SpringBoot

Satya Kaveti

Re-Write SpringBoot

1. Introduction

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

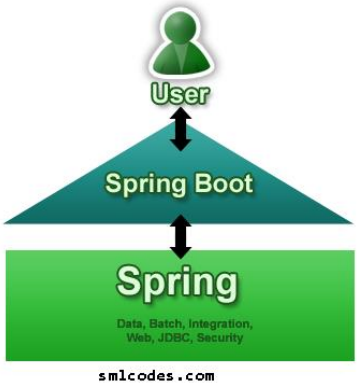
Spring vs SpringBoot

Spring: Spring started as a lightweight alternative to Java Enterprise Edition (J2EE). Spring offered a simpler approach to enterprise Java development, utilizing dependency injection and aspect-oriented programming to achieve the capabilities of EJB with plain old Java objects (POJOs).

But while spring was lightweight in terms of component code**, it was heavyweight in terms of configuration**. Initially, spring was configured with **XML & Spring 2.5** introduced **annotation-based component-scanning, even so, there was no escape from configuration.**

Spring boot: project is just a regular spring project that happens to leverage Spring Boot starters and auto-configuration. Spring Boot is not a framework, it is a way to ease to create **stand-alone application with minimal or zero configurations.**

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



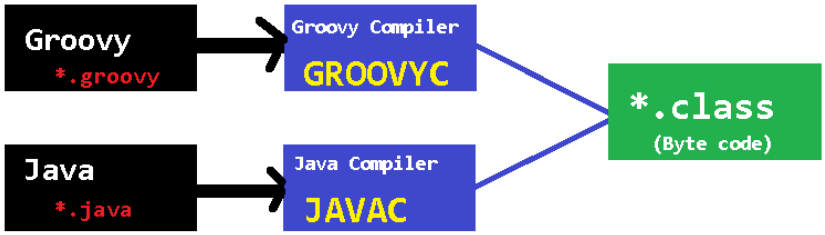
Spring Boot Features

- Create stand-alone Spring applications
- Embed **Tomcat, Jetty or Undertow directly** (no need to deploy WAR files)
- Provide opinionated **'starter' POMs to simplify your Maven configuration**
- **Automatically configure Spring whenever possible**
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely **no code generation and no requirement for XML configuration**

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

1. **Java-Based Applications**
2. **Groovy Application**

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. Spring Boot internally uses some Groovy based techniques and tools to provide default imports and configuration.

Creating Spring Boot Application

To create Spring Boot based applications The Spring Team (The Pivotal Team) has provided the following three approaches.

1. ***Using Maven***
2. ***Using Spring Initializer (http://start.spring.io/)***
3. ***Using Spring STS IDE***
4. ***Using Spring Boot CLI Tool***

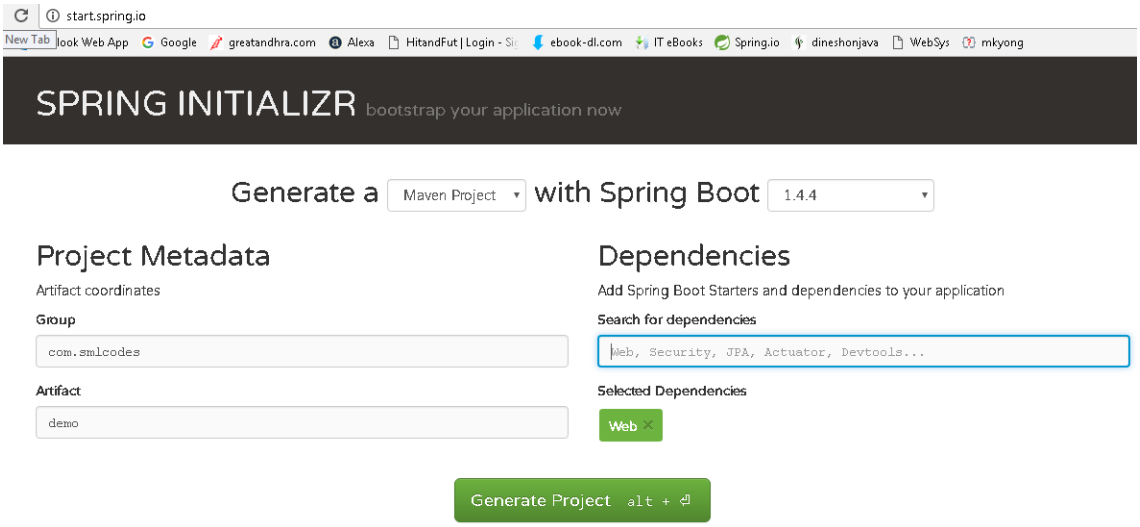
1. Spring Boot using Maven

Add maven dependencies in pom.xml & do maven build

```
<project>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>1.5.6.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>

</project>
```

2.Spring Initializer

Spring Initializer provides an extensible API to **generate quick start projects**. It also provides a configurable service: you can see our default instance at **https://start.spring.io**. It provides a simple web UI to configure the project to generate and endpoints that you can use via plain HTTP.



It will generate the Artifact.zip file, extract it and run maven build: **`mvn clean install package`**

```
[INFO] --- spring-boot-maven-plugin:1.4.4.RELEASE:repackage (default) @ SpringBootDemo ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:27 min
[INFO] Finished at: 2017-01-30T18:55:21+05:30
[INFO] Final Memory: 27M/160M
[INFO] ------------------------------------------------------------------------
```

3.Spring STS IDE

The **Spring Tool Suite** is an Eclipse-based development environment that is customized for developing **spring** applications. We can download it from **here**.

4.Spring Boot CLI Tool

The Spring Boot CLI is a command line tool that can be used if you want to quickly prototype (creates project Structure) 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 don't need to use the CLI to work with Spring Boot but it's definitely the quickest way to get a spring application off the ground.

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

- spring-boot-cli-1.5.0.RELEASE-bin.zip
- spring-boot-cli-1.5.0.RELEASE-bin.tar.gz

SDKMAN! (The Software Development Kit Manager) can be used for managing multiple versions of various binary SDKs, including Groovy and the Spring Boot CLI. Get SDKMAN! from sdkman.io and install Spring Boot with

```
$ sdk install springboot
$ spring --version
Spring Boot v1.5.0.RELEASE
```

A simple web application that you can use to test your installation. Create a file called `app.groovy as`

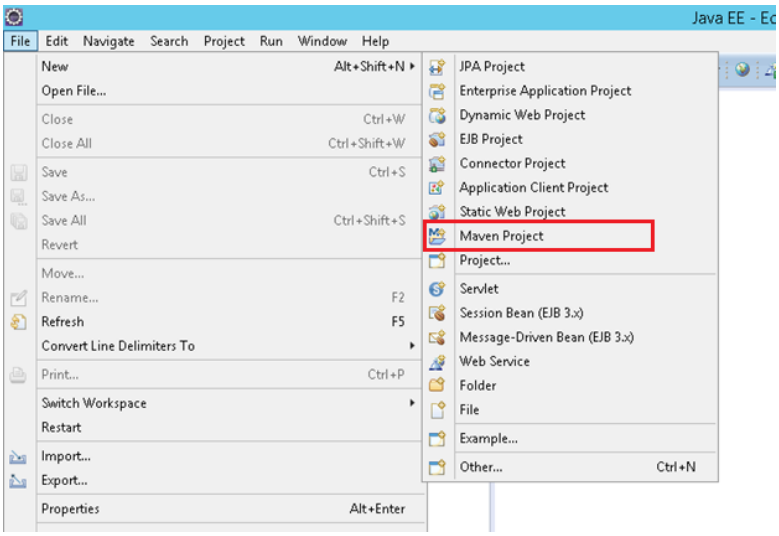
```
$ spring run app.groovy
```

It will take some time when you first run the application as dependencies are downloaded. Subsequent runs will be much quicker.

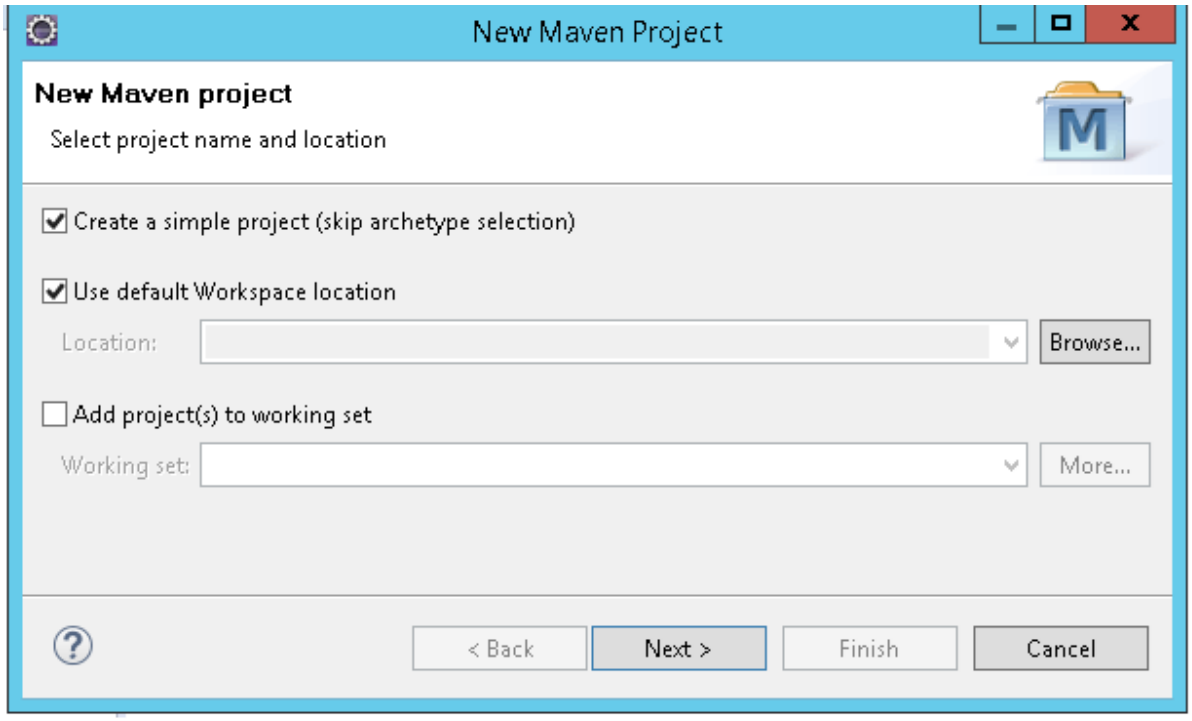
2. Spring Boot Examples

1.Spring Boot with maven and Eclipse Example

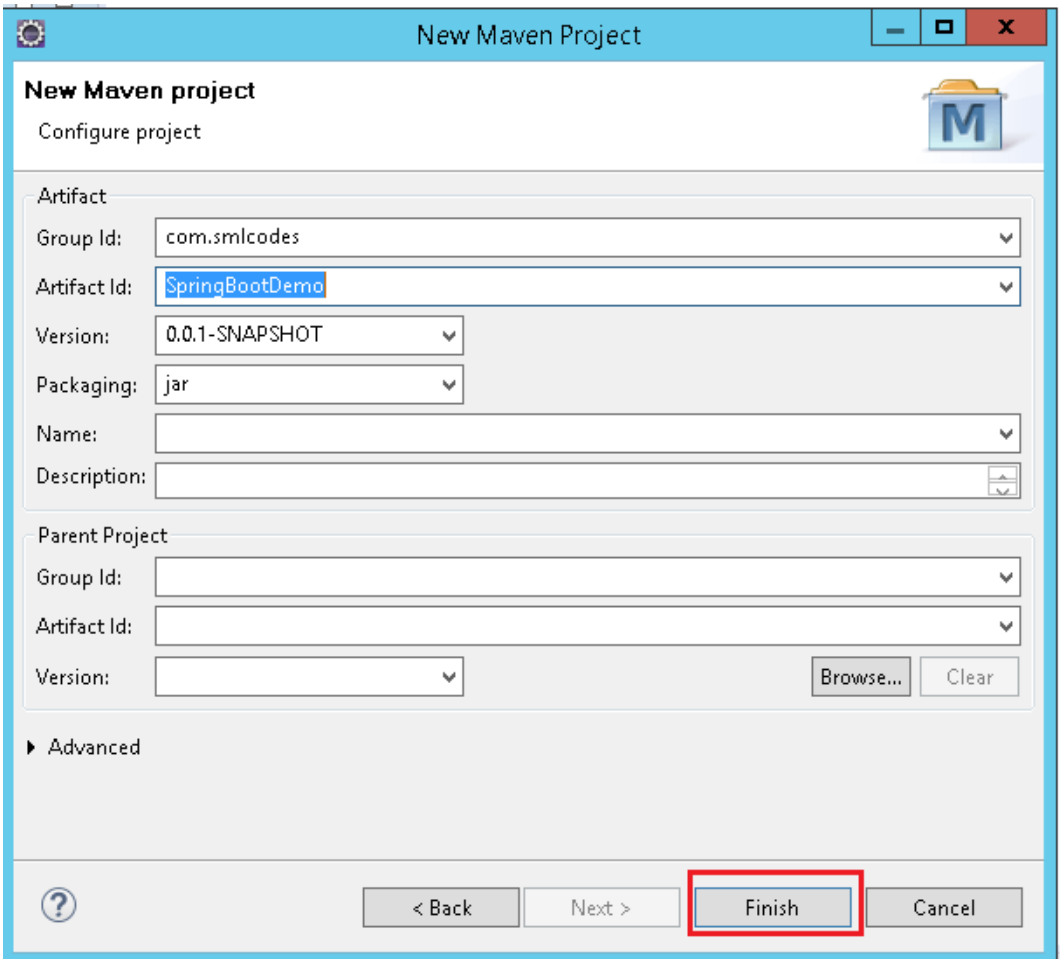
1. Open Eclipse > File > New > Maven Project



2.Tick 'Create a simple project (skip archetype selection) ' check box > click Next



3. Provide Group Id (its your package), Artifact Id (project name) and click Finish



4. open pom.xml, add Spring Boot dependencies

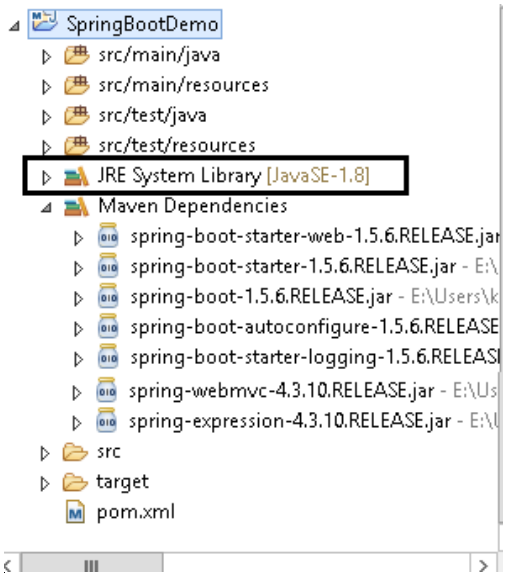
```
<project>
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.smlcodes</groupId>
	<artifactId>SpringBootDemo</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<parent>
		<groupId>org.springframework.boot</groupId>
		<artifactId>spring-boot-starter-parent</artifactId>
		<version>1.5.6.RELEASE</version>
	</parent>

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

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

- `spring-boot-starter-parent`: is an existing project given by spring team which **contains Spring Boot supporting configuration data** (just configuration data, it won't download any jars), we have added this in a **<parent>** tag means, we are instructing Maven to consider our SpringBootHelloWorld project as a child to it

- `spring-boot-starter-web`: Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container

5.Now right click on the application > Maven > Update Project,

if you observe the directory structure of the project, it will create a new folder named "*Maven Dependencies*" which contains all supporting. jars to run the Spring Boot application and the Java version also changed to **1.8.**



- if you observe pom.xml, we haven't included version number for **spring-boot-starter-web** but maven downloaded some jar files with **some version(s) related to spring-boot-starter-web**, that's because of Maven's parent child relation.

- While adding spring boot parent project, we included version as 1.5.6.RELEASE, so again we no need to add version numbers for the dependencies. As we know spring-boot-starter-parent contains configuration meta data, this means, it knows which version of dependency need to be downloaded. So we no need to worry about dependencies versions., it will save lot of our time.

6.create a java class with main() method, in a pakage.com.smlcodes.app.SpringBootApp.java.

```
package com.smlcodes.app;

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

@SpringBootApplication
public class SpringBootApp {
        public static void main(String[] args) {
                SpringApplication.run(SpringBootApp.class, args);
                System.out.println("****\n Hello, World \n ***");
        }
}
```

- **@SpringBootApplication** annotation, is the starting point for our Spring Boot application
- **SpringApplication.*run*(SpringBootApp.class, args);** it will bootstrapping the application

remember, for every spring boot application we have to create a main class and that need to be annotate with @SpringBootApplication and bootstrap it

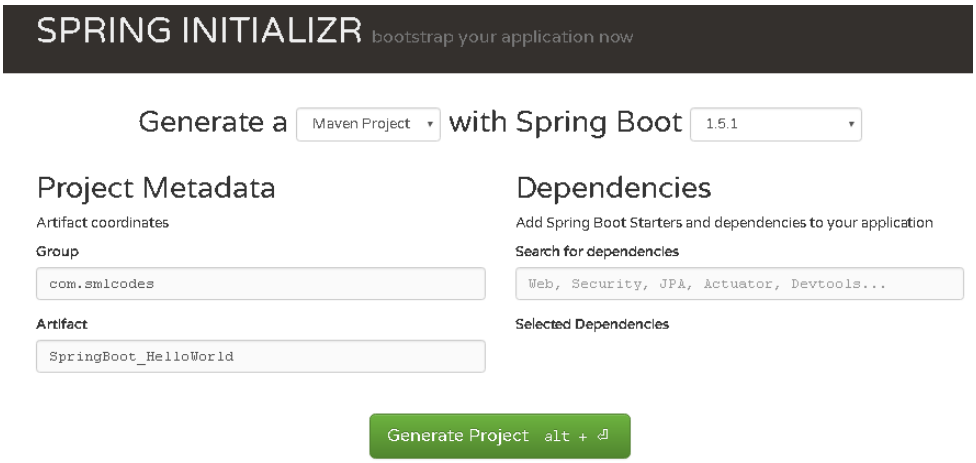
8.Finally, right click on the application > Run As > Java Application

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.6.RELEASE)

2017-09-11 16:04:04.006  INFO 6724 --- [           main] com.smlcodes.app.SpringBootApp           :
Starting SpringBootApp on HYDPCMCSTS with PID 6724
2017-09-11 16:04:13.444  INFO 6724 --- [           main] com.smlcodes.app.SpringBootApp           :
Started SpringBootApp in 10.738 seconds (JVM running for 12.252)
************
 Hello, World
 ****************
```

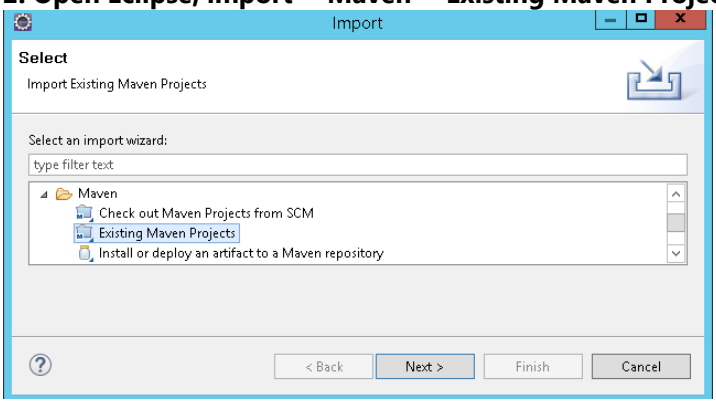
2.Spring Boot with Spring Initializr Example

we are using Spring Initializer(**https://start.spring.io**) to create a template for spring boot application

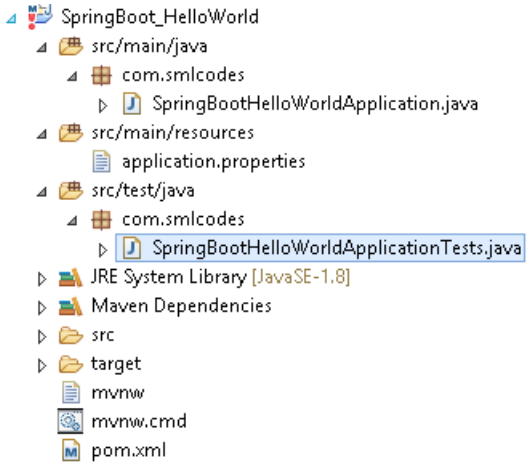
1. Go to https://start.spring.io , Choose Dependencies & Generate Project. Here we are not selecting any dependencies because it is just a Hello world program



2. Open Eclipse, import→ Maven →Existing Maven Projects → Select Project →Finish



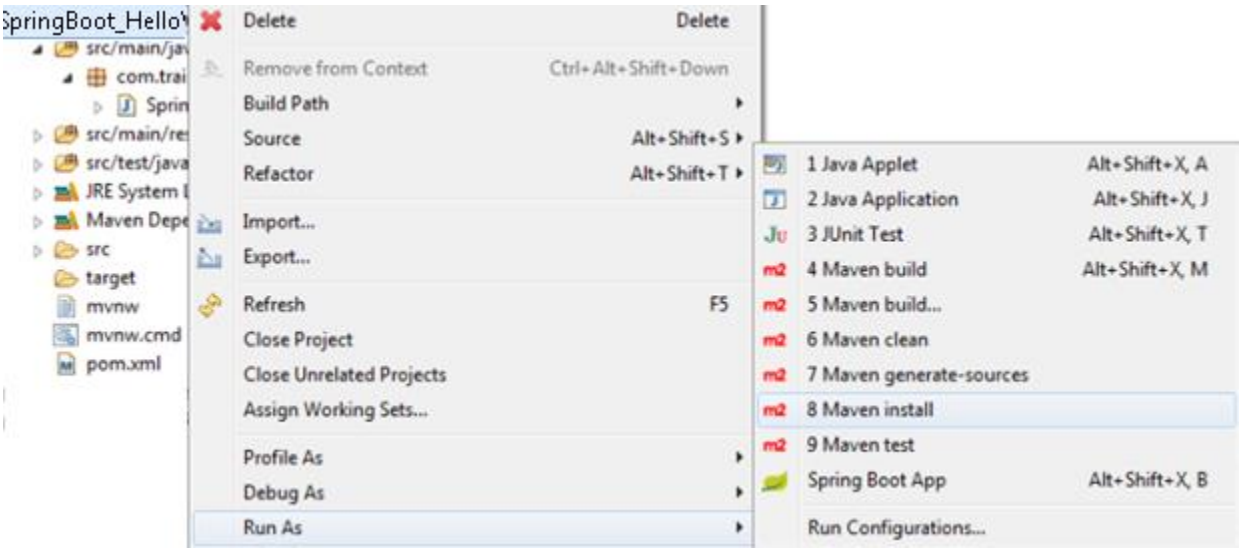
3. The Project structure will be as follows if we open eclipse Package explorer



4. If we open the pom.xml it contains only basic dependencies like `spring-boot-starter` which allows start spring boot application

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

5. Select Project, Right click Run as→ `maven install` to download and install the dependencies.



6.Spring boot generates the default java class which contains main() method. The main method calls the run method of SpringApplication. `SpringApplication.run(SpringBootHelloWorldApplication.class, args);` This run method bootstraps the application starting spring which will run the embedded Tomcat Server. Let's add some helloworld message to print

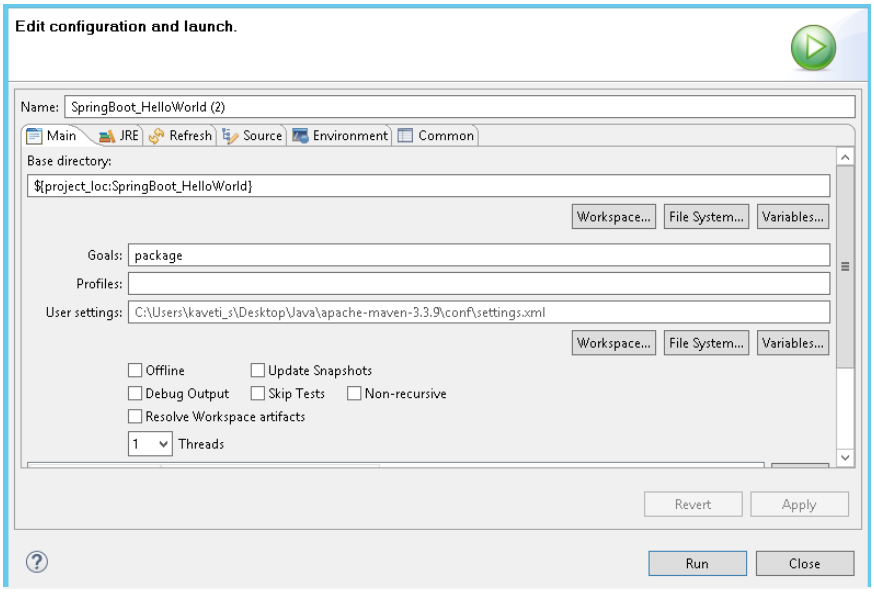
```
package com.smlcodes;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringBootHelloWorldApplication {
	public static void main(String[] args) {
		SpringApplication.run(SpringBootHelloWorldApplication.class, args);
		System.out.println("============================================");
		System.out.println("Hello World, Spring Boot!!!!");
		System.out.println("=============www.smlcodes.com===============");
	}
}
```

7. Select the Java file and right click RunAs → `Java Application`

```
  /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
 ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
```

```
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.1.RELEASE)
2017-01-31 11:05:32.745  INFO 29596 --- [           main] s.c.a.AnnotationConfigApplicationContext :
Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@646007f4: startup
date [Tue Jan 31 11:05:32 IST 2017]; root of context hierarchy
2017-01-31 11:05:43.108  INFO 29596 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        :
Registering beans for JMX exposure on startup
2017-01-31 11:05:43.200  INFO 29596 --- [           main] c.s.SpringBootHelloWorldApplication      :
Started SpringBootHelloWorldApplication in 12.855 seconds (JVM running for 16.094)
==========================================
Hello World, Spring Boot!!!!
=============www.smlcodes.com===============
```

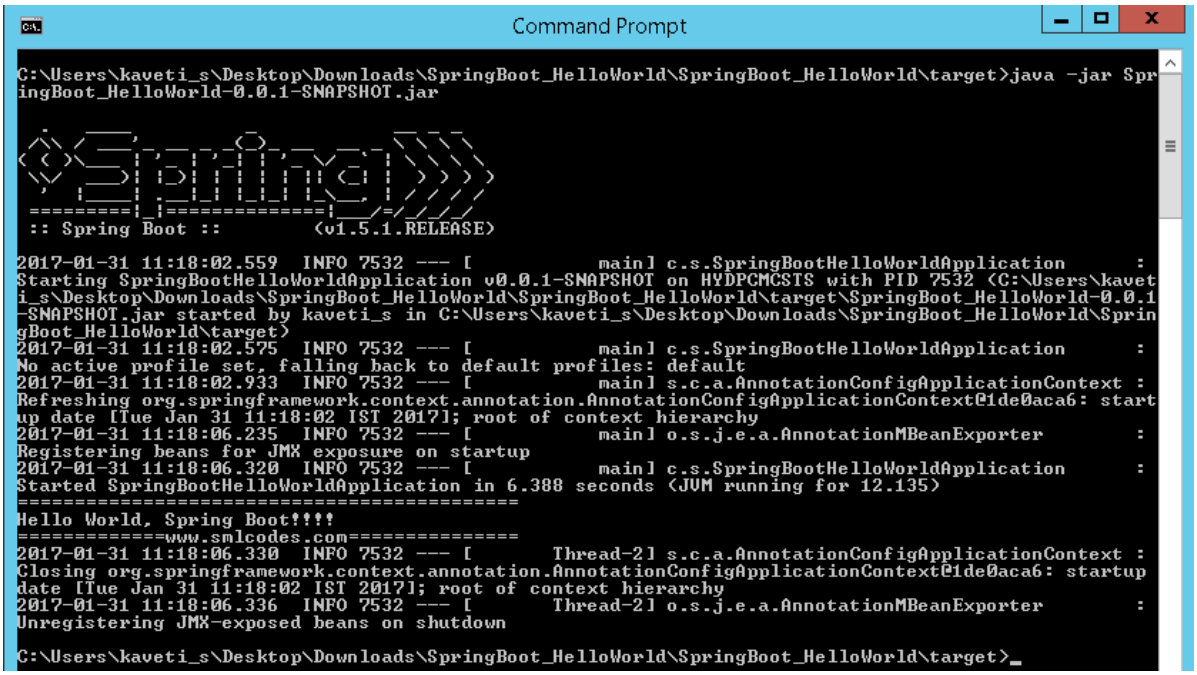
8. We can also run this application from the command line using the **jar file that is generated**. To get the jar file, select **pom.xml** right click **RunAs** → `Maven Build (2nd one)`, **goals=package, Apply & Run**




```
[INFO] Scanning for projects
[INFO] ------------------------------------------------------------------------
[INFO] Building SpringBoot_HelloWorld 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
C:\Users\kaveti_s\Desktop\Downloads\SpringBoot_HelloWorld\SpringBoot_HelloWorld\target\SpringBoot_HelloWor
ld-0.0.1-SNAPSHOT.jar
[INFO] --- spring-boot-maven-plugin:1.5.1.RELEASE:repackage (default) @ SpringBoot_HelloWorld
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

9. Open command line and go to the folder where your project is located. Next, move to the target folder and then type **java -jar <<filename>>.jar**.

