

Spring Boot *again*

Latest Revision
20-Sep-2021

Written By
Satya Kaveti

1. INTRODUCTION	4
CREATING SPRING BOOT APPLICATION.....	5
2. SPRING BOOT EXAMPLES	7
1.SPRING BOOT WITH MAVEN AND ECLIPSE EXAMPLE.....	7
2.SPRING BOOT WITH SPRING INITIALIZR EXAMPLE.....	9
3. SPRING BOOT –CONFIGURATIONS.....	15
HOW TO CHANGE SPRING BOOT BANNER TEXT.....	15
APPLICATION.PROPERTIES	15
4. SPRING BOOT –MVC.....	16
SPRING MVC SUMMARY	16
SPRING BOOT MVC EXAMPLE – USING THEAMLEAF.....	18
SPRINGBOOT – MVC USING STANDARD JSP PAGES.....	24
SPRING BOOT –RESTFUL WEB SERVICE EXAMPLE.....	27
SPRING BOOT - REST API EXCEPTION HANDLING (@CONTROLLERADVICE)	29
5. SPRING BOOT –DATABASE	32
SPRING BOOT –JDBC EXAMPLE	32
SPRING BOOT –JPA EXAMPLE.....	35
JPAREPOSITORY	40
SPRING BOOT –MONGODB REST EXAMPLE	43
6. SPRINGBOOT - SECURITY.....	47
SPRING SECURITY ARCHITECTURE	48
SPRING SECURITY AUTHENTICATION TYPES.....	52
1.SPRING SECURITY – QUICK START EXAMPLE.....	53
2.SPRING SECURITY – BASIC & IN-MEMORY AUTHENTICATION EXAMPLE	55
3.SPRING SECURITY – CUSTOM LOGIN FORM.....	56
4.SPRING SECURITY – JDBC AUTHENTICATION.....	59
5. SPRING SECURITY – LDAP AUTHENTICATION	60
6. SPRING SECURITY – JWT AUTHENTICATION	64
7. SPRING SECURITY – OAUTH 2.0 SSO AUTHENTICATION	66
8. SPRING SECURITY – SUMMARY	70
9.SPRING SECURITY – REF.	70
7. SPRING BOOT – ADVANCED	71
SPRING BOOT – LOGGING.....	75
SWITCH BETWEEN ENVIRONMENTS.....	75
SPRING BOOT – ASYNCHRONOUS IMPLEMENTATION.....	76
SPRINGBOOT REACTIVE WEB.....	78
SPRING BOOT - DEVTOOLS	79
SPRINGBOOT ACUATOR - HEALTH CHECK, AUDITING, METRICS, MONITORING.....	133
SPRINGBOOT – PROJECT LOMBOK	79
SPRING VAULT	80
DIFFERENCE BETWEEN @COMPONENT & @AUTOWIRE.....	80
DIFFERENCE BETWEEN @COMPONENTSCAN & @ENABLE AUTOCONFIGURATION.....	81
8. SPRING CLOUD – MICROSERVICES	92
1. SPRINGCLOUD – EXAMPLE	94
2. SPRING CLOUD CONFIG SERVER – TO STORE OUR PROPERTIES.....	99

2.AZURE KEY VALUT.....	102
3. SPRING CLOUD SERVICE REGISTRY AND DISCOVERY	104
4. SPRING CLOUD LOAD BALANCING	106
5. SPRING CLOUD CIRCUIT BREAKER USING NETFLIX HYSTRIX.....	107
6. SPRING CLOUD ZUUL PROXY AS API GATEWAY	113
7. DISTRIBUTED TRACING WITH SPRING CLOUD SLEUTH(SLUUTH) AND ZIPKIN.....	116
8.SPRING CLOUD STREAMS – KAFKA & ZOOKEPER.....	119
SUMMARY OF MICROSERVICES	126
9. SPRING DATA REDIS	81
10. SPRINGBOOT – SWAGGER	128
11. SPRINGBOOT – SESSION MANAGEMENT	71
12. TYPES OF AUTHENTICATION	140
HTTP BASIC AUTHENTICATION	141
DIGEST AUTHENTICATION	141
API KEYS : FOR DEVELOPER QUICKSTART.....	142
OAUTH TOKENS: GREAT FOR ACCESSING USER DATA	143
LDAP.....	145
SSH – ONLY FOR LINUX SERVER / COMMADLINE(GIT) RELATED ACCESS	146
BASE64 – NOT AUTHENTICATION.....	147
REFERENCES	148
ERRORS & SOLUTIONS	149
REACT APPLICATION	150
CREATE REACT APP & INTEGRATE AZURE AD.....	150
CONFIGURE ROLES & GROUPS OIN AZURE AD	151
SPRINGBOOT JWT AUTHENTICATION.....	152
REF.	168

1. Introduction

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

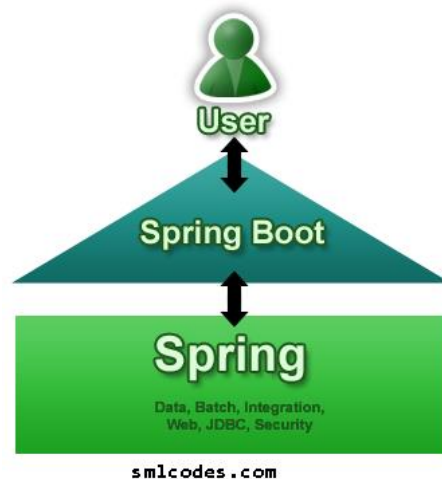
Spring vs SpringBoot

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 takes advantage of Spring Boot starters and auto-configuration. Spring Boot is not a framework, it is a way to easily 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.



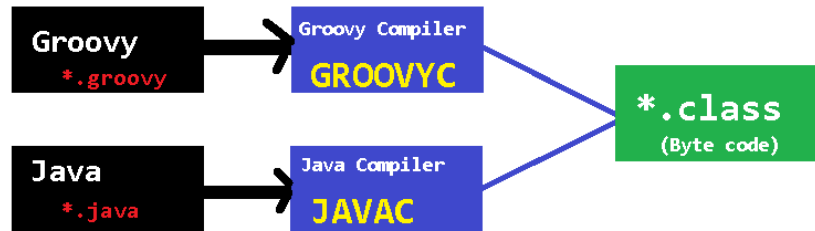
Spring Boot Features

- 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

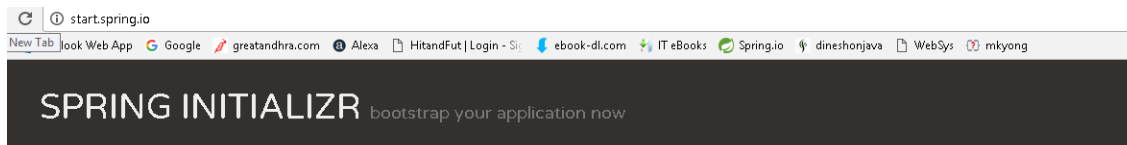
Add maven dependencies in pom.xml & do maven build

```
<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.



Generate a with Spring Boot

Project Metadata

Artifact coordinates

Group

Artifact

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

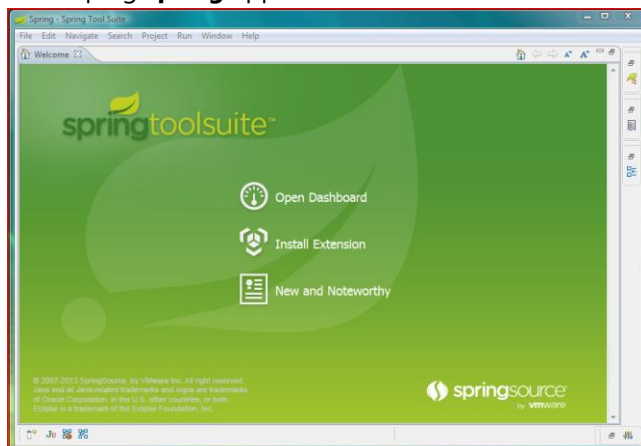
Selected Dependencies

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 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

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>

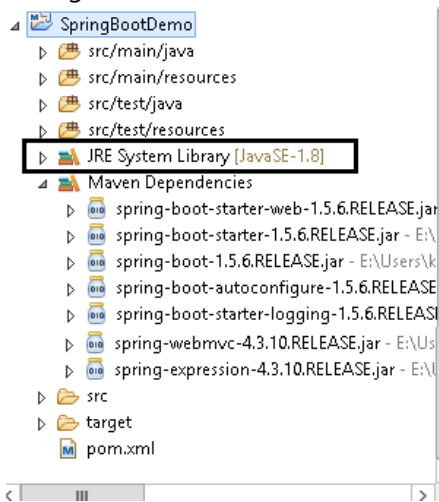
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>

  <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 package `com.smlcodes.app.SpringBootApplication.java`.

```
package com.smlcodes.app;

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

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

- **@SpringBootApplication** annotation, is the starting point for our Spring Boot application
- **SpringApplication.run(SpringBootApplication.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 annotated with **@SpringBootApplication** and bootstrap it

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

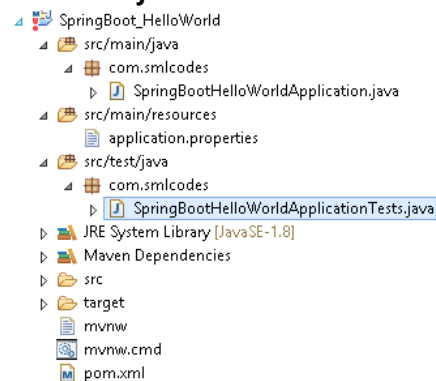
Generate a with Spring Boot

Project Metadata
Artifact coordinates
Group
Artifact

Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies
Selected Dependencies

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("Hello World, Spring Boot!!!!");
    }
}
```

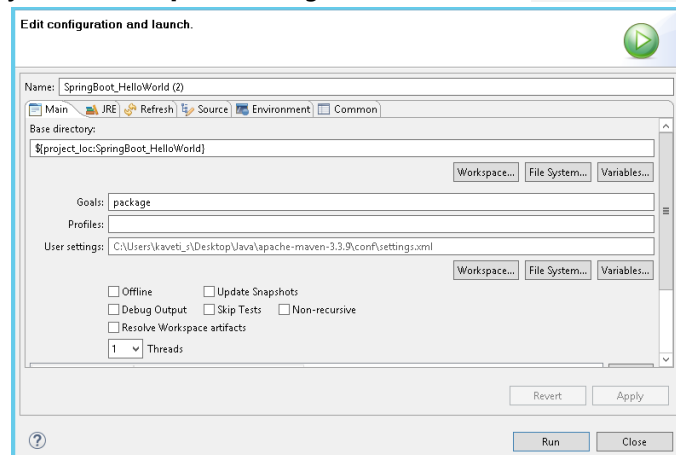
7. Select the Java file and right click RunAs → Java Application

```

:: Spring Boot :: (v1.5.1.RELEASE)
2017-01-31 11:05:32.745 INFO 29596 --- [           main]
c.s.SpringBootHelloWorldApplication : Started SpringBootHelloWorldApplication in 12.855
seconds (JVM running for 16.094)
Hello World, Spring Boot!!!!

```

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_HelloWorld-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 -server.port=8081
```

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, 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() **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-<module>**, where - **<module>** 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
<code>spring-boot-starter-web-services</code>	Starter for using Spring Web Services
<code>spring-boot-starter-web</code>	Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
<code>spring-boot-starter-test</code>	Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito
<code>spring-boot-starter-jdbc</code>	Starter for using JDBC with the Tomcat JDBC connection pool
<code>spring-boot-starter-aop</code>	Aspect-oriented programming with Spring AOP and AspectJ
<code>spring-boot-starter-security</code>	Starter for using Spring Security
<code>spring-boot-starter-data-jpa</code>	Starter for using Spring Data JPA with Hibernate
<code>spring-boot-starter</code>	Core starter, including auto-configuration support, logging, YAML

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 Configuration class – which configures the application beans, external beans by adding jar 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 SpringConfiguration class
2. **@EnableAutoConfiguration** –
 - 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 disable 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

```
@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 (`BeanFactory / ApplicationContext`)
- Register a `CommandLinePropertySource` to expose command line arguments as Spring properties(`--server.port = 9090`)
- 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);  
}
```

4. Embaded Servlet containers

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+
Undertow 1.3	3.1	Java 7+

Actually, Spring boot by default comes up with the embedded tomcat server once we add "**spring-boot-starter-web**" dependency. To exclude tomcat from spring boot, just need to add an additional block to the Spring Boot Starter dependency.

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-web</artifactId>  
  <exclusions>  
    <exclusion>  
      <groupId>org.springframework.boot</groupId>  
      <artifactId>spring-boot-starter-tomcat</artifactId>  
    </exclusion>  
  </exclusions>  
</dependency>
```

When declaring the `@SpringBootApplication` annotation, there is a way to exclude all servers and do consider the spring boot application like the web.

```
@SpringBootApplication(exclude = {EmbeddedServletContainerAutoConfiguration.class,  
WebMvcAutoConfiguration.class})
```

To Add Jetty Server or other server in Spring Boot

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

5.Spring Boot Profiles (@Profile Annotation)

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>
```



```

    }
    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.

Spring MVC Summary

Spring MVC

```

@Controller
@RequestMapping("student")

@RequestMapping(value = "/add" method=RequestMethod.GET)
@RequestMapping(value = "/add" method=RequestMethod.POST)

//student/fetch/100/satya
@RequestMapping(value="/fetch/{sno}/{name}")
public String getInfo(@PathVariable("sno") String sno, (@PathVariable("sno") String n ) {
}

//student/fetch? sno=100&name=satya
@RequestMapping(value="/fetch")
public String getBoth(@RequestParam("id") String id, @RequestParam("name") String n) {
}

//Form Data
@RequestMapping(value = "/addEmployee", method = RequestMethod.POST)
public String submit( @ModelAttribute("employee") Employee employee ) {
}

```

WebServices

<pre>@Path("/student")</pre>
<pre>/student/add @Path("/add")</pre>
<pre>@GET @Path("/usa") @Produces("text/html") @POST @Path("/usa") @Produces("text/html")</pre>
<pre>@Path("/{rollno}/{name}/{address}") @Produces("text/html") public Response get(@PathParam("rollno") String rollno,@PathParam("name") String name, @PathParam("address") String address) { } students?rollno=1218&name=SATYA KAVETI&address=VIJAYAWADA @GET @Produces("text/html") public Response get (@QueryParam("rollno") String rollno,@QueryParam("name") String name, @QueryParam("address") String address) { } //DefaultValue @GET @Produces("text/html") public Response getResultByPassingValue(@DefaultValue("1000") @QueryParam("rollno") String rollno,@DefaultValue("XXXX") @QueryParam("name") String name, @DefaultValue("XXXX") @QueryParam("address") String address) { customers;custNo=100;custName=Satya @GET @Produces("text/html") public Response getResultByPassingValue(@MatrixParam("rollno") String rollno, @MatrixParam("name") String name, @MatrixParam("address") String address) {} //Form @POST @Path("/registerStudent") @Produces("text/html") public Response getResultByPassingValue(@FormParam("rollno") String rollno, @FormParam("name") String name, @FormParam("address") String address) {} // HeaderParam @GET @Path("/headerparam") public Response getHeader(@HeaderParam("user-agent") String userAgent, @HeaderParam("Accept") String accept, @HeaderParam("Accept-Encoding") String encoding, @HeaderParam("Accept-Language") String lang) { //Context @Path("Context ") public Response getHttpheaders(@Context HttpHeaders headers){ String output = "<h1>@@Context Example - HTTP headers</h1>"; output = output+"
ALL headers -- "+ headers.getRequestHeaders().toString(); output = output+"
All Cookies -- "+ headers.getCookies().values(); return Response.status(200).entity(output).build();</pre>

Spring 4

Spring4–Introduced GetMapping/PostMapping/XXMapping in place of RequestMethod.POST/RequestMethod.GET etc.

Spring3 -`@RequestMapping(value="/user/create", method=RequestMethod.POST)`

```
Spring4 -
@GetMapping("/students/{sno}")
    public ResponseEntity getStudent(@PathVariable("sno") int sno) {
}

@PostMapping(value = "/students")
    public ResponseEntity createStudent(@RequestBody Student student) {
}

@DeleteMapping("/students/{sno}")
    public ResponseEntity deleteStudent(@PathVariable int sno) {
}

public ResponseEntity<Student> createUser(@RequestBody User user, UriComponentsBuilder ub){
}
```

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 – Using ThymLeaf

Thymeleaf Quick Tutorial

Thymeleaf is a modern server-side Java template engine for both web and standalone environments.

We can use Thymeleaf directly in `.html` files. By just placing below line to activate Thymeleaf in our HTML.

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

If you want to use Thymeleaf elements in our program, we have use `th:<element>` Tag in our HTML Tags

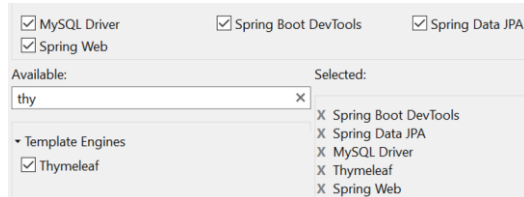
```
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod: ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price, 1, 2)}">0.99</td>
    </tr>
  </tbody>
</table>
```

We need add **spring-boot-starter-thymeleaf** dependency in our application's pom.xml file to use Thymeleaf in our Application

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

Example – Employee CRUD operations

Create Project: [Ref](#) :



Build with maven & Open Eclipse. If we see the Folder Structure

- **All static content like CSS, JS files will** be placed under →[resources\static](#)
- **all the result pages** must be placed under→[resources\templates](#)

This type of folder structure we will use in Thymeleaf based UI.

Static Content

By default, Spring Boot serves static content from a directory called `/static`. You can also customize the static resource locations by using the [spring.web.resources.static-locations](#) property in `application.properties` file.

Template Content (Template Engines)

Spring MVC supports a variety of templating technologies, including **Thymeleaf, FreeMarker, and JSPs**. If you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`

1. application.properties (DB Properties)

```
spring.datasource.url=jdbc:mysql://localhost:3306/webapp?useSSL=false
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.hibernate.ddl-auto = update
```

2. Employee.java entity class to map with Employee Table

```
package.springboot;

@Entity
@Table(name = "employee")
public class Employee {

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

    @Column
    private String name;

    @Column
    private String address;

    @Column
    private Double salary;

    //Setters & Getters
}
```

3. EmployeeRepository.java – for performing CRUD Operations on DB

```
package.springboot.repository;
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
}
```

4. EmployeeService.java – Service Layer

```
public interface EmployeeService {
    List<Employee> getAllEmployees();

    void saveEmployee(Employee employee);

    Employee getEmployeeById(long id);

    void deleteEmployeeById(long id);
}
```

EmployeeServiceImpl.java – Service Layer Implementation

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    EmployeeRepository repository;

    @Override
    public List<Employee> getAllEmployees() {
        return repository.findAll();
    }

    @Override
    public void saveEmployee(Employee employee) {
        Employee e = repository.save(employee);
        System.out.println(" Employee Data Saved : " + e);
    }
}
```



```

@Override
public Employee getEmployeeById(long id) {
    Employee e = repository.getById(id);
    System.out.println("Employee getEmployeeById : " + e);
    return e;
}

@Override
public void deleteEmployeeById(long id) {
    Employee e = repository.getById(id);
    repository.delete(e);
    System.out.println("Employee Deleted : ");
}
}

```

So far Good. Now come to actual Thymeleaf changes.

Model Interface

Java-5-specific interface that defines a holder for **model attributes**. Primarily designed for adding attributes to the model. Allows for accessing the overall model as a `java.util.Map`.

Display Homepage

1. Create `index.html` thymeleaf template under "`resources/templates`" folder and add below code.

```

<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
<link rel="stylesheet"
href=https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css>
</head>

<body>
<div class="container my-2">
<h1>ThemeLeaf - EmployeeApp</h1>
<div class="list-group">
<a href="#" class="list-group-item list-group-item-action">All Employees</a>
<a href="#" class="list-group-item list-group-item-action">Add Employee</a>
</div>
</div>
</body>
</html>

```

2. Create a `EmployeeController` class, and add method to display homepage

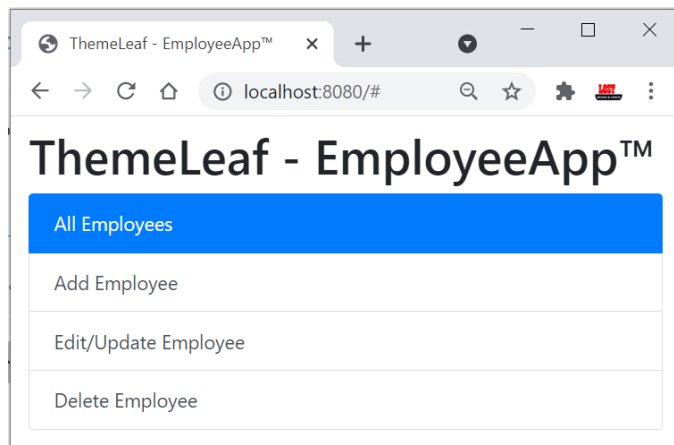
```

@Controller
public class EmployeeController {
    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/")
    public String viewHomePage(Model model) {
        System.out.println("Calling Home Page...");
        return "index";
    }
}

```

Now Run main class & open Browser



To make our application more stylish, add **CSS** and **JS** files in **src/main/resources/static** folder.

You can see Complete code [Example here \(Code Ref.\)](#)

https://gitlab.com/satyacodes/books-sync-github_new/-/tree/main/Codes/SpringBoot-Thymeleaf

If you got this Error: **Whitelabel Error page (type=Not Found, status=404)**

Move your SpringBoot main class from sub package to Root package, example from **spring.app** package to **spring** package

```
@Controller
public class EmployeeController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/")
    public String viewHomePage(Model model) {
        System.out.println("Calling Home Page...");
        return "index";
    }

    // display list of employees
    @GetMapping("/all")
    public String listEmployees(Model model) {
        model.addAttribute("listEmployees", employeeService.getAllEmployees());
        return "listEmployees";
    }

    // display list of employees
    @GetMapping("/addEmployee")
    public String addEmployee(Model model) {
        // create model attribute to bind form data
        Employee employee = new Employee();
        model.addAttribute("employee", employee);
        //An error happened during template parsing (template: "class path resource [templates/add.html]")
        return "addEmployee";
    }

    @PostMapping("/saveEmployee")
    public String saveEmployee(@ModelAttribute("employee") Employee employee) {
        // save employee to database
        employeeService.saveEmployee(employee);
        return "redirect:/all";
    }

    @GetMapping("/showEditForm/{id}")
    public String showEditForm(@PathVariable(value = "id") long id, Model model) {

        // get employee from the service
        Employee employee = employeeService.getEmployeeById(id);
        employee.setId(id);
    }
}
```

```

        // set employee as a model attribute to pre-populate the form
        model.addAttribute("employee", employee);
        return "editEmployeeForm";
    }

    @GetMapping("/viewEmployee/{id}")
    public String viewEmployee(@PathVariable(value = "id") long id, Model model) {

        // get employee from the service
        Employee employee = employeeService.getEmployeeById(id);

        // set employee as a model attribute to pre-populate the form
        model.addAttribute("employee", employee);
        System.out.println("viewEmployee : " + employee);
        return "viewEmployee";
    }

    @GetMapping("/deleteEmployee/{id}")
    public String deleteEmployee(@PathVariable(value = "id") long id) {
        // call delete employee method
        this.employeeService.deleteEmployeeById(id);
        return "redirect:/all";
    }
}

```

```

<!-- index.html -->
<a href="all" class="list-group-item list-group-item-action active"> All Employees</a>
<a href="addEmployee" class="list-group-item list-group-item-action">Add Employee</a>

```

```

<!-- addEmployee -->
<form action="#" th:action="@{/saveEmployee}" th:object="${employee}" method="POST">
    <input type="text" th:field="*{name}" placeholder="Employee Name" class="form-control mb-4 col-4">
    <input type="text" th:field="*{address}" placeholder="Employee Adress" class="form-control mb-4 col-4">
    <input type="text" th:field="*{salary}" placeholder="Employee Salary" class="form-control mb-4 col-4">
    <button type="submit" class="btn btn-info">Save</button>
</form>

<!-- editEmployee -->
<form action="#" th:action="@{/saveEmployee}" th:object="${employee}" method="POST">
    <input type="text" th:field="*{id}" class="form-control mb-4 col-4">
    <input type="text" th:field="*{name}" placeholder="Employee Name" class="form-control mb-4 col-4">
    <input type="text" th:field="*{address}" placeholder="Employee Adress" class="form-control mb-4 col-4">
    <input type="text" th:field="*{salary}" placeholder="Employee Salary" class="form-control mb-4 col-4">
    <button type="submit" class="btn btn-info">Save</button>
</form>

<!-- viewEmployee -->
ID : <span th:text="${employee.id}">
NAME : <span th:text="${employee.name}"></span>
ADDRESS : <span th:text="${employee.address}">
SALARY : <span th:text="${employee.salary}">
<a th:href="@{/all}"> Back to Employee List</a>

<!-- listEmployees -->
<table border="1" class="table table-striped table-responsive-md">
    <thead>
    <tr>
        <th># Emp. ID</th>
        <th>Name </th>
        <th>Address </th>
        <th>Salary </th>
        <th>Actions </th>
    </tr>
    </thead>
    <tbody>
    <tr th:each="employee : ${ListEmployees}">
        <td th:text="${employee.id}"></td>
        <td th:text="${employee.name}"></td>
        <td th:text="${employee.address}"></td>
        <td th:text="${employee.salary}"></td>
        <td>
            <a th:href="@{/viewEmployee/{id}(id=${employee.id})}" class="btn btn-primary">View <span>&nbsp;</span></a>
            <a th:href="@{/showEditForm/{id}(id=${employee.id})}" class="btn btn-warning"> EDIT</a> <span>&nbsp;</span>
            <a th:href="@{/deleteEmployee/{id}(id=${employee.id})}" class="btn btn-danger">Delete</a>
        </td>
    </tr>
    </tbody>
</table>

```

SpringBoot – MVC using Standard JSP Pages

Add this extra dependency in pom.xml to work with jsp pages

```
<!-- JSTL -->
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
</dependency>
<!-- Tomcat Embed -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<!-- Optional, test for static content, bootstrap CSS-->
<dependency>
  <groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>3.3.7</version>
</dependency>
```

CSS, JS, assets related files

In old Spring, to use static files, we need configure those folders paths in **SpringConfig.xml** file

```
<!-- OLD Spring - These for Static images, css calls in jsp pages -->
<mvc:annotation-driven />
<mvc:resources mapping="/assets/*" location="/assets/" />
<mvc:resources mapping="/js/*" location="/js/" />
<mvc:resources mapping="/css/*" location="/css/" />
```

But in Spring Boot, no need to declare the resource mapping like above. The resource mapping will handle automatically. We need to place CSS or Javascript, in **/src/main/resources/static/** folder.

JSP Files

JSP view files would be created inside **src/main/resources/META-INF/resources/WEB-INF/jsp/**. If you want use css, js files which are placed in static folder, just link into JSP view via

```
<link href="/css/main.css" rel="stylesheet">
```

To resolve **JSP** file's location, you can have two approaches.

1. Using **application.properties**
2. Using **Java configuration**

1.Using application.properties (Recommnded)

To Tell Spring, where JSP files located - we need to place those details in **application.properties**

```
spring.mvc.view.prefix: /WEB-INF/jsp/
spring.mvc.view.suffix: .jsp
```

2. Using Java configuration

Configure InternalResourceViewResolver to serve JSP pages using **WebMvcConfigurerAdapter**

```
package springboot.controller;
@Configuration
@EnableWebMvc
@ComponentScan
public class MVCConfiguration extends WebMvcConfigurerAdapter {
    @Override
```

```

    public void configureViewResolvers(ViewResolverRegistry registry) {
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();
        resolver.setPrefix("/WEB-INF/view/");
        resolver.setSuffix(".jsp");
        resolver.setViewClass(JstlView.class);
        registry.viewResolver(resolver);
    }
}

```

Spring Boot Application_INITIALIZER

So far we used .jar file to run our SpringBoot application. But for Web Applications recommended way is .war file Deployment. To Achieve the .war file related stuff in SpringBoot we need to use **SpringBootServletInitializer**

Our SpringBoot main class need to **extend SpringBootServletInitializer** & override **configure()** method to produce .war file. This makes use of Spring Framework's Servlet 3.0 support and allows you to configure your application when it's launched by the servlet container.

```

@SpringBootApplication
public class SpringBootAppApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(SpringBootAppApplication.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(SpringBootAppApplication.class, args);
        System.out.println(" Hello, Again");
    }

}

```

So finally,

- **Controller** classes created inside **src/main/java**
- **JSP pages** created inside **src/main/webapp/WEB-INF/view**
- **CSS and JS** files created inside **src/main/resources/static**
- **application.properties** created inside **src/main/resources**
- **Application.java** is a launch file for Spring Boot created inside **src/main/java**

Ref : <https://mmafrar.medium.com/implementing-spring-boot-mvc-crud-operations-with-jpa-and-jsp-4dfa1882b4a3>

pom.xml

```

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

```

```

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
    </dependency>
    <dependency>
        <groupId>org.webjars</groupId>
        <artifactId>bootstrap</artifactId>
        <version>3.3.7</version>
    </dependency>
</dependencies>

```

JSP's - src\main\webapp\WEB-INF\views

```

//index.jsp
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<form action="getUser" method="post">
    Username      <input type="text" name="username" > <br>
    Email         <input type="email" name="email" > <br>
    Password      <input type="password" name="password" > <br>
    <button type="submit">Sign in</button>
</form>

```

```

//user.jsp
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<h1>User Data is</h1>
<h4>${user.username}</h4>
<h4>${user.password}</h4>
<h4>${user.email}</h4>

```

Add entries in application.properties

```

# for SpringMVC - JSP related Config
spring.mvc.view.prefix:/WEB-INF/views/
spring.mvc.view.suffix:.jsp

#For detailed logging during development
logging.level.org.springframework=TRACE
logging.level.com=TRACE

```

HomeController.java

```

package springboot.controller;

@Controller
public class HomeController {

    @RequestMapping("/")
    public ModelAndView homePage() {
        ModelAndView view = new ModelAndView();
        view.setViewName("index");
        return view;
    }
}

```

```

@RequestMapping("/getUser")
public ModelAndView getUser(@ModelAttribute UserBo user) {
    ModelAndView view = new ModelAndView();
    view.addObject("user", user);
    view.setViewName("user");
    return view;
}
}

```

UserBo.java to store input data

```

public class UserBo {
    private int id;
    private String username;
    private String email;
    private String password;
    //Setters & Getters
}

```

SpringBootApplication.java

```

@SpringBootApplication
public class SpringBootApplication extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(SpringBootApplication.class);
    }

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

```



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.

@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;
@RestController
public class SpringBootRestController {
    @RequestMapping("/")
    public String welcome() {
        return "Spring Boot Home Page";
    }
    @RequestMapping("/hello")
    public String myData() {
        return "Smalcodes : Hello Spring Boot";
    }
}
```

app.SpringBootApplication.java

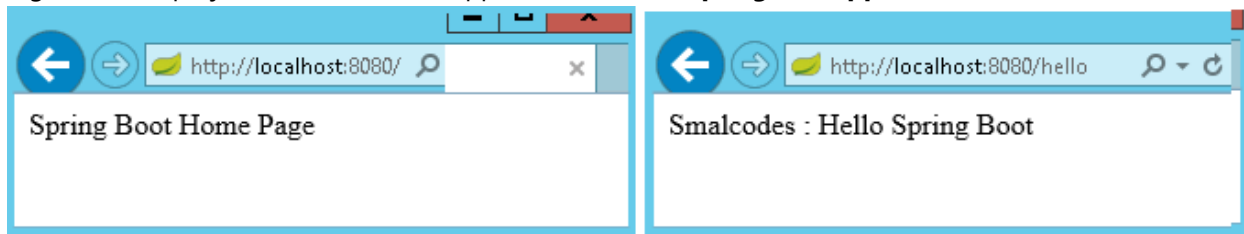
```
package app;

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

Create pom.xml same as first example.

Test the Application

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



- In above Spring Boot main application class in **app** package and controller class in **app.controller**.
- While starting our application, SpringBootApplication class will scan all the components under **app** package. we have created our controller class in **app.controller** which is inside **app** package.so our controller was registered by spring boot.
- If you create the controller class outside of the main package, lets say **com.smalcodes.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="smalcodes.controller")
public class SpringBootApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringBootApplication.class, args);
    }
}
```


Spring Boot - Rest API exception handling (@ControllerAdvice)

In below Code, you can observe multiple Try/Catch blocks to handle exceptions.

```
@RestController
public class TutorialController {
    @Autowired
    TutorialRepository tutorialRepository;
    @GetMapping("/tutorials")
    public ResponseEntity<List<Tutorial>> getAllTutorials(@RequestParam(required = false) String
title) {
    try {
        ...
        return new ResponseEntity<>(tutorials, HttpStatus.OK);
    } catch (Exception e) {
        return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

@GetMapping("/tutorials/{id}")
public ResponseEntity<Tutorial> getTutorialById(@PathVariable("id") long id) {
    Optional<Tutorial> tutorialData = tutorialRepository.findById(id);
    if (tutorialData.isPresent()) {
        return new ResponseEntity<>(tutorialData.get(), HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}

@PutMapping("/tutorials/{id}")
public ResponseEntity<Tutorial> updateTutorial(@PathVariable("id") long id, @RequestBody
Tutorial tutorial) {
    Optional<Tutorial> tutorialData = tutorialRepository.findById(id);
    if (tutorialData.isPresent()) {
        ...
        return new ResponseEntity<>(tutorialRepository.save(_tutorial), HttpStatus.OK);
    } else {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
}
...
@DeleteMapping("/tutorials/{id}")
public ResponseEntity<HttpStatus> deleteTutorial(@PathVariable("id") long id) {
    try {
        tutorialRepository.deleteById(id);
        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    } catch (Exception e) {
        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
@DeleteMapping("/tutorials")
public ResponseEntity<HttpStatus> deleteAllTutorials() {
    // try and catch
}
@GetMapping("/tutorials/published")
public ResponseEntity<List<Tutorial>> findByPublished() {
    // try and catch
}
}
```

You can see that we use try/catch many times for similar exception ([INTERNAL_SERVER_ERROR](#)), and there are also many cases that return [NOT_FOUND](#).

To avoid handling same exceptions multiple times in a Single controller, we can use the concept called Global Exception handling using `@ControllerAdvice` and `@ExceptionHandler`

```
@GetMapping("/testReport/{id}")
public ResponseEntity<TestReport> getReportByID(@PathVariable int id){
    TestReport data = repository.getById(id);
    try {
        return ResponseEntity.ok()
            .contentType(MediaType.APPLICATION_JSON)
            .body(data);
    } catch (Exception e) {
        return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
    }
}
```

If we access above api call with invalid id like 12, you will get following response.

