Spring Boot Table of Contents

1. Introducing to Spring Boot

Spring Team has released one of major innovation on the top of existing Spring Framework is Spring Boot. It is a completely new project from Pivotal Team (The Spring Team). Spring Boot is their latest innovation to keep up to date with the changing technology needs. The primary motivation behind developing Spring Boot is to simplify the process for configuring and deploying the spring applications. This Spring Boot Tutorial gives complete introduction about Spring Boot.

Spring Boot offers a new paradigm for developing Spring applications with minimal friction. With Spring Boot, you’ll be able to develop Spring applications with more agility and be able to focus on addressing your application’s functionality needs with minimal (or possibly no) thought of configuring Spring itself. It uses completely new development model to make Java Development very easy by avoiding some tedious development steps and boilerplate code and configuration.

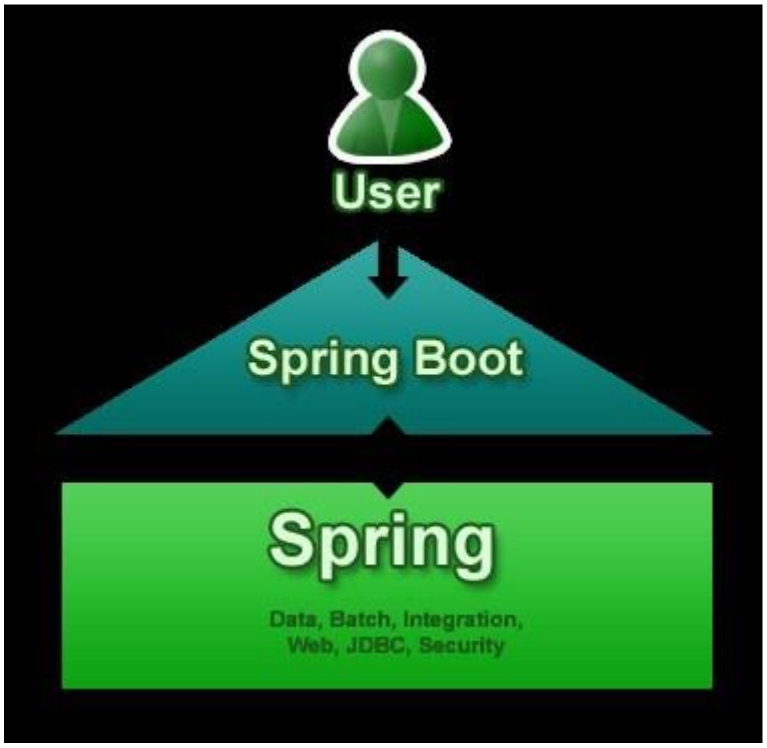
Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”. We take an opinionated view of the spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

You can use Spring Boot to create Java applications that can be started using java -jar or more traditional war deployments. We also provide a command line tool that runs “spring scripts”.

2. What is spring boot?

In this Spring Boot Tutorial, first of all Spring Boot is not a framework, it is a way to ease to create stand-alone application with minimal or zero configurations. It is approach to develop spring based application with very less configuration. It provides defaults for code and annotation configuration to quick start new spring projects within no time. Spring Boot leverages existing spring projects as well as Third party projects to develop production ready applications. **It provides a set of Starter Pom's or gradle build files which one can use to add required dependencies and also facilitate auto configuration.**

Spring Boot automatically configures required classes depending on the libraries on its classpath. Suppose your application want to interact with DB, if there are Spring Data libraries on class path then it automatically sets up connection to DB along with the Data Source class.



Let's see the primary goals of the Spring Boot in this Spring Boot Tutorial.

3. Spring Boot Primary Goals

Let's see Spring Boot primary goals are following in this Spring Boot Tutorial:

- Provide a radically faster and widely accessible getting started experience for all Spring development.
- Be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults.
- Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).
- Absolutely no code generation and no requirement for XML configuration, to avoid XML Configuration completely
- To avoid defining more Annotation Configuration(It combined some existing Spring Framework Annotations to a simple and single Annotation)
- Spring Boot avoid writing lots of import statements
- To provide some defaults to quick start new projects within no time.

4. Why New Project Need Spring Boot?

Suppose we are developing one of **Hello World** application in **Spring Framework**, for that there is only one item is specific to developing the **Hello World** functionality: the controller. The rest of it is generic boilerplate that you'd need for any web application developed with Spring. But if all Spring web applications need it, why should you have to provide it? So there are following things any new project need Spring Boot.

- To ease the Java-based applications Development, Unit Test and Integration Test Process.
- To reduce Development, Unit Test and Integration Test time by providing some defaults.
- Spring Boot increase Productivity.
- When we talk about defaults, Spring Boot has its own opinions. If you are not specifying the details, it will use its own default configurations. If you want persistence, but don't specify anything else in your POM file, then Spring Boot configures Hibernate as a JPA provider with an HSQLDB database.
- To provide bunch of non-functional features/solutions that are very much common to large scale projects (e.g. embedded servers, security, metrics, health checks, externalized configuration).

5.What Spring Boot Isn't?

As above mention Spring Boot is not a framework to write applications, it helps you to develop and build, package and deploy application with minimal configuration or zero configuration.

It is not an application server. But it's the embedded servlet container that provides application server functionality, not Spring Boot itself.

Similarly, Spring Boot doesn't implement any enterprise Java specifications such as JPA or JMS. For example, Spring Boot doesn't implement JPA, but it does support JPA by auto-configuring the appropriate beans for a JPA implementation (such as Hibernate)

Finally, Spring Boot doesn't employ any form of code generation to accomplish its magic. Instead, it leverages conditional configuration features from Spring 4, along with transitive dependency resolution offered by Maven and Gradle, to automatically configure beans in the Spring application context.

In short, at its heart, Spring Boot is just Spring.

Future Spring projects would not have any XML configurations as part of it, everything will be handled by the project Spring Boot.

6. Pros/Cons of Spring Boot

Pros of Spring Boot:

- It is very easy to develop Spring Based applications with Java or Groovy.
- Spring Boot reduces lots of development time and increases productivity.
- It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
- It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security etc.
- Spring Boot follows "Opinionated Defaults Configuration" Approach to reduce Developer effort
- It provides Embedded HTTP servers like Tomcat, Jetty etc. to develop and test our web applications very easily.
- It provides CLI (Command Line Interface) tool to develop and test Spring Boot (Java or Groovy) Applications from command prompt very easily and quickly.
- Spring Boot provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle
- It provides lots of plugins to work with embedded and in-memory Databases very easily.

Limitation of Spring Boot:

It is very tough and time consuming process to convert existing or legacy Spring Framework projects into Spring Boot Applications. It is applicable only for brand new/Greenfield Spring Projects.

7. Spring boot releases

Latest Release: Spring Boot 1.3.3 and 1.4.0 available now. You require minimum Spring Framework 4.2.2 for this version.

Features Added in SpringBoot 1.4.0 are:

- · Executable JAR Layout
- · Startup error improvements
- · Hibernate 5
- · Spring Framework 4.3
- · Third Party Library
- · Custom JSON Serializer and Deserializer
- · New auto-configuration support
  - Couchbase
  - Neo4j
  - Narayana transactional manager
  - Caffeine Cache
- · Actuator improvements
- · Testing improvements

8. Getting started with Spring Boot

Ultimately, a Spring Boot project is just a regular Spring project that happens to leverage Spring Boot starters and auto-configuration. To Start Opinionated Approach to create Spring Boot Applications, The Spring Team (The Pivotal Team) has provided the following three approaches.

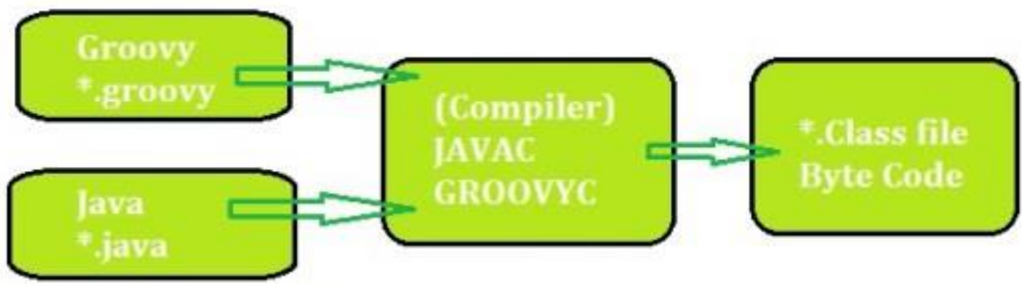
- Using Spring Boot CLI Tool
- Spring STS IDE
- Using Spring Initializr (**Website** http://start.spring.io/)

We can develop two flavors of Spring-Based Applications using Spring Boot

- Java-Based Applications

- Groovy Applications

We can use Spring Boot CLI or Spring STS IDE or Spring Initializr Website to develop Spring Boot Groovy Applications. However, we can use Spring STS IDE or Spring Initializr Website to develop Spring Boot Java Applications.



Anyhow, Groovy is also JVM language almost similar to Java Language. We can combine both Groovy and Java into one Project. Because like Java files, Groovy files are finally compiled into *.class files only. Both *.groovy and *.java files are converted to *.class file (Same byte code format).

Spring Boot Framework Programming model is inspired by Groovy Programming model. It internally uses some Groovy based techniques and tools to provide default imports and configuration.

Spring Boot Framework also combined existing Spring Framework annotations into some simple or single annotations. We will explore those annotations one by one in coming posts with some real-time examples.

Spring Boot Framework drastically changes Spring-Java Based Applications Programming model into new Programming model. As of now, Spring Boot is at initial stage only but future is all about Spring Boot only.

Spring Boot CLI

It is the easiest and quickest way to start using the Spring Boot. It is a command line tool used for executing the groovy scripts. In summary, you can install this tool by following these steps:

1. Download the binary distributions for this project from here. Spring Boot CLI requires Java JDK v1.6 or above in order to run. Groovy v2.1 is packaged as part of this distribution, and therefore does not need to be installed (any existing Groovy installation is ignored)

2. If you unpack the zip file, you will find **spring.bat** which will check the all the settings. This script can be found under the directory /bin.

9. Hello World example using spring boot

We can develop two flavors of Spring-Based Applications using Spring Boot.

- Groovy Applications
- Java-Based Applications

Groovy Applications:

Let's develop a simple "Hello World!" web application. Create the **app.groovy** with the following lines of code. Here we are going explain same example from Spring Boot documentation.

```
@RestController

class ThisWillActuallyRun {



    @RequestMapping("/")

    String home() {

        return "Hello World!"

    }



}
```

Run it as follows:

```
$ spring run app.groovy
```

You can invoke the **http://localhost:8080** in your browser and you will see the result “Hello World!”.The above command can invoke the application and run it in the web server.

Spring Boot does this by dynamically adding key annotations to your code and leveraging Groovy Grape to pull down needed libraries to make the app run. If any dependencies are required like web server, etc. can be resolved by the Spring Boot itself.

Java-Based Applications

Let’s develop a simple “**Hello World**!” web application in Java that highlights some of Spring Boot’s key features. We’ll use Maven to build this project since most IDEs support it.

If you are Java developer you can use start.spring.io to generate a basic project.

Build with Maven

Pom.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project                              xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>



    <groupId>com.doj</groupId>

    <artifactId>my-spring-boot-project</artifactId>

    <version>0.0.1-SNAPSHOT </version>



    <parent>

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

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

        <version>1.3.5.RELEASE</version>

    </parent>



    <dependencies>

        <dependency>

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

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

        </dependency>

    </dependencies>
    <properties>

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

    </properties>




    <build>

        <plugins>

            <plugin>

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

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

            </plugin>

        </plugins>

    </build>


</project>
```

Once maven is in the class path, it is very easy to create a maven project and run the example. Just create a Java file with the below code snippet:

```
package hello;



import org.springframework.boot.*;
```

```java
import org.springframework.boot.autoconfigure.*;

import org.springframework.stereotype.*;

import org.springframework.web.bind.annotation.*;


@Controller

@EnableAutoConfiguration

public class SampleController {


    @RequestMapping("/")

    @ResponseBody

    String home() {

        return "Hello World!";

    }


    public static void main(String[] args) throws Exception {

        SpringApplication.run(SampleController.class, args);

    }

}
```

Once you have created the above Java file and the pom.xml in the same directory, please run the following command:

```
$ mvn package
```

Name	Description	Pom
spring-boot-starter-test	Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito	Pom
spring-boot-starter-mobile	Starter for building web applications using Spring Mobile	Pom
spring-boot-starter-social-twitter	Starter for using Spring Social Twitter	Pom
spring-boot-starter-cache	Starter for using Spring Framework's caching support	Pom
spring-boot-starter-activemq	Starter for JMS messaging using Apache ActiveMQ	Pom
spring-boot-starter-jta-atomikos	Starter for JTA transactions using Atomikos	Pom
spring-boot-starter-aop	Starter for aspect-oriented programming with Spring AOP and AspectJ	Pom
spring-boot-starter-web	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container	Pom
spring-boot-starter-data-elasticsearch	Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch	Pom
spring-boot-starter-jdbc	Starter for using JDBC with the Tomcat JDBC connection pool	Pom
spring-boot-starter-batch	Starter for using Spring Batch, including HSQLDB in-memory database	Pom
spring-boot-starter-social-facebook	Starter for using Spring Social Facebook	Pom
spring-boot-starter-web-services	Starter for using Spring Web Services	Pom
spring-boot-starter-jta-narayana	Spring Boot Narayana JTA Starter	Pom
spring-boot-starter-thymeleaf	Starter for building MVC web applications using Thymeleaf views	Pom
spring-boot-starter-mail	Starter for using Java Mail and Spring Framework's email sending support	Pom
spring-boot-starter-jta-bitronix	Starter for JTA transactions using Bitronix	Pom
spring-boot-starter-data-mongodb	Starter for using MongoDB document-oriented database and Spring Data MongoDB	Pom
spring-boot-starter-validation	Starter for using Java Bean Validation with Hibernate Validator	Pom
spring-boot-starter-jooq	Starter for using jOOQ to access SQL databases. An alternative to spring-boot-starter-data-jpa or spring-boot-starter-jdbc	Pom

spring-boot-starter-redis	Starter for using Redis key-value data store with Spring Data Redis and the Jedis client. Deprecated as of 1.4 in favor of spring-boot-starter-data-redis	Pom
spring-boot-starter-data-cassandra	Starter for using Cassandra distributed database and Spring Data Cassandra	Pom
spring-boot-starter-hateoas	Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS	Pom
spring-boot-starter-integration	Starter for using Spring Integration	Pom
spring-boot-starter-data-solr	Starter for using the Apache Solr search platform with Spring Data Solr	Pom
spring-boot-starter-freemarker	Starter for building MVC web applications using Freemarker views	Pom
spring-boot-starter-jersey	Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to spring-boot-starter-web	Pom
spring-boot-starter	Core starter, including auto-configuration support, logging and YAML	Pom
spring-boot-starter-data-couchbase	Starter for using Couchbase document-oriented database and Spring Data Couchbase	Pom
spring-boot-starter-artemis	Starter for JMS messaging using Apache Artemis	Pom
spring-boot-starter-cloud-connectors	Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku	Pom
spring-boot-starter-social-linkedin	Stater for using Spring Social LinkedIn	Pom
spring-boot-starter-velocity	Starter for building MVC web applications using Velocity views. Deprecated since 1.4	Pom
spring-boot-starter-data-rest	Starter for exposing Spring Data repositories over REST using Spring Data REST	Pom
spring-boot-starter-data-gemfire	Starter for using GemFire distributed data store and Spring Data GemFire	Pom
spring-boot-starter-groovy-templates	Starter for building MVC web applications using Groovy Templates views	Pom
spring-boot-starter-amqp	Starter for using Spring AMQP and Rabbit MQ	Pom

spring-boot-starter-hornetq	Starter for JMS messaging using HornetQ. Deprecated as of 1.4 in favor of spring-boot-starter-artemis	Pom
spring-boot-starter-ws	Starter for using Spring Web Services. Deprecated as of 1.4 in favor of spring-boot-starter-web-services	Pom
spring-boot-starter-security	Starter for using Spring Security	Pom
spring-boot-starter-data-redis	Starter for using Redis key-value data store with Spring Data Redis and the Jedis client	Pom
spring-boot-starter-websocket	Starter for building WebSocket applications using Spring Framework's WebSocket support	Pom
spring-boot-starter-mustache	Starter for building MVC web applications using Mustache views	Pom
spring-boot-starter-data-neo4j	Starter for using Neo4j graph database and Spring Data Neo4j	Pom
spring-boot-starter-data-jpa	Starter for using Spring Data JPA with Hibernate	Pom

```
$ java -jar target/ my-spring-boot-project-0.0.1-SNAPSHOT.jar
```

First statement invokes the pom.xml and builds the complete package by downloading all the dependencies. The second statement makes the spring application and bundle with the web server. As said in the above groovy example, you can invoke the application in the browser.

10. Starters

Starters are a set of convenient dependency descriptors that you can include in your application. The starters contain a lot of the dependencies that you need to get a project up and running quickly and with a consistent, supported set of managed transitive dependencies.

The starter POMs are convenient dependency descriptors that can be added to your application's Maven. In simple words, if you are developing a project that uses Spring Batch for batch processing, you just have to include **spring-boot-starter-batch** that will import all the required dependencies for the Spring Batch application. This reduces the burden of searching and configuring all the dependencies required for a framework.

List of Starters

Spring offers wide range of started POMs that can be used in your application. Here is the list of started POMs mention in this Spring Boot Tutorial.

11. Servlet containers Support

The following embedded servlet containers are supported out of the box.

Name	Servlet Version	Java Version
Tomcat 8	3.1	Java 7+
Tomcat 7	3.0	Java 6+
Jetty 9	3.1	Java 7+
Jetty 8	3.0	Java 6+
Undertow 1.1	3.1	Java 7+

You can also deploy Spring Boot applications to any Servlet 3.0+ compatible container.

12. Template engines Support

Spring Boot includes auto-configuration support for the following templating engines mention in this Spring Boot Tutorial.

- · FreeMarker
- · Groovy
- · Thymeleaf
- · Velocity (deprecated in 1.4)
- · Mustache

JSPs should be avoided if possible; there are several known limitations when using them with embedded servlet containers. If you are using any of the above template engines, spring boot will automatically pick the templates from src/main/resources/templates.

13. Caching Support

The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to methods, reducing thus the number of executions based on the information available in the cache. The caching logic is applied transparently, without any interference to the invoker. Let;s see the code as mention below here Spring Boot Tutorial.

```
import javax.cache.annotation.CacheResult;
import org.springframework.stereotype.Component;



@Component

public class MathService {



    @CacheResult

    public int computePiDecimal(int i) {

        // ...

    }



}
```

In this Spring Boot Tutorial, I have mentioned following list that Spring Boot tries to detect the following providers (in this order):

- · Generic
- · JCache (JSR-107)
- · EhCache 2.x
- · Hazelcast
- · Infinispan
- · Couchbase
- · Redis
- · Caffeine
- · Guava
- · Simple

14. Spring Boot & Spring MVC

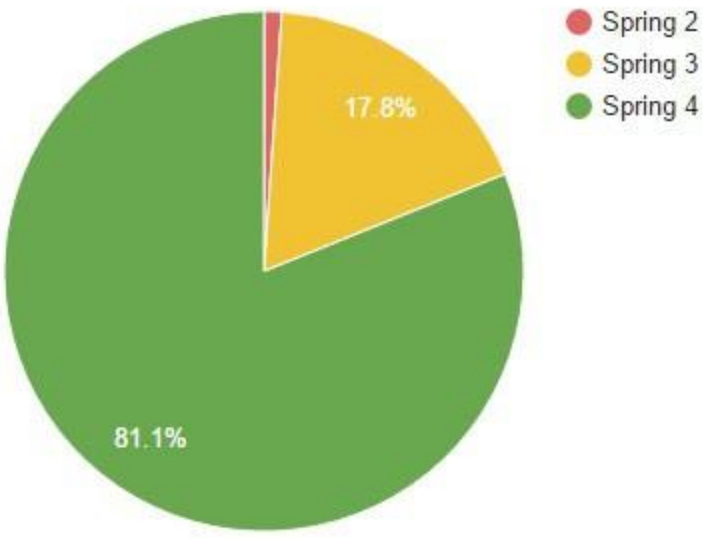
Spring Boot is well suited for web application development. You can easily create a self-contained HTTP server using embedded Tomcat, Jetty, or Undertow. Most web applications will use the spring-boot-starter-web module to get up and running quickly.

Spring MVC lets you create special **@Controller or @RestController** beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP using **@RequestMapping** annotations.

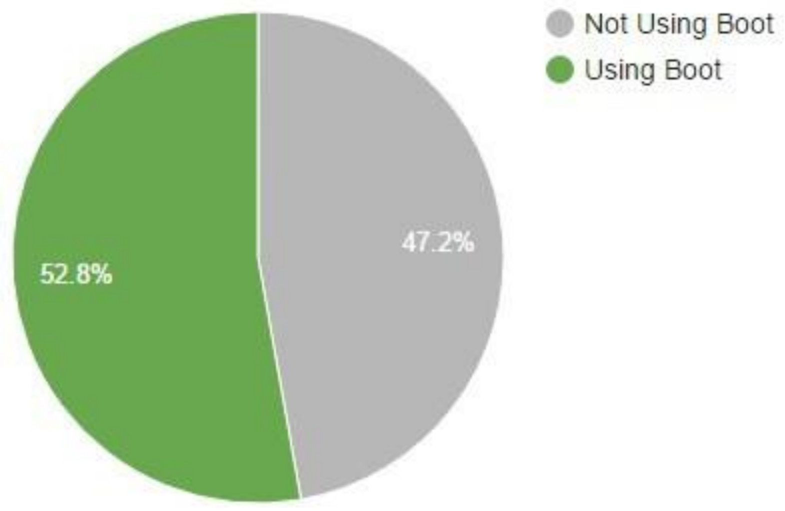
Spring MVC auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications.The auto-configuration adds the following features on top of Spring’s defaults:

- Inclusion of ContentNegotiatingViewResolver and BeanNameViewResolver beans.
- Support for serving static resources, including support for WebJars (see below).
- Automatic registration of Converter, GenericConverter, Formatter beans.
- Support for HttpMessageConverters (see below).
- Automatic registration of MessageCodesResolver (see below).
- Static index.html support.
- Custom Favicon support.
- Automatic use of a ConfigurableWebBindingInitializer bean
    1.

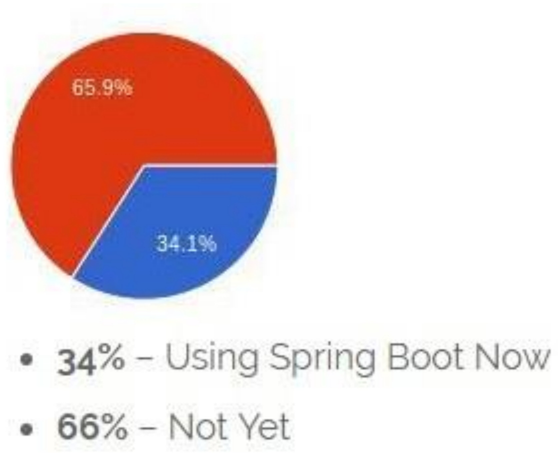


Spring Boot is even more interesting, **growing from 34% last year to 52.8%**:



Spring Boot Adoption in 2015

Eugen has published a report about the Spring Boot adoption here. This chart shows that there is more developers started using Spring Boot for their project development.



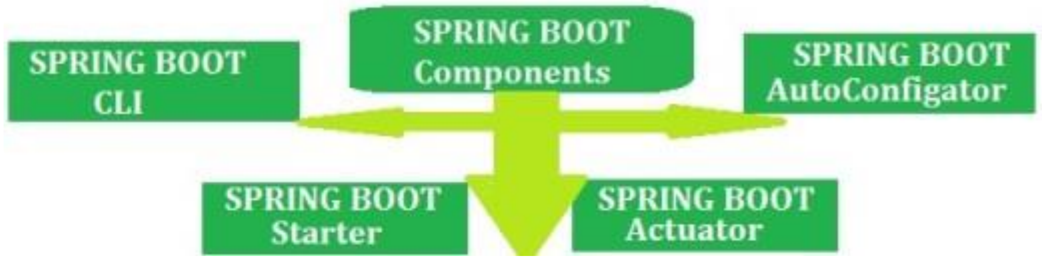
Essentials and Internals of Spring Boot Key Components:

Spring Boot gives us a great deal of magic to Spring application development. But there are four core essentials or key components that it performs:

- **Spring Boot Starters-** We just tell Spring Boot what kind of functionality we need; now it is responsibility of Spring Boot so that it will ensure that the libraries needed are added to the build.
- **Spring Boot AutoConfigurator-** Spring Boot can automatically provide configuration for application functionality common to many Spring applications.
- **Spring Boot CLI-** This optional feature of Spring Boot lets you write complete applications with just application code, but no need for a traditional project build.
- **Spring Boot Actuator-** Gives us insight of application into what's going on inside of a running Spring Boot application.

Each of these key components serves to simplify Spring application development in its own way. There are two ways for creating Spring Boot application or help to create spring boot application.

- **Spring Initializr-** To quick start new Spring Boot projects, we can use "Spring Initializr" web interface. Spring Initializr URL: ***http://start.spring.io***.
- **Spring Boot IDEs-** We has many Spring Boot IDEs like Eclipse IDE, IntelliJ IDEA, Spring STS Suite etc.



1. Spring Boot Starters

When we start to working with a spring project then it can be challenging to add dependencies to a project's build. What libraries do you need? What are its group and artifact? Which version do you need? Will that version play well with other dependencies in the same project?

Spring Boot Starters is one of the major key features or components of Spring Boot. Spring Boot offers help with project dependency management by way of starter dependencies. The main responsibility of Spring Boot Starter is to combine a group of common or related dependencies into single dependencies.

For example, suppose that we're going to build a REST API with Spring MVC that works with JSON resource representations. To accomplish all of this, you'll need (at minimum) the following four dependencies in your Maven or Gradle build:

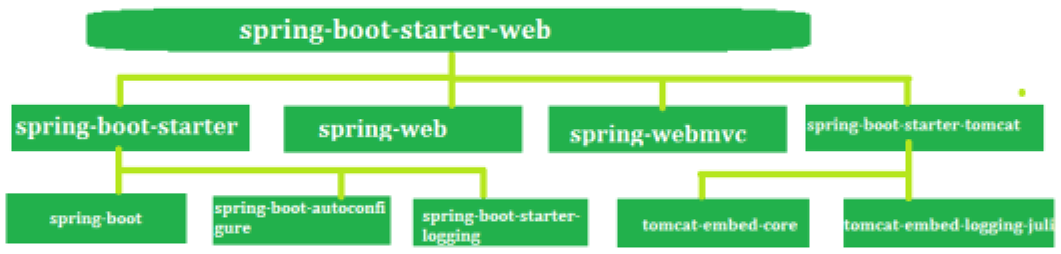
- *org.springframework:spring-core*
- *org.springframework:spring-web*
- *org.springframework:spring-webmvc*
- *com.fasterxml.jackson.core:jackson-databind*

Additionally, you want to apply declarative validation per the JSR-303 specification and serve the application using an embedded Tomcat server. Then we have added four more dependencies as below:

- *org.hibernate:hibernate-validator*
- *org.apache.tomcat.embed:tomcat-embed-core*
- *org.apache.tomcat.embed:tomcat-embed-el*
- *org.apache.tomcat.embed:tomcat-embed-logging-juli*

It is very tedious and cumbersome tasks for a Developer. And also it increases our build file size. What is the solution to avoid this much dependencies definitions in our build files? The solution is Spring Boot Starter component.

On the other hand, we are using Spring Boot Starter Component; it combines all related jars into single jar file so that we can add only one jar file dependency to our build files. If we were to take advantage of Spring Boot starter dependencies, We could simply add the Spring Boot "web" starter (org.springframework.boot:spring-boot-starter-web) as a build dependency. This single dependency will transitively pull in all of those other dependencies so you don't have to ask for them all.


spring-boot-starter-web
spring-boot-starter
spring-web
spring-webmvc
spring-boot-starter-tomcat
spring-boot
spring-boot-autoconfigure
spring-boot-starter-logging
tomcat-embed-core
tomcat-embed-logging-juli

One thing we have to notice that by adding the “web” starter to our build, we’re specifying a type of functionality that your application needs. Here our app is a web application, so we add the “web” starter. Likewise, if our application will use JPA persistence, then we can add the “jpa” starter. If it needs security, we can add the “security” starter. In short, we no longer need to think about what libraries we’ll need to support certain functionality; we simply ask for that functionality by way of the pertinent starter dependency. Also note that Spring Boot’s starter dependencies free us from worrying about which versions of these libraries we need.

Major Advantages of Spring Boot Starter

- Spring Boot Starter reduces defining many dependencies simplify project build dependencies.
- Spring Boot Starter simplifies project build dependencies.
- Spring Boot Starter sync version compatibilities of dependencies.

2. Spring Boot AutoConfigurator

There is one of them important key component of Spring Boot is Spring Boot AutoConfigurator. In any given Spring application’s source code, we’ll find either Java configuration or XML configuration (or both) that enables certain supporting features and functionality for the application that is why it requires lot of configuration (either Java configuration or XML configuration).

The main responsibility of Spring Boot AutoConfigurator is to reduce the Spring Configuration. If we develop Spring applications in Spring Boot, then We don’t need to define single XML configuration and almost no or minimal Annotation configuration. Spring Boot AutoConfigurator component will take care of providing that information.

For example, if we’ve ever written an application that accesses a relational database with JDBC, we’ve probably configured Spring’s JdbcTemplate as a bean in the Spring application context. I’ll bet the configuration looked a lot like this:

```
@Bean

public JdbcTemplate jdbcTemplate(DataSource dataSource) {

return new JdbcTemplate(dataSource);

}
```

This very simple bean declaration creates an instance of JdbcTemplate, injecting it with its one dependency, a DataSource. Of course, that means that we’ll also need to configure a DataSource bean so that the dependency will be met. To complete this configuration scenario, suppose that we were to configure an embedded H2 database as the DataSource bean:

```
@Bean

public DataSource dataSource() {

return new EmbeddedDatabaseBuilder()

.setType(EmbeddedDatabaseType.H2)

.addScripts('schema.sql', 'data.sql')

.build();

}
```

This bean configuration method creates an embedded database, specifying two SQL scripts to execute on the embedded database. The build() method returns a Data-Source that references the embedded database.

Neither of these two bean configuration methods is terribly complex or lengthy. But they represent just a fraction of the configuration in a typical Spring application. Moreover, there are countless Spring applications that will have these exact same methods. Any application that needs an embedded database and a JdbcTemplate will need those methods. In short, it’s boilerplate configuration.

If it’s so common, then why should you have to write it? The solution to this problem is Spring Boot AutoConfigurator.

And also Spring Boot reduces defining of Annotation configuration. If we use @SpringBootApplication annotation at class level, then Spring Boot AutoConfigurator will automatically add all required annotations to Java Class ByteCode.

```
Target(value=TYPE)

@Retention(value=RUNTIME)

@Documented
```

```
@Inherited

@Configuration

@EnableAutoConfiguration

@ComponentScan

public @interface SpringBootApplication
```

That is, ***@SpringBootApplication*** = ***@Configuration*** + ***@ComponentScan*** + ***@EnableAutoConfiration***.

Note- in Shot, **Spring Boot Starter** reduces **build's dependencies** and **Spring Boot AutoConfigurator** reduces the **Spring Configuration.**

3. Spring Boot CLI

In addition to auto-configuration and starter dependencies, Spring Boot also offers an intriguing new way to quickly write Spring applications. Spring Boot's CLI leverages starter dependencies and auto-configuration to let us focus on writing code. When we run Spring Boot applications using CLI, then it internally uses Spring Boot Starter and Spring Boot AutoConfigurate components to resolve all dependencies and execute the application.

Spring Boot's CLI is an optional piece of Spring Boot's power. Although it provides tremendous power and simplicity for Spring development, it also introduces a rather unconventional development model.

Spring Boot CLI has introduced a new "spring" command to execute Groovy Scripts from command prompt.

```
spring run app.groovy
```

Spring Boot CLI component requires many steps like CLI Installation, CLI Setup, Develop simple Spring Boot application and test it.

4. Spring Boot Actuator

The final key component of the Spring Boot is the Actuator. Where the other parts of Spring Boot simplify Spring development, the Actuator instead offers the ability to inspect the internals of your application at runtime. Spring Boot Actuator components gives many features, but two major features are

- Providing Management EndPoints to Spring Boot Applications.
- Spring Boot Applications Metrics.

The Actuator exposes this information in two ways: via web endpoints or via a shell interface.

We inspect the inner workings of your application with Actuator as below:

- What beans have been configured in the Spring application context
- What decisions were made by Spring Boot's auto-configuration
- What environment variables, system properties, configuration properties, and command-line arguments are available to your application
- The current state of the threads in and supporting your application
- A trace of recent HTTP requests handled by your application
- Various metrics pertaining to memory usage, garbage collection, web requests, and data source usage

When we run our Spring Boot Web Application using CLI, Spring Boot Actuator automatically provides hostname as "**localhost**" and default port number as "**8080**". We can access this application using "**http://localhost:8080/**" end point.
We actually use HTTP Request methods like GET and POST to represent Management EndPoints using Spring Boot Actuator.

Spring Boot CLI installation and Hello World Example

The quickest way to get started with Spring Boot is to install the Spring Boot CLI so that you can start writing code.

What is Spring Boot CLI?

The Spring Boot CLI is a command line tool that can be used if we want to quickly develop with Spring. It allows us to run Groovy scripts, which means that we have a familiar Java-like syntax, without so much boilerplate code. We can also bootstrap a new project or write your own command for it.

It is Spring Boot software to run and test Spring Boot applications from command prompt. When we run Spring Boot applications using CLI, then it internally uses Spring Boot Starter and Spring Boot AutoConfigurate components to resolve all dependencies and execute the application.

It internally contains Groovy and Grape (JAR Dependency Manager) to add Spring Boot Defaults and resolve all dependencies automatically.

Installing the Spring Boot CLI

There are several ways to install the Spring Boot CLI:

- From a downloaded distribution
- Using the Groovy Environment Manager
- With OS X Homebrew
- As a port using MacPorts

MANUALLY INSTALLING THE SPRING BOOT CLI

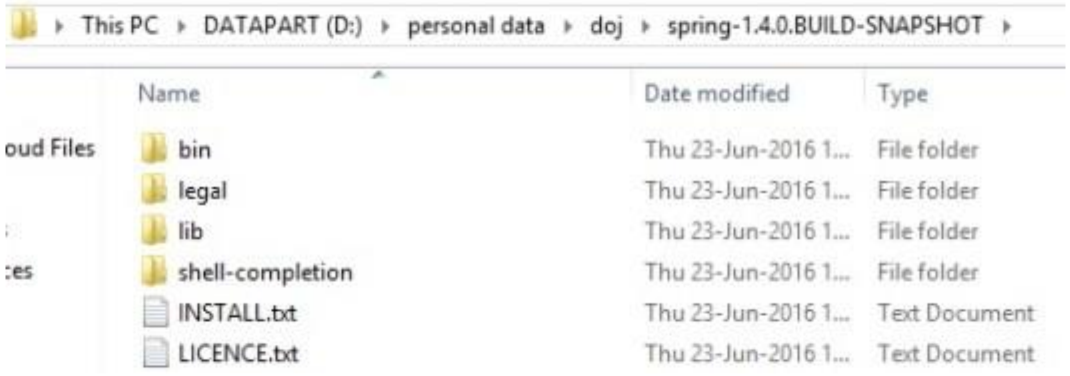
We can install Spring Boot CLI software using either Windows Installer or Zip file. Both approaches are easy to install and will give us same Spring Boot CLI software.

Step 1: You can download the Spring CLI distribution from the Spring software repository:

- **spring-boot-cli-1.4.0.BUILD-SNAPSHOT-bin.zip**
- **spring-boot-cli-1.4.0.BUILD-SNAPSHOT-bin.tar.gz**

Once downloaded, follow the INSTALL.txt instructions from the unpacked archive. In summary: there is a spring script (spring.bat for Windows) in a bin/ directory in the .zip file, or alternatively you can use java -jar with the .jar file (the script helps you to be sure that the classpath is set correctly).

Step 2: Extract **spring-boot-cli-1.4.0.BUILD-SNAPSHOT-bin.zip** file into our local FileSystem.


This PC › DATAPART (D:) › personal data › doj › spring-1.4.0.BUILD-SNAPSHOT ›
oud Files
:es

Name	Date modified	Type
bin	Thu 23-Jun-2016 1...	File folder
legal	Thu 23-Jun-2016 1...	File folder
lib	Thu 23-Jun-2016 1...	File folder
shell-completion	Thu 23-Jun-2016 1...	File folder
INSTALL.txt	Thu 23-Jun-2016 1...	Text Document
LICENCE.txt	Thu 23-Jun-2016 1...	Text Document

Step 3: Set Spring Boot CLI Environment Variables in Windows System as shown below.

```
set PATH= D:personal datadojspring-1.4.0.BUILD-SNAPSHOTbin
```

Step 4: Execute the below command to verify our installation process.

Using the CLI

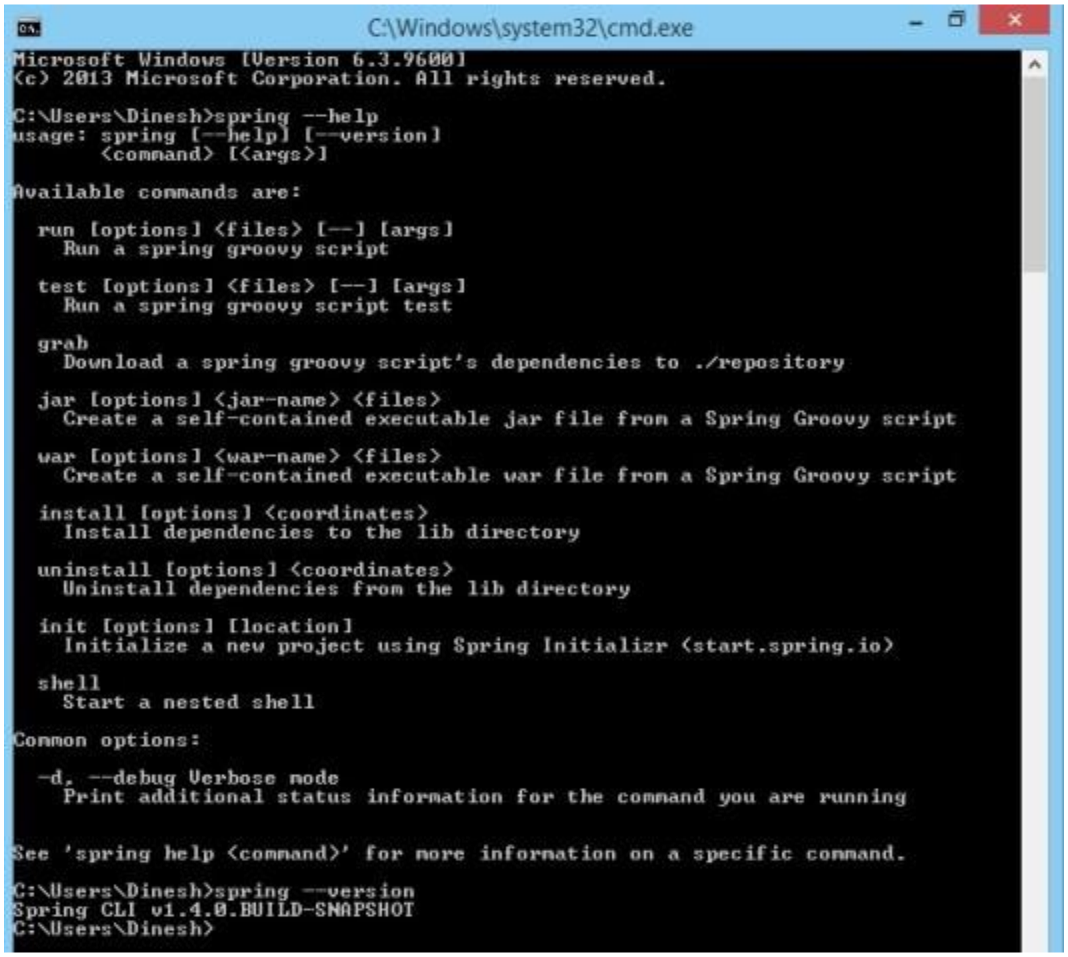
Once you have installed the CLI you can run it by typing spring.

We can use "spring –version" to know the Spring Boot CLI Version as shown below.

```
C:UsersDinesh>spring --version

Spring CLI v1.4.0.BUILD-SNAPSHOT
```

We can use “spring –help” to know the Spring Boot CLI Version as shown below.

```
C:UsersDinesh>spring --help

usage: spring [--help] [--version]

       <command> [<args>]


Available commands are:


  run [options] <files> [--] [args]

    Run a spring groovy script


  test [options] <files> [--] [args]

    Run a spring groovy script test

……..more command help is shown here
```



C:\Windows\system32\cmd.exe

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Dinesh>spring --help
usage: spring [--help] [--version]
       <command> [<args>]

Available commands are:

  run [options] <files> [--] [args]
    Run a spring groovy script

  test [options] <files> [--] [args]
    Run a spring groovy script test

  grab
    Download a spring groovy script's dependencies to ./repository

  jar [options] <jar-name> <files>
    Create a self-contained executable jar file from a Spring Groovy script

  war [options] <war-name> <files>
    Create a self-contained executable war file from a Spring Groovy script

  install [options] <coordinates>
    Install dependencies to the lib directory

  uninstall [options] <coordinates>
    Uninstall dependencies from the lib directory

  init [options] [location]
    Initialize a new project using Spring Initializr (start.spring.io)

  shell
    Start a nested shell

Common options:

  -d, --debug Verbose mode
    Print additional status information for the command you are running


See 'spring help <command>' for more information on a specific command.

C:\Users\Dinesh>spring --version
Spring CLI v1.4.0.BUILD-SNAPSHOT
C:\Users\Dinesh>
```

Now our Spring Boot CLI Installation process is done successfully.

Quick start Spring Boot CLI Hello World Example

Here we are using same example provided in Spring Boot Documentation.

Please follow the following steps to develop a Spring Boot HelloWorld Example:

Step 1: Create an “app” Folder in our Local FileSystem to place our groovy scripts.

Step 2: Create a file called app.groovy

```
@RestController

class app{

  @RequestMapping("/")

  String home() {

    "Hello World!"
    }

}
```

To compile and run the application type:

$ spring run app.groovy

To pass command line arguments to the application, you need to use a — to separate them from the “spring” command arguments, e.g.

$ spring run app.groovy — –server.port=9000

To set JVM command line arguments you can use the JAVA_OPTS environment variable, e.g.

$ JAVA_OPTS=-Xmx1024m spring run app.groovy

Observation on Code:

If we observe our **app.groovy**, we can find the following important points.