```
java -jar SpringBoot_HelloWorld-0.0.1-SNAPSHOT.jar
```



How it works internally

1. we place the all the required **Spring boot starters in pom.xml** which are requires for implementing Spring Boot application. on loading project, the **all required starter dependencies are added automatically to the project**

2. By running Spring Boot main class, at **@SpringBootApplication** line it will do the **Auto configuration** things, it will automatically add all required annotations to Java Class ByteCode.

3.On Executing main() method **SpringApplication.run()** used to bootstrap and launches Spring Boot application.

1.Spring Boot Starters

Spring boot Starters are the one-stop-shop for all the Spring and related technology that we need**. For example**, if you want to get started using **Spring and JPA for database access**, just include the **spring-boot-starter-data-jpa** dependency in your project, and you are good to go.

All **official** starters follow a similar naming pattern; **spring-boot-starter-*,** where * is a particular type of application. The following are the some of the application starters are provided by Spring Boot under the **org.springframework.boot** group

Name	Description
`spring-boot-starter-web-services`	Starter for using Spring Web Services
`spring-boot-starter-web`	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
`spring-boot-starter-test`	Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito
`spring-boot-starter-jdbc`	Starter for using JDBC with the Tomcat JDBC connection pool
`spring-boot-starter-jersey`	Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to `spring-boot-starter-web`
`spring-boot-starter-aop`	Aspect-oriented programming with Spring AOP and AspectJ
`spring-boot-starter-security`	Starter for using Spring Security
`spring-boot-starter-data-jpa`	Starter for using Spring Data JPA with Hibernate
`spring-boot-starter`	Core starter, including auto-configuration support, logging,YML

In above Example we used `spring-boot-starter & spring-boot-starter-test` Starters which are configure in pom.xml

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

2.@SpringBootApplication Annotation

This annotation marks the class as a spring bean, configures the application by adding all the jars based on the dependencies and also scans the other packages for spring beans.

Spring Boot developers always have their main class annotated with ***@Configuration, @EnableAutoConfiguration and @ComponentScan.***

1. **@Configuration** – Specifies this class as a spring bean
2. **@EnableAutoConfiguration** –
    - This tells how you want to configure Spring, based on the jar dependencies that you have added. & also Enable / Disable auto configuration.
    - Spring Boot auto-configuration attempts to automatically configure your Spring application based on the jar dependencies that you have added.

- For example, if HSQLDB is on your classpath, and you have not manually configured any database connection beans, then Spring Boot auto-configures an in-memory database
- We can disble AutoConfiguration specific files by `@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})`

3. **@ComponentScan** – is to scan other packages for spring beans.
4. **`@Import`** - used to import additional configuration classes
5. **`@ImportResource`** -annotation to load XML configuration files

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

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

Since these annotations are so frequently used together Spring Boot provides a convenient **@SpringBootApplication** as an alternative. The @SpringBootApplication annotation **is equivalent to using @Configuration, @EnableAutoConfiguration and @ComponentScan** with their default attributes.

@SpringBootApplication = @Configuration + @ComponentScan + @EnableAutoConfiration.

The original @SpringBootApplication annotation class is defined as below

```
package org.springframework.boot.autoconfigure; @Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication {
        Class<?>[] exclude() default {};

        String[] excludeName()default{};

        @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
        String[] scanBasePackages() default {};

        @AliasFor(annotation = ComponentScan.class, attribute = "basePackageClasses")
        Class<?>[] scanBasePackageClasses()default{};
}
```

3 SpringApplication Class

SpringApplication class is used to bootstrap and launch a Spring application from a Java main method. By default, class will perform the following steps to bootstrap your application:

- Create an appropriate ApplicationContext instance (depending on your classpath)
- Register a CommandLinePropertySource to expose command line arguments as Spring properties

- Refresh the application context, loading all singleton beans
- Trigger any CommandLineRunner beans

In most circumstances the static **run(Object, String[])** method can be called directly from your main method to bootstrap your application:

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

The following embedded servlet containers are supported out of the box.By default we will get Tomcat

Name	Servlet Version	Java Version
Tomcat 8	3.1	Java 7+
Tomcat 7	3.0	Java 6+
Jetty 9.3	3.1	Java 8+
Jetty 9.2	3.1	Java 7+
Jetty 8	3.0	Java 6+
Undertow 1.3	3.1	Java 7+

Spring Profiles provide a way to segregate parts of your application configuration and make it only available in certain environments. Any **@Component or @Configuration can** be marked with **@Profile** to limit when it is loaded

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```

In the normal Spring way, you can use a **spring.profiles.active** Environment property to specify which profiles are active. You can specify the property in any of the usual ways, for example you could include it in your **application.properties**:**spring.profiles.active=dev,hsqldb** or specify on the command line using the **switch --spring.profiles.active=dev,hsqldb.**

6. Spring Boot Actuator

Spring Boot provides actuator to monitor and manage our application. Actuator is a tool which has HTTP endpoints. **when application is pushed to production, you can choose to manage and monitor your application using HTTP endpoints.**

To get production-ready features, we should use spring-boot-actuator module. We can enable this feature by adding it to the **pom.xml** file.

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

3. Spring Boot –Configurations

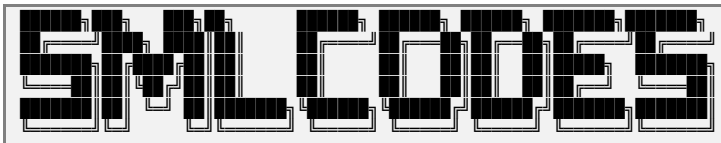
How to change Spring Boot Banner Text

The banner that is printed on startup can be changed by adding a **banner.txt** file to **src\main\resources** folder or your classpath, or by setting **banner.location** to the location of such a file.

You can also add a **banner.gif, banner.jpg or banner.png image file to your classpath**, or set a **banner.image.location property**. Images will be converted into an ASCII art representation and printed above any text banner.

```
//Default Banner
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.1.RELEASE)
2017-02-01 14:07:22.957  INFO 72716 --- [           main] c.s.SpringBootHelloWorldApplication
```

1. Go to any ANCII Text generator website & generate your logo. for ex: **http://patorjk.com/**
2. Create banner.txt under **Proj_Home\src\main\resources, paste the logo text**
3. **Refresh** the project & run Spring Boot Application.the banner will change as below



```
2017-02-01 14:07:22.957  INFO 72716 --- [           main] c.s.SpringBootHelloWorldApplication
```

application.properties

Spring Boot provides a very neat way to load properties for an application. we can define properties in **application.properties** (`PROJ_HOME\src\main\resources\application.properties`) file the following way

```
db.name=smlcodesdb
db.username=smlcodes
db.password=wEB20R1XPJtg9
```

In traditional Spring application would have loaded up the properties in the following way

```
public class SmlcodesPropTest {
    @Value("${db.name}") //dbname is KEY here
    private String dbname;

    @Value("${db. username }")
    private String server_port;
```

In Spring boot it takes application. properties file to define a bean that can hold all the related properties in following way

```
@ConfigurationProperties(prefix = "db")
@Component
public class DBConfig {

        public String dbname;
        public String username;
        public String password;

        public String getDbname() {
                return dbname;
        }
        public void setDbname(String dbname) {
                this.dbname = dbname;
        }
        public String getUsername() {
                return username;
        }
        public void setUsername(String username) {
                this.username = username;
        }
        public String getPassword() {
                return password;
        }
        public void setPassword(String password) {
                this.password = password;
        }

}
```

@ConfigurationProperties is used to bind and validate some external Properties. If we want to validate before going to use, we have to place **@Validated** & place type of validation on the filed

```
@ConfigurationProperties(prefix="foo")
@Validated
public class FooProperties {

    @NotNull
    private InetAddress remoteAddress;
    // ... getters and setters
}
```

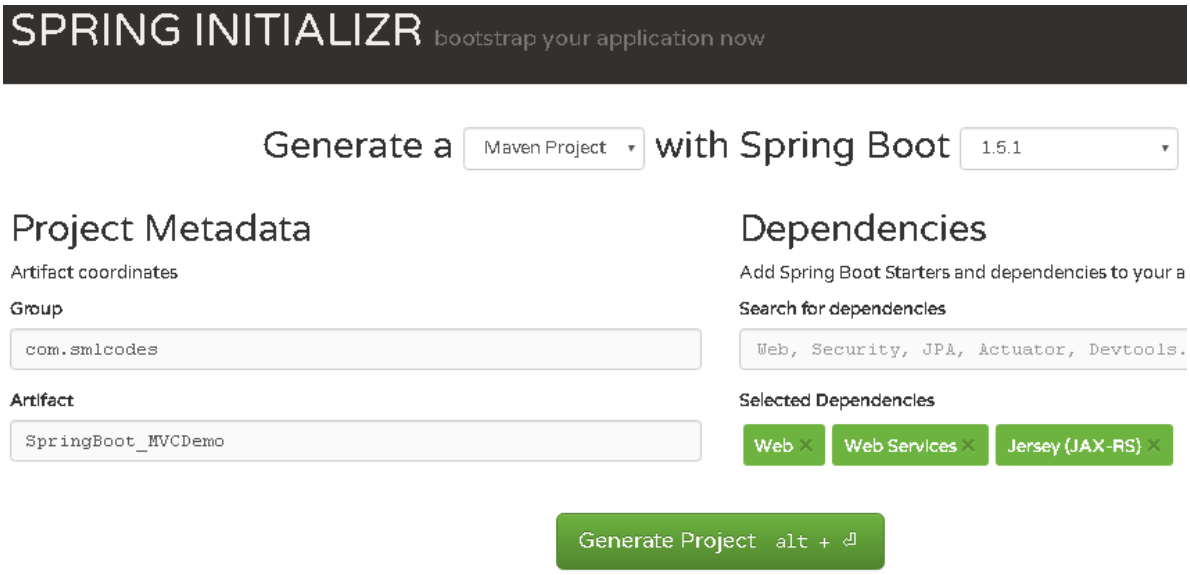
4. Spring Boot –MVC

In old 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.

SpringMVC example **@RestController** to serve JSON data

```
@RestController
@RequestMapping(value="/users")
public class SmlCodesRestController {

    @RequestMapping(value="/{user}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}/customers", method=RequestMethod.GET)
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}", method=RequestMethod.DELETE)
    public User deleteUser(@PathVariable Long user) {
        // ...
    }

}
```

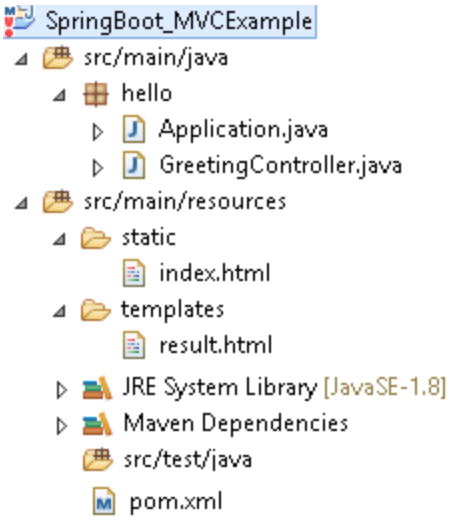
Spring Boot MVC Example



Extract, import to eclipse as Existing Maven project, & **Run as→** `maven install`

If we see the Folder Structre

- **all index, welcome files must** be placed unter →**resources\static**
- **all the result pages** must be placed under→**resources\templates**



1.Choose the SpringBoot Stater Dependencies and place in pom.xml & build the 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>SpringBoot_MVCExample</groupId>
    <artifactId>gs-serving-web-content</artifactId>
    <version>SpringBoot_MVCExample</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>

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

2.Create index.html to provide user input

```
///SpringBoot_MVCExample/src/main/resources/static/index.html
<h2>Spring Boot MVC Example</h2>
<form method="get" action="/hello">
        <h3>Enter Your name : <input type="text" name="name"></h3>
        <button type="submit">Enter</button>
```

```
</form>
```

3.Create Controller to hadle request given by user (/hello)

```
package hello;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class GreetingController {

    @RequestMapping("/hello")
    public String greeting(@RequestParam(value="name", required=false, defaultValue="World") String name,
Model model) {
        model.addAttribute("name", name);
        return "result";
    }
}
```

Here Model is a interface which conatins some usefull method to return result data to result page

4.Create result.html template to display the Results given by Controller

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<body>
    <h1><p th:text="'Hello, ' + ${name} + '!'" /></h1>
</body>
</html>
```

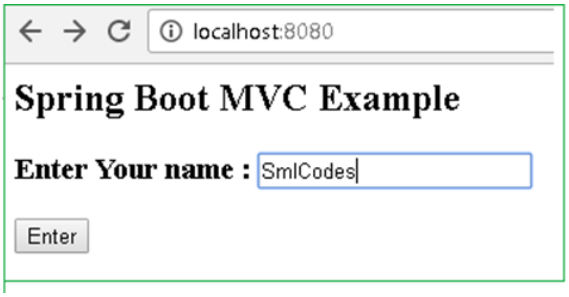
5.Create Application.java to Start & Run Spring Boot Application

```
package hello;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

6.Strat the Application by Run as→ `Java Application (Application.java)`

```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.1.RELEASE)
2017-02-02TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080
```

7.Open browser, access `localhost:8080` the Output should be as below



Explanation

1.On Running Applicatio.java, Spring Boot Engine Starts and reads the all files in the projects and autowires the data and auto configures the Controller details

2.on submitting the form, Spring Boot Searchers for controller classes which are annotated with **@Controller**

3.compairs **("/hello")** path with **controller @RequestMapping("/hello"),** if matches execute the business logic method and it returns the resultpage name("**result**")

4.SpringBoot Engine Searches the appropriate resultpage template having "**result**" as page name & displays the **result.html** page to the user

1.Model Interface

Model interface designed for adding attributes to the model. Allows for accessing the overall model as **a java.util.Map.**

Method Summary	
Model	**addAllAttributes**(Collection<?> attributeValues) Copy all attributes in the supplied Collection into this Map,
Model	**addAllAttributes**(Map<String,?> attributes) Copy all attributes in the supplied Map into this Map.
Model	**addAttribute**(Object attributeValue) Add the supplied attribute to this Map using a generated name.
Model	**addAttribute**(String attributeName, Object attributeValue) Add the supplied attribute under the supplied name.
Map<String,Object>	**asMap**() - Return the current set of model attributes as a Map.
boolean	**containsAttribute**(String attributeName) Does this model contain an attribute of the given name?
Model	**mergeAttributes(Map<String,?> attributes)**

2.Static Content

By default, Spring Boot will serve static content from a directory called **/static** (`or /public or /resources or /META-INF/resources`)

You can also customize the static resource locations using **spring.resources.static-locations** (replacing the default values with a list of directory locations).

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.**

Spring Boot –RESTful Web Service Example

To work with webservices in SpringBoot we have to use two annotations

- **@RestController**: tells Spring Boot to consider this class as REST controller
- **@RequestMapping**: used to register paths inside it to respond to the HTTP requests.

The @RestController is a stereotype annotation. It adds @Controller and @ResponseBody annotations to the class.

@RestController = @Controller + @ResponseBody

Note - The @RestController and @RequestMapping annotations are Spring MVC annotations. They are not specific to Spring Boot.

app.controller.SpringBootRestController.java

```
package app.controller;

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

@RestController
public class SpringBootRestController {

	@RequestMapping("/")
	public String welcome() {
		return "Spring Boot Home Page";
	}

	@RequestMapping("/hello")
	public String myData() {
		return "Smalcodes : Hello Spring Boot";
	}
}
```

app.SpringBootApp.java

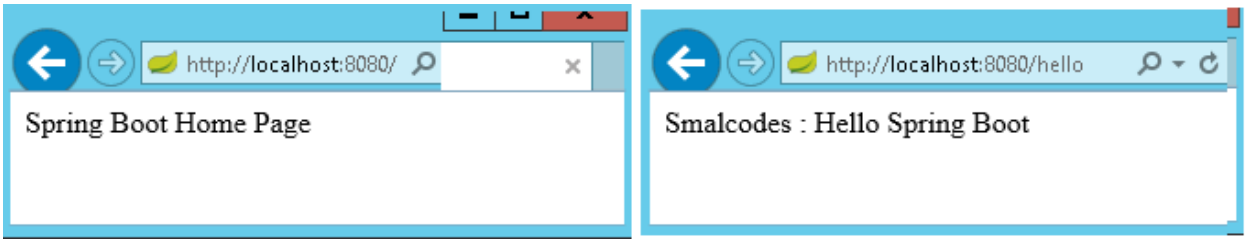
```
package app;
```

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

@SpringBootApplication
public class SpringBootApp {
	public static void main(String[] args) {

		SpringApplication.run(SpringBootApp.class, args);

	}
}
```

Create pom.xml same as fisrt example.

Test the Application

Right click on project > Run as >Java Application >**select SpringBootApp**



- In above Spring Boot main application class in ***app*** package and controller class **in *app.controller***. While starting our application, SpringBootApp class will scan all the components under that package. As we have created our controller class in **app.controller** which is inside app package, our controller was registered by spring boot.

- If you create the controller class outside of the main package, lets say com.smlcodes.*controller,* If you run the application it gives 404 error.To resolve this, we have to add ***@ComponentScan*** annotation in our Spring Boot main class, as below

```
@SpringBootApplication
@ComponentScan(basePackages="smlcodes.controller")
public class SpringBootApp {
	public static void main(String[] args) {
		SpringApplication.run(SpringBootApp.class, args);
	}
}
```

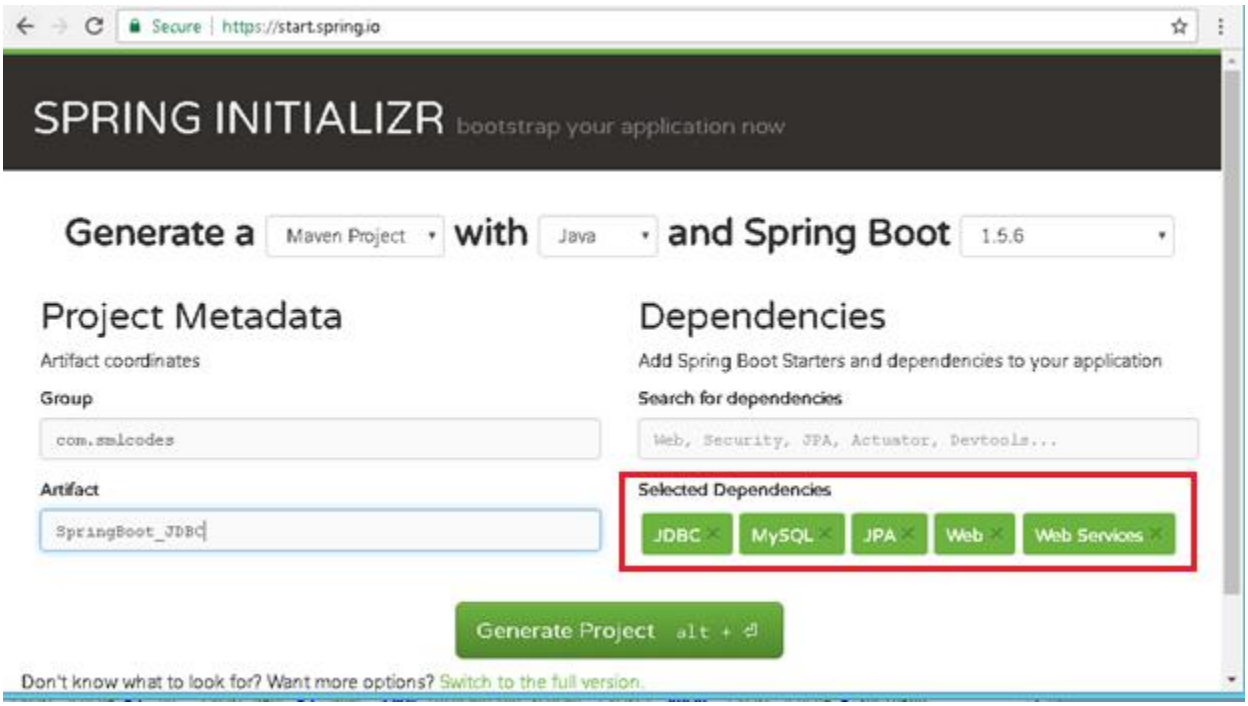
5. Spring Boot –Database

Spring Boot –JDBC Example

Spring Boot provides starter and libraries for connecting to our application with JDBC. Spring JDBC dependencies can be resolved by using either `spring-boot-starter-jdbc` or `spring-boot-starter-data-jpa` spring boot starters.

1.Create project Structure

To create project go to https://start.spring.io/ and add JDBC,MySQL,JPA dependencies to the Project.



Configure DataSource (application. properties)

DataSource and Connection Pool are configured in `application.properties` file using prefix `spring.datasource`. Spring boot uses `javax.sql.DataSource` interface to configure DataSource

```
spring.datasource.url=jdbc:mysql://localhost:3306/springdb?useSSL=false
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
```

Model Class(model.Student.java)

Find the MySQL table used in our example.

```
CREATE TABLE `student` (
`sno` INT(11) NOT NULL,
`name` VARCHAR(50) NULL DEFAULT NULL,
`address` VARCHAR(50) NULL DEFAULT NULL,
PRIMARY KEY (`sno`)
)
COLLATE='latin1_swedish_ci'
ENGINE=InnoDB
;
```

Create Student class with table properties

```
package app.model;

public class Student {
	private int sno;
	private String name;
	private String address;

	public Student() {
		super();
	}

	public Student(int sno, String name, String address) {
		super();
		this.sno = sno;
		this.name = name;
		this.address = address;
	}

	public int getSno() {
		return sno;
	}

	public void setSno(int sno) {
		this.sno = sno;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getAddress() {
		return address;
	}

	public void setAddress(String address) {
		this.address = address;
	}

}
```

DAO Class with JdbcTemplate (StudentDAO.java)

- `JdbcTemplate` is the central class to handle JDBC. It executes SQL queries and fetches their results. To use `JdbcTemplate`.
- `JdbcTemplate` dependency injection using `@Autowired` with constructor.

```
package app.dao;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;
import app.model.Student;

@Repository
public class StudentDAO {

	@Autowired
	private JdbcTemplate template;

	public List<Student> findAll() {

	List<Student> result = template.query("SELECT sno,name, address FROM Student", new
StudentRowMapper());
		return result;
	}

	public void addStudent(int sno, String name, String address) {
template.update("INSERT INTO Student(sno,name, address) VALUES (?,?,?)", sno, name, address);
	}

}
```

RowMapper Class

Spring JDBC provides `RowMapper` interface that is used to map row with a java object. We need to create our own class implementing `RowMapper` interface to map row with java object. Find the sample code to implement `RowMapper`interface.

