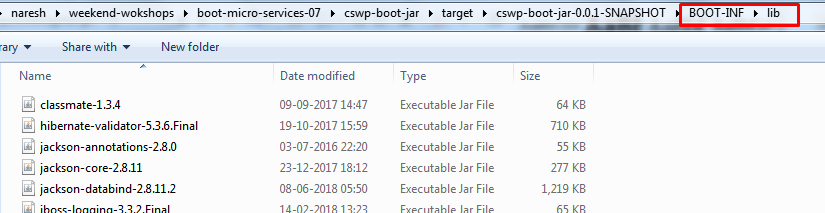


The **cswp-boot-jar-0.0.1-SNAPSHOT.jar.original** file will contain only the compiled classes and classpath resources. But if we look at **cswp-boot-jar-0.0.1-SNAPSHOT.jar**, we find the following:

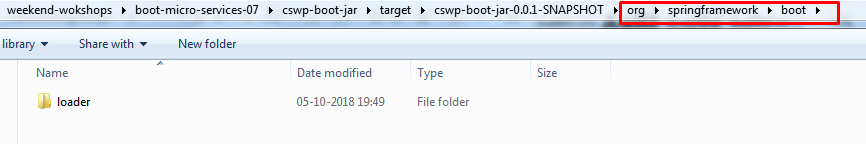
• Compiled classes of our own source code in src/main/java and static resources from src/main/resources will be in the BOOT-INF/classes directory:



• All the dependent JARs in the BOOT-INF/lib directory



• Classes in the org.springframework.boot.loader package that do the Spring Boot magic of running the Spring Boot application



We can create self-contained deployment units for JAR-type modules using plugins like maven-shade-plugin, which packages all the dependent JAR classes into a single JAR file. But Spring Boot follows a different approach and it allows us to nest JARs directly within our Spring Boot application JAR file. We can read more about at: http://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar.

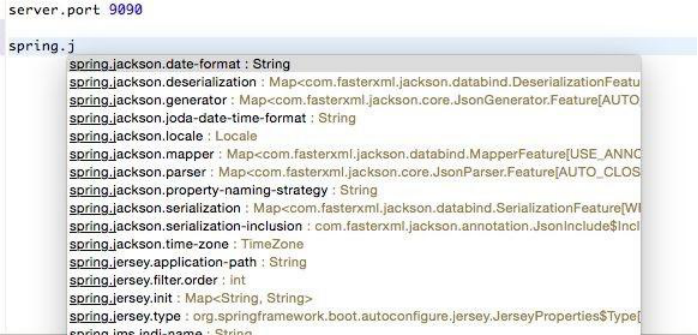
We can run the application using the following command:

**java -jar cswp-boot-jar-0.0.1-SNAPSHOT.jar**

**Overriding the default configuration values**

It is also possible to override default configuration values using the application.properties file.

STS provides an easy-to-autocomplete, contextual help on application.properties, as shown in the following screenshot:



In the preceding screenshot, server.port is edited to be set as 9090. Running this application again will start the server on port 9090.

**Changing the location of the configuration file:**

According to the Twelve-Factor app principles,configuration parameters need to be externalized from the code. Spring Boot externalizes all configurations into application.properties. However, it is still part of the application's build. Furthermore, properties can be read from outside the package by setting the following properties:

**spring.config.name= # config file name**

**spring.config.location= # location of config file**

Here, spring.config.location could be a local file location.

The following command starts the Spring Boot application with an externally provided configuration file:

**$java -jar target/bootadvanced-0.0.1-SNAPSHOT.jar -- spring.config.name=bootrest.properties**

**Reading the Custom Properties:**

At startup, SpringApplication loads all the properties and adds them to the Spring Environment class. Add a custom property to the application.properties file. In this case, the custom property is named **bootrest.customproperty**. Autowire the Spring Environment class into the GreetingController class. Edit the GreetingController class to read the custom property from Environment and add a log statement to print the custom property to the console.

Perform the following steps to do this:

1. Add the following property to the application.properties file:

bootrest.customproperty=hello

2. Then, edit the GreetingController class as follows:

@Autowired

Environment env;

Greet greet(){

logger.info("bootrest.customproperty "+

env.getProperty("bootrest.customproperty")); return new Greet("Hello World!");

}

3. Rerun the application. The log statement prints the custom variable in the console, as follows:

**GreetingController : bootrest.customproperty hello**

1. **Using a .yml file for configuration:**

As an alternate to application.properties, one may use a .yaml file. YAML provides a JSON-like structured configuration compared to the flat properties file.To see this in action, simply replace application.properties with application.yaml and add the following property:



Rerun the application to see the port printed in the console.

**Changing the Default Embedded Web Server**

Embedded HTTP listeners can easily be customized as follows. By default, Spring Boot supports Tomcat, Jetty, and Undertow. In the following example, Tomcat is replaced with Undertow:

