Spring boot in eclipse MARS :

In Eclipse, go to Help > Install New Software

Use the update url

http://dist.springsource.org/snapshot/GRECLIPSE/e4.5/ for

Eclipse Mars (4.5.x).

Select the

– Groovy Compiler 2.4 Feature and

– Groovy-Eclipse Feature.

They both give us the Groovy support we need. Next.

Optionally, Install YAML support

In Eclipse, again go to Help > Install New Software

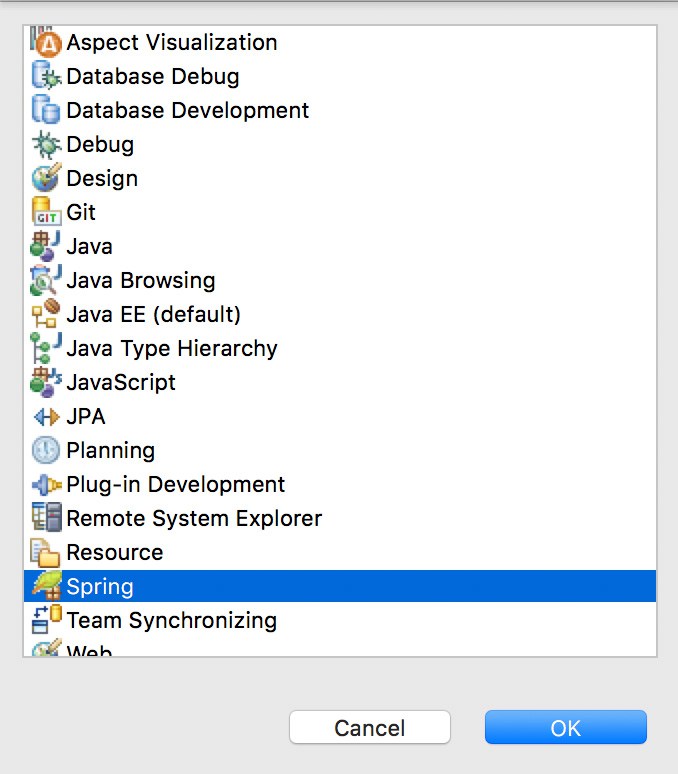
Use the SpringSource Update Site for Eclipse 4.5 update url

http://dist.springsource.com/release/TOOLS/update/e4.5/

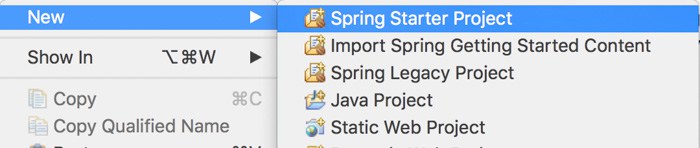
Select and install Spring IDE Core

Creating spring boot application in eclipse mars :

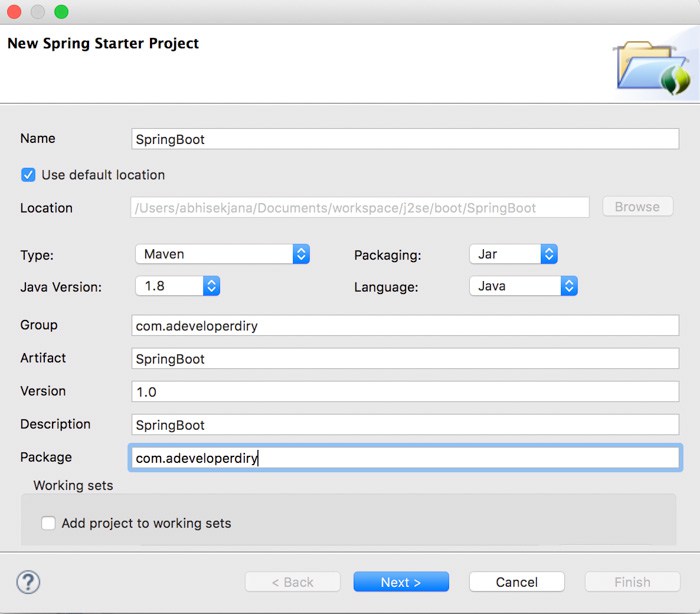
* open Spring perspective.



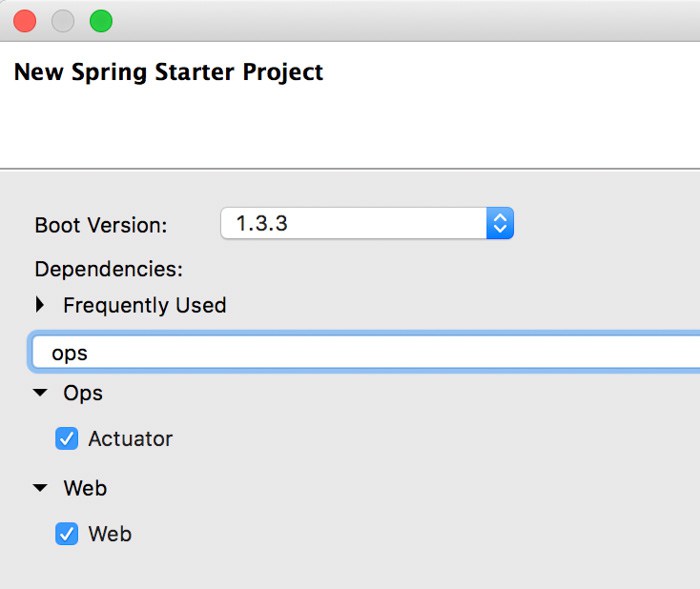
Create a New -> Spring Starter Project.



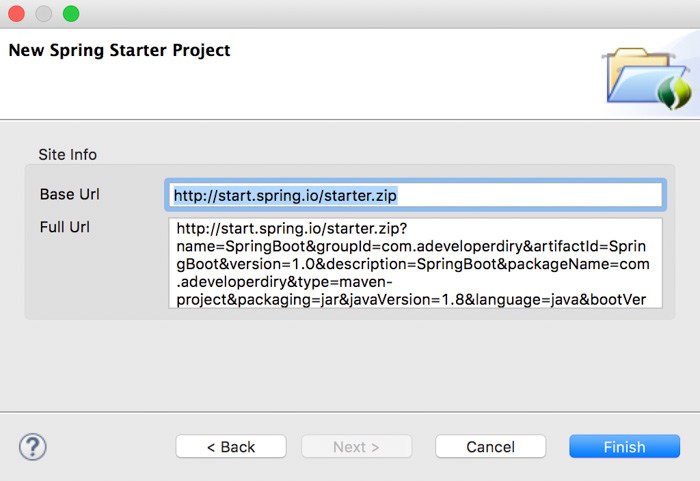
Enter the following details. Click Next.



Select Web and Actuator. Click on Next.



Click on Finish.



This should create the Spring Boot Project in Eclipse.

Running the app :

Mvn spring-boot:run

This will open a web browser to localhost:8080

To gracefully exit the application hit ctrl-c

To create an executable jar we need to add the spring-boot-maven-plugin to our pom.xml

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

Then “your-project”-1.0-snapshot.jar it is around in high size.If you want to peek inside ,you can use jar tvf:

jar tvf target/“your-project”-1.0-snapshot.jar

To run that application ,use the java –jar command:

java –jar target/“your-project”-1.0-snapshot.jar

with Spring Boot, you can create standalone applications that use an embedded server, making them 100% runnable applications.

the SpringApplication singleton class in the main method that will execute the application. The run method call accepts two parameters—the class that actually contains the annotated @SpringBootApplication annotation and the application’s arguments.

Spring Boot has many features that make it suitable for:

• Cloud Native Applications that follow the 12 factor patterns (developed by the Netflix

engineering team at http://12factor.net/ )

• Productivity increases by reducing time of development and deployment

• Enterprise-production-ready Spring applications

• Non-functional requirements, such as the Spring Boot Actuator (a module that

brings metrics, health checks, and management easily) and embedded containers

for running web applications (such as Tomcat, Undertow, Jetty, etc.)

• The SpringApplication class. I showed you that in a Java Spring Boot application,

the main method executes this singleton class. This particular class provides a

convenient way to initiate a Spring application.

• Spring Boot allows you to create applications without requiring any XML

configuration. Spring Boot doesn’t generate code.

• Spring Boot provides a fluent builder API through the SpringApplicationBuilder

singleton class that allows you to create hierarchies with multiple application

contexts.

• The ApplicationArguments interface. Spring Boot allows you to access any

application arguments. This is useful when you want to run your application with

some parameters. For example, you can use --debug mylog.txt or --audit=true

and have access to those values.

• Spring Boot allows you to execute code after the application has started. The only

thing you need to do is implement the CommandLineRunner interface and provide

the implementation of the run(String ...args) method. A particular example is to

initialize some records in a database as it starts or check on some services and see if

they are running before your application starts.

• Spring Boot allows you to externalize configurations by using an

application.properties or application.yml file.

• You can add administration-related features, normally through JMX. You do this simply by enabling the **spring.application.admin.enabled** property in the **application.properties or application.yml files**.

• Spring Boot allows you to have profiles that will help your application run in different environments.

UNIX OSs: Linux, OS X, and Solaris

There are a lot of tools that can help you install the Spring Boot CLI. If you are using any UNIX environment,

including Linux, OS X, or Solaris, you can use a very good tool named SDKMAN. You can find it at

http://sdkman.io/ . Open a terminal window and execute the following:

**$ curl -s get.sdkman.io | bash**

After it finishes, you can execute the following line to run the sdk command:

**$ source "$HOME/.sdkman/bin/sdkman-init.sh"**

Then make sure that the sdk command is working by executing this line:

**$ sdk version**

SDKMAN 3.2.4

Next, it’s time to install the Spring Boot CLI, which you do by executing this command:

**$ sdk install springboot**

Once the CLI is installed, you can check if everything went okay by executing this request:

**$ spring --version**

Spring CLI v1.3.2.RELEASE

You should get the latest version of Spring Boot; in my case it’s release 1.3.2. Now you are ready to start

using the Spring Boot CLI on a UNIX system.

You can use the same s dk command to install Groovy and Gradle. You can install those two by

executing: $ sdk install groovy and $ sdk install gradle .

**$ ruby -e "$(curl -fsSL** [**https://raw.githubusercontent.com/Homebrew/install/master/install**](https://raw.githubusercontent.com/Homebrew/install/master/install)**)"**

You can then execute the following command to install Spring Boot:

**$ brew tap pivotal/tap**

**$ brew install springboot**

**$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/linuxbrew/go/install)"**

Then execute the same commands from above:

**$ brew tap pivotal/tap**

**$ brew install springboot**

Windows OS

If you are a Windows user or you don’t want to use the previous methods, you can download the ZIP binary

distribution and uncompress it. These are the links of release 1.3.2:

• http://repo.spring.io/release/org/springframework/boot/spring-bootcli/

1.3.2.RELEASE/spring-boot-cli-1.3.2.RELEASE-bin.zip

• http://repo.spring.io/snapshot/org/springframework/boot/spring-boot-cli/

These links are the binary versions, but if you wonder where those links are coming from, you can

find them here: https://docs.spring.io/spring-boot/docs/current/reference/html/gettingstarted-

installing-spring-boot.html#getting-started-manual-cli-installation . You must have

the JAVA\_HOME variable set (pointing to your Java SDK) and the SPRING\_HOME variable pointing to where

you uncompress the binary distribution. Also make sure to set up your PATH variable, which includes the

% SPRING\_HOME%\bin path (or, if you are using UNIX, it’s $SPRING\_HOME/bin) . By setting these variables to the environment, you will have access to the spring.bat or spring scripts.

**$ spring init --build gradle myapp – this will create gradle based build application structure**

the Spring Boot project structure created when you execute the **spring init** command**. If you want to add more features—such as web, JPA, and Maven projects—you can execute the**

**following command:**

**$ spring init -dweb,data-jpa,h2,thymeleaf --build maven myapp --force**

This command will create a Spring Boot Maven project and will include all the necessary dependencies

in the pom.xml file to run a Spring Boot web application. It will include libraries to handle web files (this will

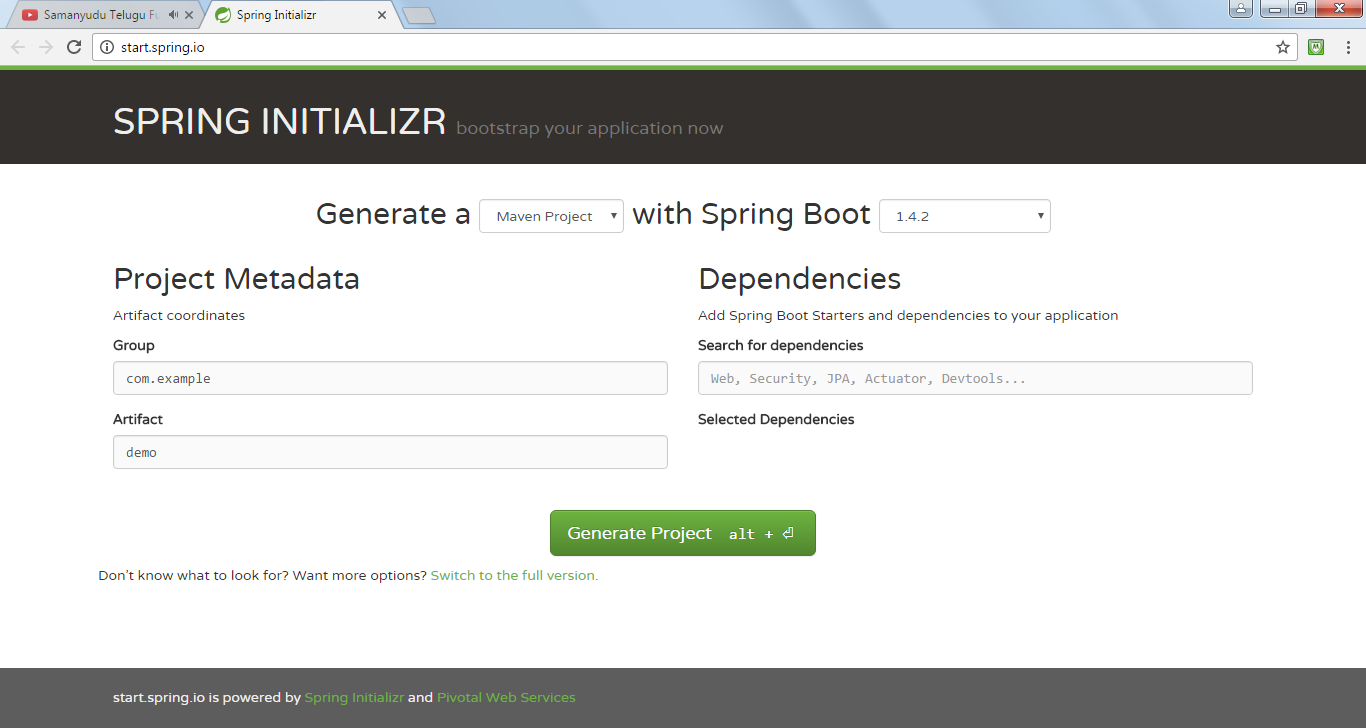
include the embedded Tomcat server), persistence ( data-jpa ), the H2 database engine ( h2 ), and a viewer

engine ( thymeleaf ). **You need to use --force to override the previous myapp directory or you can change**

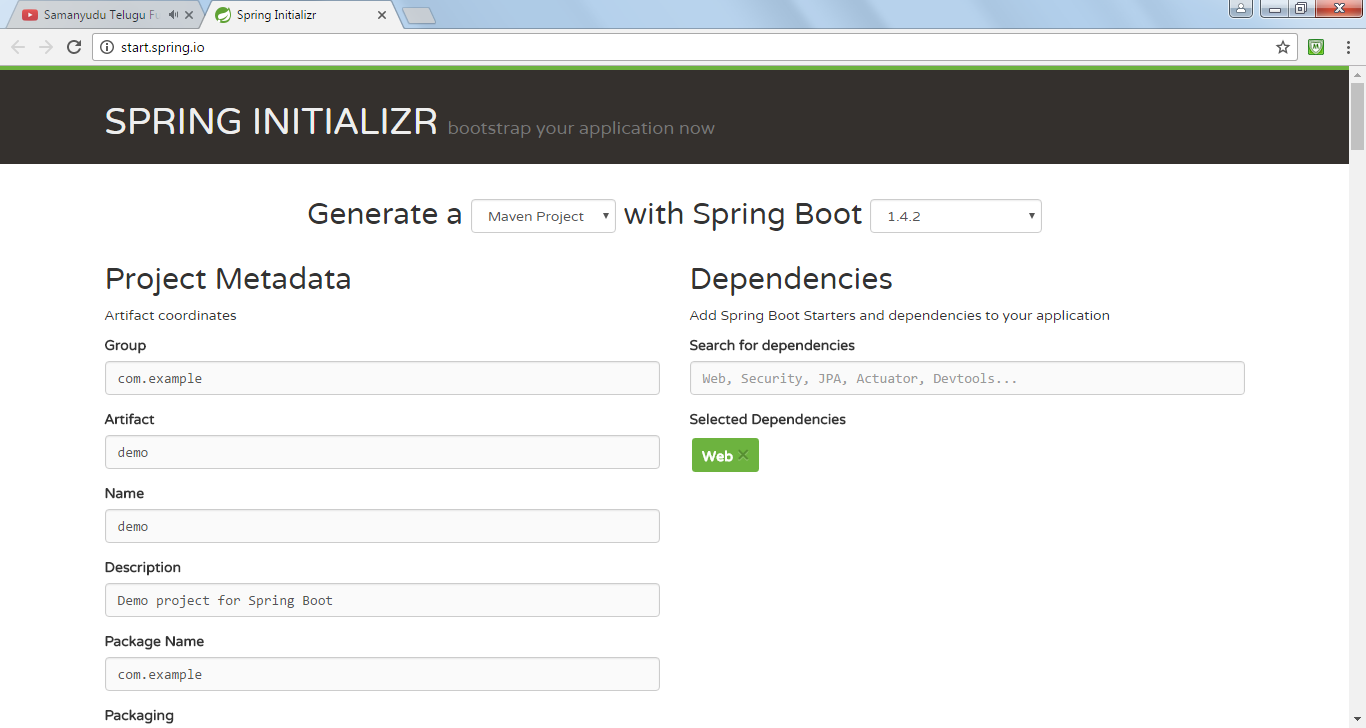
**the name.**

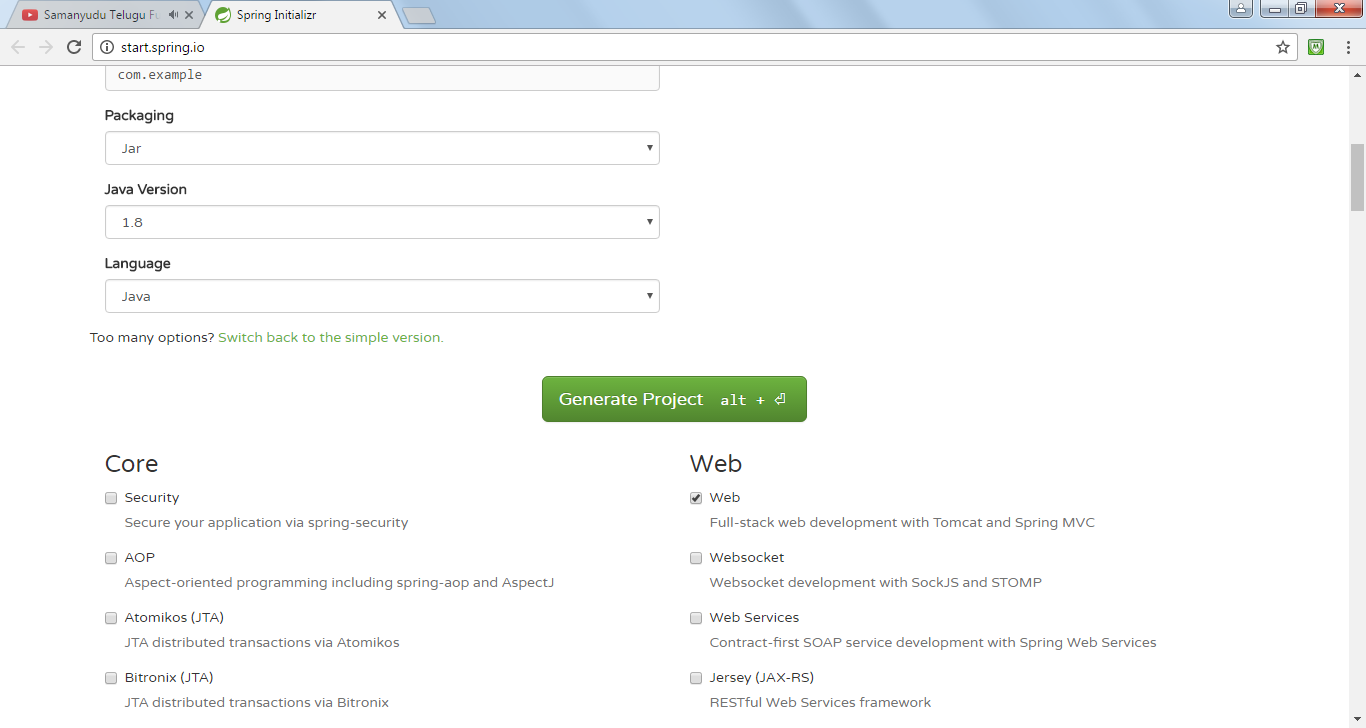
**Spring Initializer :**

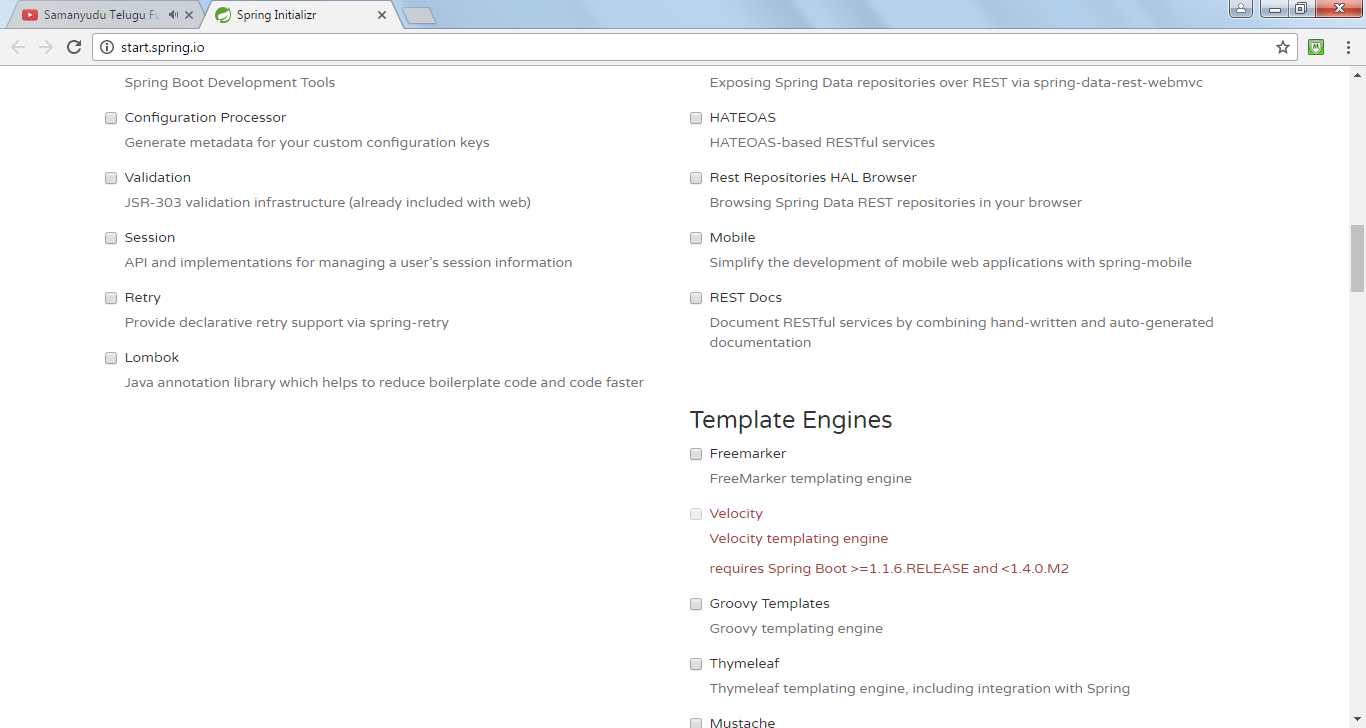
*Simple view of the Spring Initializr ( http://start.spring.io )*