```
"timestamp" "2020-11-28T13:24:02.239+00:00"
"status" 500
"error" "Internal Server Error"
"message" ""
"path" "/ testReport/12"
```

We can see that besides a well-formed error response, the payload is not giving us any useful information. Even the message field is empty, which we might want to contain something like **“Report with id 12 not found”**.

We can handle this type of Global Exception Handling with below Annotations

- `@ControllerAdvice` : is used with **class** level for global error/exception handling
- `@ExceptionHandler` : is used with **methods** (not with class).
- `@ResponseStatus_` is used to annotate defined exception classes.

We can configure multiple exceptions in this class so that in our application if that exception will occur, this class will get invoked and we will have a proper error message.

Steps to for Global Exception Handling

1.Create Custom Exception class for **INTERNAL_SERVER_ERROR & NOT_FOUND** exceptions & annotate our Exception class with `@ResponseStatus` and pass value to HTTP.Exception property.

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class RecordNotFoundException extends RuntimeException {
    String message;

    public RecordNotFoundException(String message) {
        super(message);
        this.message = message;
    }
}
```

```
@ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
public class InternalServerErrorException extends RuntimeException {
    String message;
    public InternalServerErrorException(String message) {
        super(message);
        this.message = message;
    }
}
```

2. Now Create class which do Global Exception handling by

- annotating class with `@ControllerAdvice`
- annotating methods with `@ExceptionHandler(RecordNotFoundException.class)`

```
@ControllerAdvice
public class GlobalExceptionHandlerForAllExceptions extends ResponseEntityExceptionHandler {

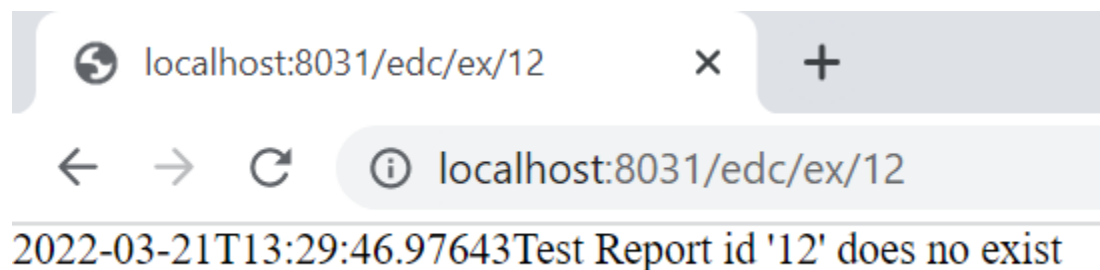
    @ExceptionHandler(RecordNotFoundException.class)
    public ResponseEntity<Object> handleRecordNotFoundException(RecordNotFoundException ex,
    WebRequest request) {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("message", ex.getLocalizedMessage());
        return new ResponseEntity<>(body, HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(InternalServerError.class)
    public ResponseEntity<Object> handleNodataFoundException(InternalServerError ex,
    WebRequest request) {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", LocalDateTime.now());
        body.put("message", "Server Has Issue while Processing this Request");
        return new ResponseEntity<>(body, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

3. Update Controller class method Response to throw UserDefined exception.

```
@GetMapping(path = "/ex/{id}", produces = MediaType.APPLICATION_JSON_VALUE)
public TestReport getReportByID(@PathVariable int id) {
    return repository.findById(id)
        .orElseThrow(
            () -> new RecordNotFoundException("Test Report id '" + id + "' does no exist"));
}
```

4. When ever Controller Class Throws exception, it will Automatically forward exception handling to ControllerAdvice class & Calls the method which is annotated with `@ExceptionHandler` with thrown exception class ex : `InternalServerError.class`)



<https://reflectoring.io/spring-boot-exception-handling/>

<https://zetcode.com/springboot/controlleradvice/>

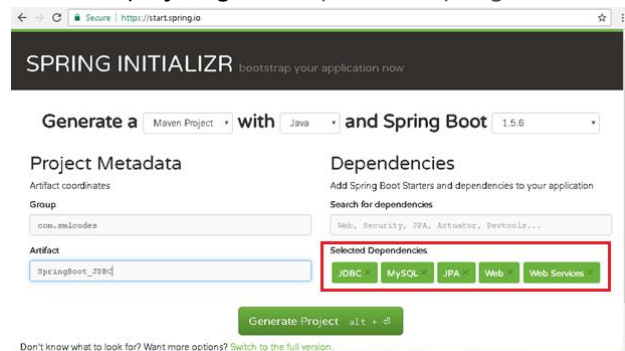
5. Spring Boot –Database

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;
    //Setters & Getters
}
```

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;

@Repository
public class StudentDAO {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    public List<Student> findAll() {

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

    public void addStudent(int sno, String name, String address) {
        jdbcTemplate.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;

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;

@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) {
    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;
}
}

```

SpringBootApplication.java

```

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

```

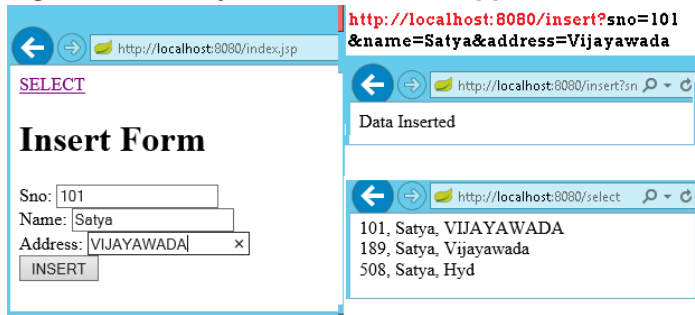
Static/index.jsp

```

<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>

```

Rightclick 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.

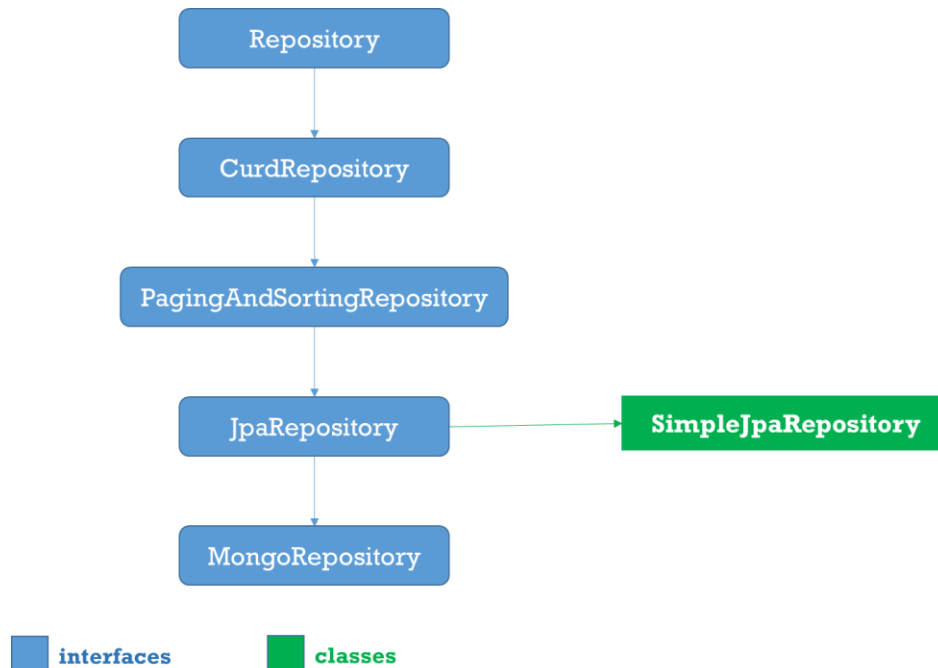
<code>@Entity</code>	Marks java class to a Table name
<code>@Table(name="tablename")</code>	Provides table name, when table name & class names are different .
<code>@Id</code>	Identifies the unique ID of the database entry
<code>@GeneratedValue</code>	Together with an ID this annotation defines that value is generated automatically.
<code>@Transient</code>	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. It provides classes & methods, which are common for all the SQL, NoSQL, BigData databases

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



1. Repository

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

2. CurdRepository

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

```
public interface CurdRepository<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 - pageNumber(), pageSize(), 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;  
@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.

```
package app.repository;
import org.springframework.data.repository.CrudRepository;
import app.entity.Student;
public interface StudentRepository extends CrudRepository<Student, String>{
}
```

StudentService.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);
    }

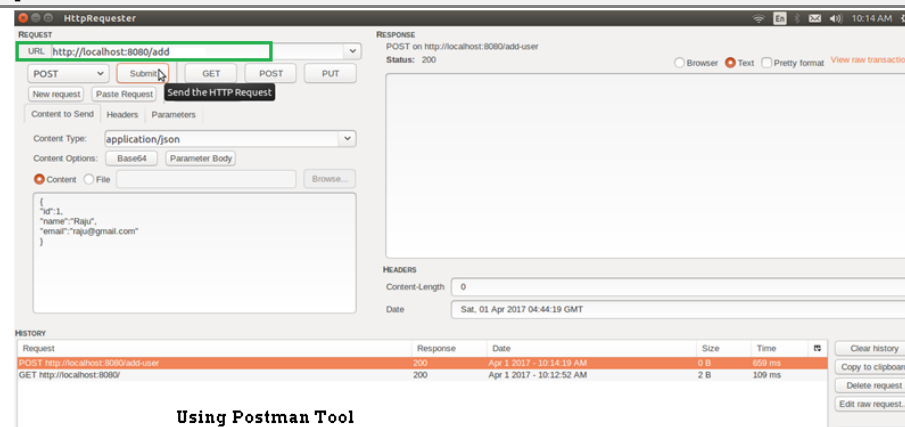
    @RequestMapping(value = "/get/{id}", method = RequestMethod.GET)
    public Student getStudent(@PathVariable String id) {
        return studentService.getStudent(id);
    }
}
```

SpringBootApplication.java

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

<http://localhost:8080/> -get All Students

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



Using Postman Tool

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();
}
```

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;

    //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

```
@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. Below is **Student** model class

```
@Document
public class Student{
    @Id
    String sno;
    String name;
    String address;
    //Setters & getters
}
```

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

StudentRepository.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);
}
```

StudentController.java

```
@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>



SpringBoot – with Multiple Databases / DataSources

Ref. <https://www.javadevjournal.com/spring-boot/multiple-data-sources-with-spring-boot/>

In your application you have two different databases MySQL, MongoDB. For using multiple databases, we need to create multiple datasources following steps.

application.properties

Create two different datasources in **application.properties** by changing **prefix**

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

#Database
spring.mongodb.datasource.url=jdbc:mongodb://localhost/mongodb
spring.mongodb.username=root
spring.mongodb.password=root
spring.mongodb.driver-class-name=com.mongodb.jdbc.Driver
```

we have created Entity & Repository classes for each database in separate packages.

- **com.mysql.entity.*** : **com.mysql.repository.MySQLRepository**
- **com.mongo.entity.*** : **com.mongo.repository.MongoRepository**

Spring Database configuration files

We need to create Two different Spring Configuration files & each of will contains following details.

1. **DataSource** : **It will Build DataSource**
2. **EntityManagerFactory** : **Scans the entity classes in given package**
3. **TransactionManager** : **Manages transactions using JPATransactions**

MySQLDataSourceConfig.java

```
@Configuration
@EnableTransactionManagement

@EnableJpaRepositories(
    entityManagerFactoryRef = "mysqlEntityManager",
    transactionManagerRef = "mysqlTransactionManager",
    basePackages = {
        "com.mysql.repository"
    }
)
public class MySQLDataSourceConfig {

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

    @Primary
    @Bean(name = "mysqlEntityManager")
    public LocalContainerEntityManagerFactoryBean
    mysqlEntityManager(EntityManagerFactoryBuilder builder, @Qualifier("mysqlDatasource")
    DataSource dataSource) {
        return builder.dataSource(dataSource)
            .packages("com.mysql.entity")
            .persistenceUnit("db1")
            .build();
    }

    @Primary
    @Bean(name = "mysqlTransactionManager")
    public PlatformTransactionManager
    customerTransactionManager(@Qualifier("mysqlEntityManager") EntityManagerFactory
    customerEntityManagerFactory) {
        return new JpaTransactionManager(customerEntityManagerFactory);
    }
}
```

MongoDBDataSourceConfig.java

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
    entityManagerFactoryRef = "mongoEntityManager",
    transactionManagerRef = "mongoTransactionManager",
    basePackages = {
        "com.mongo.repository"
    }
)
public class MongoDBDataSourceConfig {

    @Primary
    @Bean(name = "mongoDatasource")
    @ConfigurationProperties(prefix = "spring.mongo")
    public DataSource mongoDatasource() {
        return DataSourceBuilder.create().build();
    }

    @Primary
    @Bean(name = "mongoEntityManager")
```

```

    public LocalContainerEntityManagerFactoryBean
    mongoEntityManager(EntityManagerFactoryBuilder builder, @Qualifier("mongoDataSource")
DataSource dataSource) {
    return builder.dataSource(dataSource)
        .packages("com.mongo.entity")
        .persistenceUnit("db1")
        .build();
    }

    @Primary
    @Bean(name = "mongoTransactionManager")
    public PlatformTransactionManager
    customerTransactionManager(@Qualifier("mongoEntityManager") EntityManagerFactory
customerEntityManagerFactory) {
    return new JpaTransactionManager(customerEntityManagerFactory);
    }
}

```

Now if we use Repository classes, it will automatically be mapped to configured databases only.

```

public class TestReportController {
    @Autowired
    MySQLRepository mysqlRepository;

    @Autowired
    MongoRepository mongoRepository;
}

```

6. SpringBoot - Security

Spring Security is a framework that focuses on providing both **authentication** and **authorization** (or "access-control") to Java web application and SOAP/RESTful web services.

Spring Security currently supports integration with all of the following technologies:

- **HTTP** basic access authentication
- **LDAP** system
- **OpenID** identity providers – Google, Facebook etc
- **JAAS** API: JAAS stands for Java Authentication and Authorization Service. This is a pluggable module which is implemented in Java and the Spring Security Framework uses it for its authentication purposes.
- **Single Sign-On**: This allows the user to get access to multiple applications with only one account (username and password).
- CAS Server
- ESB Platform
- Your own authentication systems

3. Authentication Manager

Authentication Manager is the core for the Spring security authentication process.

AuthenticationManager is the API that defines how Spring Security's Filters perform authentication.

```
public interface AuthenticationManager {  
    Authentication authenticate(Authentication authentication) throws AuthenticationException;  
}
```

Before we move ahead, let's cover important points for the AuthenticationManager.

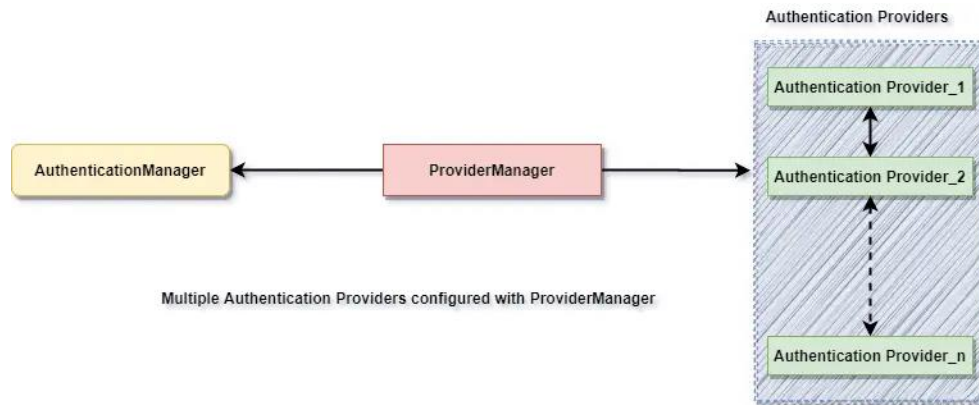
1. Internally the AuthenticationProvider responsible to perform the authentication process.
2. ProviderManager manager is the most used implementation of AuthenticationProvider.
3. ProviderManager delegates the request to the list of AuthenticationProvider.

4. Authentication Providers

The AuthenticationProvider are responsible to process the request and perform a specific authentication. It provides a mechanism for getting the user details with which we can perform authentication. This is how the AuthenticationProvider interface looks like:

```
public interface AuthenticationProvider {  
    Authentication authenticate(Authentication authentication) throws AuthenticationException;  
    boolean supports(Class<?> authentication);  
}
```

Our AuthenticationProvider can't execute directly but we can configure multiple providers using the AuthenticationProvider.



If you are curious, here is the list of some OOTB authentication providers.

- **DaoAuthenticationProvider.**
- JAAS Authentication
- OpenID Authentication
- X509 Authentication
- SAML 2.0
- OAuth 2.0
- RememberMeAuthenticationProvider
- LdapAuthenticationProvider

5. Custom Authentication Provider

For enterprise applications, we may need a custom authentication provider. Implement the `AuthenticationProvider` interface to create a custom authentication provider for your application.

```
@Component
public class CustomAuthenticationProvider implements AuthenticationProvider {

    @Override
    public Authentication authenticate(Authentication auth) throws AuthenticationException {
        String username = authentication.getName();
        String pwd = authentication.getCredentials().toString();
        if ("javadevjournal".equals(username) && "pass".equals(pwd)) {
            return new UsernamePasswordAuthenticationToken(username, password,
Collections.emptyList());
        } else {
            throw new BadCredentialsException("User authentication failed!!!!");
        }
    }

    @Override
    public boolean supports(Class<?>auth) {
        return auth.equals(UsernamePasswordAuthenticationToken.class);
    }
}
```

The last step is to configure our *custom authentication provider with Spring security*. We do that by creating a custom configuration class and extending the `WebSecurityConfigurerAdapter`.

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    CustomAuthenticationProvider customAuthenticationProvider;

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        //configuring our custom auth provider
        auth.authenticationProvider(customAuthProvider);
    }
}
```

6. Spring Security UserDetailsService

Not all, but few authentication providers may need `UserDetailsService` to get the user details stored in the database by username (e.g. `DaoAuthenticationProvider`). Most of the standard web application may use the `UserDetailsService` to get user information during login process. This is how the `UserDetailsService` interface look like:

```
public interface UserDetailsService {
    UserDetails loadUserByUsername(String var1) throws UsernameNotFoundException;
}
```

It's a common use case to define a custom `UserDetailsService` for our application.

7. Authentication and Authentication Exception

During the authentication process, if the user authentication is successful, we will send a fully initialized `Authentication` object back. For failed authentication, `AuthenticationException` will be thrown. A fully populated authentication object carries the following details:

- User credentials.
- List of granted authorities (for authorization).

- Authentication flag.

```
public interface Authentication extends Principal, Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();

    Object getCredentials();
    Object getDetails();
    Object getPrincipal();
    boolean isAuthenticated();
    void setAuthenticated(boolean var1) throws IllegalArgumentException;
}
```

8. Setting Authentication SecurityContext

The last step on the successful authentication is setting up the authentication object in the SecurityContext. It wraps the SecurityContext around the SecurityContextHolder. Keep in mind following points:

- SecurityContextHolder is where Spring Security stores the details about authenticated users.
- Spring security will not validate how the SecurityContextHolder is populated.
- If it finds values in the SecurityContextHolder, it assumes that current user is an authenticated user. This is how the SecurityContextHolder populates

```
SecurityContext context = //get the context from security holder

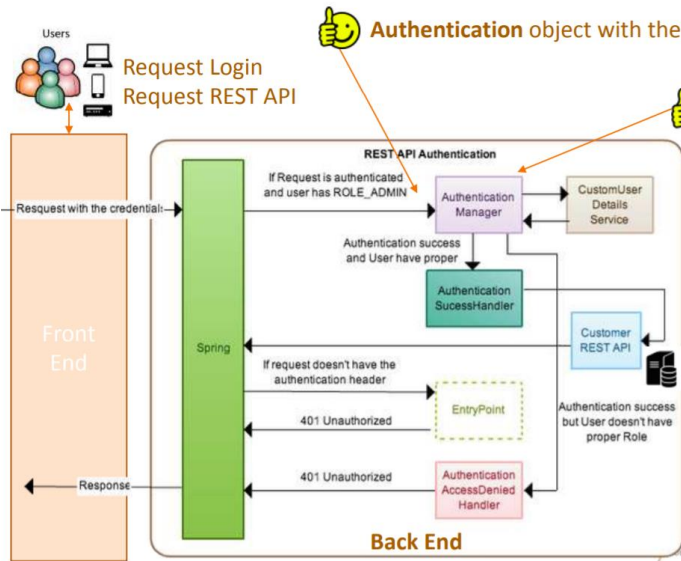
UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(username, password);

context.setAuthentication(authentication);

SecurityContextHolder.setContext(context);
```

REf <https://www.javadevjournal.com/spring-security/spring-security-authentication/>

- | | |
|---|--|
| <ul style="list-style-type: none"> › Principal <ul style="list-style-type: none"> › User that performs the action › Authentication <ul style="list-style-type: none"> › Confirming truth of credentials › Authorization <ul style="list-style-type: none"> › Define access policy for principal › GrantedAuthority <ul style="list-style-type: none"> › Application permission granted to a principal › SecurityContext <ul style="list-style-type: none"> › Hold the authentication and other security information › SecurityContextHolder <ul style="list-style-type: none"> › Provides access to SecurityContext | <ul style="list-style-type: none"> › AuthenticationManager <ul style="list-style-type: none"> › Controller in the authentication process › AuthenticationProvider <ul style="list-style-type: none"> › Interface that maps to a data store which stores your user data. › Authentication Object <ul style="list-style-type: none"> › Object is created upon authentication, which holds the login credentials. › UserDetails <ul style="list-style-type: none"> › Data object which contains the user credentials, but also the Roles of the user. › UserDetailsService <ul style="list-style-type: none"> › Collects the user credentials, authorities(roles) and build an UserDetails object. |
|---|--|

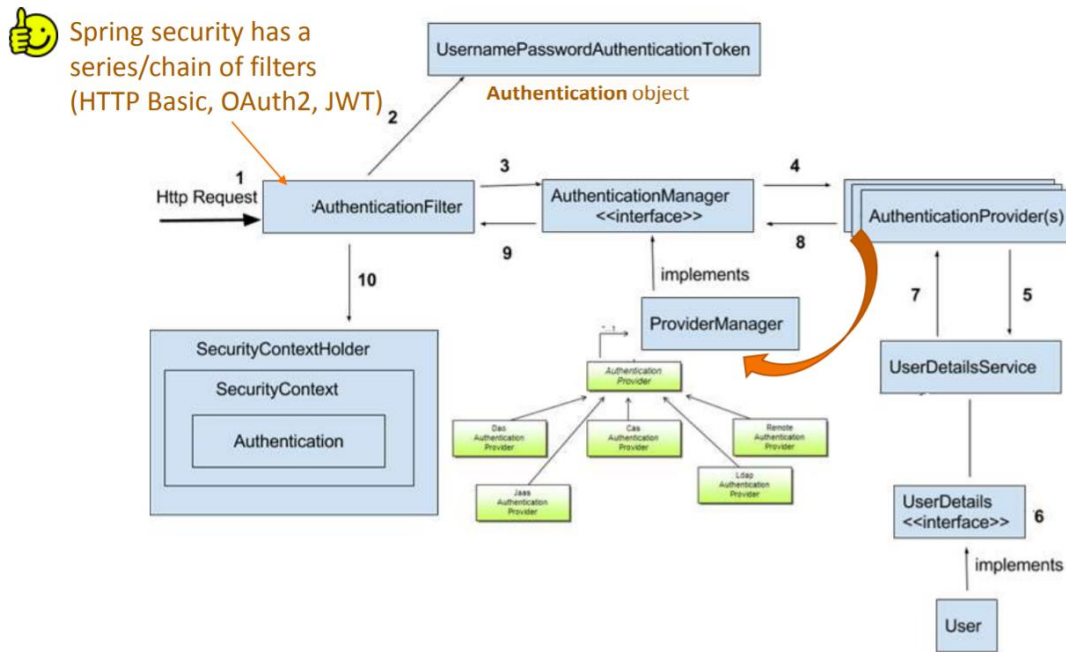


👍 Authentication object with the login credentials is created

👍 Controller in the process that it just delegates to a collection of AuthenticationProvider instances.

Call an object that implements the **UserDetailsService** interface, which it looks up the user data and returns a **UserDetails** object.

Will check that the password matches the password the user entered.



Spring Security Authentication Types

- Web Security Authentication (@EnableWebSecurity)
 1. Basic Authentication
 2. In-Memory Authentication
 3. JDBC Authentication
 4. LDAP Authentication
- JWT(JSON Web Token) Authentication
- OAuth 2.0 SSO Authentication
- REST API

With the latest Spring Security and/or Spring Boot versions, the way to configure Spring Security is by having a class that:

1. Is annotated with `@EnableWebSecurity`.
2. Extends `WebSecurityConfigurerAdapter`, which basically offers you a configuration DSL/method. With those methods, you can specify what URIs in your application to protect or what exploit protections to enable/disable.

Here's what a typical `WebSecurityConfigurerAdapter` looks like

```
@Configuration
@EnableWebSecurity // (1)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter { // (1)

    @Override
    protected void configure(HttpSecurity http) throws Exception { // (2)
        http.authorizeRequests()
            .antMatchers("/", "/home").permitAll() // (3)
            .anyRequest().authenticated() // (4)
            .and()
            .formLogin().loginPage("/login").permitAll()
            .and().logout() // (6)
            .permitAll()
            .and()
            .httpBasic(); // (7)
    }
}
```

1. A normal Spring `@Configuration` with the `@EnableWebSecurity` annotation, extending from `WebSecurityConfigurerAdapter`.
2. By overriding the adapter's `configure(HttpSecurity)` method, you get a nice little DSL with which you can configure your FilterChain.
3. All requests going to `/` and `/home` are allowed (permitted) - the user does *not* have to authenticate. You are using an `antMatcher`, which means you could have also used wildcards (`*`, `**`, `?`) in the string.
4. Any other request needs the user to be authenticated first, i.e. the user needs to login.
5. You are allowing form login (username/password in a form), with a custom loginPage (`/login`, i.e. not Spring Security's auto-generated one). Anyone should be able to access the login page, without having to log in first (permitAll; otherwise, we would have a Catch-22!).
6. The same goes for the logout page
7. On top of that, you are also allowing Basic Auth, i.e. sending in an HTTP Basic Auth Header to authenticate.

1.Spring Security – Quick Start Example

We have SpringBoot Application, which doesn't have login page. After running application, it will show all Employees Page.

Home | Add Employee | **All Employees**

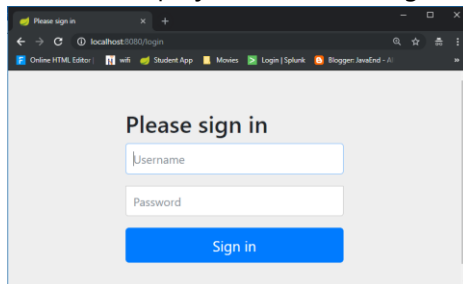
# Emp. ID	Name	Address	Salary
1	Satya	Killaru	20000.0
2	satya sassa	Nellore	33333.0

We are going to add Spring Security jar to this project. **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 – It will Directly open Login Page



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

In our application, we did not create this login page but configured the Spring Security. This is a built-in login page provided by the framework itself to authenticate the user. **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.username 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
```

Thus, by just adding the spring boot security starter dependency the basic security has already been configured by default. Let's customize the security configuration by writing our own authorization and authentication. For this create a new class *SecurityConfig* that extends the *WebSecurityConfigurerAdapter* and overrides its methods.

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
```

2.Spring Security – Basic & In-Memory Authentication Example

If we are disabling security auto-configuration, we need to provide our own configuration by creating new class annotated with **@EnableWebSecurity** which extends **WebSecurityConfigurerAdapter**.

Spring Security provides HTTP basic authentication to authenticate the user at the client-side and send the user credentials with the request header to the server. The server receives these credentials, extract them from the header, and map them with the existing record to validate the user.

The **BasicAuthenticationFilter** handles the request and check whether the request contains an authentication header or not. The **httpBasic()** method enables the Basic HTTP security in our application.

```
@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(); // This means it is Basic Authentication
    }

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

        return new InMemoryUserDetailsManager(user);
        // This means We provide User data using InMemory
    }
}
```

Here we provided Static user content. This type of Authorization Known as **InMemoryAuthorization**.

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/maven.
- 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 `userService` implementation.

3.Spring Security – Custom Login Form

By default, the Spring Security framework provides its own login page. But we want to render a login page that matches the company website theme, we need to define our own login page. We will do it by creating a `login.jsp`. It will submit to the `"/authenticateTheUser"` action and then Spring Security will perform authentication based on the provided input.

`EmployeeController.java` – add `/login` to display login page

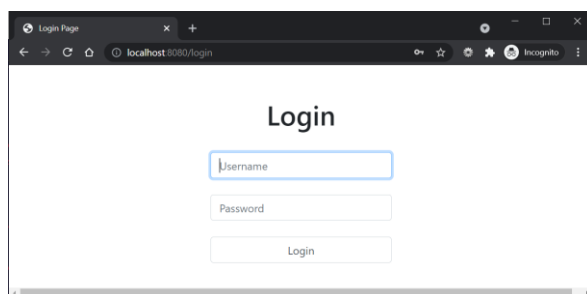
```
@GetMapping("/login")
public String login() {
    return "login";
}
```

`SecurityConfig.java`

We used `HttpSecurity` class to configure the login page. The `loginPage()` is used to specify our `login.jsp` page. We can also use any other name for login form such as `login-form.jsp` or `user-login.jsp` and then specify the mapping to this method. The `"/login"` value passed here will map to the controller's action and then render the JSP page.

```
@Override
public void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().anyRequest().
        authenticated().and()
        .formLogin()
        .loginPage("/login")
        .loginProcessingUrl("/authenticateTheUser").permitAll();
}
```

Now It will display our own custom form.



Custom Error message

If the user enters the wrong credentials, then Spring Security responds by attaching an `error` parameter to the URL. Update `login.jsp` page

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<c:if test="${param.error!=null}">
    <p style="color: red">You entered wrong credentials!</p>
</c:if>
```

Logout User

To implement this feature, follow these steps

1. Add a Logout Button
2. Configure Security
3. Show Message

1. Add a Logout Button(index.jsp)

add a button to the JSP page that enables logout to the user. This button is a submit button that submits a form request to the logout URL that will be handle by Spring Security. Put it on the JSP home page.

```
<form:form action="${pageContext.request.contextPath}/logout"
    method="post">
    <input type="submit" value="Logout" class="list-group-item list-group-item-action">
</form:form>
```

2. Configure Security (SecurityConfig.java)

Now, update the `SecurityConfig.java` class by adding `logout()` and `permitAll()` method.

- `logout()` method will logout the user
- `permitAll()` method give access to all the users to use the logout feature.

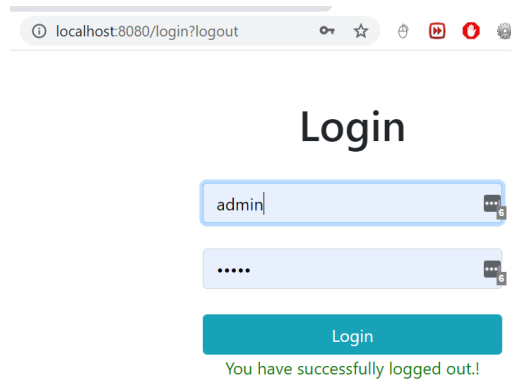
```
.logout().permitAll();
```

```
@Override
public void configure(HttpSecurity http) throws Exception {
    // 2.Custom Login Form
    http.authorizeRequests().anyRequest()
        .authenticated()
        .and()
        .formLogin()
        .loginPage("/login")
        .loginProcessingUrl("/authenticateTheUser")
        .permitAll()
        .and()
        .logout()
        .permitAll();
}
```

3.Show Message(login.jsp)

Now, set a message to show the user when the user logged out successfully. Add it to the `login.jsp` to display a logout message to the user after successful logout.

```
<c:if test="${param.logout!=null}">
    <p style="color: green">You have successfully logged out.!</p>
</c:if>
```

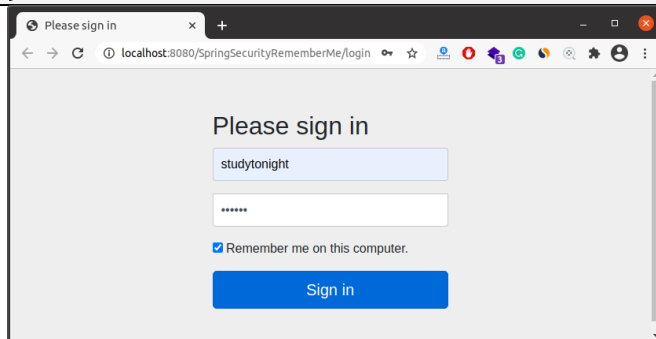


Remember Me Feature

remember-me feature of Spring Security allows a user to remember even after the session is closed. It performs automatic login by using the stored **cookies**.

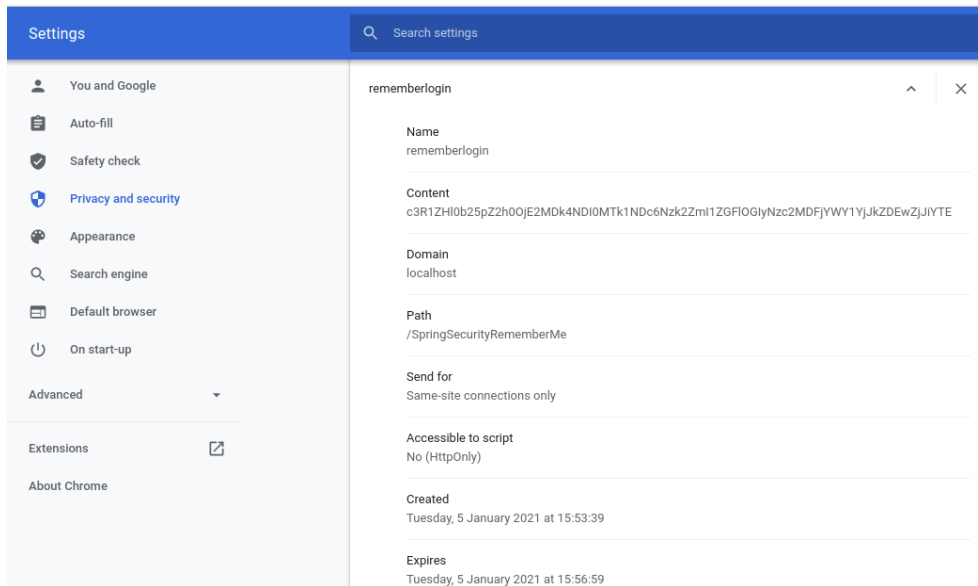
An application that has a remember-me feature, stores a cookie in the browser which is used to identify the user and avoid entering the user credentials each time to log in. So what we need to do is just simply use the `rememberMe()` method in the `configure()` method of `SecurityConfig` class.