- No imports
- No other XML configuration to define Spring MVC Components like Views,ViewResolver etc.
- No web.xml and No DispatcherServlet declaration
- No build scripts to create our Application war file
- No need to build war file to deploy this application

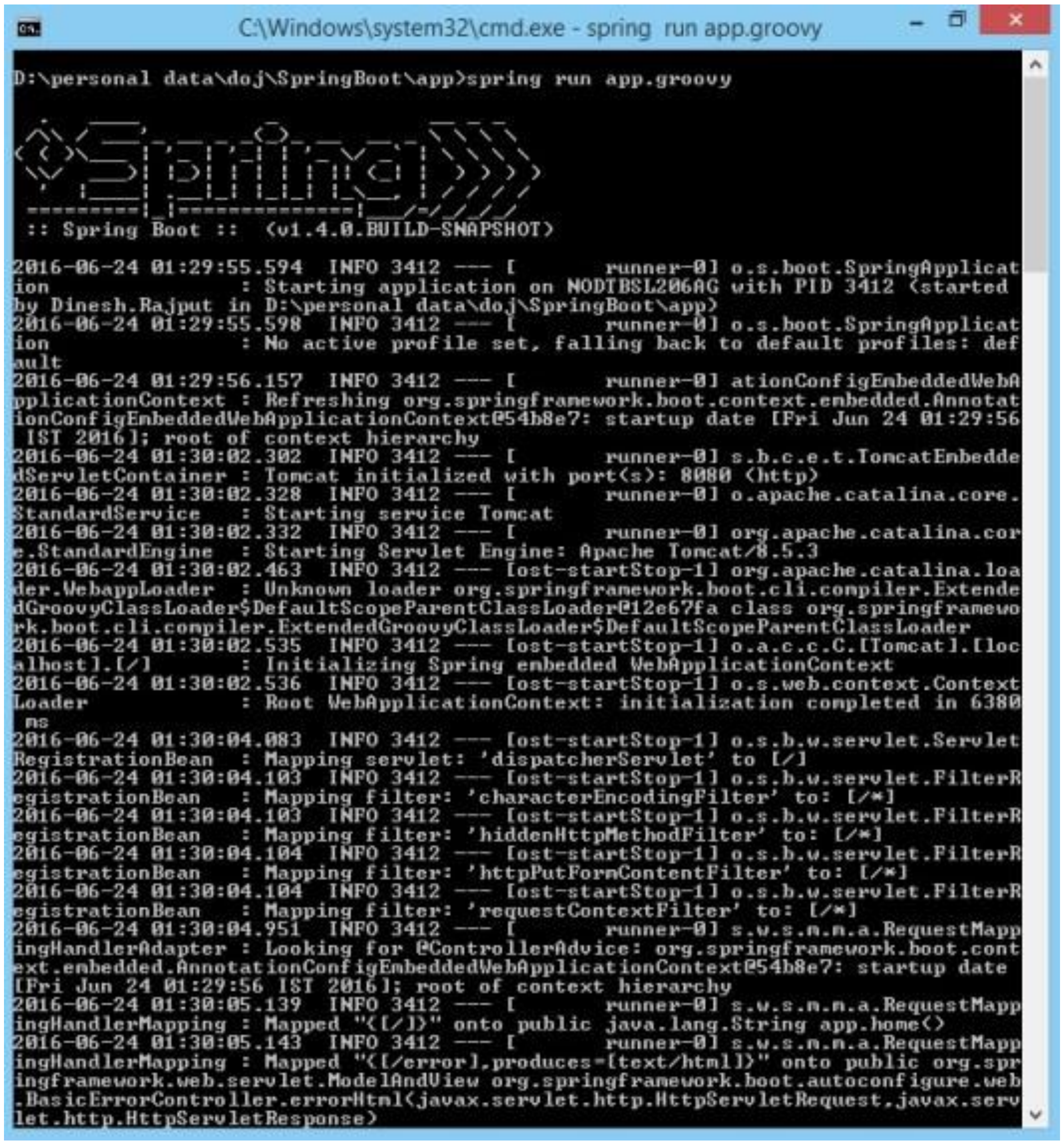
Now for running the application

Open command prompt at “**app**” Folder in our Local FileSystem.

Execute the following command

spring run app.groovy

Observe the output at “**spring run**” command console.

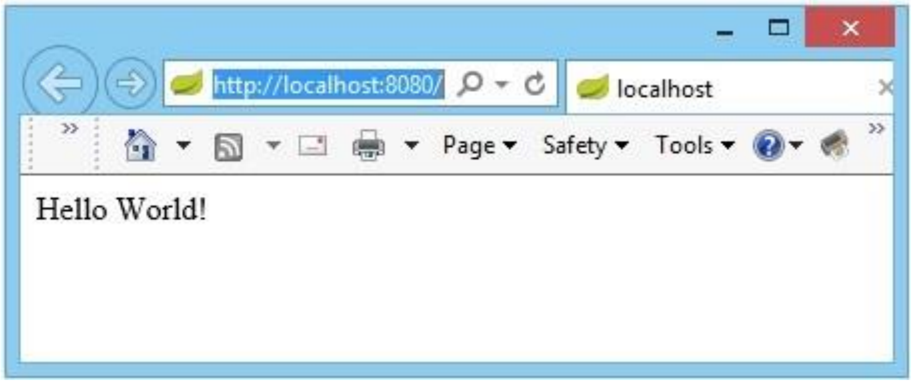


```
D:\personal data\doj\SpringBoot\app>spring run app.groovy

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v1.4.0.BUILD-SNAPSHOT)

2016-06-24 01:29:55.594  INFO 3412 --- [       runner-0] o.s.boot.SpringApplicat
ion                 : Starting application on NODTBSL206AG with PID 3412 (started
by Dinesh.Rajput in D:\personal data\doj\SpringBoot\app)
2016-06-24 01:29:55.598  INFO 3412 --- [       runner-0] o.s.boot.SpringApplicat
ion                 : No active profile set, falling back to default profiles: def
ault
2016-06-24 01:29:56.157  INFO 3412 --- [       runner-0] ationConfigEmbeddedWebA
pplicationContext : Refreshing org.springframework.boot.context.embedded.Annotat
ionConfigEmbeddedWebApplicationContext@54b8e7: startup date [Fri Jun 24 01:29:56
 IST 2016]; root of context hierarchy
2016-06-24 01:30:02.302  INFO 3412 --- [       runner-0] s.b.c.e.t.TomcatEmbedde
dServletContainer : Tomcat initialized with port(s): 8080 (http)
2016-06-24 01:30:02.328  INFO 3412 --- [       runner-0] o.apache.catalina.core.
StandardService   : Starting service Tomcat
2016-06-24 01:30:02.332  INFO 3412 --- [       runner-0] org.apache.catalina.cor
e.StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.5.3
2016-06-24 01:30:02.463  INFO 3412 --- [ost-startStop-1] org.apache.catalina.loa
der.WebappLoader  : Unknown loader org.springframework.boot.cli.compiler.Extende
dGroovyClassLoader$DefaultScopeParentClassLoader@12e67fa class org.springframewo
rk.boot.cli.compiler.ExtendedGroovyClassLoader$DefaultScopeParentClassLoader
2016-06-24 01:30:02.535  INFO 3412 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[loc
alhost].[/]       : Initializing Spring embedded WebApplicationContext
2016-06-24 01:30:02.536  INFO 3412 --- [ost-startStop-1] o.s.web.context.Context
Loader            : Root WebApplicationContext: initialization completed in 6380
 ms
2016-06-24 01:30:04.083  INFO 3412 --- [ost-startStop-1] o.s.b.w.servlet.Servlet
RegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
2016-06-24 01:30:04.103  INFO 3412 --- [ost-startStop-1] o.s.b.w.servlet.FilterR
egistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2016-06-24 01:30:04.103  INFO 3412 --- [ost-startStop-1] o.s.b.w.servlet.FilterR
egistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2016-06-24 01:30:04.104  INFO 3412 --- [ost-startStop-1] o.s.b.w.servlet.FilterR
egistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*]
2016-06-24 01:30:04.104  INFO 3412 --- [ost-startStop-1] o.s.b.w.servlet.FilterR
egistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2016-06-24 01:30:04.951  INFO 3412 --- [       runner-0] s.w.s.m.m.a.RequestMapp
ingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.cont
ext.embedded.AnnotationConfigEmbeddedWebApplicationContext@54b8e7: startup date
[Fri Jun 24 01:29:56 IST 2016]; root of context hierarchy
2016-06-24 01:30:05.139  INFO 3412 --- [       runner-0] s.w.s.m.m.a.RequestMapp
ingHandlerMapping : Mapped "{[/]}" onto public java.lang.String app.home()
2016-06-24 01:30:05.143  INFO 3412 --- [       runner-0] s.w.s.m.m.a.RequestMapp
ingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.spr
ingframework.web.servlet.ModelAndView org.springframework.boot.autoconfigure.web
.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,javax.serv
let.http.HttpServletResponse)
```

If we observe here, when we execute "***spring run app.groovy***", it starts Embedded Tomcat server at Default port number: **8080**.

Now our Spring Boot **HelloWorld** Example application is up and running. It's time to test it now.

- Open browser and access the following link.
- Access this URL: ***http://localhost:8080/***



Hello World!

In the above example we can observe following points.

- There is no Import
- There is no XML or JAVA configuration
- There is no DispatcherServlet and web.xml
- There is no build file for creating war like Maven or Gradle

It is the responsibility of Spring Boot Core Components, Groovy Compiler (groovyc) and Groovy Grape (Groovy's JAR Dependency Manager).

Spring Boot Components uses Groovy Compiler and Groovy Grape to provide some Defaults lime adding required imports, providing required configuration, resolving jar dependencies, adding main() method etc. As a Spring Boot Developer, We don't need to worry all these things. Spring Boot Framework will take care of all these things for us.

That's the beauty of Spring Boot Framework.

Default import statements

To help reduce the size of your Groovy code, several import statements are automatically included. Notice how the example above refers to @Component,@RestController and @RequestMapping without needing to use fully-qualified names or import statements.

Automatic main method

Unlike the equivalent Java application, you do not need to include a public static void main(String[] args) method with your Groovy scripts. ASpringApplication is automatically created, with your compiled code acting as the source.

Summary

The Spring Boot CLI takes the simplicity offered by Spring Boot auto-configuration and starter dependencies and turns it up a notch. Using the elegance of the Groovy language, the CLI makes it possible to develop Spring applications with minimal code noise.

Spring Boot Initializr Web Interface and Examples

Hello friends lets discuss another important components Spring Boot Initializr of Spring Boot, it is a quick way to create spring boot project structure. In this chapter we are going to explore about a web interface which create Spring Boot Application online.

Related tutorials previously we have discussed

- **Introduction of Spring Boot**
- **Key Components of Spring Boot**
- **Spring Boot CLI**

What is Spring Boot Initializr?

The Spring Initializr is ultimately a **web application** (at "**http://start.spring.io/**") that can generate a Spring Boot project structure for you. It doesn't generate any application code, but it will give you a basic project structure and either a **Maven** or a **Gradle** build specification to build your code with.

Spring Initializr can be used in several ways:

- Through a web-based interface
- Via Spring Tool Suite
- Via IntelliJ IDEA
- Using the Spring Boot CLI

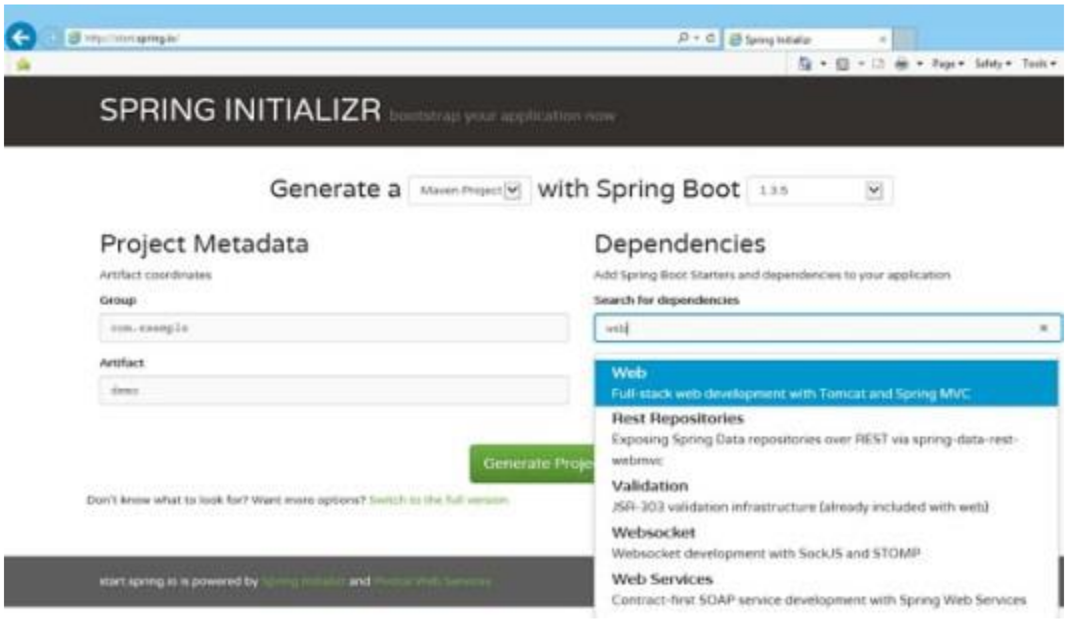
Why we need Spring Boot Initializr?

Sometimes the hardest part of a project is getting started. You need to set up a directory structure for various project artifacts, create a build file, and populate the build file with dependencies. The **Spring Boot CLI** removes much of this setup work, but if you favor a more traditional Java project structure, you'll want to look at the Spring Initializr.

USING SPRING INITIALIZR'S WEB INTERFACE

Now we are starting with the web-based interface.

The Spring Team has provided a Web Interface for Spring Boot Initializr at "**http://start.spring.io/**". We can use it to create our new Project's base structure for Maven/Gradle build tools.



SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Spring Boot 1.3.5

Project Metadata

Artifact coordinates

Group: com.example

Artifact: demo

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies: web

Web
Full-stack web development with Tomcat and Spring MVC

Rest Repositories
Exposing Spring Data repositories over REST via spring-data-rest-webmvc

Validation
JSR-303 validation infrastructure (already included with web)

Websocket
Websocket development with SockJS and STOMP

Web Services
Contract-first SOAP service development with Spring Web Services

Don't know what to look for? Want more options? Switch to the full version.

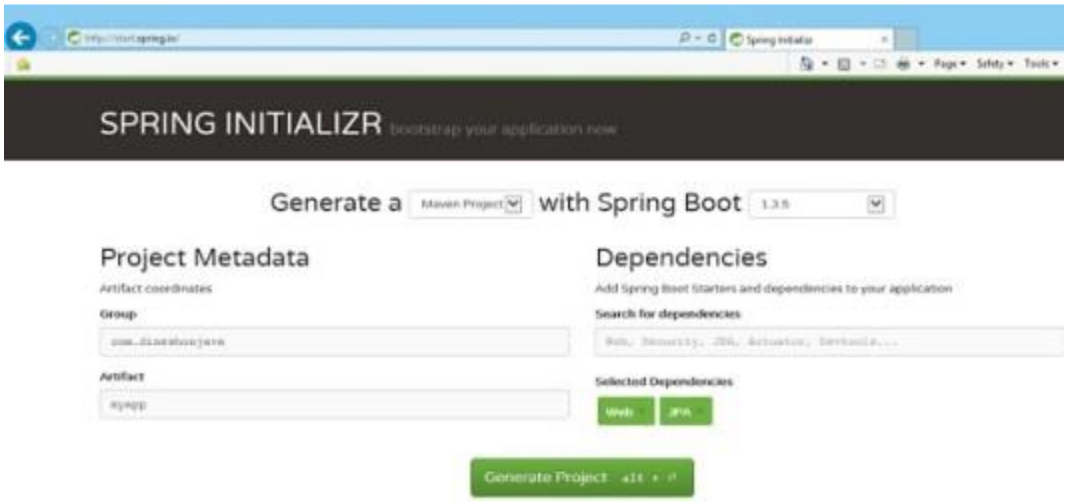
start.spring.io is powered by Spring Initializr and Pivotal Web Services

Creating Maven Example with Spring Boot Initializr Web Interface

There are following steps to create new Spring Boot Web Application for Maven Build tool and Spring STS Suite IDE.

Step 1: Go to Spring Boot Initializr at “**http://start.spring.io/**”.

Step 2: Once you’ve filled in the form and made your dependency selections, click the Generate Project button to have Spring Initializr generate a project for you.



SPRING INITIALIZR bootstrap your application now

Generate a Maven Project with Spring Boot 1.3.5

Project Metadata

Artifact coordinates

Group: com.dineshonjava

Artifact: myapp

Dependencies

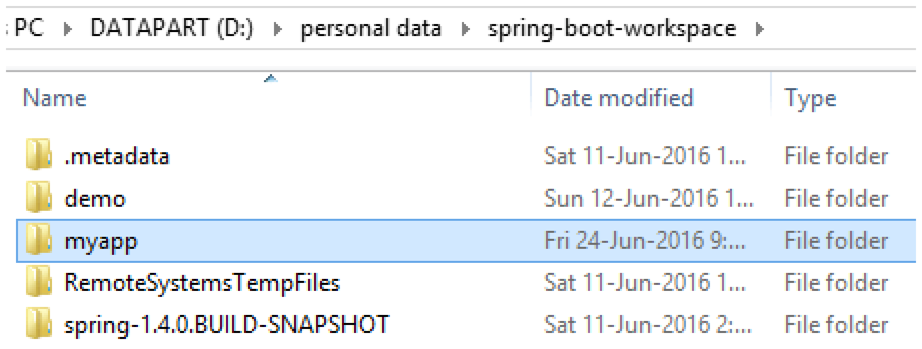
Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Dependencies: Web, JPA

Step 3: Now click on “**Generate Project**” Button, it creates and downloads a Maven Project as “**myapp.zip**” file into our local file system.

Step 4: Move "**myapp.zip**" to our Spring STS Suite Workspace and Extract this zip file

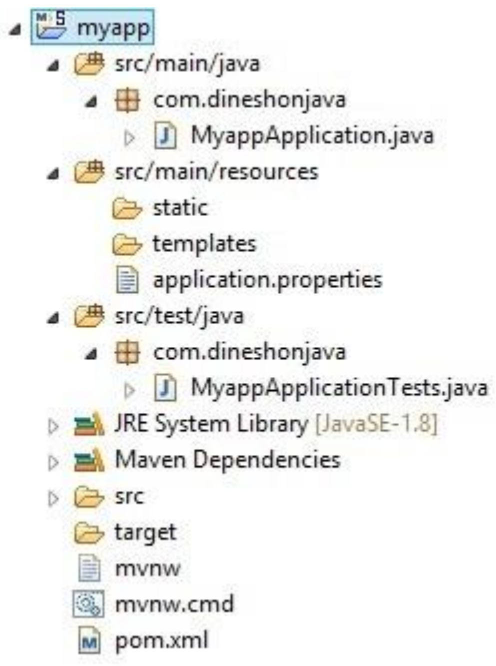


PC ▸ DATAPART (D:) ▸ personal data ▸ spring-boot-workspace ▸

Name	Date modified	Type
.metadata	Sat 11-Jun-2016 1...	File folder
demo	Sun 12-Jun-2016 1...	File folder
myapp	Fri 24-Jun-2016 9:...	File folder
RemoteSystemsTempFiles	Sat 11-Jun-2016 1...	File folder
spring-1.4.0.BUILD-SNAPSHOT	Sat 11-Jun-2016 2:...	File folder

Step 5: Import this "myapp" Maven project into Spring STS IDE.

Step 6: We'd have a project structure similar.



```
myapp
  src/main/java
    com.dineshonjava
      MyappApplication.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
    com.dineshonjava
      MyappApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

If you observe this project files, it generates **pom.xml** file, two Spring Boot Java files and one JUnit Java file.

```
<?xml version="1.0" encoding="UTF-8"?>

<project                xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
 <modelVersion>4.0.0</modelVersion>


 <groupId>com.dineshonjava</groupId>

 <artifactId>myapp</artifactId>

 <version>0.0.1-SNAPSHOT</version>

 <packaging>jar</packaging>


 <name>myapp</name>

 <description>Demo project for Spring Boot</description>


 <parent>

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

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

  <version>1.3.5.RELEASE</version>

  <relativePath/> <!-- lookup parent from repository -->

 </parent>


 <properties>

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

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

 </properties>


 <dependencies>
```

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

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

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



</project>
```

<u>MyappApplication.java</u>

```
package com.dineshonjava;


import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication

public class MyappApplication {


 public static void main(String[] args) {

  SpringApplication.run(MyappApplication.class, args);

 }

}
```

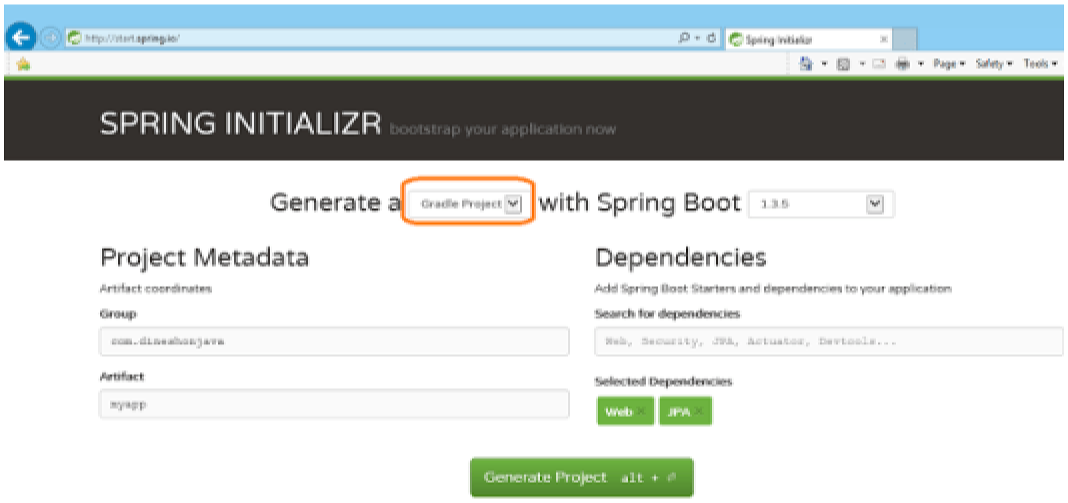
<u>MyappApplicationTests.java</u>

```
package com.dineshonjava;
```

```
import org.junit.Test;

import org.junit.runner.RunWith;

import
org.springframework.boot.test.SpringApplicationConfiguration;

import
org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import org.springframework.test.context.web.WebAppConfiguration;



@RunWith(SpringJUnit4ClassRunner.class)

@SpringApplicationConfiguration(classes = MyappApplication.class)

@WebAppConfiguration

public class MyappApplicationTests {



 @Test

 public void contextLoads() {

 }



}
```

Creating Gradle Example with Spring Boot Initializr Web Interface

All steps for creating gradle project with Spring Boot Initializr is same as like Maven Project as we have created above unlike select **Gradle Project** instead of **Maven Project** in Spring Boot Initializr Web Interface.


SPRING INITIALIZR bootstrap your application now

Generate a Gradle Project with Spring Boot 1.3.5

Project Metadata

Artifact coordinates

Group: com.dineshonjava

Artifact: myapp

Dependencies

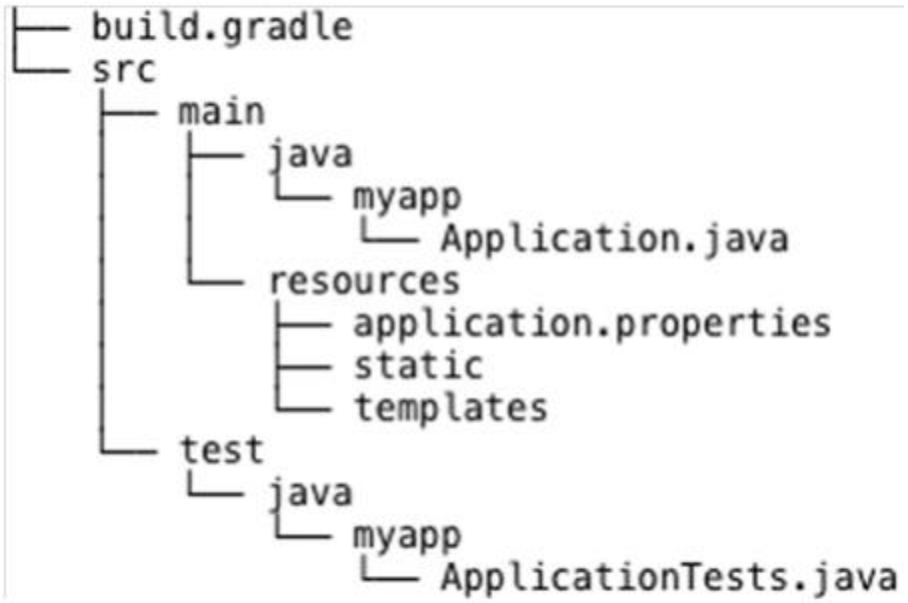
Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Dependencies: Web, JPA

Generate Project alt + ⏎

All source code files name and structure of project is also same as like maven project unlike creating build file name **build.gradle** instead of **pom.xml**


```
├── build.gradle
└── src
    ├── main
    │   ├── java
    │   │   └── myapp
    │   │       └── Application.java
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── java
            └── myapp
                └── ApplicationTests.java
```

build.gradle

```
buildscript {

 ext {

  springBootVersion = '1.3.5.RELEASE'

 }

 repositories {
  mavenCentral()
 }
 dependencies {
  classpath("org.springframework.boot:spring-boot-gradle-
plugin:${springBootVersion}")
 }
}

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

jar {
 baseName = 'mygradleapp'
 version = '0.0.1-SNAPSHOT'
}
sourceCompatibility = 1.8
targetCompatibility = 1.8

repositories {
 mavenCentral()
}
```

```
dependencies {

 compile('org.springframework.boot:spring-boot-starter-data-
jpa')

 compile('org.springframework.boot:spring-boot-starter-web')

 testCompile('org.springframework.boot:spring-boot-starter-
test')

}




eclipse {

 classpath {

   containers.remove('org.eclipse.jdt.launching.JRE_CONTAINER')

   containers
'org.eclipse.jdt.launching.JRE_CONTAINER/org.eclipse.jdt.interna
l.debug.ui.launcher.StandardVMType/JavaSE-1.8'

 }

}
```

As you can see, there’s very little code in this project. Aside from a couple of empty directories, it also includes the following:

- **build.gradle**—A Gradle build specification. Had you chosen a Maven project, this would be replaced with pom.xml.
- **Application.java**—A class with a main() method to bootstrap the application.
- **ApplicationTests.java**— an empty JUnit test class instrumented to load a Spring application context using Spring Boot auto-configuration.
- **application.properties**—an empty properties file for you to add configuration properties to as you see fit.

Summary
Whether you use Initializr's web-based interface, create your projects from Spring Tool Suite, or use the Spring Boot CLI to initialize a project, projects created using the Spring Boot Initializr have a familiar project layout, not unlike other Java projects you may have developed before.
Spring Boot is an exciting new way to develop Spring applications with minimal friction from the framework itself. Auto-configuration eliminates much of the boilerplate configuration that infests traditional Spring applications. Spring Boot starters enable you to specify build dependencies by what they offer rather than use explicit library names and version. The Spring Boot CLI takes Spring Boot's frictionless development model to a whole new level by enabling quick and easy development with Groovy from the command line. And the Actuator lets you look inside your running application to see what and how Spring Boot has done.
Happy Spring Boot Learning!!

Spring Boot Initializr with IDEs (via Spring Tool Suite)

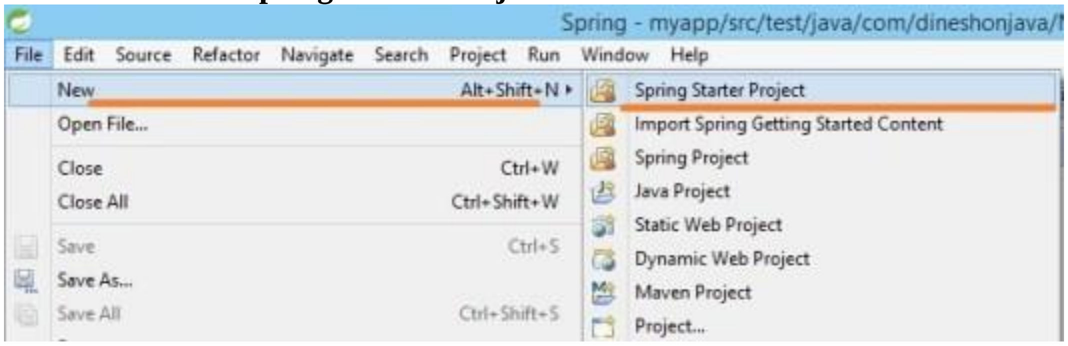
Hello friends lets discuss another important components Spring Boot Initializr of Spring Boot, it is a quick way to create spring boot project structure. In this article we are going to explore about Spring Boot Initializr with STS IDE. Spring Boot Initializr is used to quick start new Spring Boot Maven/Gradle projects within no time. It generates initial project structure and builds scripts to reduce Development time.Spring Tool Suite3 has long been a fantastic IDE for developing Spring applications. Since version 3.4.0 it has also been integrated with the Spring Initializr, making it a great way to get started with Spring Boot.

Related tutorials previously we have discussed

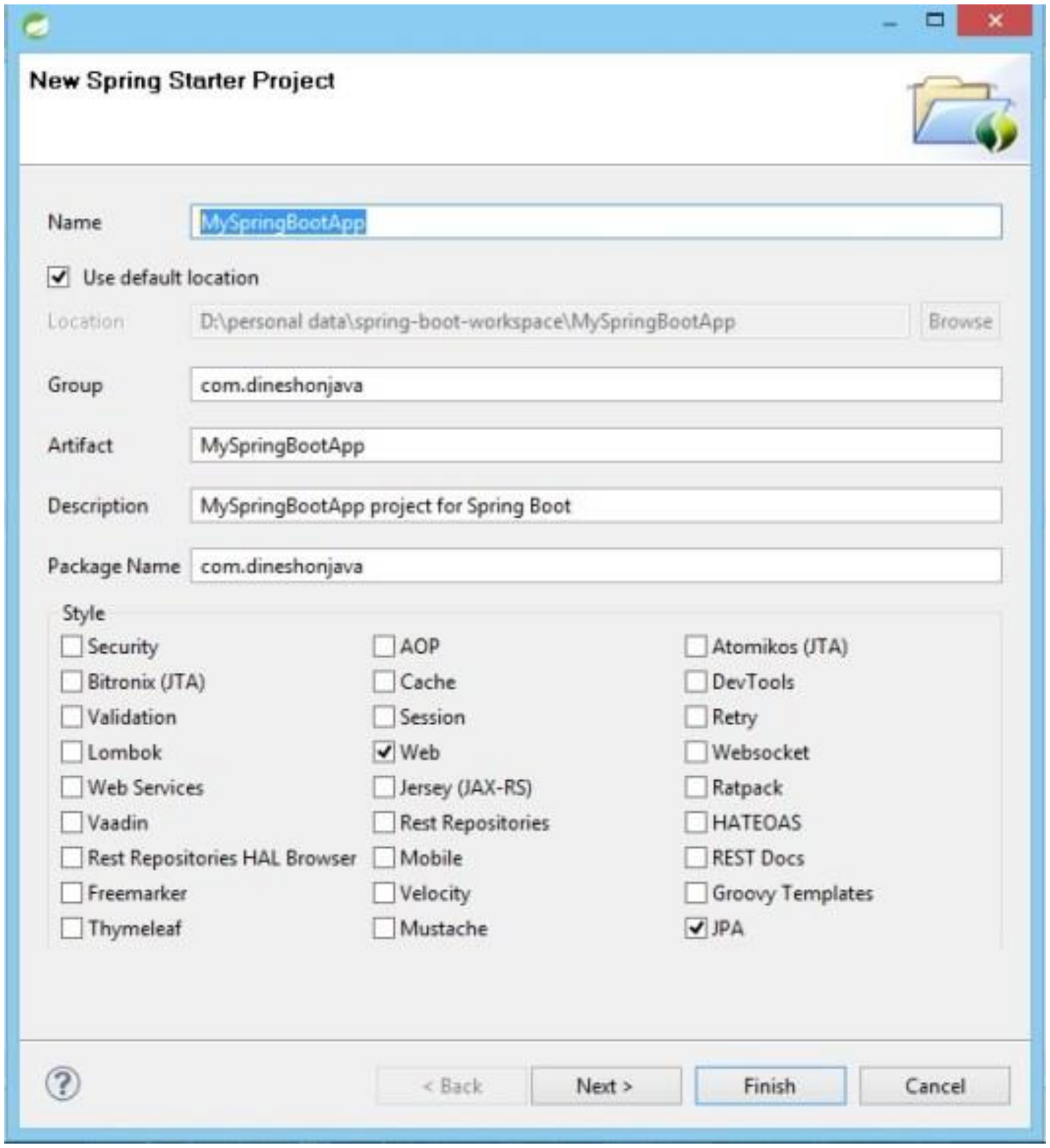
- **Introduction of Spring Boot**
- **Key Components of Spring Boot**
- **Spring Boot CLI**
- **Spring Boot Initializr Web Interface**

Create a new Spring Boot application in Spring Tool Suite

Step 1: Select the **New** > **Spring Starter Project** menu item from the **File** menu.



Step 2: We will get the following "**Spring Starter Project**" Wizard to provide our project related information.



Name: MySpringBootApp

[✓] Use default location

Location: D:\personal data\spring-boot-workspace\MySpringBootApp

Group: com.dineshonjava

Artifact: MySpringBootApp

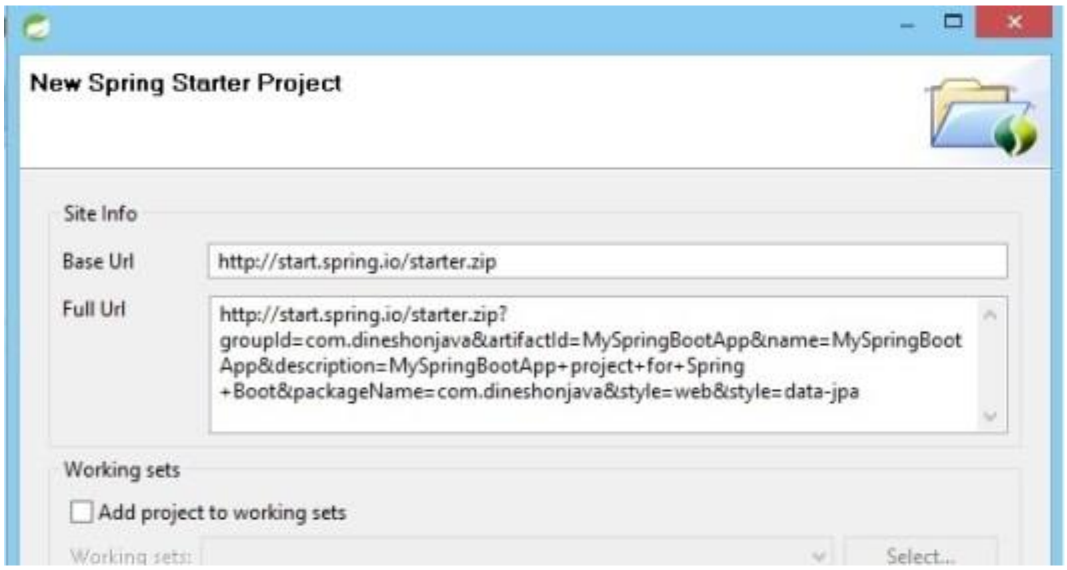
Description: MySpringBootApp project for Spring Boot

Package Name: com.dineshonjava

Style

☐ Security	☐ AOP	☐ Atomikos (JTA)
☐ Bitronix (JTA)	☐ Cache	☐ DevTools
☐ Validation	☐ Session	☐ Retry
☐ Lombok	✓ Web	☐ Websocket
☐ Web Services	☐ Jersey (JAX-RS)	☐ Ratpack
☐ Vaadin	☐ Rest Repositories	☐ HATEOAS
☐ Rest Repositories HAL Browser	☐ Mobile	☐ REST Docs
☐ Freemarker	☐ Velocity	☐ Groovy Templates
☐ Thymeleaf	☐ Mustache	✓ JPA

Step 3: Please provide our Spring MVC Maven Web Application Project details as shown below and Click on "**Next**" Button


New Spring Starter Project

Site Info

Base Url: http://start.spring.io/starter.zip

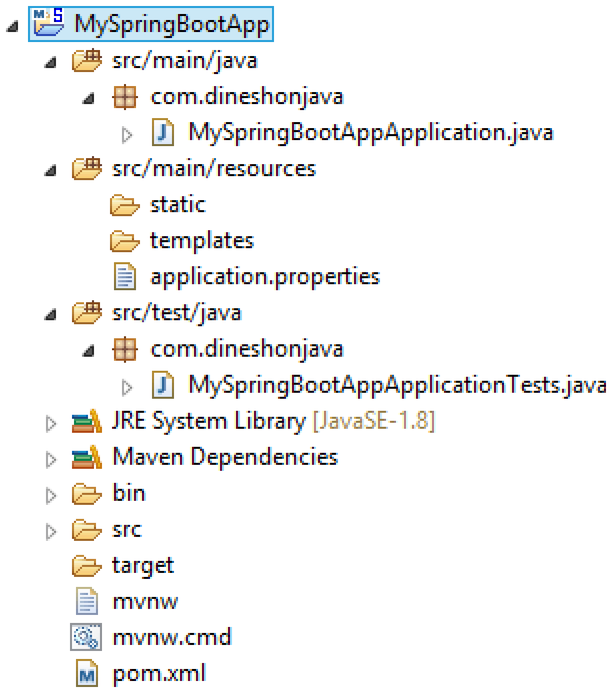
Full Url: http://start.spring.io/starter.zip?groupId=com.dineshonjava&artifactId=MySpringBootApp&name=MySpringBootApp&description=MySpringBootApp+project+for+Spring+Boot&packageName=com.dineshonjava&style=web&style=data-jpa

Working sets

☐ Add project to working sets

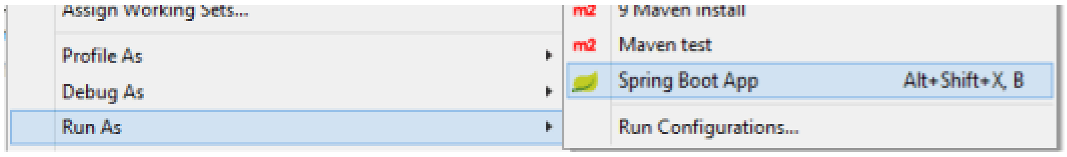
Working sets:

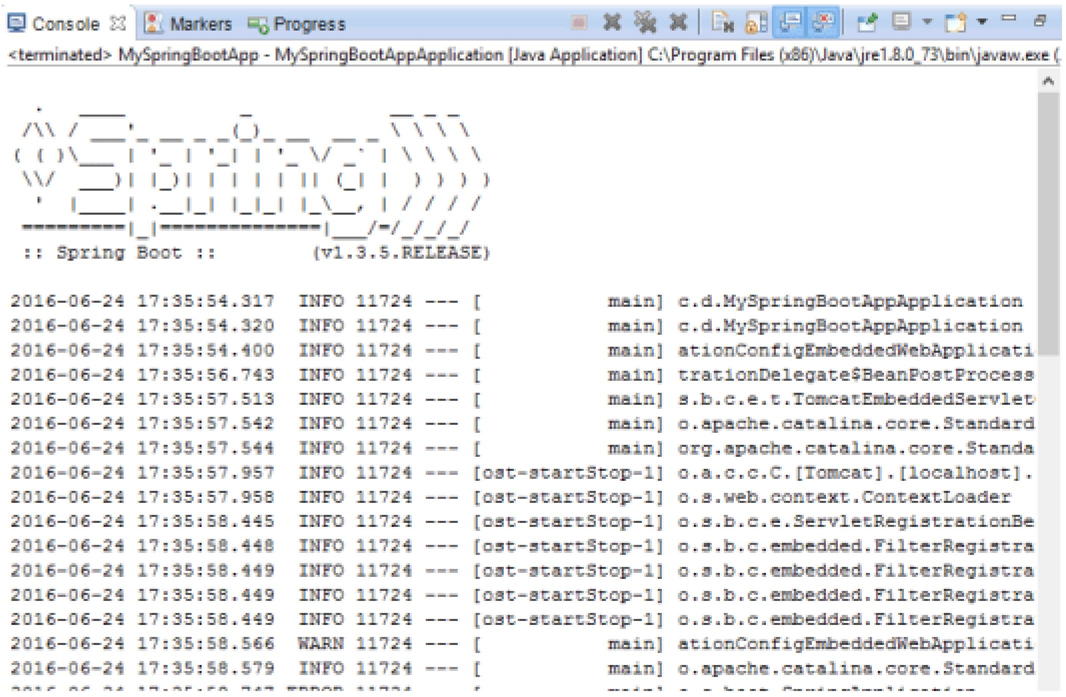
Step 4: Click on "**Finish**" button to create our new Spring Boot Project.
Step 5: Now Spring STS Suite creates a **Maven Project** and downloads all required Jars to our Maven Local Repository.



```
MySpringBootApp
  src/main/java
    com.dineshonjava
      MySpringBootAppApplication.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
    com.dineshonjava
      MySpringBootAppApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  bin
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

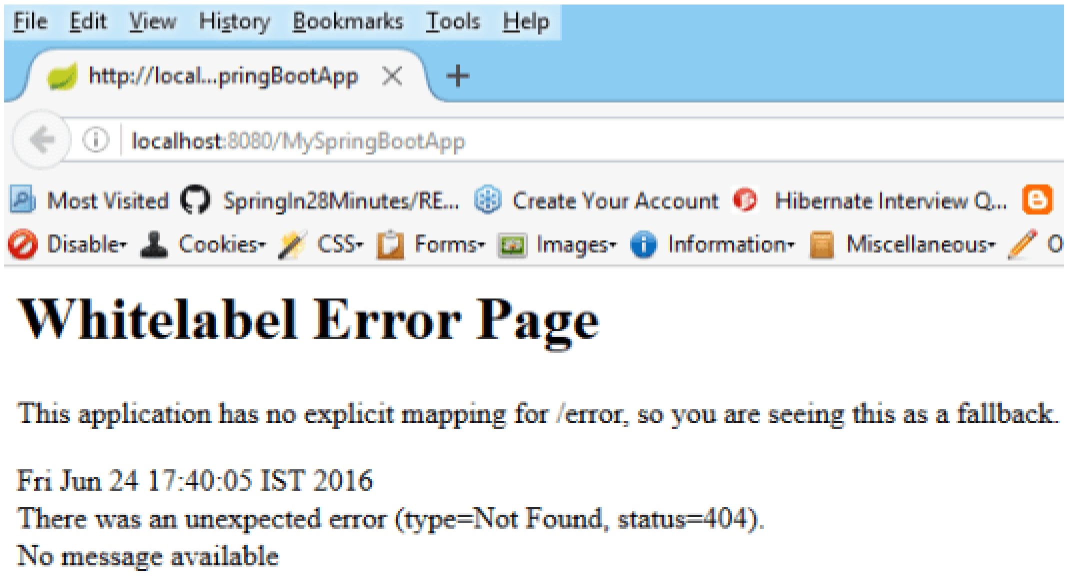
Step 6: Once the project has been imported into your workspace, you're ready to start developing your application.
Step 7: Execute Spring Boot Application **Run As > Spring Boot Application** from the **Run** menu.





```
<terminated> MySpringBootApp - MySpringBootAppApplication [Java Application] C:\Program Files (x86)\Java\jre1.8.0_73\bin\javaw.exe (

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.3.5.RELEASE)

2016-06-24 17:35:54.317  INFO 11724 --- [           main] c.d.MySpringBootAppApplication
2016-06-24 17:35:54.320  INFO 11724 --- [           main] c.d.MySpringBootAppApplication
2016-06-24 17:35:54.400  INFO 11724 --- [           main] ationConfigEmbeddedWebApplicati
2016-06-24 17:35:56.743  INFO 11724 --- [           main] trationDelegate$BeanPostProcess
2016-06-24 17:35:57.513  INFO 11724 --- [           main] s.b.c.e.t.TomcatEmbeddedServlet
2016-06-24 17:35:57.542  INFO 11724 --- [           main] o.apache.catalina.core.Standard
2016-06-24 17:35:57.544  INFO 11724 --- [           main] org.apache.catalina.core.Standa
2016-06-24 17:35:57.957  INFO 11724 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].
2016-06-24 17:35:57.958  INFO 11724 --- [ost-startStop-1] o.s.web.context.ContextLoader
2016-06-24 17:35:58.445  INFO 11724 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBe
2016-06-24 17:35:58.448  INFO 11724 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistra
2016-06-24 17:35:58.449  INFO 11724 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistra
2016-06-24 17:35:58.449  INFO 11724 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistra
2016-06-24 17:35:58.449  INFO 11724 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistra
2016-06-24 17:35:58.566  WARN 11724 --- [           main] ationConfigEmbeddedWebApplicati
2016-06-24 17:35:58.579  INFO 11724 --- [           main] o.apache.catalina.core.Standard
```

Step 8: Access our Spring MVC application with "**http://localhost:8080/MySpringBootApp**" and observe the results



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Fri Jun 24 17:40:05 IST 2016
There was an unexpected error (type=Not Found, status=404).
No message available

SpringApplication:

The **SpringApplication** class provides a convenient way to bootstrap a Spring application that will be started from a**main()** method. In many situations you can just delegate to the static **SpringApplication.run** method:

- **SpringApplication** is one of the Spring Boot API classes.

- **SpringApplication** class is used to bootstrap a Spring application that will be started from a main() method

```
package com.dineshonjava;



import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;



@SpringBootApplication

public class MySpringBootAppApplication {



 public static void main(String[] args) {

  SpringApplication.run(MySpringBootAppApplication.class, args);

 }

}
```

MySpringBootAppApplication class is annotated with **@SpringBootApplication** annotation. **@SpringBootApplication** does the following things:

- Because of **@Configuration** annotation, It scans for **@Bean** methods to create beans.
- Because of **@ComponentScan** annotation, It does component scanning (Components means Beans annotated with **@Component,@Service,@Repository,@Controller** etc).
- Because of **@EnableAutoConfiguration** annotation, It triggers Spring Boot Auto-Configuration.

When we run **MySpringBootAppApplication** class **main()** method, it make a calls to "**SpringApplication.run()**" method. Then this call done following things

- This call is used to create "**AnnotationConfigEmbeddedWebApplicationContext**".
- This "**AnnotationConfigEmbeddedWebApplicationContext**" instance is used to create an instance of "**TomcatEmbeddedServletContainerFactory**" class.
- This "**TomcatEmbeddedServletContainerFactory**" is used to create an instance of "**TomcatEmbeddedServletContainer**" class.
- "**TomcatEmbeddedServletContainer**" instance starts a Tomcat Container at default port number: **8080** and deploys our Spring Boot **WebApplication**.

Summary

Congratulation!!! Here we have learned how to create Spring Boot Application with Spring Boot Intilizr via STS IDE. And also discussed code flow of run this spring boot application.

Spring Boot Initializr with Spring Boot CL

Hello friends lets discuss another way to create spring boot project structure by using Spring Boot Initializr. Spring Boot Initializr is used to quick start new Spring Boot Maven/Gradle projects within no time. It generates initial project structure and builds scripts to reduce Development time.

As previous chapter we have seen following ways for Spring Boot Initialzr.

- **Spring Boot Initilizr through Web Interface**
- **Spring Boot Initilizr through IDEs/IDE Plugins**
- **Spring Boot Initilizr through Spring Boot CLI**

Before starting discussion for this tutorial we have to go through my previous chapter (**Spring Boot CLI installation and Hello World Example**) for Basic and Installation guide for Spring Boot CLI. As in previous chapter we saw, the Spring Boot CLI is a great way to develop Spring applications by just writing code. However, the Spring Boot CLI also has a few commands that can help us use the Initializr to kick-start development on a more traditional Java project.

Spring Boot CLI provides a "**spring init**" command to bootstrap Spring Applications.

Syntax for Init Command:

spring init [options] [location]

Here "**options**" are command options and "**location**" is specified our file system location to create new Spring Boot Project.

Examples for Spring Init Command:

Example 1: Use default setting

$ spring init

After contacting the Initializr web application, the init command will conclude by downloading a **demo.zip** file with following default setting.

- Default Build tool is "**maven**".
- Default Spring Initilizr service target URL: ***https://start.spring.io***
- Default project name: "**demo**"
- Default maven type: "**jar**"

- Default only baseline starter dependencies for **Spring Boot** and **testing**

Example 2: Using required dependencies on web project

Let's say you want to start out by building a web application that uses JPA for data persistence and that's secured with Spring Security. You can specify those initial dependencies with either **–dependencies** or **-d**:

$ spring init –dweb, jpa, security

This will give you a demo.zip containing the same project structure as before, but with Spring Boot's web, JPA, and security starters expressed as dependencies in **pom.xml**.

Example 3: Using required dependencies on web project with change Build (Maven to Gradle)

For Gradle

$ spring init –dweb, jpa, security –build gradle

For Maven

$ spring init –dweb, jpa, security –build maven (maven is default build type)

Example 4: Change packaging

By default, the build specification for both Maven and Gradle builds will produce an executable JAR file. If you'd rather produce a WAR file, you can specify so with the **–packaging** or **-p** parameter:

$ spring init –dweb, jpa, security –build gradle –p war

Example 5: Change application name

$ spring init –dweb, jpa, security –build gradle –p war myapp.zip

Example 6: Specifying the Java version to compile with, and selecting a version of Spring Boot

$ spring init –dweb, jpa, security –build gradle –java-version=1.7 –boot-version=1.3.0.RELEASE –p war myapp.zip

Other more commands

We can discover all of the parameters by using the help command:

$ spring help init

For lists several parameters that are supported by the Initializr.

spring init -l

Summary

Whether we use Initializr's web-based interface, create our projects from Spring Tool Suite, or use the Spring Boot CLI to initialize a project, projects created using the Spring Boot Initializr have a familiar project layout, not unlike other Java projects we may have developed before.

Installing Spring Boot

You can use Spring Boot in the same way as any standard Java library. Simply include the appropriate **spring-boot-*.jar** files on your classpath. Spring Boot does not require any special tools integration, so you can use any IDE or text editor; and there is nothing special about a Spring Boot application, so you can run and debug as you would any other Java program.
You can also add simply all spring boot jar files to classpath of application but we are recommending use any build tools (**Maven**/**Gradle**) for dependencies management.
Maven Installation:
Spring Boot is compatible with Apache Maven 3.2 or above. If you don't already have Maven installed you can follow the instructions at **How to Install Maven in Windows**. Ubuntu users can run ***sudo apt-get install maven***.
Spring Boot dependencies use the **org.springframework.boot** groupId. Typically your Maven POM file will inherit from the **spring-boot-starter-parent** project and declare dependencies to one or more "**Starters**".
Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<project                xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>



    <groupId>com.example</groupId>

    <artifactId>myproject</artifactId>

    <version>0.0.1-SNAPSHOT</version>



    <!-- Inherit defaults from Spring Boot -->

    <parent>

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

        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.4.0.BUILD-SNAPSHOT</version>

    </parent>


    <!-- Add typical dependencies for a web application -->

    <dependencies>

        <dependency>

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

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

        </dependency>

    </dependencies>


    <!-- Package as an executable jar -->

    <build>

        <plugins>

            <plugin>

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

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

            </plugin>

        </plugins>

    </build>


    <!-- Add Spring repositories -->
    <!-- (you don't need this if you are using a .RELEASE version)
-->

    <repositories>

        <repository>

            <id>spring-snapshots</id>

            <url>http://repo.spring.io/snapshot</url>

            <snapshots><enabled>true</enabled></snapshots>

        </repository>

        <repository>

            <id>spring-milestones</id>

            <url>http://repo.spring.io/milestone</url>

        </repository>

    </repositories>

    <pluginRepositories>

        <pluginRepository>

            <id>spring-snapshots</id>

            <url>http://repo.spring.io/snapshot</url>

        </pluginRepository>

        <pluginRepository>

            <id>spring-milestones</id>

            <url>http://repo.spring.io/milestone</url>

        </pluginRepository>

    </pluginRepositories>
```

```
</project>
```

Gradle installation:
Spring Boot is compatible with Gradle 1.12 or above. If you don't already have Gradle installed you can follow the instructions at How to Install Gradle. Spring Boot dependencies can be declared using the **org.springframework.boot** group. Typically your project will declare dependencies to one or more "**Starters**". Spring Boot provides a useful Gradle plugin that can be used to simplify dependency declarations and to create executable jars.
build.gradle

```
buildscript {

    repositories {

        jcenter()

        maven { url "http://repo.spring.io/snapshot" }

        maven { url "http://repo.spring.io/milestone" }

    }

    dependencies {

        classpath("org.springframework.boot:spring-boot-gradle-
plugin:1.4.0.BUILD-SNAPSHOT")

    }

}



apply plugin: 'java'

apply plugin: 'spring-boot'



jar {

    baseName = 'myproject'

    version =  '0.0.1-SNAPSHOT'
}


repositories {

    jcenter()

    maven { url "http://repo.spring.io/snapshot" }

    maven { url "http://repo.spring.io/milestone" }

}


dependencies {

    compile("org.springframework.boot:spring-boot-starter-web")

    testCompile("org.springframework.boot:spring-boot-starter-
test")

}
```

Installing the Spring Boot CLI:

The Spring Boot CLI is a command line tool that can be used if you want to quickly prototype with Spring. It allows you to run Groovy scripts, which means that you have a familiar Java-like syntax, without so much boilerplate code.
You can follow the instructions at **Spring Boot CLI installation** to setup the Spring Boot CLI.
In the coming tutorial we will create the first Spring Boot Application.

Developing your first Spring Boot application Hello World

July 14, 2016

NO COMMENTS

Developing your first Spring Boot application Hello World

Previous

Next

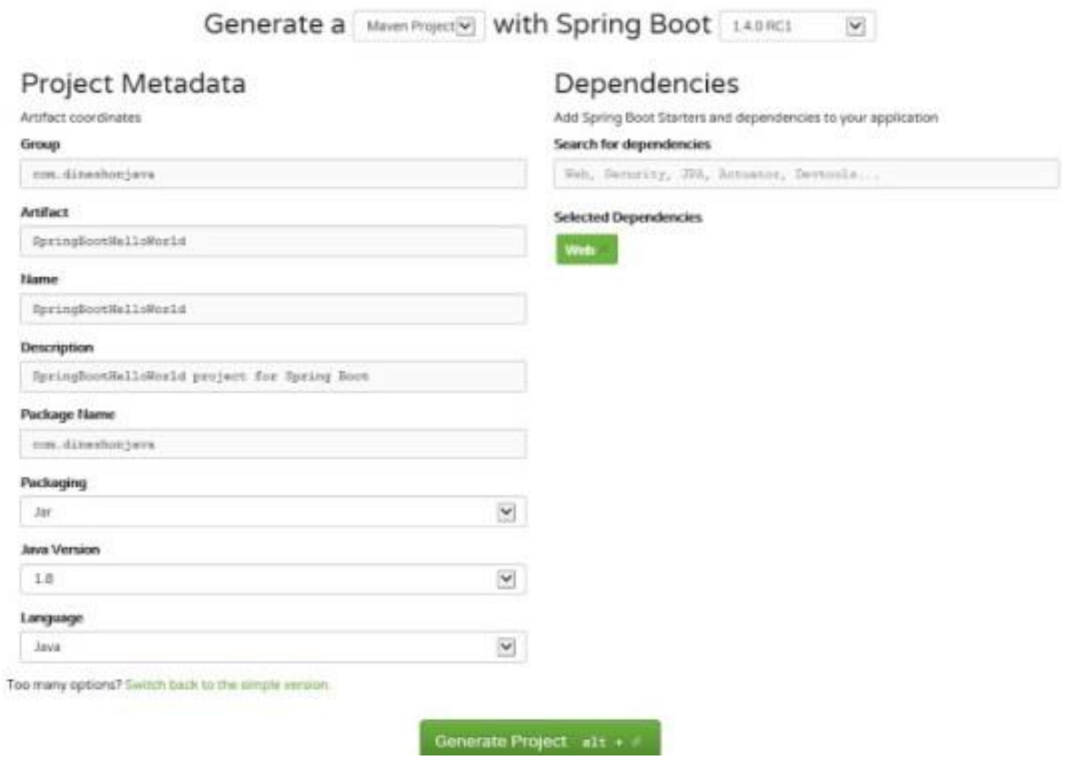
Hello friends!!! Let's develop a simple "**Hello World!**" web application in Java that highlights some of Spring Boot's key features. We'll use Maven to build this project since most IDEs support it.

We are using intializr to quick create project so we can choose either via one of following Initializr interfaces to create web application project structure to initiate.

- Use Web Interface (*http://start.spring.io*)
- Use STS IDE
- Use Spring Boot CLI

Spring Boot application Hello World

Here we are going to use web interface to create project structure for "Hello World" web application. As below figure I have created



Generate a Maven Project with Spring Boot 1.4.0 RC1

Project Metadata

Artifact coordinates

Group: com.dineshonjava

Artifact: SpringBootHelloWorld

Name: SpringBootHelloWorld

Description: SpringBootHelloWorld project for Spring Boot

Package Name: com.dineshonjava

Packaging: Jar

Java Version: 1.8

Language: Java

Too many options? Switch back to the simple version.

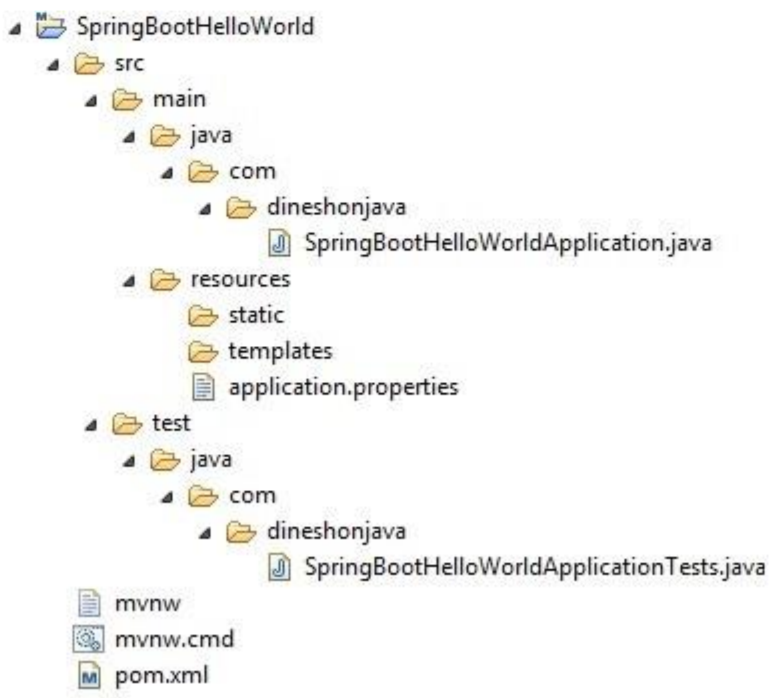
Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Dependencies: Web

Once the project has been created, you should have a project structure similar to that shown in figure below.



```
SpringBootHelloWorld
  src
    main
      java
        com
          dineshonjava
            SpringBootHelloWorldApplication.java
      resources
        static
        templates
        application.properties
    test
      java
        com
          dineshonjava
            SpringBootHelloWorldApplicationTests.java
  mvnw
  mvnw.cmd
  pom.xml
```

Examining a newly initialized Spring Boot project

Now let's slow down and take a closer look at what's contained in the initial project. First thing to noticed in the project structure follow the layout of Maven or Gradle project That is, the main application code is placed in the src/main/java branch of the directory tree, resources are placed in the src/main/ resources branch, and test code is placed in the src/test/java branch At this point we don't have any test resources, but if we did we'd put them in src/test/resources.

Let see auto generated files in deeper:

Pom.xml: The Maven build specification

This is the recipe that will be used to build your project.

```
<?xml version="1.0" encoding="UTF-8"?>

<project                xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>



 <groupId>com.dineshonjava</groupId>
 <artifactId>SpringBootHelloWorld</artifactId>

 <version>0.0.1-SNAPSHOT</version>

 <packaging>jar</packaging>


 <name>SpringBootHelloWorld</name>

 <description>SpringBootHelloWorld     project     for     Spring
Boot</description>


 <parent>

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

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

  <version>1.4.0.RC1</version>

  <relativePath/> <!-- lookup parent from repository -->

 </parent>


 <properties>

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

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

 </properties>


 <dependencies>

  <dependency>

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

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

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

 <repositories>
  <repository>
   <id>spring-snapshots</id>
   <name>Spring Snapshots</name>
```

```
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
     <enabled>true</enabled>
    </snapshots>
   </repository>
   <repository>
    <id>spring-milestones</id>
    <name>Spring Milestones</name>
    <url>https://repo.spring.io/milestone</url>
    <snapshots>
     <enabled>false</enabled>
    </snapshots>
   </repository>
  </repositories>
  <pluginRepositories>
   <pluginRepository>
    <id>spring-snapshots</id>
    <name>Spring Snapshots</name>
    <url>https://repo.spring.io/snapshot</url>
    <snapshots>
     <enabled>true</enabled>
    </snapshots>
   </pluginRepository>
   <pluginRepository>

    <id>spring-milestones</id>

    <name>Spring Milestones</name>

    <url>https://repo.spring.io/milestone</url>

    <snapshots>

     <enabled>false</enabled>

    </snapshots>

   </pluginRepository>

  </pluginRepositories>



</project>
```

Spring Boot provides a number of “**Starters**” that make easy to add jars to your classpath. Our sample application has already used spring-boot-starter-parent in the parent section of the POM. The spring-boot-starter-parent is a special starter that provides useful Maven defaults.

SpringBootHelloWorldApplication.java

The application’s bootstrap class and primary Spring configuration class

```
package com.dineshonjava;



import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;

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

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

```
@RestController

@SpringBootApplication

public class SpringBootHelloWorldApplication {



 @RequestMapping("/")

    String home() {

        return "Hello World!";

    }



 public static void main(String[] args) {

  SpringApplication.run(SpringBootHelloWorldApplication.class,
args);

 }

}
```

The @RestController and @RequestMapping annotations

@RestController is known as a stereotype annotation. The **@RequestMapping** annotation provides "routing" information. It is telling Spring that any HTTP request with the path "/" should be mapped to the home method. The @RestController annotation tells Spring to render the resulting string directly back to the caller.

The @SpringBootApplication annotation

The **@SpringBootApplication** enables Spring component-scanning and Spring Boot auto-configuration. In fact, @SpringBootApplication combines three other useful annotations:

- **Spring's @Configuration**—Designates a class as a configuration class using Spring's Java-based configuration. Although we won't be writing a lot of configuration in this tutorial, we'll favor Java-based configuration over XML configuration when we do.
- **Spring's @ComponentScan**—Enables component-scanning so that the web controller classes and other components you write will be automatically discovered and registered as beans in the Spring application context. Here we'll write a simple Spring MVC controller that will be annotated with @RestController so that component-scanning can find it.

- **Spring Boot’s @EnableAutoConfiguration**— This annotation tells Spring Boot to “guess” how you will want to configure Spring, based on the jar dependencies that you have added. Since spring-boot-starter-web added Tomcat and Spring MVC, the auto-configuration will assume that you are developing a web application and setup Spring accordingly.

application.properties—A place to configure application and Spring Boot properties. The application.properties file given to you by the Initializr is initially empty. In fact, this file is completely optional, so you could remove it completely without impacting the application.

Suppose we are adding following line to this property file

server.port=1208

With this line, you’re configuring the embedded Tomcat server to listen on port **1208** instead of the default port **8080**.

SpringBootHelloWorldApplicationTests.java—A basic integration test class

```
package com.dineshonjava;


import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.boot.test.context.SpringBootTest;

import org.springframework.test.context.junit4.SpringRunner;


@RunWith(SpringRunner.class)

@SpringBootTest

public class SpringBootHelloWorldApplicationTests {


 @Test

 public void contextLoads() {

 }


}
```

In a typical Spring integration test, you’d annotate the test class with @Context- Configuration to specify how the test should load the Spring application context. But in order to take full advantage of Spring Boot magic, the @SpringApplication- Configuration annotation should be used instead.

The “main” method:

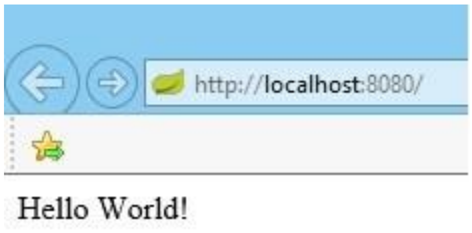
The final part of our application is the main method. This is just a standard method that follows the Java convention for an application entry point. Our main method delegates to Spring Boot’s SpringApplication class by calling run. SpringApplication will bootstrap our application, starting Spring which will in turn start the auto-configured Tomcat web server.

Running the example

Type mvn spring-boot:run from the root project directory to start the application:

```
.   ____          _            __ _ _

 / / ___’_ __ _ _(_)_ __  __ _

( ( )___ | ‘_ | ‘_| | ‘_ / _` |

 /  ___)| |_)| | | | | || (_| |  ) ) ) )

  ‘  |____| .__|_| |_|_| |___, | / / / /

 =========|_|==============|___/=/_/_/_/

 :: Spring Boot ::            (v1.4.0.RC1)

2016-07-14 08:54:35.959  INFO 12868 — [           main] c.d.SpringBootHelloWorldApplication       : Starting SpringBootHelloWorldApplication on NODTBSL206AG with PID 12868 (started by Dinesh.Rajput in D:personal dataspring-boot-workspaceSpringBootHelloWorld)

2016-07-14 08:54:35.961  INFO 12868 — [           main] c.d.SpringBootHelloWorldApplication       : No active profile set, falling back to default profiles: default

2016-07-14 08:54:36.096  INFO 12868 — [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@1809907: startup date [Thu Jul 14 08:54:36 IST 2016]; root of context hierarchy



2016-07-14 08:54:39.630  INFO 12868 — [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
```

If you open a web browser to **localhost:8080** you should see the following output:



Hello World!

Creating an executable jar

Let’s finish our example by creating a completely self-contained executable jar file that we could ruSn in production. Executable jars (sometimes called “fat jars”) are archives containing your compiled classes along with all of the jar dependencies that your code needs to run.

To create an executable jar we need to add the spring-boot-maven-plugin to our pom.xml. Insert the following lines just below the dependencies section:

```
<build>

    <plugins>

        <plugin>

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

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

        </plugin>

    </plugins>

</build>
```

Save your **pom.xml** and run ***mvn package*** from the command line:

```
D:personal dataspring-boot-workspaceSpringBootHelloWorld>mvn package

[INFO] Scanning for projects…

[INFO]

[INFO] ————————————————————————————————————

[INFO] Building SpringBootHelloWorld 0.0.1-SNAPSHOT

[INFO] ————————————————————————————————————

[INFO]
```

```
[INFO] — maven-resources-plugin:2.6:resources (default-resources) @ SpringBoot
HelloWorld —
[INFO] Using ‘UTF-8’ encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 0 resource
[INFO]
[INFO] — maven-compiler-plugin:3.5.1:compile (default-compile) @ SpringBootHel
loWorld —
[INFO] Nothing to compile – all classes are up to date
```

If you look in the target directory you should see ***SpringBootHelloWorld -0.0.1-SNAPSHOT.jar.***

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

$ java -jar target/ SpringBootHelloWorld -0.0.1-SNAPSHOT.jar

```
/ / ___’_ __ _ _(_)_ __  __ _
( ( )___ | ‘_ | ‘_| | ‘_ / _` |
 /  ___)| |_)| | | | | || (_| |  ) ) ) )
  ‘  |____| .__|_| |_|_| |___, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::           (v1.4.0.RC1)

2016-07-14  08:54:35.959   INFO  12868  — [           main]  c.d.SpringBootHelloWorldApplication       : Starting SpringBootHelloWorldApplication on NODTBSL206AG with PID 12868 (started by Dinesh.Rajput in D:personal dataspring-boot-workspaceSpringBootHelloWorld)

2016-07-14 08:54:35.961  INFO 12868 — [           main] c.d.SpringBootHelloWorldApplication       : No active profile set, falling back to default profiles: default

2016-07-14 08:54:36.096  INFO 12868 — [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@1809907: startup date [Thu Jul 14 08:54:36 IST 2016]; root of context hierarchy

2016-07-14  08:54:39.630   INFO  12868  — [           main]  s.b.c.e.t.TomcatEmbeddedServletContainer  : Tomcat initialized with port(s): 8080 (http)
```

Customizing Spring Boot Auto Configuration

.

Hello friends!!! Once again lets us discuss one of the important topics for spring boot is customizing spring boot autoconfiguration. As all we know that Freedom of choice is an awesome thing. If you like PIZZA so you have many choices to select your favorite PIZZA like PAN PIZZA, PIZZA with cheese bust, pizza with different different tops yummy :). Now friends please come to our discussion, here we're going to look at two ways to influence auto-configuration: **explicit configuration overrides** and **fine-grained configuration with properties**.

Overriding auto-configured beans

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. If you can get what you need without it then why would you do extra work, writing and maintaining extra configuration code, but sometimes there are many cases Spring Boot auto configuration is not good enough. For example one of the cases is when you're applying security to your application. Security in your application is one of major concern so it is not single fit for all because there are decisions around application security that Spring Boot has no business making for you even though Spring Boot provides auto configuration for some basic spring security things.

For securing the application:

Just adding Spring Security Starter to add spring security auto configuration to the application. In gradle add following line to **build.gradle** file.

```
compile("org.springframework.boot:spring-boot-starter-security")
```

In **Maven** add following dependency for spring security starter.

```
<dependency>

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

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

</dependency>
```

That's it! Rebuild your application and run it. It's now a secure web application! The security starter adds Spring Security to the application's classpath. With Spring Security on the classpath, auto-configuration kicks in and a very basic Spring Security setup is created. When open this application browser basic authentication is needed then user name is "user" and password is printed in logs.

Custom security configuration in the application

Basic default configuration probably is not fit for your application because unlike authentication page and password is printed in the logs. That is why you will prefer customized security configuration in the application. For overriding default spring boot auto configuration just writing explicit XML based or Java based configuration in the application this means writing a configuration class that extends **WebSecurityConfigurerAdapter**.

```
@Configuration

@EnableWebSecurity

public class SecurityConfig extends WebSecurityConfigurerAdapter {

  @Autowired

  UserRepository userRepository;



  @Override

  protected void configure(HttpSecurity http) throws Exception {

    http

    .authorizeRequests()

    .antMatchers("/").access("hasRole(ADMIN)")

    .antMatchers("/**").permitAll()

    .and()

    .formLogin()

    .loginPage("/login")

    .failureUrl("/login?error=true");

 }

    @Override

   protected void configure(

     AuthenticationManagerBuilder auth) throws Exception {
```

```
    auth

      .userDetailsService(new UserDetailsService() {

      @Override

      public UserDetails loadUserByUsername(String username)

              throws UsernameNotFoundException {

        return userRepository.findUserName(username);

      }

   });

 }

}
```

SecurityConfig is a very basic Spring Security configuration. Even so, it does a lot of what we need to customize security of the application.

Configuring with external properties

Spring Boot allows you to externalize your configuration so you can work with the same application code in different environments. You can use properties files, YAML files, environment variables, Java system properties, JNDI and command-line arguments to externalize configuration. Spring Boot offers over 300 properties for auto configuration of beans in the application.

Let's see one of property when you start the application one banner is printed at log screen. So you can disable that banner by setting property spring.main.show-banner to false.

There are following ways to setting up this property when we run the application

Command Line Parameters:

```
$ java -jar myapp-0.0.1-SNAPSHOT.jar --spring.main.show-banner=false
```

Setting in "application.properties" property file:

```
spring.main.show-banner=false
```

Create a YAML file named “application.yml”:

```
spring:

   main:

      show-banner: false
```

Setting property as an environment variable:

```
$ export spring_main_show_banner=false
```

Note: the use of underscores instead of periods and dashes, as required for environment variable names.

There are, in fact, several ways to set properties for a Spring Boot application. Spring Boot uses a very particular PropertySource order that is designed to allow sensible overriding of values. Properties are considered in the following order:

1. **@TestPropertySource** annotations on your tests.
2. Command line arguments.
3. Properties from **SPRING_APPLICATION_JSON** (inline JSON embedded in an environment variable or system property)
4. **ServletConfig** init parameters.
5. **ServletContext** init parameters.
6. **JNDI** attributes from **java:comp/env**.
7. Java System properties (**System.getProperties()**).
8. OS environment variables.
9. A **RandomValuePropertySource** that only has properties in **random.***.
10. Profile-specific application properties outside of your packaged jar (**application-{profile}.properties** and **YAML** variants)
11. Profile-specific application properties packaged inside your jar (**application-{profile}.properties** and **YAML** variants)
12. Application properties outside of your packaged jar (**application.properties** and **YAML** variants).
13. Application properties packaged inside your jar (**application.properties** and **YAML** variants).
14. **@PropertySource** annotations on your **@Configuration** classes.
15. Default properties (specified using **SpringApplication.setDefaultProperties**)

As for the **application.properties** and **application.yml** files, they can reside in any of four locations:

1. **Externally**, in a /**config** subdirectory of the directory from which the application is run
2. **Externally**, in the directory from which the application is run

3. **Internally**, in a package named “**config**”
4. **Internally**, at the root of the **classpath**

Again, this list is in order of precedence. That is, an **application.properties** file in a **/config** subdirectory will override the same properties set in an **application.properties** file in the application’s **classpath**.

Also, I’ve found that if you have both **application.properties** and **application.yml** side by side at the same level of precedence, **properties** in **application.yml** will override those in **application.properties**.

Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added. You should only ever add one **@EnableAutoConfiguration** annotation.

Gradually replacing auto-configuration

Auto-configuration is noninvasive, at any point you can start to define your own configuration to replace specific parts of the auto-configuration. For example, if you add your own DataSource bean, the default embedded database support will back away.

Disabling specific auto-configuration

1. DISABLING TEMPLATE CACHING

Thymeleaf templates are cached by default that is why changes is never replicate unless you restart the application. You can disable Thymeleaf template caching by setting **spring.thymeleaf.cache** to **false**.

Using command-line argument:

```
$ java -jar myapp-0.0.1-SNAPSHOT.jar --spring.thymeleaf.cache=false
```

Using application.yml file

```
spring:

  thymeleaf:

    cache: false
```

You’ll want to make sure that this **application.yml** file doesn’t follow the application into production.

Using environment variable:

```
$ export spring_thymeleaf_cache=false
```

Others template caching can be turned off for Spring Boot's other supported template options by setting these properties:

- spring.freemarker.cache (Freemarker)
- spring.groovy.template.cache (Groovy templates)
- spring.velocity.cache (Velocity)

2. CONFIGURING THE EMBEDDED SERVER

By default embedded server for spring boot application is **tomcat** with port **8080**. It is fine in case of single application running but it will be become problem in case run multiple applications simultaneously. If all of the applications try to start a Tomcat server on the same port, there'll be port collisions starting with the second application.
So to prevent these collisions of port we need to do is set the **server.port** property.
If this is a one-time change, it's easy enough to do this as a **command-line argument**:

```
$ java -jar myapp-0.0.1-SNAPSHOT.jar --server.port=8181
```

For permanent port change use **application.yml:**

```
server:

   port: 8000
```

3. CONFIGURING LOGGING

By default, Spring Boot configures logging via Logback (http://logback.qos.ch) to log to the console at INFO level.
For **Maven builds**, you can exclude Logback by excluding the default logging starter transitively resolved by the root starter dependency:

```
<dependency>

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

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

<exclusions>

<exclusion>

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

<artifactId>spring-boot-starter-logging</artifactId>
```

```
</exclusion>

</exclusions>

</dependency>
```

In **Gradle**, it’s easiest to place the exclusion under the configurations section:

```
configurations {

all*.exclude group:'org.springframework.boot',

module:'spring-boot-starter-logging'

}
```

With the default logging starter excluded, you can now include the starter for the logging implementation you’d rather use. With a **Maven** build you can add Log4j like this:

```
<dependency>

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

<artifactId>spring-boot-starter-log4j</artifactId>

</dependency>
```

In a **Gradle** build you can add Log4j like this:

```
compile("org.springframework.boot:spring-boot-starter-log4j")
```

For full control over the logging configuration, you can create a logback.xml file at the root of the classpath (in src/main/resources). Here’s an example of a simple logback.xml file you might use:

```
<configuration>
```

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>

%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n

</pattern>

</encoder>

</appender>

<logger name="root" level="INFO"/>

<root level="INFO">

<appender-ref ref="STDOUT" />

</root>

</configuration>
```

To set the logging levels, you create properties that are prefixed with **logging.level,** followed by the name of the logger for which you want to set the logging level. **application.yml:**

```
logging:

    level:

        root: WARN

        org:

             springframework:

                       security: DEBUG
```

Now suppose that you want to write the log entries to a file named **MyApp.log** at **/opt/logs/**. The **logging.path**and **logging.file** properties can help with that:

```
logging:

     path: /opt/logs/

     file: MyApp.log

     level:

         root: WARN

     org:

        springframework:

              security: DEBUG
```

By default, the log files will rotate once they hit 10 megabytes in size.

application.properties:

```
logging.path=/opt/logs/

logging.file=MyApp.log

logging.level.root=WARN

logging.level.root.org.springframework.security=DEBUG
```

You can also change name of logger configuration file name i.e. you can change name of file **logback.xml** to **log-config.xml** by setting the property **logging.config** in property file or **YML** file.

```
logging:

    config:

        classpath:log-config.xml
```

4. CONFIGURING A DATA SOURCE

application.yml: if you're using a MySQL database, your **application.yml** file might look like this

```
spring:

     datasource:

           url: jdbc:mysql://localhost/dojdb

           username: root

          password: root
```

Specify driver name for database with property **spring.datasource .driver-class-name** property as follows

```
spring:

     datasource:

           url: jdbc:mysql://localhost/dojdb

           username: root

          password: root

         driver-class-name: com.mysql.jdbc.Driver
```

Using JNDI for DataSource:
By using property **spring.datasource.jndi-name**

```
spring:

      datasource:

            jndi-name: java:/comp/env/jdbc/dojDBDS
```

Type-safe Configuration Properties
If you are working with multiple properties or your data is hierarchical in nature then sometimes it may be problem in configuration properties. Spring Boot provides an alternative method of working with properties that allows strongly typed beans to govern and validate the configuration of your application.
For example:

```
@Component

@ConfigurationProperties(prefix="database")

public class DatabaseSettings {



    private String dbname;



    private String dburl;



    // ... getters and setters



}
```

The **@EnableConfigurationProperties** annotation is automatically applied to your project so that any beans annotated with **@ConfigurationProperties** will be configured from the Environment properties. The **@ConfigurationProperties** annotation won't work unless you've enabled it by adding **@EnableConfigurationProperties** in one of your Spring configuration classes. This style of configuration works particularly well with the **SpringApplication** external **YAML** configuration:
application.yml

```
database:

    dbname: dojdb

    dburl: jdbc:mysql//192.168.1.1:3309/dojdb
```

```
# additional configuration as required
```

To work with **@ConfigurationProperties** beans you can just inject them in the same way as any other bean.

```
@Service

public class UserService {

    @Autowired

    private DatabaseSettings connection;

     //...

    @PostConstruct

    public void openConnection() {

        Server server = new Server();

        this.connection.configure(server);

    }

}
```

It is also possible to shortcut the registration of **@ConfigurationProperties** bean definitions by simply listing the properties classes directly in the **@EnableConfigurationProperties** annotation:

```
@Configuration

@EnableConfigurationProperties(DatabaseSettings.class)

public class DBConfiguration {

}
```

Using @ConfigurationProperties for outside beans:
We can use **@ConfigurationProperties** for beans which either outside from the application or not in your control. Let's see below

```
@ConfigurationProperties(prefix = "foo")

@Bean

public FooComponent fooComponent() {

    ...

}
```

Any property defined with the foo prefix will be mapped onto that **FooComponent** bean in a similar manner as the **DatabaseSettings** example above.
Note: Biding Property in Spring Boot
It's also worth noting that Spring Boot's property resolver is clever enough to treat camel-cased properties as interchangeable with similarly named properties with hyphens or underscores. In other words, a property named **database.dbName** is equivalent to both **database.db_name** and **database.db-name**. And also **DATABASE_DB_NAME** as environment variable in OS.
@ConfigurationProperties Validation
Spring Boot will attempt to validate external configuration, by default using JSR-303 (if it is on the classpath). You can simply add **JSR-303 javax.validation** constraint annotations to your @ConfigurationProperties class:

```
@Component

@ConfigurationProperties(prefix="database")

public class DatabaseSettings {

    @NotNull

    private String dbName;

    @NotNull

    private String dbUrl;

    // ... getters and setters
}
```

Configuring with profiles
When applications are deployed to different runtime environments, there are usually some configuration details that will differ. The details of a database connection, for instance, are likely different in a development environment than in a quality assurance environment, and different still in a production environment.

```
@Configuration

@Profile("production")

public class ProductionConfiguration {

    // ...

}
```

The **@Profile** annotation used here requires that the "**production**" profile be active at runtime for this configuration to be applied. If the "**production**" profile isn't active, this configuration will be ignored and applied any default auto configuration.
Activate profiles:
Profiles can be activated by setting the ***spring.profiles.active*** property using any of the means available for setting any other configuration property. For example, you could activate the "**production**" profile by running the application at the **command line** like this:

```
$ java -jar MyApp-0.0.1-SNAPSHOT.jar -- spring.profiles.active=production
```

Or you can add the ***spring.profiles.active*** property to **application.yml:**

```
spring:

    profiles:

        active: production
```

Profile-specific properties:
If you're using **application.properties** to express configuration properties, you can provide profile-specific properties by creating additional properties files named with the pattern "***application-{profile}.properties***". Profile-specific properties are loaded from the same locations as standard ***application.properties***, with profile-specific files always overriding the non-specific ones irrespective of whether the profile-specific files are inside or outside your packaged jar.
For Example the **development configuration** would be in a file named **application-development.properties** and contain properties for verbose, console written logging:

```
logging.level.root=DEBUG
```

But for **production**, **application-production.properties** would configure logging to be at WARN level and higher and to write to a log file:

```
logging.path=/opt/logs/

logging.file=MyApp.log

logging.level.root=WARN
```

Multi-profile YAML documents:
You can specify multiple profile-specific YAML documents in a single file by using a ***spring.profiles*** key to indicate when the document applies. For example:

```
logging:

    level:

        root: INFO

---

spring:

    profiles: development

logging:

     level:
         root: DEBUG

---

spring:

      profiles: production

logging:

      path: /opt/

      file: MyApp.log

     level:

          root: WARN
```

As you can see, this **application.yml** file is divided into three sections by a set of triple hyphens (—). The second and third sections each specify a value for ***spring .profiles***. This property indicates which profile each section's properties apply to.
The properties defined in the middle section apply to development because it sets ***spring.profiles*** to "**development**". Similarly, the last section has ***spring.profiles*** set to "**production**", making it applicable when the "**production**" profile is active.
The first section, on the other hand, doesn't specify a value for ***spring.profiles***. Therefore, its properties are common to all profiles or are defaults if the active profile doesn't otherwise have the properties set.

Customizing error pages

Spring Boot offers this "**whitelabel**" error page by default as part of auto-configuration. The default error handler that's auto-configured by Spring Boot looks for a view whose name is "**error**". The easiest way to customize the error page is to create a custom view that will resolve for a view named "**error**".
Ultimately this depends on the view resolvers in place when the error view is being resolved. This includes

- Any bean that implements Spring's View interface and has a bean ID of "**error**" (resolved by Spring's BeanNameViewResolver)
- A Thymeleaf template named "**error.html**" if Thymeleaf is configured
- A FreeMarker template named "**error.ftl**" if FreeMarker is configured
- A Velocity template named "**error.vm**" if Velocity is configured
- A JSP template named "**error.jsp**" if using JSP views

Summary
Spring Boot Auto configuration do almost configuration for your application. When autoconfiguration doesn't fit your needs, Spring Boot allows you to override and fine-tune the configuration it provides.
Happy Spring Boot Learning!!!

Logging Configuration in Spring Boot

Hello friends!!! Once again we are going to discuss very important topic about spring boot configuration. We always curious about defining tacking or logs basis things own way. Like Pizza organization provide the tracking about the ordered pizza what is the current status of pizza. Still we supposed to see more detailed status like information about pizza base or how to make pizza and also supposed there no problem happens in preparing the pizza.

Logging is not the core requirement for any business application but we cannot ignore it because logging is very important part for any application for tracking information about application and helpful for developer in debugging the application. In some applications developers use synchronous logging which can impact the application's performance. Logging libraries like **Logback, Log4j2 provide async logging**.

Spring Boot provides support for logging and uses Commons Logging for all internal logging, but leaves the underlying log implementation open. Default configurations are provided for **Java Util Logging, Log4J2 and Logback**. In each case loggers are pre-configured to use console output with optional file output also available.

By default, if you use the '**Starters**', Logback will be used for logging. Here we are going to use default logging by Spring Boot.

Default Logging Format in Spring Boot

The default logs look like as below

```
2016-07-24 20:31:45.280  INFO 10220 --- [           main] c.d.SpringBootHelloWorldApplication      : Starting SpringBootHelloWorldApplication on NODTBSL206AG with PID 10220 (started by Dinesh.Rajput in D:personal dataspring-boot-workspaceSpringBootHelloWorld)

2016-07-24 20:31:45.286  INFO 10220 --- [           main] c.d.SpringBootHelloWorldApplication      : No active profile set, falling back to default profiles: default
```

```
2016-07-24 20:31:45.502 INFO 10220 --- [ main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@1809907: startup date [Sun Jul 24 20:31:45 IST 2016]; root of context hierarchy

2016-07-24 20:31:48.201  INFO 10220 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)

2016-07-24 20:31:48.226  INFO 10220 --- [           main] o.apache.catalina.core.StandardService   : Starting service Tomcat

2016-07-24 20:31:48.228  INFO 10220 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.5.3

2016-07-24 20:31:48.485  INFO 10220 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext

2016-07-24 20:31:48.485  INFO 10220 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 2993 ms
```

The following items are output:

- **Date and Time**: Millisecond precision and easily sort able.
- **Log Level:** ERROR, WARN, INFO, DEBUG or TRACE.
- **Process ID.**
- A — Separator to distinguish the start of actual log messages.
- **Thread name:** Enclosed in square brackets (may be truncated for console output).
- **Logger name:** This is usually the source class name (often abbreviated).

The log message.

The default log level configured by logback is **DEBUG** i.e any messages logged at **ERROR**, **WARN**, **INFO** and **DEBUG** get printed on the console.

Create Spring Boot Maven Project

Let’s create a Maven Spring Boot project for customizing logging configuration.

POM.XML

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project                xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>



 <groupId>com.dineshonjava</groupId>

 <artifactId>SpringBootHelloWorld</artifactId>

 <version>0.0.1-SNAPSHOT</version>

 <packaging>jar</packaging>



 <name>SpringBootHelloWorld</name>

 <description>SpringBootHelloWorld    project    for    Spring
Boot</description>



 <parent>

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

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

  <version>1.4.0.RC1</version>

  <relativePath/> <!-- lookup parent from repository -->

 </parent>



 <properties>

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

```
  <java.version>1.8</java.version>
 </properties>

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

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

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

</project>
```

In above **pom.xml** we do not need to add the logging dependency explicitly because dependency **spring-boot-starter** includes the dependencies for logging **spring-boot-starter-logging**. The **spring-boot-starter-logging** includes **SLF4J** and **logback** dependencies with appropriate **SLF4J** wrappers for other logging libraries.

Now let’s see Class file where we have logs message

```
package com.dineshonjava;



import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;

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

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



@RestController

@SpringBootApplication

public class SpringBootHelloWorldApplication {



private static final Logger logger =
LoggerFactory.getLogger(SpringBootHelloWorldApplication.class);
@RequestMapping("/")
    String home() {
        return "Hello World!";
    }


public static void main(String[] args) {
    SpringApplication.run(SpringBootHelloWorldApplication.class,
args);
    logger.error("Message logged at ERROR level");
    logger.warn("Message logged at WARN level");
    logger.info("Message logged at INFO level");
    logger.debug("Message logged at DEBUG level");
}
}
```

There are following logs look like after running above class.

```
2016-07-24 20:57:40.828 ERROR 3024 --- [                  main]
c.d.SpringBootHelloWorldApplication      : Message logged at ERROR
level
2016-07-24 20:57:40.828  WARN 3024 --- [                  main]
c.d.SpringBootHelloWorldApplication      : Message logged at WARN
level
2016-07-24 20:57:40.828  INFO 3024 --- [                  main]
c.d.SpringBootHelloWorldApplication      : Message logged at INFO
level
```

Customizing default Configuration for Logging:

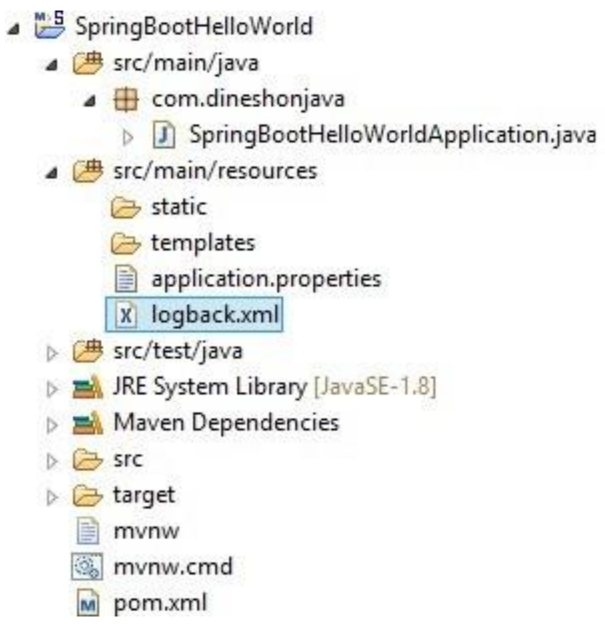
By adding **logback.xml** file to the application we can override the default logging configuration providing by the Spring Boot. This file place in the **classpath (src/main/resources)** of the application for Spring Boot to pick the custom configuration.

Logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>

  <appender                                          name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">

    <!-- Log message format -->

    <encoder>

      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n

      </pattern>

    </encoder>

  </appender>



  <!-- Setting the root level of logging to INFO -->

  <root level="info">

    <appender-ref ref="STDOUT" />

  </root>

</configuration>
```

There are following things we have note:

- We are using a appender **STDOUT** using **ConsoleAppender** which prints to the console
- We are giving **pattern** to the **appender** to build the **log message**
- Set up a root logger which logs any message above **INFO** level using the **STDOUT** appender



There are following logs look like after running this application.