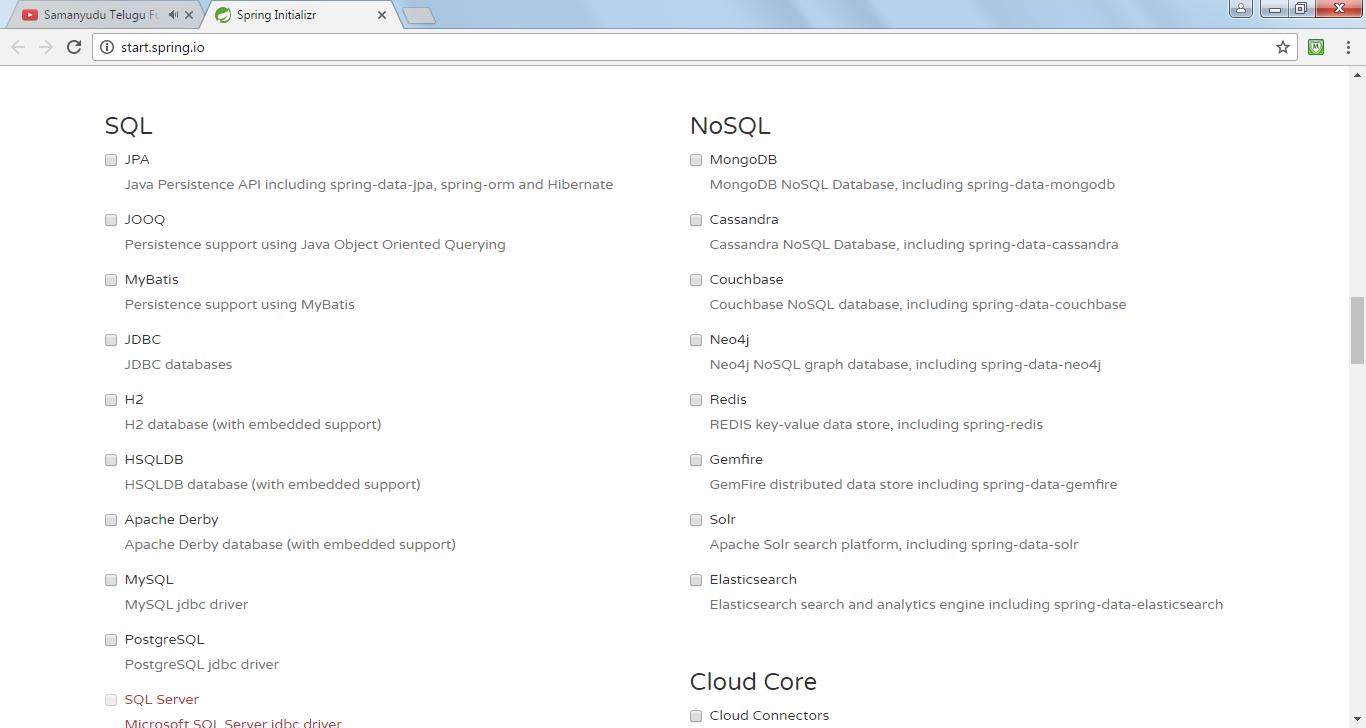
****

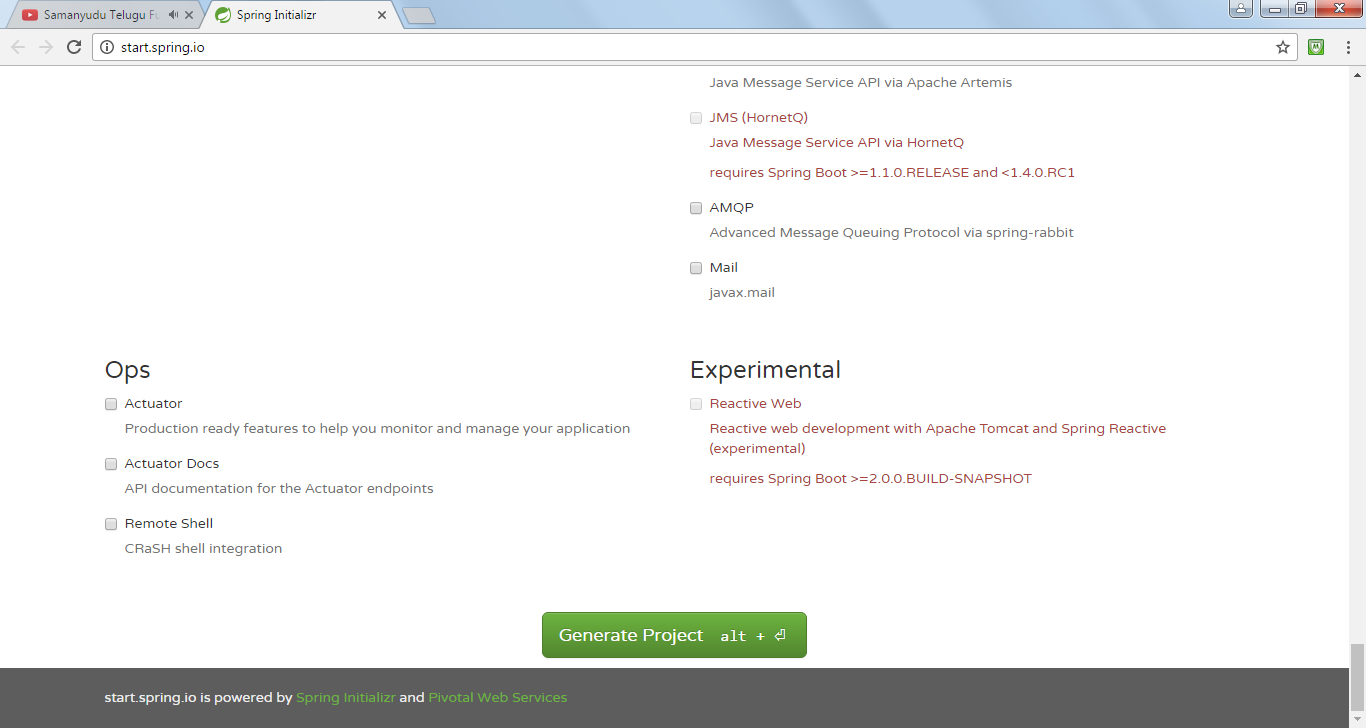
Click on The Switch to the full version











After you select the features you want to use, click the Generate Project button to get the ZIP file that contains your project.

Using the Spring Initializr with UNIX cURL

The Spring Initializr can be accessed using the UNIX cURL command because at the end it is a web service

and it exposes a RESTful API. So, for example, if you wanted to create a simple project that contains just the

minimum files, you could execute the following command:

**$ curl -s https://start.spring.io/starter.zip -o myapp.zip**

**This command will create a myapp.zip** file that contains all the structure for the Spring Boot app. And

by default it contains a Maven project with its pom.xml file and a Maven wrapper.

Spring Boot Using Spring Tool Suite (STS)

If you are already using the Eclipse IDE , you can install the STS as a plugin or download it at

<https://spring.io/tools/sts/all>

**Spring Boot Auto-Configuration, Features :**

@SpringBootApplication annotation a bit. This annotation is equivalent to the @Configuration , @ComponentScan , and @EnableAutoConfiguration annotations.

You can disable a specific auto-configuration by adding the @EnableAutoConfiguration annotation to your class with the exclude parameter.

@SpringBootApplication( exclude={ActiveMQAutoConfiguration.class,DataSourceAutoConfiguration.class} )

@ EnableAutoConfiguration and

@Enable<Technology> Annotations

You will find that the Spring Framework and some of its modules—like Spring Core, Spring Data,

Spring AMQP, and Spring Integration—provide @Enable<Technology> annotations. For example,

@EnableTransactionManagement , @EnableRabbit , and @EnableIntegration are part of the modules

mentioned.

Spring Boot Java project with the spring init command

$ spring init -g=com.apres.spring -a=spring-boot-simple --package=com.apress.spring

-name=spring-boot-simple –x

This command will create a Maven Java project with a groupId=com.apress.spring ,

an artifactId=spring-boot-simple , and a package=com.apress.spring with a project’s

name=spring-boot-simple . It will be created in the current directory ( -x ).

To run it, execute the following command in the same terminal window:

$ ./mvnw spring-boot:run

• SpringApplication . This class provides the bootstrap for the Spring Boot

application that is executed in the main method. You need to pass the class that will

be executed.

public static void main(String[] args) {

SpringApplication app = new SpringApplication(SpringBootSimpleApplication.class);

//add more features here.

app.run(args);

}

Custom Banner

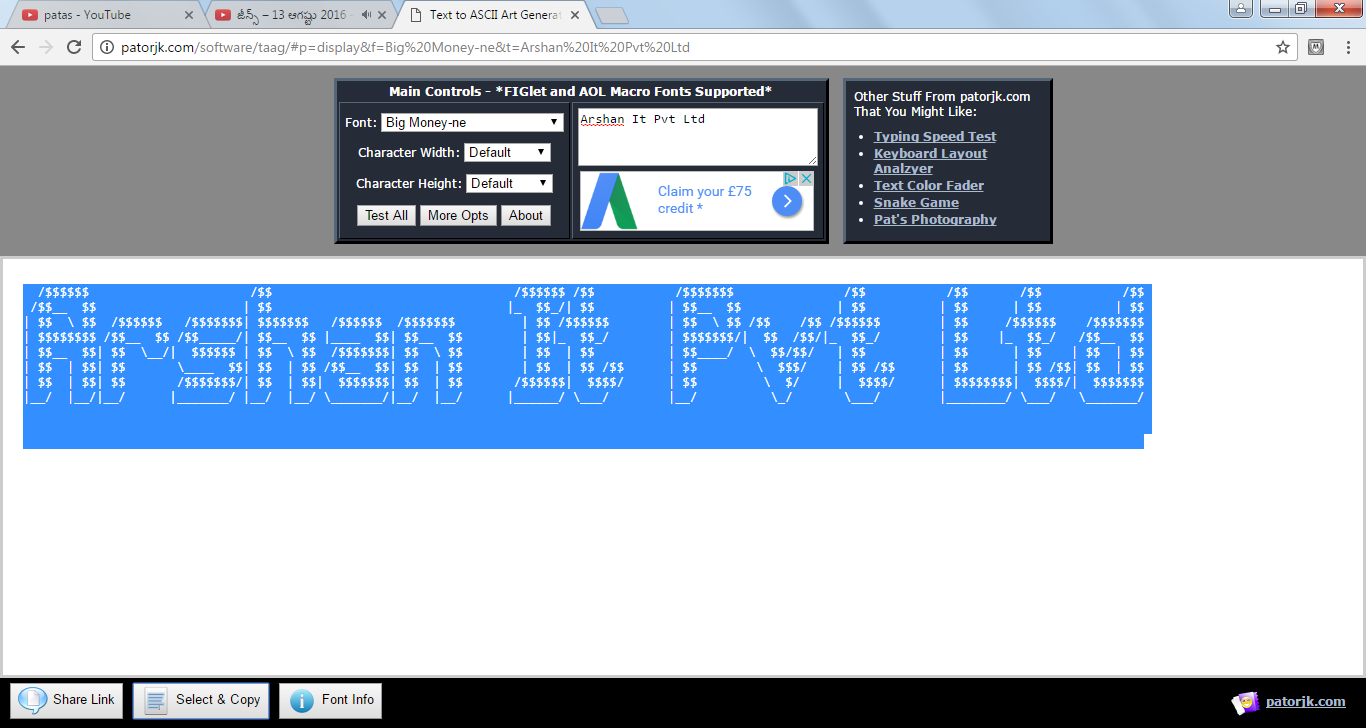
To implement the org.springframework.boot.Banner interface.

You can also create your own ASCII banner and display it. There is a very cool site that creates ASCII art

from text ( http://patorjk.com ).

Below screen shot shows you the http://patorjk.com site. You can click the “Text to ASCII Art Generator” link.

Once you are there, add the text in the text field (or whatever you want).



Select and copy your Ascii art and then create a file named banner.txt in the src/main/resources/ directory.

You can run your application again using this command:

$ ./mvnw spring-boot:run

Create another banner.txt file (or copy the one you have already) in the src/main/resources/META-INF/

directory. Then you can run the application by passing a -D parameter. Execute the following command:

$ ./mvnw spring-boot:run –Dbanner.location=classpath:/META-INF/banner.txt

You can declare this property in the src/main/resources/application.properties file, as follows:

banner.location=classpath:/META-INF/banner.txt

You have several options for the setting up the banner.txt file. You can remove completely the banner.

You can define it in src/main/resources/applications.property like this:

**spring.main.banner-mode=off**

Programmatically you can do like below:

public static void main(String[] args) {

SpringApplication app = new SpringApplication(SpringBootSimpleApplication.class);

**app.setBannerMode(Mode.OFF);**

app.run(args);

}

SpringApplicationBuilder

The SpringApplicationBuilder class provides a fluent API and is a builder for the SpringApplication

and ApplicationContext instances. It also provides hierarchy support.

This is another way of creating Spring boot application:

new SpringApplicationBuilder()

.bannerMode(Banner.Mode.OFF)

.sources(SpringBootSimpleApplication.class)

.run(args);

You can log the info at startup or not; by default, this is set to true:

new SpringApplicationBuilder(SpringBootSimpleApplication.class)

.logStartupInfo(false)

.run(args);

You can activate profiles:

new SpringApplicationBuilder(SpringBootSimpleApplication.class)

.profiles("prod","cloud")

.run(args);

You can attach listeners for some of the ApplicationEvent events:

Logger log = LoggerFactory.getLogger(SpringBootSimpleApplication.class);

new SpringApplicationBuilder(SpringBootSimpleApplication.class)

.listeners(new ApplicationListener<ApplicationEvent>() {

@Override

public void onApplicationEvent(ApplicationEvent event) {

log.info("#### > " + event.getClass().getCanonicalName());

}

})

.run(args);

In addition, you can have these events:

**ApplicationStartedEvent (sent at the start),**

**ApplicationEnvironmentPreparedEvent(sent when the environment is known),**

**ApplicationPreparedEvent (sent after the bean definitions),**

**ApplicationReadyEvent (sent when the application is ready),**

**ApplicationFailedEvent (sent in case of exception during the startup).**

Application Arguments

Spring Boot allows you to get the arguments passed to the application. When you have this:

SpringApplication.run(SpringBootSimpleApplication.class, args );

You can access the args in your beans.

When you execute args.containsOption("enable") , it will expect the argument as --<arg> , it will be expecting --enable . The getNonOptionArgs will take other arguments. To test it, you can execute the following command:

$ ./mvnw spring-boot:run -Drun.arguments="--enable"

You should see the text: ## > you are enabling. Also you can run it like this:

$ ./mvnw spring-boot:run -Drun.arguments="arg1,arg2"

To create a executable JAR, simply execute the following command:

$ ./mvnw package

This command will create an executable JAR, meaning that you can run it like this:

$ java -jar target/spring-boot-simple-0.0.1-SNAPSHOT.jar

You can pass arguments like this:

$ java -jar target/spring-boot-simple-0.0.1-SNAPSHOT.jar --enable arg1 arg2

You should get the same text for the enable arg and a list of arg1 and arg2 .

ApplicationRunner and CommandLineRunner

Spring Boot allows you to execute code before your application starts. Spring Boot has the **ApplicationRunner** and the **CommandLineRunner** interfaces that expose the **run** methods.

Ex: ApllicationRunner run method:

@Override

public void run(ApplicationArguments args) throws Exception {

log.info("## > ApplicationRunner Implementation...");

log.info("Accessing the Info bean: " + info);

args.getNonOptionArgs().forEach(file -> log.info(file));

}

CommandLineRunner run method:

@ Override

public void run(String... args) throws Exception {

log.info("## > CommandLineRunner Implementation...");

log.info("Accessing the Info bean: " + info);

for(String arg:args)

log.info(arg);

}

@Bean

CommandLineRunner myMethod(){

return args -> {

log.info("## > CommandLineRunner Implementation...");

log.info("Accessing the Info bean: " + info);

for(String arg:args)

log.info(arg);

};

}

The above method that’s annotated with the @Bean annotation returning a CommandLineRunner implementation. This example uses the Java 8 syntax (lambda) to do the return. You can add as many

Methods that return a CommandLineRunner as you want. If you want to execute these in certain order, you can

use the @Order annotation.

Application Configuration

With Spring you can use XML and the <context:property-placeholder/> tag, or you can use the

@PropertySource annotation to declare your properties.