It is a Functional interface we have only one method `mapRow(ResultSet rs, int rowno)`

```
package app.dao;

import org.springframework.jdbc.core.RowMapper;
import app.model.Student;

public class StudentRowMapper implements RowMapper<Student> {
	@Override
	public Student mapRow(ResultSet rs, int rowno) throws SQLException {
		// TODO Auto-generated method stub
		Student s = new Student();
		s.setSno(rs.getInt("sno"));
		s.setName(rs.getString("name"));
		s.setAddress(rs.getString("address"));
		return s;
	}
}
```

SpringBootJdbcController.java

```
package app.controller;
import org.springframework.web.bind.annotation.RestController;
import app.dao.StudentDAO;
import app.model.Student;
import java.util.Iterator;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@RestController
public class SpringBootJDBCController {

	@Autowired
	private StudentDAO dao;

	@RequestMapping("/jdbc")
	public String welcome() {
		return "Spring Boot Home Page";
	}

	@RequestMapping("/insert")
	public String insert(@RequestParam("sno") int sno, @RequestParam("name") String name,
			@RequestParam("address") String adr) {
		System.out.println(" ************** Inside Method ************");

		dao.addStudent(sno, name, adr);
		return "Data Inserted";
	}

	@RequestMapping("/select")
	public String select() {
		String result="";
		List<Student> list = dao.findAll();
		Iterator<Student> itr = list.iterator();
		while (itr.hasNext()) {
			Student s = (Student) itr.next();
			result = result+ s.getSno()+", ";
			result = result+ s.getName()+", ";
			result = result+ s.getAddress()+" <br>";
		}
		System.out.println("Result : "+result);
		return result;
	}
}
```

SpringBootApp.java

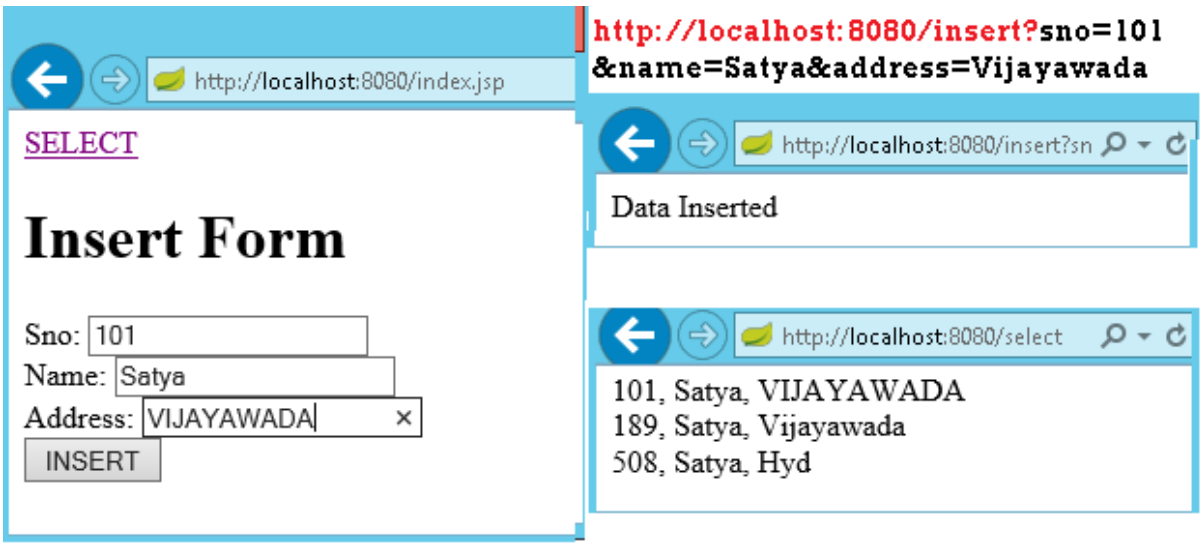
```
package app;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

Static/index.jsp

```
<a href="/select">SELECT</a><br />
```

```
<h1>Insert Form</h1>
<form action="/insert">
        Sno: <input name="sno" type="text" /> <br>
        Name: <input name="name" type="text" /> <br>
        Address: <input name="address" type="text" /> <br>
        <input type="submit" value="INSERT" /> <br>
</form>
</body>
</html>
```

Rightclikc on Project> Runas> Java Application



We can discard RowMapper class if we write following code in StudentDAO class it self.

```
@Repository
public class StudentDAO {

        @Autowired
        private JdbcTemplate template;

        public List<Student> findAll() {

                List<Student> result = template.query("SELECT sno,name, address FROM Student",
                                (rs, rowNum) -> new Student(rs.getInt("sno"),
                              rs.getString("name"), rs.getString("address")));
                return result;

        }

        public void addStudent(int sno, String name, String address) {
                template.update("INSERT INTO Student(sno,name, address) VALUES (?,?,?)",
               sno, name, address);
        }
}
```

Spring Boot –JPA Example

Spring Boot provides **spring-boot-starter-data-jpa** starter to connect Spring application with relational database efficiently. You can use it into project POM (Project Object Model) file.

JPA Annotations

By default, each field is mapped to a column with the name of the field. You can change the default name via **@Column (name="newColumnName").**

The following annotations can be used.

@Entity	Marks java class to a Table name
@Table(name="tabname")	Provides table name, when table name & class names are different .
@Id	**Identifies the unique ID of the database entry**
@GeneratedValue	Together with an ID this annotation defines that value is generated automatically.
@Transient	Field will not be saved in database

The central interface in Spring Data repository abstraction is **Repository** (probably not that much of a surprise). It takes the domain class to manage as well as the id type of the domain class as type arguments.

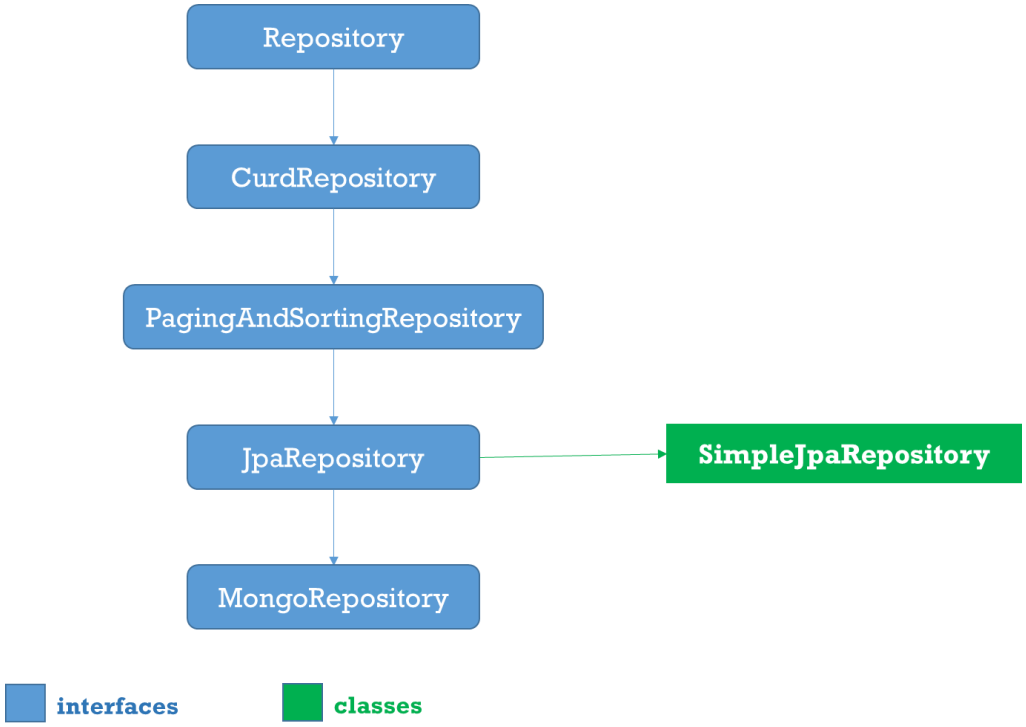
Spring 4 Data

Spring Data Commons provides all the common abstractions that enable you to connect with different data stores.

Spring Data Coomons provides classes & methods, which are common for all the SQL, NoSQL, BigData databases

The Spring Data Commons project provides general infrastructure and interfaces for the other, more specific data projects. Regardless of the type of datastore, Spring Data supports the following aspects with a single API:

- Execute CRUD (create, read, update, delete) operations
- Sorting
- Page-wise reading (pagination)



1.Repositoy

Root interface for all Repositoty classes. It is a **marker interface(no methods)**

2.CurdRepositoy

It provides generic **CRUD** operations irrespective of the underlying database. It extends **Repository** interface.

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {
        save(S entity);
        saveAll(Iterable<S> entities);

        Optional<T> findById(ID id);
        Iterable<T> findAll();
        Iterable<T> findAllById(Iterable<ID> ids);

        void deleteById(ID id);
        void delete(T entity);
        void deleteAll(Iterable<? extends T> entities);
        void deleteAll();

        boolean existsById(ID id);
        long count();
}
```

3.PagingAndSortingRepository

PagingAndSortingRepository provides options to

- **Sort** your data using **Sort interface**
- **Paginate** your data using **Pageable interface**, which provides methods for pagination - getPageNumber(), getPageSize(), next(), previousOrFirst() etc

```
public abstract interface PagingAndSortingRepository extends CrudRepository {

    public Iterable findAll(Sort sort);
    public Page findAll(Pageable pageable);

}
```

4.JpaRepository

JPA specific extension of Repository

```
public interface JpaRepository<T, ID extends Serializable> extends
PagingAndSortingRepository<T, ID> {
  List<T> findAll();
  List<T> findAll(Sort sort);
  List<T> save(Iterable<? extends T> entities);
  void flush();
  T saveAndFlush(T entity);
  void deleteInBatch(Iterable<T> entities);
}
```

5.MongoRepository

Mongo specific Repository interface.

```
public interface MongoRepository<T, ID> extends PagingAndSortingRepository {
List<T> findAll()
List<T> findAll(Sort sort)

List<S> saveAll(Iterable<S> entities)
List<S> insert(Iterable<S> entities)
S       insert(S entity)

}
```

6.Custom Repository

- You can create a custom repository extending any of the repository classes - Repository, PagingAndSortingRepository or CrudRepository. For example,

```
interface PersonRepository extends CrudRepository<User, Long> {
}
```

- Spring Data also provides the feature of query creation from interface method names.

```
interface PersonRepository extends Repository<User, Long> {
```

```
  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

  // Enables the distinct flag for the query
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling ignoring case for all suitable properties
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

7.Defining Query Methods

The repository proxy has two ways to derive a store-specific query from the method name:

- By deriving the query from the method name directly.

```
List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);
```

- By using a manually defined query.

```
@Query("select u from User u")
List<User> findAllByCustomQueryAndStream();
```

- Limiting the result size of a query with `Top` and `First`

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

1.Entity class : Student.java

1. create an entity class that contains the information of a single Student entry

```
package app.entity;

import javax.persistence.Column;
```

```java
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "student")
public class Student {

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int sno;

	@Column(name = "name")
	private String name;

	@Column(name = "address")
	private String address;

	public Student() {
		super();
	}

	public Student(int sno, String name, String address) {
		super();
		this.sno = sno;
		this.name = name;
		this.address = address;
	}

	//Setters & getters
}
```

StudentRepository.java

We can create the repository that provides CRUD operations for ***Student*** objects by using one of the following methods:

1. Create an interface that extends the ***CrudRepository*** interface.
2. Create an interface that extends the ***Repository*** interface and add the required methods to the created interface.

```java
package app.repository;
import org.springframework.data.repository.CrudRepository;
import app.entity.Student;

public interface StudentRepository extends CrudRepository<Student, String>{

}
```

StudentService.java

```java
package app.service;

@Service
public class StudentService {

	@Autowired
	private StudentRepository repository;

	public List<Student> getAllStudents() {
```

```
                List<Student> studentRecords = new ArrayList<>();
                repository.findAll().forEach(studentRecords::add);
                return studentRecords;
        }

        public Student getStudent(String id) {
                return repository.findOne(id);
        }

        public void addStudent(Student studentRecord) {
                repository.save(studentRecord);
        }

        public void delete(String id) {
                repository.delete(id);
        }
}
```

StudentController.java

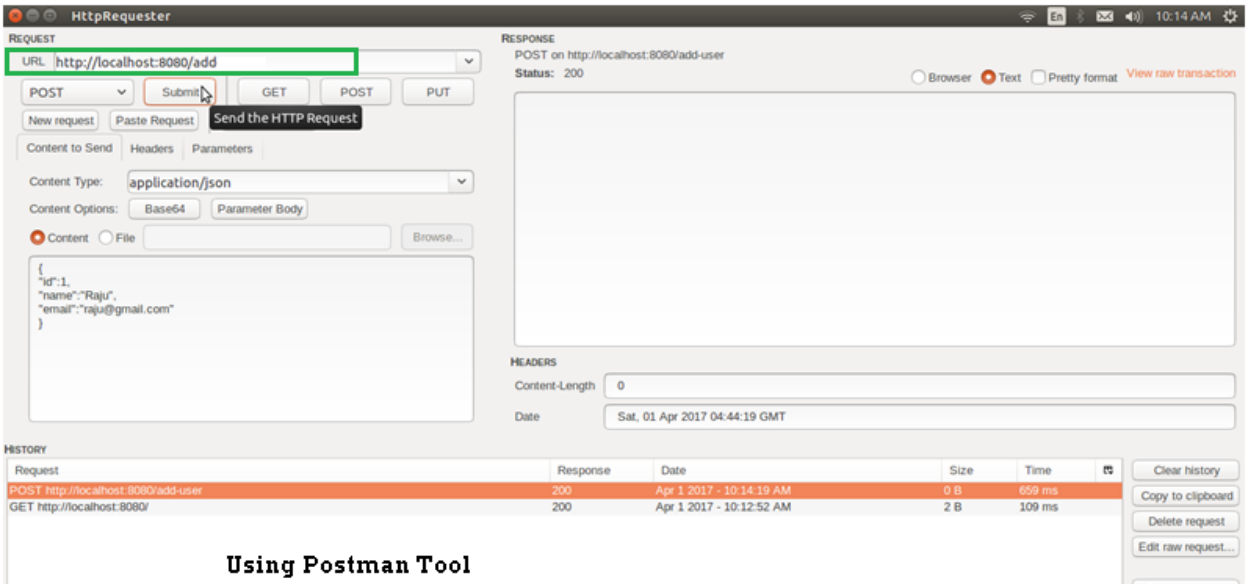
```
package app.controller;

@RestController
public class StudentController {
        @Autowired
        private StudentService studentService;

        @RequestMapping("/")
        public List<Student> getAllStudent() {
                return studentService.getAllStudents();
        }

        @RequestMapping(value = "/add", method = RequestMethod.POST)
        public void addStudent(@RequestBody Student student) {
                studentService.addStudent(student);
        }

        @RequestMapping(value = "/get/{id}", method = RequestMethod.GET)
        public Student getStudent(@PathVariable String id) {
                return studentService.getStudent(id);
        }
}
```

SpringBootApp.java

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

http://localhost:8080/ -get All Srudents

```
[
        {
                "sno": 189,
                "name": "Satya",
                "address": "Vijayawada"
        },
        {
                "sno": 508,
                "name": "Satya",
                "address": "Hyd"
        }
]
```



JpaRepository

JpaRepository provides some JPA related method such as flushing the persistence context and delete record in a batch. JpaRepository extends PagingAndSortingRepository which in turn extends CrudRepository.

Their main functions are:

- **CrudRepository** mainly provides CRUD functions.
- **PagingAndSortingRepository** provide methods to do pagination and sorting records.
- **JpaRepository** provides some JPA related method such as flushing the persistence context and delete record in a batch.

Because of the inheritance mentioned above, **JpaRepository** will have all the functions of **CrudRepository** and **PagingAndSortingRepository**.

Custom Queries

Spring Data JPA provides **three different approaches for creating custom queries** with query methods. Each of these approaches is described in following.

Using Method Name

- Spring Data JPA has a built in query creation mechanism which can be used for parsing queries straight from the method name of a query method.
- the method names of your repository interface are created **by combining the property names of an entity object and the supported keywords.**

```
public interface PersonRepository extends Repository<User, Long> {

        List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

        // Enables the distinct flag for the query
List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

        // Enabling ignoring case for an individual property
        List<Person> findByLastnameIgnoreCase(String lastname);

        // Enabling ignoring case for all suitable properties
List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

        // Enabling static ORDER BY for a query
        List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
        List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

JPA Named Queries

Spring Data JPA provides also support for the JPA Named Queries. You have got following alternatives for declaring the named queries:

- You can use either ***named-query*** **XML** element or ***@NamedQuery*** annotation to create named queries with the JPA query language.
- You can use either ***named-native-query*** XML element **or *@NamedNative*** query annotation to create queries with SQL if you are ready to tie your application with a specific database platform.

The only thing you have to do to use the created named queries is to name the query method of your repository interface to match with the name of your named query. See below Example code

```
@Entity
@NamedQuery(name = "Person.findByName", query = "SELECT p FROM Person p WHERE LOWER(p.lastName) =
LOWER(?1)")
@Table(name = "persons")
public class Person {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;

        @Column(name = "creation_time", nullable = false)
        private Date creationTime;

        @Column(name = "first_name", nullable = false)
        private String firstName;
}
```

The relevant part of my ***PersonRepository*** interface looks following

```
public interface PersonRepository extends JpaRepository<Person, Long> {
```

```
    //A list of persons whose last name is an exact match with the given last name.
        public List<Person> findByName(String lastName);
}
```

@Query Annotation

- The ***@Query*** annotation can be used to create queries by using the JPA query language and to **bind these queries directly to the methods of your repository interface**.
- When the query method is called, Spring Data **JPA will execute the query specified by the *@Query* annotation**
- If there is a collision between the *@Query* annotation and the named queries, the query specified by using *@Query* annotation will be executed

```
public interface ProductRepository
   extends CrudRepository<Product, Long> {
    @Query("FROM Product")
    List<Product> findAllProducts();
}
```

You may use positional parameters instead of named parameters in queries. Positional parameters are prefixed with a question mark **(?)** followed the numeric position of the parameter in the query. The `Query.setParameter(integer position, Object value)` method is used to set the parameter values.

```
public List findWithName(String name) {
    return em.createQuery(
        “SELECT c FROM Customer c WHERE c.name LIKE ?1”)
        .setParameter(1, name)
        .getResultList();
}
```

Automatic Query Generation

The **<jpa:repositories/>** has an option query-lookup-strategy which defaults to “**create-if-not-found” which will generate queries for us.**The default is “create-if-not-found”. Other options are “create” or “use-declared-query”.

```
<jpa:repositories base-package="com.gordondickens.myapp.repository"
     query-lookup-strategy="create-if-not-found"/>
```

To create a find method that effectively does `@Query("FROM Product p where p.productId =:productId")`

```
public interface ProductRepository extends CrudRepository<Product, Long> {
   ...

   @Query
   Product findByProductId(String productId);

   ...
```

Example

Student.java

```
package app.entity;

@Entity
@Table(name = "student")
public class Student {

	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int sno;

	@Column(name = "name")
	private String name;

	@Column(name = "address")
	private String address;

	@Override
	public String toString() {
	String str = "Student[" + "Sno: " + getSno() + ", Name:" + getName() + ", " +      "Address : " +
getAddress() + "]";
		return str;
	}
//Setters & getters
}
```

StudentRepository.java

```
public interface StudentRepository extends CrudRepository<Student, Long> {

	List<Student> findBySno(int sno);

	List<Student> findByName(String name);

	// custom query example and return a stream
	@Query("select c from Student c where c.address = :address")
	Stream<Student> findByAddress(@Param("address") String address);
}
```

Application.java

```
package app;
@SpringBootApplication
public class Application implements CommandLineRunner {

	@Autowired
	DataSource dataSource;

	@Autowired
	StudentRepository repository;

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

	@Transactional(readOnly = true)
	@Override
	public void run(String... args) throws Exception {
```

```
                System.out.println("DATASOURCE = " + dataSource);

                System.out.println("\n1.findAll()...");
                for (Student student : repository.findAll()) {
                        System.out.println(student);
                }

                System.out.println("\n2.findByName(String name)...");
                for (Student student : repository.findByName("Satya")) {
                        System.out.println(student);
                }

                System.out.println("\n3.findByAddress(@Param(\"name\"))...");
                try (Stream<Student> s =repository.findByAddress("Vijayawada")) {
                        s.forEach(x -> System.out.println(x));
                        System.out.println("Done!");
                        exit(0);
                }
        }
}
1.findAll()...
Student[Sno: 0, Name:null, Address : null]
Student[Sno: 101, Name:Satya, Address : VIJAYAWADA]
Student[Sno: 102, Name:Satya, Address : Vijayawada]
Student[Sno: 147, Name:kumar, Address : Hyderabad]
Student[Sno: 189, Name:Satya, Address : Vijayawada]
Student[Sno: 508, Name:Satya, Address : Hyd]

2.findByName(String name)...
Student[Sno: 101, Name:Satya, Address : VIJAYAWADA]
Student[Sno: 102, Name:Satya, Address : Vijayawada]
Student[Sno: 189, Name:Satya, Address : Vijayawada]
Student[Sno: 508, Name:Satya, Address : Hyd]

4.findByAddress(@Param("name") String name)...
Student[Sno: 101, Name:Satya, Address : VIJAYAWADA]
Student[Sno: 102, Name:Satya, Address : Vijayawada]
Student[Sno: 189, Name:Satya, Address : Vijayawada]
```

Spring Boot –MongoDB REST Example

Configuration file application.properties

```
# Create new database : ‘smlcodes’
spring.data.mongodb.database=smlcodes
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
```

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

```
package smlcodes.repository;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document
public class Student{

        @Id
        String sno;
        String name;
        String address;
        public String getSno() {
                return sno;
        }
        public void setSno(String sno) {
                this.sno = sno;
        }
        public String getName() {
```

```
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public String getAddress() {
                return address;
        }
        public void setAddress(String address) {
                this.address = address;
        }
}
```

- **@Id-** id provided by Mongo for a document.
- **@Document-** provides a collection name.

BookingRepository.java

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

```
@Transactional
public interface StudentRepository extends MongoRepository<Student, String> {
          public Student findBySno(int sno);
}
```

BookingController.java

```
package smlcodes.controller;

@RestController
@RequestMapping("/student")
public class StudentController {

        @Autowired
        StudentRepository studentRepository;

        @RequestMapping("/create")
        public Map<String, Object> create(Student student) {
                student = studentRepository.save(student);
                Map<String, Object> dataMap = new HashMap<String, Object>();
                dataMap.put("message", "Student created successfully");
                dataMap.put("status", "1");
                dataMap.put("student", student);
            return dataMap;
        }

        @RequestMapping("/read")
        public Map<String, Object> read(@RequestParam int sno) {
                Student student = studentRepository.findBySno(sno);
                Map<String, Object> dataMap = new HashMap<String, Object>();
                dataMap.put("message", "Student found successfully");
                dataMap.put("status", "1");
                dataMap.put("student", student);
            return dataMap;
        }

        @RequestMapping("/readall")
        public Map<String, Object> readAll() {
                List<Student> students = studentRepository.findAll();
                Map<String, Object> dataMap = new HashMap<String, Object>();
                dataMap.put("message", "Student found successfully");
                dataMap.put("totalStudent", students.size());
```

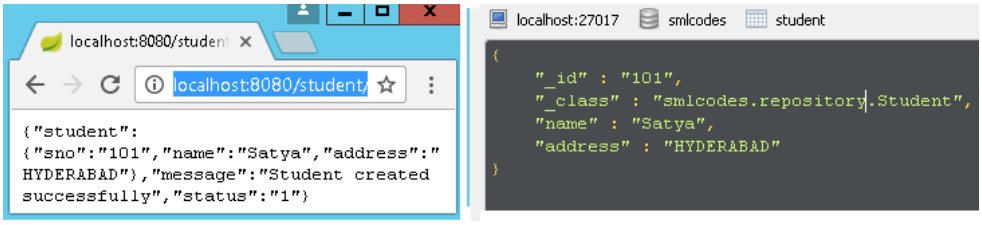
```
                dataMap.put("status", "1");
                dataMap.put("students", students);
            return dataMap;
        }
}
```

SpringBootMongoDbApplication.java

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

Test

http://localhost:8080/student/create?sno=101&name=Satya&address=HYDERABAD



Spring Missing/ Confusing Topics

Diffrence between @Component & @Autowire

@Compoent is just creating bean object of that class at Auto Scanning time.

@Autowire is, after creating bean object in @Componet time, it will autowires/ injecets the Dependent classes.

Diffrence beteen @ComponentScan & @Enable AutoCOnfiguration

One of the main advantages of Spring Boot is its annotation driven versus traditional xml based configurations, **@EnableAutoConfiguration** automatically configures the Spring application based on its included jar files, it sets up defaults or helper based on dependencies in pom.xml. Auto-configuration is usually applied based on the classpath and the defined beans. Therefore, we donot need to define any of the DataSource, EntityManagerFactory, TransactionManager etc and magically based on the classpath, Spring Boot automatically creates proper beans and registers them for us. For example when there is a tomcat-embedded.jar on your classpath you likely need a TomcatEmbeddedServletContainerFactory (unless you have defined your own EmbeddedServletContainerFactory bean). @EnableAutoConfiguration has a exclude attribute to disable an auto-configuration explicitly otherwise we can simply exclude it from the pom.xml, for example if we donot want Spring to configure the tomcat then exclude spring-bootstarter-tomcat from spring-boot-starter-web.

@ComponentScan provides scope for spring component scan, it simply goes though *the provided base package* and picks up dependencies required by @Bean or @Autowired etc, In a typical Spring application, @ComponentScan is used in a configuration classes, the ones annotated with @Configuration. Configuration classes contains methods annotated with @Bean. These @Bean annotated methods generate beans managed by Spring container. Those beans will be auto-detected by @ComponentScan annotation. There are some annotations which make beans auto-detectable like @Repository , @Service, @Controller, @Configuration, @Component. In below code Spring starts scanning from the package including BeanA class.