```
public void configure(HttpSecurity http) throws Exception {  
    // 2.Custom Login Form  
    http.authorizeRequests().anyRequest()  
        .authenticated()  
        .and()  
        .formLogin().loginPage("/login").loginProcessingUrl("/authenticateTheUser")  
        .permitAll()  
        .and()  
        .logout().permitAll()  
        .and()  
        .rememberMe().key("rem-me-key")  
                    .rememberMeParameter("remember")  
                    .rememberMeCookieName("rememberlogin")  
                    .tokenValiditySeconds(200);  
}
```



Verify the Cookies

See it stores the cookie that we set in the **SecurityConfig** file. It has the same name that we set and a token in encrypted form and expiring life of the cookie.



4.Spring Security – JDBC Authentication

By default, spring security expects tables named **users** table for storing username, passwords and **authorities** table for storing the associated roles.

```
CREATE TABLE `users` (  
  `username` varchar(50) NOT NULL,  
  `password` varchar(50) NOT NULL,  
  `enabled` tinyint(1) NOT NULL,  
  PRIMARY KEY (`username`)  
) ENGINE=InnoDB DEFAULT CHARSET=latin1;  
  
INSERT INTO `users`  
VALUES  
( 'satya', '{noop}satya',1),  
( 'a', '{noop}a',1);  
  
CREATE TABLE `authorities` (  
  `username` varchar(50) NOT NULL,  
  `authority` varchar(50) NOT NULL,  
  UNIQUE KEY `authorities_idx_1` (`username`,`authority`),  
  CONSTRAINT `authorities_ibfk_1` FOREIGN KEY (`username`) REFERENCES `users` (`username`)  
) ENGINE=InnoDB DEFAULT CHARSET=latin1;  
  
INSERT INTO `authorities`  
VALUES  
( 'satya', 'ROLE_ADMIN'),  
( 'a', 'ROLE_GUEST');
```

Override **configure(AuthenticationManagerBuilder)** — this method will help us in fetching user data from DB and validate if the credentials provided by the user is valid

SecurityConfig.java

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
  
  @Autowired  
  DataSource dataSource;  

```

```

//Enable jdbc authentication
@Autowired
public void configAuthentication(AuthenticationManagerBuilder auth) {
    auth.jdbcAuthentication().dataSource(dataSource);
}

@Override
public void configure(HttpSecurity http) throws Exception {
//3. JDBC Authentication role should not start with 'ROLE_' since it is automatically inserted
    // ADMIN - Allowed All Operations - ADD, EDIT , DELETE
    // GUEST- Allowed VIEW Operations - VIEW, SHOW
    http.authorizeRequests()
        .antMatchers("/").hasAnyRole("GUEST", "ADMIN")
        .antMatchers("/all").hasAnyRole("GUEST", "ADMIN")
        .antMatchers("/addEmployee").hasRole("ADMIN")
        .antMatchers("/saveEmployee").hasRole("ADMIN")
        .antMatchers("/deleteEmployee").hasRole("ADMIN")
        .and().formLogin().loginPage("/login")
        .loginProcessingUrl("/authenticateTheUser")
        .permitAll().and().logout()
        .permitAll().and().exceptionHandling().accessDeniedPage("/unauthorized");
}
}

```

5. Spring Security – LDAP Authentication

What is LDAP?

LDAP is Lightweight Directory Access Protocol that is used to interact with directory server. LDAP is a centralized directory acts as a data source, which contains all important information related to the organization such as user details, system information, etc. LDAP is used for authentication and storing information about users, groups and applications.

Spring Security provides `LdapAuthenticationProvider` class to authenticate a user against a LDAP server. The equivalent XML element is `<ldap-authentication-provider>`.

LDAP node is created with following keywords.

- **uid** : User Id
- **cn** : Common Name
- **sn** : Surname
- **o** : Organization
- **ou** : Organizational unit
- **dn** : Distinguished name
- **dc** : Domain Component

When the user submits login form, then to find the user a LDAP DN is created. Suppose the username is 'krishna' then the actual name used to authenticate to LDAP will be the full DN as following.

```
uid=krishna, ou=people, dc=concretepage, dc=com
```

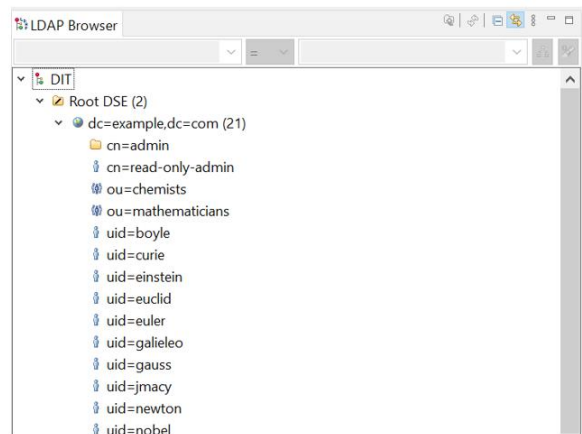
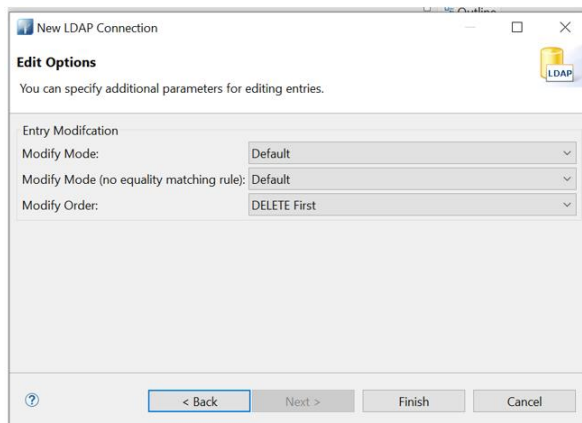
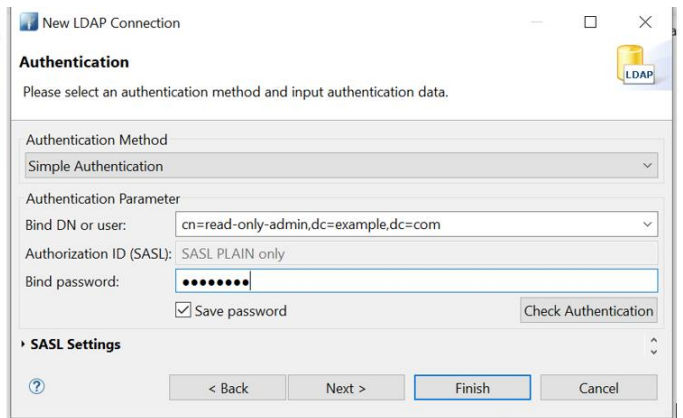
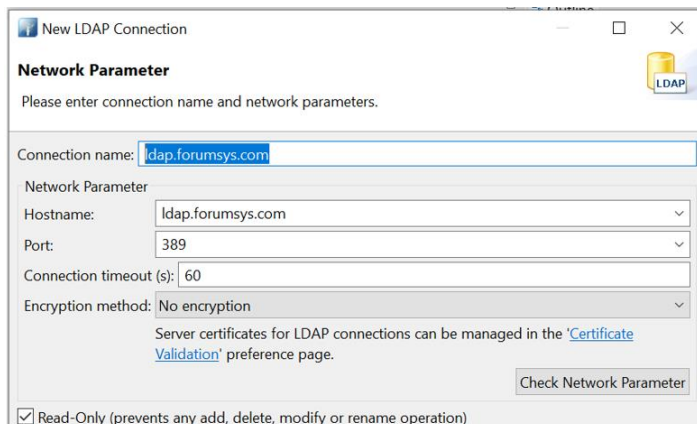
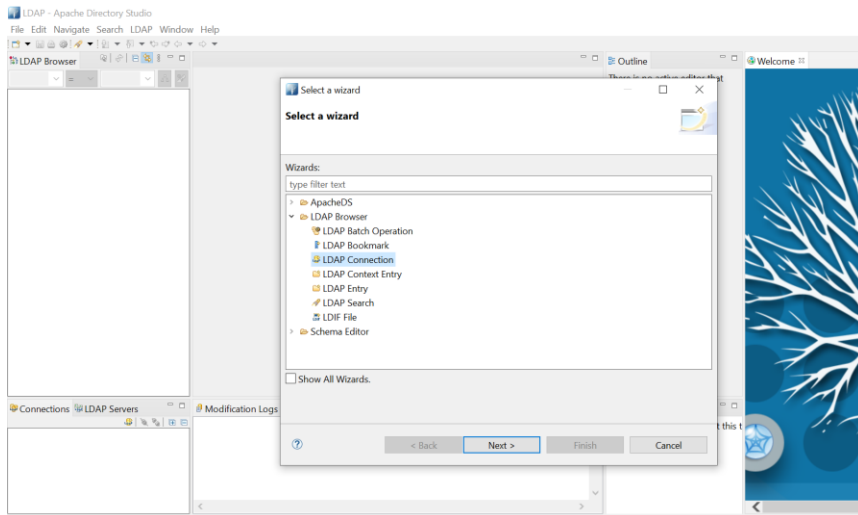

LDAP Server Configuration (Free Online)

LDAP Server Information (read-only access):

Server: ldap.forumsys.com
Port: 389
Bind DN: cn=read-only-admin,dc=example,dc=com
Bind Password: password

All user passwords are *password*.

Ref. <https://www.forumsys.com/tutorials/integration-how-to/ldap/online-ldap-test-server/>



Example Code

Pom.xml – Update Maven Dependency

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

<dependency>
  <groupId>org.apache.directory.server</groupId>
  <artifactId>apacheds-server-jndi</artifactId>
  <version>1.5.5</version>
</dependency>
```

application.properties – update LDAP Server Properties

```
ldap.urls=ldap://ldap.forumsys.com:389/
ldap.base.dn=dc=example,dc=com
ldap.username=cn=read-only-admin,dc=example,dc=com
ldap.password=password
ldap.user.dn.pattern = uid={0}
```

SpringSecurityConfig

At this point, we need to tell spring how to we are doing authentication, as in which URLs should be authenticated. We need to update Authentication provider as LDAP in our code.

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

    @Value("${ldap.urls}")
    private String ldapUrls;

    @Value("${ldap.base.dn}")
    private String ldapBaseDn;

    @Value("${ldap.username}")
    private String ldapSecurityPrincipal;

    @Value("${ldap.password}")
    private String ldapPrincipalPassword;

    @Value("${ldap.user.dn.pattern}")
    private String ldapUserDnPattern;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.ldapAuthentication()
            .contextSource()
            .url(ldapUrls + ldapBaseDn)
            .managerDn(ldapSecurityPrincipal)
            .managerPassword(ldapPrincipalPassword)
            .and()
            .userDnPatterns(ldapUserDnPattern);
    }
}

//4. LDAP Authentication role should not start with 'ROLE_' since it is automatically inserted
// authenticated - Allowed All Operations - ADD, EDIT, DELETE
// permitAll- Allowed VIEW Operations - VIEW, SHOW
```

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/login**").anonymous()
        .antMatchers("/all").permitAll()
        .antMatchers("/view**").permitAll()
        .antMatchers("/addEmployee").authenticated()
        .antMatchers("/addEmployee").authenticated()
        .antMatchers("/deleteEmployee").authenticated()
        .and().formLogin().loginPage("/login")
        .loginProcessingUrl("/authenticateTheUser")
        .permitAll().and()
        .logout().permitAll().and()
        .csrf()
        .disable();
}
}

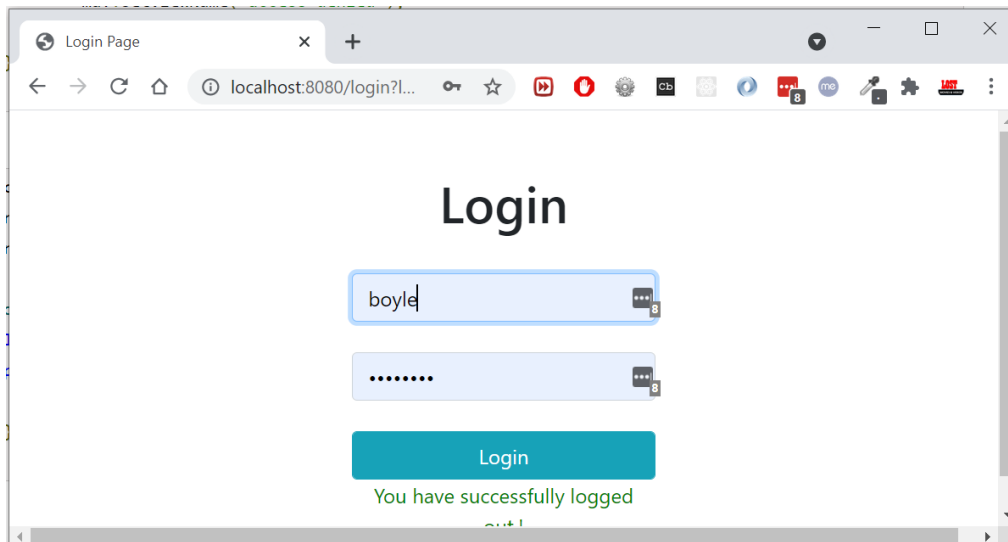
```

For Testing

```

//All UID's password is: password
uid=boyle
uid=curie
uid=einstein
uid=euclid
uid=euler

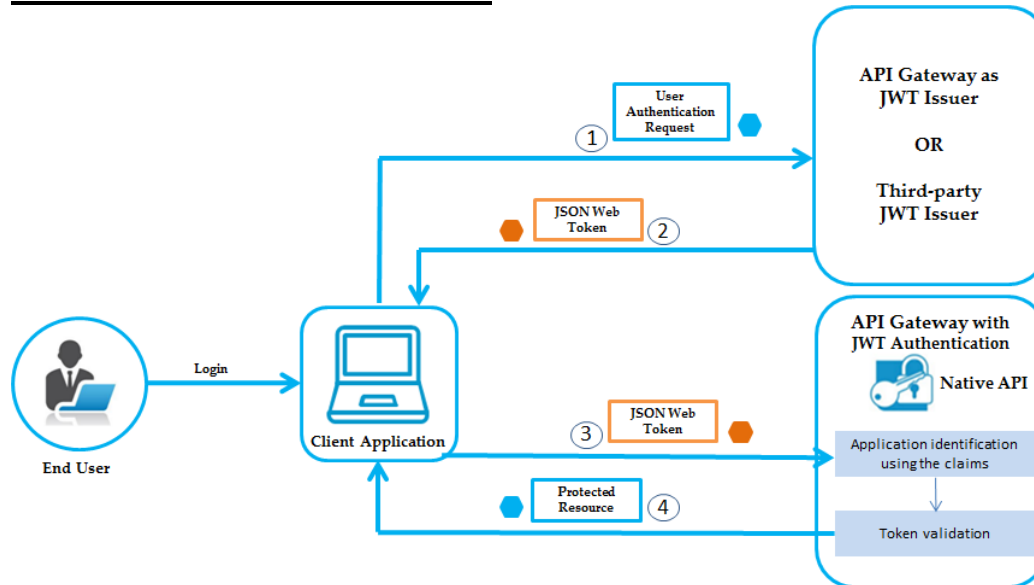
```



We can access /all , /view** - with out login. But for /addEmployee, /saveEmployee user must login with LDAP credentials.

6. Spring Security – JWT Authentication

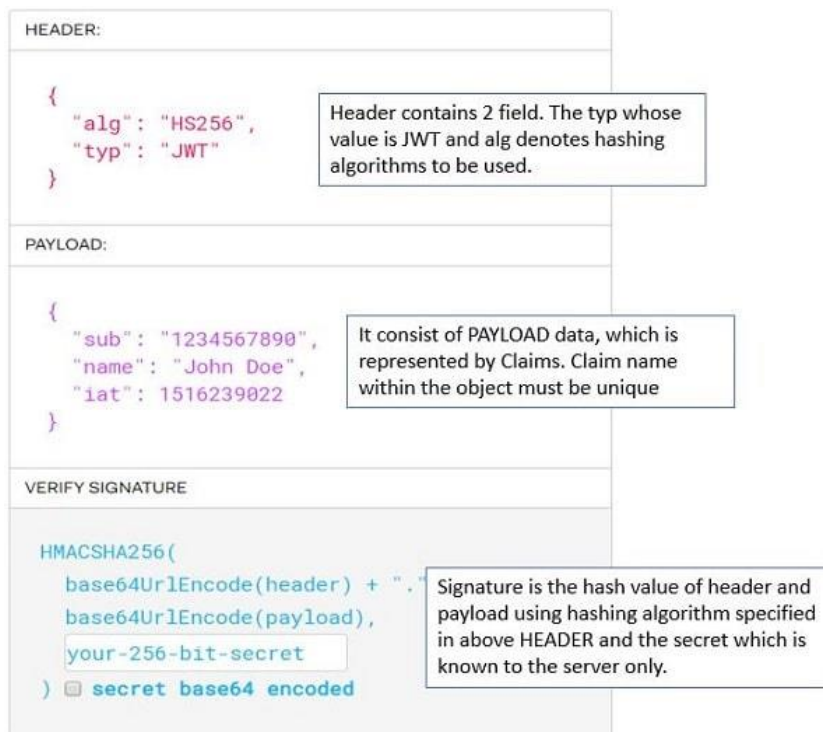
What is JWT Authentication



By using JWT authentication you no longer have to store user information in a session to authenticate each request. Instead, we send this information with each request along with any data we want.

When the client logs in, the server gives them a JSON Web Token (JWT). Afterward, the client has to put that token in the Authorization header of each request after the word *Bearer* and space.

The token has three different parts, **header**, **payload**, **signature**.



Create JWT Token Online

Will generate JWT Token by using [JWT Online Token Generator](#). Provide the payload as given below:

Standard JWT Claims

Issuer	<input type="text" value="Online JWT Builder"/>	Identifier (or, name) of the server or system issuing the token. Typically a DNS name, but doesn't have to be.
Issued At	<input type="text" value="2019-11-11T16:50:28.422Z"/>	Date/time when the token was issued. (defaults to now) now
Expiration	<input type="text" value="2020-11-10T16:50:28.422Z"/>	Date/time at which point the token is no longer valid. (defaults to one year from now) now in 20 minutes in 1 year
Audience	<input type="text"/>	Intended recipient of this token; can be any string, as long as the other end uses the same string when validating the token. Typically a DNS name.
Subject	<input type="text"/>	Identifier (or, name) of the user this token represents.

Provide Claim data.

Additional Claims

Claim Type	Value	
<input type="text" value="GivenName"/>	<input type="text" value="John"/>	✕
<input type="text" value="Surname"/>	<input type="text" value="Doe"/>	✕

Use this section to define 0 or more custom claims for your token. The claim type can be anything, and so can the value.

If recipient of the token is a .NET Framework application, you might want to follow the Microsoft [ClaimType](#) names. You can also use the .NET-oriented claim buttons below.

clear all add one add email claim
add name claim (.NET) add role claim (.NET) add email claim (.NET)

We'll have the following claims in the payload.

Generated Claim Set (plain text)

```
{
  "iss": "Online JWT Builder",
  "iat": 1573491028,
  "exp": 1605027028,
  "aud": "",
  "sub": "",
  "GivenName": "John",
  "Surname": "Doe"
}
```

This section displays the claims that will be signed and base64-encoded into a complete JSON Web Token.

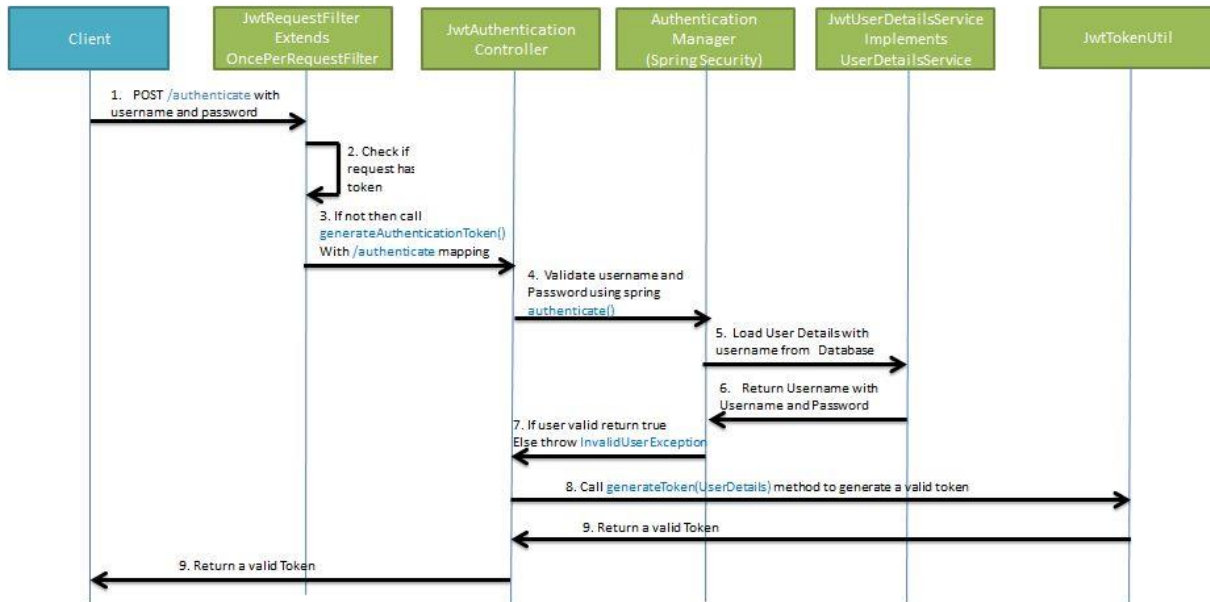
Sign the payload using the hashing algorithm.

Signed JSON Web Token

Key 8 HS512 Create Signed JWT

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJpbmtpbmUgSlldUEJ1aWxkZXIiLCJpYXQiOiJlbnZM0TEwMjgsImVz
```

Copy JWT to Clipboard



Example

[Pom.xml](#)

```

<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.9.1</version>
</dependency>
  
```

We have API for Employee Services

/	GET	Homepage
/login	GET	Loginform
/all	GET	All Employees
/saveEmployee	POST	saveEmployee

Not much Important for now, Skipping.

Ref. https://www.tutorialspoint.com/spring_security/spring_security_with_jwt.htm

7. Spring Security – OAuth 2.0 SSO Authentication

Spring Boot 2.x provides full auto-configuration for OAuth2 login. We just need to configure client id and client secret for OAuth2 provider such as GitHub, Facebook and Google in `application.property` file and we are done.

We can customize the configuration by overriding methods

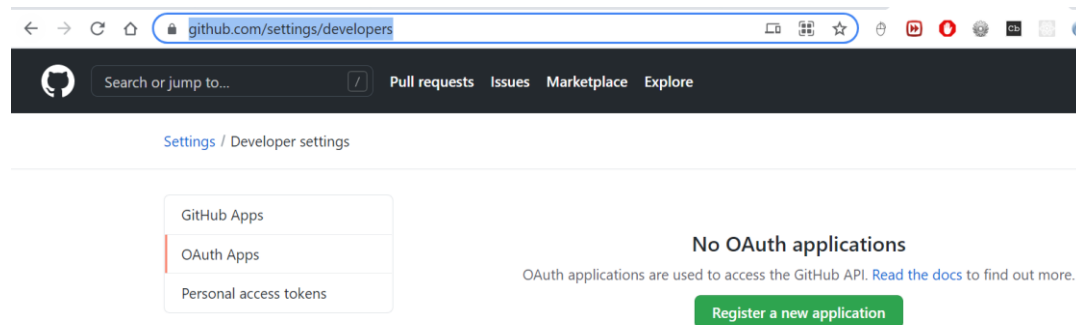
of `WebSecurityConfigurerAdapter` and using `HttpSecurity.oauth2Login()` method introduced in Spring 5.0.

We need to create a bean for `ClientRegistrationRepository` to override OAuth2 property value. Here on this page we will create a Spring Boot Security application for OAuth2 login using GitHub, Facebook and Google authentication provider.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

Example

1. Generate github API Keys by going <https://github.com/settings/developers>



redirect URI as

```
http://localhost:8080/login/oauth2/code/github
http://localhost:8081/login/oauth2/code/google
http://localhost:8081/login/oauth2/code/facebook
```

Watch this : <https://www.youtube.com/watch?v=R9lxXQcy-nM>

Register a new OAuth application

Application name *

Something users will recognize and trust.

Homepage URL *

The full URL to your application homepage.

Application description

This is displayed to all users of your application.

Authorization callback URL *

Your application's callback URL. Read our [OAuth documentation](#) for more information.

2. Update [application.properties](#)

```
spring.security.oauth2.client.registration.google.client-id=<your client id>
spring.security.oauth2.client.registration.google.client-secret=<your client secret>

spring.security.oauth2.client.registration.facebook.client-id=<your client id>
```

```
spring.security.oauth2.client.registration.facebook.client-secret=<your client secret>
spring.security.oauth2.client.registration.github.client-id=4d381Dummy
spring.security.oauth2.client.registration.github.client-secret=2876951aSerccrit
```

3. Update Login Page

```
<a href="/oauth2/authorization/github">Login with GitHub</a>
```

4. CustomOAuth2User.java

this class extends OAuthUser interface as defined by Spring OAuth2 API. this class wraps an instance of `OAuth2User`, which will be passed by Spring OAuth2 upon successful OAuth authentication. And we override the `getName()` method to return `username` associated with GitHub account.

```
public class CustomOAuth2User implements OAuth2User {
    private OAuth2User oauth2User;

    public CustomOAuth2User(OAuth2User oauth2User) {
        this.oauth2User = oauth2User;
    }
    @Override
    public Map<String, Object> getAttributes() {
        return oauth2User.getAttributes();
    }
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return oauth2User.getAuthorities();
    }
    @Override
    public String getName() {
        return oauth2User.getAttribute("name");
    }
}
```

`CustomOAuth2UserService` subclass of `DefaultOAuth2UserService`. we override the `loadUser()` method which will be called by Spring OAuth2 upon successful authentication, and it returns a new `CustomOAuth2User` object

```
@Service
public class CustomOAuth2UserService extends DefaultOAuth2UserService {

    @Override
    public OAuth2User loadUser(OAuth2UserRequest userRequest) throws OAuth2AuthenticationException
    {
        OAuth2User user = super.loadUser(userRequest);
        return new CustomOAuth2User(user);
    }
}
```

`OAuth2AuthenticationSecurityConfig` To integrate single sign on with GitHub with traditional username and password login, update configuration for Spring security as follows

```
@Configuration
@EnableWebSecurity
public class OAuth2AuthenticationSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private CustomOAuth2UserService userService;
```

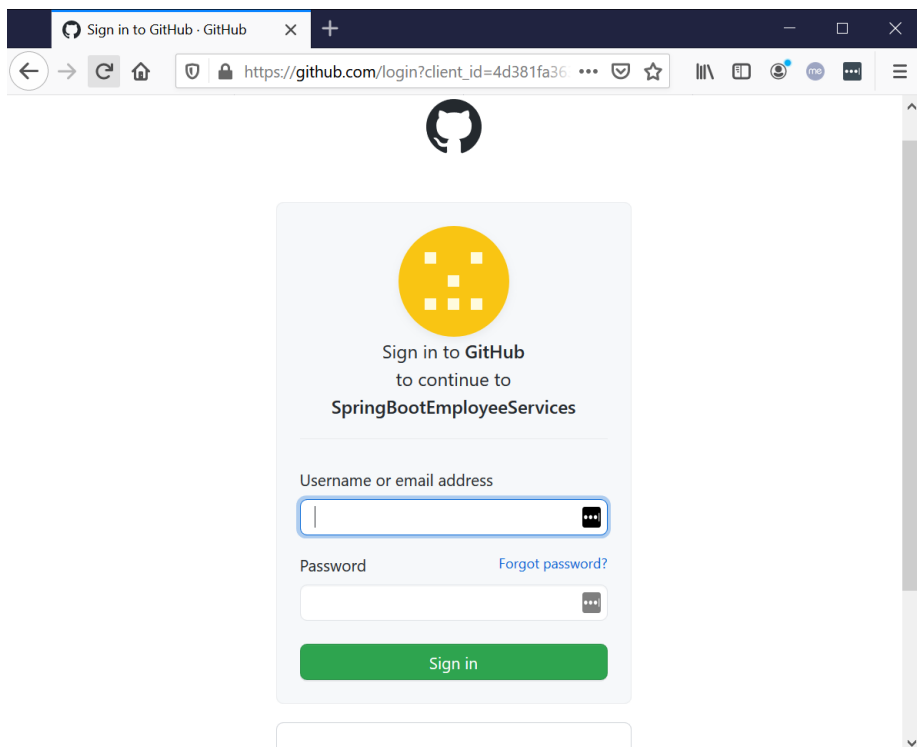
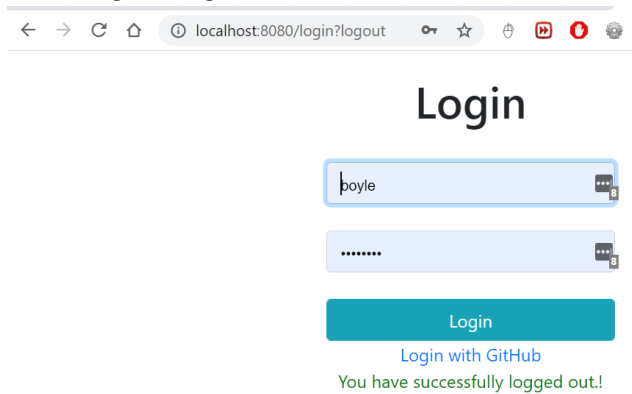


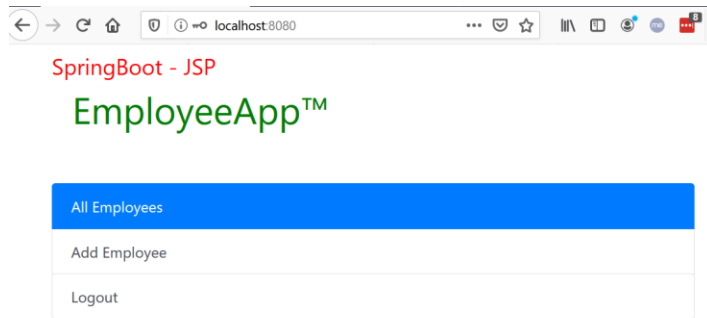
```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests().antMatchers("/", "/login").permitAll()
        .and()
        .formLogin()
        .loginPage("/login")
        .loginProcessingUrl("/authenticateTheUser")
        .permitAll()
        .and()
        .logout()
        .permitAll()
        .and()
        .oauth2Login()
        .loginPage("/login")
        .userInfoEndpoint()
        .userService(userService);
}
}

```

7. Test Login using GitHub





8. Spring Security – Summary

To customize Spring Security, we need a configuration class annotated with `@EnableWebSecurity` annotation in our classpath. Also, to simplify the customization process, the framework exposes a `WebSecurityConfigurerAdapter` class.

We will extend this adapter and override both of its functions so as to:

- **Configure** the `AuthenticationManager` with the correct provider
- **Configure** web security (public URLs, private URLs, authorization, etc.)