Spring Boot has different options for saving your application configuration:

• You can use a file named application.properties , which should be located in the

root classpath of your application (there are more places where you can add this file

that you’ll learn about later).

• You can use a YAML notation file named application.yml that also needs to be

located in the root classpath (there are more places where you can add this file that

you’ll learn about later).

• You can use environment variables. This is becoming the default practices for cloud

scenarios.

• You can use command-line arguments.

You can get the complete list of the common application properties

here: https://docs.spring.io/spring-boot/docs/current/reference/html/common-applicationproperties.

html .

One of the best features from Spring (and of course from Spring Boot as well) is that you can access the properties values by using the @Value annotation (with the name of the property) or from the org.springframework.core.env.Environment interface, which extends from the org.springframework.

core.env.PropertyResolver interface.

For example, if you have a src/main/resources/application.

properties file with the following content:

data.server=remoteserver:3030

You can access the d ata.server property in y our application by using the @ Value annotation, as shown

in the following snippet:

//...

@Service

public class MyService {

@Value("${data.server}")

private String server;

//...

}

If you don’t want to use the application.properties , you can inject the properties via the command line:

$ java -jar target/myapp.jar --data.server=remoteserver:3030

If you don’t like the application.properties file or you hate the YAML syntax, you can also use a specialized environment variable named SPRING\_APPLICATION\_JSON to expose the same properties and its values. For example:

$ SPRING\_APPLICATION\_JSON='{ "data":{"server":"remoteserver:3030"}}' java -jar target/myapp.jar

Configuration Properties Examples

Spring Boot uses an order if you want to override your application configuration properties:

• Command-line arguments

• SPRING\_APPLICATION\_JSON

• JNDI (java:comp/env)

• System.getProperties()

• OS environment variables

• RandomValuePropertySource (random.\*)

• Profile-specific ( application-{profile}.jar ) outside of the package JAR

• Profile-specific ( application-{profile}.jar ) inside of the package JAR

• Application properties ( application.properties ) outside of the package JAR

• Application properties ( application.properties ) inside of the package JAR

• @PropertySource

• SpringApplication.setDefaultProperties

Changing Location and Name

Spring Boot has an order to find the application.properties or YAML file. It will look in:

• The /config subdirectory located in the current directory

• The current directory

• A classpath /config package

• The classpath root

Spring Boot allows you to change the name and location of the properties file. So for example, imagine

that you will use the /config subdirectory and the name of the properties file is now mycfg.properties

(its content is server.ip=127.0.0.1 ). Then you can run the app with the following command:

$./mvnw spring-boot:run -Dspring.config.name=mycfg

or

$ ./mvnw package -DskipTests=true

$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar --spring.config.name=mycfg

or

$ SPRING\_CONFIG\_NAME=mycfg java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar

$ ./mvnw spring-boot:run -Dspring.config.name=mycfg -Dspring.config.location=file:app/

or

$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar --spring.config.location=file:app/

--spring.config.name=mycfg

You can add the mycfg.properties file to the src/main/resources/META-INF/conf (you can create it)

and execute this:

$ mkdir -p src/main/resources/META-INF/conf

$ cp config/mycfg.properties src/main/resources/META-INF/conf/

$ ./mvnw clean spring-boot:run -Dspring.config.name=mycfg -Dspring.config.

location=classpath:META-INF/conf/

Spring Boot also has an order to search for the properties file:

• classpath

• classpath:/config

• file:

• file:config/

Profile Based

You simply have to specify the active profile with the @ActiveProfiles annotation (when you are testing classes) or get the current environment and use the setActiveProfiles method. You can also use the SPRING\_PROFILES\_ACTIVE environment variable or the spring.profiles.active property.

Create a file application-qa.properties, application-prod.properties in config/ subdirectory.

you can run your example with the following:

$ ./mvnw clean spring-boot:run -Dspring.profiles.active=prod

Custom Properties Prefix:

Spring Boot allows you to write and use your own custom property prefix for your properties. The only thing

you need to do is annotate with the @ConfigurationProperties annotation a Java class that will have setters

and getters as their properties.

If you are using the STS IDE, I recommend including a dependency in your pom.xml . This dependency

will create a code insight and it will trigger the editor’s code completion for the properties. So add the next

dependency in your pom.xml :

<dependency>

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

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

**Spring Boot CLI**

## 1. Prerequisites

* Spring Boot CLI requires Java 6 or above in order to run.
* JAVA\_HOME environment variable should be set to a JDK

## 2. Install Spring Boot Command Line Interface on Windows

### 2.1. Download Spring Boot Command Line Interface

You can visit Spring Boot repository to download the Spring Boot CLI version that you desire by follow below link:

<http://repo.spring.io/release/org/springframework/boot/spring-boot-cli>

### 2.2. Set SPRING\_HOME environment variable.

**2.2.1.** In Windows 7 right click **My Computer** and select **Properties** > **Advanced**.  
In Windows 8, Windows 10,  go to **Control Panel** > **System** > **Advanced System Settings**.

**2.2.2.**Under **System Variables**, click **New**

**2.2.3.** In the **Variable Name** field, enter **SPRING\_HOME**

**2.2.4.** In the **Variable Value** field, enter your Spring Boot CLI path, for ex:

**C:\Program Files\SpringBootCLI\spring-1.4.0.RELEASE**

### 2.3. Add SPRING\_HOME/bin to PATH environment variable.

If you are using a Mac/Linux, you can use homebrew

( http://brew.sh/ ) with the following command:

**$ brew tap pivotal/tap**

**$ brew install springboot**

If you are using Linux, you can use the sdkman tool ( http://sdkman.io/ ) and install it with the

following command:

**$ sdk install springboot**

**Some useful commands are :**

**Spring run**

**Spring test**

The grab Command

T he g rab command will download all the Spring Groovy scripts and Java dependencies to the . /repository directory. Its syntax is the following:

**spring grab [options] files [--] [args]**

**The jar Command**

The jar command will create a self- contained executable JAR file from a Groovy or Java script. Its syntax is

the following:

**spring jar [options] <jar-name> <files>**

**The war Command**

This is very similar to the previous command. The war command will create a self-contained executable

WAR file from a Groovy or Java script. Its syntax is the following:

**spring war [options] <war-name> <files>**

The install Command

T he install command i s very similar to the grab command; the only difference is that you need to specify the library you want to install

**spring install [options] <coordinates>**

The uninstall Command

The uninstall command will uninstall the dependencies from the lib directory. Its syntax is the following:

**spring uninstall [options] <coordinates>**

**The init Command**

The init command will help you initialize a new project by using the Spring Initializr ( http://start.

spring.io/ ).

**spring init [options] [location]**

**spring help**

You can run some processes or jobs after SpringApplication.run is called,by implementing the org.springframework.boot.CommandLineRunner interface and implementing the run(String... args) method. This is useful when you want to execute jobs or services, such as send a notification about the application or execute a SQL statement to update some rows before your application runs. This is not a web application; it is a standalone app.

This code shows you how to use the CommandLineRunner interface as a bean by marking the method

with @Bean annotation. Or, if you are using Java 8, you can use the lambdas feature like this:

@Bean

public CommandLineRunner runner(Repository repo){

return args -> {

//Run some process here

};

}

Maybe you are wondering what you need to do if you need to run some code even before the

CommandLineRunner . You can do this by returning an InitializingBean interface.

@Bean

InitializingBean saveData(Repository repo){

return () -> {

//Do some DB inserts

};

}

Using Spring Technologies in Spring Boot

The only thing you need to know now is that there is an annotation called @Enable<Technology> for

each of these technologies;

*Spring Technologies Used in Spring Boot*

**Annotation Description**

@EnableJms Messaging with JMS technology

@EnableCaching Caching abstraction

@EnableRabbit Messaging for the AMQP with RabbitMQ

@EnableBatchProcessing Spring batch

@EnableWebSecurity Spring security

@EnableRedisHttpSession Spring session

@EnableJpaRepositories Spring data

@EnableIntegration Spring integration

Testing Spring Boot:

**<dependency>**

**<groupId>org.springframework.boot</groupId>**

**<artifactId>spring-boot-starter-test</artifactId>**

**<scope>test</scope>**

**</dependency>**

Every time you create a project via Spring Initializr, you will get the spring-boot-starter-test pom by default. This will include spring-test , junit , hamcrest , objenesis , and mockito JARs. you can use Spock or another framework together with Spring test.

@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(classes = SprintBootApplication.class)

The SpringJUnit4ClassRunner supports the following annotations:

• @Test(expected=...)

• @Test(timeout=...)

• @Timed

• @Repeat

• @Ignore

• @ProfileValueSourceConfiguration

• @IfProfileValue

• You can also use the SpringClassRule and SpringMethodRule classes, both a custom JUnit TestRule interface that supports class-level features of the TestContext Framework. They are used together with the @ClassRule and @Rule annotations.

• @SpringApplicationConfiguration(classes = SprintBootApplication.class) .This is a class-level annotation that knows how to load and configure an ApplicationContext , which means that you can have direct access to all the Spring container classes by just using the @Autowired annotation. In this case, the main SpringBootApplication class wires everything up.

• @Test . This is a JUnit test annotation that will execute the method when the tests start. You can have one or more methods. If you have several methods with thisannotation, it won’t execute them in order. For that you need to add the @FixMethodOrder(MethodSorters.NAME\_ASCENDING) annotation to the class.

If the project is a web app, the tests include a new annotation called @WebAppConfiguration . It’s a class-level annotation that loads the org.springframework.web.context.WebApplicationContext implementation, which will ensure that all your files and beans related to the web app are accessible.

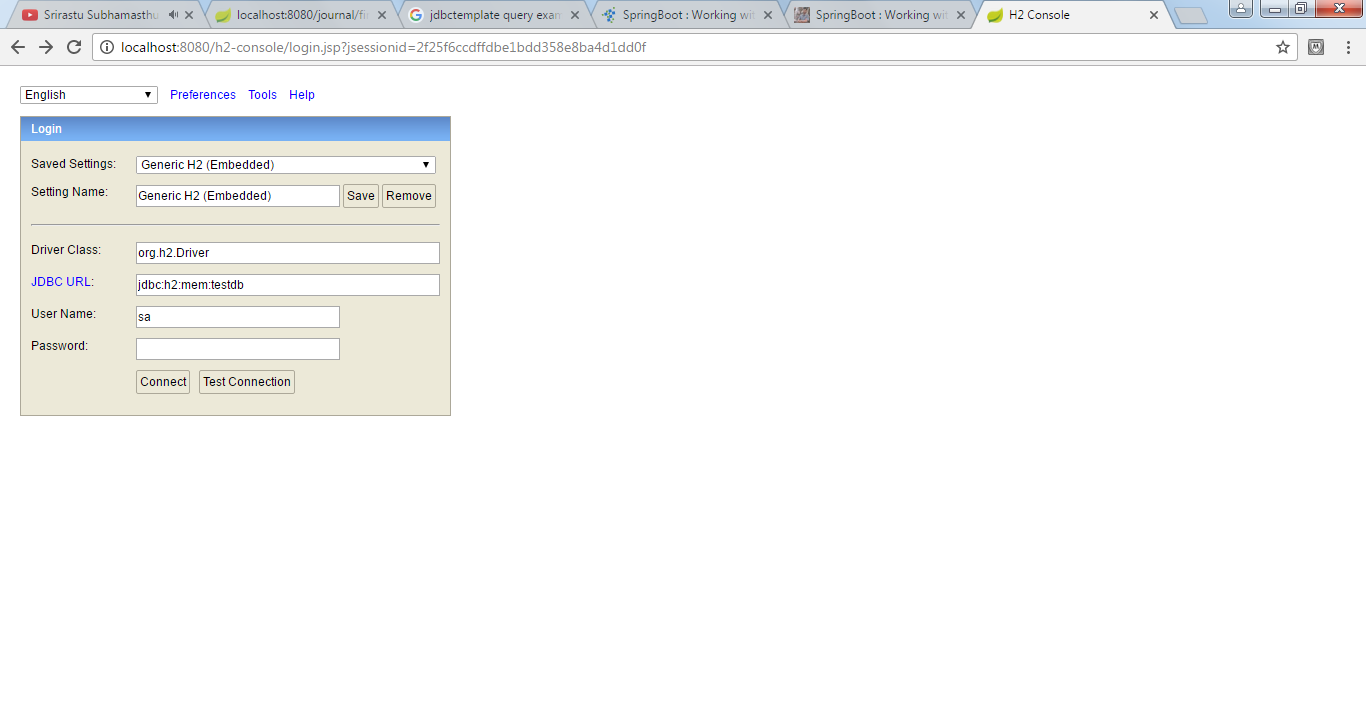
Data Access Object implementation:

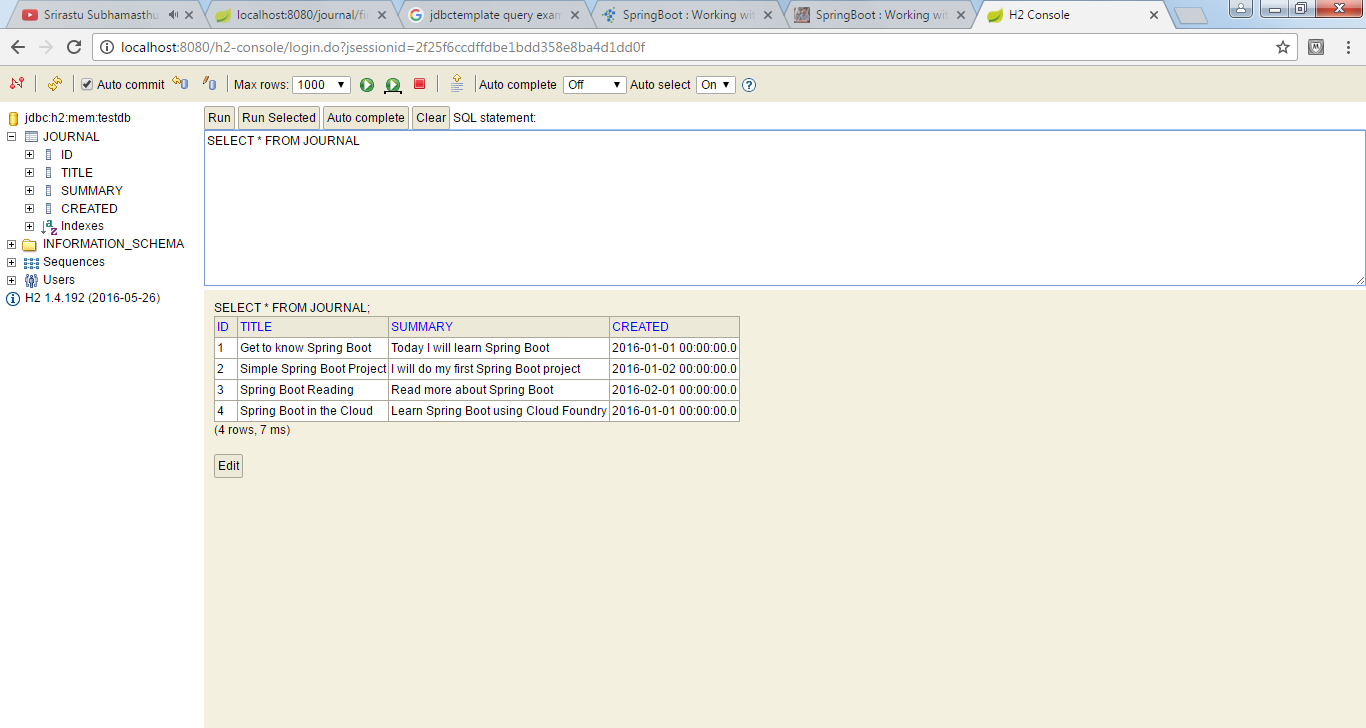
src/main/resources/application.properties.

spring.h2.console.enabled=true

This property will enable the H2 web console.

<http://localhost:8080/h2-console>





Data Access Using JPA with Spring Boot

Another Spring Boot feature using the spring-data enables you to use the schema.sql and data.sql

files (in the root of the classpath) to create the database and insert data. This feature is useful when you have

a dump of data and must initialize the database. So instead of using a service to insert the data, you can write

data.sql and remove the insertData call from your service.

If you want to see the SQL statements that the JPA/Hibernate engine is executing, you can use the

following property in the src/main/resources/application.properties file: **spring.jpa.show-sql=true** .

**NoSQL Databases**

NoSQL databases are another way to persist data, but in different way from the tabular relationships of the

relational databases. There is already a classification system for these emergent NoSQL databases. You can

find it based on its data model:

• Column (Cassandra, HBase, etc.)

• Document (CouchDB, MongoDB, etc.)

• Key-Value (Redis, Riak, etc.)

• Graph (Neo4J, Virtuoso, etc.)

• Multi-Model (OrientDB, ArangoDB, etc.)

If you are using Mac/Linux with the brew command ( http://brew.sh/ ), execute the following

command:

$ brew install mongodb

You can run it with this command:

$ mongod

Or you can install MongoDB by downloading it from the web site at https://www.mongodb.org/

downloads#production and following the instructions.

Include below dependency in pom.xml or use Spring initilizer for creating Spring-boot-starter-mongodb app.

<dependency>

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

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

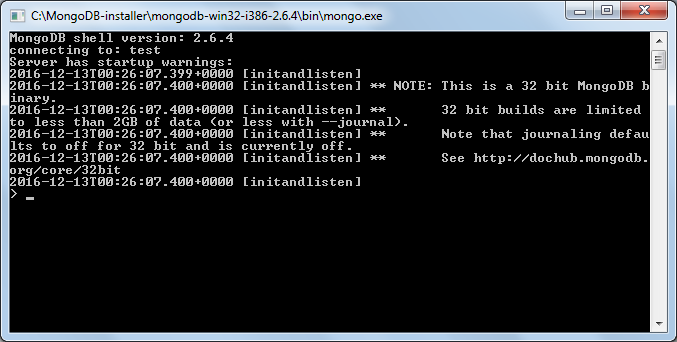
2. Define your domain model for persistence

3. Define your interface extending MongoRepository<urmodel,String>

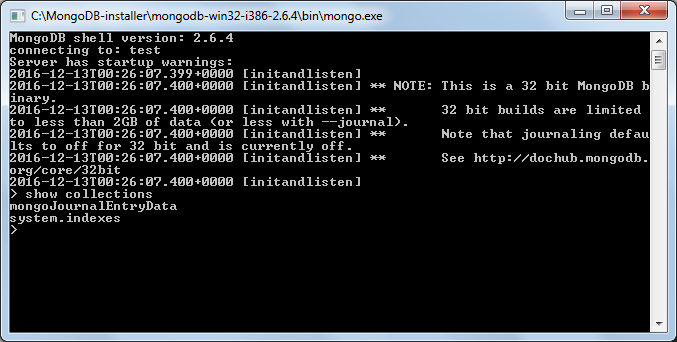
4. Define your CRUD operation in Spring main boot application as a bean commandlinerunner.

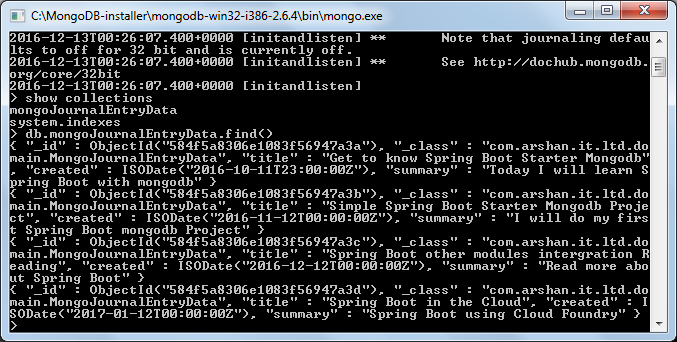
5. Run the spring boot after clean install

6. Check the result

****

**Show collections**

****

****

**>db.mongoJournalEntryData.find() :** query to get all the data.

Spring Boot allows you to define the name of your database if you don’t want to use the default one.

You only need to add the following property to **the src/main/resources/application.properties** file:

**spring.data.mongodb.database=springBootMongoJournal**

Then the MongoRepository will create the database using the my journal name and will create the

**mongoJournalEntryData** collection as well.

**> show databases**

**> use springBootMongoJournal**

**> show collections**

**>db.mongoJournalEntryData.find()**

****

**------------------------------Web Development with Spring Boot --------------------------**

the Spring Framework has the spring-web, spring-webmvc , spring-websocket , and spring-webmvc-portlet modules.

**Security with Spring Boot**

Spring security is centered around **AuthenticationProvider** and specialized **UserDetailsService** ; it also provides integration with identity provider systems, such as LDAP, Active Directory, Kerberos, PAM, AOuth, and so on.

The starter pom you need is spring-boot-starter-security

<dependency>

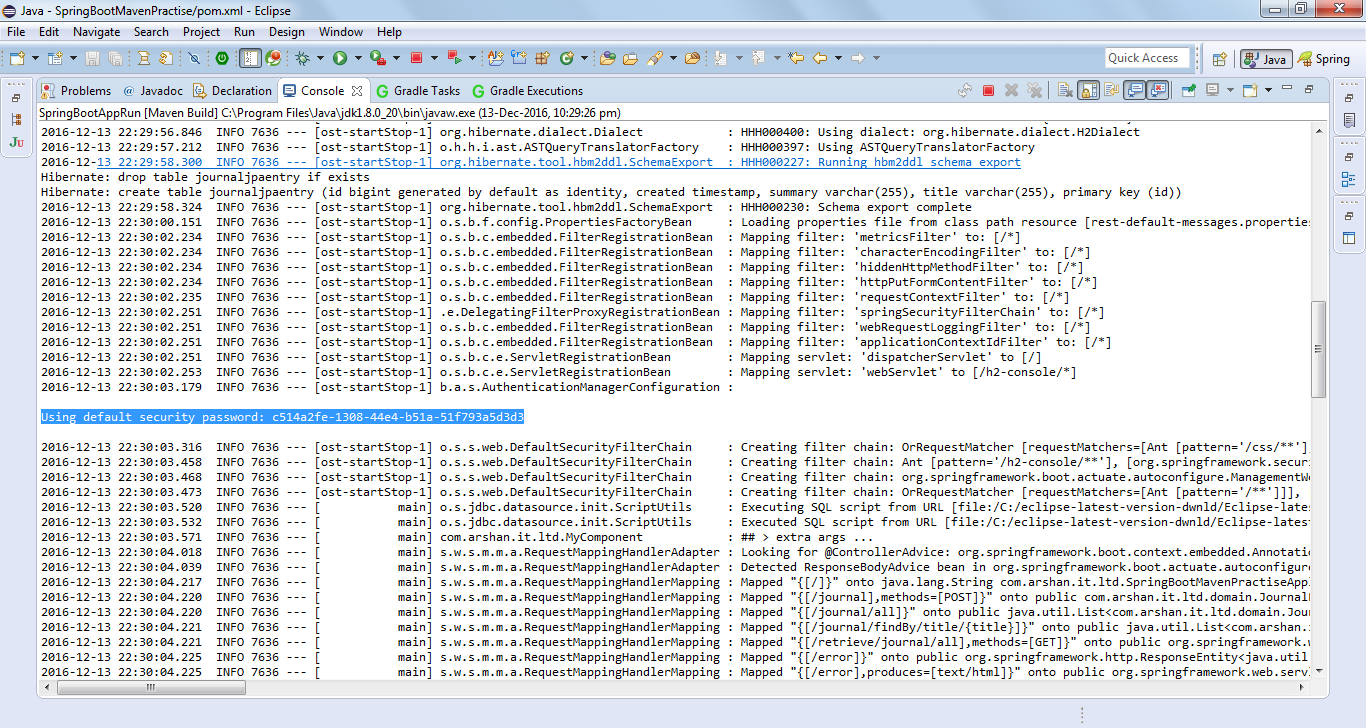
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