References

- http://sivalabs.in/2016/03/springboot-working-with-jdbctemplate/
- https://www.javatpoint.com/spring-boot-jpa
- https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-two-crud/
- https://www.dineshonjava.com/spring-boot-and-mongodb-in-rest-application/
- https://spring.io/guides/gs/spring-boot/#scratch
- http://docs.spring.io/autorepo/docs/spring-boot/current/reference/html/(best)
- http://www.dineshonjava.com/2016/06/introduction-to-spring-boot-a-spring-boot-complete-guide.html#.WI7wB1N965t
- http://websystique.com/spring-boot-tutorial/
- https://www.mkyong.com/tag/spring-boot/
- (Best)Helloworld : http://www.shristitechlabs.com/introduction-to-spring-boot/
- https://www.mkyong.com/spring-boot/spring-boot-spring-data-mongodb-example/
- https://tests4geeks.com/spring-data-boot-mongodb-example/
- https://avaldes.com/building-a-realtime-angularjs-dashboard-using-spring-rest-and-mongodb-part-1/
- Final:https://www.callicoder.com/spring-boot-mongodb-angular-js-rest-api-tutorial/

AngularJS with SpringBoot

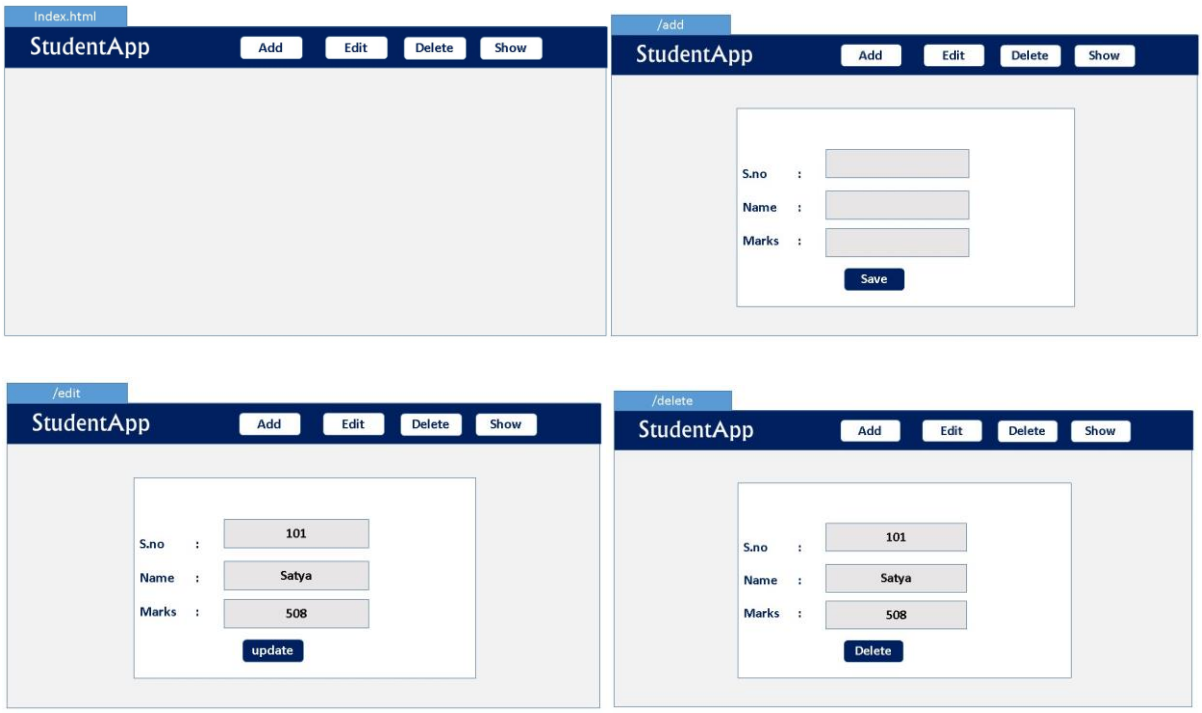
AngularJs is a Single page application. That means we have only index.html file, but we can change the views in single page.

Index.html

```
<body ng-app="">

   ….
```

```
    <div ng-view="">
          // we can change the views as per controller logic display view
     </div>

    ….

<body>
```

SpringBoot App UI



Before going to the implementation, we need to know Basics once.

Basics

AngularJS extends HTML attributes with **Directives**, and binds data to HTML with **Expressions**.

AngularJS is distributed as a JavaScript file, and can be added to a web page with a script tag:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.9/angular.min.js"></script>
```

- The **ng-app** directive defines an AngularJS application.
- The **ng-model** directive binds the value of HTML controls (input, select, textarea) to application data.
- The **ng-bind** directive binds application data to the HTML view
- **{{ expression }}**. expressions bind AngularJS data to HTML the same way as the **ng-bind** directive

```
<div ng-app="">
  <p>Name: <input type="text" ng-model="name"></p>
  <p ng-bind="name"></p>

 <p>{{name}} </p> //will print the same
</div>
```

Understand with Example

AngularJS Example : index.html

```
<div ng-app="myApp" ng-controller="myCtrl">

First Name: <input type="text" ng-model="firstName"><br> //binds input with model
Last Name: <input type="text" ng-model="lastName"><br> ////binds input with model
Full Name: {{firstName + " " + lastName}}            //Prints the model values
</div>

<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
    $scope.firstName= "John";
    $scope.lastName= "Doe";
});
</script>
```

1.ng-app

- The `ng-app` directive tells AngularJS that this is the root element of the AngularJS application.
- All AngularJS applications must have a root element.
- You can only have one `ng-app` directive in your HTML document. If more than one `ng-app` directive appears, the first appearance will be used

In index.html. We will have to tell Angular in which part of the application it should be active.

You saw that when declaring the angular module, we named it **app**. To tell it where it should be active we add the attribute **ng-app="app"** in the tag and everything inside of it turns into an AngularJS application.

In our case, as the whole page will be an ngapp it is better to place the attribute in the <html> tag or in the <body> tag.

only one AngularJS application can be auto-bootstrapped per HTML document. The first `ngApp` found in the document will be used to define the root element to auto-bootstrap as an application. To run multiple applications in an HTML document you must manually bootstrap them using `angular.bootstrap` instead.

2.ng-controller

The `ngController` directive specifies a Controller class; the class contains business logic behind the application to decorate the scope with functions and values.

In above '**myCtrl**'has following business logic

```
app.controller('myCtrl', function($scope) {
    $scope.firstName= "John";
    $scope.lastName= "Doe";
});
```

3.angular.module

The `angular.module` is a global place for creating, registering and retrieving AngularJS modules. All modules (AngularJS core or 3rd party) that should be available to an application must be registered using this mechanism.

```
var app = angular.module('myApp', []);
```

in this line we registred the module with variable 'app'. Now we can access this module in whole application with variable name to perform any kind of operations.

A module is a collection of services, directives, controllers, filters, and configuration information. `angular.module` is used to configure the $injector.

```
// Create a new module
var myModule = angular.module('myModule', []);

// register a new service
myModule.value('appName', 'MyCoolApp');

// configure existing services inside initialization blocks.
myModule.config(['$locationProvider', function($locationProvider) {
  // Configure existing providers
  $locationProvider.hashPrefix('!');
}]);
```

3.$scope

- $scope is a Global Object which can accessed by both view and controller.
- When adding properties to the `$scope` object in the controller, the view (HTML) gets access to these properties
- All applications have a `$rootScope` which is the scope created on the HTML element that contains the `ng-app` directive.
- The rootScope is available in the entire application.

AngularJs internal working

Now you will take a look at the architecture concepts of AngularJS. When an HTML document is loaded into the browser and is evaluated by the browser, the following happens:

1. The **AngularJS JavaScript** file is loaded, and the Angular global object `$scope` is created. The JavaScript file that registers the controller functions is executed.

2. AngularJS scans the HTML to look for **AngularJS apps and views** and finds **a controller function corresponding to the view.**

3. AngularJS **executes the controller functions and updates the views** with data from the model populated by the controller.

4. AngularJS listens for browser events, such as button clicked, mouse moved, input field being changed, and so on. If any of these events happen, then AngularJS will update the view accordingly

Bootstrapping AngularJS by Adding ng-app in an HTML Page

```
<html lang="en" ng-app="userregistrationsystem">...</html>
```

This is also known as automatic initialization. So, when AngularJS finds the **ng-app** directive after analyzing the **index.html** file, it loads the associated modules and then compiles the DOM.

Few More Examples

AngularJS Example : ng-init, {{ expression }} = ng-bind

```
<div ng-app="" ng-init="firstName='John';lastName='Doe'">
        <p>The name is {{ firstName + " " + lastName }}</p>
        <p>The name is <span ng-bind="firstName + ' ' + lastName"></span></p>
</div>
```

AngularJS Objects

AngularJS objects are like JavaScript objects:

```
<div ng-app="" ng-init="person={firstName:'John',lastName:'Doe'}">
       <p>The name is {{ person.lastName }}</p> //or
      <p>The name is <span ng-bind="person.lastName"></span></p>
</div>
```

AngularJS Arrays

AngularJS arrays are like JavaScript arrays:

```
<div ng-app="" ng-init="points=[1,15,19,2,40]">
     <p>The third result is {{ points[2] }}</p>
</div>
```

AngularJS Module

- Creating a Module

```
<div ng-app="myApp">...</div>
<script>
     var app = angular.module("myApp", []);
</script>
```

- Adding a Controller

```
<script>

var app = angular.module("myApp", []);

app.controller("myCtrl", function($scope) {
  $scope.firstName = "John";
  $scope.lastName = "Doe";
});

</script>
```

AngularJs Validation

The `ng-model` directive can provide type validation for application data (number, e-mail, required):

```
<form ng-app="" name="myForm">
  Email:
  <input type="email" name="myAddress" ng-model="text">
  <span ng-show="myForm.myAddress.$error.email">Not a valid e-mail address</span>
</form>
```

AngularJs Controllers – with Different Data

- Inside another function

```
<script>
var app = angular.module('myApp', []);
app.controller('personCtrl', function($scope) {
  $scope.firstName = "John";
  $scope.lastName = "Doe";
  $scope.fullName = function() {
    return $scope.firstName + " " + $scope.lastName;
  };
});
</script>
```

- With Array data

```
File : namesController.js
angular.module('myApp', []).controller('namesCtrl', function($scope) {
  $scope.names = [
    {name:'Jani',country:'Norway'},
    {name:'Hege',country:'Sweden'},
    {name:'Kai',country:'Denmark'}
  ];
});
```

```
<div ng-app="myApp" ng-controller="namesCtrl">
<ul>
  <li ng-repeat="x in names">
    {{ x.name + ', ' + x.country }}
  </li>
</ul>
</div>
<script src="namesController.js"></script>
```

AngularJS Filters

AngularJS provides filters to transform data:

- `uppercase` Format a string to upper case.

```
<p>The name is {{ lastName | uppercase }}</p>
```

- `lowercase` Format a string to lower case.

```
<p>The name is {{ lastName | lowercase }}</p>
```

- `currency` Format a number to a currency format.

```
<h1>Price: {{ price | currency }}</h1> //output is : Price: $58.00
```

- `filter` Select a subset of items from an array.

  This example displays only the names containing the letter "i".

```
<ul>
  <li ng-repeat="x in names | filter : 'i'">
    {{ x }}
  </li>
</ul>
```

- `orderBy` Orders an array by an expression.

```
<ul>
  <li ng-repeat="x in names | orderBy:'country'">
    {{ x.name + ', ' + x.country }}
  </li>
</ul>
```

- `json` Format an object to a JSON string.
- `date` Format a date to a specified format.
- `number` Format a number to a string.
- `limitTo` Limits an array/string, into a specified number of elements/characters.

AngularJS Services

- In AngularJS, a service is a **function, or object**, that is available for your AngularJS application.
- AngularJS has about 30 built-in services. One of them is the `$location` service.

1.$location

returns information about the location of the current web page:

```
var app = angular.module('myApp', []);
app.controller('customersCtrl', function($scope, $location) {
    $scope.myUrl = $location.absUrl();
});

{{myUrl}} //prints
https://www.w3schools.com/angular/tryit.asp?filename=try_ng_services
```

$2.$http

$http is an AngularJS service for reading data from remote servers. The AngularJS `$http` service makes a request to the server, and returns a response.

```
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope, $http) {
  $http.get("welcome.htm").then(function (response) {
    $scope.myWelcome = response.data;
  });
});
```

More on Http

The example above uses the `.get` method of the `$http` service.

The `.get` method is a shortcut method of the `$http` service. There are several shortcut methods:

- `.get()`
- `.post()`
- `.put()`
- `.delete()`
- `.head()`
- `.jsonp()`
- `.patch()`

```
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope, $http) {
  $http({
    method : "GET",
      url : "welcome.htm"
  }).then(function mySuccess(response) {
    $scope.myWelcome = response.data;
  }, function myError(response) {
    $scope.myWelcome = response.statusText;
  });
});
```

Responnse Types

- `.config` the object used to generate the request.
- `.data` a string, or an object, carrying the response from the server.
- `.headers` a function to use to get header information.
- `.status` a number defining the HTTP status.
- `.statusText` a string defining the HTTP status.

```
<div ng-app="myApp" ng-controller="myCtrl">
  <p>Data : {{content}}</p>
  <p>Status : {{statuscode}}</p>
  <p>StatusText : {{statustext}}</p>
</div>

var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope, $http) {
  $http.get("welcome.htm")
  .then(function(response) {
    $scope.content = response.data;
    $scope.statuscode = response.status;
    $scope.statustext = response.statusText;
  });
});
```

```
Data : Hello AngularJS Students
Status : 200
StatusText :
```

`$timeout`

The `$timeout` service is AngularJS' version of the `window.setTimeout` function.

```
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope, $timeout) {
  $scope.myHeader = "Hello World!";
  $timeout(function () {
    $scope.myHeader = "How are you today?";
  }, 2000);
});
```

AngularJs DOM Elements

Like in JavaScript we can hide,show,disable DOM elements

```
<button disabled>Click Me!</button>     //to Disable button
```

- Hide/show

```
<p ng-show="true">I am visible.</p>
<p ng-show="false">I am not visible.</p>

<p ng-hide="true">I am not visible.</p>
<p ng-hide="false">I am visible.</p>
```

AngularJS Events

You can add AngularJS event listeners to your HTML elements by using one or more of these directives:

- ng-blur
- ng-change
- ng-click
- ng-copy
- ng-cut
- ng-dblclick
- ng-focus
- ng-keydown
- ng-keypress
- ng-keyup
- ng-mousedown
- ng-mouseenter
- ng-mouseleave
- ng-mousemove
- ng-mouseover
- ng-mouseup
- ng-paste

Increase the count variable when the mouse clicked.

```
<div ng-app="myApp" ng-controller="myCtrl">
  <button ng-click="count = count + 1">Click me!</button>
  <p>{{ count }}</p>
</div>

//At Start count is 0 once Js load
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.count = 0;
});
</script>
```

Same, replace with function call

```
<div ng-app="myApp" ng-controller="myCtrl">

<button ng-click="myFunction()">Click me!</button>
```

```
<p>{{ count }}</p>

</div>
<script>
var app = angular.module('myApp', []);
app.controller('myCtrl', function($scope) {
  $scope.count = 0;
  $scope.myFunction = function() {
    $scope.count++;
  }
});
</script>
```

AngularJS Includes

With **ng-include** directive, you can include HTML from an external file.

```
<div ng-include="'myFile.htm'"></div>
```

AngularJS Routing

The `ngRoute` module helps your application to become a Single Page Application.

- If you want to navigate to different pages in your application, but you also want the application to be a SPA (Single Page Application), with no page reloading, you can use the `ngRoute` module.
- The `ngRoute` module *routes* your application to different pages without reloading the entire application.

For doing this

- you must include the AngularJS Route module JS

```
<script src="https://ajax.googleapis.com/angular-route.js"></script>
```

- Then you must add the `ngRoute` as a dependency in the application module:

```
var app = angular.module("myApp", ["ngRoute"]);
```

- Now your application has access to the route module, which provides the `$routeProvider`. Use the `$routeProvider` to **configure(`app.config`)** different routes in your application:

```
app.config(function($routeProvider) {
  $routeProvider
  .when("/", {
    templateUrl : "main.htm"
  })
  .when("/red", {
    templateUrl : "red.htm"
  })
  .when("/green", {
    templateUrl : "green.htm"
  })
  .when("/blue", {
    templateUrl : "blue.htm"
  });
});
```

Example

```
<body ng-app="myApp">
<a href="#/!">Main</a></p>
<a href="#!red">Red</a>
<a href="#!green">Green</a>
<a href="#!blue">Blue</a>

<div ng-view>
       <! –HERE CONTENT CHANGES -->
</div>

<script>
var app = angular.module("myApp", ["ngRoute"]);


app.config(function($routeProvider) {
  $routeProvider
  .when("/", {
    templateUrl : "main.htm"
  })
  .when("/red", {
    templateUrl : "red.htm"
  })
  .when("/green", {
    templateUrl : "green.htm"
  })
  .when("/blue", {
```

```
    templateUrl : "blue.htm"
  });
});
</script>
</body>
```

1.ng-view

Your application needs a container to put the **content** provided by the routing.This container is the `ng-view` directive. There are three different ways to include the ng-view directive in your application:

1. `<div ng-view></div>`
2. `<div class="ng-view"></div>`
3. `<ng-view></ng-view>`

Applications can only have one `ng-view` directive, and this will be the placeholder for all views provided by the route.

2.routeProvider

With the `$routeProvider` you can also define a controller for each "view".

```
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
  $routeProvider
  .when("/", {
    templateUrl : "main.htm"
  })
  .when("/london", {
    templateUrl : "london.htm",
    controller : "londonCtrl"
  })
  .when("/paris", {
    templateUrl : "paris.htm",
    controller : "parisCtrl"
  });
});
app.controller("londonCtrl", function ($scope) {
  $scope.msg = "I love London";
});
app.controller("parisCtrl", function ($scope) {
```

```
  $scope.msg = "I love Paris";
});
```

3.template

In the previous examples we have used the `templateUrl` property in the `$routeProvider.when` method. You can also use the `template` property, which allows you to **write HTML directly in the property value**, and not refer to a page.

```
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
  $routeProvider
  .when("/", {
    template : "<h1>Main</h1><p>Click on the links to change this content</p>"
  })
  .when("/banana", {
    template : "<h1>Banana</h1><p>Bananas contain around 75% water.</p>"
  })
  .when("/tomato", {
    template : "<h1>Tomato</h1><p>Tomatoes contain around 95% water.</p>"
  });
});
```

4. otherwise

In the previous examples we have used the `when` method of the `$routeProvider`.You can also use the `otherwise` method, which is the default route when none of the others get a match.

If use not clicked **/banana,/tamoto** for example user clicks **/apple** then otherwise will do the job.

```
var app = angular.module("myApp", ["ngRoute"]);
app.config(function($routeProvider) {
  $routeProvider
  .when("/banana", {
    template : "<h1>Banana</h1><p>Bananas contain around 75% water.</p>"
  })
  .when("/tomato", {
    template : "<h1>Tomato</h1><p>Tomatoes contain around 95% water.</p>"
  })
  .otherwise({
    template : "<h1>None</h1><p>Nothing has been selected</p>"
  });
});
```

AngularJS Architecture Concepts

Now you will take a look at the architecture concepts of AngularJS. When an HTML document is loaded into the browser and is evaluated by the browser, the following happens:

1. The **AngularJS JavaScript** file is loaded, and the Angular global object `$scope` is created. The JavaScript file that registers the controller functions is executed.

2. AngularJS scans the HTML to look for **AngularJS apps and views** and finds **a controller function corresponding to the view.**

3. AngularJS **executes the controller functions and updates the views** with data from the model populated by the controller.

4. AngularJS listens for browser events, such as button clicked, mouse moved, input field being changed, and so on. If any of these events happen, then AngularJS will update the view accordingly

Bootstrapping AngularJS by Adding ng-app in an HTML Page

```
<html lang="en" ng-app="userregistrationsystem">...</html>
```

This is also known as automatic initialization. So, when AngularJS finds the **ng-app** directive after analyzing the **index.html** file, it loads the associated modules and then compiles the DOM.

The AngularJS application **UserRegistrationSystem** is defined as the AngularJS module (**angular.module**) in app.js, which is the entry point into the application.

```
var app = angular.module('userregistrationsystem', ['ngRoute', 'ngResource']);
```

As you can see, the two dependencies have been defined in **app.js** as needed by **userregistrationsystem** at startup. The two dependencies in the previous code are defined in an array in the module definition.

- **ngRoute:** The first dependency is the AngularJS **ngRoute** module, which provides routing to the application. The ngRoute module is used for deep-linking URLs to controllers and views.
- **ngResource:** The second dependency is the AngularJS **ngResource** module, which provides interaction support with RESTful services

AngularJS Routes

AngularJS routes are configured using them **$routeProvider** API. Routes are dependent on the ngRoute module, which is why its dependency is defined in an array in the module definition.

app.js. You will define four routes in your AngularJS application.

- The first is /list-all-users.
- The second one is /register-new-user
- The third one is /update-user/:id
- The fourth one is different from the other three

```
var app = angular.module('userregistrationsystem', [ 'ngRoute', 'ngResource' ]);

app.config(function($routeProvider) {
   $routeProvider.when('/list-all-users', {
          templateUrl : '/template/listuser.html',
          controller : 'listUserController'
   }).when('/register-new-user',{
          templateUrl : '/template/userregistration.html',
          controller : 'registerUserController'
   }).when('/update-user/:id',{
          templateUrl : '/template/userupdation.html' ,
          controller : 'usersDetailsController'
   }).otherwise({
          redirectTo : '/home',
          templateUrl : '/template/home.html',
   });
});
```

- When the user clicks the link in the application specified at http://localhost:8080/#/list-all-users, the /list-all-users route will be followed, and the content associated with the /list-all-users URL will be displayed

in app.config , each route is mapped to a template and controller (optional).

- The controller listUserController will be called when you navigate to the URL /list-all-users
- controller registerUserController will be called when you navigate to the URL /register-new-user

AngularJS Templates

- AngularJS templates, also known as HTML partials, are HTML code that are bound to the <div ng-view> tag shown in the index.html file. If you look at the code from the
- app.js file, you can see that different templateUrl values are defined for different routes, as shown in above code
- The listuser.html, userregistration.html , userupdation.html , and home.html pages are four different partials or templates, which contain HTML code and AngularJS's built-in template language to display dynamic data in your template.

```
<html lang="en" ng-app="userregistrationsystem">
<head>
<title>Full Stack Development</title>
<link rel="stylesheet" href="/css/app.css">
</head>
<body>
	<div class="page-header text-center">
		<h2>User Registration System</h2>
	</div>
<nav class="navbar navbar-default">
   <div class="container-fluid">
	<a href="#/" class=" navbar-btn" role="button">Home</a>
	<a href="#/register-new-user" role="button">  Register New User</a>
	<a href="#/list-all-users" class="" role="button">List All Users</a>
  </div>
</nav>
	<div ng-view></div>
	<script src="/webjars/angularjs/1.4.9/angular.js"></script>
	<script src="/webjars/angularjs/1.4.9/angular-resource.js"></script>
	<script src="/webjars/angularjs/1.4.9/angular-route.js"></script>
	<script src="/js/app.js"></script>
	<script src="/js/controller.js"></script>
	<link rel="stylesheet"
		href="/webjars/bootstrap/3.3.6/css/bootstrap.css">
</body>
</html>
```

You have included a separate app.js, which is where you will be defining the application behavior.

Let's create four view pages in the template folder inside the src/main/resources/static directory.

•Home page :src/main/resources/template/home.html

•Register New User page :src/main/resources/template/userregistration.html

```
<form ng-submit="submitUserForm()" name="myForm" class="form-horizontal">

	<input type="text" ng-model="user.name" id="uname" />
	<input type="text" ng-model="user.address" id="address"/>
	<input type="email" ng-model="user.email" id="email" />

	<input type="submit" value="Register User"/>
	<button type="button" ng-click="resetForm()">Reset Form</button>
</form>
```

•List Of User page :**src/main/resources/template/listuser.html**

```
<table class="table table-hover table-bordered ">
	<thead>
		<tr>
			<th>Name</th>
			<th>Email</th>
			<th>Address</th>
			<th width="100">Edit</th>
			<th width="100">Delete</th>
		</tr>
	</thead>
	<tbody>
```

```
                <tr ng-repeat="user in users">
                        <td>{{user.name}}</td>
                        <td>{{user.email}}</td>
                        <td>{{user.address}}</td>
                        <td>
        <button type="button" ng-click="editUser(user.id)">      Edit</button>
                        </td>
                <td>
        <button type="button" ng-click="deleteUser(user.id)"> Delete</button></td>
        </tr>
        </tbody>
</table>
```

AngularJS Controller

controller.js file defined in the **src/main/resources/static/js** folder contains the implementation of AngularJS controllers

- -On a successful POST call, it will redirect to list-all-users
- -$scope is used to set up dynamic content for the UI elements that this controller is responsible for
- -$http: The $http service is the core feature provided by AngularJS and is used to consume the REST

```
app.controller('registerUserController', function($scope, $http, $location,
                $route) {

        $scope.submitUserForm = function() {
                $http({
                        method : 'POST',
                        url : 'http://localhost:8080/api/user/',
                        data : $scope.user,
                }).then(function(response) {
                        $location.path("/list-all-users");
                        $route.reload();
                }, function(errResponse) {
                        $scope.errorMessage = errResponse.data.errorMessage;
                });
        }

        $scope.resetForm = function() {
                $scope.user = null;
        };
});
```

AngularJs + SpringBoot Example implementation

Index.html

- Create index.html & add all Anuglar related jars
- Declare body ng-app="*stApp*"> which will be the Startig point of our AngularJs App
- Create module

```
var app = angular.module("myApp", []);
```

if no dependencies are there [ ] will be empty

```
<body ng-app="stApp">

<nav class="navbar navbar-inverse">
ADD, EDIT, DELETE etc
</nav>

<div class="container">
 <h1>Student Application</h1>
</div>

</body>
</html>
```

References

https://www.slideshare.net/lochnguyen/angular-js-for-java-developers-40120787

https://medium.com/@swhp/build-single-page-application-with-java-ee-and-angularjs-4eaacbdfcd

https://examples.javacodegeeks.com/desktop-java/ide/eclipse/eclipse-ide-angularjs-tutorial/

https://www.webcodegeeks.com/download/16411/?dlm-dp-dl-force=1&dlm-dp-dl-nonce=33f2636c9a

JUNIT

1.JUNIT Introduction

Testing is the process of checking the functionality of an application to ensure it runs as per requirements. **Unit testing** comes into picture at the developers' level; it is the testing of single entity **(class or method).** Unit testing can be done in two ways –**manual testing & automated testing**

1. Manual Testing: If you execute the test cases manually without any tool support, it is known as manual testing. It is time consuming and less reliable.

2. Automated Testing: If you execute the test cases by tool support, it is known as automated testing. It is fast and more reliable.

XUnit architecture **introduced automated unit testing**. There are many unit testing frame works for different programming. Few of the unit testing frame works are:

- JAVA - **JUnit**,
- C - **CUnit**,
- C++ - **CPPUnit**,
- .NET - .**NUnit** etc..

XUnit architecture was first implemented for java.**It is known as JUnit**

JUnit

It is an open-source testing framework for java programmers. The java programmer can create test cases and test his/her own code.It is one of the unit testing framework. Current version is **junit 5**.you can download it from **JUnit website (github)** or use below maven dependency in your **pom.xml**

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
```

```
  <scope>test</scope>
</dependency>
```

The Junit 4.x framework is annotation based. Here're some basic JUnit annotations

- **@Test -**Given method is the test method.= `public void`
- **@Test(timeout=1000)** -method will be failed if it takes more then 1000 milliseconds (1 sec).
- **@BeforeClass** - method will be invoked only once, before starting all the tests. `public static void`
- **@AfterClass** -method will be invoked only once, after finishing all the tests `public static void`
- **@Before** -method will be invoked before each test. Run before @Test, `public void`
- **@After** - method will be invoked after each test. Run after @Test, `public void`

The most important package in JUnit is **junit.framework**, which contains all the core classes. Some of the most important classes are given below

1.Assert - set of assert methods.
2.TestCase - It is the testing of single entity (class or method)
3.TestSuite - A test suite bundles a few unit test cases and runs them together.
3.TestResult - Contains methods to collect the results of executing a test case.