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        // TODO configure authentication manager
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        // TODO configure web security
    }
}
```

9. Spring Security – Ref.

<https://www.studytonight.com/spring-framework/spring-security-introduction>

<https://www.section.io/engineering-education/an-all-in-one-spring-security-crash-course-for-java-developers/>

<https://www.concretepage.com/spring-5/spring-security-ldap-authentication>

<https://medium.com/codeops/spring-boot-security-ldap-example-1ce1bdfc5816>

<https://www.logicbig.com/tutorials/spring-framework/spring-security.html>

<https://examples.javacodegeeks.com/enterprise-java/spring/boot/spring-boot-in-memory-basic-authentication-example/>

<https://www.techgeeknext.com/spring/spring-boot-security-token-authentication-jwt>

<https://www.codejava.net/frameworks/spring-boot/oauth2-login-with-github-example>

7. SpringBoot – Session Management

We all know that **HTTP is a stateless protocol**. All requests and responses are independent. The server cannot distinguish between new visitors and returning visitors. But sometimes we may need to keep track of client's activity across multiple requests. This is achieved using Session Management. It is a mechanism used by the Web container to store session information for a particular user.

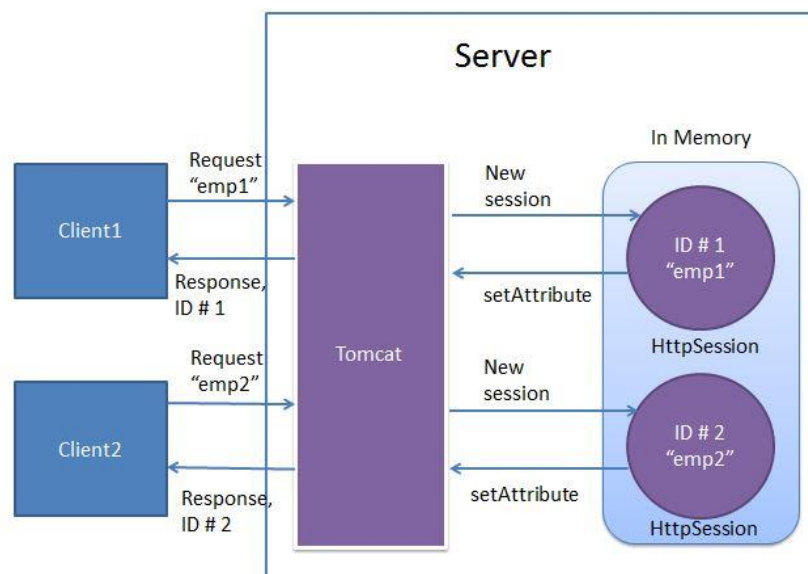
Session management can be achieved in one of the following ways-

- Cookies
- Hidden form field
- URL Rewriting
- HttpSession

In this example we will be making use of HttpSession to achieve Session management. Also we will be using the **Spring Session module**. Spring Session consists of the following modules:

- **Spring Session Core** - provides core Spring Session functionalities and APIs
- **Spring Session Data Redis** - provides **SessionRepository** and **ReactiveSessionRepository** implementation backed by Redis and configuration support
- **Spring Session JDBC** - provides **SessionRepository** implementation backed by a relational database and configuration support
- **Spring Session Hazelcast** - provides **SessionRepository** implementation backed by Hazelcast and configuration support

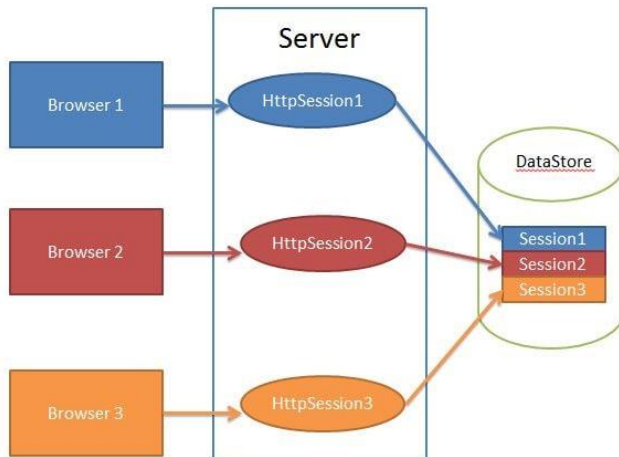
here we will be using **Spring Session JDBC** to store spring session information. By default, Apache Tomcat stores HTTP session objects in memory.



In order to achieve writing the session objects to MySQL database, **we dont have to write any code**. Spring Boot provides us this functionality out of the box by specifying the following configuration property

```
spring.session.store-type=jdbc
```

Spring session replaces the **HttpSession** implementation by a custom implementation. To perform this task spring session creates a SessionRepositoryFilter bean named as springSessionRepositoryFilter.



Example

```
<dependency>
  <groupId>org.springframework.session</groupId>
  <artifactId>spring-session-core</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.session</groupId>
  <artifactId>spring-session-jdbc</artifactId>
</dependency>
```

```
@RestController
public class SpringSessionController {

    @GetMapping("/")
    public List<String> process(Model model, HttpSession session) {
        @SuppressWarnings("unchecked")
        List<String> messages = session.getAttribute("MY_SESSION_MESSAGES");

        if (messages == null) {
            messages = new ArrayList<>();
        }
        model.addAttribute("sessionMessages", messages);

        return messages;
    }

    @PostMapping("/persistMessage")
    public List<String> persistMsg(@RequestParam("msg") String msg, HttpServletRequest request) {

        List<String> messages = request.getSession().getAttribute("MY_SESSION_MESSAGES");
        if (messages == null) {
            messages = new ArrayList<>();
        }
    }
}
```

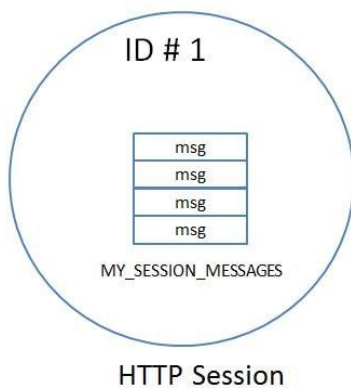
```

        request.getSession().setAttribute("MY_SESSION_MESSAGES", messages);
    }
    messages.add(msg);
    request.getSession().setAttribute("MY_SESSION_MESSAGES", messages);
    return messages;
}

@PostMapping("/destroy")
public String destroySession(HttpServletRequest request) {
    request.getSession().invalidate();
    return "Session Destroyed";
}
}
}

```

So if not already present, we create an ArrayList named `MY_SESSION_MESSAGES` in a `HTTPSession` and persist messages in this list



Application.properties

```

spring.datasource.url= jdbc:mysql://localhost:3306/microservices?useSSL=false
spring.datasource.username= root
spring.datasource.password= root
spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MySQL5InnoDBDialect
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto= update

spring.session.store-type=jdbc
spring.session.jdbc.initialize-schema=always
server.servlet.session.timeout=90s

```

How it works

On Starting the application Spring Session creates 2 tables for storing session related information.(`spring_session`, `spring_session_attributes`)

#	Name	Datatype	Length/Set
1	PRIMARY_ID	CHAR	30
2	SESSION_ID	CHAR	36
3	CREATION_TIME	BIGINT	20
4	LAST_ACCESS_TIME	BIGINT	20
5	MAX_INACTIVE_INTERVAL	INT	11
6	EXPIRY_TIME	BIGINT	20
7	PRINCIPAL_NAME	VARCHAR	100

#	Name	Datatype
1	SESSION_PRIMARY_ID	CHAR
2	ATTRIBUTE_NAME	VARCHAR
3	ATTRIBUTE_BYTES	BLOB

Instead of Storing session data in browser, it will store in DB. We can retrieve all attributes by comparing SESSION_ID from HTTP Headers & Database SESSION_ID.

The Data in database will be removed after session timeout or Session close
`server.servlet.session.timeout=90s`

microservices.spring_session: 1 rows total (approximately)					
PRIMARY_ID	SESSION_ID	CREATION_TIME	LAST_ACCESS_TIME	MAX_INACTIVE_INTERVAL	
5a1332ee-ac46-405e-8ac1-e2cc71aadeab	501d3abb-bc6d-4104-aff0-af754562c6d8	1,647,942,132,700	1,647,942,132,700	190	

microservices.spring_session_attributes: 1 rows total (approximately)		
SESSION_PRIMARY_ID	ATTRIBUTE_NAME	ATTRIBUTE_BYTES
5a1332ee-ac46-405e-8ac1-e2cc71aadeab	MY_SESSION_MESSAGES	0xACED0005737200136A6176612E7574696C2E417272...

7. Spring Boot – Advanced

Spring Boot – Logging

Spring boot's provide default logging mecahnisum, which is written with *Apache Commons Logging*

SpringBoot supports **ERROR, WARN, INFO, DEBUG, or TRACE** as logging level. By default, logging level is set to **INFO**. It means that code>DEBUG and **TRACE** messages are not visible.

To enable debug or trace logging, we can set the logging level in `application.properties` file. Also, we can pass the `-debug` or `-trace` arguments on the command line while starting the application.

```
# In Console
-Dlogging.level.org.springframework=ERROR
-Dlogging.level.com.howtodoinjava=TRACE

# In properties file
logging.level.org.springframework=ERROR
logging.level.com.howtodoinjava=TRACE
```

Example

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@SpringBootApplication
public class Application
{
    private static final Logger LOGGER=LoggerFactory.getLogger(Application.class);

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
        LOGGER.info("Simple log statement with inputs {}, {} and {}", 1,2,3);
    }
}
```

Switch Between Environmets

<https://stackabuse.com/how-to-access-property-file-values-in-spring-boot/>

Spring Boot already has support for profile based properties.

Simply add an `application-[profile].properties` file and specify the profiles to use using the `spring.profiles.active` property.

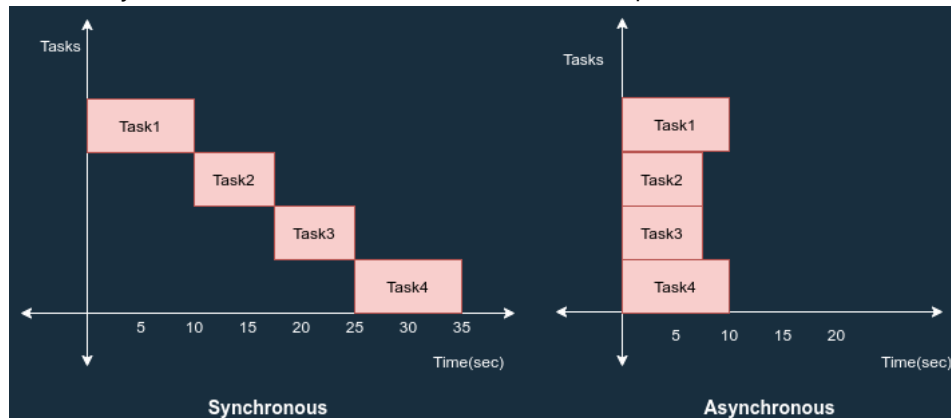
-Dspring.profiles.active=local

This will load the `application.properties` and the `application-local.properties` with the latter overriding properties from the first.

Spring Boot – Asynchronous Implementation

This article is about how asynchronous behaviour can be achieved in spring boot. But first of all, let's see the difference between synchronous and asynchronous.

- **Synchronous Programming:** In synchronous programming, tasks are performed one at a time and only when one is completed the next is unblocked.
- **Asynchronous Programming:** In asynchronous programming, multiple tasks can be executed simultaneously. You can move to another task before the previous one finishes.



In spring boot, we can achieve asynchronous behaviour using `@EnableAsync` & `@Async` annotations.

- `@EnableAsync` - to enable async support by annotating the main application class.
- `@Async` - annotate the method

When you annotate a method with `@Async` annotation, it creates a **proxy** for that object based on `"proxyTargetClass"` property.

When spring executes this method, by default it will be searching for associated thread pool definition. Either a unique spring framework **TaskExecutor** bean in the context or `Executor` bean named `"taskExecutor"`. If neither of these two is resolvable, default it will use spring framework **SimpleAsyncTaskExecutor** to process async method execution.

Example – Without Async

```
@RestController
public class DecomController {
    @Autowired
    DecomService service;

    @GetMapping("/decomTrial")
    public String decomClinicalTrial() {
        System.out.println(" *** decomClinicalTrial :: Started*** ");
        service.ArchiveUsers();
        service.ArchiveReports();
        System.out.println(" *** decomClinicalTrial :: Completed.*** ");
        return "decomClinicalTrial ComPLETED.";
    }
}
```



```

@Service
public class DecomService {

    public void ArchiveUsers() {
        S.o.p(Thread.currentThread().getName() + " :ArchiveUsers :Start "+new Date());
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        S.o.p (Thread.currentThread().getName() + " :ArchiveUsers :End " + new Date());
    }

    public void ArchiveReports() {
        S.o.p(Thread.currentThread().getName() + ":ArchiveReports : Start" + new Date());
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        S.o.p(Thread.currentThread().getName() + " :ArchiveReports : End" + new Date());
    }
}

@SpringBootApplication
public class TrialServicesApplication {
    public static void main(String[] args) {
        SpringApplication.run(TrialServicesApplication.class, args);
        System.out.println("*****");
        System.out.println("Trial Services Started ... ");
        System.out.println("*****");
    }
}

```

```

*** decomClinicalTrial: Started***
http-nio-8080-exec-1:ArchiveUsers :Start 24 Sep 2021 07:52:32 GMT
http-nio-8080-exec-1:ArchiveUsers :End 24 Sep 2021 07:52:37 GMT
http-nio-8080-exec-1:ArchiveReports :Start24 Sep 2021 07:52:37 GMT
http-nio-8080-exec-1:ArchiveReports :End24 Sep 2021 07:52:42 GMT
*** decomClinicalTrial: Completed.***

```

Here all methods execute by single Thread [http-nio-8080-exec-1](#)

With @Aync

Place [@EnableAsync](#) on the Top of SpringBoot Main class - [TrialServicesApplication](#)

```

@EnableAsync
@SpringBootApplication
public class TrialServicesApplication {
    public static void main(String[] args) {
        SpringApplication.run(TrialServicesApplication.class, args);
        System.out.println("*****");
        System.out.println("Trial Services Started ... ");
        System.out.println("*****");
    }
}

```

Place [@Async](#) on the top of each methods, which we want to execute in parallel.

```

@Service
public class DecomService {
    @Async
    public void ArchiveUsers() {
        S.o.p(Thread.currentThread().getName() + " :ArchiveUsers :Start " + new Date());
        try {
            Thread.sleep(5000);
        }
    }
}

```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    S.o.p(Thread.currentThread().getName() + " :ArchiveUsers :End " + new Date());
}

@Async
public void ArchiveReports() {
    S.o.p(Thread.currentThread().getName() + ":ArchiveReports : Start" + new Date());
    try {
        Thread.sleep(5000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    S.o.p(Thread.currentThread().getName() + " :ArchiveReports : End" + new Date());
}
}

```

```

*** decompClinicalTrial :: Started***
*** decompClinicalTrial :: Completed.***
task-1 :ArchiveUsers : Start 24 Sep 2021 08:01:20 GMT
task-2 :ArchiveReports : Start24 Sep 2021 08:01:20 GMT
task-2 :ArchiveReports : End24 Sep 2021 08:01:25 GMT
task-1 :ArchiveUsers :End 24 Sep 2021 08:01:25 GMT

```

In above ArchiveUsers, ArchiveReports are executed parallel by two different threads **task-1**, **task-2**

Async methods with Return Types

Previously, The async method only used a void return type. And in my opinion, that is the best way to write self sufficient asynchronous functions. However, If you want to handle the results from an @Async function, you are in luck. Because Spring framework provides out of the box support for these situations using **Future** type. This is possible by wrapping the result inside of an **AsyncResult** class.

```

@Async
public Future<String> longRunningProcessThatReturns() {
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return new AsyncResult<>("I take 10 seconds to return on a Thread named : " +
Thread.currentThread().getName());
}

```

<https://medium.com/globant/asynchronous-calls-in-spring-boot-using-async-annotation-d34d8a82a60c>

<https://springhow.com/spring-async/>

<https://howtodoinjava.com/spring-boot2/rest/enableasync-async-controller/>

SpringBoot Reactive Web

Spring WebFlux provides reactive, async, non-blocking programming support for web applications in an annotated Controller format similar to SpringMVC.

This approach is similar to how Node.js uses an async, non-blocking model which helps make it more scalable. Spring WebFlux uses a similar model but with multiple event loops.

Spring WebFlux moves away from the thread-per-request blocking model in traditional SpringMVC(with Tomcat by default) and moves towards a multi-EventLoop, async, non-blocking(with Netty by default) paradigm with back-pressure that is more scalable and efficient than traditional blocking code.

Spring Boot - Devtools

If you have worked on latest UI development frameworks e.g. Node, angular, gulp etc. then you must have appreciated the auto-reload of UI in browser whenever there is change in some code. Its pretty useful and saves a lot of time.

To enable dev tools in spring boot application is very easy. Just add the **spring-boot-devtools** dependency in your build file.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
  </dependency>
</dependencies>
```

Static Resource Caching

To improve the performance, dev tools cache the static content/template files to serve them faster to browser/client.

There are many such UI template libraries that support this feature. e.g. thymeleaf, freemarker, groovy, mustache etc.

```
#spring.freemarker.cache = true //set true in production environment
spring.freemarker.cache = false //set false in development environment; It is false by default.

//Other such properties
spring.thymeleaf.cache = false
spring.mustache.cache = false
spring.groovy.template.cache = false
```

Automatic UI refresh

The **spring-boot-devtools** module includes an embedded LiveReload server that can be used to trigger a browser refresh when a resource is changed

```
spring.devtools.livereload.enabled = false #Set false to disable live reload
```

SpringBoot – Project Lombok

Project Lombok is a Java library tool that generates code for minimizing boilerplate code. The library replaces boilerplate code with easy-to-use annotations.

For example, by adding a couple of annotations, you can get rid of code clutters, such as getters and setters' methods, constructors, hashCode, equals, and toString methods, and so on.

- **val** : Finally! Hassle-free final local variables.
- **var** : Mutably! Hassle-free local variables.
- **@Data** : All together now: A shortcut for **@ToString**, **@EqualsAndHashCode**, **@Getter** on all fields, and **@Setter** on all non-final fields, and **@RequiredArgsConstructor**!
- **@NonNull** : How I learned to stop worrying and love the **NullPointerException**.
- **@Cleanup** : Automatic resource management: Call your **close()** methods safely
- **@Getter/@Setter** : Never write **public int getFoo() {return foo;}** again.
- **@ToString** : generate a **toString** for you!
- **@EqualsAndHashCode** : Generates **hashCode** and **equals** implementations from the fields of your object.
- **@NoArgsConstructor**, **@RequiredArgsConstructor** and **@AllArgsConstructor** Constructors made to order: Generates constructors that take no arguments, one argument per final / non-nullfield, or one argument for every field.
- **@Value** :Immutable classes made very easy.
- **@Builder** No-hassle fancy-pants APIs for object creation!
- **@SneakyThrows** :To boldly throw checked exceptions where no one has thrown them before!
- **@Synchronized**:**synchronized** done right: Don't expose your locks.
- **@With**:Immutable 'setters' - methods that create a clone but with one changed field.
- **@Getter(lazy=true)**:Laziness is a virtue!
- **@Log** :Captain's Log, stardate 24435.7: "What was that line again?"
- **experimental** :Head to the lab: The new stuff we're working on.

Maven dependency

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

Spring Vault

HashiCorp's Vault is a tool to store and secure secrets. Vault, in general, solves the software development security problem of how to manage secrets. **Spring Vault** provides Spring abstractions to the HashiCorp's Vault.

<https://dzone.com/articles/spring-cloud-hashicorp-vault-hello-world-example>

Diffrence between @Component & @Autowired

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

@Autowired is, after creating bean object in **@Component** time, it will autowires/ injejects 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).

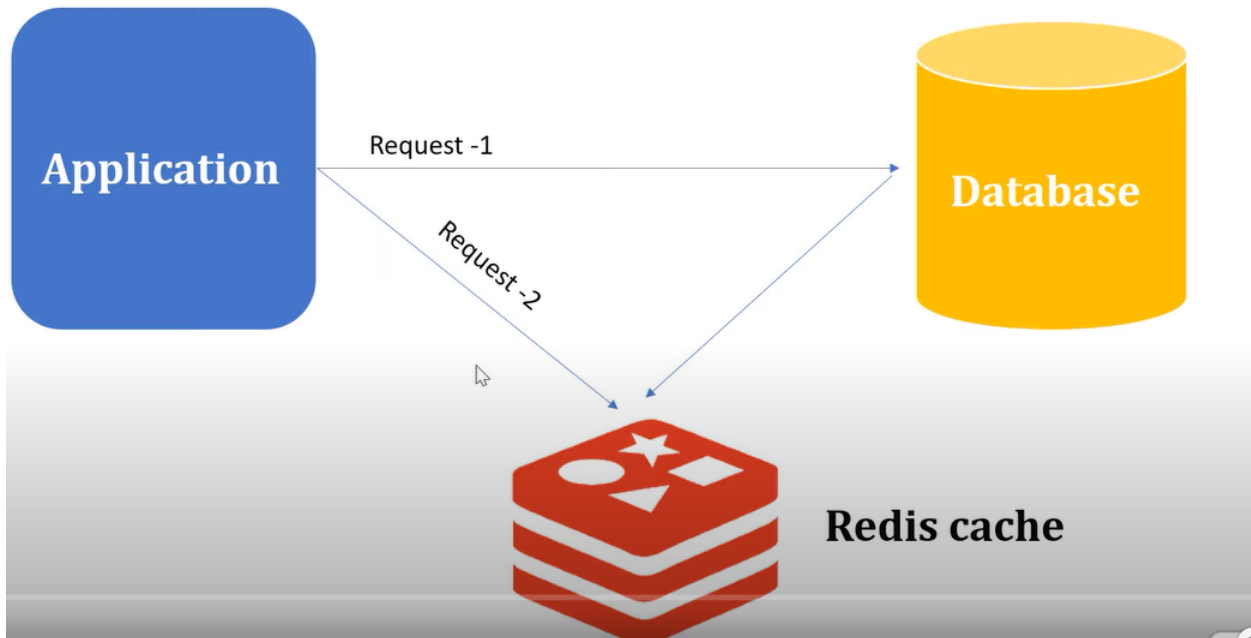
@EnableAutoConfiguration has a exclude attribute to disable an auto-configuration explicitly otherwise we can simply exclude it from the pom.xml, for example if we do not want Spring to configure the tomcat then exclude spring-bootstarter-tomcat from spring-boot-starter-web.

@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.

8. Spring Data Redis

Spring Data Redis - Spring Boot Cache with Redis is used to reduce the no.of round trips between Application & Database



The screenshot shows an IDE with the following components:

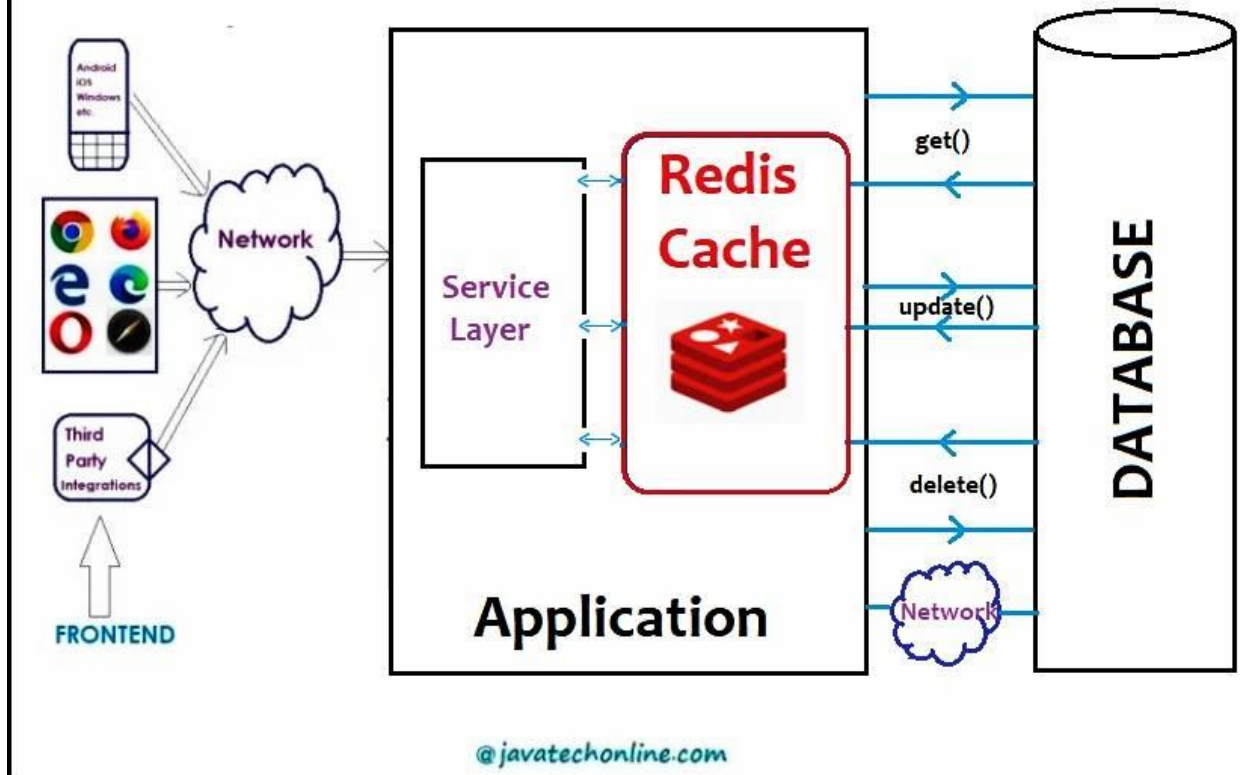
- Code Editor:** Displays `SpringDataRedisExampleApplication.java` with methods:


```

      @PostMapping
      public Product save(Product product) {
          return dao.save(product);
      }

      @GetMapping
      public List<Product> findProducts() {
          return dao.findAll();
      }
      
```
- Run Console:** Shows logs for `SpringDataRedisExampleApplication` starting at 19:42:40.160. It indicates that a request was `called findProductById() from DB`.
- REST Client:** Shows a request to `http://localhost:8080/api/products/1` with a response of `Status: 200 OK` and `Time: 504 ms`. A red callout bubble points to the response time, stating **Taking More Time Without cache**.

Caching with Redis Cache



When we perform a DB retrieve operation via an Application, the Redis Cache stores the result in its cache. Further, when we perform the same retrieve operation, it returns the result from the cache itself and ignore the second call to database.

What is Redis Database?

- Redis Database is an in-memory database that persists on **disk**. It means when we use Redis Database, we occupy **a memory on the disk** to use as a Database.
- The data model is **key-value**, but many several kind of values are supported such as Strings, Lists, Sets, Sorted Sets, Hashes, Streams, HyperLogLogs, Bitmaps etc.

What is Redis Server?

The full form of Redis is **RE**mote **DI**ctionary **S**erver. When we use Redis in any form such as database, cache or Message Broker, we need to download a Redis Server in our system. People in the industry just call it Redis Server.

- Visit any of the link below to download Redis
[Download Redis Server](#) : version 3.2.100
[Download Redis Server](#) : version 5.0.10
- unzip 'Redis-x64-5.0.10', you will find redis-server.exe
- In order to start Redis Server, double click on 'redis-server.exe' to start Redis Server


```
[1424] 21 Feb 18:26:12.895 # Warning: no config file specified, using the default config. In order to
file use C:\Users\kavetis\Downloads\Redis-x64-3.0.504\redis-server.exe /path/to/redis.conf

Redis 3.0.504 (00000000/0) 64 bit

Running in standalone mode
Port: 6379
PID: 1424

http://redis.io

[1424] 21 Feb 18:26:12.900 # Server started, Redis version 3.0.504
[1424] 21 Feb 18:26:12.900 * The server is now ready to accept connections on port 6379
```

```
SpringDataRedisExampleApplication.java × Product.java × pom.xml × RedisConfig.java ×
4 import org.springframework.context.annotation.Configuration;
5 import org.springframework.data.redis.connection.RedisStandaloneConfiguration;
6 import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;
7 import org.springframework.data.redis.repository.configuration.EnableRedisRepositories;
8
9 @Configuration
10 @EnableRedisRepositories
11 public class RedisConfig {
12
13     @Bean
14     public JedisConnectionFactory connectionFactory(){
15         RedisStandaloneConfiguration configuration=new RedisStandaloneConfiguration();
16         configuration.setHostName("localhost");
17         configuration.setPort(6379);
18         return new JedisConnectionFactory(configuration);
19     }
20
21 }
```

@EnableCaching : We apply this annotation at the main class (starter class) of our application in order to tell Spring Container that we need Caching feature in our application.

@Cacheable :@Cacheable is used to **fetch(retrieve)** data from the DB to application and store in Redis Cache. We apply it on the **methods** that get (retrieve) data from DB. (**SELECT oprations**).

@CachePut :We use @CachePut in order to update data in the Redis Cache while there is any update of data in DB. We apply it on the methods that make modifications in DB (**DML oprations**).

@CacheEvict :We use @CachePut in order to remove data in the Redis Cache while there is any removal of data in DB. We apply it on the methods that delete data from DB.

```
@Service
public class InvoiceServiceImpl implements InvoiceService {

    @Autowired
    private InvoiceRepository invoiceRepo;

    @Override
    public Invoice saveInvoice(Invoice inv) {
```



```

        return invoiceRepo.save(inv);
    }

    @Override
    @CachePut(value="Invoice", key="#invId")
    public Invoice updateInvoice(Invoice inv, Integer invId) {
        Invoice invoice = invoiceRepo.findById(invId)
            .orElseThrow(() -> new InvoiceNotFoundException("Invoice Not Found"));
        invoice.setInvAmount(inv.getInvAmount());
        invoice.setInvName(inv.getInvName());
        return invoiceRepo.save(invoice);
    }

    @Override
    @CacheEvict(value="Invoice", key="#invId")
    // @CacheEvict(value="Invoice", allEntries=true) //in case there are multiple entires to
delete
    public void deleteInvoice(Integer invId) {
        Invoice invoice = invoiceRepo.findById(invId)
            .orElseThrow(() -> new InvoiceNotFoundException("Invoice Not Found"));
        invoiceRepo.delete(invoice);
    }

    @Override
    @Cacheable(value="Invoice", key="#invId")
    public Invoice getOneInvoice(Integer invId) {
        Invoice invoice = invoiceRepo.findById(invId)
            .orElseThrow(() -> new InvoiceNotFoundException("Invoice Not Found"));
        return invoice;
    }

    @Override
    @Cacheable(value="Invoice")
    public List<Invoice> getAllInvoices() {
        return invoiceRepo.findAll();
    }
}

```

9. Spring Batch

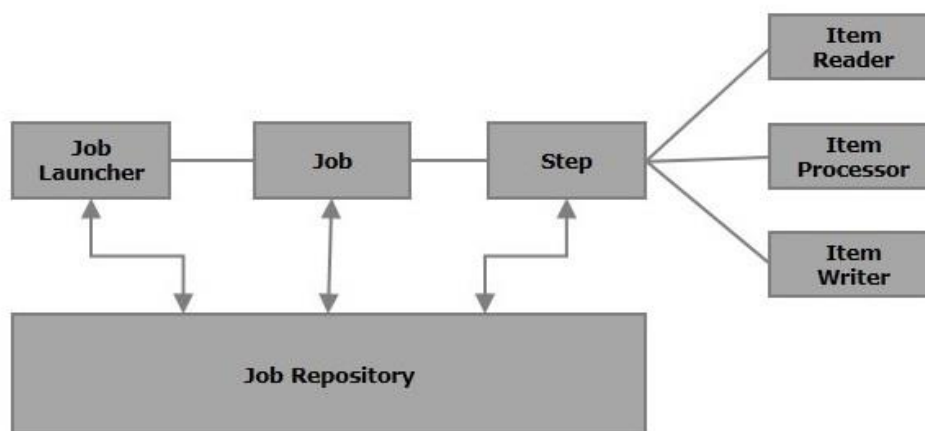
Batch processing is a processing mode which involves execution of series of automated complex jobs without user interaction. A batch process handles bulk data and runs for a long time.

Spring Batch applications support –

- Automatic retry after failure.
- Tracking status and statistics during the batch execution and after completing the batch processing.
- To run concurrent jobs.
- Services such as logging, resource management, skip, and restarting the processing.

Example: Import Users via Excel

Components of Spring Batch



1.Job

In a Spring Batch application, a job is the batch process that is to be executed. It runs from start to finish without interruption. This job is further divided into **steps** (or a job contains steps).

We will configure a job in Spring Batch using an XML file or a Java class. Following is the XML configuration of a Job in Spring Batch.

```
<job id = "jobid">
  <step id = "step1" next = "step2"/>
  <step id = "step2" next = "step3"/>
  <step id = "step3"/>
</job>
```

2.Step

- A **step** is an independent part of a job which contains the necessary information to define and execute the job (its part).
- As specified in the diagram, each step is composed of an **ItemReader**, **ItemWriter**, **ItemProcessor** (optional). **A job may contain one or more steps.**

3.Readers, Writers, and Processors

- **Item reader** reads data into a Spring Batch application from a particular source.
- **Item writer** writes data from the Spring Batch application to a particular destination.
- **Item processor** is a class which contains the processing code which processes the data read into the spring batch. If the application reads "**n**" records, then the code in the processor will be executed on each record.

For example, if we are writing a job with a simple step in it where we read data from MySQL database and process it and write it to a file (flat), then our step uses –

- A **reader** which reads from Excel file.