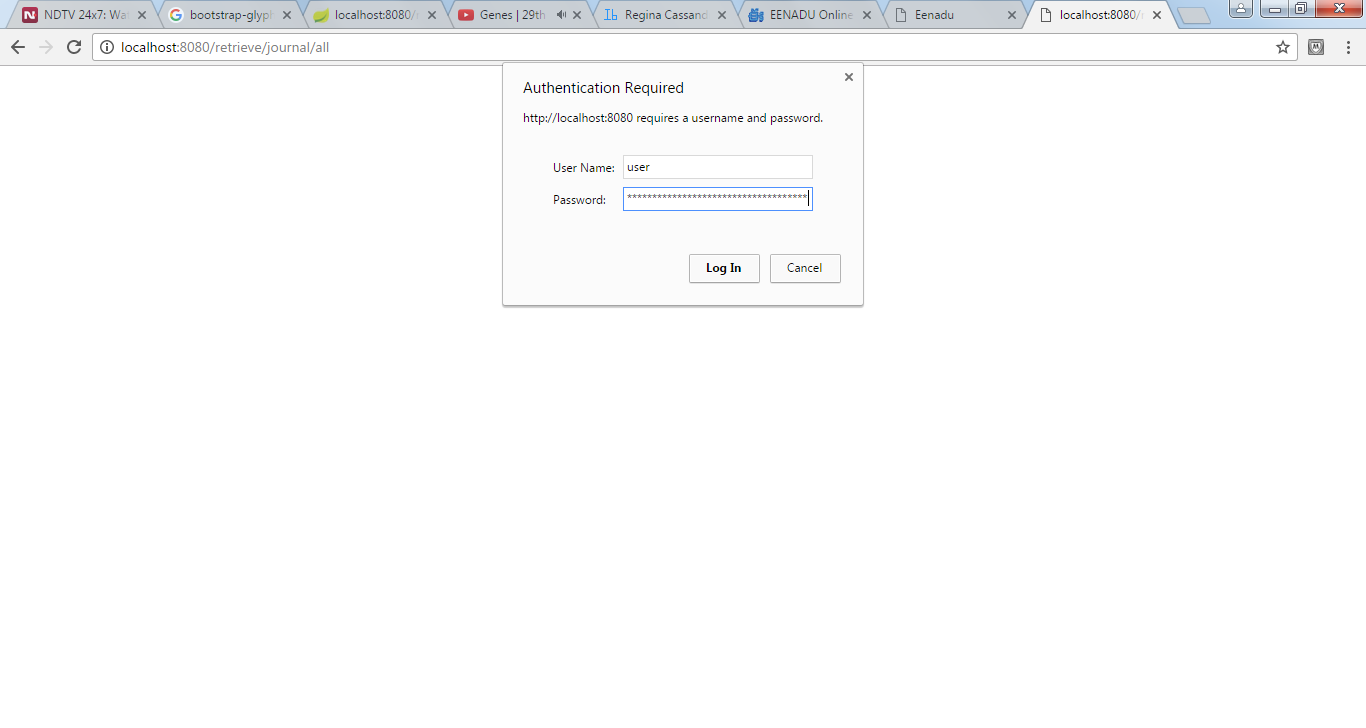
</dependency>

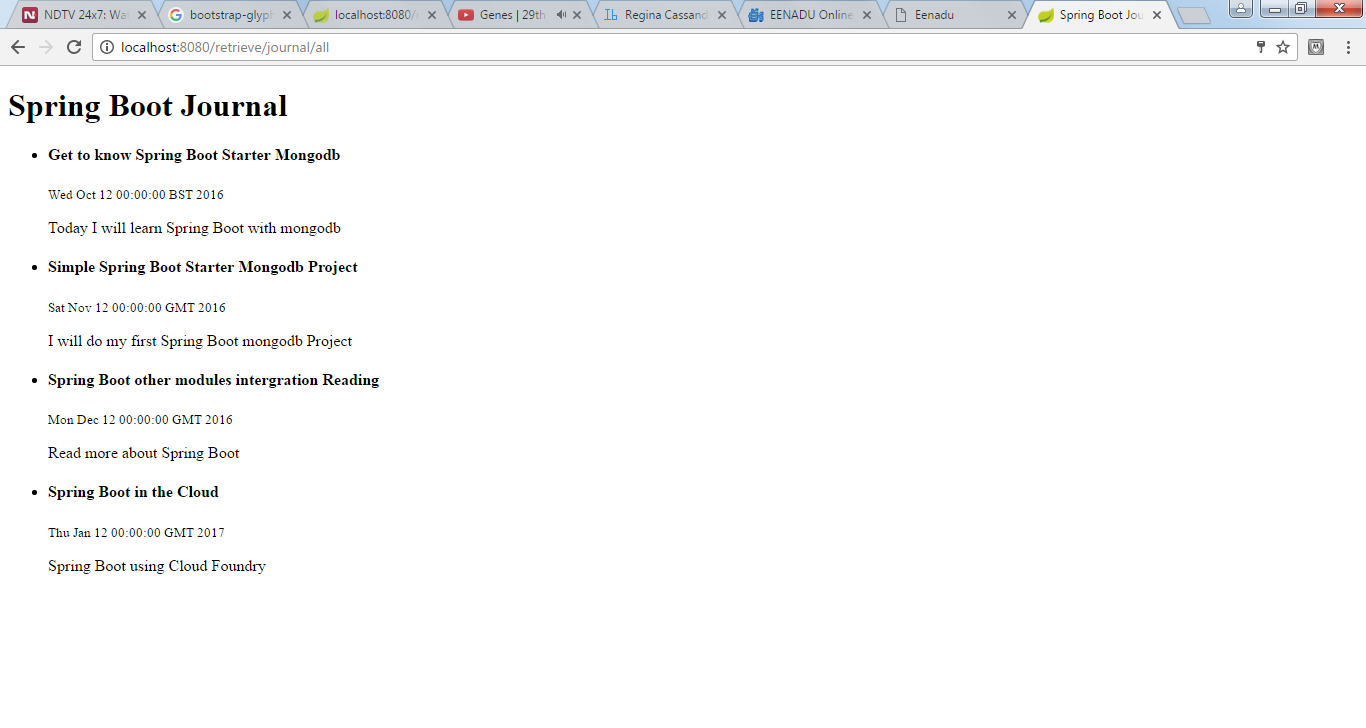
By default, the **AuthenticationManager** interface implementation has a single username, called user. So, in the User Name box, you enter the value user. The password is the GUID that you saw on the logs—a random password.

After running the spring boot, you should be able to see a new line about the **AuthenticationManagerConfiguration** class . You will find below default password in the console as below (Note : every time of Spring boot run you will get the new password)



In the console, you should see the text: "Using default security password: xxx-xxxx-xxx ..."



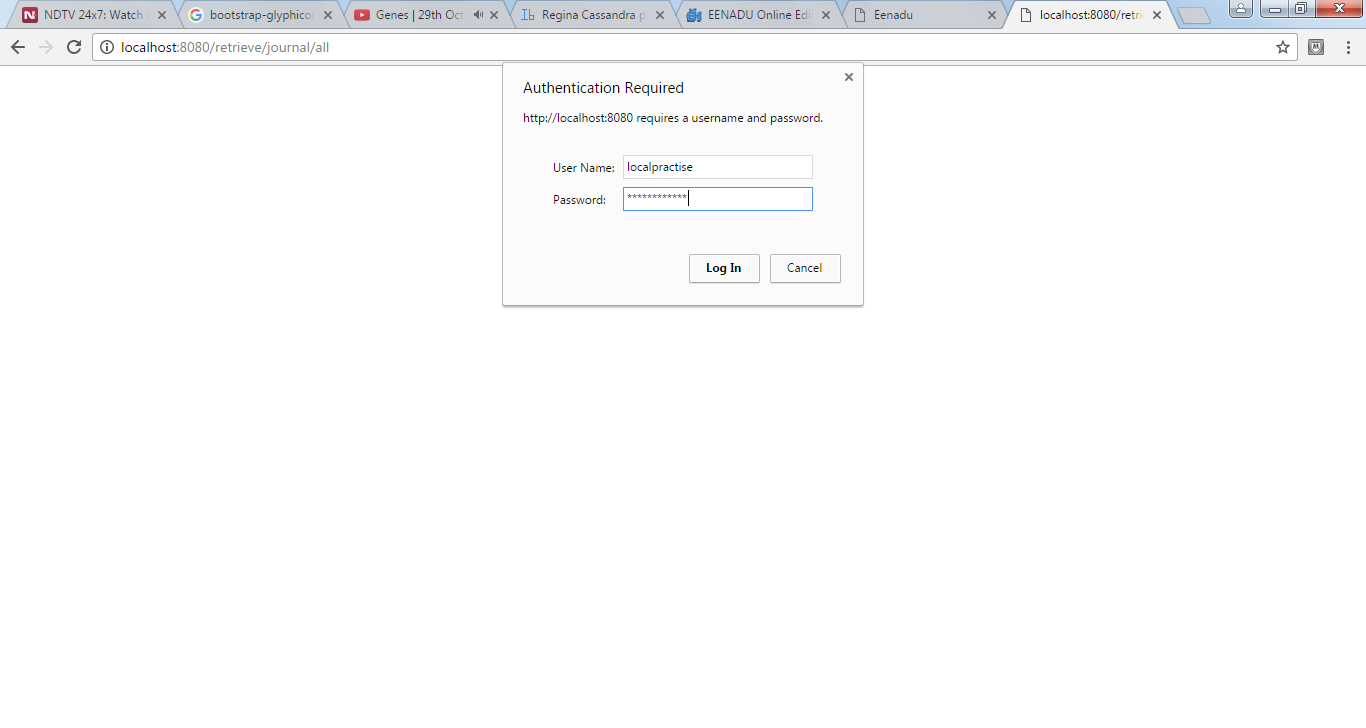


Security Using the application.properties File

# Security

security.user.name = localpractise

security.user.password = capgemini123



In-Memory Security:

@Configuration

@EnableGlobalAuthentication

**public** **class** InMemorySecurityGlobalAuthenticationConfig {

@Autowired

**public** **void** configGlobalAuthentication(AuthenticationManagerBuilder authManagerBuilder) **throws** Exception {

authManagerBuilder.inMemoryAuthentication().withUser("user").password("user123").roles("USER").and()

.withUser("localpractise").password("capgemini123").roles("USER","LOCALPRACTISE")

.and().withUser("admin").password("admin123").roles("USER", "LOCALPRACTISE","ADMIN");

}

}

• **@Configuration** . This annotation tells the Spring Boot to use it as part of the configuration; it’s similar to using XML files.

**• @EnableGlobalAuthentication** . This annotation marks the class and configures all the necessary beans to activate the security on the application; it signals that the annotated class can be used to configure a global instance of the AuthenticationManagerBuilder .

**• @Autowired/ configGlobalAuthentication(AuthenticationManagerBuilder** authManagerBuilder**)** .This method is called to auto-wire the **AuthenticationManagerBuilder** . The **AuthenticationManagerBuilder** allows you to easily build your authentication by adding **UserDetailsService** and the authentication providers. You are going to learn more about the options in the following sections. In this case, it will use in-memory because it’s calling the **inMemoryAuthentication** method and setting up two users with their passwords and roles.

Security Using a Database

Remember that you have the InMemorySecurityConfiguration class, so this means that only one can

be used, not both. So you can leave it and the JdbcSecurityConfiguration will take precedence and all

the users will be in the MySQL database. Another option is that you can comment out the main annotation (@Configuration and @EnableGlobalAuthentication) and it will be the same. The best solution is to use profiles, by using the @Profile annotation and activating the profiles at run time with -Dspring.active.profiles=memory or whatever name you give to the profile.

Securing Resources:

Sometimes you will require to secure just some parts of your application.

@Configuration

@EnableGlobalAuthentication

**public** **class** ResourceSecurityConfiguration **extends** WebSecurityConfigurerAdapter{

@Override

**protected** **void** configure(HttpSecurity http) **throws** Exception {

**super**.configure(http);

http.authorizeRequests()

.antMatchers("/").permitAll()

.antMatchers("/retrieve/\*\*").authenticated()

.and()

.httpBasic();

}

the ResourceSecurityConfiguration class . Let’s review it:

**• @Configuration** . This annotation is picked up by Spring as part of the context

configuration. Here is where you declare beans or in this case configure part of the

security.

**• WebSecurityConfigurerAdapter** . There are different ways to configure the resources of

your web application and extending from the abstract **WebSecurityConfigurerAdapter**

class is one of them. One of the common patterns is to override the

**configure(HttpSecurity) and configure(AuthenticationManagerBuilder)** methods,

but because you have the **init(AuthenticationManagerBuilder)** method overridden

from the **GlobalAuthenticationConfigurerAdapter** of **the JdbcSecurityConfiguration**

class, it’s not necessary to do it here. That’s why the only method you need to override

is the one with the HttpSecurity instance as a parameter.

• **configure(HttpSecurity) .** This method is overridden from the abstract class

**WebSecurityConfigurerAdapter** , and here is where you specify which resources

to secure. In this case, the HttpSecurity instance class allows you to configure

web-based security for specific HTTP requests. By default it will be applied to all

requests, but you can restrict it by using its fluent API. In the example, you get into

the root ( http://localhost:8080 ) of your web app with the **.antMatchers("/").**

**permitAll()** call and restrict the endpoint /api with .antMatchers("/api/\*\*").

**authenticated()** call by making this restricting as HttpBasicConfigurer security .

Spring Boot with OAuth2

OAuth2 is an open standard, and it’s used by companies like Pivotal, Google, Amazon, Facebook, Twitter,

and much more. These companies provide access to services by providing access tokens that are based on

credentials (client IDs and secret keys).



**Messaging with Spring Boot**

HornetQ for implementing the JMS (Java Message Service), RabbitMQ for implementing AMQP (Advanced Message Queuing Protocol),Redis for Pub/Sub, and WebSockets for implementing STOMP (Simple or Streaming Text Oriented Message Protocol) with Spring Boot.

There is always a sender and one or more receivers. Messaging can be synchronous and asynchronous, pub-sub and peer-to-peer, RPC and enterprise-based, Message Broker, ESB (Enterprise Service Bus), MOM (Message Oriented Middleware), etc.

**JMS with Spring Boot**

you can use Spring integration,

Google Protobuffers, Apache Thrift, and another technologies to integrate JMS, but it’s still a lot of work,

because you need to know and maintain code from all these technologies.

The Spring Boot team has a HornetQ starter pom available, so that’s the one you are going to use. HornetQ is an open source asynchronous messaging project from JBoss.

<dependency>

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

<artifactId>spring-boot-starter-hornetq</artifactId>

</dependency>

<dependency>

<groupId>org.hornetq</groupId>

<artifactId>hornetq-jms-server</artifactId>

</dependency>

The configuration will take place in the application.properties file.

**src/main/resources/application. Properties**

spring.hornetq.mode=embedded

spring.hornetq.embedded.enabled=true

spring.hornetq.embedded.queues=springbootQueue,pivotalQueue

myqueue=springbootQueue

create producer

**public** **class** Producer {

**private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(Producer.**class**);

**private** JmsTemplate jmsTemplate;

**public** Producer(JmsTemplate jmsTemplate) {

**this**.jmsTemplate = jmsTemplate;

}

**public** **void** sendTo(String queue, String message) {

**this**.jmsTemplate.convertAndSend(queue, message);

***log***.info("Producer> Message Sent");

}

}

• sendTo(queue, message) . This method has two parameters—the name of

queue (destination) and the message, both as type String . This method uses the

jmsTemplate to use the convertAndSend method call to send the message and

pass the name of the queue and the actual message. The convertAndSend method

will try to use the best available message converter, and by default it will use the

SimpleMessageConverter class. The SimpleMessageConverter will identify if the

message is a String , Map , byte[] array, or Serializable object.

create consumer

**public** **class** Consumer **implements** MessageListener {

**private** Logger log = LoggerFactory.*getLogger*(Consumer.**class**);

@Override

**public** **void** onMessage(Message message) {

**try** {

log.info("Consumer> " + message.getBody(Object.**class**));

} **catch** (JMSException ex) {

ex.printStackTrace();

}

}

}

@Configuration

**public** **class** MessagingConfig {

@Autowired

**private** ConnectionFactory connectionFactory;

@Value("${myqueue}")

**private** String myQueue;

@Bean

**public** DefaultMessageListenerContainer messageListener()

{

DefaultMessageListenerContainer defaultMessageListenerContainer = **new** DefaultMessageListenerContainer();

defaultMessageListenerContainer.setConnectionFactory(**this**.connectionFactory);

defaultMessageListenerContainer.setDestinationName(myQueue);

defaultMessageListenerContainer.setMessageListener(**new** Consumer());

**return** defaultMessageListenerContainer;

}

}

Write below code in Spring boot main application

@Value("${myqueue}")

String myQueue;

@Bean

CommandLineRunner sendMessage(JmsTemplate jmsTemplate) {

**return** args -> {

Producer producer = **new** Producer(jmsTemplate);

producer.sendTo(myQueue, "JMS with Spring Boot Rocks!");

};

}

* Spring-boot:run

After running the program you should have the logs from the consumer and producer, something similar to this:

2016-12-16 21:56:39.874 INFO 4264 --- [ssageListener-1] com.arshan.it.ltd.messageing.Consumer : Consumer> JMS with Spring Boot Rocks!