REMEMBER: If Junit tests not Running means

By default Maven uses the following naming conventions when looking for tests to run:

- **Test***
- ***Test**
- ***TestCase**

1.Assert

The org.junit.Assert class provides methods to assert the program logic. Assert methods are usually used to **compare the actual value with the expected value**. All assert methods are static methods. Return type of all assert methods are void

1. **void assertEquals(boolean expected,boolean actual)**: checks that two primitives/objects are equal. It is overloaded.
2. **void assertTrue(boolean condition)**: checks that a condition is true.
3. **void assertFalse(boolean condition)**: checks that a condition is false.
4. **void assertNull(Object obj)**: checks that object is null.
5. **void assertNotNull(Object obj)**: checks that object is not null.

2.Test Case

It is the testing of single entity (class or method)

```
public class TestJunit1 {

   String message = "Robert";
   MessageUtil messageUtil = new MessageUtil(message);

   @Test
   public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      assertEquals(message, messageUtil.printMessage());
   }
}
```

3.Test Suites

A test suite bundles a few unit test cases and runs them together.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

//JUnit Suite Test
@RunWith(Suite.class)

@Suite.SuiteClasses({
   TestJunit1.class ,TestJunit2.class
})

public class JunitTestSuite {
}
```

4.Test Runners

Test runner is used for executing the test cases.

```
public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

TestResult(`Result`**)** – Contains methods to collect the results of executing a test case.

Example 1 : Testing Annotations Working

```
package basic;
import org.junit.*;
public class AnnotationsTest {
        // Run once, e.g. Database connection, connection pool
    @BeforeClass
    public static void runOnceBeforeClass() {
        System.out.println("@BeforeClass - runOnceBeforeClass");
    }

    // Run once, e.g close connection, cleanup
    @AfterClass
    public static void runOnceAfterClass() {
        System.out.println("@AfterClass - runOnceAfterClass");
    }

    // Should rename to @BeforeTestMethod
    // e.g. Creating an similar object and share for all @Test
    @Before
    public void runBeforeTestMethod() {
        System.out.println("@Before - runBeforeTestMethod");
    }

    // Should rename to @AfterTestMethod
    @After
    public void runAfterTestMethod() {
        System.out.println("@After - runAfterTestMethod");
    }

    @Test
    public void TestMethod1() {
        System.out.println("@Test - TestMethod1");
    }

    @Test
    public void TestMethod2() {
        System.out.println("@Test - TestMethod2");
    }
}
```

```
@BeforeClass – runOnceBeforeClass

@Before - runBeforeTestMethod
@Test - TestMethod1
@After – runAfterTestMethod

@Before - runBeforeTestMethod
@Test - TestMethod2
@After – runAfterTestMethod

@AfterClass - runOnceAfterClass
```

Note: All sources of production code commonly reside in the `src/main/java` directory, while all test source files are kept at `src/test/java`

2. JUnit Hello World!

To write testcases we must figute out below points

1. ***Class to be tested***
2. ***Write Testcases for selected class***

3. *Run the Test (Commandline / TestRunner class)*

1. Class to be tested

```
package junit;
public class Calculator {
public int square(int x){
        return x*x;
}
}
```

2. Write Testcases for selected class

```
package junit;
import static org.junit.Assert.*;
import org.junit.Test;
public class CalculatorTest {
        @Test
          public void squareTest() {
            Calculator calculator = new Calculator();
            int sqr = calculator.square(2);
            //Checking for 2
            assertEquals("2*2=4 Passed",4, sqr);//pass
            assertEquals("2*2=4 Passed",6, sqr); //Fail
          }
}
```

3. Run the Test (Commandline / TestRunner class)

Using command line

```
java -cp .;junit-4.XX.jar;hamcrest-core-1.3.jar org.junit.runner.JUnitCore CalculatorTest
```

Using TestRunner class

```
package junit;
import org.junit.runner.*;
public class TestRunner {
        public static void main(String[] args) {
                Result result = JUnitCore.runClasses(CalculatorTest.class);

                for (Failure failure : result.getFailures()) {
                        System.out.println("Failure : " + failure.toString());
                }
                System.out.println("Success : " + result.wasSuccessful());
        }
}
```

```
Failure : squareTest(junit.CalculatorTest): 2*2=4 Passed expected:<6> but
was:<4>
Success : false  //for assertEquals("2*2=4 Passed",6, sqr); //Fail

Success : true //for assertEquals("2*2=4 Passed",4, sqr);//pass
```

4. JUnit Examples

4.1 Assert all Methods Example

```
import static org.hamcrest.CoreMatchers.*;
```

```
import static org.junit.Assert.*;
import java.util.Arrays;
import org.hamcrest.core.CombinableMatcher;
import org.junit.Test;

public class AssertTests {
  @Test
  public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    assertArrayEquals("failure - byte arrays not same", expected, actual);
  }

  @Test  public void testAssertEquals() {
    assertEquals("failure - strings are not equal", "text", "text");
  }

  @Test  public void testAssertFalse() {
    assertFalse("failure - should be false", false);
  }

  @Test  public void testAssertNotNull() {
    assertNotNull("should not be null", new Object());
  }

  @Test  public void testAssertNotSame() {
    assertNotSame("should not be same Object", new Object(), new Object());
  }

  @Test  public void testAssertNull() {
    assertNull("should be null", null);
  }

  @Test  public void testAssertSame() {
    Integer aNumber = Integer.valueOf(768);
    assertSame("should be same", aNumber, aNumber);
  }

  // JUnit Matchers assertThat
  @Test  public void testAssertThatBothContainsString() {
    assertThat("albumen", both(containsString("a")).and(containsString("b")));
  }

  @Test  public void testAssertThatHasItems() {
    assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
  }

  @Test  public void testAssertThatEveryItemContainsString() {
    assertThat(Arrays.asList(new String[] { "fun", "ban", "net" }), everyItem(containsString("n")));
  }

  // Core Hamcrest Matchers with assertThat
  @Test  public void testAssertThatHamcrestCoreMatchers() {
    assertThat("good", allOf(equalTo("good"), startsWith("good")));
    assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
    assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
    assertThat(7, not(CombinableMatcher.<Integer> either(equalTo(3)).or(equalTo(4))));
    assertThat(new Object(), not(sameInstance(new Object())));
  }

  @Test  public void testAssertTrue() {
    assertTrue("failure - should be true", true);
  }
}
```

4.2 Test Suite

Test suite is used to bundle a few unit test cases and run them together. **@RunWith and @Suite** annotations are used to run the suite tests.

In below Example we are running Test1 & Test2 together using Test Suite.

1. Class to be tested

```
package testsuite;

public class Calculator {
        public int square(int x) {
                return x * x;
        }

        public int sum(int x, int y) {
                return x + y;
        }
}
```

2. Write Testcases for selected class

```
//1.Test1.java
package testsuite;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class Test1 {
        @Test
          public void squareTest() {
            Calculator calculator = new Calculator();
            int sqr = calculator.square(2);
            assertEquals("2*2=4 Passed",4, sqr);
          }
}
-----------------------------------------------------------------------------------------------------
//2.Test2.java
package testsuite;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class Test2 {
        @Test
          public void addTest() {
            Calculator calculator = new Calculator();
            int sum = calculator.sum(8,2);
            assertEquals("8+2=10 Passed",10, sum);
          }
}
```

3.Create Test Suite Class

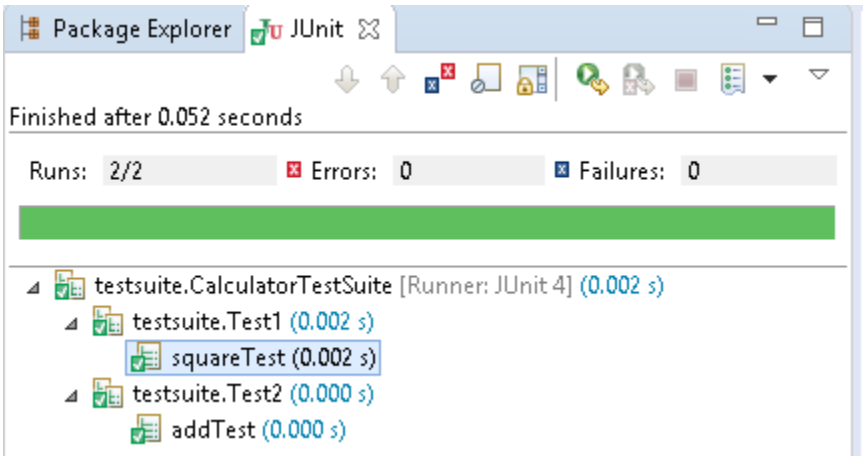
```
package testsuite;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
   Test1.class,
   Test2.class
})
public class CalculatorTestSuite {
}
```

4. Run the Test (Commandline / TestRunner class)

Remember, TestRunner class is same for all Examples

```
package testsuite;

import org.junit.runner.JUnitCore;
```

```
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
	public static void main(String[] args) {
		Result result = JUnitCore.runClasses(CalculatorTestSuite.class);

		for (Failure failure : result.getFailures()) {
			System.out.println("Failure : " + failure.toString());
		}
		System.out.println("Success : " + result.wasSuccessful());
	}
}
```



4.3 Ignore Test

@Ignore annotation is used to create Ignore test. We use @ignore in two cases

1. **At Method Level** : if a method annotated with @Ignore, **that method will not be executed.**
2. **At Class Level Level** : if a class annotated with @Ignore**, all its methods will not be executed.**

1. Class to be tested

```
package ignoretest;
public class IgnoreTestClassLevel {
	private String str1;
	private String str2;
	private String str3;

	public IgnoreTestClassLevel(String str1, String str2) {
		this.str1 = str1;
		this.str2 = str2;
	}

	public String addStrings() {
		str3 = str1 + str2;
		System.out.println("addStrings : " + str3);
		return str1 + str2;
	}

	public String upperCase() {
		str3 = (str1 + str2).toUpperCase();
		System.out.println("upperCase : " + str3);
		return str1 + str2;
	}
```

```
}
```

2. Ignore Test at Method Level

```
package ignoretest;
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class IgnoreTestMethodLevel {
        StringUtil util = new StringUtil("a", "b");
        String res = "";

        @Ignore
        @Test
        public void testAddStrings() {
                System.out.println("Inside testAddStrings()");
                res = "ab";
                assertEquals(res, util.addStrings());
        }

        @Test
        public void testUpperCase() {
                System.out.println("Inside testUpperCase()");
                res = "AB";
                assertEquals(res, util.upperCase());
        }
}
```

```
Inside testUpperCase()
upperCase : AB
Success : true
```

3. Ignore Test at Class Level

```
package ignoretest;
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

@Ignore
public class IgnoreTestClassLevel {
        StringUtil util = new StringUtil("a", "b");
        String res = "";

        @Test
        public void testAddStrings() {
                System.out.println("Inside testAddStrings()");
                res = "ab";
                assertEquals(res, util.addStrings());
        }

        @Test
        public void testUpperCase() {
                System.out.println("Inside testUpperCase()");
                res = "AB";
                assertEquals(res, util.upperCase());
        }
}
```

Empty Output, because none of its test methods will be executed.

4.4 Time Test

`@Test(timeout)-` timeout parameter along with @Test annotation as used for Time Test.If a test case takes more time than the specified number of milliseconds, then JUnit will automatically mark it as failed.

Example: Time Test example for above StringUtil.java Class

```
package ignoretest;
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class StringUtilTimeTest {
        StringUtil util = new StringUtil("a", "b");
        String res = "";

        @Test(timeout = 1000)
        public void testAddStrings() {
                System.out.println("Inside testAddStrings()");
                res = "ab";
                assertEquals(res, util.addStrings());
        }

        @Test
        public void testUpperCase() {
                System.out.println("Inside testUpperCase()");
                res = "AB";
                assertEquals(res, util.upperCase());
        }
}
```

4.5 Exceptions Test

`@Test(expected)-` expected parameter along with @Test annotation as used for Exceptions Test. we can test whether our code throws an expected exception or not

Example: ExceptionsTest

```
package junit;

public class Calculator {
        public int square(int x) {
                return x * x;
        }
        public int div(int a, int b) {
                return a / b;
        }
}
```

```
package junit;
import static org.junit.Assert.*;
import org.junit.Test;

public class CaluculatorExceptionTest {
         @Test(expected = ArithmeticException.class)
          public void divTest() {
            Calculator calculator = new Calculator();
            int res = calculator.div(12,0);      //success
           // int res = calculator.div(12,0);    //error
            assertEquals(4, res);
        }
}
```

4.6 Parameterized Test

Parameterized tests allow a developer to run the same test over and over again using different values.we use **`@RunWith(Parameterized.class)`** to achive this type of tests.

Example

```
public class EvenNumbers {
        public Boolean checkEven(final Integer num) {

                for (int i = 1; i <= num; i++) {
                        if (i % 2 == 0) {
                                return true;
                        }
                }
                return false;
        }
}
```

```
package parameterizedtest;

import java.util.*;
import org.junit.*;
import static org.junit.Assert.assertEquals;

@RunWith(Parameterized.class)
public class PrimeNumberCheckerTest {
   private Integer inum;
   private Boolean res;
   private EvenNumbers evenObj;

   @Before
   public void initialize() {
      evenObj = new EvenNumbers();
   }


   public PrimeNumberCheckerTest(Integer inum, Boolean res) {
      this.inum = inum;
      this.res = res;
   }

   @Parameterized.Parameters
   public static Collection evenNumbers() {
      return Arrays.asList(new Object[][] {
         { 2, true },
         { 6, true },
         { 18, true },
         { 19, false },
         { 48, true }
      });
   }

   @Test
   public void testPrimeNumberChecker() {
      System.out.println("Parameterized Number is : " + inum);
      assertEquals(res, evenObj.checkEven(inum));
   }
}
```

```
Parameterized Number is : 2
Parameterized Number is : 6
Parameterized Number is : 18
Parameterized Number is : 19
Parameterized Number is : 48
```

4.7 JUnit List Example

```
package other;

import org.junit.Test;
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import org.hamcrest.collection.IsEmptyCollection;
import static org.hamcrest.CoreMatchers.*;
import static org.hamcrest.collection.IsCollectionWithSize.hasSize;
import static org.hamcrest.collection.IsIterableContainingInAnyOrder.containsInAnyOrder;
import static org.hamcrest.collection.IsIterableContainingInOrder.contains;
import static org.hamcrest.number.OrderingComparison.greaterThanOrEqualTo;
import static org.hamcrest.number.OrderingComparison.lessThan;
import static org.hamcrest.MatcherAssert.assertThat;

public class ListExample{

    @Test
    public void testAssertList() {

        List<Integer> actual = Arrays.asList(1, 2, 3, 4, 5);
        List<Integer> expected = Arrays.asList(1, 2, 3, 4, 5);

        //All passed / true

        //1. Test equal.
        assertThat(actual, is(expected));

        //2. Check List has this value
        assertThat(actual, hasItems(2));

        //3. Check List Size
        assertThat(actual, hasSize(4));

        assertThat(actual.size(), is(5));

        //4.  List order

        // Ensure Correct order
        assertThat(actual, contains(1, 2, 3, 4, 5));

        // Can be any order
        assertThat(actual, containsInAnyOrder(5, 4, 3, 2, 1));

        //5. check empty list
        assertThat(actual, not(IsEmptyCollection.empty()));

        assertThat(new ArrayList<>(), IsEmptyCollection.empty());

                //6. Test numeric comparisons
        assertThat(actual, everyItem(greaterThanOrEqualTo(1)));

        assertThat(actual, everyItem(lessThan(10)));

    }


}
```

4.8 JUnit Map Example

```
public class MapExample {
        @Test
```

```
    public void testAssertMap() {

        Map<String, String> map = new HashMap<>();
        map.put("j", "java");
        map.put("c", "c++");
        map.put("p", "python");
        map.put("n", "node");

        Map<String, String> expected = new HashMap<>();
        expected.put("n", "node");
        expected.put("c", "c++");
        expected.put("j", "java");
        expected.put("p", "python");

        //All passed / true

        //1. Test equal, ignore order
        assertThat(map, is(expected));

        //2. Test size
        assertThat(map.size(), is(4));

        //3. Test map entry, best!
        assertThat(map, IsMapContaining.hasEntry("n", "node"));

        assertThat(map, not(IsMapContaining.hasEntry("r", "ruby")));

        //4. Test map key
        assertThat(map, IsMapContaining.hasKey("j"));

        //5. Test map value
        assertThat(map, IsMapContaining.hasValue("node"));

    }
}
```

4.9 JUnit Tools

Following are the JUnit Tools used for Testing –

1. Cactus
2. JWebUnit
3. XMLUnit
4. MockObject

References

http://junit.org/junit4/

https://www.tutorialspoint.com/junit/

http://www.javatpoint.com/junit-tutorial

http://www.mkyong.com/tutorials/junit-tutorials/

Mockito

Mockito facilitates creating mock objects(sample , similar Objects). Mock objects are nothing but proxy for actual implementations.

1.Creating Mock Objects

1.Normal way

```
      StockService stockServiceMock = mock(StockService.class);
```

2.Using Annotation

```
@Mock
 StockService stockServiceMock
```

2.Methods & Usage

All trhese methods are **static,** import provided by Mockito

```
import static org.mockito.Mockito.*;
```

when(...).thenReturn(...); - adds a functionality to a mock object using the methods when(). Then()

```
when(...).thenReturn(...);
```

// mock the behavior of stock service to return the value of various stocks

```
      when(stockServiceMock.getPrice(googleStock)).thenReturn(50.00);

      when(stockServiceMock.getPrice(microsoftStock)).thenReturn(1000.00);
```

```
// @RunWith attaches a runner with the test class to initialize the test data
@RunWith(MockitoJUnitRunner.class)
public class MathApplicationTester {

   //@InjectMocks annotation is used to create and inject the mock object
```

```
    @InjectMocks //main Object
    MathApplication mathApplication = new MathApplication();

    //@Mock annotation is used to create the dependent mock object
    @Mock //Calculator object id decalted inside MathApplication
    CalculatorService calcService;

    @Test
    public void testAdd(){
        //add the behavior of calc service to add two numbers
        when(calcService.add(10.0,20.0)).thenReturn(30.00);

        //test the add functionality
        Assert.assertEquals(mathApplication.add(10.0, 20.0),30.0,0);
    }
}
```

verify() method is ensure whether a mock method is being called with reequired arguments or not

```
//verify call to calcService is made or not with same arguments
verify(calcService).add(20.0, 30.0);
```

Times() method - Suppose MathApplication should call the CalculatorService.serviceUsed() method only once

```
      //check if add function is called three times
      verify(calcService, times(3)).add(10.0, 20.0);

      //verify that method was never called on a mock
      verify(calcService, never()).multiply(10.0,20.0)
```

Mockito provides the following additional methods to vary the expected call counts

```
      //check a minimum 1 call count
      verify(calcService, atLeastOnce()).subtract(20.0, 10.0);

      //check if add function is called minimum 2 times
      verify(calcService, atLeast(2)).add(10.0, 20.0);

      //check if add function is called maximum 3 times
      verify(calcService, atMost(3)).add(10.0,20.0);
```

Exception

```
@Test(expected = RuntimeException.class)
   public void testAdd(){
      //add the behavior to throw exception
      doThrow(new RuntimeException("Add operation not implemented"))
         .when(calcService).add(10.0,20.0);
}
```

InOrder class - takes care of the order of method calls

```
      //create an inOrder verifier for a single mock
      InOrder inOrder = inOrder(calcService);

      //following will make sure that add is first called then subtract is called.
      inOrder.verify(calcService).subtract(20.0,10.0);
      inOrder.verify(calcService).add(20.0,10.0);
```

reset - reset a mock so that it can be reused later

```
//reset the mock
      reset(calcService);
```

spy()/@Spy: partial mocking, real methods are invoked but still can be verified and stubbed

Hamcrest Matchers

Hamcrest provides a more readable, declarative approach to asserting and matching your test results.

Hamcrest has the target to make tests as readable as possible. For example, the `is` method is a thin wrapper for `equalTo(value)`.

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.is;
import static org.hamcrest.Matchers.equalTo;

boolean a;
boolean b;

// all statements test the same
assertThat(a, equalTo(b));
assertThat(a, is(equalTo(b)));
assertThat(a, is(b));
```

The following snippets compare pure JUnit 4 assert statements with Hamcrest matchers.

```
// JUnit 4 for equals check
   assertEquals(expected, actual);
   // Hamcrest for equals check
   assertThat(actual, is(equalTo(expected)));

   // JUnit 4 for not equals check
   assertNotEquals(expected, actual)
   // Hamcrest for not equals check
   assertThat(actual, is(not(equalTo(expected))));
```

The following are the most important Hamcrest matchers:

- `allOf` - matches if all matchers match (short circuits)
- `anyOf` - matches if any matchers match (short circuits)
- `not` - matches if the wrapped matcher doesn't match and vice
- `equalTo` - test object equality using the equals method
- `is` - decorator for equalTo to improve readability
- `hasToString` - test Object.toString
- `instanceOf`, `isCompatibleType` - test type
- `notNullValue`, `nullValue` - test for null
- `sameInstance` - test object identity
- `hasEntry`, `hasKey`, `hasValue` - test a map contains an entry, key or value
- `hasItem`, `hasItems` - test a collection contains elements
- `hasItemInArray` - test an array contains an element
- `closeTo` - test floating point values are close to a given value
- `greaterThan`, `greaterThanOrEqualTo`, `lessThan`, `lessThanOrEqualTo`
- `equalToIgnoringCase` - test string equality ignoring case
- `equalToIgnoringWhiteSpace` - test string equality ignoring differences in runs of whitespace
- `containsString`, `endsWith`, `startsWith` - test string matching

Integration Testing

Integration testing is all about testing all pieces of an application working together as they would in a live or production environment

To convert any JUnit test into a proper integration test, there are really two basic things that you need to do.

- The first is you need to annotate your tests with the `@RunWith` annotation and specify that you want to run it with the `SpringJUnit4ClassRunner.class`

- The second is you need to add the `@SpringApplicationConfiguration` annotation and provide your main Spring Boot class for your application.

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(Demo.class)
public class UserRepoIntegrationTest {
    @Autowired
    private UserRepository userRepository;

    @Test
    public void testFindAll() {
        List<User> users = userRepository.findAll();
        assertThat(users.size(), is(greaterThanOrEqualTo(0)));
    }
}
```

Regardless of the test result - successful or unsuccessful, open your IDE Console tab and you should notice that it looks like your application started (Spring logo, info etc). This happens because our application actually starts with integration tests

MockMvc

Spring MockMVC to perform **integration testing** of spring webmvc controllers

MockMVC class is part of Spring MVC test framework which helps in testing the controllers explicitly starting a Servlet container.

```
@RunWith(SpringRunner.class)
@WebMvcTest(StudentController.class)
public class StudentIntegrationTests {

        @Autowired
         private MockMvc mvc;
}
```

- `SpringRunner` is an alias for the `SpringJUnit4ClassRunner`.
- `@WebMvcTest` annotation is used for Spring MVC tests. It disables full auto-configuration and instead apply only configuration relevant to MVC tests.
- `StudentController.class` means initialize only this controller and provide dependent Mock object of this controller.

MockMvcRequestBuilders -hit the APIs & passing the path parameters and verify the status response codes

MockMvcResultMatchers – get the response content & matches with expected content

Types of Authentication

A servlet-based web application can choose from the following types of authentication, from least secure to most:

- Basic authentication
- Form-based authentication
- Digest authentication
- SSL and client certificate authentication

"Authentication" and "Authorization". Authentication can be defined as the process of verifying someone's identity by using pre-required details (Commonly username and password). Authorization is the process of allowing an authenticated user to access a specified resource (eg:-right to access a file).

HTTP Basic Authentication

- One solution is that of **HTTP Basic Authentication**. In this approach, an HTTP user agent simply provides a **username**and **password** to prove their authentication.

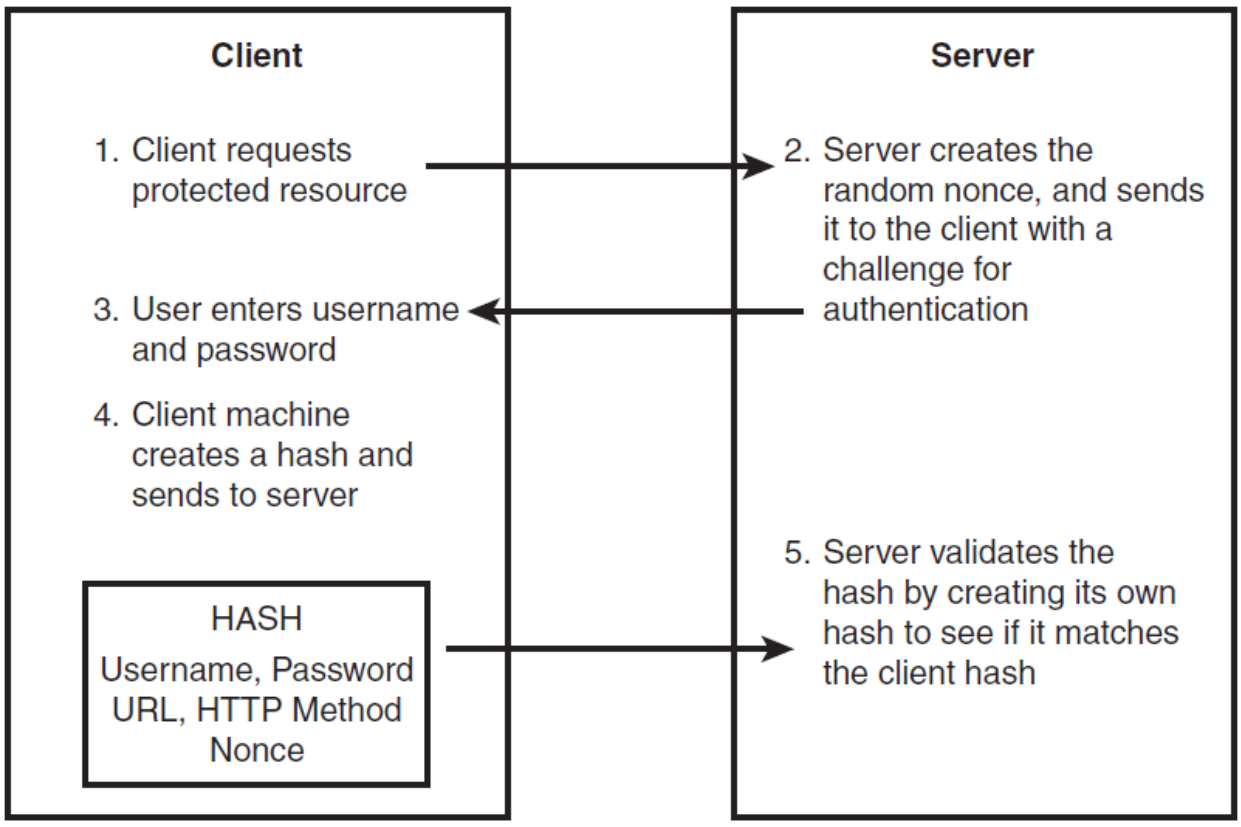
- This approach does not require cookies, session IDs, login pages, and other such specialty solutions, and because it uses the **HTTP header** itself, there's no need to handshakes or other complex response systems.