```
21:14:46.352 [main] ERROR c.d.SpringBootHelloWorldApplication -
Message logged at ERROR level

21:14:46.352 [main] WARN  c.d.SpringBootHelloWorldApplication -
Message logged at WARN level

21:14:46.352 [main] INFO  c.d.SpringBootHelloWorldApplication -
Message logged at INFO level
```

We have to note here in above logs there is no **DEBUG** logs printed.

Printing Logs into a File:

By default, Spring Boot will only log to the console and will not write log files. If you want to write log files in addition to the console output you need to set a ***logging.file*** or ***logging.path*** property in your **application.properties**.

There are following cases for logging property:

- If No ***logging.path*** and No ***logging.file*** then Console only logging.
- If No ***logging.path*** and Specify ***logging.file*** then writes to the specified log file. Names can be an exact location or relative to the current directory.
- If Specify ***logging.path*** and No ***logging.file*** then writes spring.log to the specified directory. Names can be an exact location or relative to the current directory.

Log files will rotate when they reach **10 Mb** and as with console output, **ERROR**, **WARN** and **INFO** level messages are logged by default.

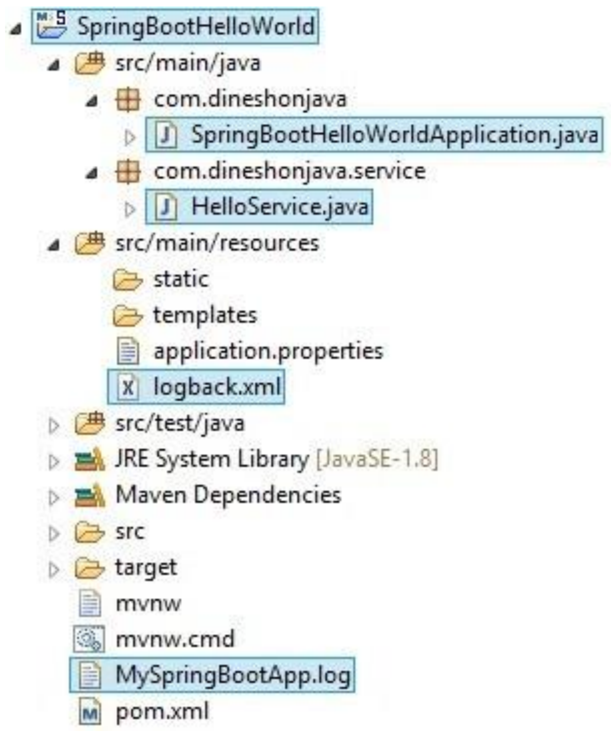
Now let’s see the following changes we are made in the **logback.xml** as below

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>

  <appender                                              name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">

    <!-- Log message format -->

    <encoder>

      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>

    </encoder>

  </appender>

  <!-- Need appender to write to file -->

  <appender name="FILE" class="ch.qos.logback.core.FileAppender">

    <!-- Name of the file where the log messages are written -->

    <file>MySpringBootApp.log</file>

    <encoder>

      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>

    </encoder>

  </appender>

  <logger name="com.dineshonjava.service" level="warn">

    <appender-ref ref="FILE" />

  </logger>



  <!-- Setting the root level of logging to INFO -->
  <root level="info">

    <appender-ref ref="FILE" />

  </root>

</configuration>
```

Here we are introducing one more file “**HelloService.java**” for logging purpose as below:

```
package com.dineshonjava.service;



import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class HelloService {

private               Logger               logger              =
LoggerFactory.getLogger(HelloService.class);

public void service(){

  logger.info("Message       at       INFO       level       from
HelloService.service()");

  logger.warn("Message       at       WARN       level       from
HelloService.service()");

}

}
```

Now running the application will redirect all the log messages to the file **MySpringBootApp.log** present in the current directory.



Custom log configuration

The various logging systems can be activated by including the appropriate libraries on the classpath, and further customized by providing a suitable configuration file in the root of the classpath, or in a location specified by the Spring Environment property ***logging.config***.

Depending on your logging system, the following files will be loaded:

1. **Logback** -> logback-spring.xml, logback-spring.groovy, logback.xml or logback.groovy
2. **Log4j2** -> log4j2-spring.xml or log4j2.xml
3. **JDK (Java Util Logging)** -> logging.properties

There are following properties we can use for custom log configuration

1. ***logging.file*** -> Used in default log configuration if defined for logging file name.
2. ***logging.path*** -> Used in default log configuration if defined for logging file path of directory.
3. ***logging.pattern.console*** -> The log pattern to use on the console.
4. ***logging.pattern.file*** -> The log pattern to use in a file.
5. ***logging.pattern.level*** -> The format to use to render the log level.
6. ***logging.exception-conversion-word*** -> The conversion word that's used when logging exceptions.

Summary

Here we have seen enough things about how to use the logging processes in Spring Boot application. First we have see about the default logging support in the Spring Boot applications and then we have made it how to override them by modifying the **logback.xml** file. And also we have discussed list of all properties those we have used to customize the logging configuration in the spring boot application.

Happy Spring Boot Learning!!!

Spring Boot with Spring MVC Application

Spring Boot reduced lots of spring configuration and kick o fast development. Spring Boot offers a new paradigm for developing Spring applications with minimal friction. With Spring Boot, you'll be able to develop Spring applications with more agility and be able to focus on addressing your application's functionality needs with minimal.

Spring Boot is well suited for web application development. You can easily create a self-contained HTTP server using embedded Tomcat, Jetty, or Undertow. Most web applications will use the spring-boot-starter-web module to get up and running quickly.

Spring Web MVC framework

MVC stands for **Model**, **View**, **Controller**. The MVC design pattern is probably the most popular design pattern used when writing code to generate dynamic web content.

- **Model** refers to a data model, or some type of data structure.
- The **view** layer, in Java frequently a JSP. This will take data from the Model and render the view.
- Spring MVC lets you create special **@Controller** or **@RestController** beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP using **@RequestMapping** annotations.

Spring MVC auto-configuration

Spring Boot provides auto-configuration for Spring MVC. There are following features for auto-configuration for Spring MVC.

- Inclusion of **ContentNegotiatingViewResolver** and **BeanNameViewResolver** beans.
- Support for serving **static** resources, including support for **WebJars**. By default Spring Boot will serve static content from a directory called **/static** (or **/public** or **/resources** or **/META-INF/resources**) in the **classpath** or from the root of the **ServletContext**. It uses the **ResourceHttpRequestHandler** from Spring MVC so you can modify that behavior by adding your own **WebMvcConfigurerAdapter** and overriding the **addResourceHandlers** method. Any resources with a path in **/webjars/**** will be served from jar files if they are packaged in the **Webjars** format.
- Automatic registration of **Converter**, **GenericConverter**, **Formatter** beans.
- Support for **HttpMessageConverters**. Spring MVC uses the **HttpMessageConverter** interface to convert HTTP requests and responses.
- Automatic registration of **MessageCodesResolver**. Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: **MessageCodesResolver**
- Static **index.html** support. If you do this the default welcome page detection will switch to your custom locations, so if there is an **index.html** in any of your locations on startup, it will be the home page of the application.
- Custom **Favicon** support.
- Automatic use of a **ConfigurableWebBindingInitializer** bean. Spring MVC uses a **WebBindingInitializer** to initialize a **WebDataBinder** for a particular request. If you create your own **ConfigurableWebBindingInitializer @Bean**, Spring Boot will automatically configure Spring MVC to use it.

- When you’re using one of these **templating** engines (**thymeleaf, freemarker, velocity, Groovy** etc.) with the default configuration, your templates will be picked up automatically from **src/main/resources/templates**.
- **Error Handling** Spring Boot provides an **/error** mapping by default that handles all errors in a sensible way, and it is registered as a ‘global’ error page in the servlet container. For machine clients it will produce a JSON response with details of the error, the HTTP status and the exception message. For browser clients there is a ‘**whitelabel**’ error view that renders the same data in HTML format. If you want to display a custom HTML error page for a given status code, you add a file to an /error folder. Error pages can either be static HTML (i.e. added under any of the static resource folders) or built using templates. The name of the file should be the exact status code or a series mask.
- **Spring Boot** includes support for **embedded Tomcat**, **Jetty**, and **Undertow servers**. Most developers will simply use the appropriate ‘**Starter**’ to obtain a fully configured instance. By default the embedded server will listen for HTTP requests on port **8080**.
- Any **Servlet**, **Filter** or **Servlet *Listener** instance that is a Spring bean will be registered with the embedded container. This can be particularly convenient if you want to refer to a value from your **application.properties**during configuration.
- By default, if the context contains only a **single Servlet** it will be **mapped to /**. In the case of **multiple Servlet beans** the bean **name will be used as a path prefix**. **Filters** will **map to /***.
- **EmbeddedWebApplicationContext** The EmbeddedWebApplicationContext is a special type of **WebApplicationContext** that bootstraps itself by searching for a single **EmbeddedServletContainerFactory** bean.

Customizing embedded servlet containers

Common servlet container settings can be configured using Spring Environment properties. Usually you would define the properties in your **application.properties** file.

- **Network settings:** listen port for incoming HTTP requests (**server.port**), interface address to bind to **server.address**, etc.
- **Session settings:** whether the session is persistent (**server.session.persistence**), session timeout (**server.session.timeout**), location of session data (**server.session.store-dir**) and session-cookie configuration (**server.session.cookie.***).
- **Error management:** location of the error page (**server.error.path**), etc.

Working with SQL Databases in Spring Boot Application

Hello friends here I am going to explain how to use **SQL Database** or **Embedded Databases** with Spring Boot. The Spring Framework provides extensive support for working with SQL databases. SQL Databases are an integral part of any application being development. They help in persisting application data. Spring provides a nice abstraction on top of JDBC API using **JdbcTemplate** and also provides great transaction management capabilities using annotation based approach. Spring provides support for any **ORM** tools like **Hibernate**. Spring Data provides an additional level of functionality, creating Repository implementations directly from interfaces and using conventions to generate queries from your method names.

Configuring a DataSource

We can configure **DataSource** with Spring Boot for two type databases as below

1. Configure Embedded Database or In Memory Database
2. Configure Production Database
3. Configure Database by using JNDI

1. Configure Embedded Database or In Memory Database

For development environment for any project it's often convenient to select an in-memory embedded database. Obviously, in-memory databases do not provide persistent storage; you will need to populate your database when your application starts and be prepared to throw away data when your application ends. Spring Boot auto configure databases like H2, HSQL and Derby etc. You don't need to provide any connection URLs, simply include a build dependency to the embedded database that you want to use.

Let's see **pom.xml** file

```
<dependencies>

  <dependency>

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

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

  </dependency>



  <dependency>

   <groupId>org.hsqldb</groupId>

   <artifactId>hsqldb</artifactId>

   <scope>runtime</scope>

  </dependency>

  <dependency>

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

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

   <scope>test</scope>

  </dependency>
  <dependency>

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

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

  </dependency>

 </dependencies>
```

- The **spring-boot-starter-jdbc** module transitively pulls **tomcat-jdbc-{version}.jar** which is used to configure the **DataSource** bean. In the above dependencies we have included the JDBC dependency – this gives us **JdbcTemplate** and other JDBC libraries, the **org.hsqldb** dependency adds embedded **hsqldb**.
- If you have not defined any **DataSource** bean explicitly and if you have any embedded database driver in classpath such as **H2, HSQL** or **Derby** then SpringBoot will automatically registers DataSource bean using **in-memory** database settings.
- These embedded DBs are in-memory and each time the application shuts down the schema and data gets erased. One way to keep schema and data in the in-memory is to populate it during application startup. This is taken care by Spring Boot.
- We can have **schema.sql** and **data.sql** files in root classpath which SpringBoot will automatically use to initialize database. Spring JDBC uses these sql files to create schema and populate data into the schema.

In addition to **schema.sql** and **data.sql**, Spring Boot will load **schema-${platform}.sql** and **data-${platform}.sql** files if they are available in root classpath. One can create multiple **schema.sql** and **data.sql** files, one for each db platform. So we can have **schema-hsqldb.sql**, **data-hsqldb.sql**, **schema-mysql.sql** and so on. And the file to be picked is decided by the value assigned to the property **spring.datasource.platform**. In this post we are going to create a **schema-hsqldb.sql**file with the following contents:

```
CREATE TABLE users(userId INTEGER NOT NULL,userName VARCHAR(100)
NOT NULL,userEmail VARCHAR(100) DEFAULT NULL,address VARCHAR(100)
DEFAULT NULL,PRIMARY KEY (userId));
```

Create **data.sql** in **src/main/resources** as follows:

```
insert into users(userId, userName, userEmail, address) values
(1000, 'Dinesh', 'dinesh@gmail.com', 'Delhi');

insert into users(userId, userName, userEmail, address) values
(1001, 'Kumar', 'kumar@gmail.com', 'Greater Noida');
```

```
insert into users(userId, userName, userEmail, address) values
(1002, 'Rajput', 'rajput@gmail.com', 'Noida');
```

Next is to create the **User.java** model class:

```
/**

 *

 */

package com.dineshonjava.model;


/**

 * @author Dinesh.Rajput

 *

 */

public class User {

 private Integer userId;

 private String userName;

 private String userEmail;

 private String address;

 public Integer getUserId() {

  return userId;

 }

 public void setUserId(Integer userId) {

  this.userId = userId;

 }
```

```
public String getUserName() {
 return userName;
}
public void setUserName(String userName) {
 this.userName = userName;
}
public String getUserEmail() {
 return userEmail;
}
public void setUserEmail(String userEmail) {
 this.userEmail = userEmail;
}
public String getAddress() {
 return address;
}
public void setAddress(String address) {
 this.address = address;
}
@Override
public String toString() {
 return "User [userId=" + userId + ", userName=" + userName
   + ", userEmail=" + userEmail + ", address=" + address + "]";
}
```

```
}
```

Next is to create a service class **UserService** which makes use of **JdbcTemplate** to insert data and retrieve data from **hsqldb**. There are two methods in the service class- **createUser** and **findAllUsers**. **createUser** adds a new row to the user table and **findAllUsers** fetches all the rows in the user table. Below is the **UserService.java** class definition:

```
/**
 *
 */
package com.dineshonjava.service;


import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.List;


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

import org.springframework.jdbc.core.JdbcTemplate;

import org.springframework.jdbc.core.PreparedStatementCreator;

import org.springframework.jdbc.support.GeneratedKeyHolder;

import org.springframework.jdbc.support.KeyHolder;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;
import com.dineshonjava.model.User;
import com.dineshonjava.utils.UserRowMapper;

/**
 * @author Dinesh.Rajput
 *
 */
@Service
public class UserService {

 @Autowired
    private JdbcTemplate jdbcTemplate;

    @Transactional(readOnly=true)
    public List<User> findAll() {
        return jdbcTemplate.query("select * from users",
                new UserRowMapper());
    }

    @Transactional(readOnly=true)
    public User findUserById(int id) {
        return jdbcTemplate.queryForObject(
            "select * from users where userId=?",
            new Object[]{id}, new UserRowMapper());
    }


    public User create(final User user)
    {
        final      String      sql      =      "insert      into
users(userId,userName,userEmail,address) values(?,?,?,?)";


        KeyHolder holder = new GeneratedKeyHolder();
        jdbcTemplate.update(new PreparedStatementCreator() {
            @Override
            public                              PreparedStatement
createPreparedStatement(Connection       connection)       throws
SQLException {
                PreparedStatement              ps              =
connection.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);
                ps.setInt(1, user.getUserId());
                ps.setString(2, user.getUserName());
                ps.setString(3, user.getUserEmail());
                ps.setString(4, user.getAddress());
                return ps;
            }
        }, holder);
        int newUserId = holder.getKey().intValue();

        user.setUserId(newUserId);

        return user;

    }

}




/**

 *

 */

package com.dineshonjava.utils;


import java.sql.ResultSet;

import java.sql.SQLException;


import org.springframework.jdbc.core.RowMapper;


import com.dineshonjava.model.User;


/**

 * @author Dinesh.Rajput

 *
 */
public class UserRowMapper implements RowMapper<User>{


  @Override
    public User mapRow(ResultSet rs, int rowNum) throws
SQLException {
        User user = new User();
        user.setUserId(rs.getInt("userId"));
        user.setUserName(rs.getString("userName"));
        user.setUserEmail(rs.getString("userEmail"));
        user.setAddress(rs.getString("address"));
        return user;
    }


}
```

Now creating main application class and controller create users into user table and fetching all users from the table as json format.

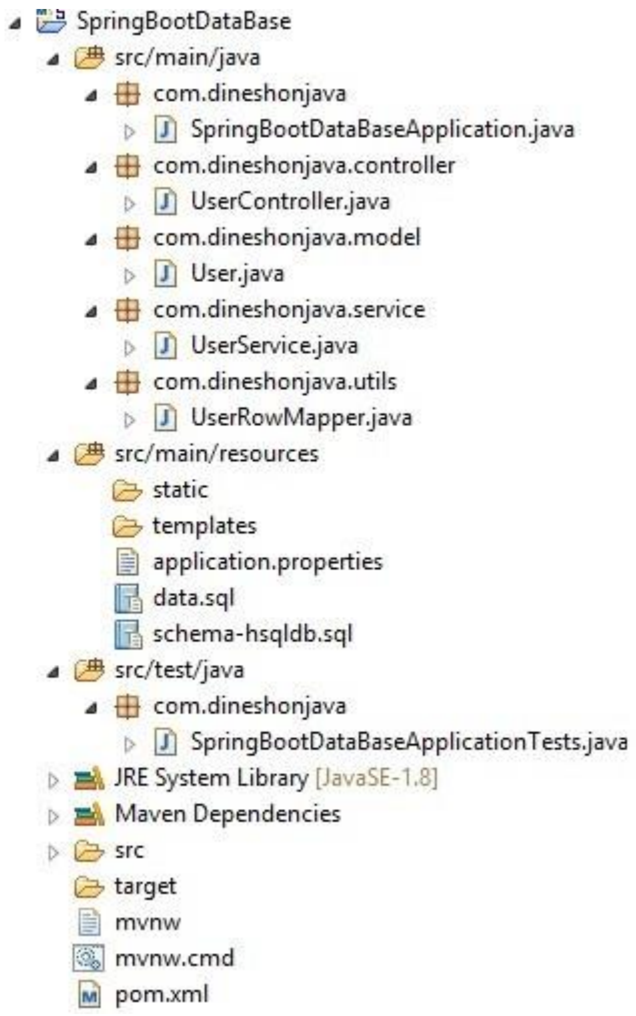
```
/**
 *
 */
package com.dineshonjava.controller;
```

```
import java.util.List;


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

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

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


import com.dineshonjava.model.User;

import com.dineshonjava.service.UserService;


/**

 * @author Dinesh.Rajput

 *

 */

@RestController

public class UserController {


 @Autowired

 UserService userService;


 @RequestMapping("/")

    User home(User user) {

  user = userService.create(user);

        return user;
    }


 @RequestMapping("/users")

    List<User> findAllUsers() {

  List<User> users = userService.findAll();

        return users;

    }

}
```

Now let’s see main class of application **SpringBootDataBaseApplication.java**

```
package com.dineshonjava;


import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication

public class SpringBootDataBaseApplication {


 public static void main(String[] args) {

  SpringApplication.run(SpringBootDataBaseApplication.class,
args);

 }
}
```

Following is the **project structure** whatever we made.



```
SpringBootDataBase
  src/main/java
    com.dineshonjava
      SpringBootDataBaseApplication.java
    com.dineshonjava.controller
      UserController.java
    com.dineshonjava.model
      User.java
    com.dineshonjava.service
      UserService.java
    com.dineshonjava.utils
      UserRowMapper.java
  src/main/resources
    static
    templates
    application.properties
    data.sql
    schema-hsqldb.sql
  src/test/java
    com.dineshonjava
      SpringBootDataBaseApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

Now run this application as Spring Boot application in STS.

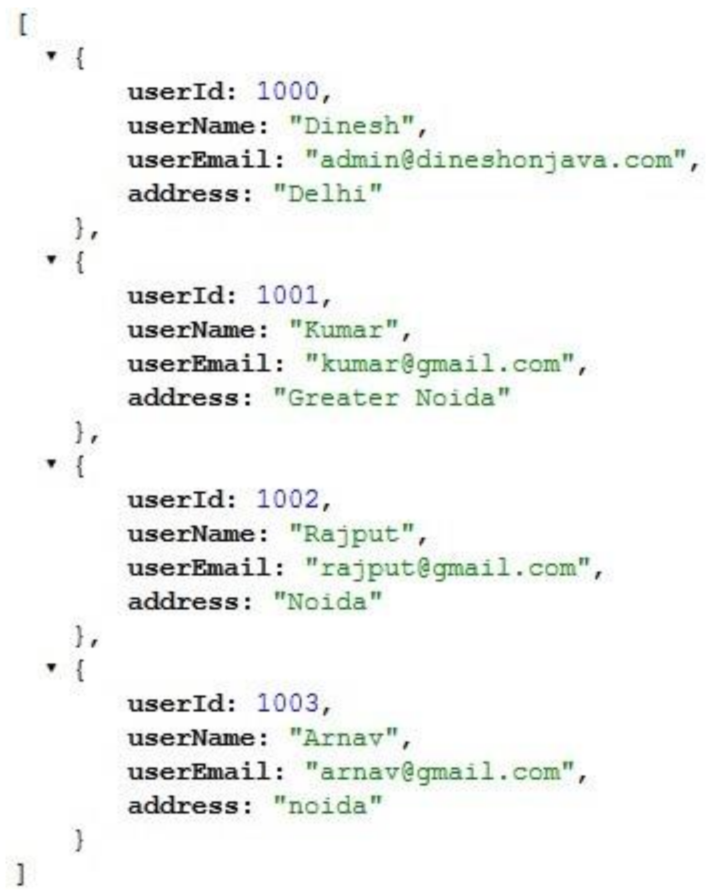
Whenever hit following URL then new row has been created into table suppose.

http://localhost:8080/?userId=1003&userName=Arnav&userEmail=arnav@gmail.com&address=Noida

One row has been created let's see all data with following URL

http://localhost:8080/users

This display all users from database as JSON format in the browser as below



```
[
  ▾ {
        userId: 1000,
        userName: "Dinesh",
        userEmail: "admin@dineshonjava.com",
        address: "Delhi"
    },
  ▾ {
        userId: 1001,
        userName: "Kumar",
        userEmail: "kumar@gmail.com",
        address: "Greater Noida"
    },
  ▾ {
        userId: 1002,
        userName: "Rajput",
        userEmail: "rajput@gmail.com",
        address: "Noida"
    },
  ▾ {
        userId: 1003,
        userName: "Arnav",
        userEmail: "arnav@gmail.com",
        address: "noida"
    }
]
```

We have learned how to get started quickly with Embedded database. What if we want to use Non-Embedded databases like **MySQL**, **Oracle** or **PostgreSQL** etc? **In-memory databases** have lot of restriction and are useful in the early stages of the application and that too in local environments. As the application development progresses we would need data to be present even after application ends.

2. Configure Production Database

In the production environment In Memory database is not good choice because of there are lots of limitations. For production environment we can use any database like **MySQL, DB2, Oracle, PostgreSQL** etc. Production database connections can also be auto-configured using a pooling DataSource. Here's the algorithm for choosing a specific implementation:

- First we prefer the **Tomcat pooling DataSource** for its performance and concurrency, so if that is available we always choose it.
- Otherwise, if **HikariCP** is available we will use it.
- If T**omcat pooling datasource** and **HikariCP** are not available then we can choose **Commons DBCP**, but we don't recommend it in production.
- Lastly, if Commons **DBCP2** is available we will use it.

If you use the **spring-boot-starter-jdbc** or **spring-boot-starter-data-jpa** '**starters**' you will automatically get a dependency to **tomcat-jdbc**.

We can configure the database properties in **application.properties** file so that **SpringBoot** will use those jdbc parameters to configure **DataSource** bean.

```
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/dojdb

spring.datasource.username=root

spring.datasource.password=root
```

For any reason if you want to have more control and configure DataSource bean by yourself then you can configure DataSource bean in a Configuration class. If you register DataSource bean then SpringBoot will not configure DataSource automatically using AutoConfiguration.

You could also customize many additional settings for connection pooling either **tomcat connection pooling (spring.datasource.tomcat.*)** or any else like **HikariCP (spring.datasource.hikari.*)**, **DBCP (spring.datasource.dbcp.*)** and **DBCP2 (spring.datasource.dbcp2.*)** as following:

```
# Number of ms to wait before throwing an exception if no connection is available.

spring.datasource.tomcat.max-wait=10000

# Maximum number of active connections that can be allocated from this pool at the same time.

spring.datasource.tomcat.max-active=50

# Validate the connection before borrowing it from the pool.

spring.datasource.tomcat.test-on-borrow=true
```

Using another Connection Pooling library

By default Spring Boot pulls in **tomcat-jdbc-{version}.jar** and uses **org.apache.tomcat.jdbc.pool.DataSource** to configure **DataSource** bean. Spring Boot check the availability of classes in the classpath in following order by default:

- org.apache.tomcat.jdbc.pool.DataSource
- com.zaxxer.hikari.HikariDataSource
- org.apache.commons.dbcp.BasicDataSource
- org.apache.commons.dbcp2.BasicDataSource

If you want to override default behavior suppose you want use **HikariDataSource** then you can exclude **tomcat-jdbc** and add **HikariCP** dependency as follows:

```
<dependency>

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

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

    <exclusions>

        <exclusion>

        <groupId>org.apache.tomcat</groupId>

        <artifactId>tomcat-jdbc</artifactId>

        </exclusion>

    </exclusions>

</dependency>



<dependency>

    <groupId>com.zaxxer</groupId>

    <artifactId>HikariCP</artifactId>

</dependency>
```

With this dependency configuration **SpringBoot** will use **HikariCP** to configure **DataSource** bean.

3. Connection to a JNDI DataSource

If you are deploying your Spring Boot application to an Application Server you might want to configure and manage your DataSource using your Application Servers built-in features and access it using JNDI.

The **spring.datasource.jndi-name** property can be used as an alternative to the **spring.datasource.url,spring.datasource.username** and **spring.datasource.password** properties to access the **DataSource** from a specific JNDI location.

For example, the following section in **application.properties** shows how you can access a Tomcat AS defined DataSource:

```
spring.datasource.jndi-name=java:tomcat/datasources/users
```

JPA & Spring Data with Spring Boot

In this section we will see how the same can be achieved using Java **Persistance** API. Spring Data provides excellent mechanism to achieve the persistence using JPA. The Java Persistence API is a standard technology that allows you to 'map' objects to relational databases. The **spring-boot-starter-data-jpa** POM provides a quick way to get started. It provides the following key dependencies:

- **Hibernate** — One of the most popular JPA implementations.
- **Spring Data JPA** — Makes it easy to implement JPA-based repositories.
- **Spring ORMs** — Core ORM support from the Spring Framework.

Update **pom.xml** with adding following line for Spring Data JPA implementation.

```
<dependency>

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

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

</dependency>
```

Entity Classes

Next is to create an entity class that maps to the underlying table. Let us create User as shown below:

```
@Entity

public class User implements Serializable{

@Id

 private Integer userId;

private String userName;

 private String userEmail;
private String address

    // setters & getters}
```

Spring Data JPA Repositories

Next is to create a repository class that will provide us with basic APIs to interact with db and also provide facility to add new APIs to interact with db. We will be using the **CrudRepository** provided by spring data. It provides us with APIs to do **CRUD** operations and some find operations like **findAll**, **findOne**, **count**.

```
package com.dineshonjava.domain;


import org.springframework.data.domain.*;

import org.springframework.data.repository.*;


public interface UserRepository extends CrudRepository {


}
```

With Spring Data we do not need to write and query for inserting data to the table. In above **UserRepository** class can have all default method provided by the **CrudRepository**.

```
@Service

public class UserService {


@Autowired
```

```
    private UserRepository userRepository;


    @Transactional(readOnly=true)

    public List<User> findAll() {

        return userRepository.findAll();

    }

}
```

Summary

In this article we saw how we moved from in-memory database to installed databases and also saw how we could use JdbcTemplate and JPA to interact with the db. We didn't have to write any sort of XML configuration and everything was managed by auto configuration provided by SpringBoot. Overall, you would have got good idea on how to use SQL Databases and Spring Boot together for persisting the application data.

MySQL Configuration with Spring Boot

Hello friends as we have already discussed in the previous chapter about how you **in memory database** or **embedded** database of Spring Boot like **H2**, **HSQL** etc. Out of the box, Spring Boot is very easy to use with the H2 or HSQL Database. If the HSQL database is found on your classpath, Spring Boot will automatically set up an in memory HSQL database for your use., I have used in-memory databases like HSQL for persisting data. But, these in-memory databases cannot be used for the production. These are only used for the development environment. But what if you want to use **MySQL**? Naturally, Spring Boot has support for MySQL and a number of other popular relational databases.

Let's see we want to deploy this application to production and we've decided to use MySQL for the database. Changing Spring Boot from HSQL to MySQL is easy to do.

Maven Configurations for MySQL Dependencies

```
<dependency>

 <groupId>mysql</groupId>

 <artifactId>mysql-connector-java</artifactId>
```

```
</dependency>
```

Here in the maven's pom.xml, you just added mysql connector. Once this is added to the classpath, Spring Boot will automatically search for the MySQL configurations and use it.

MySQL Configuration

You'll need to have a database defined for your use. For this example, I want to create a database for my use. Using the command prompt, you can log into MySQL with this command:

```
mysql -u root
```

Use the following command to create a database.

```
CREATE DATABASE dojdb;
```

There are following property we have to add the application property file application.properties and also we have to install MySQL DB at your machine firstly and after that you can go ahead with MySQL configuration.

```
spring.datasource.url=jdbc:mysql://localhost:3306/dojdb

spring.datasource.username=root

spring.datasource.password=root

spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

If you are using ORM (Hibernate) with this application then we can add one more property in the file.

```
spring.jpa.hibernate.ddl-auto=create-drop
```

If this was actually a production database, you not tell Hibernate to use the create-drop option. This tells Hibernate to recreate the database on startup. Definitely not the behavior we want. You can set this property to the following values: none, validate, update, create-drop. If this was actually a production database, you probably would want to use validate.

Summary

This is all about how to use and configure MySQL database with Spring Boot application and discussed all that needs to be changed to use MySQL with Spring Boot. When you start the project now, MySQL will be used by the Spring Boot application for the database.

Happy Spring Boot Learning!!!

Spring Data JPA using Spring Boot Application

Hello friends !!! In this tutorial we are going to discuss how to use and configure **Spring Data JPA** with **Spring Boot Application** with minimal configuration as always with Spring Boot. Spring Data JPA and Hibernate (as JPA implementation) will be used to implement the data access layer.

Spring Data?

Spring Data is a high level project developed by Spring community aimed at simplifying the data access operations for the applications. If you are using Spring Data in your project, you are not going to write most of the low level data access operations like writing SQL query DAO classes etc. Spring Data will generate everything dynamically at run-time by creating the proxy instances of your abstract repositories and perform the required operations.

There are several sub projects provided by Spring Data like Spring Data JPA, Spring Data Solr, Spring Data MongoDB, Spring Data REST etc. Here we are going to discuss one of them Spring Data JPA.

Spring Data JPA?

The Spring Data JPA is the implementation of JPA for simplifying operation like data accessing, querying, pagination and removes lots of boilerplate codes. The Java Persistence API is a standard technology that allows you to 'map' objects to relational databases. The spring-boot-starter-data-jpa POM provides a quick way to get started. It provides the following key dependencies:

- **Hibernate** — One of the most popular JPA implementations.
- **Spring Data JPA** — Makes it easy to implement JPA-based repositories.
- **Spring ORMs** — Core ORM support from the Spring Framework

Spring Data Repositories

Spring data provides the abstract repositories, which are implemented at run-time by the spring container and perform the CRUD operations. As a developer we have to just provide the abstract methods in the interface. This reduces the amount of boilerplate code required to write data access layers. There are following are base interfaces providing by Spring Data.

Repository

It is the central interface in the spring data repository abstraction. This is a marker interface.

CrudRepository

CrudRepository provides methods for the CRUD operations. This interface extends the Repository interface. If you are extending the CrudRepository, there is no need for implementing your own methods. Just extend this interface and leave it as blank.

JpaRepository

This is the special version of CrudRepository specific to the JPA technology. Recommended to use CrudRepository because it will not tie your application with any specific store implementations.

PagingAndSortingRepository

It is specialized version for the paging operations and extension of CrudRepository.

Spring Data JPA and Spring Boot Application

Here I am going make at REST API application for providing the booking information of train ticket and also we can the book the train ticket using API. Here I am using MySQL database in a Spring Boot web application, using less code and configurations as possible. First we have to create the project so I am using here Spring Boot Web Initializr for creating project with selecting maven build and required dependencies for this project like WEB, SPRING DATA JPA, and MYSQL. Now let’s see required part of the maven file for this project.

```
<?xml version="1.0" encoding="UTF-8"?>

<project         xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>



 <groupId>com.dineshonjava.sdjpa</groupId>

 <artifactId>SpringBootJPASpringData</artifactId>

 <version>0.0.1-SNAPSHOT</version>

 <packaging>jar</packaging>



 <name>SpringBootJPASpringData</name>

 <description>SpringBootJPASpringData project for Spring
Boot with Spring Data JPA implementation</description>



 <parent>

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

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

  <version>1.4.0.RELEASE</version>

  <relativePath/> <!-- lookup parent from repository -->

 </parent>



 <properties>

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

  <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
```

```
  <java.version>1.8</java.version>
 </properties>

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

  <!-- <dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
  </dependency> -->
  <dependency>
   <groupId>mysql</groupId>
   <artifactId>mysql-connector-java</artifactId>
   <scope>runtime</scope>
  </dependency>
```

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

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



</project>
```

If you look at the above maven dependencies.

- **spring-boot-starter-data-jpa** dependency will download the files required for spring data jpa.
- **spring-boot-starter-web** is required because it is web based application to expose the REST endpoints.
- **mysql-connector-java** dependency will download the files required for mysql database.

Configuration file application.properties

```
# DataSource settings: set here your own configurations for
the database

# connection. In this example we have "dojsb" as database
name and

# "root" as username and password.

spring.datasource.url = jdbc:mysql://localhost:3307/dojdb

spring.datasource.username = root

spring.datasource.password = root


# Keep the connection alive if idle for a long time (needed
in production)

spring.datasource.testWhileIdle = true

spring.datasource.validationQuery = SELECT 1


# Show or not log for each sql query

spring.jpa.show-sql = true


# Hibernate ddl auto (create, create-drop, update)

spring.jpa.hibernate.ddl-auto = create


# Naming strategy

spring.jpa.hibernate.naming-strategy                     =
org.hibernate.cfg.ImprovedNamingStrategy
```

```
# Use spring.jpa.properties.* for Hibernate native properties (the prefix is

# stripped before adding them to the entity manager)



# The SQL dialect makes Hibernate generate better SQL for the chosen database

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect



server.port = 8181
```

Using hibernate configuration **ddl-auto = update** the database schema will be automatically **created** (and **updated**), creating tables and columns, accordingly to java entities found in the project. For this application I am using server port is **8181**.

Entity file Booking.java

Create an entity class representing a table in your db. Here I am using **Booking.java** as entity files which map to the **booking** table in the database **dojdb**.

```
/**

 *

 */

package com.dineshonjava.sdjpa.models;



import java.io.Serializable;

import java.util.Date;



import javax.persistence.Column;
import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;


/**

 * @author Dinesh.Rajput

 *

 */

@Entity

@Table(name = "BOOKING")

public class Booking implements Serializable{


 /**

  *

  */

 private static final long serialVersionUID = 1L;

 @Id

 @GeneratedValue(strategy = GenerationType.AUTO)

 Long bookingId;

 @Column

 String psngrName;
```

```
@Column
String departure;
@Column
String destination;
@Column
Date travelDate;

public Long getBookingId() {
 return bookingId;
}

public void setBookingId(Long bookingId) {
 this.bookingId = bookingId;
}

public String getPsngrName() {
 return psngrName;
}

public void setPsngrName(String psngrName) {
 this.psngrName = psngrName;
}
```

```
public String getDeparture() {
 return departure;
}

public void setDeparture(String departure) {
 this.departure = departure;
}

public String getDestination() {
 return destination;
}

public void setDestination(String destination) {
 this.destination = destination;
}

public Date getTravelDate() {
 return travelDate;
}

public void setTravelDate(Date travelDate) {
 this.travelDate = travelDate;
}
```

```
}
```

The **@Entity** annotation marks this class as a JPA entity. The **@Table** annotation specifies the db table’s name.

The Data Access Object (Repository): BookingRepository.java

Repository is needed to works with entities in database’s table, with methods like ***save, delete, update,*** etc. With Spring Data JPA a DAO for your entity is simply created by extending the CrudRepository interface provided by Spring. The following methods are some of the ones available from such interface: ***save***, ***delete***, ***deleteAll***, ***findOne*** and ***findAll***.

```
package com.dineshonjava.sdjpa.models;


import org.springframework.data.repository.CrudRepository;

import
org.springframework.transaction.annotation.Transactional;


@Transactional

public interface BookingRepository extends CrudRepository
{


 /**

    * This method will find an Boooking instance in the
database by its departure.
```

```
    * Note that this method is not implemented and its
working code will be

    * automatically generated from its signature by Spring
Data JPA.

    */

   public Booking findByDeparture(String departure);

}
```

Adding Rest Controller BookingController.java

Here adding one more file for accessing Rest APIs is RestController it provide me some URLs for CRUD operation the booking table.

```
/**

 *

 */

package com.dineshonjava.sdjpa.controller;



import java.util.Date;



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

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

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

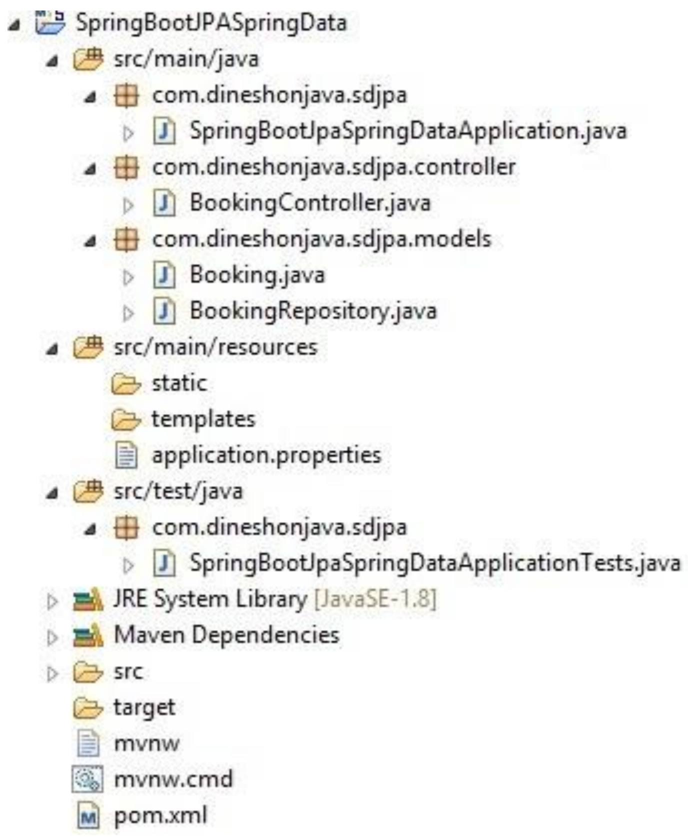
```
import
org.springframework.web.bind.annotation.RestController;


import com.dineshonjava.sdjpa.models.Booking;

import com.dineshonjava.sdjpa.models.BookingRepository;


/**

 * @author Dinesh.Rajput

 *

 */

@RestController

@RequestMapping("/booking")

public class BookingController {


 @Autowired

 BookingRepository bookingRepository;

 /**

  * GET /create  --> Create a new booking and save it in
the database.

  */

 @RequestMapping("/create")

 public Booking create(Booking booking) {

  booking.setTravelDate(new Date());

  booking = bookingRepository.save(booking);
```

```
     return booking;
 }


 /**
  * GET /read  --> Read a booking by booking id from the
database.
  */
 @RequestMapping("/read")
 public Booking read(@RequestParam Long bookingId) {
  Booking booking = bookingRepository.findOne(bookingId);
     return booking;
 }


 /**
  * GET /update  --> Update a booking record and save it
in the database.
  */
 @RequestMapping("/update")
 public  Booking  update(@RequestParam  Long  bookingId,
@RequestParam String psngrName) {
  Booking booking = bookingRepository.findOne(bookingId);
  booking.setPsngrName(psngrName);
  booking = bookingRepository.save(booking);
     return booking;
 }
 /**
  * GET /delete  --> Delete a booking from the database.
  */
 @RequestMapping("/delete")
 public String delete(@RequestParam Long bookingId) {
  bookingRepository.delete(bookingId);
     return "booking #"+bookingId+" deleted successfully";
 }
}
```

That’s all! The connection with the database is done when we run the following file as Java Application or Spring Boot application.

```
import org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplicat
ion;


@SpringBootApplication
public class SpringBootJpaSpringDataApplication {


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

}
```

Project Structure:



```
SpringBootJPASpringData
  src/main/java
    com.dineshonjava.sdjpa
      SpringBootJpaSpringDataApplication.java
    com.dineshonjava.sdjpa.controller
      BookingController.java
    com.dineshonjava.sdjpa.models
      Booking.java
      BookingRepository.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
    com.dineshonjava.sdjpa
      SpringBootJpaSpringDataApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

Test the controller launching the Spring Boot web application and using these urls:

Creating New Record:

/booking/create?psngrName=Dinesh&departure=Noida&destination=Pune create a new booking with an auto-generated.

For Example:

http://localhost:8181/booking/create?psngrName=Arnav&destination=USA&departure=Delhi

{

• *bookingId: 4,*

• *psngrName: "Arnav",*

• *departure: "Delhi",*

• *destination: "USA",*

• *travelDate: 1470828738680*

}

Reading a Record:

/booking/read?bookingId=[bookingId]: Read the booking with the passed bookingId.

For Example:

http://localhost:8181/booking/read?bookingId=1

{

• *bookingId: 1,*

• *psngrName: "Dinesh",*

• *departure: "Noida",*

• *destination: "Pune",*

• *travelDate: 1470828563000*

}

Updating a Record:

/booking/update?bookingId=[bookingId]&psngrName=Sweety: Update the booking for a given booking id.

For Example:

http://localhost:8181/booking/update?bookingId=5&psngrName=Suresh

{

• *bookingId: 5,*

• *psngrName: "Suresh",*

• *departure: "Agra",*

• *destination: "Noida",*

• *travelDate: 1470828994000*

}

Deleting a Record:

/booking/delete? bookingId=[bookingId]: Delete a booking for given booking id.

distributionUrl=https://repo1.maven.org/maven2/org/apache/maven/apache-maven/3.3.9/apache-maven-3.3.9-bin.