2016-12-16 21:56:39.979 INFO 4264 --- [ main] com.arshan.it.ltd.messageing.Producer : Producer> Message Sent

…………………..

A Simpler JMS Consumer

@JmsListener(destination) . This annotation will create a consumer listener and

the message will be handled by the method. You only need to pass the destination parameter (the name of

the queue) and that’s it. Spring will take care of the rest.

@JmsListener(destination="${myqueue}")

@SendTo("${myqueuetwo}")

**public** String simpleConsumerImpl(String message)

{

***log***.info("Simpler Consumer defined in spring boot > " + message);

**return** message + " and return Spring Messaging too!";

}

@JmsListener(destination = "${myqueuetwo}")

**public** **void** anotherSimplerConsumer(String message) {

***log***.info("Another Simpler Consumer> " + message);

}

2016-12-16 23:52:13.961 INFO 3776 --- [ main] a.i.l.SpringBootMavenPractiseApplication : Sending message for simple Consumer > ...

2016-12-16 23:52:13.978 INFO 3776 --- [enerContainer-1] a.i.l.SpringBootMavenPractiseApplication : Simpler Consumer defined in spring boot > SpringBoot A simple Consumer Rocks!

2016-12-16 23:52:13.985 INFO 3776 --- [enerContainer-1] a.i.l.SpringBootMavenPractiseApplication : Another Simpler Consumer> SpringBoot A simple Consumer Rocks! and return Spring Messaging too!

Connect to Remote JMS Server:

You simply need to change the **application.properties** .

For example:

spring.hornetq.mode=native

spring.hornetq.host=\*\*host ip address\*\*

spring.hornetq.port=\*\*host port\*\*

You can read about all the properties for HornetQ in the Spring Boot reference at https://docs.

spring.io/spring-boot/docs/current/reference/html/common-application-properties.html .

RabbitMQ with Spring Boot

Installing RabbitMQ

Mac OSX/Linux, you can use the brew command:

$ brew upgrade

$ brew install rabbitmq

If you are using another UNIX or a Windows system, you can go to the RabbitMQ web site and use

the installers ( http://www.rabbitmq.com/download.html ).

RabbitMQ/AMQP: Exchanges, Bindings, and Queues

The AMQP defines three concepts that are a little different from the JMS world, but very easy to understand.

AMQP defines *exchanges* , which are entities where the messages are sent. Every *exchange* takes a message

and routes it to a zero or more *queues* . This routing involves an algorithm that is based on the exchange type

and some rules, called *bindings* .

The AMPQ protocol defines four exchange types: *Direct* , *Fanout* , *Topic,* and *Headers* . Figure 10-1 shows

these different exchange types.

The *default exchange* will be bound automatically to every queue created. The direct exchange is bound

to a queue by a routing key; you can see this exchange type as one-to-one binding. The *topic exchange*

is similar to the Direct Exchange; the only difference is that in its binding you can add a wildcard into its

routing key. The *headers exchange* is similar to the topic exchange; the only difference is that the binding is

based on the message headers (this is a very powerful exchange, and you can do *all* and *any* expressions for

its headers). The *fanout exchange* will copy the message to all the bound queues; you can see this exchange

as a message broadcast.

<dependency>

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

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

This pom will include all the spring-amqp and rabbitmq-client libraries needed for connecting to the

RabbitMQ Broker .

Rabbit mq broker :

Stopping the Broker

Use rabbitmqctl stop.

Checking the Broker Status

Use rabbitmqctl status. All rabbitmqctl commands will report the node absence if no broker is running (i.e. nodedown).

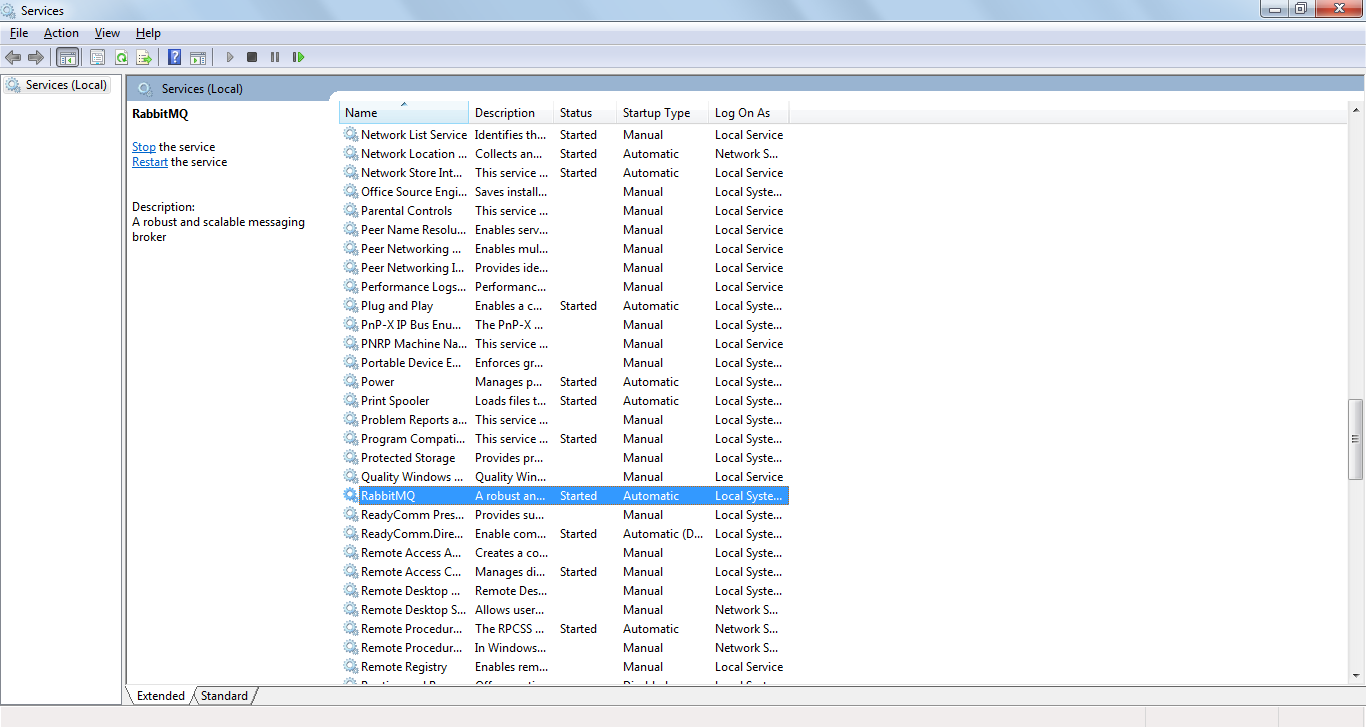
To start the broker : rabbitmqserver.bat

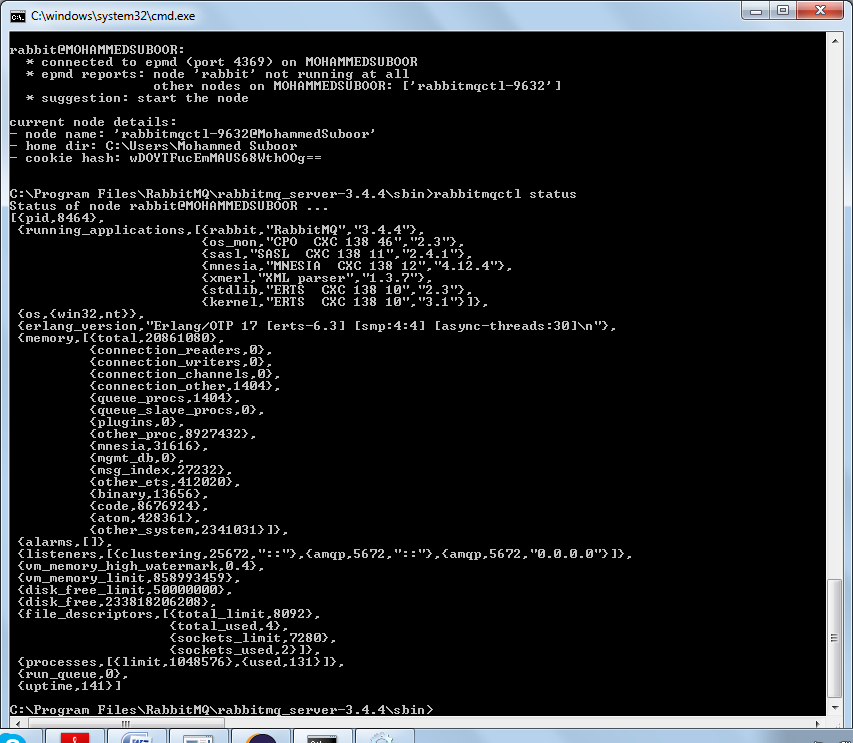
rabbitmq-server — start RabbitMQ AMQP server

rabbitmq-server –detached Runs RabbitMQ AMQP server in the background.

Starting the rabbitmq service in windows – type services in windows search programmes

After that look for rabbitmq – right click or double click and start the serices.



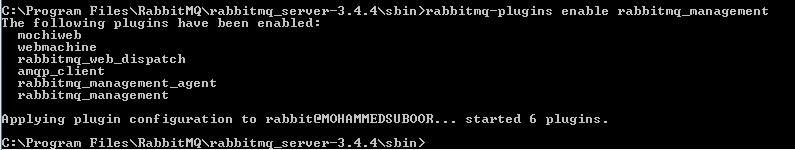


To check the rabbitmq logs in windows

C:\Users\Mohammed Suboor\AppData\Roaming\RabbitMQ\log

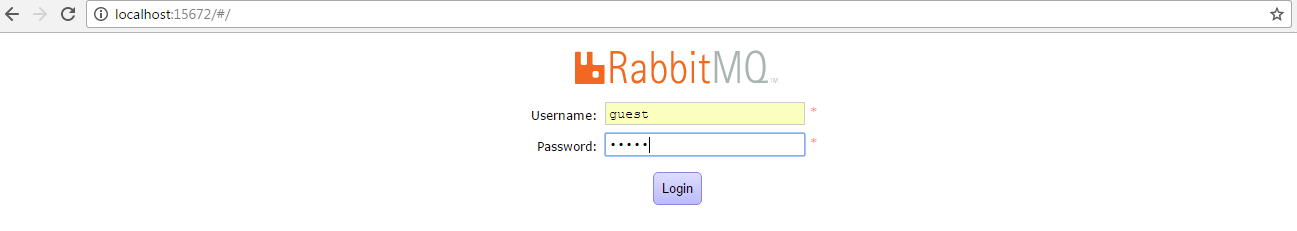
To start rabbitmq management please use below command

 rabbitmq-plugins enable rabbitmq\_management



User : guest

Pwd : guest



<dependency>

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

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

This pom will include all the spring-amqp and rabbitmq-client libraries needed for connecting to the

RabbitMQ Broker .

Example :

1. First add below queue in application.properties

myRabbitQueue = mySpringBoot-RabbitQueue

1. Create a producer

@Component

**public** **class** RabbitMqProducer {

**private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(RabbitMqProducer.**class**);

@Autowired

**private** RabbitTemplate rabbitTemplate;

**public** **void** sendTo(String routingkey, String message) {

***log***.info("Sending message using RabbitMq> ...");

**this**.rabbitTemplate.convertAndSend(routingkey, message);

}

}

1. Create a consumer

@Component

**public** **class** RabbitMqConsumer {

**private** **static** **final** Logger ***log*** = LoggerFactory.*getLogger*(RabbitMqConsumer.**class**);

@RabbitListener(queues = "${myRabbitQueue}")

**public** **void** handler(String message) {

***log***.info("Consumeing message by Consumer using Rabbit MQ> " + message);

}

}

1. Add below code in Spring boot main application class

@Value("${myRabbitQueue}")

String myrabbitQueue;

@Bean

Queue queue(){

**return** **new** Queue(myrabbitQueue, **false**);

}

@Bean

CommandLineRunner sender(RabbitMqProducer rabbitMqProducer) {

**return** args -> {

rabbitMqProducer.sendTo(myrabbitQueue, "JMS with Spring Boot using Rabbit MQ");

};

}

This example will create at runtime a *queue* named mySpringBoot-RabbitQueue, and by default all the queues are bound to a default exchange. So, when the producer sends the message it will be sent first to the default exchange then routed to the queue (mySpringBoot-RabbitQueue).

Before you run above example, make sure your RabbitMQ server is up and running. You can start it by

opening a terminal and executing the following command:

$ rabbitmq-plugins enable rabbitmq\_management

$ rabbitmq-server or rabbitmq-server.bat

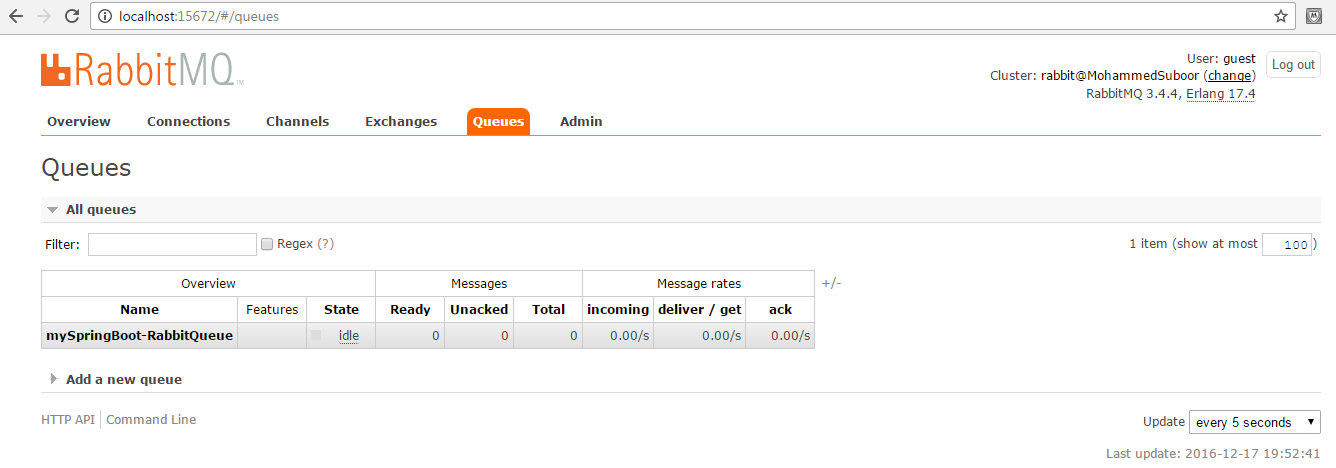
Run the spring boot

You will see below output

2016-12-17 19:10:34.144 INFO 9116 --- [ main] c.a.i.l.m.rabbitmq.RabbitMqProducer : Sending message using RabbitMq> ...