- HTTP is not encrypted in any way. It is encapsulated in base64, and is often erroneously proclaimed as encrypted due to this

Digest authentication

The difference between digest authentication and basic authentication is that in digest authentication, the username and password are never sent over the wire. Instead, a hash is created made up of the following pieces of information:

- The username
- The password
- The URL
- The randomly generated string (the nonce)
- The HTTP method being used



API Keys : for Developer Quickstart

- To access Bitbucket in Hygieia, we will generate API key, and we will place that key in properties file.

- API Keys can be used as Basic HTTP Authentication credentials and provide a substitute for the account's actual username and password.

- The best thing about an API key is its simplicity. You merely log in to a service, find your API key (often in the settings screen), and copy it to use in an application, test in the browser, or use with one of these API request tools

- Typically, an API key gives full access to every operation an API can perform, including writing new data or deleting existing data. If you use the same API key in multiple apps, a broken app could destroy your users' data without an easy way to stop just that one app.

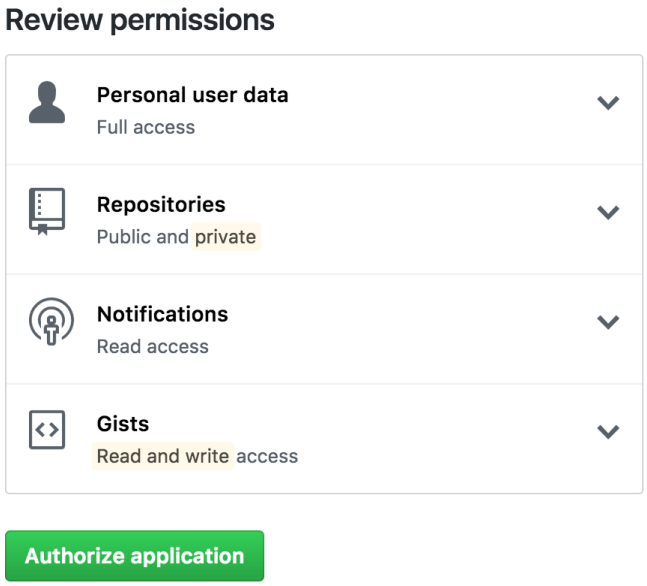
- Many API keys are sent in the query string as part of the URL, which makes it easier to discover for someone who should not have access to it. A better option is to put the API key in the

Authorization header. In fact, that's the proposed standard:
Authorization: Apikey 1234567890abcdef

OAuth Tokens: Great for Accessing User Data

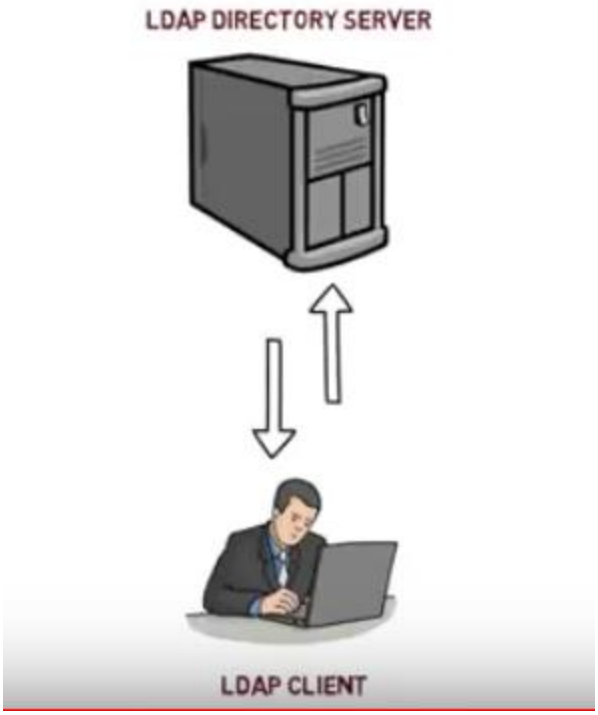
- OAuth is the answer to accessing user data with APIs.

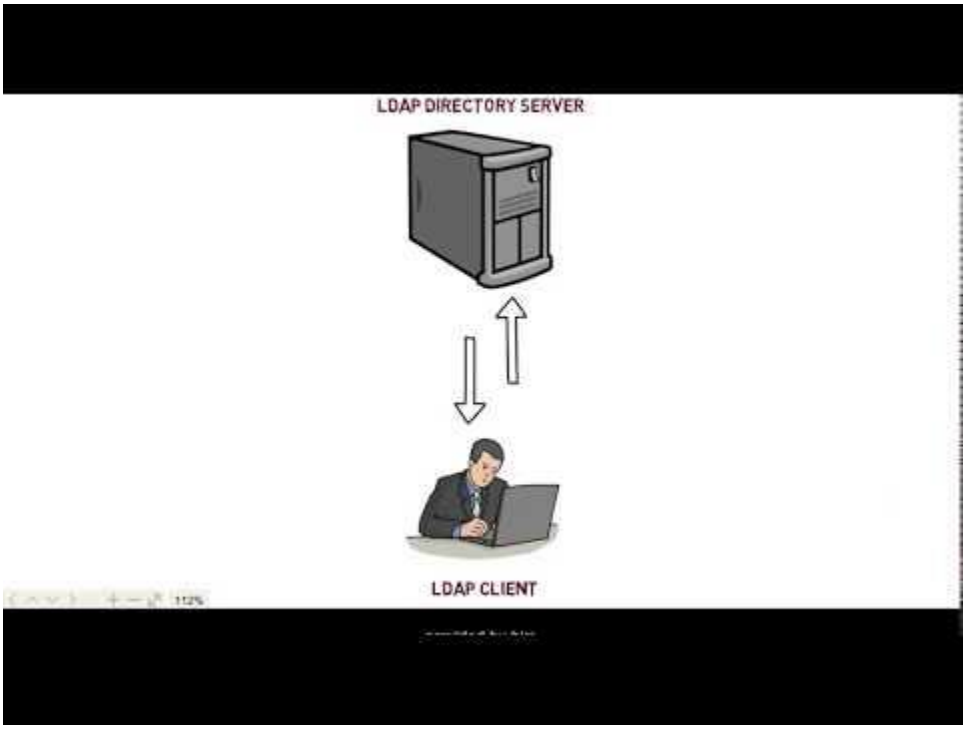
- users simply click a button to allow an application to access their accounts.



LDAP


LDAP
RESOURCES
FREQUENT READS
RARE WRITES
LDAP SERVER
LDAP CLIENT





https://www.youtube.com/watch?v=lp5z8HQGAH8

```
import javax.naming.*;
import javax.naming.directory.*;
```

```
import java.util.Hashtable;

class Simple {
    public static void main(String[] args) {
        Hashtable authEnv = new Hashtable(11);
        String userName = "johnlennon";
        String passWord = "sushi974";
        String base = "ou=People,dc=example,dc=com";
        String dn = "uid=" + userName + "," + base;
        String ldapURL = "ldap://ldap.example.com:389";

        authEnv.put(Context.INITIAL_CONTEXT_FACTORY,"com.sun.jndi.ldap.LdapCtxFactory");
           authEnv.put(Context.PROVIDER_URL, ldapURL);
           authEnv.put(Context.SECURITY_AUTHENTICATION, "simple");
           authEnv.put(Context.SECURITY_PRINCIPAL, dn);
           authEnv.put(Context.SECURITY_CREDENTIALS, passWord);

        try {
            DirContext authContext = new InitialDirContext(authEnv);
            System.out.println("Authentication Success!");
        } catch (AuthenticationException authEx) {
            System.out.println("Authentication failed!");
        } catch (NamingException namEx) {
            System.out.println("Something went wrong!");
            namEx.printStackTrace();
        }
    }
}
```

SSH – Only for LINUX Server / CommadLine(git) related Access

- SSH, or secure shell, is an encrypted protocol used to administer and communicate with servers. When working with a Linux server, chances are, you will spend most of your time in a terminal session connected to your server through SSH.

- An SSH server can authenticate clients using a variety of different methods. The most basic of these is password authentication, which is easy to use, but not the most secure.

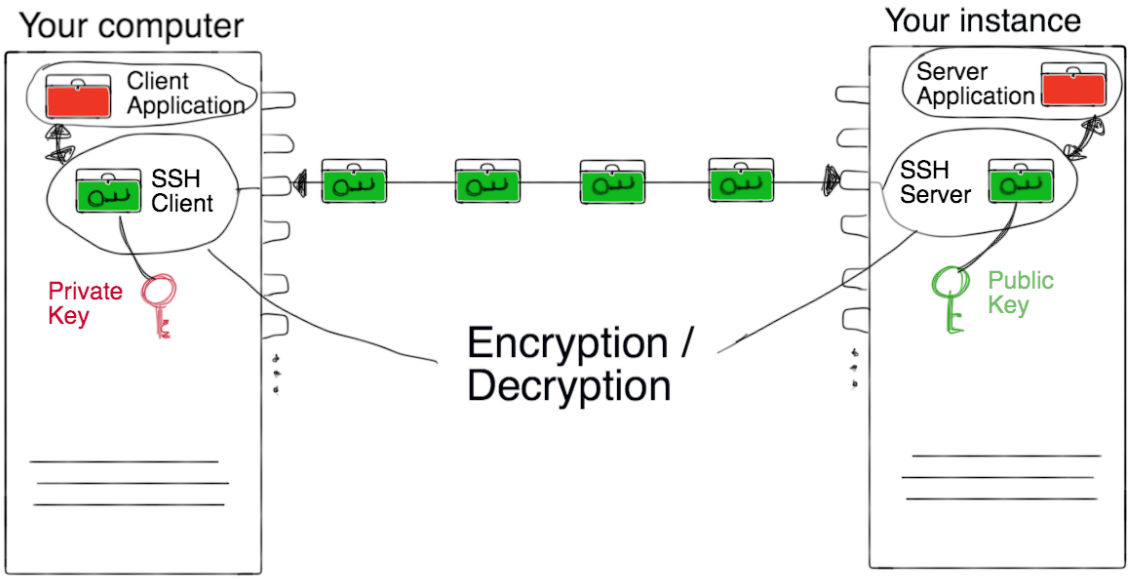
- SSH key pairs are two cryptographically secure keys that can be used to authenticate a client to an SSH server. Each key pair consists of a public key and a private key.

- The **private key** is retained by the client and should be kept absolutely secret. Any compromise of the private key will allow the attacker to log into servers that are configured with the associated public key without additional authentication. As an additional precaution, the key can be encrypted on disk with a passphrase.

- The associated public key can be shared freely without any negative consequences. The public key can be used to encrypt messages that only the private key can decrypt. This property is employed as a way of authenticating using the key pair.

- The public key is uploaded to a remote server that you want to be able to log into with SSH. The key is added to a special file within the user account you will be logging into called ~/.ssh/authorized_keys.

- When a client attempts to authenticate using SSH keys, the server can test the client on whether they are in possession of the private key. If the client can prove that it owns the private key, a shell session is spawned or the requested command is executed.



Base64 – not Authentication

represent binary data in an ASCII string format

Each Base64 digit represents exactly 6 bits of data.

Steps:

- take three ASCII numbers 155, 162, and 233
- Convert into binary stream formate 100110111010001011101001
- groupings of six characters: 100110 111010 001011 101001.
- The binary string 100110 converts to the decimal number 38: 0*2^01 + 1*2^1 + 1*2^2 + 0*2^3 + 0*2^4 + 1*2^5 = 0+2+4+0+0+32.
- Base64 6-bit values 38, 58, 11 and 41.
- Using the Base64 conversion table:

- 38 is m
- 58 is 6
- 11 is L
- 41 is p

Help.html

StudentApp Document

StudentApp + AngularJs+ CurdRepository+MockMVC References

CURD + POSTMAN

https://www.callicoder.com/spring-boot-rest-api-tutorial-with-mysql-jpa-hibernate/

AngularJS+Templates

https://examples.javacodegeeks.com/enterprise-java/spring/boot/spring-boot-and-angularjs-integration-tutorial/

Spring+AngularJS Controllers

https://java2blog.com/spring-boot-angularjs-example/

SpringBoot MockMVC Tutorial

- Mokito : https://howtodoinjava.com/spring-boot2/spring-boot-mockito-junit-example/
- MOCK MVC : https://howtodoinjava.com/spring-boot2/spring-boot-mockmvc-example/

StudentApp MongoRepository

Example App : https://www.journaldev.com/18156/spring-boot-mongodb

1. Add Maven Dependency: spring-boot-starter-data-mongodb

2.Create Sudent Collection

```
>use student
>db.createCollection("student");
>db.student.insert(
   {
      sno: 501,
      name: "Satya Kaveti",
      city:"Vijayawada",
      marks:508
   }
)
```

3.in **application.propertis** , add Mongo DB details

```
spring.data.mongodb.database=student
spring.data.mongodb.port=27017
spring.data.mongodb.host=localhost
```

2.Create **StudentModel**.java with **@Document & Id** annotations

3.create **StudentMongoRepository** extends **MongoRepository**

4.Create **StudentMongoController.java**

How can Spring Boot work without driver configuration?

Spring Boot gives you defaults on all things, the default in database is **H2**, so when you want to change this and use any other database you must define the connection attributes in the **application.properties** file.

- H2 is one of the popular in memory databases. Spring Boot has default integration for H2
- is live only during the time of execution of the application, not for real world applications
- The h2-*.jar is just an engine (the code) of the database. It is read-only and it does not store any information. The data in H2 can be stored either in memory or on disk in a specified file. You are actually specifying one:

In our Application we are using MySQL, so we provided details with out Driver detilas

```
spring.datasource.url = jdbc:mysql://localhost:3306/student?useSSL=false
spring.datasource.username = root
spring.datasource.password = root
```

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

```
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
```

- **@EnableAutoConfiguration** : This annotation tells Spring to automatically configure your application based on the dependencies that you have added in the **pom.xml** file.

- For example, If **spring-data-jpa** or **spring-jdbc** is in the classpath, then it automatically tries to configure a **DataSource** by reading the database properties from **application.properties** file.

- So,in above we just have to add the configuration and Spring Boot will take care of the rest.

- In the above properties file, the last two properties are for hibernate. Spring Boot uses Hibernate as the default JPA implementation.

- If you remove mysql-connector-java from pom.xml, SpringBoot unable to find the MySQL related java classes in insert application.propeties values to create datasource, it will throw below error.

```
Failed to bind properties under '' to com.zaxxer.hikari.HikariDataSource:
    Property: driverclassname
    Value: com.mysql.cj.jdbc.Driver
    Origin: "driverClassName" from property source "source"
Reason: Failed to load driver class com.mysql.cj.jdbc.Driver in either of HikariConfig
class loader or Thread context classloader

Action:
Update your application's configuration
```

So finally, SpringBoot uses Drivers & all normal stuff though @EnableAutoConfiguration to do the job

Spring Boot with multiple databases

In Our StudentApp we have multiple Databases.

1.if we use only one Database :MySQL we can use default `spring.data.*`properties. Spring @EnableAutoConfiguration will create Datasource by reading these properties

```
spring.datasource.url = jdbc:mysql://localhost:3306/student?useSSL=false
spring.datasource.username = root
spring.datasource.password = root
```

2.if we use two databases :MySQL, Mongo we can use default `spring.data.*`properties of therir respective databases. Spring @EnableAutoConfiguration will create two different Datasource by reading these properties

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

```
spring.data.mongodb.host=localhost
```

3.If we use two DB's with our own Config properties we need to Ovveride DataSource Manually

```
#---------- MySQL -------------
own.spring.mysql.datasource.url = jdbc:mysql://localhost:3306/student?useSSL=false
own.spring.mysql.datasource.username = root
own.spring.mysql.datasource.password = root

#---------- MongoDB -------------
own.spring.mongo.datasource.database=student
own.spring.mongo.datasource.port=27017
own.spring.mongo.datasource.host=localhost
```

Because we want the Spring Boot autoconfiguration to pick up those different properties (and actually instantiate two different DataSources), we need to instantiate our DataSource beans manually in a configuration class

```
@Configuration
public class MultipleDataSourceConfiguration {

    @Bean
    @Primary
    @ConfigurationProperties(prefix="own.spring.mysql.datasource")
    public DataSource primaryDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean
    @ConfigurationProperties(prefix="own.spring.mongo.datasource")
    public DataSource secondaryDataSource() {
        return DataSourceBuilder.create().build();
    }
}
```

- https://medium.com/@joeclever/using-multiple-datasources-with-spring-boot-and-spring-data-6430b00c02e7
- http://www.java2novice.com/spring-boot/configure-multiple-datasources/
- https://www.infoq.com/articles/Multiple-Databases-with-Spring-Boot

@Conditional & @Profilers

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

For example, you may want to register a DataSource bean pointing to the **dev** database abd different **production database** while running in production.

To address this problem, Spring 3.1 introduced the concept of **Profiles**. When you run the application you can activate the desired profiles ,and only those beans of that profiles will be registered.

```
@Configuration
public class AppConfig
{
 @Bean
```

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

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

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

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

How EnableAutoConfiguration implemented?

auto-configuration is implemented with standard `@Configuration` classes.

Additional `@Conditional` annotations are used to constrain when the auto-configuration should apply.

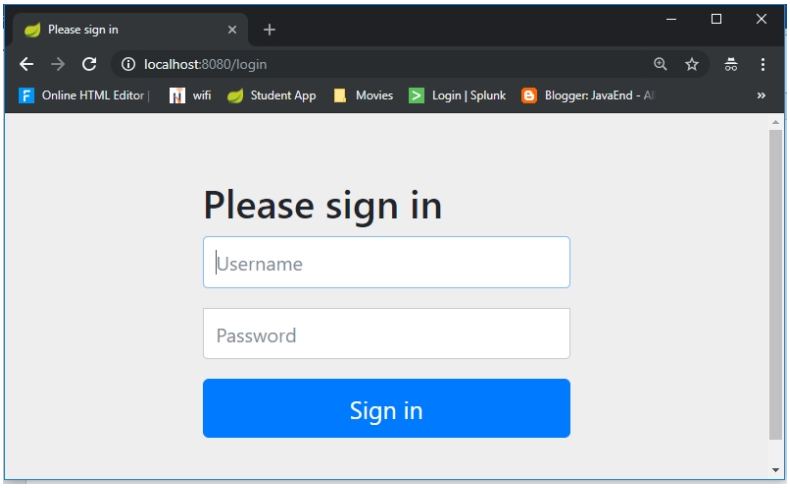
SpringBoot Security

`spring-boot-starter-security`: take care of all the required dependencies related to spring security.

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

This will include the *SecurityAutoConfiguration* class – containing the initial/default security configuration.

Just Run the project & see the magic



We never created this login form, but from where it came from?

SpringSecurity default comes with login page & you can login with generated password which is already printed in the console

```
Using generated security password: 8b4667a4-cc3a-47fd-b51f-b6f5e83745df
Def.user name is : user
```

You can change the password by providing a `security.user.password`. This and other useful properties are externalized via `SecurityProperties` (properties prefix "security").

```
security.user.name=user
security.user.name=password
security.basic.enabled=true
```

To discard the security auto-configuration and add our own configuration, we need to exclude the ***SecurityAutoConfiguration*** class.

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

Or by adding some configuration into the *application.properties* file:

```
spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration
```

If we disabling security auto-configuration, we need to provide our own configuration, by `extends WebSecurityConfigurerAdapter`

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

```
        @Override
        public void configure(HttpSecurity http) throws Exception {
                // It allows configuring web based security for specific http requests
                http
        .authorizeRequests()
            .anyRequest().authenticated()
            .and()
        .formLogin()
            .and()
        .httpBasic();


        /* ============ Custom login Page URL ==========
        http
          .authorizeRequests()
          .antMatchers("/admin/**").hasRole("ADMIN")
          .antMatchers("/anonymous*").anonymous()
          .antMatchers("/login*").permitAll()
          .anyRequest().authenticated()
          .and()
          .formLogin()
          .loginPage("/login.html")
          .loginProcessingUrl("/perform_login")
          .defaultSuccessUrl("/homepage.html", true)
          //.failureUrl("/login.html?error=true")
          .failureHandler(authenticationFailureHandler())
          .and()
          .logout()
          .logoutUrl("/perform_logout")
          .deleteCookies("JSESSIONID")
          .logoutSuccessHandler(logoutSuccessHandler());
                */
        }

    @Bean
    @Override
    public UserDetailsService userDetailsService() {
        UserDetails user =
             User.withDefaultPasswordEncoder()
                .username("user")
                .password("user")
                .roles("USER")
                .build();

        return new InMemoryUserDetailsManager(user);
    }

}
```

Let's summarize what we did in order to add Spring Boot Security to his web app. To secure his web app,

- we added Spring Boot Security to the classpath.
- Once it was in the classpath, Spring Boot Security was enabled by default.
- Then customized the security by extending `WebSecurityConfigurerAdapter` and added his own `configure` and `userDetailsService` implementation.
- http://localhost:8080/login?logout
- https://docs.spring.io/spring-security/site/docs/current/guides/html5/form-javaconfig.html
- https://examples.javacodegeeks.com/enterprise-java/spring/boot/spring-boot-security-example/

SpringBoot AOP

Actually we don't have Spring AOP starter in Springboot, but we can integrate using traditional Spring Framework

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>5.0.1.RELEASE</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.8.12</version>
  <scope>compile</scope>
</dependency>
```

Applications are generally developed with multiple layers. A typical Java application has

- Web Layer - Exposing the services to outside world using REST or a web application
- Business Layer - Business Logic
- Data Layer - Persistence Logic

While the responsibilities of each of these layers are different, there are a few common aspects that apply to all layers

- Logging
- Security

In this example we are implementing Logging – we are trying to Log the all the Application flow like, what controller is calling, what method is executing on performing certain operation.

```
package app.aop;

@Aspect
@Configuration
public class StudentAOP {

private Logger logger = LoggerFactory.getLogger(this.getClass());

        //What kind of method calls I would intercept
        //execution(* PACKAGE.*.*(..))
        //Weaving & Weaver
        @Before("execution(* app.repository.*.*(..))")
        public void before(JoinPoint joinPoint){
                //Advice
                logger.info(" \n \n====== @Before===== \t  app.repository");
                logger.info(" Allowed execution for {}", joinPoint);
                logger.info(" ====== @Before=====  \n \n");
        }


        @AfterReturning(value = "execution(* app.controller.*.*(..))",
                        returning = "result")
        public void afterReturning(JoinPoint joinPoint, Object result) {
                logger.info("\n \n ====== @AfterReturning ===== \t : app.controller ");
                logger.info("{} returned with value {}", joinPoint, result);
                logger.info(" ====== @AfterReturning =====  \n \n");
```

```
        }

        @After(value = "execution(* app.security.*.*(..))")
        public void after(JoinPoint joinPoint) {
                logger.info("\n \n ====== @After =====  \t : app.security");
                logger.info("after execution of {}", joinPoint);
                logger.info(" ====== @After =====  \n \n");
        }

}
```

```
====== @After =====       : app.security
2019-02-10 22:50:12.232  INFO 8152 --- [           main] dentAOP$$EnhancerBySpringCGLIB$$bb027782 : after
execution of execution(UserDetailsService app.security.SecurityConfig.userDetailsService())
2019-02-10 22:50:12.236  INFO 8152 --- [           main] dentAOP$$EnhancerBySpringCGLIB$$bb027782 :
====== @After =====

 ====== @AfterReturning =====     : app.controller
2019-02-10 22:43:58.212  INFO 2440 --- [nio-8080-exec-7] dentAOP$$EnhancerBySpringCGLIB$$45dcbfee :
execution(List app.controller.StudentMongoController.getAllStudents()) returned with value [Student
[sno=501, name=vinay, city=karnool, marks=545], Student [sno=502, name=VINOD, city=BNHGG, marks=456]]
2019-02-10 22:43:58.212  INFO 2440 --- [nio-8080-exec-7] dentAOP$$EnhancerBySpringCGLIB$$45dcbfee :
====== @AfterReturning =====


Getting all users.2019-02-10 22:43:58.248  INFO 2440 --- [nio-8080-exec-9]
dentAOP$$EnhancerBySpringCGLIB$$45dcbfee :

 ====== @AfterReturning =====     : app.controller
2019-02-10 22:43:58.248  INFO 2440 --- [nio-8080-exec-9] dentAOP$$EnhancerBySpringCGLIB$$45dcbfee :
execution(List app.controller.StudentMongoController.getAllStudents()) returned with value [Student
[sno=501, name=vinay, city=karnool, marks=545], Student [sno=502, name=VINOD, city=BNHGG, marks=456]]
2019-02-10 22:43:58.248  INFO 2440 --- [nio-8080-exec-9] dentAOP$$EnhancerBySpringCGLIB$$45dcbfee :
====== @AfterReturning =====
```

Pointcut - Pointcut is the **expression** used to define when a call to a method should be intercepted. In the above example, `"execution(* app.repository.*.*(..))"` is the pointcut.

Advice - It is the **logic that you would want to exceute** when you intercept a method. In the above example, it is the code inside the `before(JoinPoint joinPoint)`method.

Aspect – is the Class we defiend, combination of on which method **(Pointcut)** and what to do **(Advice)** is called an Aspect.

Join Point - When the code is executed and the condition for pointcut is met, the advice is executed. The Join Point is a specific execution instance of an advice.

Weaver - Weaver is the framework which implements AOP - AspectJ or Spring AOP.

- http://www.springboottutorial.com/spring-boot-and-aop-with-spring-boot-starter-aop
- https://www.javainuse.com/spring/spring-boot-aop

SpringBoot Acuator - Health check, Auditing, Metrics,Monitoring

Actuator brings production-ready features to our application.

Monitoring our app, gathering metrics, understanding traffic or the state of our database becomes trivial with this dependency.

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

Once above maven dependency is included in the POM file, **16 different actuator REST endpoints**, such as actuator, beans, dump, info, loggers, and metrics are exposed

Some of important and widely used actuator endpoints are given below:

ENDPOINT	USAGE
/env	Returns list of properties in current environment
/health	Returns application health information.
/auditevents	Returns all auto-configuration candidates and the reason why they 'were' or 'were not' applied.
/beans	Returns a complete list of all the Spring beans in your application.
/trace	Returns trace logs (by default the last 100 HTTP requests).
/dump	It performs a thread dump.
/metrics	It shows metrics information like JVM memory used, system CPU usage, open files, and much more.

You can access all avaible endpoint by this URL: **http://localhost:8080/actuator**