```
/**
 *
 */
package com.dineshonjava.sdjpa.controller;

import java.util.Date;

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

import com.dineshonjava.sdjpa.models.Booking;
import com.dineshonjava.sdjpa.models.BookingRepository;

/**
 * @author Dinesh.Rajput
 *
 */
@RestController
```

```
@RequestMapping("/booking")
public class BookingController {

        @Autowired
        BookingRepository bookingRepository;
        /**
         * GET /create  --> Create a new booking and save it in the database.
         */
        @RequestMapping("/create")
        public Booking create(Booking booking) {
                booking.setTravelDate(new Date());
                booking = bookingRepository.save(booking);
           return booking;
        }

        /**
         * GET /read  --> Read a booking by booking id from the database.
         */
        @RequestMapping("/read")
        public Booking read(@RequestParam Long bookingId) {
                Booking booking = bookingRepository.findOne(bookingId);
           return booking;
        }

        /**
         * GET /update  --> Update a booking record and save it in the database.
```

```
     */
    @RequestMapping("/update")
    public Booking update(@RequestParam Long bookingId, @RequestParam
String psngrName) {
        Booking booking = bookingRepository.findOne(bookingId);
        booking.setPsngrName(psngrName);
        booking = bookingRepository.save(booking);
      return booking;
    }

    /**
     * GET /delete  --> Delete a booking from the database.
     */
    @RequestMapping("/delete")
    public String delete(@RequestParam Long bookingId) {
        bookingRepository.delete(bookingId);
      return "booking #"+bookingId+" deleted successfully";
    }
}
/**
 *
 */
package com.dineshonjava.sdjpa.models;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * @author Dinesh.Rajput
 *
 */
@Entity
@Table(name = "BOOKING")
public class Booking implements Serializable{

	/**
	 *
	 */
	private static final long serialVersionUID = 1L;
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	Long bookingId;
	@Column
	String psngrName;
	@Column
```

```
String departure;
@Column
String destination;
@Column
Date travelDate;

public Long getBookingId() {
	return bookingId;
}

public void setBookingId(Long bookingId) {
	this.bookingId = bookingId;
}

public String getPsngrName() {
	return psngrName;
}

public void setPsngrName(String psngrName) {
	this.psngrName = psngrName;
}

public String getDeparture() {
	return departure;
}
```

```
	public void setDeparture(String departure) {
		this.departure = departure;
	}

	public String getDestination() {
		return destination;
	}

	public void setDestination(String destination) {
		this.destination = destination;
	}

	public Date getTravelDate() {
		return travelDate;
	}

	public void setTravelDate(Date travelDate) {
		this.travelDate = travelDate;
	}



}
package com.dineshonjava.sdjpa.models;

import org.springframework.data.repository.CrudRepository;
import org.springframework.transaction.annotation.Transactional;
```

```
@Transactional
public interface BookingRepository extends CrudRepository<Booking, Long> {

    /**
     * This method will find an Boooking instance in the database by its departure.
     * Note that this method is not implemented and its working code will be
     * automatically generated from its signature by Spring Data JPA.
     */
    public Booking findByDeparture(String departure);
}
package com.dineshonjava.sdjpa.models;

import org.springframework.data.repository.CrudRepository;
import org.springframework.transaction.annotation.Transactional;

@Transactional
public interface BookingRepository extends CrudRepository<Booking, Long> {

    /**
     * This method will find an Boooking instance in the database by its departure.
     * Note that this method is not implemented and its working code will be
     * automatically generated from its signature by Spring Data JPA.
     */
    public Booking findByDeparture(String departure);
}
```

```
/**
 *
 */
package com.dineshonjava.sdjpa.models;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

/**
 * @author Dinesh.Rajput
 *
 */
@Entity
@Table(name = "BOOKING")
public class Booking implements Serializable{

	/**
	 *
	 */
```

```
private static final long serialVersionUID = 1L;
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
Long bookingId;
@Column
String psngrName;
@Column
String departure;
@Column
String destination;
@Column
Date travelDate;

public Long getBookingId() {
        return bookingId;
}

public void setBookingId(Long bookingId) {
        this.bookingId = bookingId;
}

public String getPsngrName() {
        return psngrName;
}

public void setPsngrName(String psngrName) {
	this.psngrName = psngrName;
}

public String getDeparture() {
	return departure;
}

public void setDeparture(String departure) {
	this.departure = departure;
}

public String getDestination() {
	return destination;
}

public void setDestination(String destination) {
	this.destination = destination;
}

public Date getTravelDate() {
	return travelDate;
}

public void setTravelDate(Date travelDate) {
	this.travelDate = travelDate;
}
```

```
}
package com.dineshonjava.sdjpa.models;

import org.springframework.data.repository.CrudRepository;
import org.springframework.transaction.annotation.Transactional;

@Transactional
public interface BookingRepository extends CrudRepository<Booking, Long> {

	/**
	 * This method will find an Boooking instance in the database by its departure.
	 * Note that this method is not implemented and its working code will be
	 * automatically generated from its signature by Spring Data JPA.
	 */
	public Booking findByDeparture(String departure);
}
package com.dineshonjava.sdjpa;

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

@SpringBootApplication
public class SpringBootJpaSpringDataApplication {
	public static void main(String[] args) {
		SpringApplication.run(SpringBootJpaSpringDataApplication.class,
args);
	}
}
# DataSource settings: set here your own configurations for the database
# connection. In this example we have "dojsb" as database name and
# "root" as username and password.
spring.datasource.url = jdbc:mysql://localhost:3307/dojdb
spring.datasource.username = root
spring.datasource.password = root

# Keep the connection alive if idle for a long time (needed in production)
spring.datasource.testWhileIdle = true
spring.datasource.validationQuery = SELECT 1

# Show or not log for each sql query
spring.jpa.show-sql = true

# Hibernate ddl auto (create, create-drop, update)
spring.jpa.hibernate.ddl-auto = create

# Naming strategy
spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy

# Use spring.jpa.properties.* for Hibernate native properties (the prefix is
# stripped before adding them to the entity manager)


# The SQL dialect makes Hibernate generate better SQL for the chosen database
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect


server.port = 8181
```

Summary

In this chapter I have explained how to use Spring Data JPA and Spring Boot frameworks. Also explained in short about the Spring Data Repositories and query methods.

Happy Spring Boot Learning!!!

Spring Boot and MongoDB in REST Application

Hello friends !!! In this tutorial we are going to discuss about using NoSQL database MongoDB with Spring Boot Application. Here I will make a Spring Boot REST Application which provides REST APIs for make booking, read booking, update booking and delete booking.

Spring Boot with NoSQL

Spring Data provides additional projects that help you access a variety of NoSQL technologies including MongoDB, Neo4J, Elasticsearch, Solr, Redis, Gemfire, Couchbase and Cassandra. Spring Boot provides auto-configuration for Redis, MongoDB, Neo4j, Elasticsearch, Solr and Cassandra.

NoSQL MongoDB

NoSQL is a non-relational database management system, different from traditional relational database management systems in some significant ways. MongoDb is a Open Source database written in C++. It can be used to store data for very high performance applications (for example Foursquare is using it in production).
MongoDB stores data as documents. So it is a document oriented database.

MongoDB with Spring Boot & Spring Data

Spring Boot and **Spring Data** make it even easier to get a simple application up and running. With a little bit of configuration and minimal code, you can quickly create and deploy a **MongoDB-based** application. Spring Boot offers several conveniences for working with MongoDB, including the **spring-boot-starter-data-mongodb'Starter'**.

```
<dependency>

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

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

</dependency>
```

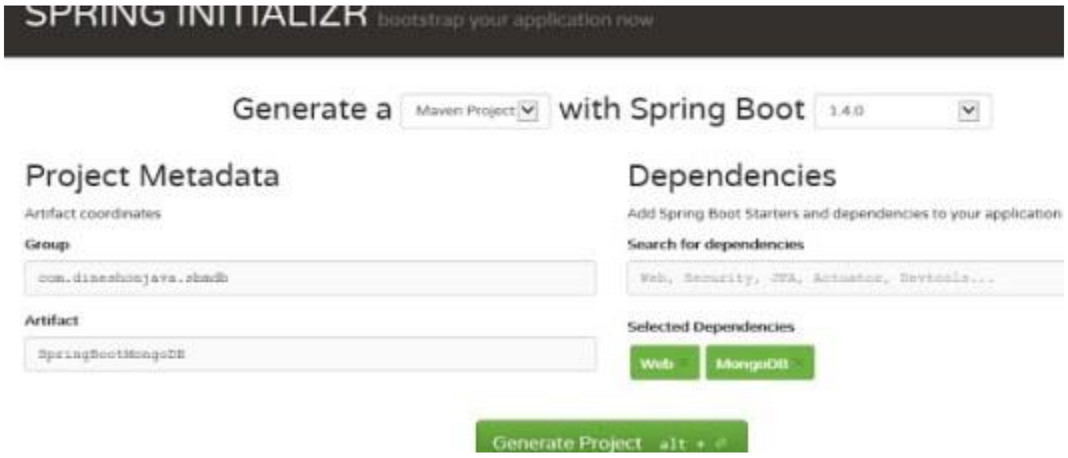
You can set ***spring.data.mongodb.uri*** property to change the URL and configure additional settings such as the replica set:

```
spring.data.mongodb.uri=mongodb://user:secret@mongo1.example.com:12345,mongo
2.example.com:23456/test
```

Spring Data includes repository support for MongoDB. As with the JPA repositories discussed earlier, the basic principle is that queries are constructed for you automatically based on method names.

MongoDB with Spring Boot Application

Let's start to develop simple REST API project for ticket booking with Spring Boot and MongoDB. First, make sure you have installed MongoDB and are able to run it. Second make sure you have installed Maven 4.X.
Next, we are going to create a Spring Boot web application. You can do so by using the web-based wizard here. In the dependencies section, you'll want to select: Web, and MongoDB.


SPRING INITIALIZR bootstrap your application now
Project Metadata
Artifact coordinates
Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies
Web, Security, JPA, Actuator, Devtools...
Generate Project alt +

Generate a Maven Project with Spring Boot 1.4.0

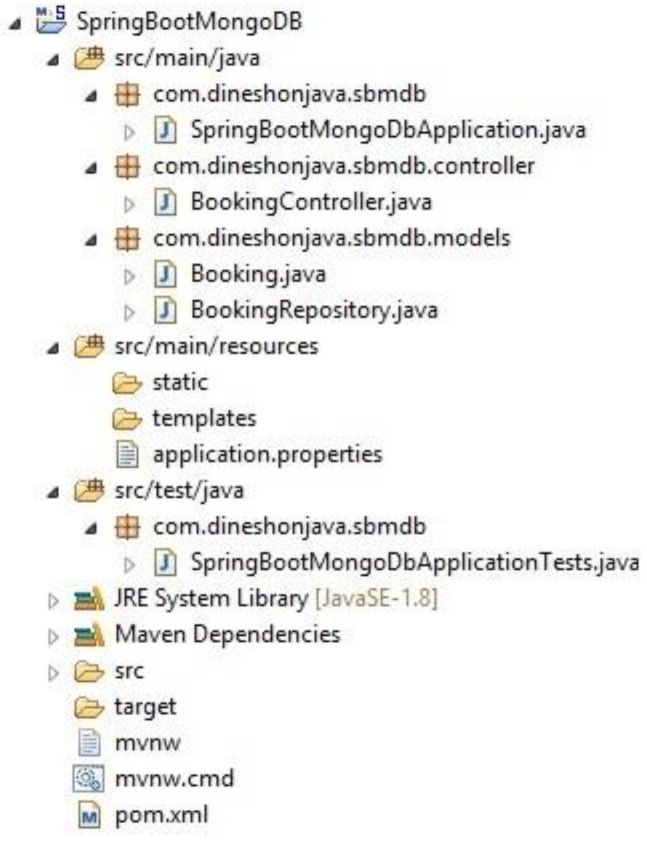
Group: com.dineshonjava.sbmdb

Artifact: SpringBootMongoDB

Selected Dependencies: Web, MongoDB

Import into STS IDE

After downloading from the web interface now unzip the project "**SpringBootMongoDB**" and import in the STS IDE or Eclipse as Maven project. As below project structure look like. Now that the basic maven based Spring boot is ready.



```
SpringBootMongoDB
  src/main/java
    com.dineshonjava.sbmdb
      SpringBootMongoDbApplication.java
    com.dineshonjava.sbmdb.controller
      BookingController.java
    com.dineshonjava.sbmdb.models
      Booking.java
      BookingRepository.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
    com.dineshonjava.sbmdb
      SpringBootMongoDbApplicationTests.java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

Let's discuss about the application files

Configuration file *application.properties*

```
spring.data.mongodb.database=dojdb

spring.data.mongodb.host=localhost

spring.data.mongodb.port=27017
```

There are no need add any configuration to this file spring boot provide auto-configuration for mongodb when added starter for mongodb to application maven file. If you want to change mongo configuration default you can add above properties to the **application.properties** file.

Here we are telling Spring Data the host and port of our Mongo instance. We also give it a database name, and if the database doesn't already exist it will be created for us.

Maven Configuration *pom.xml*

Following dependencies to be added with maven file

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>



 <groupId>com.dineshonjava.sbmdb</groupId>

 <artifactId>SpringBootMongoDB</artifactId>

 <version>0.0.1-SNAPSHOT</version>

 <packaging>jar</packaging>



 <name>SpringBootMongoDB</name>

 <description>SpringBootMongoDB project for Spring Boot with MongoDB providing
APIs</description>



 <parent>

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

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

  <version>1.4.0.RELEASE</version>

  <relativePath/> <!-- lookup parent from repository -->

 </parent>
```

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

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

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

```
 <build>

  <plugins>

   <plugin>

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

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

   </plugin>

  </plugins>

 </build>




</project>
```

Let me create a model class to hold Booking details retrieved from DB.
Now, we need to model our documents. Let’s call ours ‘**Booking**’ and give it a make, model, and description. Here is our Java class to accomplish this:
Booking.java

```
/**

 *

 */

package com.dineshonjava.sbmdb.models;



import java.util.Date;
import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;


/**

 * @author Dinesh.Rajput

 *

 */

@Document

public class Booking{


 @Id

 String id;

 String psngrName;

 String departure;

 String destination;

 Date travelDate;


 public String getId() {

  return id;

 }


 public void setId(String id) {

  this.id = id;
```

```
}


public String getPsngrName() {
 return psngrName;
}


public void setPsngrName(String psngrName) {
 this.psngrName = psngrName;
}


public String getDeparture() {
 return departure;
}


public void setDeparture(String departure) {
 this.departure = departure;
}


public String getDestination() {
 return destination;
}


public void setDestination(String destination) {
  this.destination = destination;
 }


 public Date getTravelDate() {
  return travelDate;
 }


 public void setTravelDate(Date travelDate) {
  this.travelDate = travelDate;
 }



}
```

@Id- id provided by Mongo for a document.
@Document- provides a collection name.
Lets add Repository class to interact with the DB
BookingRepository.java
A Repository is a way to manage data objects in Spring Data. For most common methods – like saving a document, updating it, deleting it, or finding it by id – Spring Data will automatically implement the necessary logic.

```
package com.dineshonjava.sbmdb.models;


import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.transaction.annotation.Transactional;


@Transactional

public interface BookingRepository extends MongoRepository {


 /**

   * This method will find an Boooking instance in the database by its departure.

   * Note that this method is not implemented and its working code will be

   * automatically generated from its signature by Spring Data JPA.

   */

  public Booking findByDeparture(String departure);

}
```

The **MongoRepository** provides basic CRUD operation methods and also an API to find all documents in the collection.

Implementing RestController for Create and Get Details API

BookingController.java

```
/**

 *

 */

package com.dineshonjava.sbmdb.controller;


import java.util.Date;
import java.util.HashMap;

import java.util.List;

import java.util.Map;


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

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

import
org.springframework.web.bind.annotation.RequestParam;

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


import com.dineshonjava.sbmdb.models.Booking;

import com.dineshonjava.sbmdb.models.BookingRepository;


/**

 * @author Dinesh.Rajput

 *

 */

@RestController

@RequestMapping("/booking")

public class BookingController {


 @Autowired
 BookingRepository bookingRepository;

 /**

  * GET /create  --> Create a new booking and save it in
the database.

  */

 @RequestMapping("/create")

 public Map<String, Object> create(Booking booking) {

  booking.setTravelDate(new Date());

  booking = bookingRepository.save(booking);

  Map<String, Object> dataMap = new HashMap<String,
Object>();

  dataMap.put("message", "Booking created successfully");

  dataMap.put("status", "1");

  dataMap.put("booking", booking);

     return dataMap;

 }



 /**

  * GET /read  --> Read a booking by booking id from the
database.

  */

 @RequestMapping("/read")

 public Map<String, Object> read(@RequestParam String
bookingId) {

  Booking booking = bookingRepository.findOne(bookingId);
  Map<String, Object> dataMap = new HashMap<String,
Object>();

  dataMap.put("message", "Booking found successfully");

  dataMap.put("status", "1");

  dataMap.put("booking", booking);

     return dataMap;

 }



 /**

  * GET /update  --> Update a booking record and save it in
the database.

  */

 @RequestMapping("/update")

 public Map<String, Object> update(@RequestParam String
bookingId, @RequestParam String psngrName) {

  Booking booking = bookingRepository.findOne(bookingId);

  booking.setPsngrName(psngrName);

  booking = bookingRepository.save(booking);

  Map<String, Object> dataMap = new HashMap<String,
Object>();

  dataMap.put("message", "Booking updated successfully");

  dataMap.put("status", "1");

  dataMap.put("booking", booking);

     return dataMap;

 }
 /**
  * GET /delete  --> Delete a booking from the database.
  */
 @RequestMapping("/delete")
 public Map<String, Object> delete(@RequestParam String
bookingId) {
  bookingRepository.delete(bookingId);
  Map<String, Object> dataMap = new HashMap<String,
Object>();
  dataMap.put("message", "Booking deleted successfully");
  dataMap.put("status", "1");
     return dataMap;
 }


 /**
  * GET /read  --> Read all booking from the database.
  */
 @RequestMapping("/read-all")
 public Map<String, Object> readAll() {
  List<Booking> bookings = bookingRepository.findAll();
  Map<String, Object> dataMap = new HashMap<String,
Object>();
  dataMap.put("message", "Booking found successfully");
  dataMap.put("totalBooking", bookings.size());
```

```
   dataMap.put("status", "1");

   dataMap.put("bookings", bookings);

      return dataMap;

  }

}
```

Lets run the application by either using Spring Boot maven plug-in i.e. by running ***mvn spring-boot:run*** or by running the main class **SpringBootMongoDbApplication.java** from Eclipse or your IDE.

```
package com.dineshonjava.sbmdb;



import org.springframework.boot.SpringApplication;

import
org.springframework.boot.autoconfigure.SpringBootApplicatio
n;



@SpringBootApplication

public class SpringBootMongoDbApplication {



 public static void main(String[] args) {

  SpringApplication.run(SpringBootMongoDbApplication.class,
args);

 }

}
```

Creating New Record:
/booking/create?psngrName=Dinesh&departure=Noida&destination=Pune create a new booking with an auto-generated.

For Example:
http://localhost:8080/booking/create?psngrName=Vinesh&destination=Delhi&departure=Farrukhabad
{
• booking:
{
o id: “57abe8d42ca52424e8e88027”,
o psngrName: “Vinesh”,
o departure: “Farrukhabad”,
o destination: “Delhi”,
o travelDate: 1470884052977
},
• message: “Booking created successfully”,
• status: “1”
}

Reading a Record:
/booking/read?id=[bookingId]: Read the booking with the passed bookingId.
For Example:
http://localhost:8080/booking/read?bookingId=57abe8932ca52424e8e88024
{
• booking:
{
o id: “57abe8932ca52424e8e88024”,
o psngrName: “Arnav”,
o departure: “Pune”,
o destination: “USA”,
o travelDate: 1470883987113
},
• message: “Booking found successfully”,
• status: “1”
}

Updating a Record:
/booking/update?bookingId=[bookingId]&psngrName=Sweety: Update the booking for a given booking id.
For Example:
http://localhost:8080/booking/update?bookingId=57abe8a72ca52424e8e88025&psngrName=Anamika

{
• booking:
{
o id: “57abe8a72ca52424e8e88025”,
o psngrName: “Anamika”,
o departure: “Noida”,
o destination: “USA”,
o travelDate: 1470884007369
},
• message: “Booking updated successfully”,
• status: “1”
}

Read All Records
http://localhost:8080/booking/read-all
{
• totalBooking: 5,
• message: “Booking found successfully”,
• bookings:
[
o {
♣ id: “57abe8322ca52424e8e88023”,
♣ psngrName: “Dinesh”,
♣ departure: “Noida”,
♣ destination: “Pune”,
♣ travelDate: 1470883890814
},
o {
♣ id: “57abe8932ca52424e8e88024”,
♣ psngrName: “Arnav”,
♣ departure: “Pune”,
♣ destination: “USA”,
♣ travelDate: 1470883987113
},
o {
♣ id: “57abe8a72ca52424e8e88025”,
♣ psngrName: “Sweety”,
♣ departure: “Noida”,
♣ destination: “USA”,
♣ travelDate: 1470884007369
},
o {

♣ id: “57abe8bf2ca52424e8e88026”,
♣ psngrName: “Adesh”,
♣ departure: “Kannauj”,
♣ destination: “Noida”,
♣ travelDate: 1470884031444
},
o {
♣ id: “57abe8d42ca52424e8e88027”,
♣ psngrName: “Vinesh”,
♣ departure: “Farrukhabad”,
♣ destination: “Delhi”,
♣ travelDate: 1470884052977
}
],
• status: “1”
}

Deleting a Record:
/booking/delete? bookingId=[bookingId]: Delete a booking for given booking id.

For Example:
http://localhost:8080/booking/delete?bookingId=57ac0d681ff9b3117caf7c85
{
• message: “Booking deleted successfully”,
• status: “1”
}

Summary

In this chapter I have explained how to use Spring Data MongoDB and Spring Boot frameworks. Great you set up a MongoDB server and wrote a simple application that uses Spring Data MongoDB to save objects to and fetch them from a database — all without writing a concrete repository implementation.
Whole Source code here
distributionUrl=https://repo1.maven.org/maven2/org/apache/maven/apache-maven/3.3.9/apache-maven-3.3.9-bin.

```
/**
 *
 */
package com.dineshonjava.sbmdb.controller;
```

```
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

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

import com.dineshonjava.sbmdb.models.Booking;
import com.dineshonjava.sbmdb.models.BookingRepository;

/**
 * @author Dinesh.Rajput
 *
 */
@RestController
@RequestMapping("/booking")
public class BookingController {

	@Autowired
	BookingRepository bookingRepository;
	/**
	 * GET /create  --> Create a new booking and save it in the database.
	 */
	@RequestMapping("/create")
	public Map<String, Object> create(Booking booking) {
		booking.setTravelDate(new Date());
		booking = bookingRepository.save(booking);
		Map<String, Object> dataMap = new HashMap<String, Object>();
		dataMap.put("message", "Booking created successfully");
		dataMap.put("status", "1");
		dataMap.put("booking", booking);
	    return dataMap;
	}

	/**
	 * GET /read  --> Read a booking by booking id from the database.
	 */
	@RequestMapping("/read")
	public Map<String, Object> read(@RequestParam String bookingId) {
		Booking booking = bookingRepository.findOne(bookingId);
		Map<String, Object> dataMap = new HashMap<String, Object>();
		dataMap.put("message", "Booking found successfully");
		dataMap.put("status", "1");
```

```
                dataMap.put("booking", booking);
            return dataMap;
        }

        /**
         * GET /update  --> Update a booking record and save it in the database.
         */
        @RequestMapping("/update")
        public Map<String, Object> update(@RequestParam String bookingId,
@RequestParam String psngrName) {
                Booking booking = bookingRepository.findOne(bookingId);
                booking.setPsngrName(psngrName);
                booking = bookingRepository.save(booking);
                Map<String, Object> dataMap = new HashMap<String, Object>();
                dataMap.put("message", "Booking updated successfully");
                dataMap.put("status", "1");
                dataMap.put("booking", booking);
            return dataMap;
        }

        /**
         * GET /delete  --> Delete a booking from the database.
         */
        @RequestMapping("/delete")
        public Map<String, Object> delete(@RequestParam String bookingId) {
                bookingRepository.delete(bookingId);
                Map<String, Object> dataMap = new HashMap<String, Object>();
                dataMap.put("message", "Booking deleted successfully");
                dataMap.put("status", "1");
            return dataMap;
        }

        /**
         * GET /read  --> Read all booking from the database.
         */
        @RequestMapping("/read-all")
        public Map<String, Object> readAll() {
                List<Booking> bookings = bookingRepository.findAll();
                Map<String, Object> dataMap = new HashMap<String, Object>();
                dataMap.put("message", "Booking found successfully");
                dataMap.put("totalBooking", bookings.size());
                dataMap.put("status", "1");
                dataMap.put("bookings", bookings);
            return dataMap;
        }
}
```

```
/**
 *
 */
package com.dineshonjava.sbmdb.models;

import java.util.Date;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

/**
 * @author Dinesh.Rajput
 *
 */
@Document
public class Booking{

	@Id
	String id;
	String psngrName;
	String departure;
	String destination;
	Date travelDate;

	public String getId() {
		return id;
	}

	public void setId(String id) {
		this.id = id;
	}

	public String getPsngrName() {
		return psngrName;
	}

	public void setPsngrName(String psngrName) {
		this.psngrName = psngrName;
	}

	public String getDeparture() {
		return departure;
	}

	public void setDeparture(String departure) {
```

```
                this.departure = departure;
        }

        public String getDestination() {
                return destination;
        }

        public void setDestination(String destination) {
                this.destination = destination;
        }

        public Date getTravelDate() {
                return travelDate;
        }

        public void setTravelDate(Date travelDate) {
                this.travelDate = travelDate;
        }


}

package com.dineshonjava.sbmdb.models;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.transaction.annotation.Transactional;

@Transactional
public interface BookingRepository extends MongoRepository<Booking, String> {

        /**
          * This method will find an Boooking instance in the database by its departure.
          * Note that this method is not implemented and its working code will be
          * automatically generated from its signature by Spring Data JPA.
          */
         public Booking findByDeparture(String departure);
}

package com.dineshonjava.sbmdb;

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

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

package com.dineshonjava.sbmdb;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringBootMongoDbApplicationTests {

	@Test
	public void contextLoads() {
	}

}
```

What’s New in the Spring Framework 5.X?

Hello friends!!! It is great feeling to heard about Spring Community has been released the first Spring Framework 5.0 milestone (Spring Framework 5.0 M1) at 28 July 2016. This initial milestone covers all of our baseline upgrade efforts, in particular going JDK 8+ all across the codebase, plus support for JUnit 5. And it delivers fundamental JDK 9 compatibility at runtime as well as for the framework build and test suite.

August 12, 2016

What’s New in the Spring Framework 5.X?

Hello friends!!! It is great feeling to heard about Spring Community has been released the first Spring Framework 5.0 milestone (Spring Framework 5.0 M1) at 28 July 2016. This initial milestone covers all of our baseline upgrade efforts, in particular going JDK 8+ all across the codebase, plus support for JUnit 5. And it delivers fundamental JDK 9 compatibility at runtime as well as for the framework build and test suite.

JDK 8+9 and Java EE 7 Baseline

- Entire framework codebase based on Java 8 source code level now.
  - Improved readability through inferred generics etc.
  - Conditional support for Java 8 features now in straight code.
- Java EE 7 API level required in Spring's corresponding modules now.
  - Servlet 3.1, JMS 2.0, JPA 2.1, Bean Validation 1.1
  - Recent servers: e.g. Tomcat 8.5+, Jetty 9.3+, WildFly 10+
- Full compatibility with JDK 9 as of July 2016.
  - Project spring-framework can be built on JDK 9; test suite passes.

Removed Packages, Classes and Methods

- Package **mock.staticmock** removed from spring-aspects module.
  - No support for **AnnotationDrivenStaticEntityMockingControl** anymore.
- Packages **web.view.tiles2** and **orm.hibernate3/hibernate4** dropped.
  - Minimum requirement: **Tiles 3** and **Hibernate 5** now.
- Dropped support: **Portlet, Velocity, JasperReports, XMLBeans, JDO, Guava**.
  - Recommendation: Stay on **Spring Framework 4.3.x** for those if needed.
- Many deprecated classes and methods removed across the codebase.
  - A few compromises made for commonly used methods in the ecosystem.

Core Container Improvements

- JDK 8+ enhancements
  - Efficient method parameter access based on Java 8 reflection enhancements.
  - Selective declarations of Java 8 default methods in core Spring interfaces.
  - Consistent use of JDK 7 Charset and StandardCharsets enhancements.
- JDK 9 preparations
  - Consistent instantiation via constructors (with revised exception handling)
- XML configuration namespaces streamlined towards unversioned schemas.
  - Always resolved against latest xsd files; no support for deprecated features.
  - Version-specific declarations still supported but validated against latest schema.
- Resource abstraction provides **isFile** indicator for defensive **getFile** access.

General Web Improvements

- Unified support for media type resolution through **MediaTypeFactory** delegate.
- Full **Servlet 3.1** signature support in Spring-provided Filter implementations.

- Support for **Protobuf 3.0** (currently beta 4).

Reactive Programming Model

- **spring-core** DataBuffer and Encoder/Decoder abstractions with non-blocking semantics.
- **spring-web** HTTP message codec implementations with JSON (Jackson) and XML (JAXB) support.
- New **spring-web-reactive** module with reactive support for the **@Controller** programming model adapting Reactive Streams to **Servlet 3.1** containers as well as **non-Servlet** runtimes such as **Netty** and **Undertow**.
- New **WebClient** with reactive support on the client side.

Testing Improvements

- Complete support for **JUnit 5's** *Jupiter* programming and extension models in the Spring **TestContext**Framework.
  - **SpringExtension**: an implementation of multiple extension APIs from JUnit Jupiter that provides full support for the existing feature set of the Spring **TestContext** Framework. This support is enabled **via @ExtendWith(SpringExtension.class).**
  - **@SpringJUnitConfig**: a composed annotation that combines **@ExtendWith(SpringExtension.class)** from JUnit Jupiter with **@ContextConfigurationfrom** the Spring **TestContext** Framework.
  - **@SpringJUnitWebConfig:** a composed annotation that combines **@ExtendWith(SpringExtension.class)** from JUnit Jupiter with **@ContextConfiguration** and**@WebAppConfiguration** from the Spring **TestContext** Framework.
- New *before* and *after* test execution callbacks in the Spring **TestContext** Framework with support for **TestNG**, **JUnit 5**, and **JUnit 4** via the **SpringRunner** (but not via JUnit 4 rules).
  - New **beforeTestExecution()** and **afterTestExecution()** callbacks in the **TestExecutionListener API**and **TestContextManager**.
- **XMLUnit** support upgraded to 2.2

Data taken from:
https://spring.io/blog/2016/07/28/spring-framework-5-0-m1-released

Spring Cache

In any application or system there may be some data such type which is used frequently in the processing the request for any client. If such data we will be fetch from the database then it should be impact the performance of the system. Spring Framework provides support for caching means we can cache the frequently used data in the application at startup time

Spring Cache

- The Spring Framework provides support for transparently adding caching to an application. At its core, the abstraction applies caching to ecutions based on the information available in the cache.
- This support is available from Spring Framework 3.1 and there is significant improvement provided in the Spring 4.1.
- This is an abstract framework where Spring only provides the layer where other third party caching implementations can be easily plugged for storing data means cache storage is not implemented by Spring, where as enabling and caching is supported by spring out of the box.
- Caching is supported for the methods and it works well if the methods return the same result for the given input for the multiple invocations.

Supported cache providers in Spring

As we know that the actual implementations of the cache is by third party library and Spring provides only the abstract layer for enabling that specific cache implementation to store the cache data. So there are following cache supported by the Spring.

- Generic
- JCache (JSR-107)
- EhCache 2.x
- Hazelcast
- Infinispan
- Couchbase
- Redis
- Caffeine
- Guava
- Simple

Generic:

Generic caching is used if the context defines at least one org.springframework.cache.Cache bean, a CacheManager wrapping them is configured.

JCache (JSR-107):

JCache is bootstrapped via the presence of a javax.cache.spi.CachingProvider on the classpath (i.e. a JSR-107 compliant caching library).

EhCache 2.x:

EhCache 2.x is used if a file named ehcache.xml can be found at the root of the classpath. If EhCache 2.x and such file is present it is used to bootstrap the cache manager. An alternate configuration file can be providing a well using:

spring.cache.ehcache.config=classpath:config/another-config.xml

Hazelcast:

Both com.hazelcast:hazelcast and com.hazelcast:hazelcast-spring should be added to the project to enable support for Hazelcast. Since there is a default hazelcast.xml configuration file at the root of the classpath, it is used to automatically configure the underlying HazelcastInstance.

Infinispan:

Add the org.infinispan:infinispan-spring4-embedded dependency to enable support for Infinispan. There is no default location that Infinispan uses to look for a config file so if you don’t specify anything it will bootstrap on a hardcoded default. You can set the spring.cache.infinispan.config property to use the provided infinispan.xml configuration instead.

Couchbase:

Add the java-client and couchbase-spring-cache dependencies and make sure that you have setup at least aspring.couchbase.bootstrap-hosts property.

Redis:

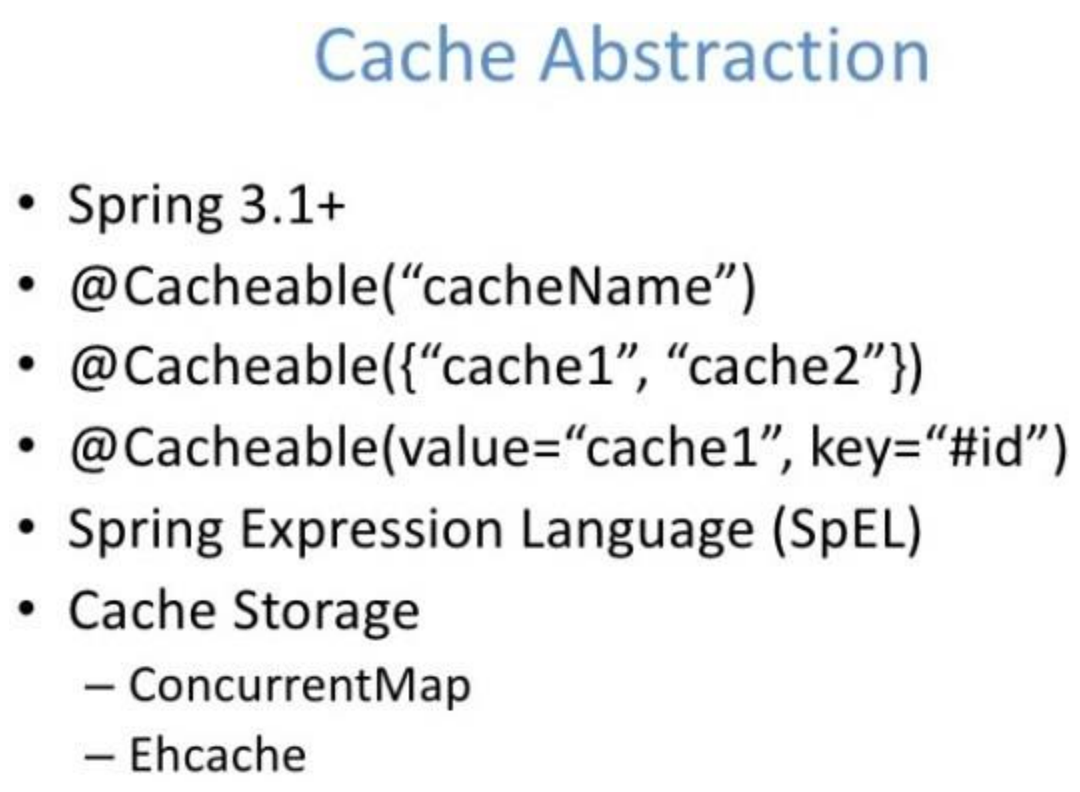
Add the spring-boot-starter-data-redis and make sure it is configured properly (by default, a redis instance with the default settings is expected on your local box).

Caffeine:

Simply add the com.github.ben-manes.caffeine:caffeine dependency to enable support for Caffeine. You can customize how caches are created in different ways, see application.properties for an example and the documentation for more details.

Guava:

Spring Boot does not provide any dependency management for Guava so you’ll have to add the com.google.guava:guavadependency with a version. You can customize how caches are created in different ways; see application.properties for an example and the documentation for more details.



Cache Abstraction

- Spring 3.1+
- @Cacheable("cacheName")
- @Cacheable({"cache1", "cache2"})
- @Cacheable(value="cache1", key="#id")
- Spring Expression Language (SpEL)
- Cache Storage
  – ConcurrentMap
  – Ehcache

Enable Caching

1. @EnableCaching to enable caching because caching in spring is not enabled by default. The caching feature can be declaratively enabled by simply adding the @EnableCaching annotation to any of the configuration classes.

```
@SpringBootApplication

@EnableCaching

public class SpringBootCacheApplication {



 public static void main(String[] args) {

   SpringApplication.run(SpringBootCacheApplication.class,
args);

 }

}
```

In the XML we can you use following tag for enable caching.

```
<cache:annotation-driven />
```

2. Cache Configurations – Configure the cache manager where the backing data is stored and retrieved for the quick response.

3. Caching declaration – Identify the methods that have to be cached and define the caching policy.

Use Caching With Annotations

After enabling the cache we can use following list of declarative annotations.

- @Cacheable
- @CacheEvict
- @CachePut
- @Caching
- @CacheConfig

@Cacheable: It is one of the most important and common annotation for caching the requests. If you annotate a method with @Cacheable, if multiple requests are received by the application, then this annotation will not execute the method multiple times, instead it will send the result from the cached storage.

The simplest way to enable caching behavior for a method is to demarcate it with @Cacheable and parameterize it with the name of the cache where the results would be stored:

```
@Cacheable(value="cities")

public List<City> findAllCity(){

 return (List<City>) cityRepository.findAll();

}
```

The findAllCity() call will first checks the cache cities before actually invoking the method and then caching the result.

Spring framework also supports multiple caches to be passed as parameters:

```
@Cacheable(value={"cities","city-list"})

public List<City> findAllCity(){

 return (List<City>) cityRepository.findAll();

}
```

@CacheEvict:

If we annotate all methods with **@Cacheable** then the size of cache may be some problem. **We don't want to populate the cache with values that we don't need often.** Caches can grow quite large, quite fast, and we could be holding on to a lot of stale or unused data. **@CacheEvict** annotation is used for removing a single cache or clearing the entire cache from the cache storage so that fresh values can be loaded into the cache again:

```
@CacheEvict(value="cities", allEntries=true)

public List<City> findAllCity(){

 return (List<City>) cityRepository.findAll();

}
```

Here allEntries indicated whether all the data in the cache has to be removed.

@CachePut:

@CachePut annotation helps for updating the cache with the latest execution without stopping the method execution. The difference between @Cacheable and @CachePut is that @Cacheable will skip running the method, whereas @CachePut will actually run the method and then put its results in the cache.

```
@CachePut(value="cities")

public List<City> findAllCity(){

 return (List<City>) cityRepository.findAll();
}
```

@Caching:

What if you want to use multiple annotations of the same type for caching a method? @Caching annotation used for grouping multiple annotations of the same type together when one annotation is not sufficient for the specifying the suitable condition. For example, you can put multiple @CacheEvict or @CachePut annotation inside @Caching to narrow down your conditions as you need.

```
@Caching(evict = {

 @CacheEvict("cities"),

 @CacheEvict(value="city-list", key="#city.name") })

public List<City> findAllCity(){

 return (List<City>) cityRepository.findAll();

}
```

@CacheConfig:

You can annotate @CacheConfig at the class level to avoid repeated mentioning in each method. For example, in the class level you can provide the cache name and in the method you just annotate with @Cacheable annotation.

```
@CacheConfig(cacheNames={"cities"})

public class CityMasterService {

 @Cacheable

    public List<City>) findAllCity() {

  return (List<City>)) cityRepository.findAll();

 }
```

```
}
```

Conditional Caching
Sometimes we don’t want to cache some result there are no need to cache. For example – reusing our example from the @CachePut annotation – this will both execute the method as well as cache the results each and every time:

```
@CachePut(value="cities")

public List<City>) findAllCity(State state){

 return (List<City>))
cityRepository.findAll(state.getStateCode());

}
```

Condition Parameter:

```
@CachePut(value="cities",
condition="#state.stateName=='UP'")

public ListList<City>) findAllCity(State state){

 return (List<City>))
cityRepository.findAll(state.getStateCode());

}
```

@CachePut can be parameterized with a condition parameter that takes a SpEL expression to ensure that the results are cached based on evaluating that expression.

Unless Parameter:

We can also control the caching based on the output of the method rather than the input – via the unless parameter

```
@CachePut(value="cities", unless="#result.length()>25")

public List<City>) findAllCity(String state){

 return (List<City>))
cityRepository.findAll(state.getStateCode());

}
```

The above annotation would cache only those cities of state that have minimum 25 cities in the state.

Spring Boot with NoSQL technologies

Hello friends!!! Here we are going to discuss Spring Boot with NoSQL, how Spring Boot provide the support for NoSQL technologies and how to use into Spring Boot Application. NoSQL is a non-relational database management systems, different from traditional relational database management systems in some significant ways.

Spring Boot with NoSQL

1. MongoDB

MongoDB is an open-source NoSQL document database that uses a JSON-like schema instead of traditional table-based relational data. Spring Boot offers several conveniences for working with MongoDB, including the spring-boot-starter-data-mongodb ‘Starter’. Spring Data includes repository support for MongoDB.

Spring Boot offers auto-configuration for Embedded Mongo. To use it in your Spring Boot application add a dependency on de.flapdoodle.embed:de.flapdoodle.embed.mongo. The port that Mongo will listen on can be configured using the spring.data.mongodb.port property. To use a randomly allocated free port use a value of zero. The MongoClient created by MongoAutoConfiguration will be automatically configured to use the randomly allocated port.

```
spring.data.mongodb.host=mongoserver
spring.data.mongodb.port=27017
```

1.1 Connecting to a MongoDB database

Spring Data Mongo provides a MongoTemplate class that is very similar in its design to Spring's JdbcTemplate. As with JdbcTemplate Spring Boot auto-configures a bean for you to simply inject:

```
@Component

public class DataBean {

    @Autowired

    MongoTemplate mongoTemplate;

// other tasks...


}
```

For MongoDB Full Example Click Here

2. Neo4j

Neo4j is an open-source NoSQL graph database that uses a rich data model of nodes related by first class relationships which is better suited for connected big data than traditional rdbms approaches. Spring Boot offers several conveniences for working with Neo4j, including the spring-boot-starter-data-neo4j 'Starter'.

You can configure the user and credentials to use via the spring.data.neo4j.* properties:

```
spring.data.neo4j.uri=http://localhost:7474

spring.data.neo4j.username=neo4j

spring.data.neo4j.password=secret
```

2.1 Connecting to a Neo4j database

```
@Component
public class DataBean{
    @Autowired
    Neo4jTemplate neo4jTemplate;

    //Other tasks
}
```

By default the instance will attempt to connect to a Neo4j server using **localhost:7474**

2.2 Using the embedded mode

If you add org.neo4j:neo4j-ogm-embedded-driver to the dependencies of your application, Spring Boot will automatically configure an in-process embedded instance of Neo4j that will not persist any data when your application shuts down. You can explicitly disable that mode using spring.data.neo4j.embedded.enabled=false.

```
spring.data.neo4j.uri=file://opt/db/graph.db
```

2.3 Spring Data Neo4j repositories

Spring Data includes repository support for Neo4j. You can customize entity scanning locations using the @NodeEntityScan annotation.

```
@EnableNeo4jRepositories(basePackages                        =
"com.dineshonjava.myapp.repository")
@EnableTransactionManagement
package com.dineshonjava.myapp.domain;
```

```
import org.springframework.data.domain.*;

import org.springframework.data.repository.*;


public        interface        UserRepository        extends
GraphRepository<User> {


    Page<User> findAll(Pageable pageable);


    User findByNameAndEamil(String name, String email);


}
```

3. Redis

Redis is a cache, message broker and richly-featured key-value store. Spring Boot offers basic auto-configuration for the Jedis client library and abstractions on top of it provided by Spring Data Redis. There is a spring-boot-starter-data-redis 'Starter' for collecting the dependencies in a convenient way.

You can inject an auto-configured RedisConnectionFactory, StringRedisTemplate or vanilla RedisTemplate instance as you would any other Spring Bean. By default the instance will attempt to connect to a Redis server using **localhost:6379.**

```
@Component

public class DataBean {

    @Autowired

    StringRedisTemplate template;


    // other tasks...
}
```

4. Gemfire

Spring Data Gemfire provides convenient Spring-friendly tools for accessing the Pivotal Gemfire data management platform. There is a spring-boot-starter-data-gemfire ‘Starter’ for collecting the dependencies in a convenient way. There is currently no auto-configuration support for Gemfire, but you can enable Spring Data Repositories with a single annotation (@EnableGemfireRepositories).

5. Solr

Apache Solr is a search engine. Spring Boot offers basic auto-configuration for the Solr 5 client library and abstractions on top of it provided by Spring Data Solr. There is a spring-boot-starter-data-solr ‘Starter’ for collecting the dependencies in a convenient way.

You can inject an auto-configured SolrClient instance as you would any other Spring bean. By default the instance will attempt to connect to a server using **localhost:8983/solr:**

```
@Component

public class DataBean {

    @Autowired

    SolrClient solr;


    // other tasks...


}
```

5.1 Spring Data Solr repositories

Spring Data includes repository support for Apache Solr.

6. Elasticsearch

Elasticsearch is an open source, distributed, real-time search and analytics engine. Spring Boot offers basic auto-configuration for the Elasticsearch and abstractions on top of it provided by Spring Data Elasticsearch. There is a spring-boot-starter-data-elasticsearch 'Starter' for collecting the dependencies in a convenient way. Spring Boot also supports Jest.

6.1 Connecting to Elasticsearch using Spring Data

You can inject an auto-configured ElasticsearchTemplate or Elasticsearch Client instance as you would any other Spring Bean. By default the instance will embed a local in-memory server (a Node in Elasticsearch terms) and use the current working directory as the home directory for the server.
you can switch to a remote server by setting spring.data.elasticsearch.cluster-nodes to a comma-separated '**host:port**' list.

```
spring.data.elasticsearch.cluster-nodes=localhost:9300


@Component

public class DataBean {

    @Autowired

    ElasticsearchTemplate template;


   //other tasks ...