- A **writer** who writes to a Database.
- A **custom processor** which processes the data as per our wish.

```
<job id = "helloWorldJob">
  <step id = "step1">
    <tasklet>
      <chunk reader = "CsvReader" writer = "mysqlWriter"
        processor = "CustomitemProcessor" ></chunk>
    </tasklet>
  </step>
</ job>
```

4. Job repository

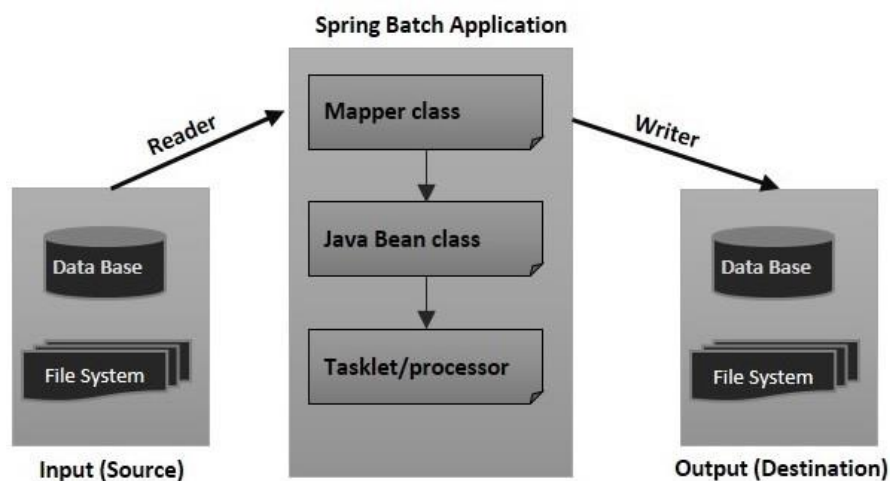
A Job repository in Spring Batch provides Create, Retrieve, Update, and Delete (CRUD) operations for the JobLauncher, Job, and Step implementations. We will define a job repository in an XML file.

```
<job-repository id = "jobRepository"
  data-source = "dataSource"
  transaction-manager = "transactionManager"
  isolation-level-for-create = "SERIALIZABLE"
  table-prefix = "BATCH_"
  max-varchar-length = "1000"/>
```

5. JobLauncher

JobLauncher is an interface which launches the Spring Batch job with the **given set of parameters**. **SampleJoblauncher** is the class which implements the **JobLauncher** interface. Following is the configuration of the JobLauncher.

```
<bean id = "jobLauncher"
  class = "org.springframework.batch.core.launch.support.SimpleJobLauncher">
  <property name = "jobRepository" ref = "jobRepository" />
</bean>
```



Spring Batch Example – Excel/CSV File To MySQL Database

CSV file data : <src/main/resources/cvs/Users.csv>

uid	address	age	name
401	Mumbai	35	Smith
402	Delhi	36	Kivell
403	Bangalore	37	Gill
404	Hyderabad	38	Jardine
405	Ahmedabad	39	Andrews
406	Chennai	40	Gill
407	Kolkata	41	Morgan
408	Surat	42	Andrews
409	Pune	43	Jardine

Create Model class to Hold CSV Data

```
public class User {
    private Integer uid;
    private String address;
    private Integer age;
    private String name;
}
```

Job Cofiguration XML files

database.xml: /src/main/resources/spring/batch/config/database.xml

- Contains *dataSource* bean contains DB connection details.
- *transactionManager* - Transaction managet
- create job-meta tables configuration.

```
<beans>
  <!-- connect to database -->
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/microservices" />
    <property name="username" value="root" />
    <property name="password" value="root" />
  </bean>
  <bean id="transactionManager"
    class="org.springframework.batch.support.transaction.ResourcelessTransactionManager" />

  <!-- create job-meta tables automatically -->
  <jdbc:initialize-database data-source="dataSource">
    <jdbc:script location="org/springframework/batch/core/schema-drop-mysql.sql" />
    <jdbc:script location="org/springframework/batch/core/schema-mysql.sql" />
  </jdbc:initialize-database>
</beans>
```

Context.xml

It Spring Batch Core Settings. It Defines *jobRepository* and *jobLauncher*.

```
<beans>
```

```

<!-- stored job-meta in database -->
<bean id="jobRepository"

class="org.springframework.batch.core.repository.support.JobRepositoryFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="transactionManager" ref="transactionManager" />
    <property name="databaseType" value="mysql" />
</bean>

<!-- stored job-meta in memory -->
<!--
<bean id="jobRepository"

class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
    <property name="transactionManager" ref="transactionManager" />
</bean>
-->

<bean id="jobLauncher"
class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
    <property name="jobRepository" ref="jobRepository" />
</bean>
</beans>

```

jobConfig.xml

This is the main xml file to configure the Spring batch job. This `jobConfig.xml` file define a job to read a `user.csv` file, match it to `User` plain pojo and write the data into MySQL database.

```

<beans>

<bean id="userOb" class="com.spring.model.User" scope="prototype" />
    <job id="CsvToMySQL-job">
        <step id="step1">
            <tasklet>
                <chunk reader="cvsFileItemReader" writer="mysqlItemWriter"
                    commit-interval="2">
                    </chunk>
            </tasklet>
        </step>
    </job>

<bean id="cvsFileItemReader" class="org.springframework.batch.item.file.FlatFileItemReader">
<!-- Read a csv file -->
    <property name="resource" value="classpath:cvs/Users.csv" />
    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
            <!-- split it -->
            <property name="lineTokenizer">
                <bean class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
                    <property name="names" value="uid,address,age,name" /></bean>
                </property>

<property name="fieldSetMapper">
            <!-- map to an object -->
            <bean class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
                <property name="prototypeBeanName" value="userOb" />
            </bean>
        </property>
    </property>

```

```

        </bean>
    </property>
</bean>

<bean id="mysqlItemWriter"
    class="org.springframework.batch.item.database.JdbcBatchItemWriter">
    <property name="dataSource" ref="dataSource" />
    <property name="sql">
        <value>
            <![CDATA[
                insert into batch_user(uid,address,age,name) values
                (:uid, :address, :age, :name)
            ]]>
        </value>
    </property>
    <!-- It will take care matching between object property and sql name parameter -
->
    <property name="itemSqlParameterSourceProvider">
        <bean
            class="org.springframework.batch.item.database.BeanPropertyItemSqlParameterSourceProvid
er" />
    </property>
</bean>
</beans>

```

Create a main class , which calls CsvToMySQL-job via jobLauncher

```

public class CSVToMySQLExample {
    public static void main(String[] args) {

        String[] springConfig =
            {
                "spring/batch/config/database.xml",
                "spring/batch/config/context.xml",
                "spring/batch/jobs/jobConfig.xml"
            };

        ApplicationContext context = new ClassPathXmlApplicationContext(springConfig);

        JobLauncher jobLauncher = (JobLauncher) context.getBean("jobLauncher");
        Job job = (Job) context.getBean("CsvToMySQL-job");

        try {
            JobExecution execution = jobLauncher.run(job, new JobParameters());
            System.out.println("Exit Status : " + execution.getStatus());

        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("Done");
    }
}

```

Does Spring Batch require a database?

Spring Batch by default uses a database to store metadata on the configured batch jobs. In this example, we will run Spring Batch without a database. Instead, an in-memory Map based repository is used.

How does Spring Batch handle transactions?

Spring Batch handles transactions **at the step level**. This means that Spring Batch will never use only one transaction for a whole job (unless the job has a single step). Remember that you're likely to implement a Spring Batch job in one of two ways: using a **tasklet** or using a **chunk-oriented** step.

```
<job id="CsvToMySQL-job">
  <step id="step1">
    <tasklet>
      <chunk reader="csvFileItemReader" writer="mysqlItemWriter"
        commit-interval="2">
      </chunk>
    </tasklet>
  </step>
</job>
```

How One can keep track of failed records in Spring Batch?

Method1 : We can keep track of records which was failed during the reading step of a job. We need to use **SkipListener** for this.

```
public class SkipListener implements org.springframework.batch.core.SkipListener {

    public void onSkipInProcess(Object arg0, Throwable arg1) {
    }

    public void onSkipInRead(Throwable arg0) {
        System.out.println(arg0);
    }

    public void onSkipInWrite(Object arg0, Throwable arg1) {
    }

}
```

I want to store the line skipped by reader in another csv file.

SpringBatch provides methods on the [StepExecution](#) objects :

- Read : stepExecution.[getReadCount\(\)](#)
- Read Failed : stepExecution.[getReadSkipCount\(\)](#)
- Processed : stepExecution.[getProcessCount\(\)](#)
- Processed Failed : stepExecution.[getProcessSkipCount\(\)](#)
- Written : stepExecution.[getWriteCount\(\)](#)
- Written Failed : stepExecution.[getWriteSkipCount\(\)](#)

Method 2 : Retry Sample

The purpose of this sample is to show how to use the automatic retry capabilities of Spring Batch. The retry is configured in the step through the SkipLimitStepFactoryBean:

```
<bean id="step1" parent="simpleStep"
      class="org.springframework.batch.core.step.item.FaultTolerantStepFactoryBean">
  ...
  <property name="retryLimit" value="3" />
  <property name="retryableExceptionClasses" value="java.lang.Exception" />
</bean>
```

Failed items will cause a rollback for all Exception types, up to a limit of 3 attempts. On the 4th attempt, the failed item would be skipped, and there would be a callback to a `ItemSkipListener` if one was provided (via the "listeners" property of the step factory bean).

10. Spring Cloud – MicroServices

The major use-case for Spring Cloud is the ready-to-use solution that it provides to common problems observed in distributed environments like load balancing, service discovery, circuit breaking, etc., which can easily be integrated in an existing Spring project.

Before we look at Spring Cloud, let's have a brief overview on Microservice Architecture and the role of Spring Boot in creating microservices.

<https://github.com/sivaprasadreddy/spring-boot-microservices-series>

<https://www.sivalabs.in/2018/03/microservices-using-springboot-spring-cloud-part-1-overview/>

Microservices

Monoliths

Traditionally we are building large enterprise applications in modularised fashion, but finally deploy them together as a single deployment unit (EAR or WAR). These are called Monolithic applications.

There are some issues with the monolithic architecture such as:

- Large codebases become mess over the time
- Multiple teams working on single codebase become tedious
- It is not possible to scale up only certain parts of the application
- Technology updates/rewrites become complex and expensive tasks

MicroServices

Microservice architecture is a style of application development where the application is broken down into small services and these services have loose coupling among them. Following are the major advantages of using microservice architecture –

Advantages of MicroServices

- Comprehending smaller codebase is easy
- Can independently scale up highly used services

- Each team can focus on one (or few) MicroService(s)
- Technology updates/rewrites become simpler

SpringBoot and SpringCloud are a good choice for MicroServices

Spring Cloud

Spring Cloud provides a collection of components which are useful in building distributed applications in cloud. We can develop these components on our own, however that would waste time in developing and maintaining this boilerplate code.

That is where Spring Cloud comes into picture. It provides ready-to-use **cloud design patterns** for common problems which are observed in a distributed environment. Some of the patterns which it attempts to address are –

- Service Registration
- Centralized Configuration
- Load Balancing
- Circuit Breakers / Fault Tolerance
- Distributed Messaging (Kafka)
- Routing (API Gateway)
- Distributed Logging
- Distributed Lock

Spring Cloud Components

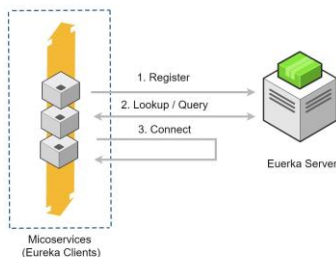
Look at the various components which Spring Cloud provides and the problems these components solve

Problem	Components
Distributed Cloud Configuration	Spring Cloud Configuration , Spring Cloud Zookeeper, Spring Cloud Config
Distributed Messaging	Spring Stream with Kafka , Spring Stream with RabbitMQ
Service Discovery	Spring Cloud Eureka , Spring Cloud Consul, Spring Cloud Zookeeper
Logging	Spring Cloud Zipkin , Spring Cloud Sleuth
Spring Service Communication(Circuit Breakers / Fault Tolerance)	Spring Hystrix , Spring Ribbon, Spring Feign, Spring Zuul(API gateWay)

Projects Of Spring Cloud

Spring Cloud Config Server: Configuring application properties, environment details etc. We can use Spring Cloud Config Server with **git** or **Consul** or **ZooKeeper** as config repository.

Service Registry and Discovery: As there could be many services and we need the ability to scale up or down dynamically, we need Service Registry and Discovery mechanism so that service-to-service communication should not depend on hard-coded hostnames and port numbers. Spring Cloud provides Netflix Eureka-based Service Registry and Discovery support with just minimal configuration. We can also use [Consul](#) or [ZooKeeper](#) for Service Registry and Discovery.



Circuit Breaker: In microservices based architecture, one service might depend on another service and if one service goes down then failures may cascade to other services as well. Spring Cloud provides Netflix **Hystrix** based Circuit Breaker to handle these kinds of issues.

Spring Cloud Data Streams: These days we may need to work with huge volumes of data streams using **Kafka** or Spark etc. Spring Cloud Data Streams provides higher-level abstractions to use those frameworks in an easier manner.

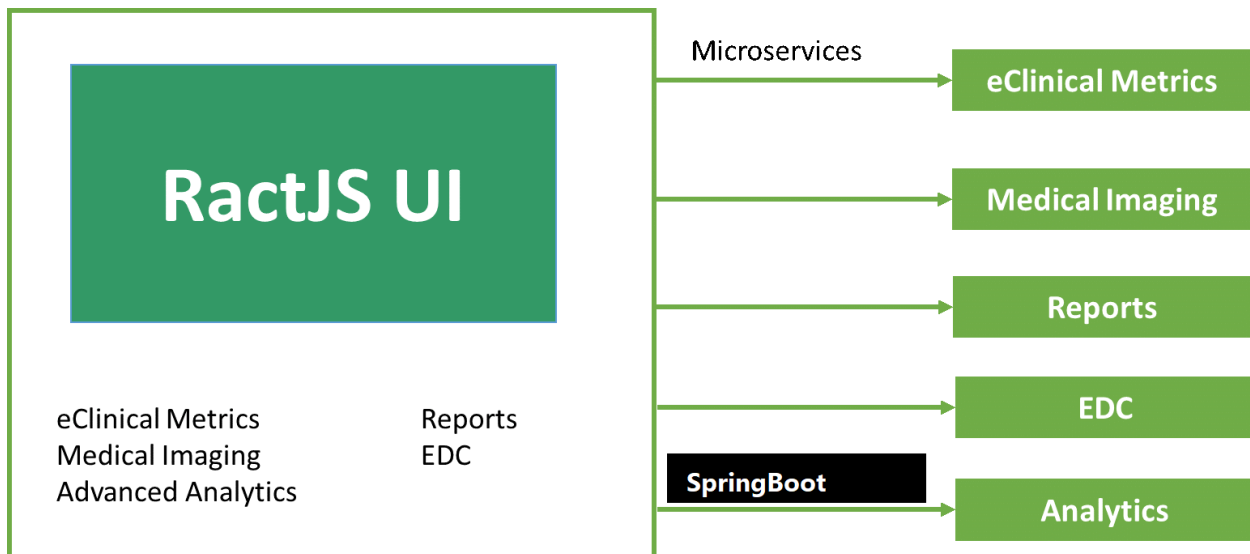
Spring Cloud Security: Some of the microservices needs to be accessible to authenticated users only and most likely we might want a **Single Sign-On** feature to propagate the authentication context across services. Spring Cloud Security provides authentication services using **OAuth2**.

Distributed Tracing: simple end-user action might trigger a chain of microservice calls, there should be a mechanism to trace the related call chains. We can use Spring Cloud **Sleuth** with **Zipkin** to trace the cross-service invocations.

Spring Cloud API gateway: There is a high chance that separate teams work on different microservices. There should be a mechanism for teams to agree upon API endpoint contracts so that each team can develop their APIs independently. Spring Cloud Contract helps to create such contracts and validate them by both service provider and consumer.

1. SpringCloud – Example

We are going to build a simple Microservices application and assume we are going to start with the following microservices:



1. **EDC-Microservice:** Electronic Data Capture (EDC) system is software that stores patient data collected in clinical trials. It provides REST API to provide Electronic Data Capture functionalities.
 - **TestReports**
 - **MedicalHistory**
 - **Visit Reports**

2. **MI- Microservice :** Medical imaging systems give health-care providers and researchers the information they need using high-frequency sound waves (ultrasound and echocardiography), magnetic fields (MRI), and electromagnetic radiation (conventional 2-D X-ray, 3-D computed tomography or CT scan, and fluoroscopy).
 - **RadiographyReports,**
 - **computed tomography (CT Scan Reports),**
 - **magnetic resonance imaging (MRI),**
 - **ultrasound,DentalScanReports**

3. **RTSM – Microservice** Randomization and Trial Supply Management system, is responsible for enabling critical functions of a clinical trial, from randomizing patients (who gets the active drug vs. the placebo), dispensing drug (ensuring patients receive the correct dose) and site resupply (controls the flow of drug from the manufacturer to the depot to the clinical site).

4. **UserManagent-ui:** It is customer facing front-end web application.

We are going to build various services and REST endpoints through various microservice concepts.

a.EDC-MicroService

Create a SpringBoot app with **Web, JPA, MySQL, Actuator, DevTools, Lombok** starters

Database : MySQL as Docker Container

We are going to use Docker and run MySQL as a Docker container.

docker-compose.yml

```
version: '3'
services:
  mysql:
    image: mysql:5.7
```

```

image: 'mysql:5.7'
container_name: mysqlldb
ports:
  - '3306:3306'
environment:
  MYSQL_ROOT_PASSWORD: passw0rd
  MYSQL_DATABASE: catalog

```

Download & Start MySQL Docker container

```
docker-compose up
```

Open any MySQL Editor & provide login details

Session name ^	Host	Network type:	Library:	Hostname / IP:	User:	Password:	Port:
freesqldb	sql6...	MariaDB or MySQL (TCP/IP)	libmariadb.dll	127.0.0.1	root	3306
Unnamed	127.0...						
Docker-MySQL	127.0...						
Xampp	127.0...						

Prompt for credentials
 Use Windows authentication
 Compressed client/server protocol

application.properties

```

spring.application.name=EDC-MicroService
server.servlet.context-path=/edc
server.port=8031

spring.datasource.url= jdbc:mysql://localhost:3306/microservices?useSSL=false
spring.datasource.username= root
spring.datasource.password= root
spring.jpa.properties.hibernate.dialect= org.hibernate.dialect.MySQL5InnoDBDialect
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto= update

# springdoc.swagger-ui.path=/swagger-ui.html

# Azure KeyValut Server
# azure.keyvault.enabled=true
# azure.keyvault.uri=https://idamtestvalut.vault.azure.net/
# azure.keyvault.client-id=put-your-azure-client-id-here
# azure.keyvault.client-key=put-your-azure-client-key-here

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
management.endpoints.web.exposure.include=*

spring.zipkin.base-url=http://localhost:9411/
spring sleuth.sampler.probability=1

# Consumer properties
spring.kafka.producer.bootstrap-servers=127.0.0.1:9092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.group-id=group_id
topic.name.consumer=user.provision.topic

# Common Kafka Properties
auto.create.topics.enable=true

```

```
azure.activedirectory.session-stateless=true
azure.activedirectory.client-id=D4N7Q~uBPMd~NIsUp8VYxX4-vFnp0ZFqfWdPo
azure.activedirectory.appIdUri=http://localhost:3000/
spring.security.oauth2.resourceserver.jwt.jwk-set-
uri=https://login.windows.net/common/discovery/keys
logging.level.org.springframework.security=DEBUG

spring.jackson.serialization.fail-on-empty-beans=false
```

Java Code

```
@Entity
@Table(name = "edc_testreports")
@NoArgsConstructor
@AllArgsConstructor
@Data
public class TestReport {

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

    @Column
    String reportName;

    @Column
    String reportType;

    @Column
    String trialId;

    @Column
    String trialName;

    @Column
    private String userId;
}
```

```
@Repository
public interface TestReportRepository extends JpaRepository<TestReport, Integer>{
}

package com.edc.controller;

@CrossOrigin(origins = "http://localhost:3000")
@RestController
public class TestReportController {

    @Autowired
    TestReportRepository repository;

    @Autowired
    RestTemplate restTemplate;

    @RequestMapping(value = "/", produces = "text/html")
    public ResponseEntity<String> dashboard() {
        String bodyStr = homePageData();
        return ResponseEntity.ok().body(bodyStr);
    }
}
```

```

}

@GetMapping("/testReport/{id}")
public ResponseEntity<TestReport> getReportById(@PathVariable int id){

    TestReport data = repository.getById(id);
    System.out.println("data : "+data);

    try {
        return ResponseEntity
            .ok()
            .contentType(MediaType.APPLICATION_JSON)
            .body(data);
    } catch (Exception e) {
        return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
    }
}

@GetMapping("/testReport/all")
public ResponseEntity<List<TestReport>> getAllTestReports() {
    try {
        List<TestReport> reports = new ArrayList<TestReport>();
        repository.findAll().forEach(reports::add);
        System.out.println(reports.size());
        if (reports.isEmpty()) {
            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        }
        return new ResponseEntity<>(reports, HttpStatus.OK);
    } catch (Exception e) {
        System.out.println(e.getMessage());
        return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

@PostMapping("/testReport/add")
public ResponseEntity<TestReport> createReport(@RequestBody TestReport report) {
    try {
        TestReport testreport = repository.save(new
TestReport(report.getReportName(), report.getReportType(), report.getTrialId(),
report.getTrialName(), report.getUserId()));

        return ResponseEntity
            .ok()
            .contentType(MediaType.APPLICATION_JSON)
            .body(testreport);

    } catch (Exception e) {
        return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

@GetMapping(value = "/testReport/mi", produces = "application/json")
public ResponseEntity<Object> getAllMIReportsNoramlcall() {
    try {
        ResponseEntity<Object> responseEntity = restTemplate.getForEntity("http://MI-
MICROSERVICE/mi/anthology/all", Object.class);
        return new ResponseEntity<>(responseEntity, HttpStatus.OK);
    } catch (Exception e) {
        return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}

```

```

    }

    @GetMapping(value = "/testReport/mihystrix", produces = "application/json")
    @HystrixCommand(fallbackMethod = "miReportFallBackMethod")
    public ResponseEntity<Object> getAllMIReportsWithHyStrix() {
        ResponseEntity<Object> responseEntity =
restTemplate.getForEntity("http://MI-MICROSERVICE/mi/anthology/all", Object.class);
        System.out.println(responseEntity);
        return new ResponseEntity<>(responseEntity, HttpStatus.OK);
    }

    public ResponseEntity<Object> miReportFallBackMethod() {
        return new ResponseEntity<>("MI Service is Not Availalbe", HttpStatus.OK);
    }
}
}

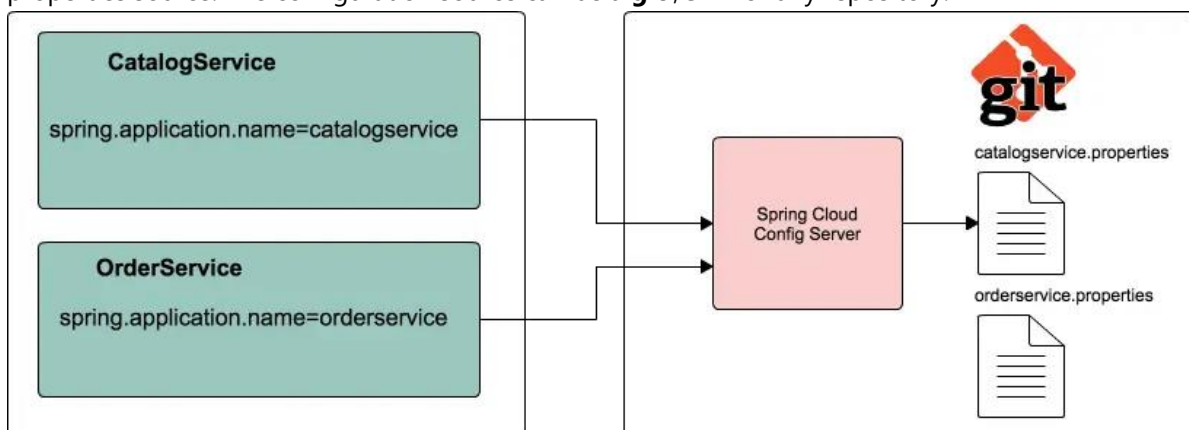
```

2. Spring Cloud Config Server – to Store our Properties

SpringBoot provides lot of flexibility in externalizing configuration properties via properties or YAML files. We can also configure properties for each environment (dev, qa, prod etc) separately using profile specific configuration files such as **application.properties**, **application-dev.properties**, **application-prod.properties** etc. But once the application is started we can not update the properties at runtime. If we change the properties we need to **restart** the application to use the updated configuration properties.

To Solve this, We can use **Spring Cloud Config Server** to centralize all the applications configuration and use **Spring Cloud Config Client** module from the applications to consume configuration properties from Config Server. We can also update the configuration properties at runtime without requiring to restart the application.

Spring Cloud Config Server is nothing but a SpringBoot application with a configured configuration properties source. The configuration source can be a **git**, **svn** or any repository.



Steps to configure Spring Cloud Config Server

1. Create a git repo to store our **properties** with different environments.

<https://github.com/smlcodes/SpringConfigServer>

smlcodes Create catalogservice.properties	
README.md	Initial commit
catalogservice-dev.properties	Create catalogservice-dev.properties
catalogservice-prod.properties	Update catalogservice-prod.properties
catalogservice-test.properties	Create catalogservice-test.properties
catalogservice.properties	Create catalogservice.properties

2. Let us create a SpringBoot application **spring-cloud-config-server** from <http://start.spring.io> by selecting the starters **Config Server** and **Actuator**.

3. To make our SpringBoot application as a SpringCloud Config Server, we just need to add **@EnableConfigServer** annotation to the main entry point class

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

4. configure **git url** property pointing to the git repository in **application.properties**

```
spring.config.name=configserver
server.port=8182
spring.cloud.config.server.git.uri=https://github.com/smlcodes/SpringConfigServer.git
spring.cloud.config.server.git.default-label=main
management.security.enabled=false
```

5. Start the application. Spring Cloud Config Server exposes the **following REST endpoints** to get application specific configuration properties:

```
/{application}/{profile}/{label}
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
```

Here **{application}** refers to value of **spring.config.name** property, **{profile}** is an active profile and **{label}** is an optional git label (defaults to "master").

Now if you access the URL <http://localhost:8182/catalogservice/default> then you will get the following response with catalogservice **default** configuration details:

```
localhost:8182/catalogservice/default
{
  name: "catalogservice",
  - profiles: [
    "default"
  ],
  label: null,
  version: "96e85da0c08237f5b257395791bdeb7bf560bd57",
  state: null,
  - propertySources: [
    - {
      name: "https://github.com/smlcodes/SpringConfigServer.git/file:C:\Users\kavetis\AppData\Local\Temp\config-rep
    - source: {
      server.port: "8181",
      logging.level.catalog: "debug",
      spring.datasource.driver-class-name: "com.mysql.jdbc.Driver",
      spring.datasource.url: "jdbc:mysql://localhost:3306/catalog?useSSL=false",
      spring.datasource.username: "root",
      spring.datasource.password: "password",
      spring.datasource.initialization-mode: "always",
      spring.jpa.hibernate.ddl-auto: "update",
      spring.jpa.show-sql: "true",
      management.endpoints.web.exposure.include: "*",
      msg: "DEFAULT Environment "
    }
  ]
}
```

Similary we can enviroment perptils like

- <http://localhost:8182/catalogservice/default>
- <http://localhost:8182/catalogservice/dev>
- <http://localhost:8182/catalogservice/test>
- <http://localhost:8182/catalogservice/prod>

Spring Cloud Config Client Example

Let us see how we can create a SpringBoot application and use configuration properties from Config Server instead of putting them inside the application.

1.Update **catalog-service** with **Config Client, Web** and **Actuator** starters.

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

2.Create REST Resource

Add one **RestController** to view the Server side property values in the response. And add a method to request/response the perticalur value from Repository values.

```
@RefreshScope
@RestController
class SpringCloudConfigClient{
    @Value("${msg:Config Server is not working. Please check...}")
    private String msg;

    @GetMapping("/msg")
```

```

public String getMsg() {
    return this.msg;
}
}

```

Usually in SpringBoot application, we configure properties in `application.properties`. But while using Spring Cloud Config Server we use `bootstrap.properties` or `bootstrap.yml` file to configure the URL of Config Server.

Configure the following properties in `src/main/resources/bootstrap.properties`:

```

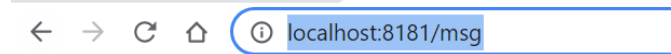
server.port=8181
spring.application.name=catalogservice
spring.cloud.config.uri=http://localhost:8182
management.security.enabled=false

```

Note that the value of `spring.application.name` property should match with base filename (catalogservice) in config-repo

Now run the following catalog-service main entry point class. We can access the actuator endpoint `http://localhost:8181/env` to see all the configuration properties.

You open `http://localhost:8181/msg` you get property value of msg

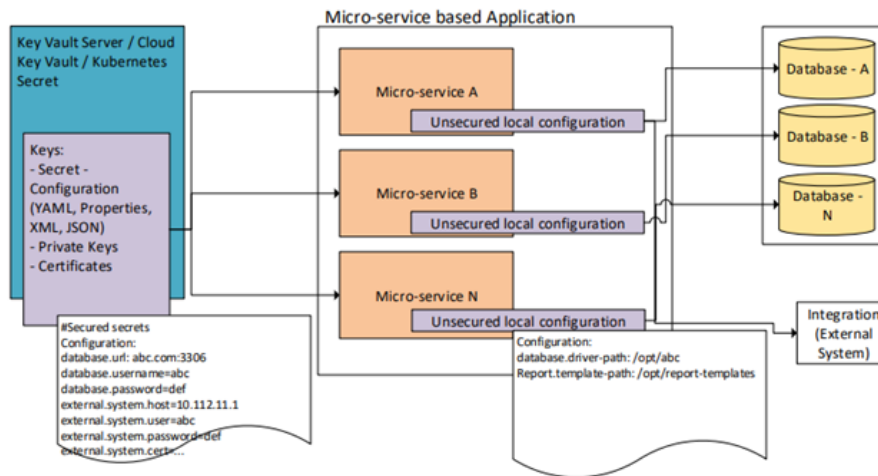


DEFAULT Environment

Now if we delete the `application.properties` file, the application will works normally.

Ref. <https://www.sivalabs.in/2017/08/spring-cloud-tutorials-introduction-to-spring-cloud-config-server/>

2.Azure Key Valut



1. Create Azure Key Vault

Microsoft Azure Search resources, services, and docs (G+)

Home > Key vaults >

Key vaults

PAREXEL Cloud (parexelcloud.com)

+ Create ...

Filter for any field...

Name ↑↓

- DeploymentCalyx03Vault

Create a key vault

Instance details

Key vault name *

Region *

Pricing tier *

Recovery options

Soft delete protection will automatically be enabled on this key vault. This feature allows you to recover or permanently delete a key vault and secrets for the duration of the retention period. This protection applies to the key vault and the secrets stored within the key vault.

To enforce a mandatory retention period and prevent the permanent deletion of key vaults or secrets prior to the retention period elapsing, you can turn on purge protection. When purge protection is enabled, secrets cannot be purged by users or by Microsoft.

Page 1 of 1

[Review + create](#) [< Previous](#) [Next: Access policy >](#)

Configure these properties:

```
azure.keyvault.enabled=true
azure.keyvault.uri=put-your-azure-keyvault-uri-here
azure.keyvault.client-id=put-your-azure-client-id-here
azure.keyvault.client-key=put-your-azure-client-key-here
```

Add dependency. <version> can be skipped because we already add azure-spring-boot-bom.

```
<dependency>
  <groupId>com.azure.spring</groupId>
  <artifactId>azure-spring-boot-starter-keyvault-secrets</artifactId>
</dependency>
```

Now we can read Azure KeyVault secrets directly as any other Spring boot configuration properties, (as Key Vault will be bound as custom Properties Datasource) Example in TestController.java, where `@Value("${my-very-secret}")` is read from Azure KeyVault.

```
@RestController
@RequestMapping("/v1")
public class TestController {

    @Value("${my-very-secret}")
    private String secretValue;

    @Value("${my-not-secret}")
    private String nosecretValue;

    @GetMapping("/values")
    public String[] values() {
        return new String[] { secretValue, nosecretValue };
    }
}
```

Azure KeyValut properties are automatically available, due to Azure Valut Dependency. So we can directly Access

3. Spring Cloud Eureka Service Registry and Discovery

In the microservices world, **Service Registry and Discovery** plays an important role because we most likely run multiple instances of services and we need a mechanism to call other services without hardcoding their hostnames or port numbers.

- Think of it as a lookup service where microservices (clients) can register themselves and discover other registered microservices.
- When a client microservice registers with Eureka it provides metadata such as host, port, and health indicator thus allowing for other microservices to discover it.
- The discovery server expects a regular heartbeat message from each microservice instance. If an instance begins to consistently fail to send a heartbeat, the discovery server will remove the instance from his registry.

Suppose we have 2 microservices **EDC-Microservice** and **MI- Microservice** and we are running 2 instances of **EDC-Microservice** at `http://localhost:8031/` and `http://localhost:8032/`. Now let's say we want to invoke some **EDC-Microservice** REST endpoint from **MI- Microservice**. Which URL should we hit? Generally, in these scenarios, we use a load balancer configuring these 2 URLs to be delegated to and we will invoke the REST endpoint on load balancer URL. Fine.

But, what if you want to spin up new instances dynamically based on load? Even if you are going to run only few server nodes, manually updating the server node details in load balancer configuration is error-prone and tedious. This is why we need automatic Service Registration mechanism and be able to invoke a service using some **logical service id** instead of using specific IP Address and port numbers.

We can use Netflix Eureka Server to create a Service Registry and make our microservices as Eureka Clients so that as soon as we start a microservice it will get registered with Eureka Server automatically with a **logical Service ID**. Then, the other microservices, which are also Eureka Clients, can use Service ID to invoke REST endpoints.

Steps to create Netflix Eureka based Service Registry

add **Eureka Server** starter.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

We need to add **@EnableEurekaServer** annotation on main class to make our SpringBoot application a Eureka Server based Service Registry.

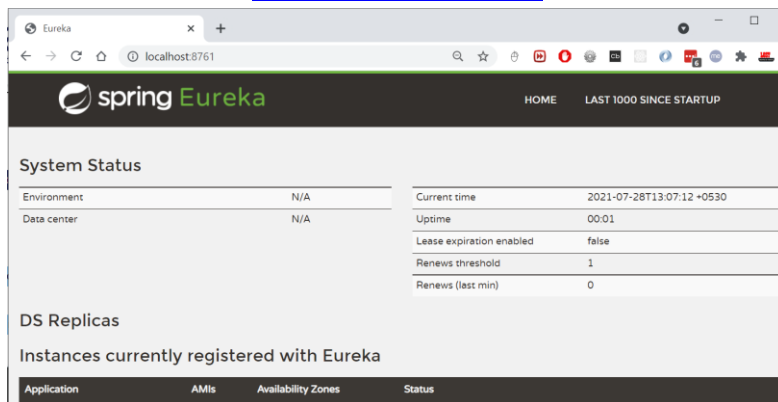
```
@SpringBootApplication
@EnableEurekaServer
public class SpringCloudEurekaServerApplication {

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

By default, each Eureka Server is also a Eureka client and needs at least one service URL to locate a peer. As we are going to have a single Eureka Server node (Standalone Mode), we are going to disable this client-side behavior by configuring the following properties in `application.properties` file.

```
spring.application.name=service-registry
server.port=8761
eureka.instance.hostname=localhost
eureka.instance.client.registerWithEureka=false
eureka.instance.client.fetchRegistry=false
eureka.instance.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/
```

Netflix Eureka Service provides UI where we can see all the details about registered services. Now run the main class, check url <http://localhost:8761/>



Register EDC-Microservice to Eureka

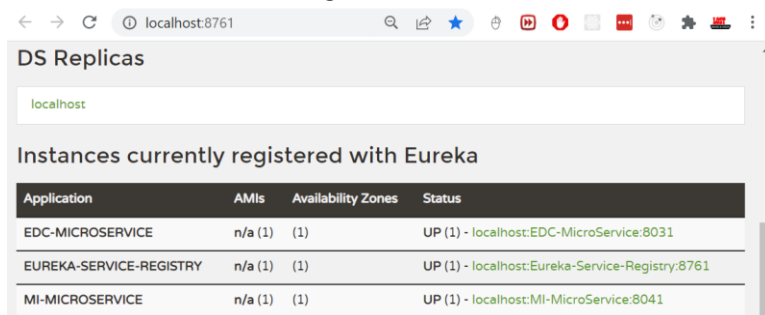
Add the **Eureka Discovery** starter to **EDC-Microservice** which will add the following dependency.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

we just need to configure `eureka.client.service-url.defaultZone` property in `application.properties` to automatically register with the Eureka Server.

```
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
```

With this configuration in place, start **EDC-Microservice** and visit <http://localhost:8761>. You should see EDC-Microservice is registered with **SERVICE ID** as **EDC-MICROSERVICE**



4. Spring Cloud Load balancing

remove `server.port=9992` from both bootstrap & git property file. And start 2 MI-MicroService instance manullay.