2016-12-17 19:10:34.205 INFO 9116 --- [cTaskExecutor-1] c.a.i.l.m.rabbitmq.RabbitMqConsumer : Consumeing message by Consumer using Rabbit MQ> JMS with Spring Boot using Rabbit MQ

*RabbitMQ web console Queues tab*



Remote RabbitMQ

If you want to access a remote RabbitMQ, you add the following properties to the application.properties file:

spring.rabbitmq.host=mydomain.com

spring.rabbitmq.username=rabbituser

spring.rabbitmq.password=thisissecured

spring.rabbitmq.port=5672

spring.rabbitmq.virtual-host=/production

You can always read about all the properties for RabbitMQ in the Spring Boot reference at https://

docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html .

Redis Messaging with Spring Boot

Redis (REmote DIctionary Server) is a NoSQL key-value store database. It’s written in

C and even though has a small footprint in its core, it’s very reliable, scalable, powerful, and super fast. Its

primary function is to store data structures like Lists, hashes, strings, sets, and sorted sets. One of the other

main features is that it provides a publish/subscribe messaging system, which is why you are going to use

Redis only as message broker.

WebSockets with Spring Boot

WebSockets is a new way of communication, and it’s replacing the client/server web technology.

It allows long-held single TCP socket connections between the client and server. It’s also called *push*

technology, and it’s where the server can send data to the web without the client do long polling to request a

new change.

<dependency>

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

<artifactId>spring-boot-starter-websocket</artifactId>

</dependency>

The spring-boot-starter-websocket is the pom that will bring all the dependencies that you

need for creating a WebSockets messaging application.

Spring Boot Actuator

Spring Boot includes an Actuator module , which introduces production-ready non-functional requirements

to your application. The Spring Boot Actuator module provides monitoring, metrics, and auditing right out

of box.

<dependency>

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

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

/ actuator

The /actuator endpoint will provide a hypermedia-based discovery page for all the other endpoints,

but it will require the Spring HATEOAS in the classpath, so if you include this in your pom.xml :

<dependency>

<groupId>org.springframework.hateoas</groupId>

<artifactId>spring-hateoas</artifactId>

</dependency>

You can rerun your application and you will see that now is listed by the EndpointHandlerMapping class

logs and you can access it through the URL /actuator . So, if you go to <http://localhost:8080/actuator>

shows all the links that you can access through the Actuator module. The Actuator gives

you all the possible endpoints that you can access.

/autoconfig

This endpoint will display the auto-configuration report. It will give you two groups: positiveMatches

and negativeMatches . Remember that the main feature of Spring Boot is that it will auto-configure your

application by seeing the classpath and dependencies.

<http://localhost:8080/autoconfig>

/beans

This endpoint will display all the Spring beans that are used in your application.

<http://localhost:8080/beans>

/configprops

This endpoint will list all the configuration properties that are defined by the @ConfigurationProperties

Beans

[*http://localhost:8080/configprops*](http://localhost:8080/configprops)

/docs

This endpoint will show HTML pages with all the documentation for all the Actuator module endpoints.

This endpoint can be activated by including the spring-boot-actuator-docs dependency in pom.xml :

<dependency>

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

<artifactId>spring-boot-actuator-docs</artifactId>

</dependency>

[*http://localhost:8080/docs*](http://localhost:8080/docs)

/dump

This endpoint will perform a thread dump of your application. It shows all the threads running and their stack

trace of the JVM that is running your app.

<http://localhost:8080/dump>

/env

This endpoint will expose all the properties from the Spring’s ConfigurableEnvironment interface. This will

show any active profiles and system environment variables and all application properties, including the

Spring Boot properties.

<http://localhost:8080/env>

/health

This endpoint will show the health of the application. If you are doing a database app like in the previous

section ( /flyway ) you will see the DB status and by default you will see also the diskSpace from your system.

<http://localhost:8080/health>

/info

This endpoint will display the public application info. This means that you need to add this information to

application.properties . It’s recommended that you add it if you have multiple Spring Boot applications.

Example :

info.app.name=Spring Boot Web Actuator Application

info.app.description=This is an example of the Actuator module

info.app.version=1.0.0

<http://localhost:8080/info>

/logfile

This endpoint will show the contents of the log file specified by the logging.file property , where you specify

the name of the log file (this will be written in the current directory). You can also set the logging.path , where

you set the path where the spring.log will be written. By default Spring Boot writes to the console/standard

out, and if you specify any of these properties, it will also write everything from the console to the log file.

Go to src/main/resources/application.properties and add this to the very end:

**logging.file=mylog.log**

Now you can rerun your application and go to the

<http://localhost:8080/logfile> endpoint will show the mylog.log

/ metrics

This endpoint shows the metrics information of the current application, where you can determine the how

much memory it’s using, how much memory is free, the uptime of your application, the size of the heap is

being used, the number of threads used, and so on.

[*http://localhost:8080/metrics*](http://localhost:8080/metrics)

/mappings

This endpoint shows all the lists of all @RequestMapping paths declared in your application. This is very

useful if you want to know more about what mappings are declared

<http://localhost:8080/mappings>

/shutdown

This endpoint is not enabled by default. It allows the application to be gracefully shut down. This endpoint

is sensitive, which means it can be used with security, and it should be. If your application is running,

you can stop it now. If you want to enable the /shutdown endpoint, you need to add the following to the

application.properties .

endpoints.shutdown.enabled=true

/trace

This endpoint shows the trace information, which is normally the last few HTTP requests. This endpoint

can be useful to see all the request info and the information returned to debug your application at the HTTP

level. You can run your application and go to <http://localhost:8080/trace>

**Deploying Spring Boot**

To create the standalone and executable journal app, execute the following command:

$ ./mvnw package

This command will create a target/spring-boot-journal-0.0.1-SNAPSHOT.jar file.

Now you can run it with the following:

$ java -jar target/spring-boot-journal-0.0.1-SNAPSHOT.jar

You can execute $ ./mvnw package -DskipTests=true to skip the tests.

If you already have application servers like Pivotal tc Server , Tomcat ,JBoss , or Web Sphere, and are used to deploying WAR files?

With Spring Boot apps, it’s really easy. You need to change two things:

1. Modify pom.xml (or build.gradle ).

• Change the <packaging> tag from jar to war (or apply the plugin war if you are

using Gradle).

• Add the spring-boot-starter-tomcat dependency to your pom.xml and set the

scope to provided (or in your build.gradle set the name in the configurations

section to a providedRuntime if you are using Gradle).

<dependency>

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

<artifactId>spring-boot-starter-tomcat</artifactId>

<scope>provided</scope>

</dependency>

when you package your application, all the libraries will now be placed in the WEB-INF/lib and

the WEB-INF/lib-provided for the Tomcat libraries within the “Fat JAR”.

This will make your application executable as standalone app and container-ready

This class extends from the SpringBootServletInitializer and it’soverriding the configure(SpringApplicationBuilder application) method. That will help to bootstrap the application. Again, this is important if you want to deploy it in application container like Pivotal tcServer,Tomcat, etc.

Execute the following command:

$ ./mvnw clean package -DskipTests=true

Now you will have your target/spring-boot-journal-0.0.1-SNAPSHOT.war file ready to be executed

with the following command:

$ java -jar target/spring-boot-journal-0.0.1-SNAPSHOT.war

Activating Profiles

One of the recommended ways is to have several application-<profile>.properties files, so you can

activate them in standalone mode or in the container.

$ ./mvnw clean package -DskipTests=true

$ java -Dspring.profiles.active="container" -jar target/spring-boot-journal-0.0.1-SNAPSHOT.war

In the logs you can see (in the first three lines) the legend: "The following profiles are active:

container" .

Creating Spring Boot Apps as a Service

Another amazing feature of Spring Boot is that you can run your app as a service. Running a Spring Boot

as a service has its benefits. It’s easy to install and manage, and if the server restarts, your app will start

automatically without you having to do it manually.

add a configuration declaration to the Spring Boot plugin in the pom.xml file and in build.gradle

<configuration>

<executable>true</executable>

</configuration>

$ ./mvnw clean package -DskipTests=true

You can execute the JAR directly:

$ target/spring-boot-journal-0.0.1-SNAPSHOT.war

And it will run! So, if you are running a UNIX environment you can just bind it to the /etc/init.d

(in a Debian environment, assuming you have the executable in the /opt folder):

$ ln -s /opt/spring-boot-journal-0.0.1-SNAPSHOT.war /etc/init.d/journal

Then you can start your application with the following:

$ service journal start

Spring Boot Apps as Windows Service

If you are looking to do this in a Windows environment, you can take a look at this URL at https://github.

com/snicoll-scratches/spring-boot-daemon, which contains all the information you need to create a

Spring Boot application and run it as a Windows service.

Spring Boot with Docker

Make sure you have Docker up and running. Before you build the image that will contain the journal

app, let’s use profiles again and add a new dependency to the pom.xml file. See Listing 12-11 .

***Listing 12-11.*** src/main/resources/application-docker.properties

spring.datasource.url=jdbc:h2:mem:testdb;MODE=Oracle;DB\_CLOSE\_DELAY=-1;DB\_CLOSE\_ON\_EXIT=FALSE

spring.datasource.username=sa

spring.datasource.password=

spring.datasource.driverClassName=org.h2.Driver

spring.data.rest.basePath=/api

management.context-path=/monitor

endpoints.shutdown.enabled=true

server.port=8443

server.ssl.key-store=classpath:keystore.jks

server.ssl.key-store-password=tomcat

server.ssl.key-password=tomcat

Listing 12-11 shows the application-docker.properites . All the spring.datasource properties are

new. It has defined the H2 database that will run in-memory and also contains the SSL. Remember that in

order to use the H2 database you must include this dependency in the pom.xml file:

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

Now, you can create in the root folder the **Dockerfile** (this file helps create Docker images) (src/main/resources folder --src/main/resources/docker/Dockerfile) and add the plugin to the pom.xml (or Gradle) file.

Dockerfile

FROM java:8

VOLUME /tmp

ADD target/spring-boot-journal-0.0.1-SNAPSHOT.war journal.war

ENV SPRING\_PROFILES\_ACTIVE docker

EXPOSE 8443

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/journal.war"]

Explanation:

• FROM java:8 . This line pulls a Debian 8 image (jessie) that contains the OpenJDK

version 8.

• VOLUME . Needed to create a volume, because Spring Boot creates working directories

for Tomcat by default.

• ADD . Copies the WAR (or JAR) file as journal.war (or Jar) in the root of the container.

• ENV . Needed to add the environment variable that will activate the Docker profile.

• EXPOSE . It’s exposing the port 8443 . Remember that this is the port for the SSL.

• ENTRYPOINT . This declaration determines how the container will execute when it

starts up. To reduce the Tomcat startup time, you need a system property pointing to

" /dev/./urandom " as a source of entropy.

Next, execute the following command to build the Docker image:

$ docker build -t springboot/journal

This command will build an image with the springboot/journal tag name. After it finishes building,

you can run it with this command:

$ docker run -p 8443:8443 springboot/journal

This command will run the container using the springboot/journal image. Now you can go to your

browser and open the journal app. If you are running this example in Linux, just go to https://localhost:8443 .

If you are using a Mac or a Windows machine, use the Docker IP:

$ docker-machine ip

192.168.99.100

Now you can go to https://192.168.99.100:8443 (or any Docker IP) and voilà! You have your journal

app running in a Docker container!

Now you can build it and create the image in the same line with:

$ ./mvnw clean package docker:build -DskipTests=true

**Spring Boot in the Cloud**

Cloud Foundry is the only open source solution that you can actually download and run it without

any problems, it just work! You can find two versions of Cloud Foundry, the open source:

https://www.cloudfoundry.org/ and the commercial version: http://pivotal.io/platform . If you are

interested in download the commercial version, you can actually do it without any trials or limited time:

https://network.pivotal.io/products/pivotal-cf , actually is a free version, but if you want to have

support or help on how to install it, that’s when you need to contact a Pivotal sales representative.

Plugins in pom.xml :

<properties>

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

<java.version>1.8</java.version>

**<docker.image.prefix>springboot</docker.image.prefix>**

</properties>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

**<plugin>**

**<groupId>com.spotify</groupId>**

**<artifactId>docker-maven-plugin</artifactId>**

**<version>0.4.1</version>**

**<configuration>**

**<imageName>${docker.image.prefix}/${project.artifactId}**

**</imageName>**

**<dockerDirectory>src/main/resources/docker</dockerDirectory>**

**<resources>**

**<resource>**

**<targetPath>/</targetPath>**

**<directory>${project.build.directory}</directory>**

**<include>${project.build.finalName}.war</include>**

**</resource>**

**</resources>**

**</configuration>**

**</plugin>**

</plugins>

</build>