}
```

Spring Data includes repository support for Elasticsearch.

6.2 Connecting to Elasticsearch using Jest

```
spring.elasticsearch.jest.uris=http://search.doj.com:9200

spring.elasticsearch.jest.read-timeout=10000

spring.elasticsearch.jest.username=user

spring.elasticsearch.jest.password=secret
```

If you have Jest on the classpath, you can inject an auto-configured JestClient targeting localhost:9200 by default.

7. Cassandra

Cassandra is an open source, distributed database management system designed to handle large amounts of data across many commodity servers. Spring Boot offers auto-configuration for Cassandra and abstractions on top of it provided by Spring Data Cassandra. There is a spring-boot-starter-data-cassandra 'Starter' for collecting the dependencies in a convenient way.

7.1 Connecting to Cassandra

You can inject an auto-configured CassandraTemplate or a Cassandra Session instance as you would with any other Spring Bean. The spring.data.cassandra.* properties can be used to customize the connection.

```
spring.data.cassandra.keyspace-name=mykeyspace

spring.data.cassandra.contact-
points=cassandrahost1,cassandrahost2



@Component

public class DataBean {

    @Autowired

    CassandraTemplate template;
    //other tasks ...

}
```

8. Couchbase

Couchbase is an open-source, distributed multi-model NoSQL document-oriented database that is optimized for interactive applications. Spring Boot offers auto-configuration for Couchbase and abstractions on top of it provided by Spring Data Couchbase. There is a spring-boot-starter-data-couchbase 'Starter' for collecting the dependencies in a convenient way.

The spring.couchbase.* properties can be used to customize the connection. Generally you will provide the bootstrap hosts, bucket name and password:

```
spring.couchbase.bootstrap-hosts=mydojhost-1,192.168.11.111

spring.couchbase.bucket.name=my-bucket

spring.couchbase.bucket.password=secret
```

Spring Data includes repository support for Couchbase. ou can inject an auto-configured CouchbaseTemplate instance.

```
@Component

public class DataBean {

    @Autowired

    CouchbaseTemplate template;

  // other tasks...
}
```

Summary

Here we have discussed some detail about NoSQL database with Spring Boot. Spring Boot Provide major support and auto configuration for connect to NoSQl databases.

Happy Spring Boot Learning!!!

Spring Security using Spring Boot Example

Hello Friends!!! In this tutorial we will discuss the Spring Security with Spring Boot and also will see an example based on Spring security with Spring Boot.

1. Spring security Overview

Spring security is the highly customizable authentication and access-control framework. This is the security module for securing spring applications. But, this can also be used for non-spring based application with few extra configurations to enable the security features. Spring Security is a framework that focuses on providing both authentication and authorization to Java applications.

- “**Authentication**” is the process of establishing a principal is who they claim to be (a “principal” generally means a user, device or some other system which can perform an action in your application).
- “**Authorization**” refers to the process of deciding whether a principal is allowed to perform an action within your application. To arrive at the point where an authorization decision is needed, the identity of the principal has already been established by the authentication process. These concepts are common, and not at all specific to Spring Security.

2. Spring Security Modules

In Spring Security 3.0, the codebase has been sub-divided into separate jars which more clearly separate different functionality areas and third-party dependencies. The following are the list of modules currently shipped by spring security framework.

- **Core (spring-security-core.jar)** – This module contains the APIs for basic authentication and access-control related mechanism. This is mandatory for ant spring security applications.

- **Remoting (spring-security-remoting.jar)** – This module provides integration to the Spring Remoting. You don't need to include this module unless you are writing remote client applications.
- **Web (spring-security-web.jar)** – This module contains APIs for servlet filters and any web based authentication like access restriction for URLs. Any web application would require this module.
- **Config (spring-security-config.jar)** – Contains the security namespace parsing code & Java configuration code. You need it if you are using the Spring Security XML namespace for configuration. If you are not using XML configurations, you can ignore this module.
- **LDAP (spring-security-ldap.jar)**– Required if you need to use LDAP authentication or manage LDAP user entries.
- **ACL (spring-security-acl.jar)** – Specialized domain object ACL implementation. Used to apply security to specific domain object instances within your application.
- **CAS (spring-security-cas.jar)** – Spring Security's CAS client integration. If you want to use Spring Security web authentication with a CAS single sign-on server.
- **OpenID (pring-security-openid.jar)** – OpenID web authentication support. Used to authenticate users against an external OpenID server.
- **Test (spring-security-test.jar)**– Support for testing with Spring Security.

3. Getting Spring Security

You can get hold of Spring Security in several ways.

3.1. Usage with Maven

A minimal Spring Security Maven set of dependencies typically looks like the following:

Pom.xml

```
<dependencies>

<!-- ... other dependency elements ... -->

<dependency>

 <groupId>org.springframework.security</groupId>

 <artifactId>spring-security-web</artifactId>

 <version>4.1.2.RELEASE</version>

</dependency>

<dependency>
```

```
 <groupId>org.springframework.security</groupId>

 <artifactId>spring-security-config</artifactId>

 <version>4.1.2.RELEASE</version>

</dependency>

</dependencies>
```

3.2. Usage with Gradle

A minimal Spring Security Gradle set of dependencies typically looks like the following:

build.gradle

```
dependencies {

 compile 'org.springframework.security:spring-security-web:4.1.2.RELEASE'

 compile 'org.springframework.security:spring-security-config:4.1.2.RELEASE'

}
```

4. Core Components

Core Components represent the building blocks of spring security and what are the core components that are actually used while user is authenticating to your application. So if you ever need to go beyond a simple namespace configuration then it’s important that you understand what they are, even if you don’t actually need to interact with them directly.

4.1 SecurityContext

As the name implies, this interface is the corner stone of storing all the security related details for your application. When you enable spring security for your application, a SecurityContext will enable for each application and stores the details of authenticated user, etc. It uses Authentication object for storing the details related to authentications.

4.2 SecurityContextHolder

The most fundamental object is SecurityContextHolder. This class is important for accessing any value from the SecurityContext. This is where we store details of the present security context of the application, which includes details of the principal currently using the application. By default the SecurityContextHolder uses a ThreadLocal to store these details, which means that the security context is always available to methods in the same thread of execution, even if the security context is not explicitly passed around as an argument to those methods.

```
Object principal =
SecurityContextHolder.getContext().getAuthentication().getPrincipal();



if (principal instanceof UserDetails) {

String username = ((UserDetails)principal).getUsername();

} else {

String username = principal.toString();

}
```

4.3 Authentication

Let’s consider a standard authentication scenario that everyone is familiar with.

1. A user is prompted to log in with a **username** and **password**.
2. The system verifies that the password is correct for the username.
3. The context information for that user is obtained their list of roles and so on.
4. A security context is established for the user
5. The user proceeds, potentially to perform some operation which is potentially protected by an access control mechanism which checks the required permissions for the operation against the current security context information.

The following are the steps to achieve the authentication:

1. **Authentication** is an interface which has several implementations for different authentication models. For a simple user name and password authentication, spring security would use **UsernamePasswordAuthenticationToken**. When user enters username and password, system creates a new instance of **UsernamePasswordAuthenticationToken**.

2. The token is passed to an instance of **AuthenticationManager** for validation. Internally what **AuthenticationManager** will do is to iterate the list of configured **AuthenticationProvider** to validate the request. There should be at least one provider to be configured for the valid authentication.
3. The **AuthenticationManager** returns a fully populated Authentication instance on successful authentication.
4. The final step is to establish a security context by invoking **SecurityContextHolder.getContext().setAuthentication(),** passing in the returned authentication object.

4.4 UserDetailsService

UserDetails is a core interface in Spring Security. It represents a principal, but in an extensible and application-specific way. Think of UserDetails as the adapter between your own user database and what Spring Security needs inside the SecurityContextHolder. UserDetailsService is a core interface in spring security to load user specific data. This interface is considered as user DAO and will be implemented by specific DAO implementations. For example, for a basic in memory authentication, there is an InMemoryUserDetailsManager. This interface declares only one method loadUserByUsername (String username) which simplifies the implementation classes to write other specific methods.

Suppose you want to use your existing DAO classes to load the user details from the database, just implement the UserDetailsService and override the method loadUserByUsername(String username). An example of this implementation would look like this:

```
@Service

public class CurrentUserDetailsService implements UserDetailsService {

   private final UserService userService;

   @Autowired

   public CurrentUserDetailsService(UserService userService) {

      this.userService = userService;

   }
   public CurrentUser loadUserByUsername(String username) throws
UsernameNotFoundException {

       User user = userService.getUserByUsername(username);

       return new CurrentUser(user);

   }

}
```

In the above code, the model CurrentUser must of of type org.springframework.security.userdetails.UserDetails. In the below example application, I have extended the org.springframework.security.userdetails.UserDetails class and written the custom user class. This will have the advantage of not exposing our domain class.

4.5 GrantedAuthority

Apart from authenticating to the application, another important component is to get the list of granted authorities for the logged in user. This comes as part of the authorization process. This is retrieved by calling the getAuthorities() in Authentication object. This returns the list of GrantedAuthority which denotes roles for the users. Such authorities are usually "roles", such as ROLE_ADMINISTRATOR orROLE_HR_SUPERVISOR. These roles are later on configured for web authorization, method authorization and domain object authorization.

Summary

The major building blocks of Spring Security that we've seen so far are:

- **SecurityContextHolder**, to provide access to the **SecurityContext**.
- **SecurityContext**, to hold the **Authentication** and possibly request-specific security information.
- **Authentication**, to represent the **principal** in a Spring Security-specific manner.
- **GrantedAuthority**, to reflect the application-wide permissions granted to a principal.
- **UserDetails**, to provide the necessary information to build an Authentication object from your application's DAOs or other source of security data.
- **UserDetailsService**, to create a UserDetails when passed in a String-based username (or certificate ID or the like).

5. Spring Security Example Application

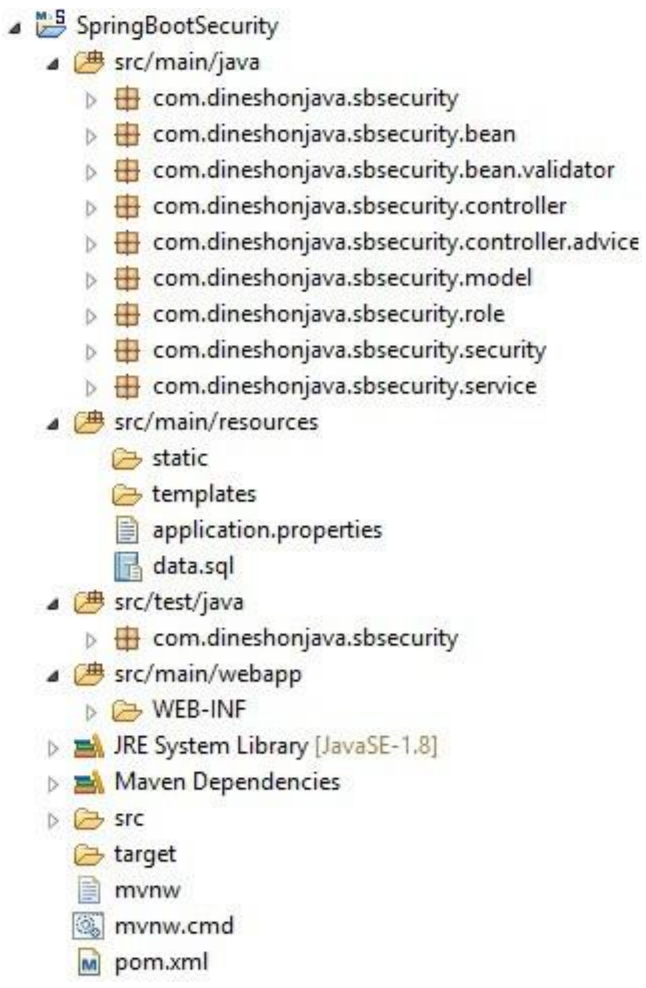
This section walks you through the process of creating a simple web application in Spring Boot with resources that are protected by Spring Security.

What you'll build

You'll build a Spring MVC application that secures the page with a login form backed by a fixed list of users. Let's start with the use cases for every common web application with some basic access restrictions. So, the requirements of such an app could be:

- The app will have users, each with role **Admin** or **User**
- They log in by their **emails** and **passwords**
- Non-admin users can view their info, but cannot peek at other users
- Admin users can list and view all the users, and create new ones as well
- Customized form for login
- "**Remember me**" authentication for laziest
- Possibility to logout
- Home page will be available for everyone, authenticated or not

Let's create Web Project with Spring Boot Web Initialzr with **Spring Web**, **Spring Data JPA**, **Spring Security** and **H2 Database**. So following project structure we have after creating from web interface and adding necessary files for this example.



```
SpringBootSecurity
  src/main/java
    com.dineshonjava.sbsecurity
    com.dineshonjava.sbsecurity.bean
    com.dineshonjava.sbsecurity.bean.validator
    com.dineshonjava.sbsecurity.controller
    com.dineshonjava.sbsecurity.controller.advice
    com.dineshonjava.sbsecurity.model
    com.dineshonjava.sbsecurity.role
    com.dineshonjava.sbsecurity.security
    com.dineshonjava.sbsecurity.service
  src/main/resources
    static
    templates
    application.properties
    data.sql
  src/test/java
    com.dineshonjava.sbsecurity
  src/main/webapp
    WEB-INF
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

Let’s discuss files of this example.

Dependencies pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>



 <groupId>com.dineshonjava.sbsecurity</groupId>
 <artifactId>SpringBootSecurity</artifactId>

 <version>0.0.1-SNAPSHOT</version>

 <packaging>jar</packaging>


 <name>SpringBootSecurity</name>

 <description>SpringBootSecurity project for Spring Boot and Spring
Security</description>


 <parent>

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

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

  <version>1.4.0.RELEASE</version>

  <relativePath/> <!-- lookup parent from repository -->

 </parent>


 <properties>

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

  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

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

 </properties>


 <dependencies>

  <dependency>

   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-freemarker</artifactId>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
  </dependency>
  <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
```

```
    <scope>test</scope>

   </dependency>

  </dependencies>



  <build>

   <plugins>

    <plugin>

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

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

    </plugin>

   </plugins>

  </build>




</project>
```

WebSecurityConfigurerAdapter
This is the Java configuration class for writing the web based security configurations. You can override the methods in this class to configure the following things:

- Enforce the user to be authenticated prior to accessing any URL in your application
- Create a user with the username user , password, and role of ROLE_USER
- Enables HTTP Basic and Form based authentication
- Spring Security will automatically render a login page and logout success page for you

SecurityConfig.java

```
/**

 *

 */
```

```
package com.dineshonjava.sbsecurity.security;


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

import org.springframework.boot.autoconfigure.security.SecurityProperties;

import org.springframework.context.annotation.Configuration;

import org.springframework.core.annotation.Order;

import
org.springframework.security.config.annotation.authentication.builders.Authen
ticationManagerBuilder;

import
org.springframework.security.config.annotation.method.configuration.EnableGlo
balMethodSecurity;

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

import
org.springframework.security.config.annotation.web.configuration.WebSecurityC
onfigurerAdapter;

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

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


/**

 * @author Dinesh.Rajput

 *

 */

@Configuration

@EnableGlobalMethodSecurity(prePostEnabled = true)
```

```
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)

public class SecurityConfig extends WebSecurityConfigurerAdapter{



 @Autowired

    UserDetailsService userDetailsService;



    @Override

    protected void configure(HttpSecurity http) throws Exception {

        http

        .authorizeRequests()

            .antMatchers("/", "/home", "/public/**").permitAll()

            .antMatchers("/users/**").hasAuthority("ADMIN")

            .anyRequest().fullyAuthenticated()

            .and()

        .formLogin()

            .loginPage("/login")

            .failureUrl("/login?error")

            .usernameParameter("email")

            .permitAll()

            .and()

        .logout()

         .logoutUrl("/logout")

            .logoutSuccessUrl("/")
            .permitAll();

    }



    @Override

    public void configure(AuthenticationManagerBuilder auth) throws Exception
{

        auth

                .userDetailsService(userDetailsService)

                .passwordEncoder(new BCryptPasswordEncoder());

    }

}
```

Domain Classes
User.java

```
/**

 *

 */

package com.dineshonjava.sbsecurity.model;



import java.io.Serializable;



import javax.persistence.Column;

import javax.persistence.Entity;
import javax.persistence.EnumType;

import javax.persistence.Enumerated;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;



import com.dineshonjava.sbsecurity.role.Role;



/**

 * @author Dinesh.Rajput

 *

 */

@Entity

@Table(name = "user")

public class User implements Serializable{



 /**

  *

  */

 private static final long serialVersionUID = 1L;

 @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
   @Column(name = "userid", nullable = false, updatable = false)

Long userid;

@Column(name = "username", nullable = false)

String username;

@Column(name = "email", nullable = false, unique = true)

String email;

@Column(name = "password", nullable = false)

String password;

@Column(name = "role", nullable = false)

@Enumerated(EnumType.STRING)

Role role;

public Long getUserid() {

 return userid;

}

public void setUserid(Long userid) {

 this.userid = userid;

}

public String getUsername() {

 return username;

}

public void setUsername(String username) {

 this.username = username;

}
```

```
public String getEmail() {

 return email;

}

public void setEmail(String email) {

 this.email = email;

}

public String getPassword() {

 return password;

}

public void setPassword(String password) {

 this.password = password;

}

public Role getRole() {

 return role;

}

public void setRole(Role role) {

 this.role = role;

}

@Override

public String toString() {

 return "User [userid=" + userid + ", username=" + username + ", email="

   + email + ", password=" + password + ", role=" + role + "]";

}
```

```
@Override

public int hashCode() {

 final int prime = 31;

 int result = 1;

 result = prime * result + ((email == null) ? 0 : email.hashCode());

 result = prime * result

   + ((password == null) ? 0 : password.hashCode());

 result = prime * result + ((role == null) ? 0 : role.hashCode());

 result = prime * result + ((userid == null) ? 0 : userid.hashCode());

 result = prime * result

   + ((username == null) ? 0 : username.hashCode());

 return result;

}

@Override

public boolean equals(Object obj) {

 if (this == obj)

  return true;

 if (obj == null)

  return false;

 if (getClass() != obj.getClass())

  return false;

 User other = (User) obj;

 if (email == null) {
  if (other.email != null)
   return false;
 } else if (!email.equals(other.email))
  return false;
 if (password == null) {
  if (other.password != null)
   return false;
 } else if (!password.equals(other.password))
  return false;
 if (role != other.role)
  return false;
 if (userid == null) {
  if (other.userid != null)
   return false;
 } else if (!userid.equals(other.userid))
  return false;
 if (username == null) {
  if (other.username != null)
   return false;
 } else if (!username.equals(other.username))
  return false;
 return true;
}
```

```
}
```

Role.java

```
/**
 *
 */
package com.dineshonjava.sbsecurity.role;


/**
 * @author Dinesh.Rajput
 *
 */
public enum Role {
 USER, ADMIN
}
```

Besides that, a form for creating a new user will be nice to have:
UserBean.java

```
/**
 *
 */
package com.dineshonjava.sbsecurity.bean;
```

```
import javax.validation.constraints.NotNull;

import org.hibernate.validator.constraints.NotEmpty;

import com.dineshonjava.sbsecurity.role.Role;

/**
 * @author Dinesh.Rajput
 *
 */
public class UserBean {
 @NotEmpty
    private String username = "";

 @NotEmpty
    private String email = "";

    @NotEmpty
    private String password = "";

    @NotEmpty
    private String passwordRepeated = "";
```

```
   @NotNull
   private Role role = Role.USER;

public String getEmail() {
 return email;
}

public void setEmail(String email) {
 this.email = email;
}

public String getPassword() {
 return password;
}

public void setPassword(String password) {
 this.password = password;
}

public String getPasswordRepeated() {
 return passwordRepeated;
}
```

```
 public void setPasswordRepeated(String passwordRepeated) {

  this.passwordRepeated = passwordRepeated;

 }


 public Role getRole() {

  return role;

 }


 public void setRole(Role role) {

  this.role = role;

 }


 public String getUsername() {

  return username;

 }


 public void setUsername(String username) {

  this.username = username;

 }


}
```

This will function as a data transfer object (DTO) between the web layer and service layer. It’s annotated by Hibernate Validator validation constraints and sets some sane defaults.

CurrentUser.java

```
/**

 *

 */

package com.dineshonjava.sbsecurity.bean;



import org.springframework.security.core.authority.AuthorityUtils;



import com.dineshonjava.sbsecurity.model.User;

import com.dineshonjava.sbsecurity.role.Role;



/**

 * @author Dinesh.Rajput

 *

 */

public class CurrentUser extends
org.springframework.security.core.userdetails.User {



 /**

  *

  */

 private static final long serialVersionUID = 1L;
 private User user;


 public CurrentUser(User user) {

        super(user.getEmail(), user.getPassword(),
AuthorityUtils.createAuthorityList(user.getRole().toString()));

        this.user = user;

    }


    public User getUser() {

        return user;

    }


    public Long getId() {

        return user.getUserid();

    }


    public Role getRole() {

        return user.getRole();

    }


    @Override

    public String toString() {

        return "CurrentUser{" +
                "user=" + user +

                "} " + super.toString();

}

}
```

Service Layer
In service layer, where the business logic should, we'd need something to retrieve the User by his id, email, list all the users and create a new one.
UserService .java

```
/**

 *

 */

package com.dineshonjava.sbsecurity.service;



import java.util.Collection;



import com.dineshonjava.sbsecurity.bean.UserBean;

import com.dineshonjava.sbsecurity.model.User;



/**

 * @author Dinesh.Rajput

 *

 */

public interface UserService {
    User getUserById(long id);



    User getUserByEmail(String email);



    Collection<User> getAllUsers();



    User create(UserBean userBean);

}
```

UserServiceImpl .java
The implementation of the service:

```
/**

 *

 */

package com.dineshonjava.sbsecurity.service;



import java.util.Collection;



import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

import org.springframework.stereotype.Service;
import com.dineshonjava.sbsecurity.bean.UserBean;

import com.dineshonjava.sbsecurity.model.User;

import com.dineshonjava.sbsecurity.model.UserRepository;



/**

 * @author Dinesh.Rajput

 *

 */

@Service

public class UserServiceImpl implements UserService {



 private static final Logger LOGGER =
LoggerFactory.getLogger(UserServiceImpl.class);



 @Autowired

 UserRepository userRepository;



 @Override

 public User getUserById(long id) {

  LOGGER.debug("Getting user={}", id);

  return userRepository.findOne(id);

 }
 @Override

 public User getUserByEmail(String email) {

  LOGGER.debug("Getting user by email={}", email.replaceFirst("@.*",
"@***"));

  return userRepository.findOneByEmail(email);

 }



 @Override

 public Collection<User> getAllUsers() {

  LOGGER.debug("Getting all users");

  return (Collection<User>) userRepository.findAll();

 }



 @Override

 public User create(UserBean userBean) {

  User user = new User();

  user.setUsername(userBean.getUsername());

        user.setEmail(userBean.getEmail());

        user.setPassword(new
BCryptPasswordEncoder().encode(userBean.getPassword()));

        user.setRole(userBean.getRole());

        return userRepository.save(user);

 }
```

```
}
```

Spring Data Repository
This section explains the service classes and spring data repository implementation.
UserRepository .java

```
/**

 *

 */

package com.dineshonjava.sbsecurity.model;



import org.springframework.data.repository.CrudRepository;



/**

 * @author Dinesh.Rajput

 *

 */

public interface UserRepository extends CrudRepository<User, Long>{



 User findOneByEmail(String email);

}
```

CurrentUserDetailsService.java

```
/**
```

```
 *

 */

package com.dineshonjava.sbsecurity.service;



import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

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

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

import org.springframework.stereotype.Service;



import com.dineshonjava.sbsecurity.bean.CurrentUser;

import com.dineshonjava.sbsecurity.model.User;



/**

 * @author Dinesh.Rajput

 *

 */

@Service

public class CurrentUserDetailsService implements UserDetailsService {
 private static final Logger LOGGER =
LoggerFactory.getLogger(CurrentUserDetailsService.class);



 @Autowired

 UserService userService;



 @Override

 public UserDetails loadUserByUsername(String email)

   throws UsernameNotFoundException {

  LOGGER.debug("Authenticating user with email={}", email.replaceFirst("@.*",
"@***"));

  User user = userService.getUserByEmail(email);

  return new CurrentUser(user);

 }



}
```

Spring MVC Configurations Web Layer
Home Page

```
/**

 *

 */

package com.dineshonjava.sbsecurity.controller;
```

```
import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Controller;

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



/**

 * @author Dinesh.Rajput

 *

 */

@Controller

public class HomeController {



 private static final Logger LOGGER =
LoggerFactory.getLogger(HomeController.class);



 @RequestMapping("/")

    public String getHomePage() {

  LOGGER.debug("Getting home page");

        return "home";

    }

}
```

Login Page

```
/**
```

```
 *

 */

package com.dineshonjava.sbsecurity.controller;



import java.util.Optional;



import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Controller;

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

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.servlet.ModelAndView;



/**

 * @author Dinesh.Rajput

 *

 */

@Controller

public class LoginController {



 private static final Logger LOGGER =
LoggerFactory.getLogger(LoginController.class);
```

```
 @RequestMapping(value = "/login", method = RequestMethod.GET)

    public ModelAndView getLoginPage(@RequestParam Optional<String> error) {

     LOGGER.debug("Getting login page, error={}", error);

        return new ModelAndView("login", "error", error);

    }

}
```

User Page

```
/**

 *

 */

package com.dineshonjava.sbsecurity.controller;



import javax.validation.Valid;



import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import org.springframework.dao.DataIntegrityViolationException;

import org.springframework.security.access.prepost.PreAuthorize;

import org.springframework.stereotype.Controller;

import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;

import org.springframework.web.bind.annotation.InitBinder;

import org.springframework.web.bind.annotation.ModelAttribute;

import org.springframework.web.bind.annotation.PathVariable;

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

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.servlet.ModelAndView;



import com.dineshonjava.sbsecurity.bean.UserBean;

import com.dineshonjava.sbsecurity.bean.validator.UserBeanValidator;

import com.dineshonjava.sbsecurity.service.UserService;



/**

 * @author Dinesh.Rajput

 *

 */

@Controller

public class UserController {



 private static final Logger LOGGER =
LoggerFactory.getLogger(UserController.class);



 @Autowired

 UserService userService;
```

```
@Autowired

UserBeanValidator userBeanValidator;



@InitBinder("form")

   public void initBinder(WebDataBinder binder) {

       binder.addValidators(userBeanValidator);

   }



@RequestMapping("/users")

public ModelAndView getUsersPage() {

 LOGGER.debug("Getting users page");

 return new ModelAndView("users", "users", userService.getAllUsers());

}



@PreAuthorize("@currentUserServiceImpl.canAccessUser(principal, #id)")

@RequestMapping("/user/{id}")

   public ModelAndView getUserPage(@PathVariable Long id) {

 LOGGER.debug("Getting user page for user={}", id);

       return new ModelAndView("user", "user", userService.getUserById(id));

   }



@PreAuthorize("hasAuthority('ADMIN')")
```

```
    @RequestMapping(value = "/user/create", method = RequestMethod.GET)

    public ModelAndView getUserCreatePage() {

      LOGGER.debug("Getting user create form");

        return new ModelAndView("user_create", "form", new UserBean());

    }



 @PreAuthorize("hasAuthority('ADMIN')")

    @RequestMapping(value = "/user/create", method = RequestMethod.POST)

    public String handleUserCreateForm(@Valid @ModelAttribute("form")
UserBean form, BindingResult bindingResult) {

      LOGGER.debug("Processing user create form={}, bindingResult={}", form,
bindingResult);

        if (bindingResult.hasErrors()) {

            return "user_create";

        }

        try {

            userService.create(form);

        } catch (DataIntegrityViolationException e) {

         LOGGER.warn("Exception occurred when trying to save the user,
assuming duplicate email", e);

            bindingResult.reject("email.exists", "Email already exists");

            return "user_create";

        }

        return "redirect:/users";

    }
```

```
}
```

CurrentUserServiceImpl.java

```
/**
 *
 */
package com.dineshonjava.sbsecurity.service;


import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Service;


import com.dineshonjava.sbsecurity.bean.CurrentUser;

import com.dineshonjava.sbsecurity.role.Role;


/**
 * @author Dinesh.Rajput
 *
 */
@Service

public class CurrentUserServiceImpl implements CurrentUserService {
 private static final Logger LOGGER =
LoggerFactory.getLogger(CurrentUserDetailsService.class);



    @Override

    public boolean canAccessUser(CurrentUser currentUser, Long userId) {

        LOGGER.debug("Checking if user={} has access to user={}",
currentUser, userId);

        return currentUser != null

                && (currentUser.getRole() == Role.ADMIN ||
currentUser.getId().equals(userId));

}



}
```

application.properties

```
#security.user.name=root

#security.user.password=111

#security.user.role=ADMIN

logging.level.org.springframework=WARN

logging.level.org.hibernate=WARN

logging.level.com.dineshonjava=DEBUG

spring.freemarker.template-loader-path=/WEB-INF/ftl

spring.freemarker.expose-request-attributes=true
spring.freemarker.expose-spring-macro-helpers=true
```

Here we are using freemarker template for views layers.

distributionUrl=https://repo1.maven.org/maven2/org/apache/maven/apache-maven/3.3.9/apache-maven-3.3.9-bin.

```
/**
 *
 */
package com.dineshonjava.sbsecurity.bean.validator;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

import com.dineshonjava.sbsecurity.bean.UserBean;
import com.dineshonjava.sbsecurity.service.UserService;

/**
 * @author Dinesh.Rajput
 *
 */
@Component
public class UserBeanValidator implements Validator{
	private static final Logger LOGGER =
LoggerFactory.getLogger(UserBeanValidator.class);

	@Autowired
	UserService userService;

	@Override
	public boolean supports(Class<?> clazz) {
		return clazz.equals(UserBean.class);
	}

	@Override
	public void validate(Object target, Errors errors) {
		LOGGER.debug("Validating {}", target);
		UserBean bean = (UserBean) target;
        validatePasswords(errors, bean);
        validateEmail(errors, bean);
	}
	private void validatePasswords(Errors errors, UserBean bean) {
        if (!bean.getPassword().equals(bean.getPasswordRepeated())) {
            errors.reject("password.no_match", "Passwords do not match");
        }
    }

    private void validateEmail(Errors errors, UserBean bean) {
        if (userService.getUserByEmail(bean.getEmail()) != null) {
            errors.reject("email.exists", "User with this email already exists");
        }
    }
}

/**
 * 
 */
package com.dineshonjava.sbsecurity.bean;

import org.springframework.security.core.authority.AuthorityUtils;

import com.dineshonjava.sbsecurity.model.User;
import com.dineshonjava.sbsecurity.role.Role;

/**
 * @author Dinesh.Rajput
 *
 */
public class CurrentUser extends org.springframework.security.core.userdetails.User {

	/**
	 * 
	 */
	private static final long serialVersionUID = 1L;

	private User user;
```

```
        public CurrentUser(User user) {
        super(user.getEmail(), user.getPassword(),
AuthorityUtils.createAuthorityList(user.getRole().toString()));
        this.user = user;
    }

    public User getUser() {
        return user;
    }

    public Long getId() {
        return user.getUserid();
    }

    public Role getRole() {
        return user.getRole();
    }

    @Override
    public String toString() {
        return "CurrentUser{" +
                "user=" + user +
                "} " + super.toString();
}
}
/**
```

```
 *
 */
package com.dineshonjava.sbsecurity.bean;

import javax.validation.constraints.NotNull;

import org.hibernate.validator.constraints.NotEmpty;

import com.dineshonjava.sbsecurity.role.Role;

/**
 * @author Dinesh.Rajput
 *
 */
public class UserBean {
	@NotEmpty
    private String username = "";

	@NotEmpty
    private String email = "";

    @NotEmpty
    private String password = "";

    @NotEmpty
    private String passwordRepeated = "";
```

```
@NotNull
private Role role = Role.USER;

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getPasswordRepeated() {
        return passwordRepeated;
    }

    public void setPasswordRepeated(String passwordRepeated) {
        this.passwordRepeated = passwordRepeated;
    }
```

```
	public Role getRole() {
		return role;
	}

	public void setRole(Role role) {
		this.role = role;
	}

	public String getUsername() {
		return username;
	}

	public void setUsername(String username) {
		this.username = username;
	}

}
/**
 *
 */
package com.dineshonjava.sbsecurity.controller.advice;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ModelAttribute;
import com.dineshonjava.sbsecurity.bean.CurrentUser;

/**
 * @author Dinesh.Rajput
 *
 */
@ControllerAdvice
public class CurrentUserControllerAdvice {

        private static final Logger LOGGER =
LoggerFactory.getLogger(CurrentUserControllerAdvice.class);

    @ModelAttribute("currentUser")
    public CurrentUser getCurrentUser(Authentication authentication) {
        LOGGER.debug("Getting getCurrentUser");
        return (authentication == null) ? null : (CurrentUser) authentication.getPrincipal();
}
}
/**
 *
 */
package com.dineshonjava.sbsecurity.controller.advice;

import java.util.NoSuchElementException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;

/**
 * @author Dinesh.Rajput
 *
 */
@ControllerAdvice
public class ExceptionHandlerControllerAdvice {

	private static Logger LOGGER =
LoggerFactory.getLogger(ExceptionHandlerControllerAdvice.class);

    @ExceptionHandler(NoSuchElementException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public String handleNoSuchElementException(NoSuchElementException e) {
	LOGGER.debug("Getting NoSuchElementException");
        return e.getMessage();
}
}

/**
 *
 */
```

```
package com.dineshonjava.sbsecurity.controller;


import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.stereotype.Controller;

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


/**

 * @author Dinesh.Rajput

 *

 */

@Controller

public class HomeController {


        private static final Logger LOGGER =
LoggerFactory.getLogger(HomeController.class);


        @RequestMapping("/")

   public String getHomePage() {

                LOGGER.debug("Getting home page");

      return "home";

   }

}


/**

 *

 */
```

```
package com.dineshonjava.sbsecurity.controller;

import java.util.Optional;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

/**
 * @author Dinesh.Rajput
 *
 */
@Controller
public class LoginController {

	private static final Logger LOGGER = 
LoggerFactory.getLogger(LoginController.class);

	@RequestMapping(value = "/login", method = RequestMethod.GET)
    public ModelAndView getLoginPage(@RequestParam Optional<String> error) {
    	LOGGER.debug("Getting login page, error={}", error);
        return new ModelAndView("login", "error", error);
    }
}
/**
 *
 */
package com.dineshonjava.sbsecurity.controller;

import javax.validation.Valid;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import com.dineshonjava.sbsecurity.bean.UserBean;
import com.dineshonjava.sbsecurity.bean.validator.UserBeanValidator;
import com.dineshonjava.sbsecurity.service.UserService;
```

```
/**
 * @author Dinesh.Rajput
 *
 */
@Controller
public class UserController {

        private static final Logger LOGGER =
LoggerFactory.getLogger(UserController.class);

        @Autowired
        UserService userService;

        @Autowired
        UserBeanValidator userBeanValidator;

        @InitBinder("form")
   public void initBinder(WebDataBinder binder) {
      binder.addValidators(userBeanValidator);
   }

        @RequestMapping("/users")
        public ModelAndView getUsersPage() {
                LOGGER.debug("Getting users page");
                return new ModelAndView("users", "users", userService.getAllUsers());
        }
        @PreAuthorize("@currentUserServiceImpl.canAccessUser(principal, #id)")
        @RequestMapping("/user/{id}")
    public ModelAndView getUserPage(@PathVariable Long id) {
                LOGGER.debug("Getting user page for user={}", id);
        return new ModelAndView("user", "user", userService.getUserById(id));
    }

        @PreAuthorize("hasAuthority('ADMIN')")
    @RequestMapping(value = "/user/create", method = RequestMethod.GET)
    public ModelAndView getUserCreatePage() {
          LOGGER.debug("Getting user create form");
        return new ModelAndView("user_create", "form", new UserBean());
    }

        @PreAuthorize("hasAuthority('ADMIN')")
    @RequestMapping(value = "/user/create", method = RequestMethod.POST)
    public String handleUserCreateForm(@Valid @ModelAttribute("form") UserBean
form, BindingResult bindingResult) {
          LOGGER.debug("Processing user create form={}, bindingResult={}", form,
bindingResult);
        if (bindingResult.hasErrors()) {
            return "user_create";
        }
        try {
            userService.create(form);
        } catch (DataIntegrityViolationException e) {
           LOGGER.warn("Exception occurred when trying to save the user, assuming
duplicate email", e);
```

```
            bindingResult.reject("email.exists", "Email already exists");
            return "user_create";
        }
        return "redirect:/users";
    }
}
/**
 *
 */
package com.dineshonjava.sbsecurity.controller;

import javax.validation.Valid;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.dao.DataIntegrityViolationException;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import com.dineshonjava.sbsecurity.bean.UserBean;
import com.dineshonjava.sbsecurity.bean.validator.UserBeanValidator;
import com.dineshonjava.sbsecurity.service.UserService;

/**
 * @author Dinesh.Rajput
 *
 */
@Controller
public class UserController {

	private static final Logger LOGGER = 
LoggerFactory.getLogger(UserController.class);

	@Autowired
	UserService userService;

	@Autowired
	UserBeanValidator userBeanValidator;

	@InitBinder("form")
    public void initBinder(WebDataBinder binder) {
        binder.addValidators(userBeanValidator);
    }

	@RequestMapping("/users")
```

```
        public ModelAndView getUsersPage() {
                LOGGER.debug("Getting users page");
                return new ModelAndView("users", "users", userService.getAllUsers());
        }

        @PreAuthorize("@currentUserServiceImpl.canAccessUser(principal, #id)")
        @RequestMapping("/user/{id}")
    public ModelAndView getUserPage(@PathVariable Long id) {
                LOGGER.debug("Getting user page for user={}", id);
        return new ModelAndView("user", "user", userService.getUserById(id));
    }

        @PreAuthorize("hasAuthority('ADMIN')")
    @RequestMapping(value = "/user/create", method = RequestMethod.GET)
    public ModelAndView getUserCreatePage() {
            LOGGER.debug("Getting user create form");
        return new ModelAndView("user_create", "form", new UserBean());
    }

        @PreAuthorize("hasAuthority('ADMIN')")
    @RequestMapping(value = "/user/create", method = RequestMethod.POST)
    public String handleUserCreateForm(@Valid @ModelAttribute("form") UserBean
form, BindingResult bindingResult) {
            LOGGER.debug("Processing user create form={}, bindingResult={}", form,
bindingResult);
        if (bindingResult.hasErrors()) {
            return "user_create";
        }
        try {
            userService.create(form);
        } catch (DataIntegrityViolationException e) {
           LOGGER.warn("Exception occurred when trying to save the user, assuming
duplicate email", e);
            bindingResult.reject("email.exists", "Email already exists");
            return "user_create";
        }
        return "redirect:/users";
    }
}
/**
 *
 */
package com.dineshonjava.sbsecurity.model;

import org.springframework.data.repository.CrudRepository;

/**
 * @author Dinesh.Rajput
 *
 */
public interface UserRepository extends CrudRepository<User, Long>{

           User findOneByEmail(String email);
}
```

```
/**
 *
 */
package com.dineshonjava.sbsecurity.role;

/**
 * @author Dinesh.Rajput
 *
 */
public enum Role {
	USER, ADMIN
}
/**
 *
 */
package com.dineshonjava.sbsecurity.security;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.autoconfigure.security.SecurityProperties;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.Order;
import
org.springframework.security.config.annotation.authentication.builders.AuthenticationM
anagerBuilder;
import
org.springframework.security.config.annotation.method.configuration.EnableGlobalMeth
odSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
```

```
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfigur
erAdapter;

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

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


/**
 * @author Dinesh.Rajput
 *
 */
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
@Order(SecurityProperties.ACCESS_OVERRIDE_ORDER)
public class SecurityConfig extends WebSecurityConfigurerAdapter{


        @Autowired
   UserDetailsService userDetailsService;


   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http
      .authorizeRequests()
         .antMatchers("/", "/home", "/public/**").permitAll()
         .antMatchers("/users/**").hasAuthority("ADMIN")
         .anyRequest().fullyAuthenticated()
         .and()
      .formLogin()
```

```
            .loginPage("/login")
            .failureUrl("/login?error")
            .usernameParameter("email")
            .permitAll()
            .and()
        .logout()
           .logoutUrl("/logout")
            .logoutSuccessUrl("/")
            .permitAll();
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth
                .userDetailsService(userDetailsService)
                .passwordEncoder(new BCryptPasswordEncoder());
    }
}
/**
 *
 */
package com.dineshonjava.sbsecurity.service;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.dineshonjava.sbsecurity.bean.CurrentUser;
import com.dineshonjava.sbsecurity.model.User;

/**
 * @author Dinesh.Rajput
 *
 */
@Service
public class CurrentUserDetailsService implements UserDetailsService {

	private static final Logger LOGGER = 
LoggerFactory.getLogger(CurrentUserDetailsService.class);

	@Autowired
	UserService userService;

	@Override
	public UserDetails loadUserByUsername(String email)
			throws UsernameNotFoundException {
		LOGGER.debug("Authenticating user with email={}", 
email.replaceFirst("@.*", "@***"));
		User user = userService.getUserByEmail(email);
		return new CurrentUser(user);
	}

}

/**
 * 
 */
package com.dineshonjava.sbsecurity.service;

import com.dineshonjava.sbsecurity.bean.CurrentUser;

/**
 * @author Dinesh.Rajput
 *
 */
public interface CurrentUserService {

	boolean canAccessUser(CurrentUser currentUser, Long userId);
}
/**
 * 
 */
package com.dineshonjava.sbsecurity.service;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import com.dineshonjava.sbsecurity.bean.CurrentUser;
import com.dineshonjava.sbsecurity.role.Role;

/**
 * @author Dinesh.Rajput
 *
 */
@Service
public class CurrentUserServiceImpl implements CurrentUserService {

        private static final Logger LOGGER =
LoggerFactory.getLogger(CurrentUserDetailsService.class);

    @Override
    public boolean canAccessUser(CurrentUser currentUser, Long userId) {
        LOGGER.debug("Checking if user={} has access to user={}", currentUser, userId);
        return currentUser != null
                && (currentUser.getRole() == Role.ADMIN ||
currentUser.getId().equals(userId));
}

}
/**
 *
 */
package com.dineshonjava.sbsecurity.service;
```

```
import java.util.Collection;

import com.dineshonjava.sbsecurity.bean.UserBean;
import com.dineshonjava.sbsecurity.model.User;

/**
 * @author Dinesh.Rajput
 *
 */
public interface UserService {

        User getUserById(long id);

    User getUserByEmail(String email);

    Collection<User> getAllUsers();

    User create(UserBean userBean);
}
/**
 *
 */
package com.dineshonjava.sbsecurity.service;

import java.util.Collection;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.stereotype.Service;

import com.dineshonjava.sbsecurity.bean.UserBean;
import com.dineshonjava.sbsecurity.model.User;
import com.dineshonjava.sbsecurity.model.UserRepository;

/**
 * @author Dinesh.Rajput
 *
 */
@Service
public class UserServiceImpl implements UserService {

	private static final Logger LOGGER = 
LoggerFactory.getLogger(UserServiceImpl.class);

	@Autowired
	UserRepository userRepository;

	@Override
	public User getUserById(long id) {
		LOGGER.debug("Getting user={}", id);
		return userRepository.findOne(id);
	}

	@Override
	public User getUserByEmail(String email) {
		LOGGER.debug("Getting user by email={}", email.replaceFirst("@.*",
"@***"));
		return userRepository.findOneByEmail(email);
	}

	@Override
	public Collection<User> getAllUsers() {
		LOGGER.debug("Getting all users");
		return (Collection<User>) userRepository.findAll();
	}

	@Override
	public User create(UserBean userBean) {
		User user = new User();
		user.setUsername(userBean.getUsername());
        user.setEmail(userBean.getEmail());
        user.setPassword(new BCryptPasswordEncoder().encode(userBean.getPassword()));
        user.setRole(userBean.getRole());
        return userRepository.save(user);
	}

}
package com.dineshonjava.sbsecurity;
```

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

@SpringBootApplication
public class SpringBootSecurityApplication {

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

```
#security.user.name=root
#security.user.password=111
#security.user.role=ADMIN
logging.level.org.springframework=WARN
logging.level.org.hibernate=WARN
logging.level.com.dineshonjava=DEBUG
spring.freemarker.template-loader-path=/WEB-INF/ftl
spring.freemarker.expose-request-attributes=true
spring.freemarker.expose-spring-macro-helpers=true
```

webapp. WEB-INF. Ftl

home.ftl

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="utf-8">
    <title>Home page</title>
</head>
<body>
<nav role="navigation">
    <ul>
    <#if !currentUser??>
        <li><a href="/login">Log in</a></li>
    </#if>
    <#if currentUser??>
        <li>
            <form action="/logout" method="post">
                <input type="hidden" name="${_csrf.parameterName}"
value="${_csrf.token}"/>
                <button type="submit">Log out</button>
            </form>
        </li>
        <li><a href="/user/${currentUser.id}">View myself</a></li>
    </#if>
    <#if currentUser?? && currentUser.role == "ADMIN">
        <li><a href="/user/create">Create a new user</a></li>
        <li><a href="/users">View all users</a></li>
    </#if>
    </ul>
</nav>
</body>
```

```
</html>
```

login.ftl

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Log in</title>
</head>
<body>
<nav role="navigation">
    <ul>
        <li><a href="/">Home</a></li>
    </ul>
</nav>