```
java -jar -Dserver.port=9898 mi-service-0.0.1-SNAPSHOT.jar
java -jar -Dserver.port=9899 mi-service-0.0.1-SNAPSHOT.jar
```

Now visit Eureka Dashboard <http://localhost:8761/> and see 3 instances of inventory-service registered.

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CATALOGSERVICE	n/a (1)	(1)	UP (1) • host.docker.internal:catalogservice:9991
INVENTORYSERVICE	n/a (3)	(3)	UP (3) • host.docker.internal:inventoryservice:9992, host.docker.internal:inventoryservice:9898, host.docker.internal:inventoryservice:9899
SERVICE-REGISTRY	n/a (1)	(1)	UP (1) • host.docker.internal:service-registry:8761

General Info

Suppocue we if want to call MI-MicroService from EDC-MicroService, which of above 3 instance will call ?

We do by enablling LoadBalancer mechanisum in Our MicroServices. Eureka by default provides **Ribbon LoadBalancer**. To enable Ribbon LoadBalancer in Our Microservices do below steps

In SpringBoot main Application class we need to following things

- Create & Return **RestTemplate Object**
- We can register **RestTemplate** as a Spring bean with **@LoadBalanced** annotation.
- The RestTemplate with **@LoadBalanced** annotation will internally use **Ribbon LoadBalancer** to resolve the **ServiceID** and invoke REST endpoint using one of the available servers.

```
Catalog-service/ CatalogServiceApplication.java
@SpringBootApplication
public class CatalogServiceApplication {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

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

In Service Class Create **RestTemplate** object & Autowire It.

Use RestTemplate to invoke **inventory-service** endpoint from [catalog-service/ProductService.java](#)

```
@Service
@Transactional
@Slf4j
public class ProductService {

    @Autowired
    private final ProductRepository productRepository;

    @Autowired
```

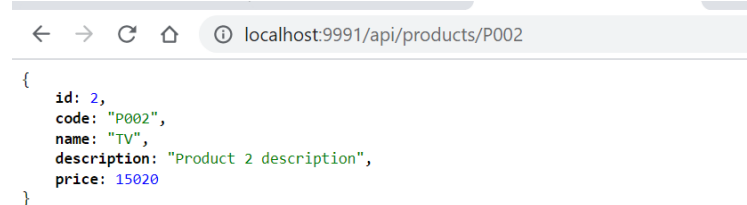
```

private final RestTemplate restTemplate;

public Optional<Product> findProductByCode(String code) {
    Optional<Product> productOptional = productRepository.findByCode(code);
    if(productOptional.isPresent()) {
        Log.info("Fetching inventory level for product_code: "+code);
        ResponseEntity<ProductInventoryResponse> itemResponseEntity =
            restTemplate.getForEntity("http://inventory-service/api/inventory/{code}",
                ProductInventoryResponse.class,
                code);
        if(itemResponseEntity.getStatusCode() == HttpStatus.OK) {
            Integer quantity = itemResponseEntity.getBody().getAvailableQuantity();
            Log.info("Available quantity: "+quantity);
        } else {
            Log.error("Unable to get inventory level for product_code: "+code +
                ", StatusCode: "+itemResponseEntity.getStatusCode());
        }
    }
    return productOptional;
}
}

```

Note that we have used <http://inventory-service/api/inventory/{code}> instead of <http://localhost:9898/api/inventory/{code}> or <http://localhost:9999/api/inventory/{code}>

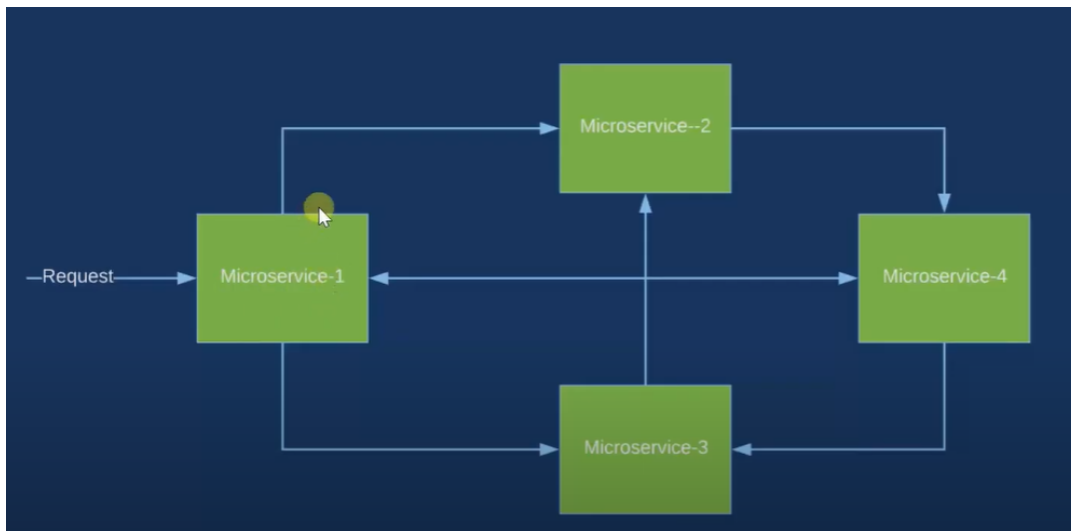


With this kind of automatic Service Registration and Discovery mechanism, we no need to worry about how many instances are running and what are their hostnames and ports etc.

5. Spring Cloud Circuit Breaker using Netflix Hystrix

In the microservices world, to fulfill a client request one microservice may need to talk to other microservices. We should minimize this kind of direct dependencies on other microservices but in some cases it is unavoidable. **If a microservice is down or not functioning properly then the issue may cascade up to the upstream services.**

Netflix created Hystrix library implementing Circuit Breaker pattern to address these kinds of issues. We can use **Spring Cloud Netflix Hystrix Circuit Breaker** to protect microservices from cascading failures.



Steps to Spring Cloud Circuit Breaker using Netflix Hystrix

From **EDC-Microservice** we are invoking REST endpoint on **MI-service** to testReports of the user. What if MI-service is down? What if MI-service is taking too long to respond thereby slowing down all the services depending on it? We would like to have some **timeouts** and implement some **fallback mechanism**.

Following Steps we need to Perform

- Add **Hystrix** Dependency
- Enable Hystrix functionality in our EDC-Microservice, by adding **@EnableHystrix** on our main SpringBoot Class
- Use **@HystrixCommand(fallbackMethod = "miReportFallBackMethod")** to define fallback method if called microservice is down or unreachable

Add **Hystrix** starter to catalog-service.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
  <version>2.2.10.RELEASE</version>
</dependency>
```

To enable Circuit Breaker add **@EnableHystrix** annotation on **catalog-service** entry-point class.

```
@SpringBootApplication
@EnableHystrix
public class EDCMicroServiceMainApplication {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    public static void main(String[] args) {
        SpringApplication.run(EDCMicroServiceMainApplication.class, args);
        System.out.println(" EDC Service... ");
    }
}
```



```

@RestController
public class TestReportController {

    @Autowired
    TestReportRepository repository;

    @Autowired
    RestTemplate restTemplate;

    @GetMapping(value = "/testReport/mi", produces = "application/json")
    @HystrixCommand(fallbackMethod = "miReportFallBackMethod")
    public ResponseEntity<Object> getAllMIReportsWithHyStrix() {
        try {
            ResponseEntity<Object> responseEntity =
restTemplate.getForEntity("http://MI-MICROSERVICE/mi/anthology/all", Object.class);
            System.out.println(responseEntity);

            return new ResponseEntity<>(responseEntity, HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }

    public ResponseEntity<Object> miReportFallBackMethod() {
        try {
            return new ResponseEntity<>("MI Service is Not Avaialable", HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}

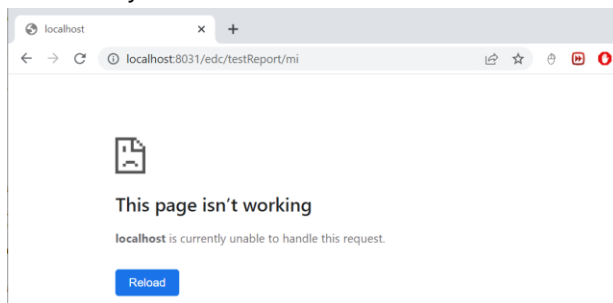
```

We have annotated the method from where we are making a REST call with **@HystrixCommand(fallbackMethod = "miReportFallBackMethod")** so that if it doesn't receive the response within the certain time limit the call gets timed out and invoke the configured fallback method.

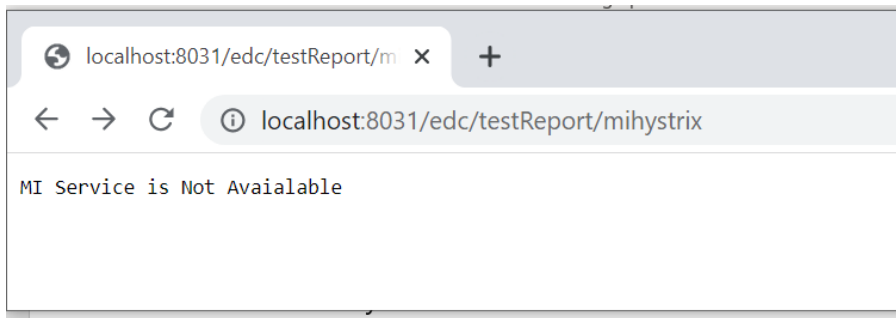
The fallback method should be defined in the same class and should have the same signature(return type). In the fallback method **miReportFallBackMethod()** we are setting default behavior depends on what business wants.

I have Stopped MI-MicroService, for checking HyStrix Working

Without Hystrix



With Hystrix



HyStrix Dashboard

We have two ways to configure Hystrix Dashboard

1.Enable Hystrix Dashboard EDC-MicroService it self.

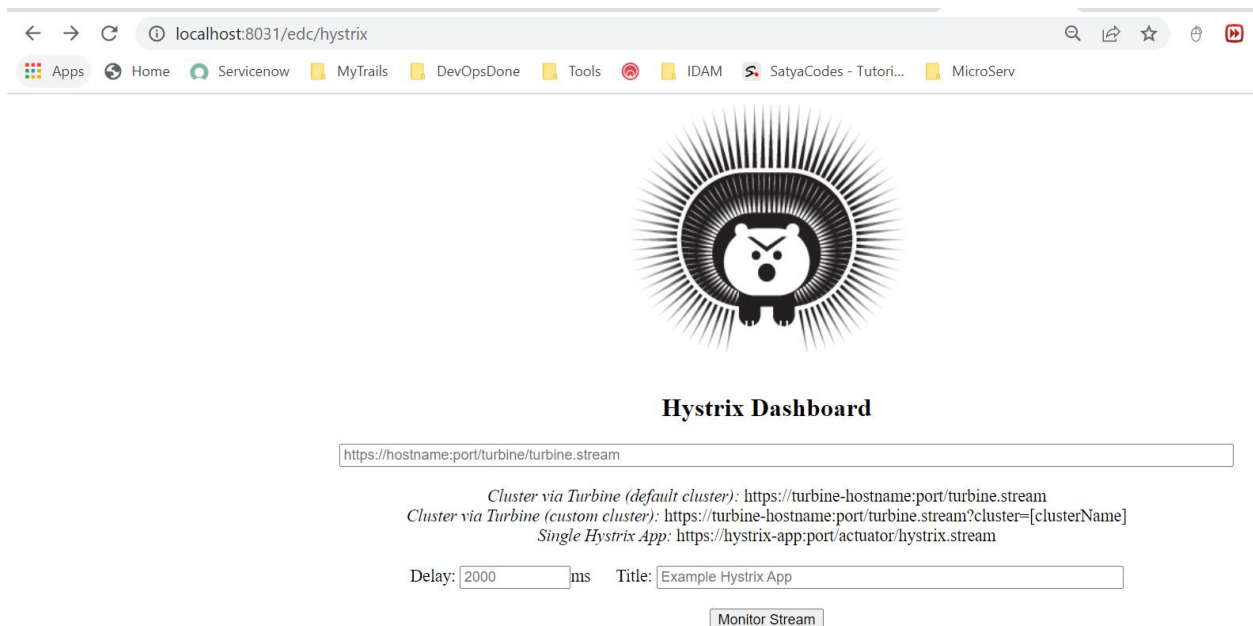
Add Dashboard Dependency

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
  <version>2.2.10.RELEASE</version>
</dependency>
```

2.Add @EnableHystrixDashboard annotation to SpringBoot main Class

```
@SpringBootApplication
@EnableHystrix
@EnableHystrixDashboard
public class EDCMicroServiceMainApplication {
}
```

Once we add **Hystrix** starter to EDC-Microservice we can get the circuits status as a stream of events using Actuator endpoint <http://localhost:8181/actuator/hystrix.stream>, assuming catalog-service is running on 8181 port.



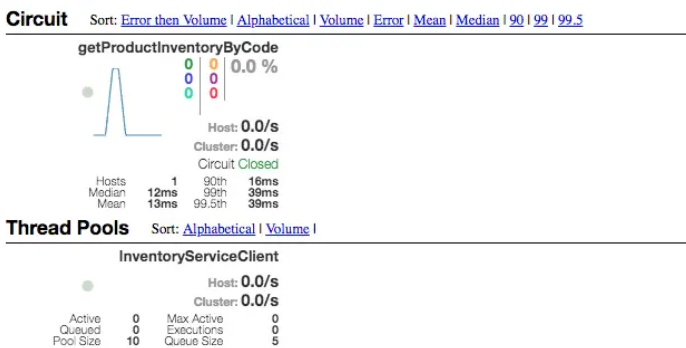
2. Separate Server for Connecting Hystrix

Spring Cloud also provides a nice dashboard to monitor the status of Hystrix commands. Create a Spring Boot application with **Hystrix Dashboard** starter and annotate the main entry-point class with **@EnableHystrixDashboard**.

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

Now in Hystrix Dashboard home page enter **http://localhost:8181/actuator/hystrix.stream** as stream URL and give Catalog Service as Title and click on Monitor Stream button.

Hystrix Stream: Catalog Service



6. Spring Cloud Feign(Pain) - Declarative REST Client

In Previous example we used **RestTemplate** class for calling external MicroService.

```
@RestController
public class TestReportController {

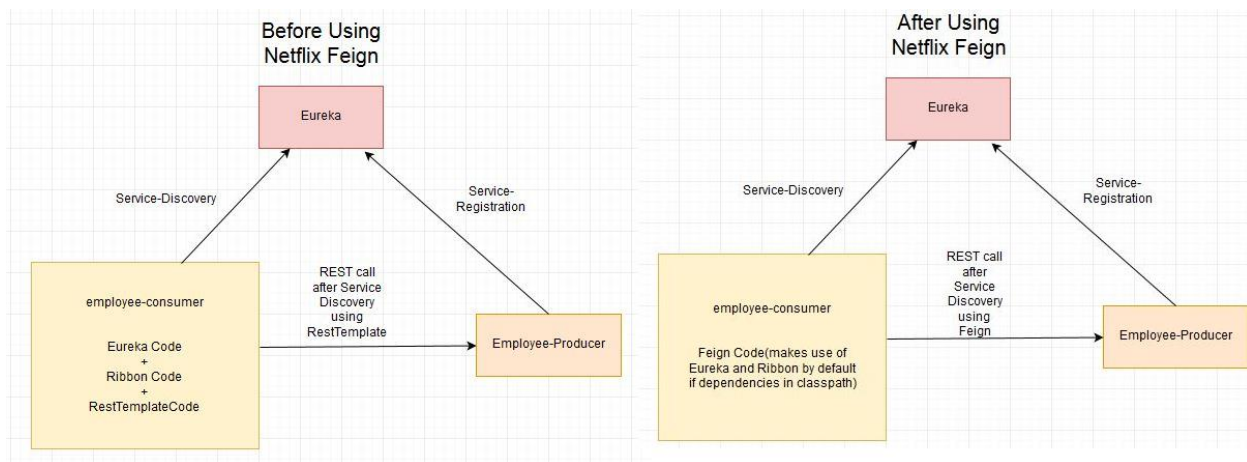
    @Autowired
    TestReportRepository repository;

    @Autowired
    RestTemplate restTemplate;

    @GetMapping(value = "/testReport/mi", produces = "application/json")
    @HystrixCommand(fallbackMethod = "miReportFallBackMethod")
    public ResponseEntity<Object> getAllMIReportsWithHyStrix() {

        ResponseEntity<Object> responseEntity =
            restTemplate.getForEntity("http://MI-MICROSERVICE/mi/anthology/all", Object.class);
        return new ResponseEntity<>(responseEntity, HttpStatus.OK);
    }

    public ResponseEntity<Object> miReportFallBackMethod() {
        try {
            return new ResponseEntity<>("MI Service is Not Availalable", HttpStatus.OK);
        } catch (Exception e) {
            return new ResponseEntity<>(null, HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}
```



```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

In above Example we are calling **MIMicroService** from **EDCMicroService** using **RestTemplate**. But this call is tightly coupled with controller method itself. If any other Controller want to call same microservice url, it needs to be rewrite whole RestTemplate calling.

To resolve this, we are using Feign client to takeout that external calls and defined in a Feign Client.

- annotate the Spring Boot Main class with **@EnableFeignClients**.
- Create a Separate interface for MIService calls in EDCService **MIServiceFeignClient.java**
- Add **@FeignClient** and provide logical ServiceID of MIService in eureka, **@FeignClient(name="MI-MICROSERVICE")**
- Now you can write all abstract methods which are same as MIMicroService Controller with **@Get** /**@Post** mappings.

```
@FeignClient(name="MI-MICROSERVICE")
public interface MIServiceFeignClient {

    @GetMapping("/{id}")
    public AnthologyDTO getAnthologyById(@PathVariable(value = "id") Integer anthologyId);

    @GetMapping(value = "/all", produces = "application/json")
    public List<AnthologyDTO> getAllAnthologys();
}
```

- Create DTO class to store MIRelated Data response.

```
@Data
public class AnthologyDTO {
    private int anthologyId;
    private int userId;
    private String reportName;
    private String reportUrl;
    private String trialId;
    private String trialName;
}
```

- Now in TestController class Autowire MIServiceFeignClient & replace RestTemplate call with MIServiceFeignClient method call.

```

@RestController
public class TestReportController {

    @Autowired
    MIServiceFeignClient miFeignClient;

    @GetMapping(value = "/testReport/mifeign", produces = "application/json")
    @HystrixCommand(fallbackMethod = "miReportFallbackMethod")
    public ResponseEntity<List<AnthologyDTO>> getAllMIReportsWithHyStrixFeign() {
        List<AnthologyDTO> response = miFeignClient.getAllAnthologys();
        return new ResponseEntity<>(response, HttpStatus.OK);
    }
}

```

6. Spring Cloud Zuul Proxy as API Gateway

Spring Cloud provides **Zuul(Zool)** proxy, similar to **Nginx**, that can be used to create API Gateway.

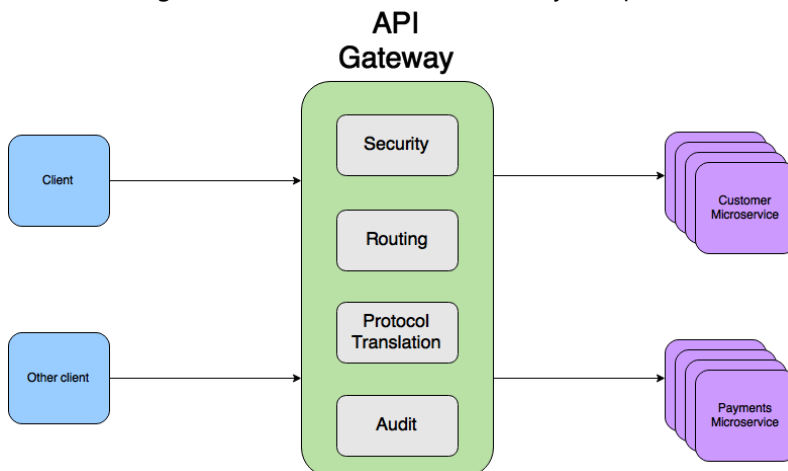
API Gateway, aka Edge Service, provides a unified interface for a set of microservices. so that clients no need to know about all the details of microservices internals like hostname & port numbers. However, there are some pros and cons of using API Gateway pattern in microservices architecture.

Pros:

- Provides easier interface to clients
- Can be used to prevent exposing the internal microservices structure to clients
- Allows to refactor microservices without forcing the clients to refactor consuming logic
- Can centralize cross-cutting concerns like security, monitoring, rate limiting etc

Cons:

- It could become a single point of failure if proper measures are not taken to make it highly available
- Knowledge of various microservice API may creep into API Gateway



<https://www.youtube.com/watch?v=-l-9gK8NWXy>

```

server:
  port: 8100

zuul:

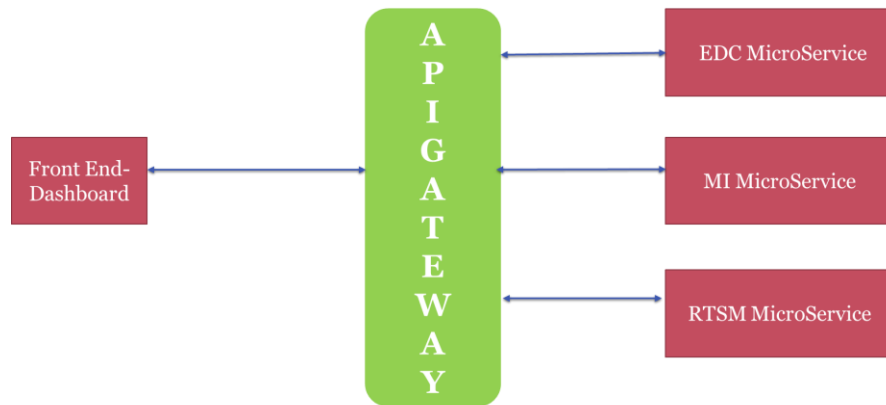
```

```

routes:
  doctors-service:
    url: http://localhost:8082
  patient-service:
    url: http://localhost:8083
  disease-service:
    url: http://localhost:8081
host:
  connect-timeout-millis: 2000
  socket-timeout-millis: 5000

```

<https://github.com/greenlearner01/ApiGateway>



Steps to Implement Zuul in our Application

1. Add Zuul & other required dependencies.

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
  <version>2.2.10.RELEASE</version>
</dependency>

```

2. Enable Zuul API gateway by adding `@EnableZuulProxy` annotation to our main class

```

@SpringBootApplication
@EnableZuulProxy
public class IDaamApiGatewayApplication {

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

}

```

3. Configure `application.yaml` file with Route Details (**without Eureka Server** Load balancer)

```

server:
  port: 8099

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

```

```
# without Eureka Server
zuul:
  routes:
    edc-service:
      url: http://localhost:8031/edc/
    mi-service:
      url: http://localhost:8041/mi/

  host:
    connect-timeout-millis: 2000
    socket-timeout-millis: 5000
```

4. To access MicroService API via API Gateway use following URL's

```
IDAAM API GATEWAY STARTED
http://localhost:8099/
  EDC Service: http://localhost:8099/edc-service
  MI Service : http://localhost:8099/mi-service
```

With Eureka Server

Application	AMIs	Availability Zones	Status
EDC-MICROSERVICE	n/a (3)	(3)	UP (3) - localhost:EDC-MicroService:8031 , localhost:EDC-MicroService:8032 , localhost:EDC-MicroService:8033
HYSTRIX-DASHBOARD	n/a (1)	(1)	UP (1) - localhost:hystrix-dashboard:8788
MI-MICROSERVICE	n/a (3)	(3)	UP (3) - localhost:MI-MicroService:8043 , localhost:MI-MicroService:8041 , localhost:MI-MicroService:8042
MY-MICROSERVICE	n/a (1)	(1)	UP (1) - localhost:my-microservice:8099
SERVICE-REGISTRY	n/a (1)	(1)	UP (1) - L506208.pii.pxl.int:service-registry:8761

```
# WITH Eureka Server
zuul:
  routes:
    edcservice:
      path: /edc/**
      serviceId: EDC-MICROSERVICE

    miservice:
      path: /mi/**
      serviceId: MI-MICROSERVICE
  host:
    connect-timeout-millis: 2000
    socket-timeout-millis: 5000
```

As Zuul act as a proxy to all our microservices, we can use Zuul service to implement some cross-cutting concerns like security, rate limiting etc. One common use-case is forwarding the **Authentication headers** to all the downstream services.

Typically in microservices, we will use OAuth service for authentication and authorization. Once the client is authenticated OAuth service will generate a token which should be included in the requests making to other microservices so that client need not be authenticated for every service separately. We can use Zuul filter to implement features like this.

```
package com.idaam;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import javax.servlet.http.HttpServletRequest;
import java.util.UUID;
import static org.springframework.cloud.netflix.zuul.filters.support.FilterConstants.PRE_TYPE;

public class AuthHeaderFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return PRE_TYPE;
    }

    @Override
    public int filterOrder() {
        return 0;
    }

    @Override
    public boolean shouldFilter() {
        return true;
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();

        if (request.getAttribute("AUTH_HEADER") == null) {
            String sessionId = UUID.randomUUID().toString();
            ctx.addZuulRequestHeader("AUTH_HEADER", sessionId);
        }
        return null;
    }
}
```

We are adding **AUTH_HEADER** as a request header using **RequestContext.addZuulRequestHeader()** which will be forwarded to downstream services. We need to register it as a Spring bean in our MicroServices.

```
@Bean
AuthHeaderFilter authHeaderFilter() {
    return new AuthHeaderFilter();
}
```

7. Distributed Tracing with Spring Cloud Sleuth(Sluuth) and Zipkin

In the microservices world, a user action on UI may invoke one microservice API endpoint, which in turn invoke another microservice endpoint.

For example, when a user sees the catalog, **shoppingcart-ui** will invoke **catalog-service** REST API `http://localhost:8181/api/products` which in turn calls **inventory-service** REST API `http://localhost:8282/api/inventory` to check for inventory availability.

Suppose, an exception has occurred or the data returned is invalid and you want to investigate what is wrong by looking at logs. But as of now, there is no way to correlate the logs of that particular user across multiple services.

Olden Days

One solution to this is at the beginning of the call chain we can create a **CORRELATION_ID** and add it to all log statements. Along with it, send **CORRELATION_ID** as a header to all the downstream services as well so that those downstream services also use **CORRELATION_ID** in logs. This way we can identify all the log statements related to a particular action across services.

We can implement this solution using **MDC** feature of Logging frameworks. Typically we will have a **WebRequest** Interceptor where you can check whether there is a **CORRELATION_ID** header. If there is no **CORRELATION_ID** in the header then create a new one and set it in **MDC**. The logging frameworks include the information set in **MDC** with all log statements.

But, instead of we doing all this work we can use **Spring Cloud Sleuth** which will do all this and much more for us.

Using Sleuth

Add Sleuth Dependency in All Microservices (EDC, MI)

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-sleuth</artifactId>
  <version>2.2.8.RELEASE</version>
</dependency>
```

Once you add **Sleuth** starter and start the services you can observe in logs something like this.

```
2018-03-20 10:19:15.512 INFO [inventory-service,,,] 53685 --- [trap-executor-0]
c.n.d.s.r.aws.ConfigClusterResolver ...
2018-03-20 10:24:15.507 INFO [inventory-service,,,] 53685 --- [trap-executor-0]
c.n.d.s.r.aws.ConfigClusterResolver ...
```

Now hit any **inventory-service** REST endpoint, say `http://localhost:8282/api/inventory`. Then you can observe **TraceID**, **SpanID** in the logs.

```
2018-03-20 10:15:38.466 INFO [inventory-service,683f8e4370413032,d8abe400c68a9a6b,false]
53685 --- [oryController-3] ...
```

Sleuth includes the pattern [**appName**,**traceId**,**spanId**,**exportable**] in logs from the **MDC**.

Now invoke the **catalog-service** endpoint `http://localhost:8181/api/products` endpoint which internally invokes **inventory-service** endpoint `http://localhost:8282/api/inventory`.

In **catalog-service** logs you can find log statements something like:

```
2018-03-20 10:54:29.625 INFO [catalog-service,0335da07260d3d6f,0335da07260d3d6f,false] 53617
--- [io-8181-exec-10] ...
```

And, check logs in inventory-service, you can find log statements something like:

```
2018-03-20 10:54:29.662 INFO [inventory-service,0335da07260d3d6f,1af68249ac3a6902,false]
53685 --- [oryController-6] ...
```

Observe that TracelD **0335da07260d3d6f** is same in both catalog-service and inventory-service for the same REST API call. This way we can easily correlate the logs across services.

The **false** in [inventory-service,0335da07260d3d6f,1af68249ac3a6902,false] indicates that this trace is not exported to any Tracing Server like Zipkin. Let us see how we can export the tracing information to Zipkin.

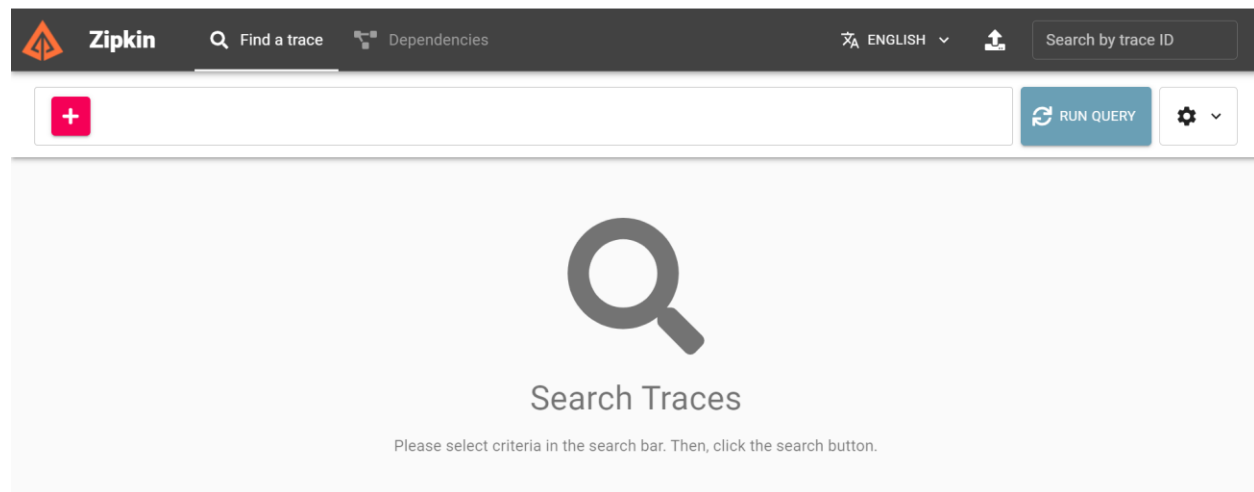
Zipkin Distributed Tracing Configuration

Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.

If you have a **tracelID** in a log file, you can jump directly to it. Otherwise, you can query based on attributes such as service, operation name, tags and duration. Some interesting data will be summarized for you, such as the percentage of time spent in a service, and whether or not operations failed.

The quick and easiest way to start a Zipkin server is using zipkin executable jar provided by Zipkin team.

- Download Jar file : <https://zipkin.io/pages/quickstart.html>
- Run Server : `java -jar zipkin.jar`
- Access using : <http://127.0.0.1:9411/>



We observed that the tracing information is printed in logs but not exported. We can export them to Zipkin server so that we can visualize traces in Zipkin Server UI Dashboard.

Add **Zipkin Client** starter to both inventory-service and catalog-service.

```
<dependency>
```

```

<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-zipkin</artifactId>
<version>2.2.8.RELEASE</version>
</dependency>

```

Configure Zipkin server URL in **application.properties** of both inventory-service and catalog-service.

```

spring.zipkin.base-url=http://localhost:9411/
spring.sleuth.sampler.probability=1

```

Now restart both inventory-service and catalog-service and invoke http://localhost:8181/api/products endpoint. You can observe that **true** is printed in logs meaning it is exported.