```
{
        "_links": {
                "self": {
                        "href": "http://localhost:8080/actuator",
                        "templated": false
                },
                "health": {
                        "href": "http://localhost:8080/actuator/health",
                        "templated": false
                },
                "health-component-instance": {
                        "href": "http://localhost:8080/actuator/health/{component}/{instance}",
                        "templated": true
                },
                "health-component": {
                        "href": "http://localhost:8080/actuator/health/{component}",
                        "templated": true
```

```
                },
                "info": {
                        "href": "http://localhost:8080/actuator/info",
                        "templated": false
                }
        }
}
```

If you see we have only 2 endpoints showing (health, info) out of 16 endpoints

By default, all the actuator endpoints are exposed over **JMX** but only the health and info endpoints are exposed over **HTTP**.

Here is how you can expose actuator endpoints over HTTP and JMX using application properties -

Exposing Actuator endpoints over HTTP

```
# Use "*" to expose all endpoints, or a comma-separated list to expose selected ones
management.endpoints.web.exposure.include=*
management.endpoints.web.exposure.exclude=
```

Exposing Actuator endpoints over JMX

```
# Use "*" to expose all endpoints, or a comma-separated list to expose selected ones
management.endpoints.jmx.exposure.include=*
management.endpoints.jmx.exposure.exclude=
```

Securing Actuator Endpoints with Spring Security

Actuator endpoints are sensitive and must be secured from unauthorized access. you can add spring security to your application using the following dependency -

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

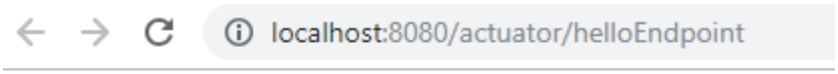
we can override the default spring security configuration and define our own access rules.

Creating a Custom Actuator Endpoint

To customize the endpoint and define your own endpoint, simply Create a classs annotate with @Endpoint URL :

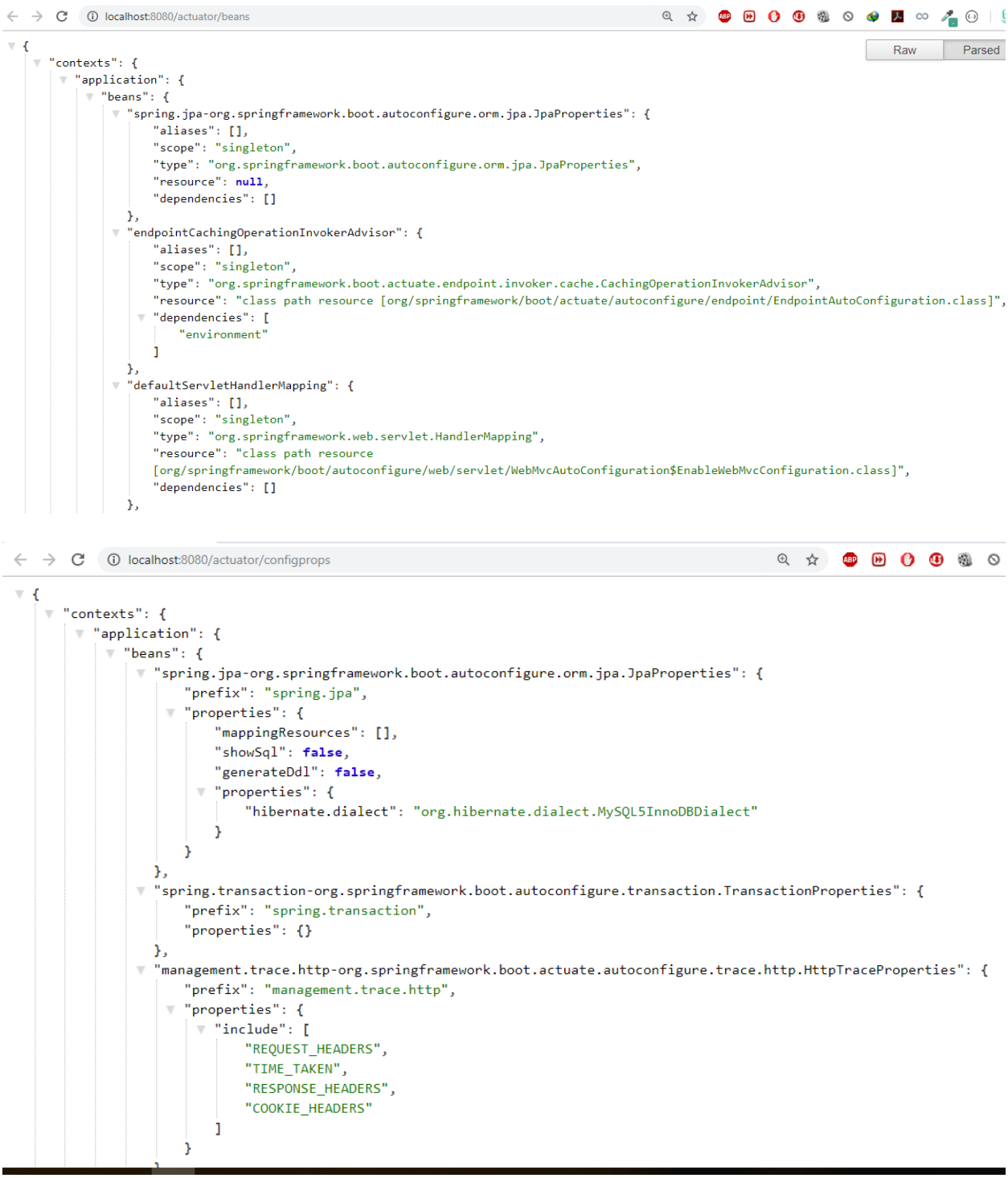
```
import org.springframework.boot.actuate.endpoint.annotation.Endpoint;
import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;
import org.springframework.stereotype.Component;

@Endpoint(id="helloEndpoint")
@Component
public class ListEndPoints {
    @ReadOperation
```

```
    public String mypoint(){
        return "Hello" ;
    }
}
```



Hello

Few more Endpoints


localhost:8080/actuator/beans
Raw
Parsed
{
"contexts": {
"application": {
"beans": {
"spring.jpa-org.springframework.boot.autoconfigure.orm.jpa.JpaProperties": {
"aliases": [],
"scope": "singleton",
"type": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties",
"resource": null,
"dependencies": []
},
"endpointCachingOperationInvokerAdvisor": {
"aliases": [],
"scope": "singleton",
"type": "org.springframework.boot.actuate.endpoint.invoker.cache.CachingOperationInvokerAdvisor",
"resource": "class path resource [org/springframework/boot/actuate/autoconfigure/endpoint/EndpointAutoConfiguration.class]",
"dependencies": [
"environment"
]
},
"defaultServletHandlerMapping": {
"aliases": [],
"scope": "singleton",
"type": "org.springframework.web.servlet.HandlerMapping",
"resource": "class path resource
[org/springframework/boot/autoconfigure/web/servlet/WebMvcAutoConfiguration$EnableWebMvcConfiguration.class]",
"dependencies": []
},
localhost:8080/actuator/configprops
{
"contexts": {
"application": {
"beans": {
"spring.jpa-org.springframework.boot.autoconfigure.orm.jpa.JpaProperties": {
"prefix": "spring.jpa",
"properties": {
"mappingResources": [],
"showSql": false,
"generateDdl": false,
"properties": {
"hibernate.dialect": "org.hibernate.dialect.MySQL5InnoDBDialect"
}
}
},
"spring.transaction-org.springframework.boot.autoconfigure.transaction.TransactionProperties": {
"prefix": "spring.transaction",
"properties": {}
},
"management.trace.http-org.springframework.boot.actuate.autoconfigure.trace.http.HttpTraceProperties": {
"prefix": "management.trace.http",
"properties": {
"include": [
"REQUEST_HEADERS",
"TIME_TAKEN",
"RESPONSE_HEADERS",
"COOKIE_HEADERS"
]
}


localhost:8080/actuator/loggers
{
"levels": […], // 6 items
"loggers": {
"ROOT": {
"configuredLevel": "INFO",
"effectiveLevel": "INFO"
},
"app": {
"configuredLevel": null,
"effectiveLevel": "INFO"
},
"app.StudentAppApplication": {
"configuredLevel": null,
"effectiveLevel": "INFO"
},
"app.aop": {
"configuredLevel": null,
"effectiveLevel": "INFO"
},
"app.aop.StudentAOP": {
"configuredLevel": null,
"effectiveLevel": "INFO"
},
"app.aop.StudentAOP$": {
"configuredLevel": null,
"effectiveLevel": "INFO"
},

https://dzone.com/articles/spring-boot-actuator-a-complete-guide

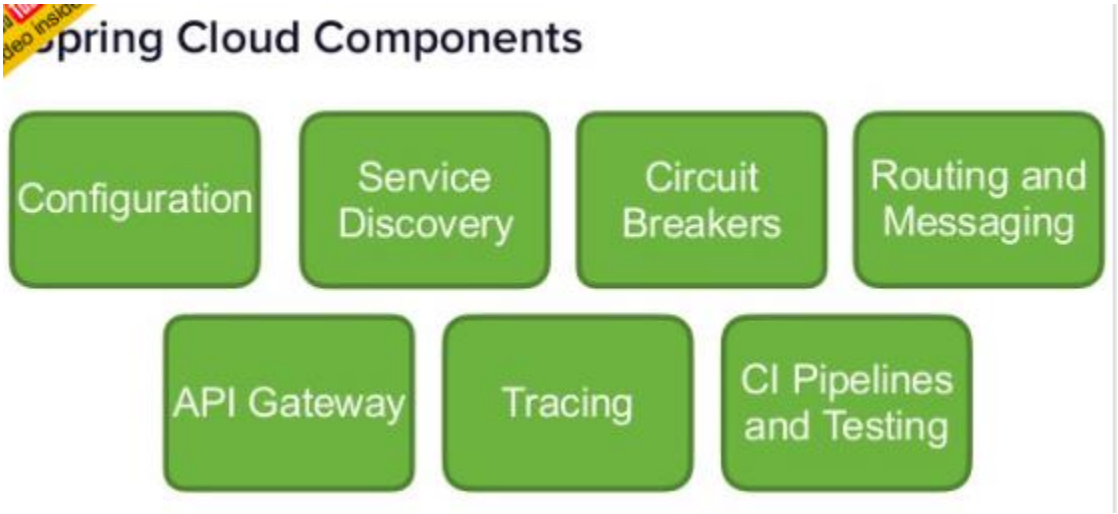
https://www.callicoder.com/spring-boot-actuator/

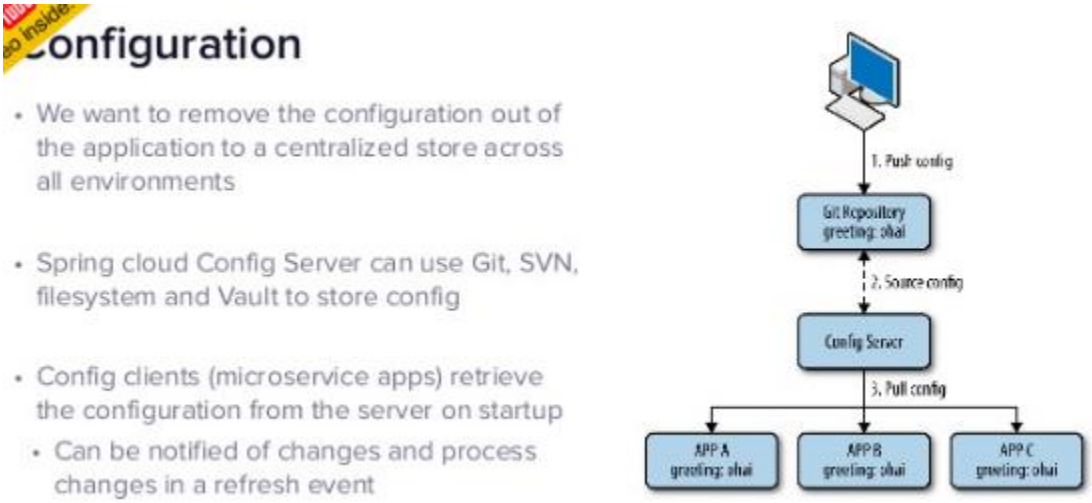
Spring cloud

Spring Cloud focuses on providing good out of box experience for typical use cases and extensibility mechanism to cover others.

- Distributed/versioned configuration
- Service registration and discovery
- Routing
- Service-to-service calls
- Load balancing
- Circuit Breakers
- Global locks
- Leadership election and cluster state
- Distributed messaging

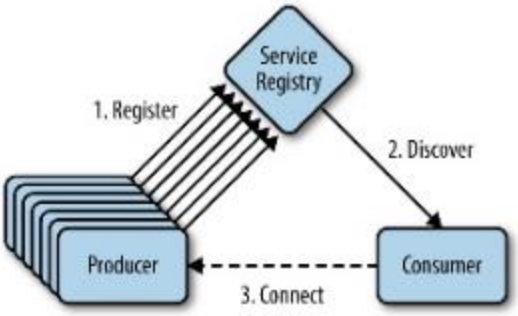
https://github.com/Debesh1234/DemoSpringBootProjects





Service Discovery

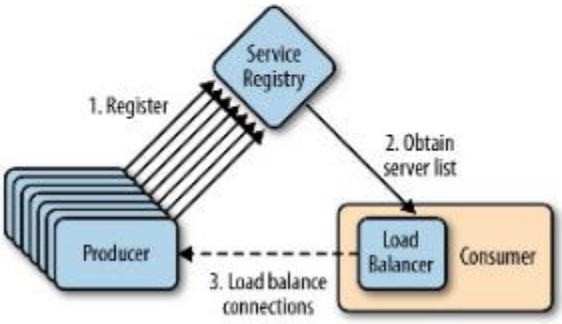
- With the dynamic nature of any cloud native application, depending on things like URLs can be problematic
- Service Discovery allows micro services to easily discover the routes to the services it needs to use
- Netflix Eureka
- Zookeeper
- Consul



Routing and Messaging

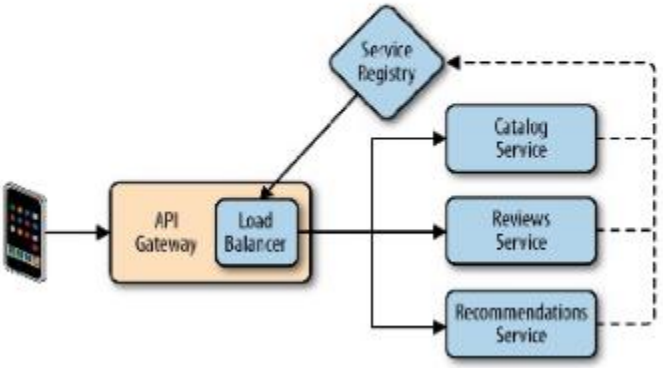
- Your cloud native app will be composed of many microservices so communication will be critical
- Spring Cloud supports communication via HTTP requests or via messaging

- Routing and Load Balancing:
  - Netflix Ribbon and Open Feign
- Messaging:
  - RabbitMQ or Kafka



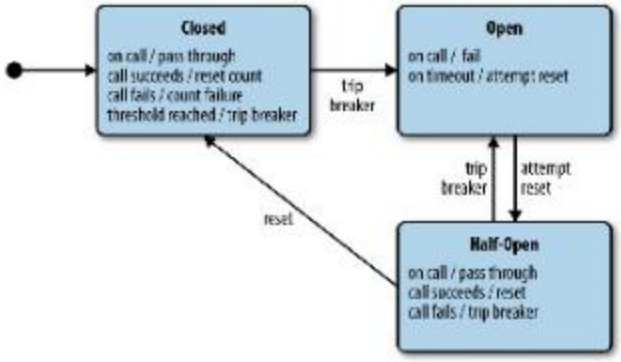
API Gateway

- API Gateways allow you to route API requests (internal or external) to the correct service
- Netflix Zuul
  - Leverages service discovery and load balancer
- Spring Cloud Gateway



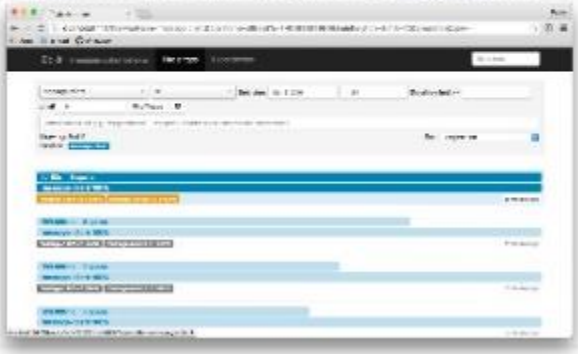
Circuit Breakers

- Failure is inevitable, but your user's don't need to know
- Circuit breakers can help an application function in the face of failure
- Netflix Hystrix



Tracing

- A single request to get data from your application may result in an exponentially larger number of requests to various microservices
- Tracing these requests through the application is critical when debugging issues
- Spring Cloud Sleuth and Zipkin



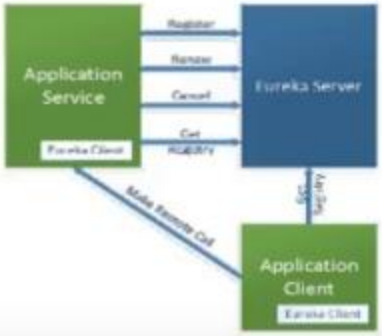
CI Pipelines and Testing

- Building, testing, and deploying the various services is critical to having a successful cloud native application
- Spring Cloud Pipelines is an opinionated pipeline for Jenkins or Concourse that will automatically create pipelines for your apps
- Spring Cloud Contract allows you to accurately mock dependencies between services using published contracts

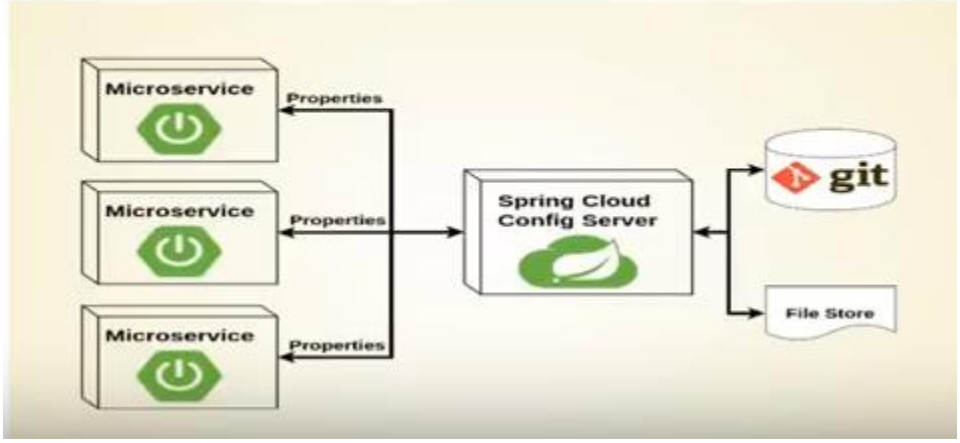
SpringOne Platform by Pivotal

Eureka Service Discovery

- Each service has unique serviced
- Service uses Eureka Client to interact with Eureka Server:
  - Register: serviceId, host, port
  - Renew: using heartbeats to check status
  - Get Registry: return list host:port of services by serviceId



CONFIG SERVER

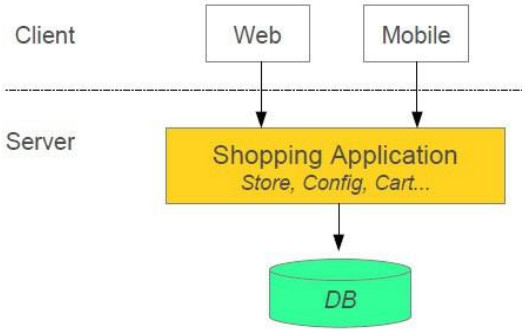


Read it Enough : http://www.optisolbusiness.com/Micro-Services-Architecture-Spring-Boot-and-Netflix-Infrastructure.pdf

SpringBoot MicroServices

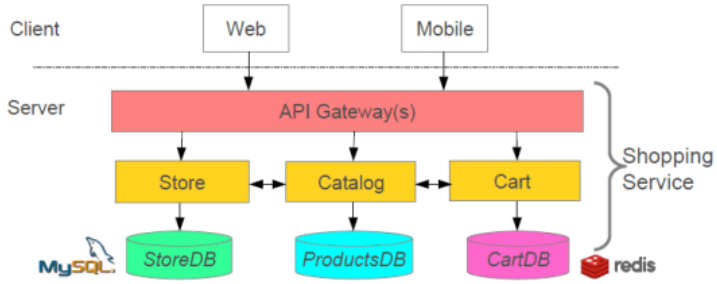
Microservices allows us **to break our large system into the number of independent** collaborating processes.

Shopping system without Microservices (Monolith architecture)



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.



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

Microservices Challenges

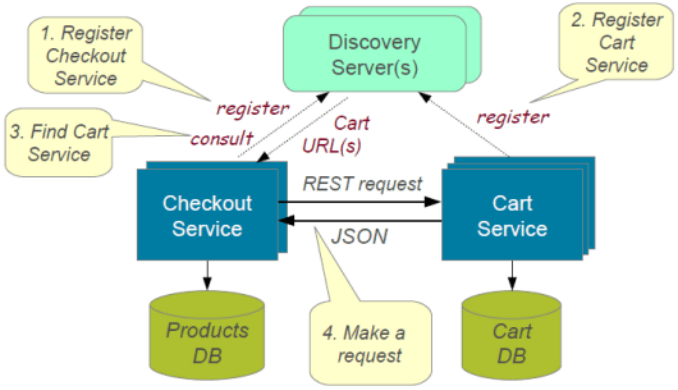
- Distributed System so hard to debug and trace the issues
- Greater need for an end to end testing
- Required cultural changes in across teams like Dev and Ops working together even in the same team.

Microservices Tooling Supports

1. Use Spring for creating Microservices

- Setup new service by using Spring Boot
- Expose resources via a RestController
- Consume remote services using RestTemplat

2.Service Discovery



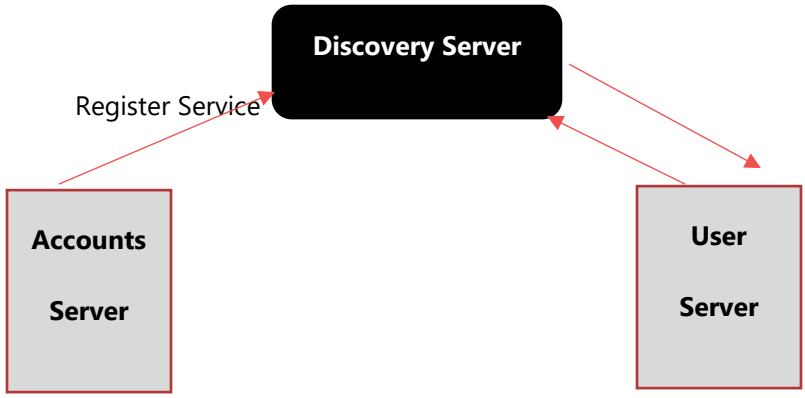
To build a simple microservices system following steps required

1. **Creating Discovery Service (Creating Eureka Discovery Service)**
2. **Creating MicroService (the Producer),** Register itself with Discovery Service with logical service.
3. **Create Microservice Consumers** find Service registered with Discovery ServiceDiscovery client using a smart **RestTemplate** to find microservice.

Example:

In this Example we have 3 Micro services should communicate each other.

Each micrpservice could have seprate Server & Separate DB



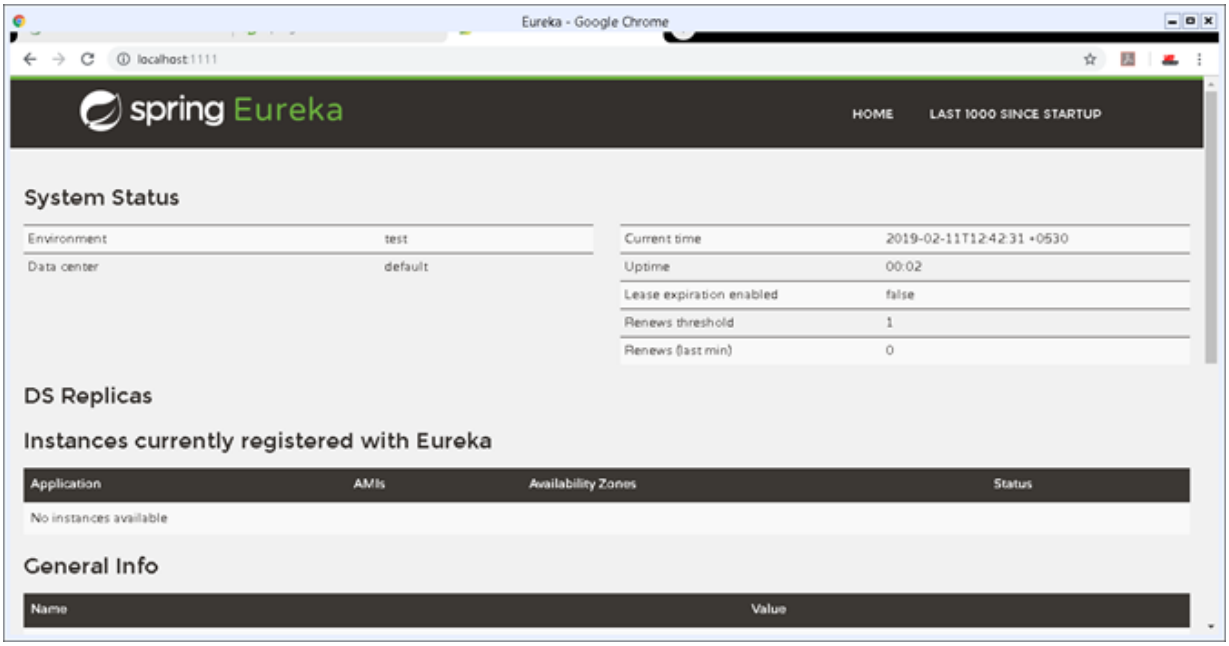
1**. Discovery Microservice server** – To register microservice URLS

https://github.com/dineshonjava/discovery-microservice-server

@EnableEurekaServer –EurekaServer Acts as a Discovery Server. To Make our class as Discovery server use @EnableEurekaServer on the top of Spring Application class