<h1>Log in</h1>

<p>You can use: demo@localhost / demo</p>

<form role="form" action="/login" method="post">
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>

    <div>
        <label for="email">Email address</label>
        <input type="email" name="email" id="email" required autofocus/>
    </div>
    <div>
```

```
        <label for="password">Password</label>
        <input type="password" name="password" id="password" required/>
    </div>
    <div>
        <label for="remember-me">Remember me</label>
        <input type="checkbox" name="remember-me" id="remember-me"/>
    </div>
    <button type="submit">Sign in</button>
</form>

<#if error.isPresent()>
<p>The email or password you have entered is invalid, try again.</p>
</#if>
</body>
</html>
```

User

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>User details</title>
</head>
<body>
<nav role="navigation">
    <ul>
        <li><a href="/">Home</a></li>
    </ul>
```

```
</nav>


<h1>User details</h1>


<p>E-mail: ${user.email}</p>


<p>Role: ${user.role}</p>
</body>
</html>
```

---

user_create

```
<#import "/spring.ftl" as spring>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Create a new user</title>
</head>
<body>
<nav role="navigation">
    <ul>
        <li><a href="/">Home</a></li>
    </ul>
</nav>


<h1>Create a new user</h1>


<form role="form" name="form" action="" method="post">
```

```
    <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
        <div>
        <label for="username">Name</label>
        <input type="username" name="username" id="username"
value="${form.username}" required autofocus/>
    </div>
    <div>
        <label for="email">Email address</label>
        <input type="email" name="email" id="email" value="${form.email}" required
autofocus/>
    </div>
    <div>
        <label for="password">Password</label>
        <input type="password" name="password" id="password" required/>
    </div>
    <div>
        <label for="passwordRepeated">Repeat</label>
        <input type="password" name="passwordRepeated" id="passwordRepeated"
required/>
    </div>
    <div>
        <label for="role">Role</label>
        <select name="role" id="role" required>
            <option <#if form.role == 'USER'>selected</#if>>USER</option>
            <option <#if form.role == 'ADMIN'>selected</#if>>ADMIN</option>
        </select>
    </div>
    <button type="submit">Save</button>
```

```
</form>

<@spring.bind "form" />
<#if spring.status.error>
<ul>
    <#list spring.status.errorMessages as error>
        <li>${error}</li>
    </#list>
</ul>
</#if>
</body>
</html>
```

Users

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>List of Users</title>
</head>
<body>
<nav role="navigation">
    <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/user/create">Create a new user</a></li>
    </ul>
</nav>
```

```
<h1>List of Users</h1>

<table>
    <thead>
    <tr>
        <th>E-mail</th>
        <th>Role</th>
    </tr>
    </thead>
    <tbody>
    <#list users as user>
    <tr>
        <td><a href="/user/${user.userid}">${user.email}</a></td>
        <td>${user.role}</td>
    </tr>
    </#list>
    </tbody>
</table>
</body>
</html>
```

```
package com.dineshonjava.sbsecurity;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
```

```
@RunWith(SpringRunner.class)

@SpringBootTest

public class SpringBootSecurityApplicationTests {


	@Test

	public void contextLoads() {

	}


}
```

Spring Boot Actuator A Complete Guide

Hello friends!!! In this article, we’re going to introduce **Spring Boot Actuator** for collecting metrics about your production grade applications and also talk about some simple ways to work with them in production. Spring Boot Actuator is a sub-project of Spring Boot. It adds several production grade services to your application with little effort on your part. Lets discuss here about spring boot actuator and different types of metrics that can be collected using this feature

Spring Boot Actuator

Related Tutorial read before it.Introduction to Spring Boot – A Spring Boot Complete Guide

Table of Contents

1. What Is Spring Boot Actuator?

Spring Boot Actuator is sub-project of Spring Boot and it is one of the nice feature that adds great value to the Spring Boot applications. It adds several production grade services to your application with little effort on your part. There are also has many features added to your application out-of-the-box for managing the service in a production (or other) environment. They're mainly used to expose different types of information about the running application – health, metrics, info, dump, env etc.

2. How To Enable Spring Boot Actuator?

The spring-boot-actuator module provides all of Spring Boot's production-ready features. The simplest way to enable the features is to add a dependency to the **spring-boot-starter-actuator** 'Starter'.

2.1 In Maven Project

```
<dependency>

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

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

  </dependency>
```

2.2 In Gradle Project

```
dependencies {

  compile("org.springframework.boot:spring-boot-starter-actuator")
}
```

3. Endpoints

Actuator endpoints allow you to monitor and interact with your application. Spring Boot includes a number of built-in endpoints and you can also add your own. Most endpoints are sensitive – meaning they're not fully public – while a handful are not: /health and /info.

Here's some of the most common endpoints Boot provides out of the box:

- **/actuator-** Provides a hypermedia-based "discovery page" for the other endpoints. Requires Spring HATEOAS to be on the classpath. Sensitive by Default.
- **/autoconfig-** Displays an auto-configuration report showing all auto-configuration candidates and the reason why they 'were' or 'were not' applied. Sensitive by Default.
- **/beans-** Displays a complete list of all the Spring beans in your application. Sensitive by Default.
- **/configprops-** This endpoint shows configuration properties used by your application. Sensitive by Default.
- **/dump-** Performs a thread dump. Sensitive by Default.
- **/env-** Exposes spring's properties from the configurations. Sensitive by Default.
- **/health** – Shows application health information (a simple 'status' when accessed over an unauthenticated connection or full message details when authenticated). It is sensitive by default.
- **/info** – Displays arbitrary application info. Not sensitive by default.
- **/metrics** – Shows 'metrics' information for the current application. It is also sensitive by default.
- **/mappings-** Displays a list of all @RequestMapping paths. Sensitive by Default.
- **/shutdown-** This endpoint allows to shutdown the application. This is not enabled by default. Sensitive by Default.
- **/trace** – Displays trace information (by default the last few HTTP requests). Sensitive by Default.
- **/logfile**– Provides access to the configured log files (This feature supported since Spring Boot 1.3.0). Sensitive by Default.
- **/flyway-** This endpoint provides the details of any flyway database migrations have been applied. Sensitive by Default.
- **/liquibase-** This endpoint provides the details of any liquibase database migrations have been applied. Sensitive by Default.

3.1 Customizing Existing Endpoints

Above listed each endpoint can be customized with properties using the following format:

endpoints.[endpoint name].[property to customize]

- id – by which this endpoint will be accessed over HTTP
- enabled – if true then it can be accessed otherwise not
- sensitive – if true then need authorization to show crucial information over HTTP

You can change how those endpoints are exposed using **application.properties**, the most common settings:

- **management.port=8081** – you can expose those endpoints on port other than the one application is using (8081 here).
- **management.address=127.0.0.1** – you can only allow to access by IP address (localhost here).
- **management.context-path=/actuator** – allows you to have those endpoints grouped under specified context path rather than root, i.e. /actuator/health.
- **endpoints.health.enabled=false** – allows to enable/disable specified endpoint by name, here /health is disabled.

Securing endpoints

For security purposes, you might choose to expose the actuator endpoints over a non-standard port – the **management.port** property can easily be used to configure that. The information exposed by endpoints is most of the time sensitive. While /health is usually harmless to be exposed, /metrics would be too much. Fortunately, you can use Spring Security for that purpose.

Adding starter to class path by adding following dependency to **pom.xml**

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

You can disable basic security it in **application.properties**, so that it leaves only the sensitive Actuator endpoints secured and leaves the rest open for access:

```
security.basic.enabled=false
```

You can secure these endpoints by defining the default security properties – user name, password and role – in **application.properties** file:

```
security.user.name=admin

security.user.password=secret

management.security.role=SUPERUSER
```

In case you're using the security features across the application and decided to secure those endpoints yourself, you can disable default security for Actuator:

```
management.security.enabled=false
```

3.2 Health information- /health Endpoint

Health information can be used to check the status of your running application. It is often used by monitoring software to alert someone if a production system goes down. The default information exposed by the health endpoint depends on how it is accessed.

http://localhost:8080/health

```
{

"status" : "UP"

}
```

This health information is collected from all the beans implementing **HealthIndicator** interface configured in your application context.

3.3 Security with HealthIndicators

Some information returned by HealthIndicator is sensitive in nature – but you can configure **endpoints.health.sensitive=false** to expose the other information like diskspace, datasource etc.

```
endpoints.health.sensitive=false
```

You can write your custom health indicator to provide additional details to the user. The following are the list of health indication implementation classes available for customization:

- DiskSpaceHealthIndicator
- DataSourceHealthIndicator
- MongoHealthIndicator
- RabbitHealthIndicator
- SolrHealthIndicator

The basic configurations for health endpoint is:

```
endpoints.health.id=health

endpoints.health.sensitive=true

endpoints.health.enabled=true
```

Customized Health checkup:

The basic idea for health checks is that they can provide more insightful information to you on the application's health. Besides checking if the application is UP or DOWN, which is done by default, you can add checks for things like database connectivity or whatever suits you. This is in fact what is being done when you add other Spring Boot starters, as they often provide additional health checks.

```
/**

 *

 */

package com.dineshonjava.sba;



import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;


/**
 * @author Dinesh.Rajput
 *
 */
public class MyAppHealth implements HealthIndicator{


 @Override
 public Health health() {
  int errorCode = check(); // perform some specific health check
  if (errorCode != 0) {
   return Health.down().withDetail("Error Code", errorCode).build();
  }
  return Health.up().build();
 }


 private int check() {
        // Your logic to check health
  return 0;
 }


}
```

As you can see all it takes is to create a bean implementing HealthIndicator with a method health() returning appropriate Health object. The checks you create will appear on the /health endpoint, so the application can be monitored for them. The output will be:

http://localhost:8080/health

```
{

status: "UP",

diskSpace: {

status: "UP",

total: 240721588224,

free: 42078715904,

threshold: 10485760

}

}
```

3.4 Application information- /info Endpoint

Application information exposes various information collected from all InfoContributor beans defined in your ApplicationContext. Spring Boot includes a number of auto-configured InfoContributors and you can also write your own. The default value for sensitive is false. There is no harm in exposing this details as that is common details that has to be exposed to others.

You can also customize the data shown by /info endpoint – for example: Configuration in **application.properties** file:

```
{

endpoints.info.id=info

endpoints.info.sensitive=false

endpoints.info.enabled=true
```

```
info.app.name=Spring Boot Actuator Application

info.app.description=This is my first Working Spring Actuator Examples

info.app.version=0.0.1-SNAPSHOT
```

And the sample output:

http://localhost:8080/info

```
{

{

app: {

version: "0.0.1-SNAPSHOT",

description: "This is my first Working Spring Actuator Examples",

name: "Spring Boot Actuator Application"

}

}
```

3.5 Metrics Information-/metrics Endpoint

The metrics endpoint is one of the more important endpoints as it gathers and publishes information about OS, JVM and Application level metrics; out of the box, we get things like memory, heap, processors, threads, classes loaded, classes unloaded, thread pools along with some HTTP metrics as well. By default this endpoint is enabled under the HTTP URL /metrics.

he example configuration for this endpoint is:

```
{
```

```
endpoints.metrics.id=metrics

endpoints.metrics.sensitive=true

endpoints.metrics.enabled=true
```

Here’s what the output of this endpoint looks like out of the box:

http://localhost:8080/metrics

```
{

{

mem: 55470,

mem.free: 5398,

processors: 4,

instance.uptime: 9452,

uptime: 14466,

systemload.average: -1,

heap.committed: 35020,

heap.init: 16384,

heap.used: 29621,

heap: 253440,

nonheap.committed: 20800,

nonheap.init: 160,

nonheap.used: 20451,

nonheap: 0,

threads.peak: 17,
threads.daemon: 14,

threads.totalStarted: 20,

threads: 16,

classes: 6542,

classes.loaded: 6542,

classes.unloaded: 0,

gc.copy.count: 60,

gc.copy.time: 232,

gc.marksweepcompact.count: 2,

gc.marksweepcompact.time: 61,

httpsessions.max: -1,

httpsessions.active: 0

}
```

There are following multiple resources matrics exposed by spring boot actualtor

3.5.1. System Metrics

The following system metrics are exposed by Spring Boot:

- The total system memory in KB (mem)
- The amount of free memory in KB (mem.free)
- The number of processors (processors)
- The system uptime in milliseconds (uptime)
- The application context uptime in milliseconds (instance.uptime)
- The average system load (systemload.average)
- Heap information in KB (heap, heap.committed, heap.init, heap.used)
- Thread information (threads, thread.peak, thread.daemon)
- Class load information (classes, classes.loaded, classes.unloaded)
- Garbage collection information (gc.xxx.count, gc.xxx.time)

3.5.2. DataSource metrics

The following metrics are exposed for each supported DataSource defined in your application:

- The number of active connections (datasource.xxx.active)
- The current usage of the connection pool (datasource.xxx.usage).

3.5.3 Cache metrics

The following metrics are exposed for each supported cache defined in your application:

- The current size of the cache (cache.xxx.size)
- Hit ratio (cache.xxx.hit.ratio)
- Miss ratio (cache.xxx.miss.ratio)

By default, Spring Boot provides cache statistics for EhCache, Hazelcast, Infinispan, JCache and Guava.

3.5.4 Tomcat session metrics

If you are using Tomcat as your embedded servlet container, session metrics will automatically be exposed. The httpsessions.active and httpsessions.max keys provide the number of active and maximum sessions.

there are many more matrics you can following spring boot docs for more info http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/.

3.6 Endpoint – Logfile

This endpoint will be automatically enabled if **logging.path** or **logging.file** properties are configured in the **application.properties** file. When you access the logile endpoint, this will return the complete log of the application.

4. Custom Metric Data

To record your own metrics inject a CounterService and/or GaugeService into your bean. The CounterService exposes increment, decrement and reset methods; the GaugeService provides a submit method.

For example – we'll customize the login flow to record a successful and failed login attempt:

```
/**
 *
 */
```

```
package com.dineshonjava.sba;


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

import org.springframework.boot.actuate.metrics.CounterService;

import org.springframework.stereotype.Service;


/**

 * @author Dinesh.Rajput

 *

 */

@Service

public class LoginService {



 CounterService counterService;


 @Autowired

 public LoginService(CounterService counterService) {

        this.counterService = counterService;

        doLogin();

   }


 public boolean login(String userName, String password) {
  boolean success;
  if (userName.equals("admin") && "secret".equals(password)) {
   counterService.increment("counter.login.success");
   success = true;
  }
  else {
   counterService.increment("counter.login.failure");
   success = false;
  }
  return success;
 }

 private void doLogin() {
  for(int i=0; i<10; i++){
   login("admin", "secret");
  }
  login("admin", "scret");
  login("admin", "scret");
 }
}
```

Here’s what the output might look like:

```
......

counter.login.failure: 2,

counter.login.success: 10

....

}
```

Besides **CounterService** the other one provided by default is **GaugeService** that is used to collect a single double value, i.e. a measured execution time. You can also create and use your own implementations of these two.

5. Create A New Endpoint

Besides using the existing endpoints provided by Spring Boot – you can also create an entirely new endpoint. This is useful when you want to expose application details which are an added feature to your application.

```
/**
 *
 */
package com.dineshonjava.sba;


import java.util.ArrayList;

import java.util.List;


import org.springframework.boot.actuate.endpoint.Endpoint;

import org.springframework.stereotype.Component;


/**
```

```
 * @author Dinesh.Rajput
 *
 */
@Component
public class MyCustomEndpoint implements Endpoint<List<String>>{

 @Override
 public String getId() {
  return "myCustomEndpoint";
 }

 @Override
 public List<String> invoke() {
  // Custom logic to build the output
       List<String> list = new ArrayList<String>();
       list.add("App message 1");
       list.add("App message 2");
       list.add("App message 3");
       list.add("App message 4");
       return list;
 }

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


 @Override
 public boolean isSensitive() {
  return true;
 }


}
```

The way to access this new endpoint is by its id, at /**myCustomEndpoint**.

http://localhost:8080/myCustomEndpoint

Output:

```
[
"App message 1",
"App message 2",
"App message 3",
"App message 4"
]
```

MyCustomEndpoint class implements Endpoint. Any class of type Endpoint will be exposed as an endpoint in the server If you look at the methods, getId(), isEnabled() and isSensitive(), which are the properties that are overridden by the **application.properties** file. invoke() is the important method responsible for writing the message.

6. A New Endpoint To List All Endpoints

It is quite useful if you have an endpoint to display all the endpoints in a single web page. There is a built-in endpoint /actuator for this purpose, but you have to add HateOAS in the classpath to enable that feature. To implement this one, you'll need to extend the AbstractEndpoint class:

```
/**
 *
 */
package com.dineshonjava.sba;


import java.util.List;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.endpoint.AbstractEndpoint;
import org.springframework.boot.actuate.endpoint.Endpoint;
import org.springframework.stereotype.Component;


/**
 * @author Dinesh.Rajput
 *
 */
@Component
```

```
@SuppressWarnings("rawtypes")

public class MyListEndpoints extends AbstractEndpoint<List<Endpoint>>{


 List<Endpoint> endpoints;


 @Autowired

    public MyListEndpoints(List<Endpoint> endpoints) {

        super("myListEndpoints");

        this.endpoints = endpoints;

    }


 @Override

 public List<Endpoint> invoke() {

  return this.endpoints;

 }


}
```

Here’s how the output will look like:

When you implement the above class, there will be a new endpoint “myListEndpoints” will be registered and exposed to the users. The output will be:

http://localhost:8080/myListEndpoints

```
[
{
id: "myCustomEndpoint",
enabled: true,
sensitive: true
},
{
id: "mappings",
sensitive: true,
enabled: true
},
{
id: "env",
sensitive: true,
enabled: true
},
{
id: "health",
sensitive: false,
enabled: true,
timeToLive: 1000
},
{
```

```
id: "beans",

sensitive: true,

enabled: true

},

{

id: "info",

sensitive: false,

enabled: true

},

{

id: "metrics",

sensitive: false,

enabled: true

},

{

id: "trace",

sensitive: true,

enabled: true

},

{

id: "dump",

sensitive: true,

enabled: true
```

```
},
{
id: "autoconfig",
sensitive: true,
enabled: true
},
{
id: "shutdown",
sensitive: true,
enabled: false
},
{
id: "configprops",
sensitive: true,
enabled: true
}
]
```

7. Actuator Example application

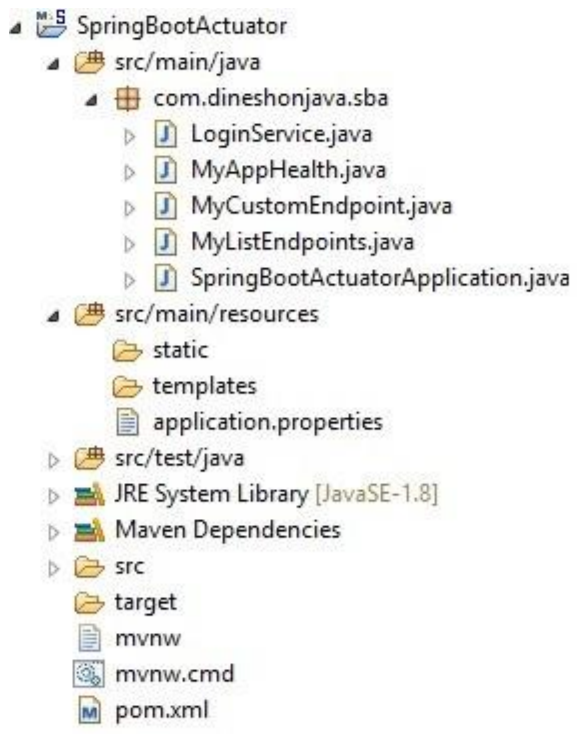
In this section we will make the application which have all required configuration. For creating the application we have used here Spring Boot Web Initializr with Maven Build and following dependencies in pom.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<project            xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

 <modelVersion>4.0.0</modelVersion>



 <groupId>com.dineshonjava.sba</groupId>

 <artifactId>SpringBootActuator</artifactId>

 <version>0.0.1-SNAPSHOT</version>

 <packaging>jar</packaging>



 <name>SpringBootActuator</name>

 <description>SpringBootActuator  project  for  Spring  Boot
Actuator</description>



 <parent>

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

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

  <version>1.4.0.RELEASE</version>

  <relativePath/> <!-- lookup parent from repository -->

 </parent>



 <properties>

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

```
  <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>

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

 </properties>


 <dependencies>

  <dependency>

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

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

  </dependency>

  <dependency>

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

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

  </dependency>

  <dependency>

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

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

  </dependency>


  <dependency>

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

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

   <scope>test</scope>

  </dependency>
 </dependencies>


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



</project>
```

And below is project struct as we have finally.



application.properties file for configuration for actuator

```
#/info endpoint configuration

endpoints.info.id=info

endpoints.info.sensitive=false

endpoints.info.enabled=true

info.app.name=Spring Boot Actuator Application

info.app.description=This is my first Working Spring Actuator
Examples

info.app.version=0.0.1-SNAPSHOT



#/metrics endpoint configuration

endpoints.metrics.id=metrics

endpoints.metrics.sensitive=false
endpoints.metrics.enabled=true


#securing endpoints by spring security

security.basic.enabled=true

security.user.name=admin

security.user.password=secret


#/health endpoint configuration (Comment when you are using
customized health check)

endpoints.health.id=health

endpoints.health.sensitive=false

endpoints.health.enabled=true


#Management for endpoints

management.port=8080

management.context-path=/

management.security.enabled=true
```

Whole Code: Here I am not going put all required files for this application you can find whole application from following **git** repository.

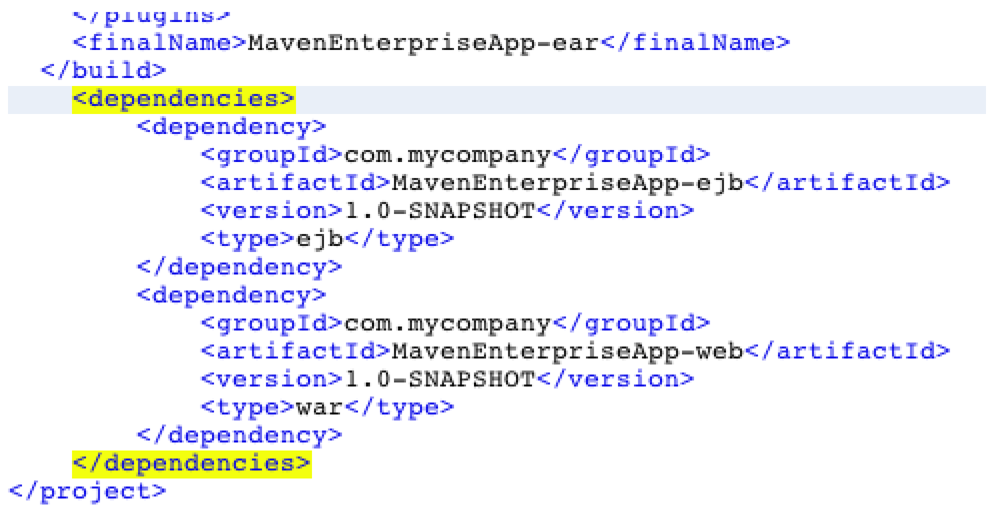
https://github.com/DOJ-SoftwareConsultant/SpringBootActuator

8. Summary:

In this tutorial we had a first look at the interesting Actuator functionality provided by Spring Boot. I have explained about the basic concepts on actuators, endpoints, list of endpoints, creating custom endpoints, health check,metrics and provided a complete working example application for spring boot actuator.

Maven Pom.xml File

Pom is short form of Project Object Model. Maven POM file in the project describe the resources of the project. This include all dependencies, directories of source code, test, plugin, goals etc. The POM file is named pom.xml. In earlier version of Maven (before 2.0) the name of this file was project.xml after version 2.0 it had changes to pom.xml. The pom file should be located into the root directory of the project.



```
        </plugins>
        <finalName>MavenEnterpriseApp-ear</finalName>
    </build>
        <dependencies>
            <dependency>
                <groupId>com.mycompany</groupId>
                <artifactId>MavenEnterpriseApp-ejb</artifactId>
                <version>1.0-SNAPSHOT</version>
                <type>ejb</type>
            </dependency>
            <dependency>
                <groupId>com.mycompany</groupId>
                <artifactId>MavenEnterpriseApp-web</artifactId>
                <version>1.0-SNAPSHOT</version>
                <type>war</type>
            </dependency>
        </dependencies>
</project>
```

POM file also has inheritance concept means a project divided into sub-projects. So parent project has one POM file and each sub-project have own POM file with inheriting parent POM file into the parent project. So with this structure we can built a project in one step including all sub-projects or either we can build one sub-project separately.

Maven reads the pom.xml file, then executes the goal.

Whenever we are creating POM file in our project first we should decide the project group id (i.e. groupId) and its name (i.e. artifactId) and its version for versioning of project in the repository it help in identifying.

Here is a minimal POM file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"

         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

                      http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.dineshonjava</groupId>

    <artifactId>java-api-learner</artifactId>

    <version>1.0.0.RELEASE</version>

</project>
```

XML elements of Maven POM file

1. **project**-It is the root XML element of POM file.
2. **modelVersion**-It is the sub element of project. It specifies the modelVersion. It should be set to 4.0.0.
3. **groupId**-It is name of group or organization of project. It is the sub element of project element.
4. **artifactId**– It is name of project (~artifact) what we are creating and it is the sub element of project element. An artifact is something that is either produced or used by a project.
5. **version**-It specifies the version of the artifact under given group.

The above groupId, artifactId and version elements would result in a JAR file being built and put into the local Maven repository at the following path:

MAVEN_REPO/com/dineshonjava/java-api-learner/1.0.0.RELEASE/java-api-learner-1.0.0.RELEASE.jar
As above listed elements are mandatory. All POM files require at least the project element and three mandatory fields: groupId, artifactId, version.

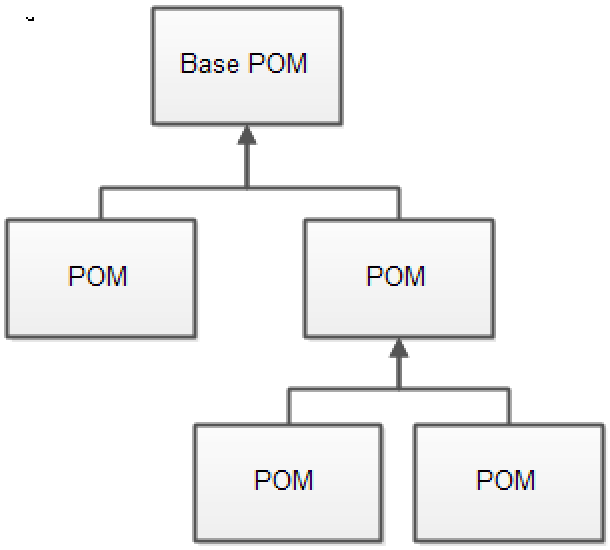
Super POM

All Maven POM files inherit from a super POM. If no super POM is specified, the POM file inherits from the base POM just like Object class in java.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"

         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

                      http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
```

```
        <parent>

          <groupId>org.dineshonjava</groupId>

          <artifactId>parent-project</artifactId>

          <version>2.0</version>

           <relativePath>../parent-project</relativePath>

        </parent>




    <artifactId>parent-project</artifactId>

    ...

</project>
```

An inheriting POM file may override settings from a super POM. Just specify new settings in the inheriting POM file

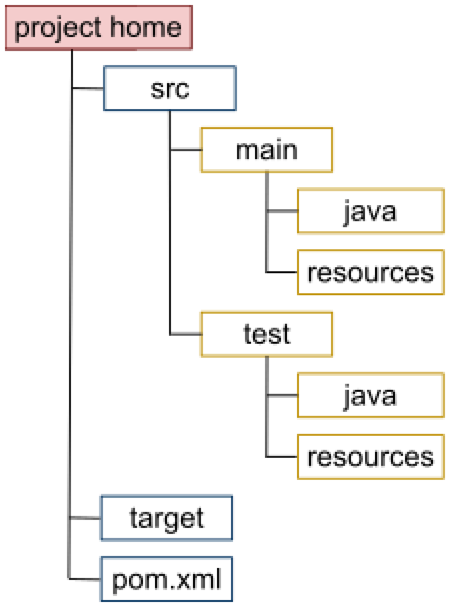


@ImageSource-tutorials.jenkov.com

An easy way to look at the default configurations of the super POM is by running the following command: ***mvn help:effective-pom***

Maven Directory Structure

Maven Directory Structure
Maven has own standard for directory structure to create in a project. So we no need specify any directory on the project level for sources, resources, tests, plugins etc. And also we don’t need to configure for creating directory structure on the pom.xml file it is internal functionality of maven.



@ImageSource-Wikipedia.com
– src
– main
– java
– resources
– webapp
– test
– java
– resources

– target

Directories level on project created by Maven

Source Directory

-src This is src directory which contains two more such as **main** and **test** directories inside. The main directory is for your source code and test directory is for test code.
-main Under the main directory there are many more directories such as **java**, **resources**, **webapps** etc.
-java The **java** directory contains all java codes and packages

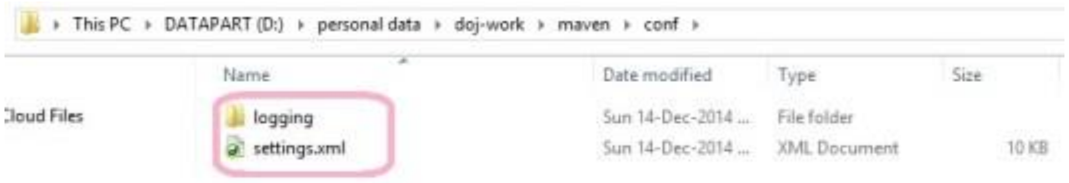
-resources The **resources** directory contains other resources needed by your project. This could be property files used for internationalization of an application, or something else.
-webapp The **webapp** directory contains your Java web application, if your project is a web application.
-test Under test directory contains two directories such as **java** and **resources**.

Target Directory

-target The target directory is created by Maven. It contains all the compiled classes, JAR files etc. produced by Maven. When executing the clean build phase, it is the target directory which is cleaned.

Maven Settings File

There are two setting files maven have. These setting files have information about maven local repository, active build profile etc.The settings element in the settings.xml file contains elements used to define values which configure Maven execution in various ways, like the pom.xml, but should not be bundled to any specific project, or distributed to an audience. These include values such as the local repository location, alternate remote repository servers, and authentication information.



There are two locations where a **settings.xml** file may live:

1. The Maven install: **${maven.home}/conf/settings.xml**
2. A user's install: **${user.home}/.m2/settings.xml (from maven 3.0)**

Both files are optional. If both files are present, the values in the user home settings file overrides the values in the Maven installation settings file.

The former settings.xml are also called global settings, the latter settings.xml are referred to as user settings. If both files exists, their contents gets merged, with the user-specific settings.xml being dominant.

global settings.xm

```
<?xml version="1.0" encoding="UTF-8"?>


<!--

Licensed to the Apache Software Foundation (ASF) under one

or more contributor license agreements.  See the NOTICE file

distributed with this work for additional information
```

```
regarding copyright ownership.  The ASF licenses this file

to you under the Apache License, Version 2.0 (the

"License"); you may not use this file except in compliance

with the License.  You may obtain a copy of the License at



    http://www.apache.org/licenses/LICENSE-2.0



Unless required by applicable law or agreed to in writing,

software distributed under the License is distributed on an

"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY

KIND, either express or implied.  See the License for the

specific language governing permissions and limitations

under the License.

-->



<!--

 | This is the configuration file for Maven. It can be specified at two
levels:

 |

 |  1. User Level. This settings.xml file provides configuration for a
single user,

 |                 and is normally provided in ${user.home}/.m2/settings.xml.

 |

 |                 NOTE: This location can be overridden with the CLI option:
```

```
 |
 |                  -s /path/to/user/settings.xml
 |
 |  2. Global Level. This settings.xml file provides configuration for all
Maven
 |                  users on a machine (assuming they're all using the same
Maven
 |                  installation). It's normally provided in
 |                  ${maven.home}/conf/settings.xml.
 |
 |                NOTE: This location can be overridden with the CLI option:
 |
 |                  -gs /path/to/global/settings.xml
 |
 | The sections in this sample file are intended to give you a running start
at
 | getting the most out of your Maven installation. Where appropriate, the
default
 | values (values used when the setting is not specified) are provided.
 |
 |-->
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
```

```
  <!-- localRepository

   | The path to the local repository maven will use to store artifacts.

   |

   | Default: ${user.home}/.m2/repository

  <localRepository>/path/to/local/repo</localRepository>

  -->



  <!-- interactiveMode

   | This will determine whether maven prompts you when it needs input. If
set to false,

   | maven will use a sensible default value, perhaps based on some other
setting, for

   | the parameter in question.

   |

   | Default: true

  <interactiveMode>true</interactiveMode>

  -->



  <!-- offline

   | Determines whether maven should attempt to connect to the network when
executing a build.

   | This will have an effect on artifact downloads, artifact deployment,
and others.

   |

   | Default: false
  <offline>false</offline>
  -->


  <!-- pluginGroups
   | This is a list of additional group identifiers that will be searched when resolving plugins by their prefix, i.e.
   | when invoking a command line like "mvn prefix:goal". Maven will automatically add the group identifiers
   | "org.apache.maven.plugins" and "org.codehaus.mojo" if these are not already contained in the list.
   |-->
  <pluginGroups>
    <!-- pluginGroup
     | Specifies a further group identifier to use for plugin lookup.
    <pluginGroup>com.your.plugins</pluginGroup>
    -->
  </pluginGroups>


  <!-- proxies
   | This is a list of proxies which can be used on this machine to connect to the network.
   | Unless otherwise specified (by system property or command-line switch), the first proxy
   | specification in this list marked as active will be used.
   |-->
  <proxies>

    <!-- proxy

     | Specification for one proxy, to be used in connecting to the network.

     |

    <proxy>

      <id>optional</id>

      <active>true</active>

      <protocol>http</protocol>

      <username>proxyuser</username>

      <password>proxypass</password>

      <host>proxy.host.net</host>

      <port>80</port>

      <nonProxyHosts>local.net|some.host.com</nonProxyHosts>

    </proxy>

    -->

  </proxies>



  <!-- servers

   | This is a list of authentication profiles, keyed by the server-id used
within the system.

   | Authentication profiles can be used whenever maven must make a
connection to a remote server.

   |-->

  <servers>
```

```
    <!-- server

     | Specifies the authentication information to use when connecting to
a particular server, identified by

     | a unique name within the system (referred to by the 'id' attribute
below).

     |

     |   NOTE:   You   should   either   specify   username/password   OR
privateKey/passphrase, since these pairings are

     |       used together.

     |

    <server>

      <id>deploymentRepo</id>

      <username>repouser</username>

      <password>repopwd</password>

    </server>

    -->


    <!-- Another sample, using keys to authenticate.

    <server>

      <id>siteServer</id>

      <privateKey>/path/to/private/key</privateKey>

      <passphrase>optional; leave empty if not used.</passphrase>

    </server>

    -->

  </servers>
  <!-- mirrors

   | This is a list of mirrors to be used in downloading artifacts from remote repositories.

   |

   | It works like this: a POM may declare a repository to use in resolving certain artifacts.

   | However, this repository may have problems with heavy traffic at times, so people have mirrored

   | it to several places.

   |

   | That repository definition will have a unique id, so we can create a mirror reference for that

   | repository, to be used as an alternate download site. The mirror site will be the preferred

   | server for that repository.

   |-->

  <mirrors>

    <!-- mirror

     | Specifies a repository mirror site to use instead of a given repository. The repository that

     | this mirror serves has an ID that matches the mirrorOf element of this mirror. IDs are used

     | for inheritance and direct lookup purposes, and must be unique across the set of mirrors.

     |

    <mirror>
```

```
      <id>mirrorId</id>

      <mirrorOf>repositoryId</mirrorOf>

      <name>Human Readable Name for this Mirror.</name>

      <url>http://my.repository.com/repo/path</url>

    </mirror>

     -->

  </mirrors>



  <!-- profiles

   | This is a list of profiles which can be activated in a variety of ways, and which can modify

   | the build process. Profiles provided in the settings.xml are intended to provide local machine-

   | specific paths and repository locations which allow the build to work in the local environment.

   |

   | For example, if you have an integration testing plugin - like cactus - that needs to know where

   | your Tomcat instance is installed, you can provide a variable here such that the variable is

   | dereferenced during the build process to configure the cactus plugin.

   |

   | As noted above, profiles can be activated in a variety of ways. One way - the activeProfiles

   | section of this document (settings.xml) - will be discussed later. Another way essentially
   | relies on the detection of a system property, either matching a particular value for the property,

   | or merely testing its existence. Profiles can also be activated by JDK version prefix, where a

   | value of '1.4' might activate a profile when the build is executed on a JDK version of '1.4.2_07'.

   | Finally, the list of active profiles can be specified directly from the command line.

   |

   | NOTE: For profiles defined in the settings.xml, you are restricted to specifying only artifact

   |       repositories, plugin repositories, and free-form properties to be used as configuration

   |       variables for plugins in the POM.

   |

   |-->

  <profiles>

    <!-- profile

     | Specifies a set of introductions to the build process, to be activated using one or more of the

     | mechanisms described above. For inheritance purposes, and to activate profiles via <activatedProfiles/>

     | or the command line, profiles have to have an ID that is unique.

     |

     | An encouraged best practice for profile identification is to use a consistent naming convention

     | for profiles, such as 'env-dev', 'env-test', 'env-production', 'user-jdcasey', 'user-brett', etc.
     | This will make it more intuitive to understand what the set of
introduced profiles is attempting

     | to accomplish, particularly when you only have a list of profile id's
for debug.

     |

     | This profile example uses the JDK version to trigger activation, and
provides a JDK-specific repo.

    <profile>

      <id>jdk-1.4</id>



      <activation>

        <jdk>1.4</jdk>

      </activation>



      <repositories>

        <repository>

          <id>jdk14</id>

          <name>Repository for JDK 1.4 builds</name>

          <url>http://www.myhost.com/maven/jdk14</url>

          <layout>default</layout>

          <snapshotPolicy>always</snapshotPolicy>

        </repository>

      </repositories>

    </profile>

    -->
    <!--
     | Here is another profile, activated by the system property 'target-env' with a value of 'dev',
     | which provides a specific path to the Tomcat instance. To use this, your plugin configuration
     | might hypothetically look like:
     |
     | ...
     | <plugin>
     |   <groupId>org.myco.myplugins</groupId>
     |   <artifactId>myplugin</artifactId>
     |
     |   <configuration>
     |     <tomcatLocation>${tomcatPath}</tomcatLocation>
     |   </configuration>
     | </plugin>
     | ...
     |
     | NOTE: If you just wanted to inject this configuration whenever someone set 'target-env' to
     |       anything, you could just leave off the <value/> inside the activation-property.
     |
    <profile>
      <id>env-dev</id>

      <activation>
        <property>
          <name>target-env</name>
          <value>dev</value>
        </property>
      </activation>

      <properties>
        <tomcatPath>/path/to/tomcat/instance</tomcatPath>
      </properties>
    </profile>
    -->
  </profiles>

  <!-- activeProfiles
   | List of profiles that are active for all builds.
   |
  <activeProfiles>
    <activeProfile>alwaysActiveProfile</activeProfile>
    <activeProfile>anotherAlwaysActiveProfile</activeProfile>
  </activeProfiles>
```

```
  -->

</settings>
```

Maven Example Hello World

Here we are creating simple Maven Example Hello World using command prompt by executing the archetype:generate command of mvn tool.First of all going to any directory of computer machine and open command prompt. Here I am using git bash as command prompt. For creating a simple hello java project using maven, we have to open command prompt and run the archetype:generate command of mvn tool.

The command as following we have used to create project-

Command-

```
mvn archetype:generate -DgroupId=groupid -DartifactId=artifactid

-DarchetypeArtifactId=maven-archetype-quickstart                          -
DinteractiveMode=booleanValue
```

Creating Java Hello World Maven Project

Above is the syntax for creating java project so let see below for actually create command.

```
mvn        archetype:generate        -DgroupId=com.dineshonjava        -
DartifactId=JavaHelloWorld         -DarchetypeArtifactId=maven-archetype-
quickstart -DinteractiveMode=false
```

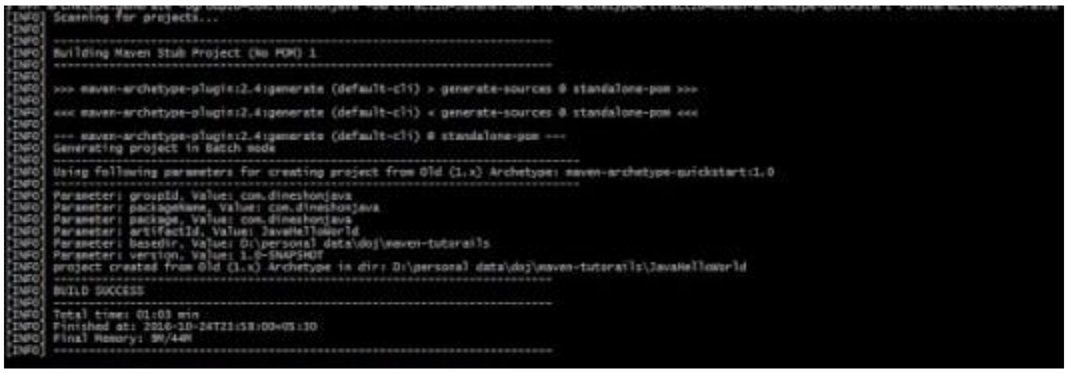
After running this command following is output:

Now it will generate following code in the command prompt:

```
$      mvn      archetype:generate      -DgroupId=com.dineshonjava      -
DartifactId=JavaHelloWorld         -DarchetypeArtifactId=maven-archetype-
quickstart -DinteractiveMode=false

[INFO] Scanning for projects...
```

```
[INFO]

[INFO] ------------------------------------------------------------------------
-----

[INFO] Building Maven Stub Project (No POM) 1

[INFO] ------------------------------------------------------------------------
-----

[INFO]

[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) > generate-
sources @ standalone-pom >>>

[INFO]

[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) < generate-
sources @ standalone-pom <<<

[INFO]

[INFO] --- maven-archetype-plugin:2.4:generate (default-cli) @ standalone-
pom ---

[INFO] Generating project in Batch mode

[INFO] ------------------------------------------------------------------------
---------

[INFO] Using following parameters for creating project from Old (1.x)
Archetype: maven-archetype-quickstart:1.0

[INFO] ------------------------------------------------------------------------
---------

[INFO] Parameter: groupId, Value: com.dineshonjava

[INFO] Parameter: packageName, Value: com.dineshonjava

[INFO] Parameter: package, Value: com.dineshonjava

[INFO] Parameter: artifactId, Value: JavaHelloWorld

[INFO] Parameter: basedir, Value: D:personal datadojmaven-tutorails
```

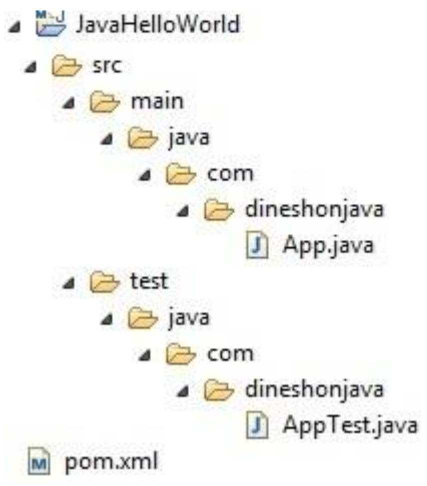
```
[INFO] Parameter: version, Value: 1.0-SNAPSHOT

[INFO] project created from Old (1.x) Archetype in dir: D:personal
datadojmaven-tutorailsJavaHelloWorld

[INFO] ------------------------------------------------------------------
-----

[INFO] BUILD SUCCESS

[INFO] ------------------------------------------------------------------
-----

[INFO] Total time: 01:03 min

[INFO] Finished at: 2016-10-24T23:58:00+05:30

[INFO] Final Memory: 9M/44M

[INFO] ------------------------------------------------------------------
-----
```



```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.4:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.4:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.4:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Batch mode
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.dineshonjava
[INFO] Parameter: packageName, Value: com.dineshonjava
[INFO] Parameter: package, Value: com.dineshonjava
[INFO] Parameter: artifactId, Value: JavaHelloWorld
[INFO] Parameter: basedir, Value: D:\personal data\doj\maven-tutorails
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: D:\personal data\doj\maven-tutorails\JavaHelloWorld
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:03 min
[INFO] Finished at: 2016-10-24T23:58:00+05:30
[INFO] Final Memory: 9M/44M
[INFO] ------------------------------------------------------------------------
```

Created Project Structure-

Now go to directory where we have created java project. We got following directory structure.



Created files:

Following files are auto generated into java project.

Maven pom.xml file

```
<project                          xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.dineshonjava</groupId>

  <artifactId>JavaHelloWorld</artifactId>

  <packaging>jar</packaging>

  <version>1.0-SNAPSHOT</version>

  <name>JavaHelloWorld</name>

  <url>http://maven.apache.org</url>

  <dependencies>

    <dependency>

      <groupId>junit</groupId>
      <artifactId>junit</artifactId>

      <version>3.8.1</version>

      <scope>test</scope>

    </dependency>

  </dependencies>

</project>
```

App.java

```
package com.dineshonjava;



/**

 * Hello world!