```

2018-03-20 11:41:02.241 INFO [catalog-service,7d0d44fe314d7758,7d0d44fe314d7758,true] 53617 -
-- [nio-8181-exec-5] c.s.c.services.ProductService

```

Now go to Zipkin UI Dashboard, you can see the service names populated in the first dropdown. Select the service you want to check or select all and then click on Find Traces button.

8.Spring Cloud Streams – Kafka & ZooKeeper

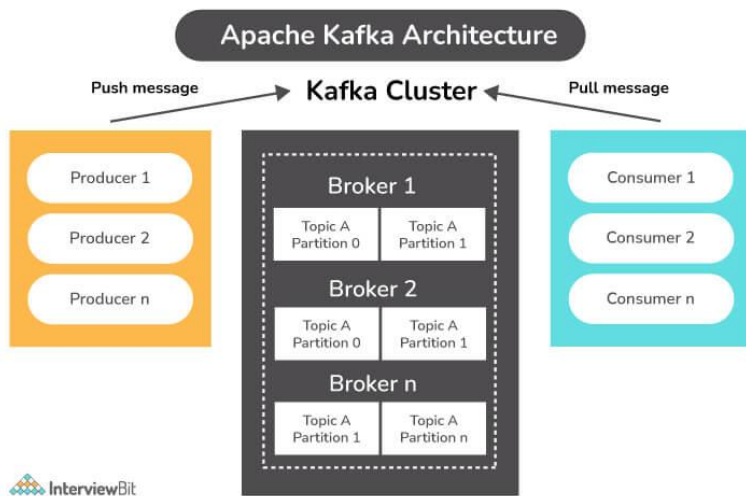
Kafka as “an open-source message broker project developed by the [Apache Software Foundation](#) written in Scala and is a distributed publish-subscribe messaging system.

Kafka is designed for transferring Big data. Kafka has better throughput, **built-in partitioning, replication** and inherent **fault-tolerance**, which makes it a good fit for large-scale message processing applications.

Two types of messaging patterns are available – **one is point** to point and the other is **publish-subscribe** (pub-sub) messaging system. Most of the messaging patterns follow **pub-sub**.

High Throughput	Support for millions of messages with modest hardware
Scalability	Highly scalable distributed systems with no downtime
Replication	Messages are replicated across the cluster to provide support for multiple subscribers and balances the consumers in case of failures
Durability	Provides support for persistence of message to disk
Stream Processing	Used with real-time streaming applications like Apache Spark & Storm
Data Loss	Kafka with proper configurations can ensure zero data loss

- **Producer:** responsible for producing messages for a specific topic.
- **Consumer:** responsible for reading the messages that the producer puts on a topic.
- **Brokers** – a set of servers where the publishes messages are stored
- **Topic:** topic is a **categorized group** of messages.
- **Message:** I think that all events in Kafka can be summarized in messages. A message can contain a simple text like “Hello World” or an **object** in **json** format for example.



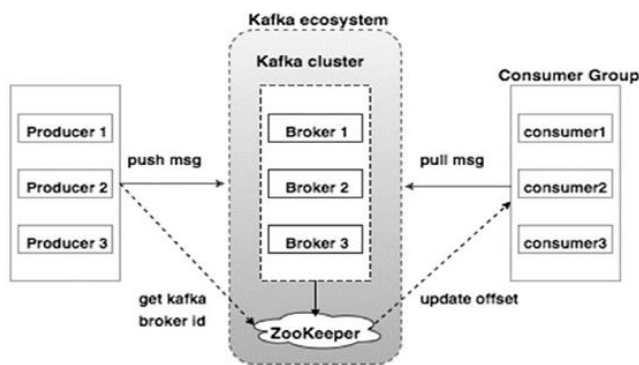
Partition & Offset

In the above diagram, a topic is configured into three partitions.

- Partition 1 has two offset factors 0 and 1.
- Partition 2 has four offset factors 0, 1, 2, 3.
- Partition 3 has one offset factor 0.
- The id of the replica is same as the id of the server that hosts it.

Assume, if the replication factor of the topic is set to 3, then Kafka will create 3 identical replicas of each partition and place them in the cluster to make available for all its operations. To balance a load in cluster, each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time.

Cluster Architecture



Broker: Kafka cluster typically consists of multiple brokers to maintain load balance. Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state. One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each broker can handle TB of messages without performance impact. Kafka broker leader election can be done by ZooKeeper.

ZooKeeper: ZooKeeper is used for managing and coordinating Kafka broker. ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or

failure of the broker in the Kafka system. As per the notification received by the Zookeeper regarding presence or failure of the broker then producer and consumer takes decision and starts coordinating their task with some other broker.

Producers: Producers push data to brokers. When the new broker is started, all the producers search it and automatically sends a message to that new broker. Kafka producer doesn't wait for acknowledgements from the broker and sends messages as fast as the broker can handle

Windows Install & Configuration

- Got to the [Apache Kafka downloads page](https://kafka.apache.org/download) and download the <https://kafka.apache.org/download> the **Scala 2.12 kafka 2.12-0.10.2.1.tgz**. Next unzip & start Apache Kafka
- This Kafka installation comes with an inbuilt zookeeper. **Zookeeper is mainly used to track status of nodes present in Kafka cluster** and also to keep track of Kafka topics, messages, etc.
- Open a command prompt and **start the Zookeeper-**

```
cd C:\kafka
.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
```

- Open a new command prompt and **start the Apache Kafka**

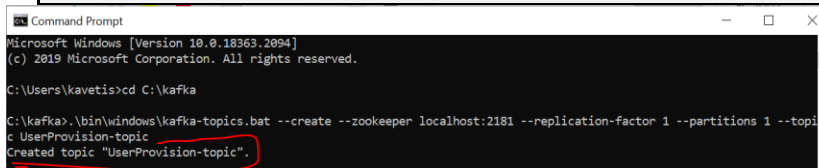
```
cd C:\kafka
.\bin\windows\kafka-server-start.bat .\config\server.properties

//If any errors add below line in Environment propertis in User Serction
%SystemRoot%\System32\Wbem;%SystemRoot%\System32\SystemRoot%
```

Create UserProvision Topic

- Open a new command prompt and **create a topic with name UserProvision-topic, that has only one partition & one replica.**

```
cd C:\kafka
.\bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic UserProvision-topic
```



```
Microsoft Windows [Version 10.0.18363.2094]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\kavetis>cd C:\kafka
C:\kafka>.\bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic UserProvision-topic
Created topic "UserProvision-topic".
```

- Next Open a new command prompt and **create a PRODUCER to send message to the above created UserProvision-topic and send a message - Hello World Javainuse to it-**

```
cd C:\kafka
.\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic UserProvision-topic Hello Message:User is Created
```

- Finally Open a new command prompt and **start the CONSUMER which listens to the topic javainuse-topic we just created above.** We will get the message we had sent using the producer

```
cd C:\kafka
.\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic
UserProvision-topic --from-beginning
```

SpringBoot Kafka Application

When ever new User is Created, we need to send the User details to Kafka. Downstream applications will consume those User details. So we have need Create below Projects.

Producer: Send messages to Topic

- UserManagementMicroServiceWithKafka

Consumer: Consumes messages

- EDC-MicroService
- MI-MicroService

1.Producer Application : UserManagementMicroServiceWithKafka

Create SpringBoot Project & Add Dependencies

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

application.properties:

```
# Producer properties
spring.kafka.producer.bootstrap-servers=127.0.0.1:9092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.group-id=group_id
topic.name.producer=user.provision.topic

# Common Kafka Properties
auto.create.topics.enable=true
```

- At first I put the `StringSerializer` class to serialize the messages, just to send some text.
- The `auto.create.topics.enable = true` property automatically creates topics registered to those properties. In that case I put `topic.name.producer = user.provision.topic`

SpringBootApplication : In the main class I put the annotation "`@EnableKafka`", which makes it possible to connect to a topic.

```
@SpringBootApplication
@EnableKafka
```

```

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

```

```

@RestController
@RequestMapping(value = "/userprovision/")
public class UserProvisionController {

    @Autowired
    KafkaSenderService kafkaSender;

    @PostMapping(value = "/add", produces = "text/html")
    public ResponseEntity<String> producer(@RequestBody UserDetails userDetails) {
        kafkaSender.send(userDetails.toString());
        return ResponseEntity.ok().body(userDetails.toString());
    }
}

```

```

@Service
public class KafkaSenderService {
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    //String kafkaTopic = "UserProvision-topic";
    @Value("${topic.name.producer}")
    private String kafkaTopic;

    public void send(String data) {
        System.out.println("KafkaSenderService : Message STRAT");
        kafkaTemplate.send(kafkaTopic, data);
        System.out.println("KafkaSenderService : Message SENT");
    }
}

```

The `KafkaTemplate` class is the class that sends messages to topics, the first String is the topic and the second the type of information.

Test <http://localhost:8051/userprovision/add>

```

{
  "userId": 101,
  "firstName": "Satya",
  "lastName": "Kaveti",
  "role": "Admin",
  "trialName": "Trial123",
  "city": "UNITEDKINGDOM"
}

```

Consumer Application : EDC-MicroService, MI-MicroService

Add Dependency to both pom.xmls

```

<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>

```

```
</dependency>
```

Update Kafka application.properties:

```
# Consumer properties
```

```
spring.kafka.producer.bootstrap-servers=127.0.0.1:9092
spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer
spring.kafka.producer.group-id=group_id
topic.name.consumer=user.provision.topic

# Common Kafka Properties
auto.create.topics.enable=true
```

main class annotated with "@EnableKafka", which makes it possible to connect to a topic.

```
@SpringBootApplication
@EnableHystrix
@EnableHystrixDashboard
@EnableKafka
public class EDCMicroServiceMainApplication {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

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

Create Service class which consumes Topic messages

```
@Slf4j
@RequiredArgsConstructor
@Service
public class KafkaConsumerService {

    @Value("${topic.name.consumer}")
    private String topicName;

    @KafkaListener(topics = "${topic.name.consumer}", groupId = "group_id")
    public void consume(ConsumerRecord<String, String> payload) {
        log.info("Tópico: {}", topicName);
        log.info("key: {}", payload.key());
        log.info("Headers: {}", payload.headers());
        log.info("Partion: {}", payload.partition());
        log.info("Order: {}", payload.value());
    }
}
```

Start Both UserKafkaService & EDC, MI Services

Open SwaggerUI of user provision sent data

user-provision-controller ^

POST /userprovision/add ^

Parameters Cancel Reset

No parameters

Request body required application/json v

```
{
  "userId": 101,
  "firstName": "Satya",
  "lastName": "Kaveti",
  "role": "Admin",
  "trialName": "Trial123",
  "city": "UNITEDKINGDOM"
}
```

Producer Log

```
2022-03-16 12:42:36.322 INFO 15408 --- [ad | producer-1]
org.apache.kafka.clients.Metadata : [Producer clientId=producer-1] Cluster ID:
kaWxu5VsTbul0sUgCos2zQ
KafkaSenderService : Message SENT
```

Consumer Log

```
Tópico: {}${topic.name.consumer}
key: {}null
Headers: {}RecordHeaders(headers = [], isReadOnly = false)
Partition: {}0
Order: {}UserDetails [userId=101, firstName=Satya, lastName=Kaveti, role=Admin,
trialName=Trial123, city=UNITEDKINGDOM]
```

Remember : We got errors while Sluth& Zipkin jars enabled in pom.xml. So while running Kafka please comment those

<https://medium.com/geekculture/implementing-a-kafka-consumer-and-kafka-producer-with-spring-boot-60aca7ef7551>

https://www.youtube.com/watch?v=L_iu8HOus8k

Summary of MicroServices

Order of Start Services

service-registry	Eureka Service Discovery Server	http://localhost:8761/
Hystrix-Dashboard(Optional)	Circuit Breaker/ Fault tolerance Dashboard only	http://localhost:8788/
IDaamApiGateway	Zuul API Gateway	http://localhost:8099
Sleuth and Zipkin	Distributed Tracing. <ul style="list-style-type: none"> Just adding dependency logs will generated with TracerID Run Server cd C:\kafka <code>java -jar zipkin.jar</code> 	http://127.0.0.1:9411/
UserServiceWithKafka	UserProisionKafkaser	http://localhost:8051/
EDCMicroService		http://localhost:8031/edc/
MIMicroService		http://localhost:8041/mi/

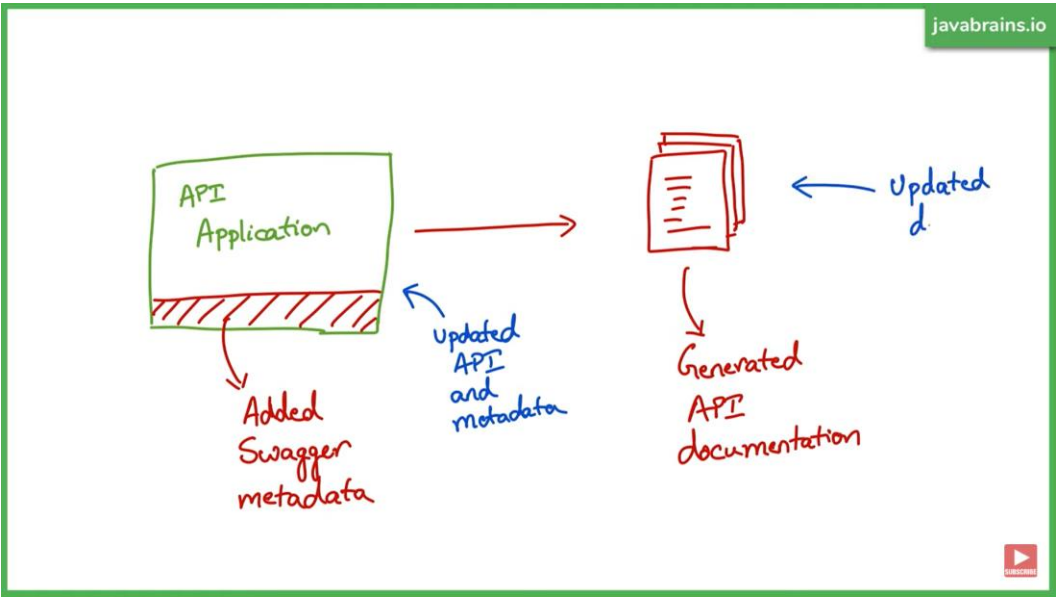
Eureka Service Discovery Server	<ul style="list-style-type: none"> Separate Server Add spring-cloud-starter-netflix-eureka-server dependency Add @EnableEurekaServer to main class Update <code>application.properties</code> with port & other details <p>EDC, MI Microservices End</p> <ul style="list-style-type: none"> Add the Eureka client dependency <code>spring-cloud-eureka-client</code>. Add Eureka Server URL in <code>application.properties</code> <code>eureka.client.service-url.defaultZone=http://localhost:8761/eureka/</code> By adding this, at start up they will register to eureka servre <p>If EDC has 4 Nodes, MI has 4 nodes Eureka Server will take care about loadbalance.</p> <ul style="list-style-type: none"> Add @LoadBalanced to SpringBoot Main Class RestTemplate I used to call other MicroService call. The RestTemplate with @LoadBalanced annotation will internally use Ribbon LoadBalancer to resolve the ServiceID and invoke REST endpoint using one of the available servers
---------------------------------	---

<p>Hystrix- Circuit Breaker/ Fault tolerance</p>	<p>In above we are calling other microservice using RestTemplate. If microservice down, it will has fallbackmethod to handle those situations.</p> <ul style="list-style-type: none"> • Add Hystrix Dependency : spring-cloud-starter-netflix-hystrix • Enable Hystrix functaionlity in our EDC-Microservice, by adding @EnableHystrix on our main SpringBoot main Class • Use @HystrixCommand(fallbackMethod = "miReportFallBackMethod") to define fallback method if called microservice is down or unreachable <pre>ResponseEntity r= restTemplate.getForEntity("http://MI-MICROSERVICE/mi/anthology/all", Object.class);</pre>
<p>Zuul API Gateway</p>	<ul style="list-style-type: none"> • Separate Server • Add spring-cloud-starter-netflix-zuul dependency • Add @EnableZuulProxy to main class • Update application.yaml file with Route Details <pre># WITH Eureka Server zuul: routes: edcservice: path: /edc/** serviceId: EDC-MICROSERVICE miservice: path: /mi/** serviceId: MI-MICROSERVICE</pre>
<p>Sleuth and Zipkin</p>	<p>Distributed Tracing.</p> <ul style="list-style-type: none"> • Just adding dependency logs will generated with TracelD • Run Server cd C:\kafka java -jar zipkin.jar <p>Zipkin is UI for log Tracing. We need to following steps in our microservices to push Distributed Trace logs to ZipKin</p> <ul style="list-style-type: none"> • Add spring-cloud-starter-sleuth Dependency in (EDC, MI) to enable Sleuth • Add spring-cloud-starter-zipkin to push logs to ZipKin • Update application.props with zipkin server deratils <pre>spring.zipkin.base-url=http://localhost:9411/ spring.sleuth.sampler.probability=1</pre>
<p>UserServiceWithKafka</p>	<p>ProducerService :</p> <ul style="list-style-type: none"> • Add kafka dependencies • @EnableKafka in SpringBoot main class. • Update application.props with Server Details & Topic name topic.name.producer=user.provision.topic

	<ul style="list-style-type: none"> Use <code>kafkaTemplate.send(kafkaTopic, data);</code> to send msg to Topic <p>ConsumerService:</p> <ul style="list-style-type: none"> Add kafka dependencies <code>@EnableKafka</code> in SpringBoot main class. Update application.props with Server Details & Topic name <code>topic.name.producer=user.provision.topic</code> Use <code>@KafkaListener</code> to listen topic & Consume messages <pre>@KafkaListener(topics="\${topic.name.consumer}", groupId = "group_id") public void consume(ConsumerRecord<String, String> msg) { System.out.println("Tópico: {}"+ msg); }</pre>
EDCMicroService	
MIMicroService	

11. SpringBoot – Swagger

Swagger is a set of open source tools for writing REST-based APIs. It simplifies the process of writing APIs by notches, specifying the standards & providing the tools required to write beautiful, safe, performant & scalable APIs.



Adding Swagger to Spring Boot

- Getting the Swagger 2 Spring dependency
- Enabling Swagger in your code
- Configuring Swagger
- Adding details as annotations to APIs



1. Add Dependency – it will generate JSON documentation

```
<!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger2 -->
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger2</artifactId>
  <version>2.9.2</version>
</dependency>

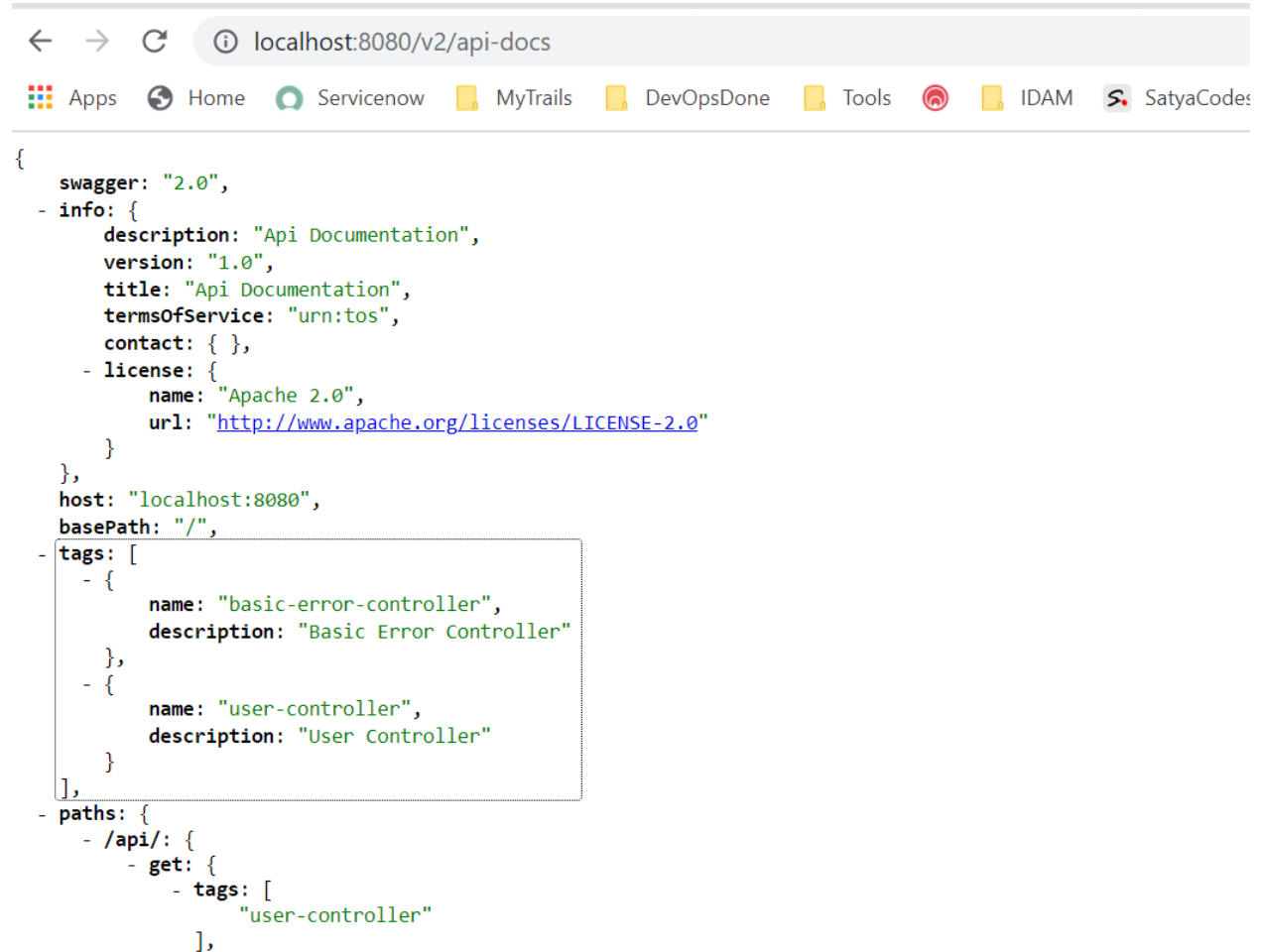
  <dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>20.0</version>
  </dependency>
```

2. Enable Swagger2 in our SpringBoot application

```
@EnableSwagger2
@SpringBootApplication
public class SatyaCodesAuthServicesApplication {

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

Run the Application, access <http://localhost:8080/v2/api-docs>



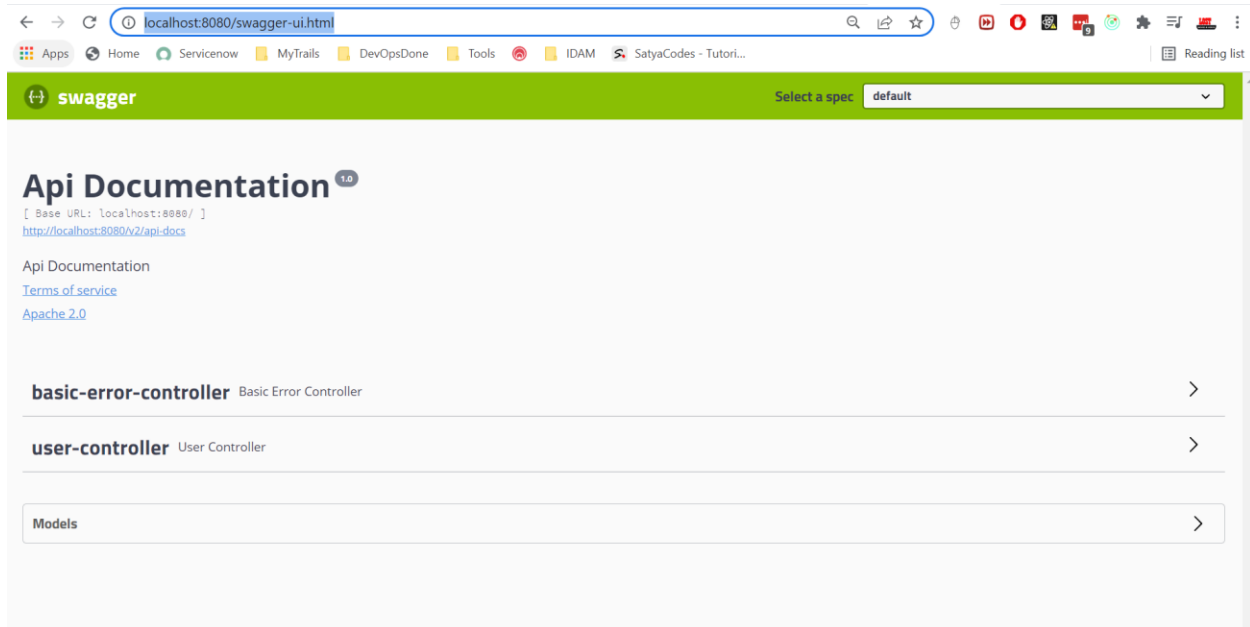
```
{
  swagger: "2.0",
  - info: {
    description: "Api Documentation",
    version: "1.0",
    title: "Api Documentation",
    termsOfService: "urn:tos",
    contact: { },
    - license: {
      name: "Apache 2.0",
      url: "http://www.apache.org/licenses/LICENSE-2.0"
    }
  },
  host: "localhost:8080",
  basePath: "/",
  - tags: [
    - {
      name: "basic-error-controller",
      description: "Basic Error Controller"
    },
    - {
      name: "user-controller",
      description: "User Controller"
    }
  ],
  - paths: {
    - /api/: {
      - get: {
        - tags: [
          "user-controller"
        ],

```

To get UI Documentenation add below dependency

```
<!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger-ui -->
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-swagger-ui</artifactId>
  <version>2.9.2</version>
</dependency>
```

To access UI : <http://localhost:8080/swagger-ui.html>



Using Swagger 3

- No need to add **@EnableSwagger2**
- **Just add below dependency & access <http://localhost:8080/swagger-ui.html>**

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.6.6</version>
</dependency>
```

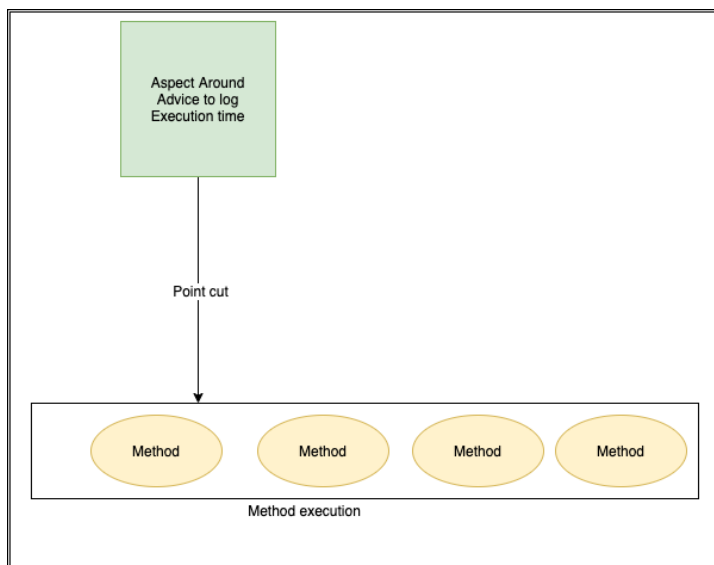
12.SpringBoot – Performance Monitoring

Development - Spring Boot Performance Logging

To calculate how much time taken to execute method, with start & end time. We use AOP `aspectj` based interceptor to write **performance logging** based on time taken in method executions of aop intercepted methods.

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

Log the method execution time using AOP for a spring boot application.



Spring AOP

1. Create Aspect class by Implementing Around advice

- Create a class and use annotation **@Aspect** on class to make it AOP aspect.
- Write logic for logging inside a method & annotate with **@Around** to make its will before & after method execution
- Provide package name where the Aspect will be introduced in method calls.
- We used **StopWatch** class to capture start time of method and end time of method.
- The **ProceedingJoinPoint** class argument provides us the class name and method name which is going to be executed.
- You may use enabling/disabling component with properties with **ConditionalOnExpression** annotation.

```

@Component
@Aspect
@ConditionalOnExpression("${performance.logging.aspect.active:true}")// enabled by default
public class PerformanceLoggingAspect {
    private static final Logger LOGGER =
    LogManager.getLogger(PerformanceLoggingAspect.class);

    //AOP expression for which methods shall be intercepted
    @Around("execution(* com.edc.controller.*(..))")
    public Object profileAllMethods(ProceedingJoinPoint proceedingJoinPoint) throws
    Throwable
    {
        MethodSignature methodSignature = (MethodSignature)
        proceedingJoinPoint.getSignature();

        //Get intercepted method details
        String className = methodSignature.getDeclaringType().getSimpleName();
        String methodName = methodSignature.getName();

        final Stopwatch stopWatch = new Stopwatch();

        //Measure method execution time
        stopWatch.start();
        Object result = proceedingJoinPoint.proceed();
        stopWatch.stop();
    }
}

```



```

        //Log method execution time
        LOGGER.info("====> \n \n \n Execution time of " + className + "." + methodName +
" :: " + stopWatch.getTotalTimeMillis() + " ms");

        return result;
    }
}

```

On executing any controller method, we can see logs with Class: method name

```

/testReport/all
2022-03-28 13:04:20.133 INFO 20128 --- [nio-8031-exec-3]
com.edc.utils.PerformanceLoggingAspect :====>
Execution time of TestReportController.getAllTestReports :: 3 ms

```

We can enable / Disable AOP by configuring below in **application.properties**

```

logging.level.root=INFO
performance.logging.aspect.active=true

```

Production: SpringBoot Acuator - Health check, Auditing, Monitoring

Actuator brings production-ready features to our application. **Monitoring our app, gathering metrics, understanding traffic or the state of our database.**

```

<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:

- **auditevents** – Audit events of your application
- **info**– Displays information about your application – you can customize to show version information.
- **health** – application's health status
- **metrics** – various metrics information of application
- **loggers** – Displays and modifies configured loggers
- **logfile** – Shows the contents of the log file
- **httptrace** – Displays HTTP trace info for the last 100 HTTP request/response
- **env** – Displays current environment information
- **flyway** – Shows Flyway database migrations details
- **liquibase** – Shows details of liquibase database migrations
- **shutdown** – Allows to shut down the application gracefully
- **mappings**– Displays a list of all **@RequestMapping** paths
- **scheduledtasks** – Displays the scheduled tasks in the application

- **threaddump** – Performs a thread dump
- **headdump** – Returns JVM head dump

You can access all available endpoints 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 class 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" ;
    }
}
```

localhost:8080/actuator/helloEndpoint

Hello

Few more Endpoints

localhost:8080/actuator/auditevents

```
{
  "events": [
    {
      "timestamp": "2019-02-10T18:21:46.464Z",
      "principal": "anonymousUser",
      "type": "AUTHORIZATION_FAILURE",
      "data": {
        "details": {
          "remoteAddress": "0:0:0:0:0:0:1",
          "sessionId": null
        },
        "type": "org.springframework.security.access.AccessDeniedException",
        "message": "Access is denied"
      }
    },
    {
      "timestamp": "2019-02-10T18:22:45.986Z",
      "principal": "user",
      "type": "AUTHENTICATION_SUCCESS",
      "data": {
        "details": {
          "remoteAddress": "0:0:0:0:0:0:1",
          "sessionId": "ADF23C6CD682F9FFAD445C497F61D38B"
        }
      }
    }
  ]
}
```

```
localhost:8080/actuator/beans

{
  "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:3080/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"
    }
  }
}
```

- env – Displays current environment information
- flyway – Shows Flyway database migrations details
- liquibase – Shows details of liquibase database migrations
- shutdown – Allows to shut down the application gracefully
- mappings– Displays a list of all @RequestMapping paths
- scheduledtasks – Displays the scheduled tasks in the application
- threaddump – Performs a thread dump
- headdump – Returns JVM head dump

13.SpringBoot – Unit Testing with Mockito

<https://frontbackend.com/spring-boot/spring-boot-2-junit-5-mockito>

- **Application** - the main Spring Boot application class used for starting web container,
- **TestReportController** - Spring Rest Controller for testing purposes,
- **TestReportRepository** - Spring Service used to check how autowire works in tests,
- **TestReportControllerMockitoTest** - test for **TestReportController** using **Mockito**,
- **TestReportControllerMockMvcTest** - test for **TestReportController** using **MockMvc**,
- **TestReportCon*RestTemplTest** - test for **TestReportController** using **TestRestTemplate**.

Controller Class

```
@RestController
public class TestReportController {
    @Autowired
    TestReportRepository repository;

    @Autowired
    RestTemplate restTemplate;

    @GetMapping("/testReport/{id}")
    public ResponseEntity<TestReport> getReportById(@PathVariable int id){
        TestReport data = repository.getById(id);
        return ResponseEntity
            .ok()
            .contentType(MediaType.APPLICATION_JSON)
            .body(data);
    }

    @GetMapping(value = "/testReport/all", produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<List<TestReport>> getAllTestReports() {
        List<TestReport> reports = new ArrayList<TestReport>();
        repository.findAll().forEach(reports::add);
        if (reports.isEmpty()) {
            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        }
        return new ResponseEntity<>(reports, HttpStatus.OK);
    }

    @GetMapping(value = "/testReport/mi", produces = "application/json")
    public ResponseEntity<Object> getAllMIReportsNormalcall() {
        ResponseEntity<Object> responseEntity = restTemplate.getForEntity("http://MI-
MICROSERVICE/mi/anthology/all", Object.class);
        return new ResponseEntity<>(responseEntity, HttpStatus.OK);
    }
}
```

Using MockMvc -

We used the `MockMvc` class and `@AutoConfigureMockMvc` that will configure it and inject it into the tested class. The `MockMvc` class is used to perform API calls, but instead of doing HTTP requests, Spring will test only the implementation that handle them in `TestReportController`

```
@SpringBootTest
@AutoConfigureMockMvc
public class TestReportControllerMockMvcTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void getReportByIdTest() throws Exception{
        this.mockMvc.perform(get("/testReport/1"))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("Blod Report")));
    }

    @Test
    public void getAllResportsTest() throws Exception{
        this.mockMvc.perform(get("/testReport/all"))
    }
```

```

        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().contentType(MediaType.APPLICATION_JSON));
    }
}

```

Using TestRestTemplate

Spring boot test that makes use of **TestRestTemplate** to call REST API. In this approach, we can use **@Autowired** annotation just like in runtime applications. Spring will interpret them and do the necessary injections. In **@SpringBootTest** tests real Spring Boot application server is being started.

In the test we used:

- **webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT** - to start the server with a random port in order to avoid any port conflicts,
- **@LocalServerPort** - this annotation tells Spring to inject a random port to the specific field,
- **TestRestTemplate - RestTemplate** for tests used to make a real HTTP requests.

```

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class TestReportControllerRestTemplateTest
    @LocalServerPort
    private int port;
    private String url;

    @Autowired
    private TestRestTemplate restTemplate;

    @BeforeEach
    public void setUp() {
        url = String.format("http://localhost:%d/", port);
    }

    @Test
    public void greetingShouldReturnDefaultMessage() {
        assertThat(this.restTemplate.getForObject(url+"testreport/1",
String.class)).contains("Report");
    }
}

```

Using Mockito unit test

- **@Mock** creates a mock.
- **@InjectMocks** creates an instance of the class and injects the mocks that are created with the **@Mock** (or **@Spy**) annotations into this instance.

In above controller class has two dependent objects. First, we need to mock those objects & then we need to inject them to main controller class

```

    @Autowired
    TestReportRepository repository;

    @Autowired
    RestTemplate restTemplate;
    @ExtendWith(MockitoExtension.class)
    public class TestReportControllerTest {

```

```

@Mock
TestReportRepository repository;

@Mock
RestTemplate restTemplate;

@InjectMocks
TestReportController controller;

@BeforeEach
void setMockOutput() {
    when(repository.getById(1)).thenReturn(new TestReport("Report", "", "", "", ""));
}

@Test
public void getReportByIDTest() throws Exception{
    assertThat(controller.getReportByID(1).getBody().getReportName().equalsIgnoreCase("Report"));
}
}

```

If any datasource object is their, just mock annotation will automatically creates Connection object with test properties. Try adding your datasource as a [@MockBean](#) too:

```

@MockBean
private DataSource dataSource

```

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).