```
@SpringBootApplication
@EnableEurekaServer
public class DiscoveryMicroserviceServerApplication {

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

Run this Eureka Server application with right click and run as Spring Boot Application and open in browser **http://localhost:1111/**



2. Accounts Microservice Server – It conatins Bank Account Deatils

https://github.com/dineshonjava/accounts-microservice-server

Account.java – for Account details

```
public class Account implements Serializable{

	private static final long serialVersionUID = 1L;
	private Long amount;
	private String number;
	private String name;
//Setter and getters
}
```

AccountController – all these Controller urs are Registers with Discovery Service

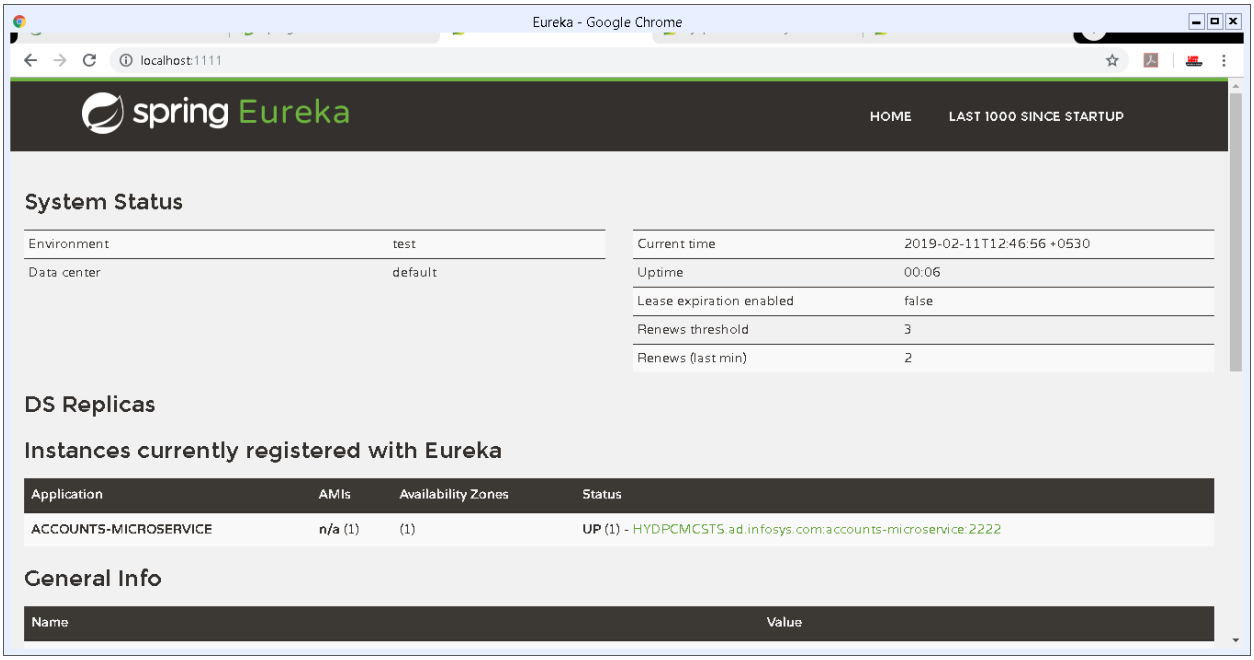
```
@RestController
public class AccountController {

	protected Logger logger = Logger
			.getLogger(AccountController.class.getName());

	@Autowired
	AccountRepository accountRepository;

	@RequestMapping("/accounts")
	public Account[] all() {
		logger.info("accounts-microservice all() invoked");
		List<Account> accounts = accountRepository.getAllAccounts();
		logger.info("accounts-microservice all() found: " + accounts.size());
		return accounts.toArray(new Account[accounts.size()]);
	}

	@RequestMapping("/accounts/{id}")
	public Account byId(@PathVariable("id") String id) {
		logger.info("accounts-microservice byId() invoked: " + id);
		Account account = accountRepository.getAccount(id);
		logger.info("accounts-microservice byId() found: " + account);
		return account;
	}
}
```

To make all our controller URLS register with Discovery server, we need to annotate our main Spring Application class with ***@EnableDiscoveryClient***

```
@SpringBootApplication
@EnableDiscoveryClient
public class AccountsMicroserviceServerApplication {

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

}
```

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/** .

Now one Service registered to the Eureka registered instances with Service Name "**ACCOUNT-MICROSERVICE**" as below



3. Webclient Mroservice server – User will ask for Specific Account details with AccNo

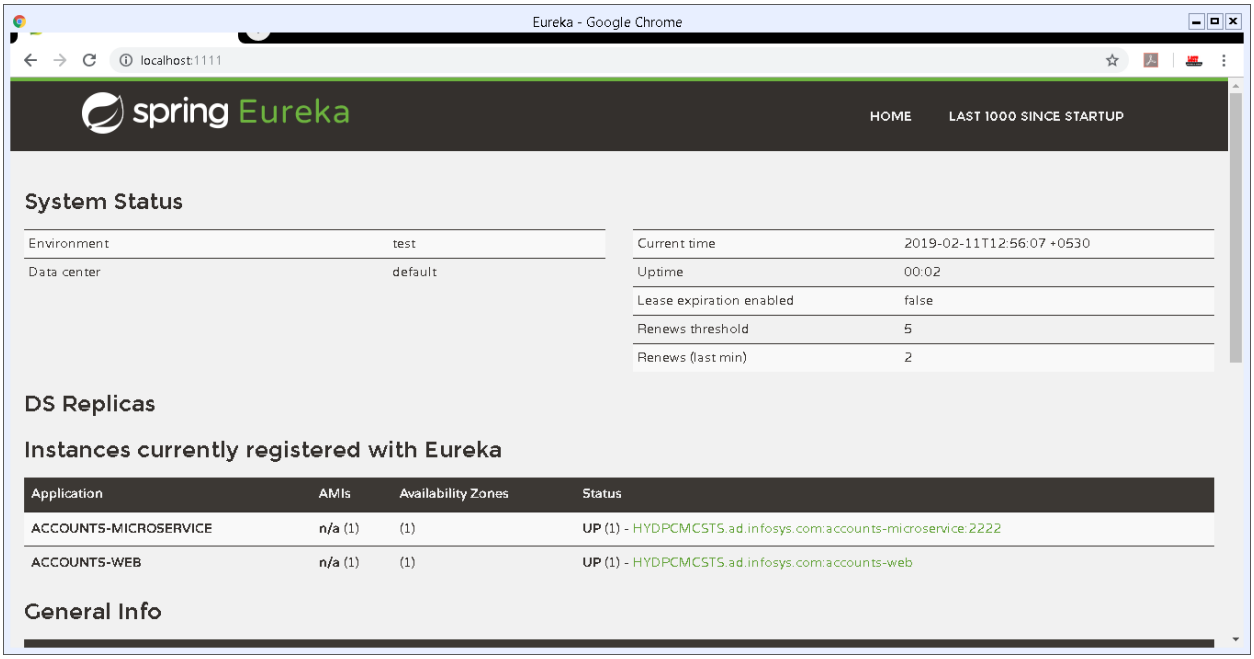
https://github.com/dineshonjava/webclient-microservice-server

Create users to find the Producer Service registered with Discovery Service by adding ***@EnableDiscoveryClient***. This annotation also allows us to query Discovery server to find microservices

```
@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);
	}
}
```

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**

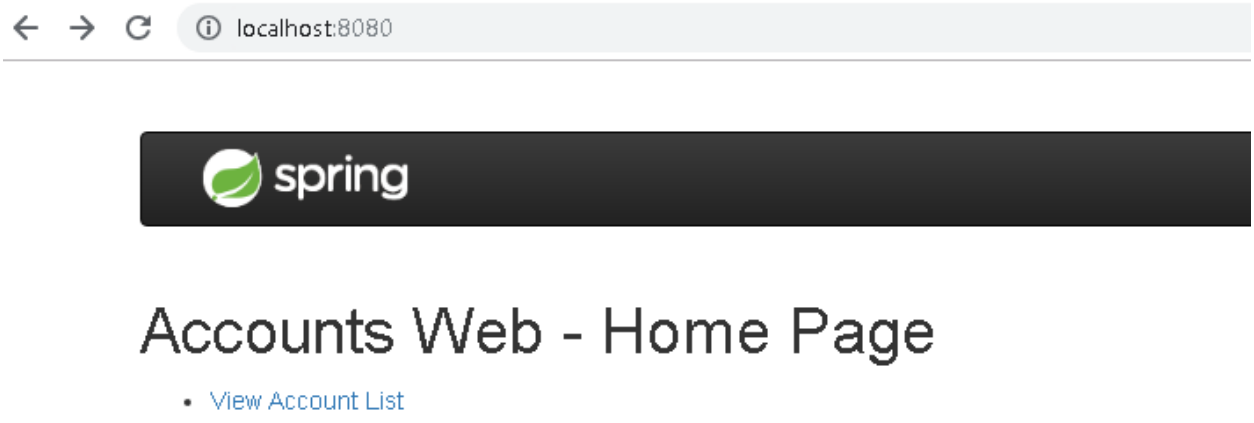
Now one more Service registered to the Eureka registered instances with Service Name "**ACCOUNTS-WEB**" as below

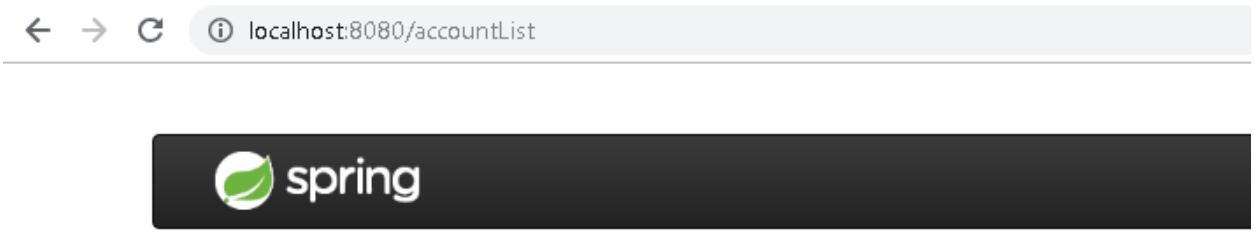


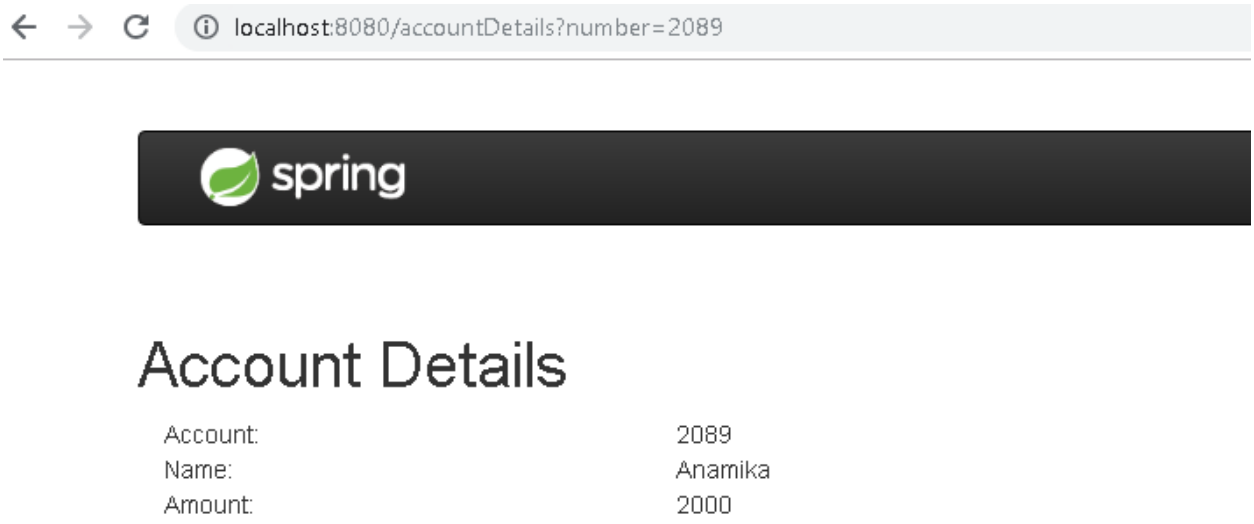
4.Access URL Through Web Application

We already created a Web Application in **Webclient Mroservice server** to call registred Discovery Server URLs.which is running on

http://localhost:8080/







https://www.dineshonjava.com/microservices-with-spring-boot/

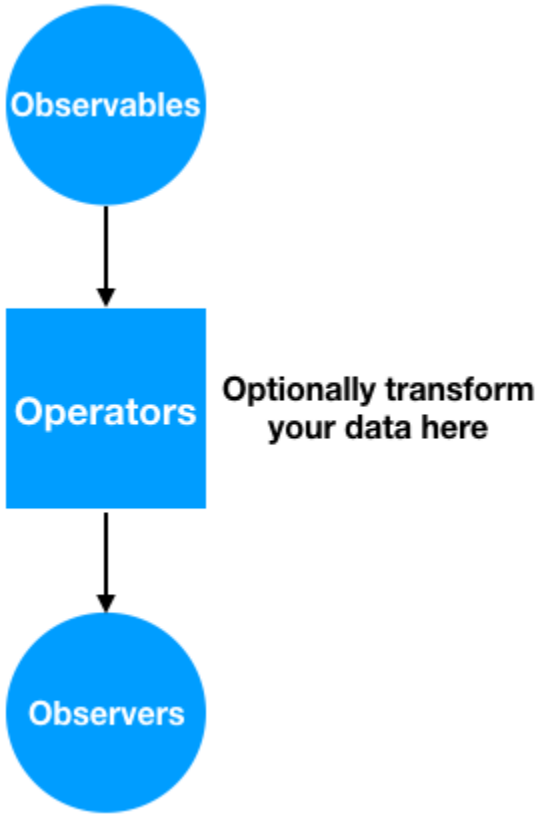
https://spring.io/blog/2015/07/14/microservices-with-spring

Reactive JavaRx

- Reactive comes from the word react, which means to **react to changes** in the state instead of actually doing the state change.
- The reactive model listens to changes in the event and runs the relevant code accordingly.
- **observer/subscriber** attached listening to the stream would receive the data.

The basic building blocks of RxJava are:

- **Observables**: That emits data streams
- **Observers and Subscribers**: That consume the data stream. The only difference between an Observer and a Subscriber is that a Subscriber class has the methods to unsubscribe/resubscribe independently without the need of the observerable methods.
- **Operators**: That transform the data stream



https://cdn.journaldev.com/wp-content/uploads/2018/02/rxjava-basics-flow.png

SpringBoot – Reactive Programming

Spring WebFlux framework is part of Spring 5 and provides reactive programming support for web applications

Spring WebFlux internally uses **Project Reactor** and its publisher implementations – ***Flux*** and ***Mono***.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
    <version>2.0.3.RELEASE</version>
</dependency>
```

We'll now build a very simple Reactive REST *EmployeeManagement* application – using Spring WebFlux:

- We'll use a simple domain model – *Employee* with an *id* and a *name* field
- We'll build REST APIs for publishing and retrieve Single as well as Collection *Employee*resources using *RestController* and *WebClient*
- And we will also be creating a secured reactive endpoint using WebFlux and Spring Security

```
@RestController
@RequestMapping("/employees")
public class EmployeeReactiveController {

    private final EmployeeRepository employeeRepository;

@GetMapping("/{id}")
private Mono<Employee> getEmployeeById(@PathVariable String id) {
    return employeeRepository.findEmployeeById(id);
}

@GetMapping
private Flux<Employee> getAllEmployees() {
    return employeeRepository.findAllEmployees();
}

}
```

Reactive Web Client

WebClient introduced in Spring 5 is a non-blocking client with support for Reactive Streams.

On the client side, we use *WebClient* to retrieve data from our endpoints created in*EmployeeController*.

```
public class EmployeeWebClient {

    WebClient client = WebClient.create("http://localhost:8080");

  Mono<Employee> employeeMono = client.get()
```

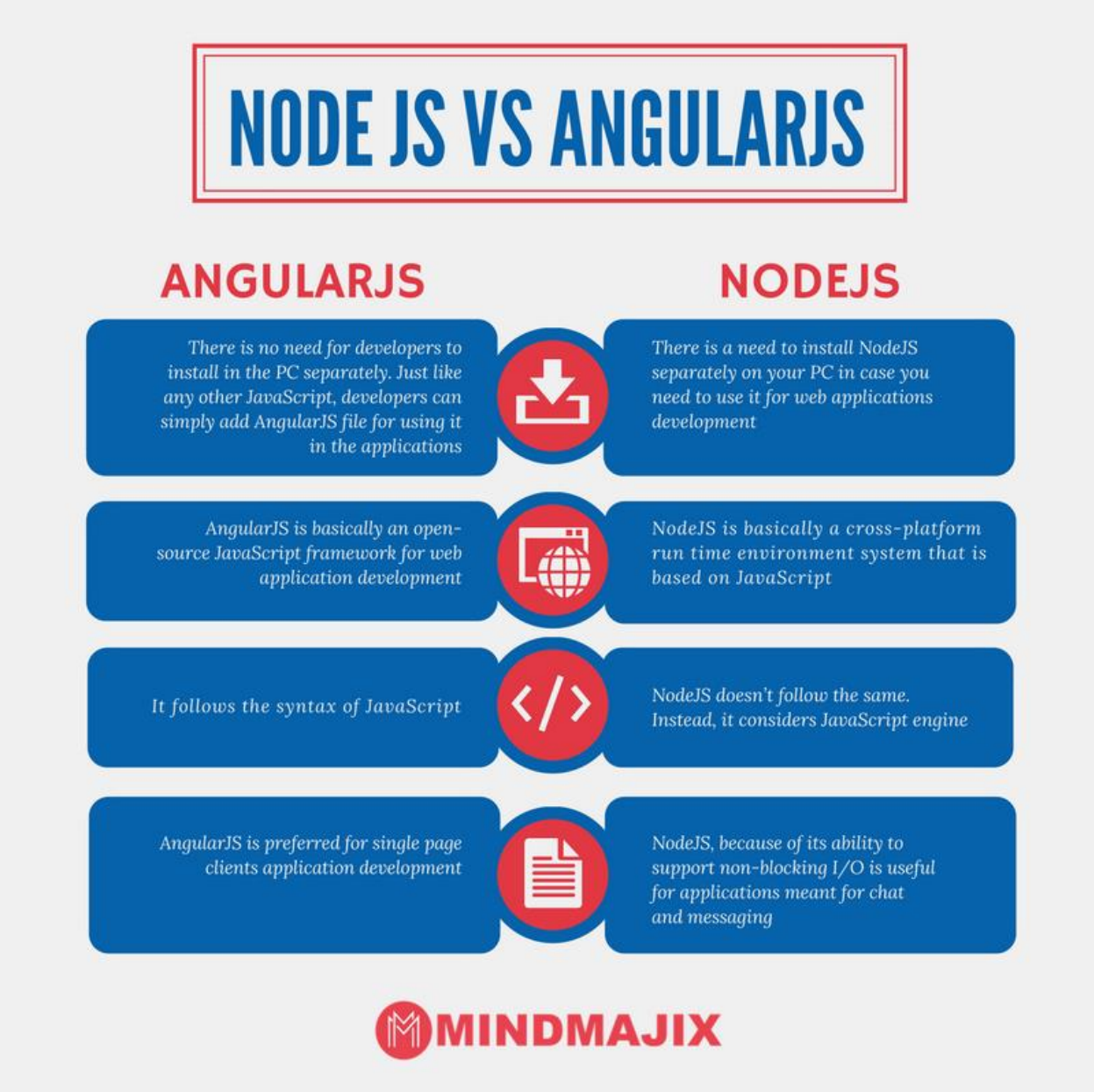
```
  .uri("/employees/{id}", "1")
  .retrieve()
  .bodyToMono(Employee.class);

employeeMono.subscribe(System.out::println);


Flux<Employee> employeeFlux = client.get()
  .uri("/employees")
  .retrieve()
  .bodyToFlux(Employee.class);

employeeFlux.subscribe(System.out::println);

}
```

https://www.baeldung.com/spring-webflux

https://www.journaldev.com/20763/spring-webflux-reactive-programming

AngularJs(Angular 1) vs Angular (Angular 2)

Technology	React	AngularJS	Angular 2
Author	Facebook community	Google	Google
Type	Open source JS library	Fully-featured MVC framework	Fully-featured MVC framework
Toolchain	High	Low	High
Language	JSX	JavaScript, HTML	TypeScript
Learning Curve	Low	High	Medium
Packaging	Strong	Weak	Medium
Rendering	Server Side	Client Side	Server Side
App Architecture	None, combined with Flux	MVC	Component-based
Data Binding	Uni-directional	Bi-directional	Bi-directional
DOM	Virtual DOM	Regular DOM	Regular DOM
Latest Version	15.4.0 (November 2016)	1.6.0	2.2.0 (November 2016)



What exactly node.js is?

Is it a web server or a programming language for server-side scripts?

- So here's how it is, how it's always been: a browser sends a request to a website. The site's server receives the request, tracks down the requested file, performs any database queries as needed, and sends a response to the browser. In traditional web servers, such as Apache, each request causes the server to create a new system process to handle that request

- Now think about what that means for a traditional web server like Apache. For each and every user connected to the site, your server has to keep a connection open. Each connection requires a process, and each of those processes will spend most of its time either sitting idle (consuming memory) or waiting on a database query to complete. This means that it's hard to scale up to high numbers of connections without grinding to a near halt and using up all your resources.

- So what's the solution? Here's where some of that jargon from before comes into play: specifically **non-blocking** and **event-driven**

- Think of **a non-blocking** server as a loop: it just keeps going round and round. A request comes in, the loop grabs it, passes it along to some other process (like a database query), sets up a callback, and keeps going round, ready for the next request. It doesn't just sit there, waiting for the database to come back with the requested info.

- If the database query comes back — fine, we'll deal with that the same way: throw a response back to the client and keep looping around. There's theoretically no limit on how many database queries you can be waiting on, or how many clients have open requests, because you're not spending any time waiting for them. You deal with them all in their own time

- **event-driven means**: the server only reacts when an event occurs. That could be a request, a file being loaded, or a query being executed — it really doesn't matter.

How to host node.js applications?

You need to Host AWS or Google Cloud or any other cloud platform because it needs Node.js to be installed.

Npm, bower packges

Let's understand by Example

I used the complete **MEAN** stack for this series

MEAN is a set of Open Source components that together, provide an end-to-end framework for building dynamic web applications;

- **M**ongoDB : Document database – used by your back-end application to store its data as JSON (JavaScript Object Notation) documents
- **E**xpress (sometimes referred to as Express.js): Back-end web application framework running on top of Node.js
- **A**ngular (formerly Angular.js): Front-end web app framework; runs your JavaScript code in the user's browser, allowing your application UI to be dynamic
- **N**ode.js : JavaScript runtime environment – lets you implement your application back-end in JavaScript

1.create **package.json** to install some Node packages.(like maven, here we can see **express.js** dependency)

```
//package.json
{
  "name": "node-rest-auth",
  "main": "server.js",
  "dependencies": {
    "bcrypt": "^0.8.5",
    "body-parser": "~1.9.2",
    "express": "~4.9.8",
    "jwt-simple": "^0.3.1",
    "mongoose": "~4.2.4",
    "morgan": "~1.5.0",
    "passport": "^0.3.0",
    "passport-jwt": "^1.2.1"
  }
}
```

To install pacakges Run

```
npm install
```

This will install all our modules to **node_modules/**.

We can also install one by one without package.json as below, it will get latest version of it

```
npm install mongojs
npm install express
```

2.create **server.js**, here we import all the needed elements and create our server with url localhost:9090

```
var express     = require('express');
var app         = express();
var bodyParser  = require('body-parser');
var morgan      = require('morgan');
var mongoose    = require('mongoose');
var passport = require('passport');
var config      = require('./config/database'); // get db config file
var User        = require('./app/models/user'); // get the mongoose model
var port        = process.env.PORT || 9090;
var jwt         = require('jwt-simple');

// get our request parameters
app.use(bodyParser.urlencoded({ extended: false }));
```

```
app.use(bodyParser.json());

// log to console
app.use(morgan('dev'));

// Use the passport package in our application
app.use(passport.initialize());

// demo Route (GET http://localhost:9090)
app.get('/', function(req, res) {
  res.send('Hello! The API is at http://localhost:' + port + '/api');
});

// Start the server
app.listen(port);
console.log('There will be dragons: http://localhost:' + port);
```

3.config/database.js

```
module.exports = {
  'secret': 'devdacticIsAwesome',
  'database': 'mongodb://localhost/node-rest-auth'
};
```

4. user model for our user authentication

```
//app/models/user.js
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var bcrypt = require('bcrypt');

// Thanks to http://blog.matoski.com/articles/jwt-express-node-mongoose/

// set up a mongoose model
var UserSchema = new Schema({
  name: {
        type: String,
        unique: true,
        required: true
    },
  password: {
        type: String,
        required: true
    }
});

UserSchema.pre('save', function (next) {
    var user = this;
    if (this.isModified('password') || this.isNew) {
        bcrypt.genSalt(10, function (err, salt) {
            if (err) {
                return next(err);
            }
            bcrypt.hash(user.password, salt, function (err, hash) {
                if (err) {
                    return next(err);
                }
                user.password = hash;
                next();
            });
        });
    } else {
        return next();
    }
});

UserSchema.methods.comparePassword = function (passw, cb) {
```

```
    bcrypt.compare(passw, this.password, function (err, isMatch) {
        if (err) {
            return cb(err);
        }
        cb(null, isMatch);
    });
};

module.exports = mongoose.model('User', UserSchema);
```

Now the basics are set up, and you can start our server from now on just with

```
node server.js
```

https://devdactic.com/restful-api-user-authentication-1/

Now NPM vs Bower

Npm and Bower are both **dependency management tools**. But the main difference between both is

- **npm is used for installing Node js modules**
- **bower is used for managing front end components like html, css, js etc**

running `bower install` will fetch the package and put it in `/vendor` directory,
running `npm install` it will fetch it and put it into `/node_modules` directory.

Grunt is quite different from Npm and Bower. Grunt is a javascript task runner tool. You can do a lot of things using grunt which you had to do manually otherwise

There are grunt plugins for **compilation, uglifying your javascript, copy files/folders, minifying javascript etc.**

References

OnlineHTML - https://www.froala.com/online-html-editor