 *

 */

public class App

{

    public static void main( String[] args )

    {

        System.out.println( "Hello World!" );

    }

}
```

AppTest.java

```
package com.dineshonjava;


import junit.framework.Test;

import junit.framework.TestCase;

import junit.framework.TestSuite;


/**

 * Unit test for simple App.

 */

public class AppTest

    extends TestCase

{

    /**

     * Create the test case

     *

     * @param testName name of the test case

     */

    public AppTest( String testName )

    {

        super( testName );

    }
    /**

     * @return the suite of tests being tested

     */

    public static Test suite()

    {

        return new TestSuite( AppTest.class );

    }



    /**


     * Rigourous Test

     */

    public void testApp()

    {

        assertTrue( true );

    }

}
```

Run the created java project

For running created java project we have to compile first then we can run it.

For compile use this command “**mvn clean compile**”

```
$ mvn clean compile

[INFO] Scanning for projects...
```

```
[INFO]

[INFO] ------------------------------------------------------------------------
-----

[INFO] Building JavaHelloWorld 1.0-SNAPSHOT

[INFO] ------------------------------------------------------------------------
-----

[INFO]

[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ JavaHelloWorld -
--

[INFO]

[INFO]  ---  maven-resources-plugin:2.6:resources  (default-resources)  @
JavaHelloWorld ---

[WARNING] Using platform encoding (Cp1252 actually) to copy filtered
resources, i.e. build is platform dependent!

[INFO]  skip  non  existing  resourceDirectory  D:personal  datadojmaven-
tutorailsJavaHelloWorldsrcmainresources

[INFO]

[INFO]   ---   maven-compiler-plugin:3.1:compile   (default-compile)   @
JavaHelloWorld ---

[INFO] Changes detected - recompiling the module!

[WARNING] File encoding has not been set, using platform encoding Cp1252,
i.e. build is platform dependent!

[INFO]   Compiling   1   source   file   to   D:personal   datadojmaven-
tutorailsJavaHelloWorldtargetclasses

[INFO] ------------------------------------------------------------------------
-----

[INFO] BUILD SUCCESS
```

```
[INFO] ------------------------------------------------------------------------
-----

[INFO] Total time: 3.829 s

[INFO] Finished at: 2016-10-25T00:19:50+05:30

[INFO] Final Memory: 9M/22M

[INFO] ------------------------------------------------------------------------
-----
```

For run this project go to the project **directorytargetclasses**,

Create Microservices Architecture Spring Boot

In this Microservices Architecture Spring Boot tutorial, we will discuss to creating a microservices with spring and will see microservices architecture. Microservices allow large systems to be built up from a number of collaborating components. Microservices allows doing loose coupling between application processes instead of loose coupling between application components as Spring does.

Microservices Architecture Spring Boot **Table of Contents**

1. Introduction

Microservices is not a new term. It coined in 2005 by Dr Peter Rodgers then called micro web services based on SOAP. It became more popular since 2010. Microservices allows us to break our large system into the number of independent collaborating processes. Let us see below microservices architecture.

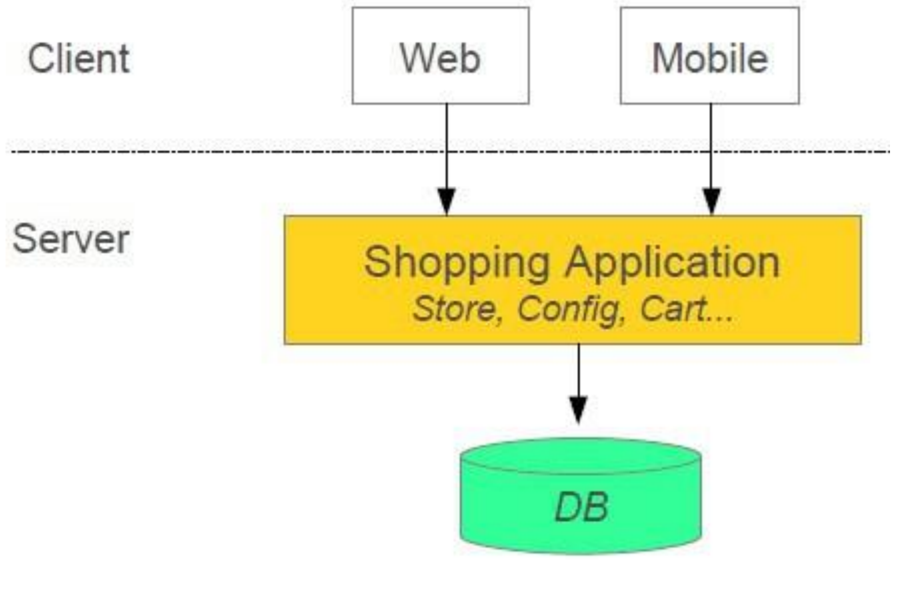
1.1 What is Microservices Architecture?

Microservices architecture allows avoiding monolith application for the large system. It provides loose coupling between collaborating processes which running independently in different environments with tight cohesion. So let’s discuss it by an example as below.

For example imagine an online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts. So these components are inevitably important for such a large online shopping portal. For online shopping system, we could use following architectures.

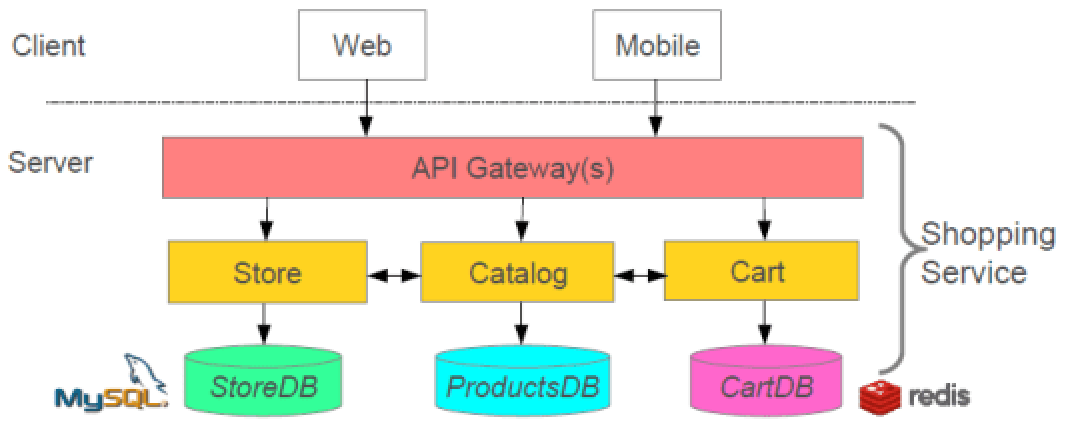
1.2 Shopping system without Microservices (Monolith architecture)

In this architecture we are using Monolith architecture i.e. all collaborating components combine all in one application.



1.3 Shopping system with Microservices

In this architecture style, the main application divided into a set of sub-applications called microservices. One large Application divided into multiple collaborating processes as below.



Spring enables separation-of-concerns

- ***Loose Coupling***– Effect of changes isolated
- ***Tight Cohesion***– Code perform a single well-defined task

Microservices provide the same strength as Spring provide

- ***Loose Coupling***– Application build from collaboration services or processes, so any process change without affecting another process.
- ***Tight Cohesion***-An individual service or process that deals with a single view of data.

2. Microservices Benefits

- The smaller code base is easy to maintain.
- Easy to scale as an individual component.
- Technology diversity i.e. we can mix libraries, databases, frameworks etc.
- Fault isolation i.e. a process failure should not bring the whole system down.
- Better support for smaller and parallel team.
- Independent deployment
- Deployment time reduce

3. Microservices Challenges

- Difficult to achieve strong consistency across services
- ACID transactions do not span multiple processes.
- Distributed System so hard to debug and trace the issues
- Greater need for an end to end testing
- Required cultural changes in across teams like Dev and Ops working together even in the same team.

4. Microservices Infrastructure

- Platform as a Service like Pivotal Cloud Foundry help to deployment, easily run, scale, monitor etc.
- It supports for continuous deployment, rolling upgrades fo new versions of code, running multiple versions of the same service at same time.

5. Microservices Tooling Supports

5.1 Using Spring for creating Microservices

- Setup new service by using Spring Boot
- Expose resources via a RestController
- Consume remote services using RestTemplate

5.2 Adding Spring Cloud and Discovery server

What is Spring Cloud?

- It is building blocks for Cloud and Microservices
- It provides microservices infrastructure like provide use services such as Service Discovery, a Configuration server and Monitoring.
- It provides several other open source projects like Netflix OSS.
- It provides PaaS like Cloud Foundry, AWS and Heroku.
- It uses Spring Boot style starters

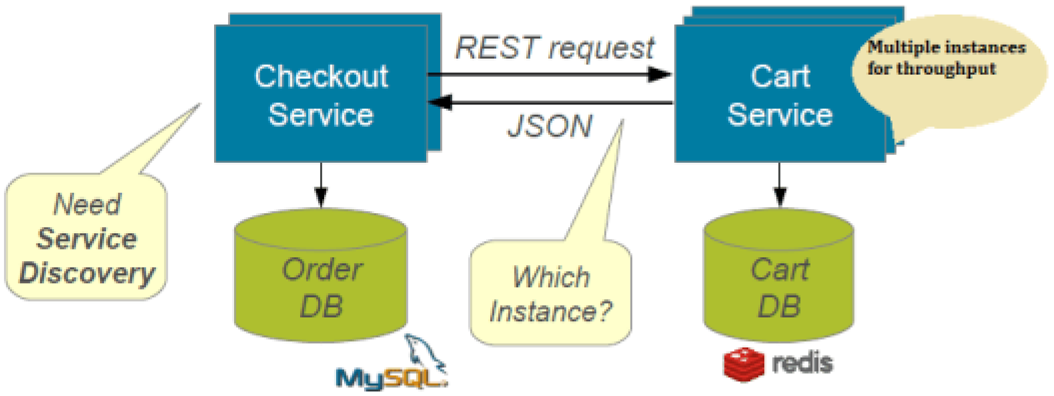
There are many use-cases supported by Spring Cloud like Cloud Integration, Dynamic Reconfiguration, Service Discovery, Security, Client-side Load Balancing etc. But in this post we concentrate on following microservices support

- Service Discovery (How do services find each other?)
- Client-side Load Balancing (How do we decide which service instance to use?)

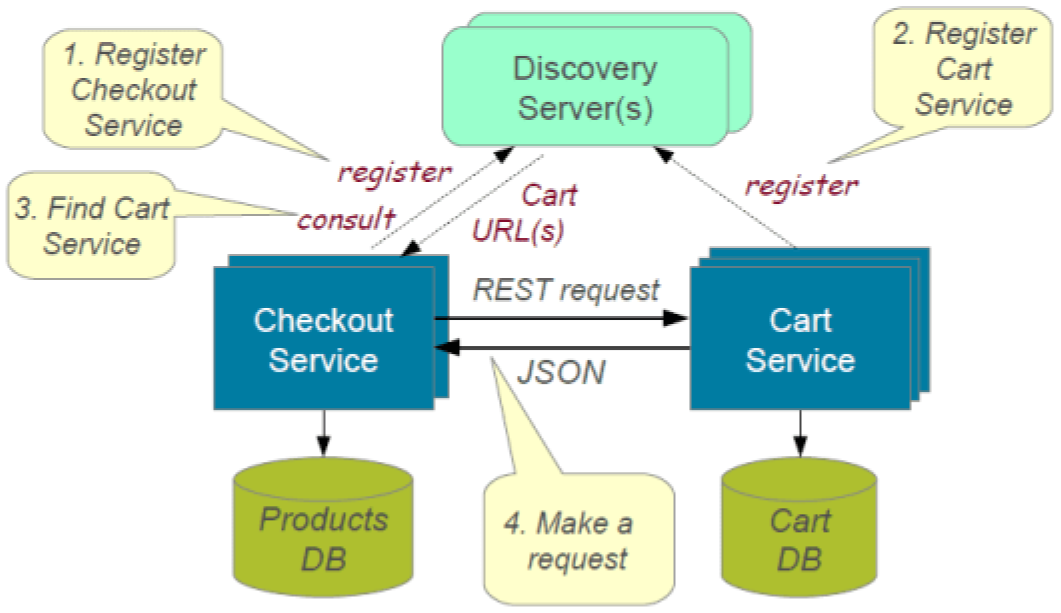
Service Discovery

Problem without discovery

- How do services find each other?
- What happens if we run multiple instances for a service



Resolution with service discovery



Implementing Service Discovery

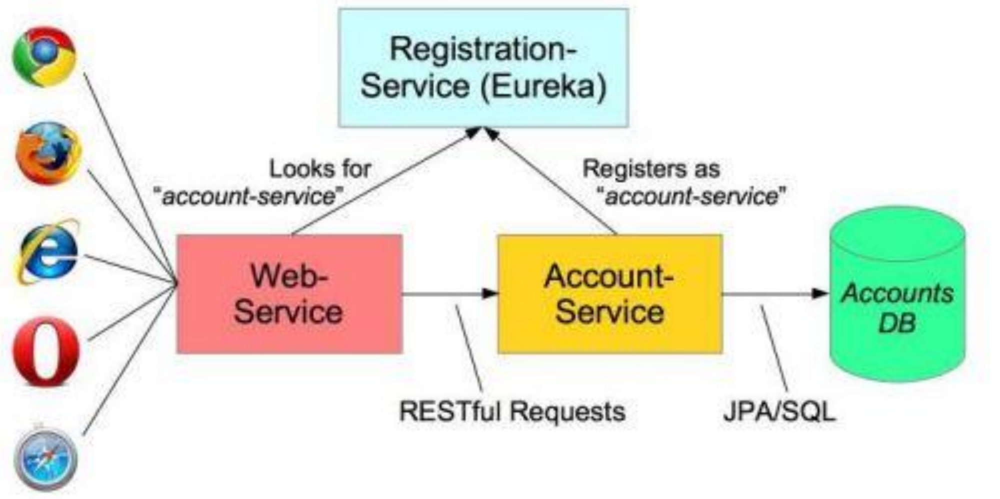
Spring Cloud support several ways to implement service discovery but for this, I am going to use Eureka created by Netflix. Spring Cloud provides several annotation to make it use easy and hiding lots of complexity.

Client-side Load Balancing

Each service typically deployed as multiple instances for fault tolerance and load sharing. But there is the problem how to decide which instance to use?

Implementing Client-Side Load Balancing

We will use Netflix Ribbon, it provides several algorithms for Client-Side Load Balancing. Spring provides smart **RestTemplate** for service discovery and load balancing by using ***@LoadBalanced*** annotation with **RestTemplate** instance.



@ImageSource-Spring.io

6. Developing Simple Microservices Example

To build a simple microservices system following steps required

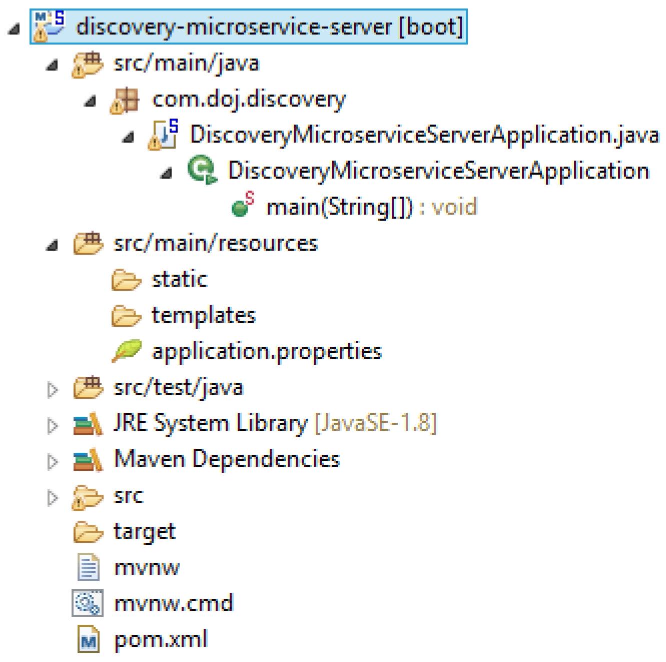
1. Creating Discovery Service (Creating Eureka Discovery Service)
2. Creating MicroService (the Producer)
    1. Register itself with Discovery Service with logical service.
3. Create Microservice Consumers find Service registered with Discovery Service
    1. Discovery client using a smart **RestTemplate** to find microservice.

Maven Dependencies

```
<dependencies>

  <dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter</artifactId>

  </dependency>

  <dependency>

   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-eureka</artifactId>

  </dependency>

  <dependency>

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

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

  </dependency>



  <dependency>

   <groupId>org.hsqldb</groupId>

   <artifactId>hsqldb</artifactId>

   <scope>runtime</scope>

  </dependency>

  <dependency>

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

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

   <scope>test</scope>

  </dependency>

 </dependencies>
```

Step 1: Creating Discovery Service (Creating Eureka Discovery Service)

- Eureka Server using Spring Cloud
- We need to implement our own registry service as below.



```
discovery-microservice-server [boot]
  src/main/java
    com.doj.discovery
      DiscoveryMicroserviceServerApplication.java
        DiscoveryMicroserviceServerApplication
          main(String[]) : void
  src/main/resources
    static
    templates
    application.properties
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

application.yml

```
# Configure this Discovery Server

eureka:

  instance:

    hostname: localhost

  client: #Not a client

    registerWithEureka: false

    fetchRegistry: false



# HTTP (Tomcat) port

server:

  port: 1111
```

DiscoveryMicroserviceServerApplication.java

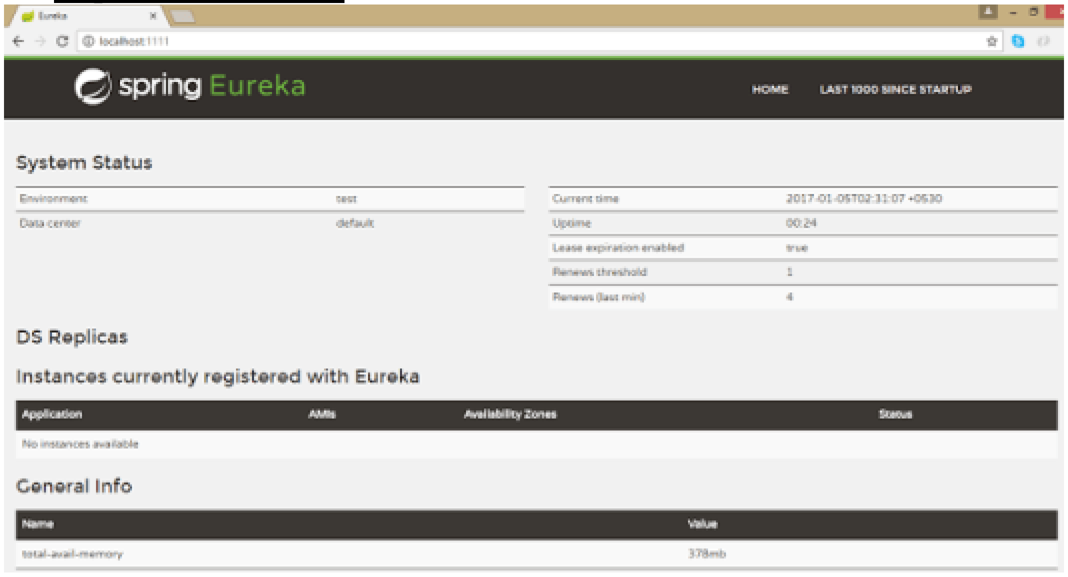
```java
package com.doj.discovery;


import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;


@SpringBootApplication

@EnableEurekaServer

public class DiscoveryMicroserviceServerApplication {


 public static void main(String[] args) {

  SpringApplication.run(DiscoveryMicroserviceServerApplication.class, args);

 }

}
```

pom.xml

```xml
<!-- Eureka registration server -->

  <dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter-eureka-server</artifactId>

  </dependency>
```

For Whole Source Code for the Discover Server Application, you could download from GitHub as below link.

discovery-microservice-server
Run this Eureka Server application with right click and run as Spring Boot Application and open in browser **http://localhost:1111/**



localhost:1111

spring Eureka

HOME LAST 1000 SINCE STARTUP

System Status

Environment	test
Data center	default

Current time	2017-01-05T02:31:07 +0530
Uptime	00:24
Lease expiration enabled	true
Renews threshold	1
Renews (last min)	4

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

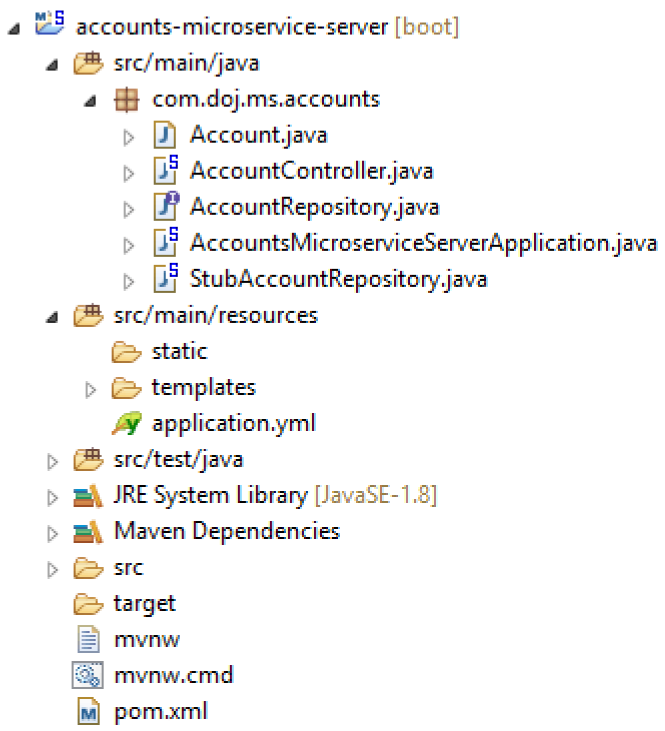
General Info

Name	Value
total-avail-memory	378mb

Step 2: Creating Account Producer MicroService
Microservice declares itself as an available service and register to Discovery Server created in **Step 1**.

- Using ***@EnableDiscoveryClient***
- Registers using its application name

Let's see the service producer application structure as below.



```
accounts-microservice-server [boot]
  src/main/java
    com.doj.ms.accounts
      Account.java
      AccountController.java
      AccountRepository.java
      AccountsMicroserviceServerApplication.java
      StubAccountRepository.java
  src/main/resources
    static
    templates
    application.yml
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

application.yml

```
### Spring properties

# Service registers under this name

spring:

  application:

    name: accounts-microservice



# Discovery Server Access

eureka:

    client:

      serviceUrl:

        defaultZone: http://localhost:1111/eureka/
```

```
# HTTP Server (Tomcat) Port

server:

  port: 2222



# Disable Spring Boot's "Whitelabel" default error page, so we can use our
own

error:

  whitelabel:

    enabled: false
```

AccountsMicroserviceServerApplication.java

```
package com.doj.ms.accounts;



import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;



@SpringBootApplication

@EnableDiscoveryClient

public class AccountsMicroserviceServerApplication {



 public static void main(String[] args) {

  SpringApplication.run(AccountsMicroserviceServerApplication.class, args);

 }
}
```

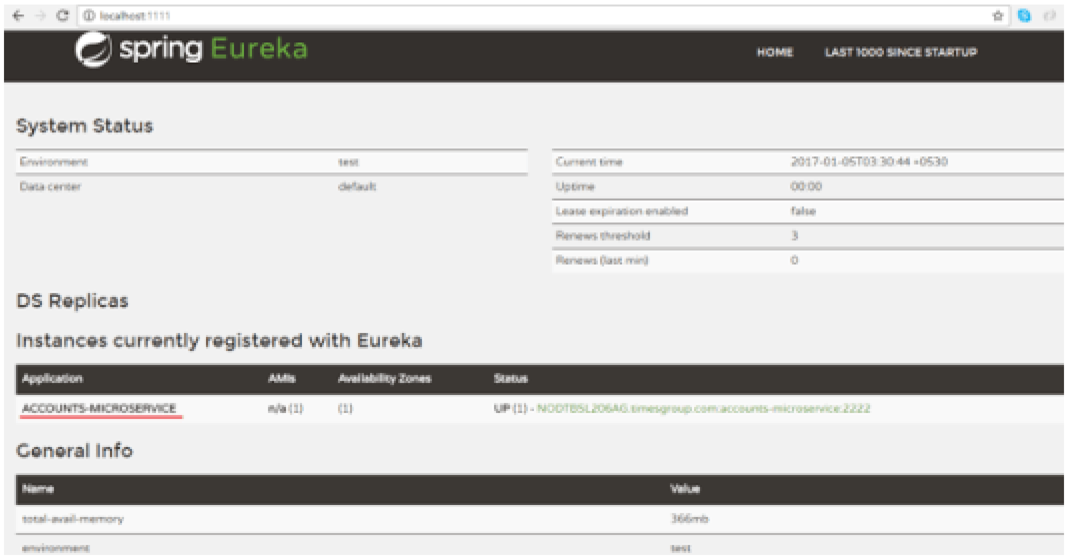
pom.xml

```
<dependencies>

  <dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter</artifactId>

  </dependency>

  <dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter-eureka</artifactId>

  </dependency>

  <dependency>

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

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

  </dependency>



 </dependencies>
```

Other required source files related to this application you could download from GitHub link as given below

accounts-microservice-server

Now run this account service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server** at **http://localhost:1111/** in previous **Step 1**. Now one Service registered to the Eureka registered instances with Service Name "**ACCOUNT-MICROSERVICE**" as below



System Status

Environment	test
Data center	default

Current time	2017-01-05T03:30:44 +0530
Uptime	00:00
Lease expiration enabled	false
Renews threshold	3
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNTS-MICROSERVICE	n/a (1)	(1)	UP (1) - NODTBSL206AG.timesgroup.com:accounts-microservice:2222

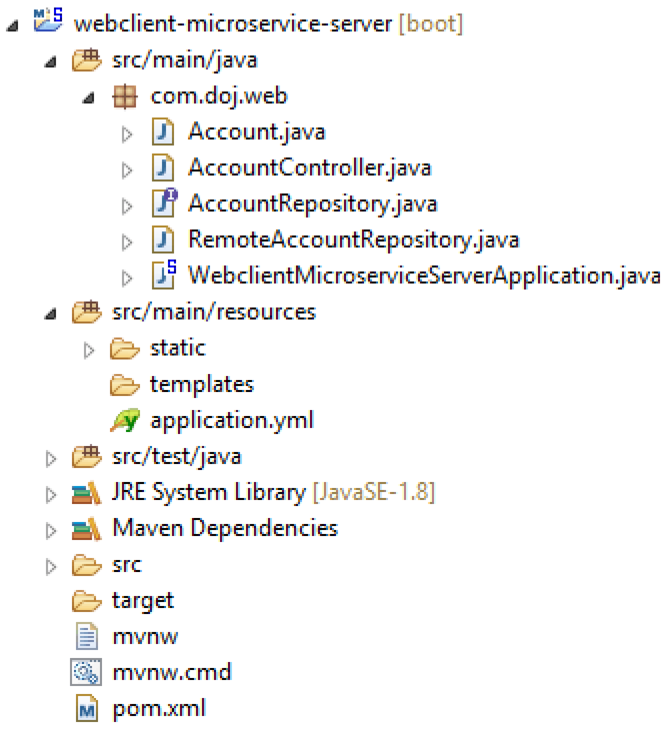
General Info

Name	Value
total-avail-memory	366mb
environment	test

Step 3: Consumer Service

- Create Consumers to find the Producer Service registered with Discovery Service at Step 1.
- ***@EnableDiscoveryClient*** annotation also allows us to query Discovery server to find microservices.

Let's see the consumer application structure as below.



```
webclient-microservice-server [boot]
  src/main/java
    com.doj.web
      Account.java
      AccountController.java
      AccountRepository.java
      RemoteAccountRepository.java
      WebclientMicroserviceServerApplication.java
  src/main/resources
    static
    templates
    application.yml
  src/test/java
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src
  target
  mvnw
  mvnw.cmd
  pom.xml
```

application.yml

```yaml
# Service registers under this name

# Control the InternalResourceViewResolver:

spring:

  application:

    name: accounts-web

  mvc:

    view:

      prefix: /WEB-INF/views/

      suffix: .jsp



# Discovery Server Access

eureka:

  client:

    serviceUrl:

      defaultZone: http://localhost:1111/eureka/



# Disable Spring Boot's "Whitelabel" default error page, so we can use our
own

error:

  whitelabel:

    enabled:  false
```

WebclientMicroserviceServerApplication.java

```
package com.doj.web;



import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.context.annotation.Bean;

import org.springframework.web.client.RestTemplate;



@SpringBootApplication

@EnableDiscoveryClient

public class WebclientMicroserviceServerApplication {



 public static final String ACCOUNTS_SERVICE_URL = "http://ACCOUNTS-
MICROSERVICE";



 public static void main(String[] args) {

  SpringApplication.run(WebclientMicroserviceServerApplication.class, args);

 }



 @Bean

 @LoadBalanced

 public RestTemplate restTemplate() {
  return new RestTemplate();

 }

 @Bean

 public AccountRepository accountRepository(){

  return new RemoteAccountRepository(ACCOUNTS_SERVICE_URL);

 }

}
```

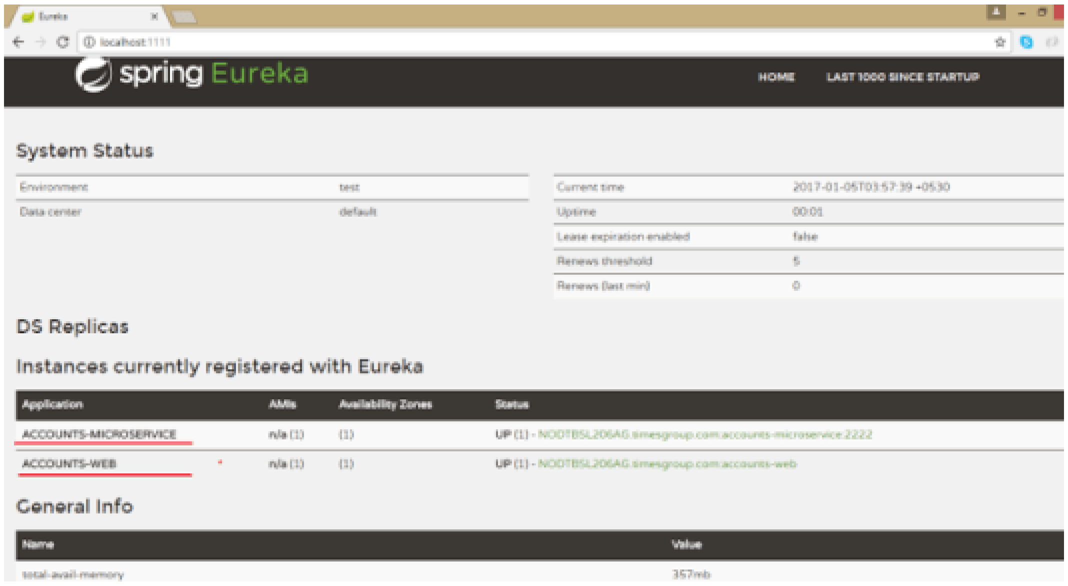
pom.xml

```
<dependencies>

  <dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter-eureka</artifactId>

  </dependency>

  <dependency>

   <groupId>org.springframework.cloud</groupId>

   <artifactId>spring-cloud-starter-ribbon</artifactId>

  </dependency>

  <dependency>

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

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

  </dependency>
  <dependency>

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

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

  </dependency>

  <dependency>

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

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

   <scope>test</scope>

  </dependency>

  <!-- These dependencies enable JSP usage -->

  <dependency>

   <groupId>org.apache.tomcat.embed</groupId>

   <artifactId>tomcat-embed-jasper</artifactId>

   <scope>provided</scope>

  </dependency>

  <dependency>

   <groupId>javax.servlet</groupId>

   <artifactId>jstl</artifactId>

  </dependency>

 </dependencies>
```

Other required source files related to this application you could download from GitHub link as given below

web client-microservice-server

Now run this consumer service application as **Spring Boot application** and after few seconds refresh the browser to the home page of **Eureka Discovery Server** at **http://localhost:1111/** in previous **Step 1**. Now one more Service registered to the Eureka registered instances with Service Name "**ACCOUNTS-WEB**" as below



System Status

Environment	test
Data center	default

Current time	2017-01-05T03:57:39 +0530
Uptime	00:01
Lease expiration enabled	false
Renews threshold	5
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNTS-MICROSERVICE	n/a (1)	(1)	UP (1) - NODTBSL206AG.timesgroup.com:accounts-microservice:2222
ACCOUNTS-WEB	n/a (1)	(1)	UP (1) - NODTBSL206AG.timesgroup.com:accounts-web

General Info

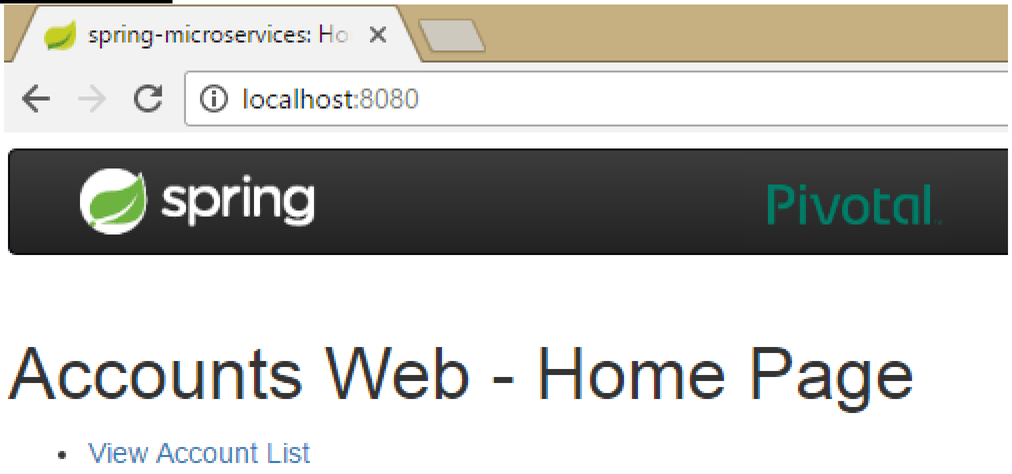
Name	Value
total-avail-memory	357mb

Lets our consumer consume the service of producer registered at discovery server.

```
package com.doj.web;



import java.util.Arrays;

import java.util.List;



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

import org.springframework.web.client.RestTemplate;



/**

 * @author Dinesh.Rajput

 *
 */
public class RemoteAccountRepository implements AccountRepository {


 @Autowired
 protected RestTemplate restTemplate;


 protected String serviceUrl;


 public RemoteAccountRepository(String serviceUrl) {
  this.serviceUrl = serviceUrl.startsWith("http") ? serviceUrl
    : "http://" + serviceUrl;
 }


 @Override
 public List<Account> getAllAccounts() {
  Account[] accounts = restTemplate.getForObject(serviceUrl+"/accounts",
Account[].class);
  return Arrays.asList(accounts);
 }


 @Override
 public Account getAccount(String number) {
  return restTemplate.getForObject(serviceUrl + "/accounts/{id}",
    Account.class, number);
  }


}
```

Let’s open web application which is a consumer of the account microservice registered at Eureka Discovery Server.

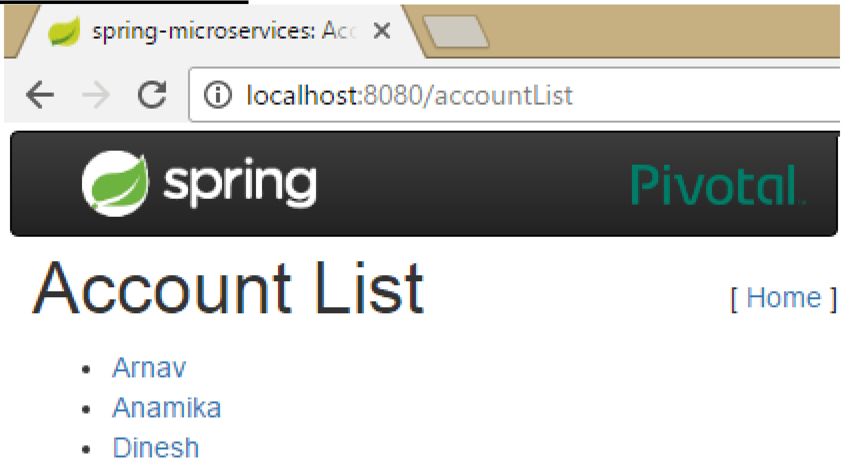
http://localhost:8080/ as below



Accounts Web - Home Page

- View Account List

Now click on **View Account List** then fetch all accounts from account microservice.
http://localhost:8080/accountList

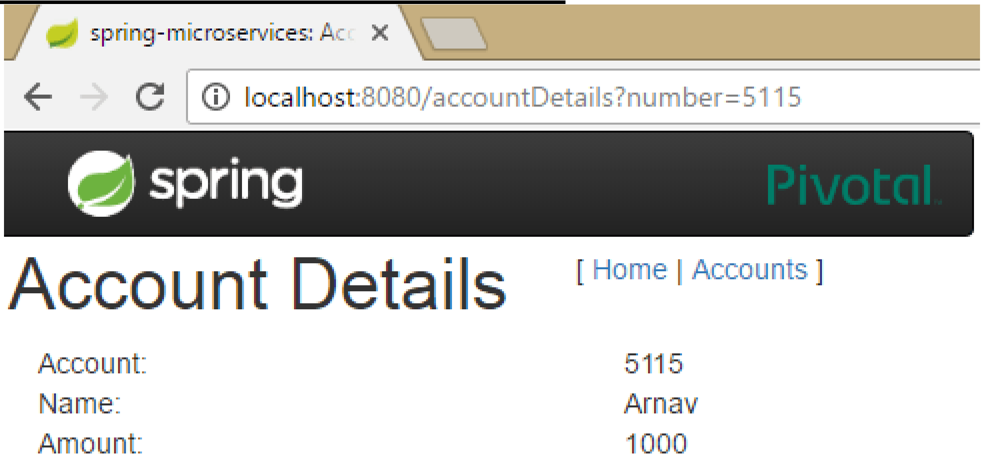


Account List [Home]

- Arnav
- Anamika
- Dinesh

Now click on any account from the list of accounts to fetch the details of the account for account number from account microservice.

http://localhost:8080/accountDetails?number=5115


spring-microservices: Acc
localhost:8080/accountDetails?number=5115
spring
Pivotal

Account Details

[Home | Accounts]

Account:	5115
Name:	Arnav
Amount:	1000

Load Balanced *RestTemplate*
Create using *@LoadBalanced*– Spring enhances it to service lookup & load balancing

```
@Bean

 @LoadBalanced

 public RestTemplate restTemplate() {

  return new RestTemplate();

 }
```

Must inject using the same qualifier-

- If there are multiple ***RestTemplate*** you get the right one.
- It can be used to access multiple microservices

```
@Autowired

        @LoadBalanced

 protected RestTemplate restTemplate;
```

Load Balancing with Ribbon
Our smart RestTemplate automatically integrates two Netflix utilities

- ***Eureka*** Service Discovery

- ***Ribbon*** Client Side Load Balancer

Eureka returns the URL of all available instances
Ribbon determine the best available service too use
Just inject the load balanced ***RestTemplate*** automatic lookup by ***logical service-name***

7. Summary

After completion of this article you should have learned:

- What is the MicroServices Architecture
- Advantages and Challenges of MicroServices
- And some information about Spring Cloud such as Eureka Discover Server by Netflix and Ribbon.

Difference between Microservices Architecture and SOA

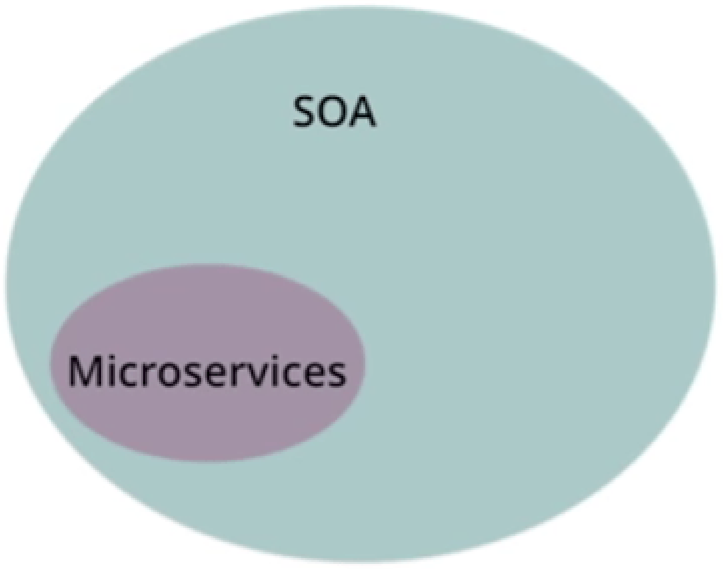
Service-oriented architecture (SOA)

It is an architectural pattern in software design and development according to SOA application components provide services to other components via a communications protocol, typically over a network.

Microservices

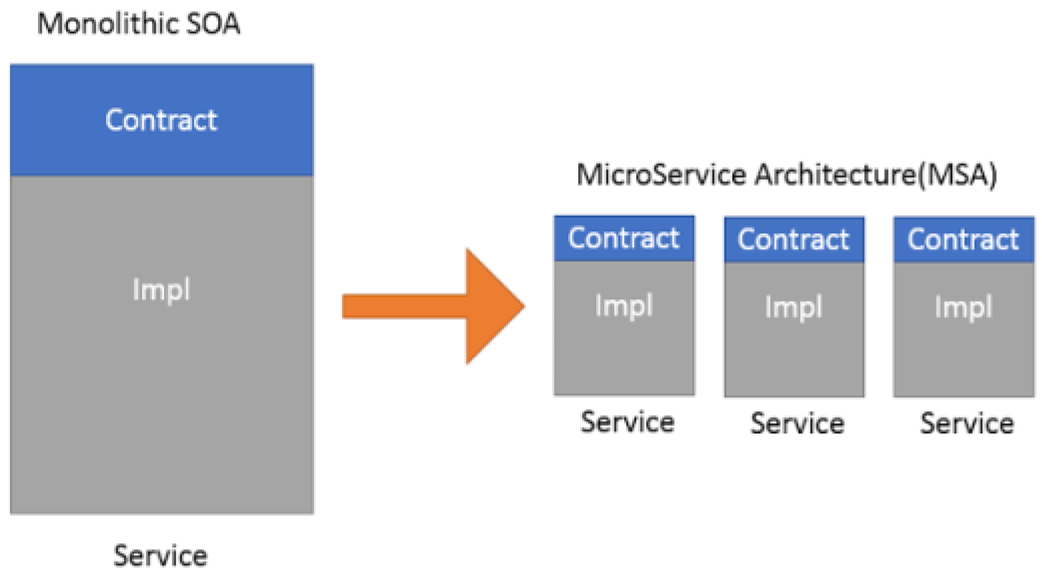
It is a software architecture style in which large and complex software system are divided into independent collaborating processes communicating with each other using language-agnostic APIs.

Microservices has a special approach of breaking a monolithic application into many smaller services that talk to each other, SOA has a broader scope.



@ImageSource-StackOverflow

Microservices are the kind of SOA we have been talking about for the last decade. Microservices must be independently deployable, whereas SOA services are often implemented in deployment monoliths. Classic SOA is more platform driven, so microservices offer more choices in all dimensions.


Monolithic SOA
Contract
Impl
Service
MicroService Architecture(MSA)
Contract
Impl
Service
Contract
Impl
Service
Contract
Impl
Service