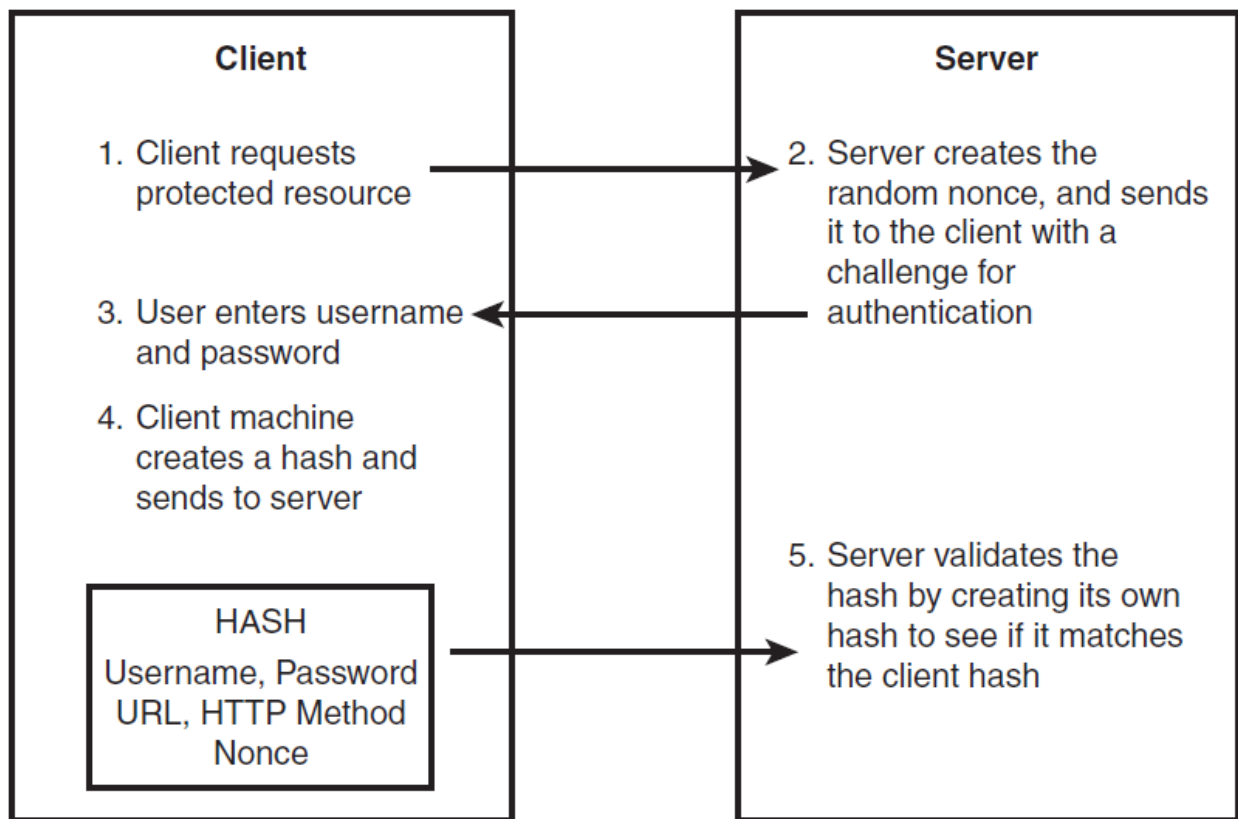
HTTP Basic Authentication

- 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.

Review permissions

 Personal user data Full access	▼
 Repositories Public and private	▼
 Notifications Read access	▼
 Gists Read and write access	▼

Authorize application

ID Tokens (Authentication only – Username/Password)

The ID token is the core extension that [OpenID Connect](#) makes to OAuth 2.0. ID tokens are issued by the authorization server and contain claims that carry information about the user.

ID tokens are [JSON web tokens \(JWT\)](#). These ID tokens consist of a **header**, **payload**, and **signature**. The **header and signature** are used to verify the authenticity of the token, while the payload contains the information about the user requested by your client.

By default, an ID token is valid for **one hour** - after one hour, the client must acquire a new ID token. You can adjust the lifetime of an ID token to control how often the client application expires the application session, and how often it requires the user to re-authenticate either **silently or interactively**.

Access Tokens

An access token is a string that identifies a user, an application, or a page. The token includes information such as when the token will expire and which app created that token.

- First, it is necessary to acquire OAuth 2.0 client credentials from API console.
- Then, the access token is requested from the authorization server by the client.
- It gets an access token from the response and sends the token to the API that you wish to access.

Used for calling Graph Api's

3.Refreshing tokens

The implicit grant does not provide refresh tokens. Both **id_tokens** and **access_tokens** will expire after a short period of time, so your app must be prepared to refresh these tokens periodically. To refresh either type of token, you can perform the same hidden iframe request from above using the **prompt=none** parameter to control the identity platform's behavior. If you want to receive a new id_token, be sure to use id_token in the response_type and scope=openid, as well as a nonce parameter.

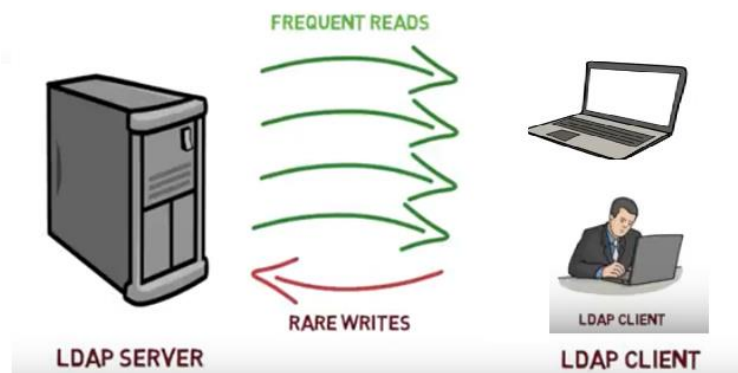
In browsers that do not support third party cookies, this will result in an error indicating that no user is signed in.

4.Send a sign out request

The OpenID Connect **end_session_endpoint** allows your app to send a request to the Microsoft identity platform to end a user's session and clear cookies set by the Microsoft identity platform. To fully sign a user out of a web application, your app should end its own session with the user (usually by clearing a token cache or dropping cookies), and then redirect the browser to:

```
https://login.microsoftonline.com/{tenant}/oauth2/v2.0/logout?post_logout_redirect_uri=https://localhost/myapp/
```

LDAP



<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);

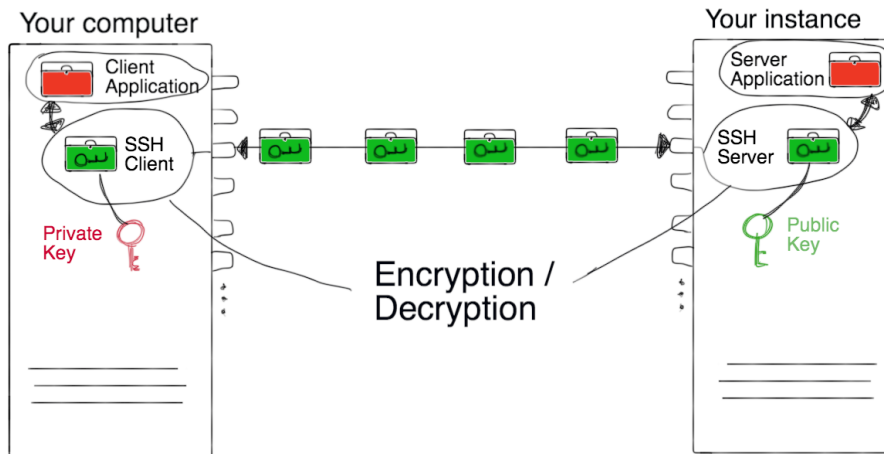
        try {
            DirContext authContext = new InitialDirContext(authEnv);
            System.out.println("Authentication Success!");
        } catch (AuthenticationException authEx) {
            System.out.println("Authentication failed!");
        } catch (NamingException namEx) {
            System.out.println("Something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

```

SSH – Only for LINUX Server / CommandLine(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: $0*2^0 + 1*2^1 + 1*2^2 + 0*2^3 + 0*2^4 + 1*2^5 = 0+2+4+0+0+32$.
- 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

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://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#.W17wB1N965t>
- <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://avalides.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/>
- <https://www.coachdevops.com/2020/06/deploy-python-app-into-kubernetes.html>
- <https://www.sivalabs.in/2018/03/microservices-using-springboot-spring-cloud-part-1-overview/>
- <https://github.com/sivaprasadreddy/spring-boot-microservices-series>
- <https://www.javainuse.com/misc/apache-kafka-hello-world>

OnlineHTML - <https://www.froala.com/online-html-editor>

Errors & Solutions

Error response from daemon: open \\.\pipe\docker_engine_linux: The system cannot find the file specified.

Restarting Docker Desktop for Windows helped me. You can do that by right-click on tray icon and selecting restart.

ERROR: ERROR: Can't construct a java object for tag:yaml.org,2002:io.kubernetes.client.openapi.models.V1Deployment; exception=Class not found: io.kubernetes.client.openapi.models.V1Deployment

I found the root cause of this behaviour. Jackson 2 API plugin version 2.11.1 is breaking kube deployments; you can find more info by the link below:

<https://issues.jenkins-ci.org/browse/JENKINS-62995>

Downgrading the following plugins worked for me:

- Jackson 2 API v2.10.0,
- Kubernetes v1.21.3,
- Kubernetes Client API v4.6.3-1,
- Kubernetes Continuous Deploy v2.1.2,
- Kubernetes Credentials v0.5.0

As those plugins are default, you would need to find the relevant version source files in <https://plugins.jenkins.io/>, and upload them to your Jenkins server by going Manage Jenkins --> Manage Plugins --> Advanced --> Upload Plugin section

To manually downgrade the Jenkins plugin:

1. Shut down Jenkins

2. Delete the plugin.jpi file and the plugin folder from `${user_home}/.jenkins/plugins`
3. Place the older plugin.hpi file
4. Start Jenkins.

React Application

Create React App & Integrate Azure AD

Download code:

```
git clone https://github.com/Azure-Samples/ms-identity-javascript-react-spa
npm start
```

Our Application will run in: <http://localhost:3000/>

Register your localhost application in Azure App Registrations

- Sign in to the [Azure portal](#).
- Search for and select **Azure Active Directory**.
- Under Manage, select **App registrations** > **New registration**.
- Under **Redirect URIs**, enter redirect URI. Select Implicit grant and hybrid flows.

The screenshot shows the Azure portal interface for a new application registration. The breadcrumb path is 'Home > Default Directory > LocalHost'. The left sidebar contains navigation options: Overview, Quickstart, Integration assistant, Manage (with sub-items: Branding & properties, Authentication, Certificates & secrets, Token configuration), and a search bar. The main content area is titled 'LocalHost' and includes a search bar and action buttons: Delete, Endpoints, and Preview features. Under the 'Essentials' section, the following details are visible: Display name (LocalHost), Application (client) ID (8514c420-02d8-47ee-8827-67b05a777249), Object ID (364b9c6f-0d01-4427-8600-06e07d9d273d), Directory (tenant) ID (afda05f5-bdcb-45cd-b43b-4f42d1042842), and Supported account types (Multiple organizations). On the right side, there are links for Client credentials (Add a certificate or secret), Redirect URIs (0 web, 1 spa, 0 public client), Application ID URI (Add an Application ID URI), and Managed application in local directory (LocalHost).

Open `authConfig.js` with `client-id`, `authority` & `redirect` Details

```
auth: {
  clientId: "8514c420-02d8-47ee-8827-67b05a777249",
  authority: "https://login.microsoftonline.com/afda05f5-bdcb-45cd-b43b-4f42d1042842",
  redirectUri: "http://localhost:3000"
},
```

<https://www.thirdrocktechkno.com/blog/microsoft-login-integration-with-react/>

Configure Roles & Groups in Azure AD

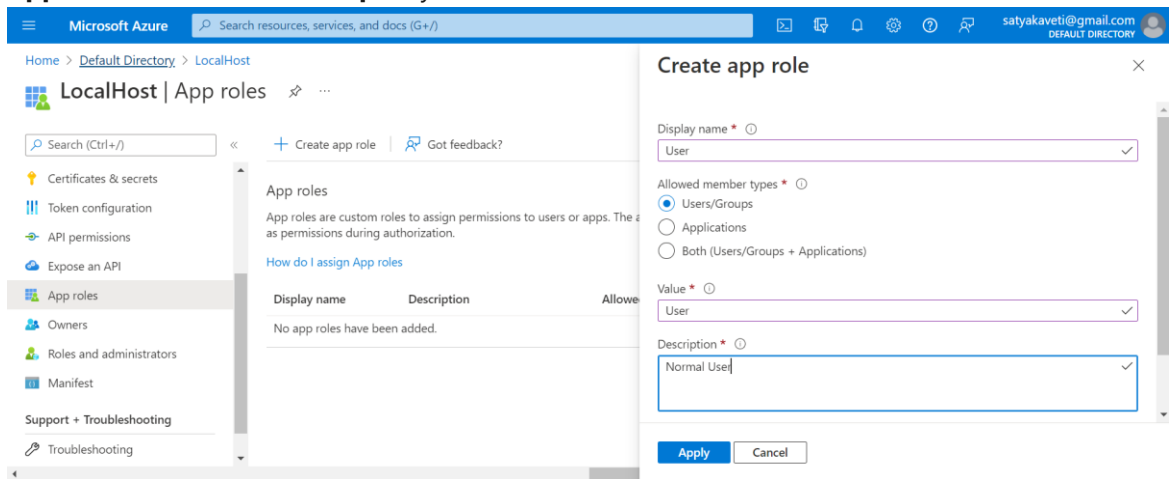
<https://stackoverflow.com/questions/62082149/validate-azure-token-in-a-spring-boot-application>

<https://github.com/Azure-Samples/azure-spring-boot-samples/tree/main/aad/azure-spring-boot-starter-active-directory/aad-resource-server-by-filter-stateless>

Add app roles to your application and receive them in the token

<https://docs.microsoft.com/en-us/azure/active-directory/develop/howto-add-app-roles-in-azure-ad-apps>

1. Sign in to the [Azure portal](#).
2. Active Directory > **App registrations**> select the **Application** > Manage > App Roles > **Create App role** > select **User/Group** only



App roles

App roles are custom roles to assign permissions to users or apps. The application defines and publishes the app roles and interprets them as permissions during authorization.

[How do I assign App roles](#)

Display name	Description	Allowed member types	Value	ID	State
User	Normal User	Users/Groups	user	6f70a538-1859-4348-...	Enabled
Admin	Admin All Access	Users/Groups	admin	e8e8e626-c0af-467d-9...	Enabled
TrilaManager	TrilaManager	Users/Groups	trilamanager	bba5e558-5031-4e1c-...	Enabled

Assign users and groups to roles

1. Sign in to the [Azure portal](#).
2. **Active Directory** > **Enterprise applications** > Select your **Application**
3. Under **Manage**, select **Users and groups** > **Add user** > **Add Assignment** pane.

+ Add user/group Edit Remove Update Credentials Columns Got feedback?

i The application will not appear for assigned users within My Apps. Set 'visible to users?' to yes in properties to enable this. →

First 200 shown, to search all users & groups, enter a display name.

	Display Name	Object Type	Role assigned
<input type="checkbox"/>	SK Satyanarayana Kaveti	User	Default Access
<input type="checkbox"/>	AO ADMIN OF EVERYTHING	User	Admin
<input type="checkbox"/>	TM trial manager	User	TrilaManager
<input type="checkbox"/>	SK Satyanarayana Kaveti	User	Admin
<input type="checkbox"/>	US user1	User	User
<input type="checkbox"/>	U2 User 2	User	User

SpringBoot JWT Authentication

<https://github.com/Azure-Samples/azure-spring-boot-samples/tree/main/aad/azure-spring-boot-starter-active-directory/aad-resource-server-by-filter-stateless>

Update Maven dependencies

```
<!-- Azure Active Directory JWT Authentication Depenedecies -->
<dependency>
  <groupId>com.azure.spring</groupId>
  <artifactId>azure-spring-boot-bom</artifactId>
  <version>3.14.0</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>

<dependency>
  <groupId>com.azure.spring</groupId>
  <artifactId>azure-spring-boot-starter-active-directory</artifactId>
  <version>3.14.0</version>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
  <version>2.5.5</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-client</artifactId>
  <version>5.5.1</version>
</dependency>

<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-oauth2-jose</artifactId>
  <version>5.4.6</version>
</dependency>
```

Enable SpringSecurity add AADAuthenticationFilter

```
@EnableGlobalMethodSecurity(securedEnabled = true,
    prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private AADAuthenticationFilter aadAuthFilter;

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http.authorizeRequests()
            .antMatchers("/edc").permitAll()
            .antMatchers("*/testReport/*").authenticated()
            .anyRequest().permitAll()
            .and()
            .csrf() // Cross-Site Request Forgery
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            .and()
            .logout()
            .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
            .deleteCookies("JSESSIONID")
            .logoutSuccessUrl("/")
            .invalidateHttpSession(true)
            .and()
            .addFilterBefore(aadAuthFilter, UsernamePasswordAuthenticationFilter.class);
    }
}
```

Update Application.yaml file

what happens if both application.properties and application.yml available ?

I can answer my own question, as it just works as you would expect. The application.yml file and the appropriate application-\${profile}.properties both get loaded and merged into the environment. Spring boot just makes this work naturally

<https://www.javaguides.net/2020/08/reactjs-axios-get-post-put-and-delete-example-tutorial.html>

React – Employee Front End

1.Create EmployeeReactApp – Empty Repo

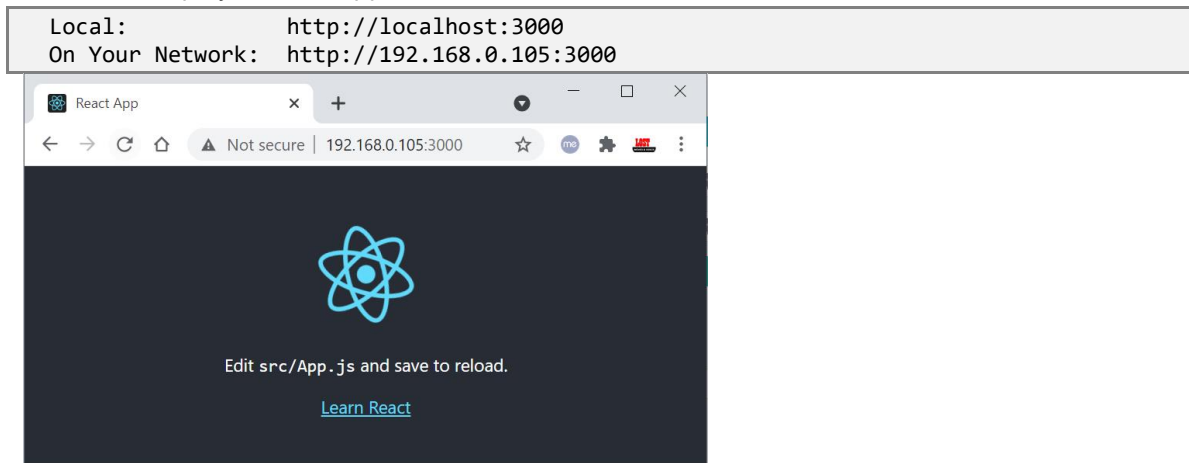
Open Visual Studio Code > Terminal

```
npm create-react-app employee-react-app
```

Start react app

```
cd employee-react-app
npm start
```

You can now view employee-react-app in the browser.



Main files

- `public/index.html` – Single page appl'n , contains `<div id="root"></div>`

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

- `src/App.js` - Contains Component code to generate UI.

```
import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">

        <h1>Hello, World!!!</h1>

      </header>
    </div>
  );
}
```

```
export default App;
```

- src/index.js - get code from component & replace div content to render UI.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

3. Get All Employees – React Code

employee-react-app\src\component\ListAllEmployeesComponent.jsx

```
import React, { Component } from 'react';
import EmployeeService from '../services/EmployeeService';

class ListAllEmployeesComponent extends Component {

  constructor(props) {
    super(props)
    this.state = {
      employees: []
    }
  }

  componentDidMount(){
    EmployeeService.getEmployees().then((res) => {
      this.setState({ employees: res.data});
    });
  }

  render() {
    return (
      <div>
        <table className="table">
          <thead>
            <tr>
              <th scope="col">ID</th>
              <th scope="col">Name</th>
              <th scope="col">Address</th>
              <th scope="col">Salary</th>
              <th scope="col">Action</th>
            </tr>
          </thead>
          <tbody>
            {
              this.state.employees.map(
                employee =>
                <tr key = {employee.id}>
```

```

        <td> {employee.name} </td>
        <td> {employee.address}</td>
        <td> {employee.salary}</td>
        <td>
            Action
        </td>
    </tr>
)
}
</tbody>
</table>
</div>
);
}
}
export default ListAllEmployeesComponent;

```

employee-react-app\src\services\EmployeeService.js

```

import axios from 'axios';
const EMPLOYEE_API_BASE_URL = "http://localhost:8080/api/v1/all";
class EmployeeService {
    getEmployees() {
        return axios.get(EMPLOYEE_API_BASE_URL);
    }
}
export default new EmployeeService()

```

Now if we open Root URL, it will display the Table Data.

<http://localhost:3000>

Employees List

Show entries Search:

ID	Name	Address	Salary	Action
	Satya	HYDERABAD	57000	Action
	DILEEP	BANGLORE	32189	Action

Showing 1 to 2 of 2 entries Previous 1 Next

3. Header & Footer

employee-react-app\src\component\HeaderComponent.jsx

```

import React, { Component } from 'react';
class HeaderComponent extends Component {
    render() {
        return (

```



```

        <div>
          <nav class="navbar navbar-dark bg-dark" >

            <div className="container">
              <h3 style={{'color':'white'}}>EmployeeServices™ </h3>
            </div>

          </nav>

        </div>
      );
    }
  }
}
export default HeaderComponent;

```

employee-react-app\src\component\FooterComponent.jsx

```

import React, { Component } from 'react';

class FooterComponent extends Component {
  render() {
    return (
      <div>
        <footer className="footer">
          <div className="container">
            <span className="text-muted">
              © SatyaCodes 2020 , Satya Kaveti's Writing.
            </span>
          </div>
        </footer>
      </div>
    );
  }
}
export default FooterComponent;

```

employee-react-app\src\App.js

```

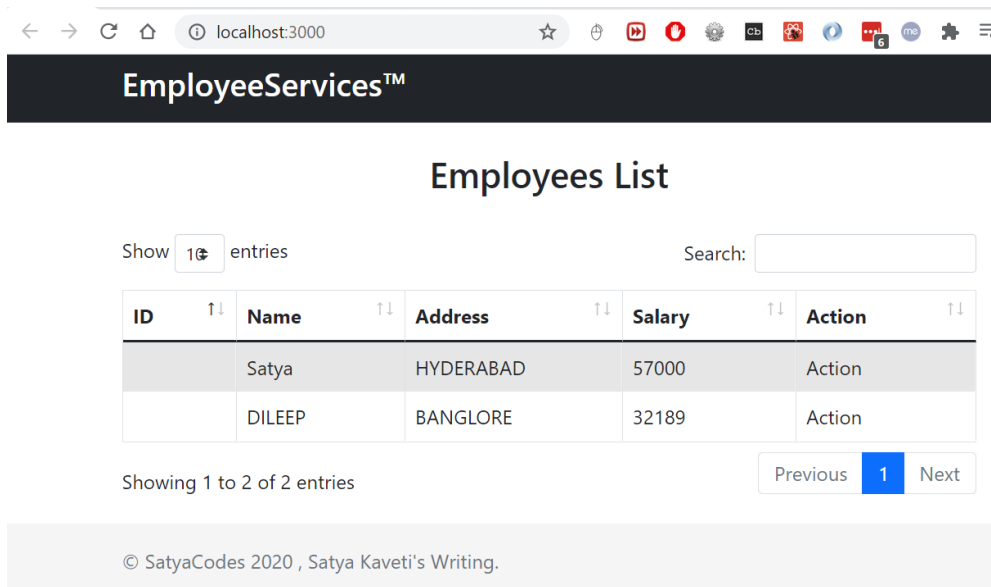
import './App.css';
import HeaderComponent from './component/HeaderComponent';
import FooterComponent from './component/FooterComponent';
import ListAllEmployeesComponent from './component/ListAllEmployeesComponent';

function App() {
  return (

    <div>
      <HeaderComponent/>
      <div className="container">
        <ListAllEmployeesComponent/>
      </div>
      <FooterComponent/>
    </div>
  );
}

export default App;

```



Routing

Install [react-router-dom](#)

```
npm install react-router-dom --save
```

```
import './App.css';
import HeaderComponent from './component/HeaderComponent';
import HomeComponent from './component/HomeComponent';
import FooterComponent from './component/FooterComponent';
import ListAllEmployeesComponent from './component/ListAllEmployeesComponent';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import Employee from './component/Employee';
function App() {
  return (
    <div>
      <HeaderComponent />
      <BrowserRouter>
        <div className="container">
          <Switch>
            <Route exact path="/" component={HomeComponent}></Route>
            <Route exact path="/one" component={Employee}></Route>
            <Route exact path="/all" component={ListAllEmployeesComponent}></Route>
            <Route component={Error} />
          </Switch>
        </div>
      </BrowserRouter>
      <FooterComponent />
    </div>
  );
}
export default App;
```

Complete Code. Github

<https://github.com/smlcodes/Books-Sync-Gitlab/tree/main/Codes/employee-react-app>

Ref.

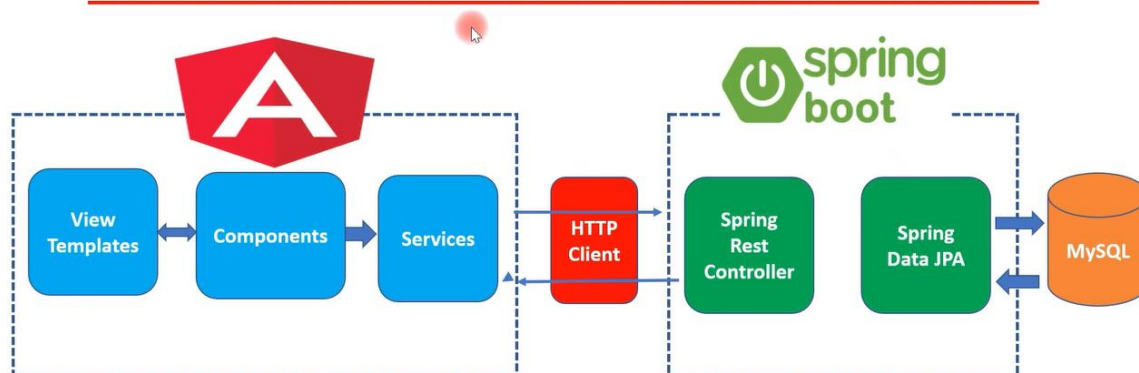
<https://github.com/RameshMF/ReactJS-Spring-Boot-CRUD-Full-Stack-App>

<https://www.javaguides.net/2020/07/spring-boot-react-js-crud-example-tutorial.html>

<https://www.youtube.com/watch?v=n43h1eJ2EUE&list=PLGRDMO4rOGcNLnW1L2vgsExTBg-VPoZHR>

Angular + SpringBoot

Angular 10 + Spring Boot CRUD Full Stack



We are using above EmployeeServices as SpringBoot rest API, we create UI using Angular to display UI.

Create Angular Project

You use the Angular CLI to create projects, generate application and library code, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

```
npm install -g @angular/cli
```

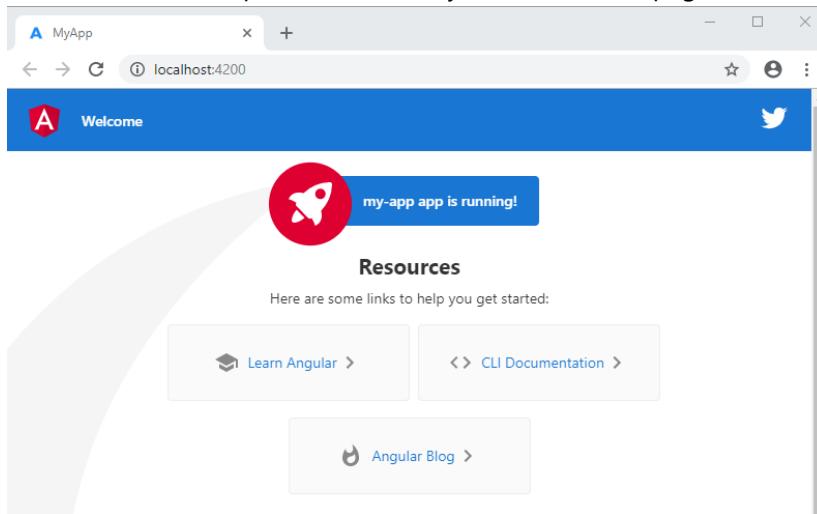
To create a new workspace and initial starter app.

```
ng new employee-angular-app
```

Run the application

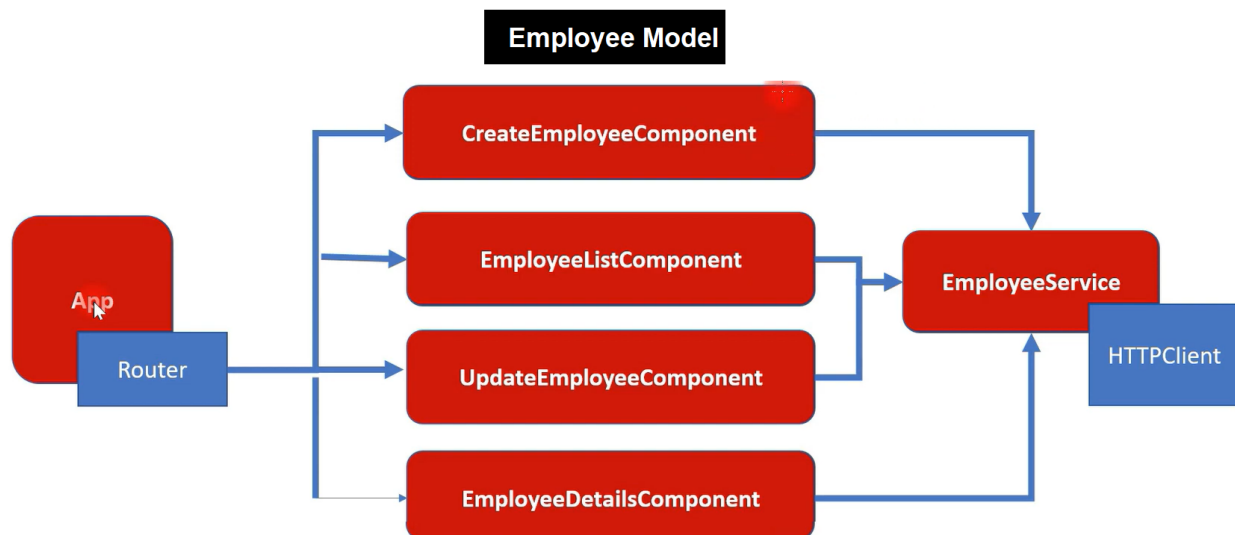
```
ng serve --open
```

The `--open` option automatically opens your browser to `http://localhost:4200/`. If your installation and setup was successful, you should see a page similar to the following.



Inside the `src/app/` folder contains your project's logic and data.

- (5) Defines the base **CSS stylesheet** for the root **AppComponent**
- (4) Defines the **HTML template** associated with the root **AppComponent**.
- (3) Defines the **logic** for the application's root component, named **AppComponent**
- (6) Defines the root module, named **AppModule**, that tells Angular how to assemble the application. Initially declares only the **AppComponent**. you can add more components if required.
- (1) Root HTML of single Page Application. contains `<app-root></app-root>`
- (2) The main **entry point for your application**. Compiles the application with the JIT compiler and bootstraps the application's root module (**AppModule**) to run in the browser.



for generate any thing we need to use ng `g c/s`

- **g** -is for Generate
- **C** – is for Component
- **S**– is for Service

EmployeeModel - Model class to hold Employee Data

```

C:\Git\books\Codes\employee-angular-app>ng g class employee
CREATE src/app/employee.spec.ts (162 bytes)
CREATE src/app/employee.ts (26 bytes)
  
```

Create Employee Model class with Employee Table Properties

```

export class Employee {
  id!: number;
  name!: string;
  address!: string;
  salary!: number;
}
  
```

EmployeeListComponent - To Display all Employee data

```

C:\Git\books\Codes\employee-angular-app>ng g c EmployeeList
CREATE src/app/employee-list/employee-list.component.html (To display UI)
CREATE src/app/employee-list/employee-list.component.spec.ts
CREATE src/app/employee-list/employee-list.component.ts (Logic)
CREATE src/app/employee-list/employee-list.component.css (Stylesheet)
UPDATE src/app/app.module.ts (newly created component is added & imported)
  
```

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { EmployeeListComponent } from './employee-list/employee-list.component';

@NgModule({
  declarations: [
    AppComponent,
    EmployeeListComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Above are automatically added after generation of component.

employee-list.component.html

```

<h1>
  Employees List Page
</h1>

```

//employee-list.component.ts (Default Generated Class)

```

import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-employee-list',
  templateUrl: './employee-list.component.html',
  styleUrls: ['./employee-list.component.css']
})
export class EmployeeListComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}

```

If we place above **app-employee-list** in main html(app.component.html) it will display data generated by employeelist component.

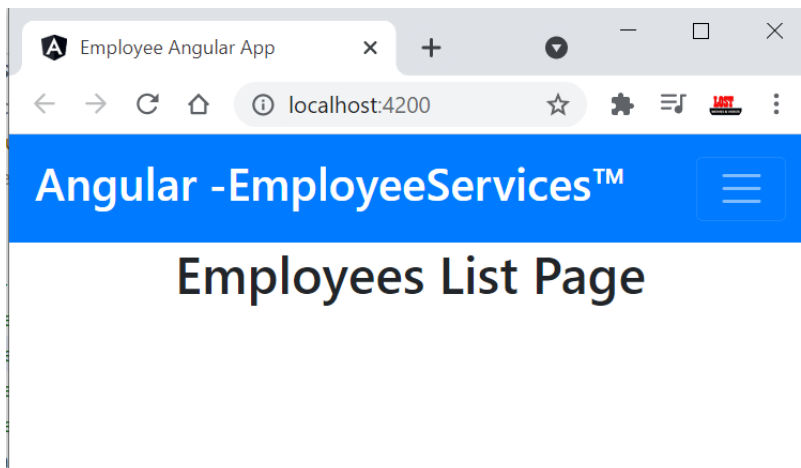
app.component.html

```

<div>
  .... Old code ....
</div>

<app-employee-list></app-employee-list>

```



Change code to display sample Employee data

employee-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { Employee } from '../employee';

@Component({
  selector: 'app-employee-list',
  templateUrl: './employee-list.component.html',
  styleUrls: ['./employee-list.component.css'],
})
export class EmployeeListComponent implements OnInit {
  //List of employee objects created, initialized to EMpty.
  employeeList!: Employee[];
  constructor() {}
  //Sample Employee Data.
  ngOnInit(): void {
    this.employeeList = [
      {
        id: 101,
        name: 'Satya',
        address: 'Init-data',
        salary: 18000,
      },
      {
        id: 102,
        name: 'RAM',
        address: 'Init-data',
        salary: 11000,
      },
    ];
  }
}
```

employee-list.component.html

```
<div class="card">
  <div class="card-body">
    <div class="col-10 mx-auto">
      <h2 class="text-center">Employees List</h2>
      <!-- <p> {{employeeList[0].name}}</p> -->
    </div>
    <br />
    <div class="row">
      <div class="col-10 mx-auto">
        <table id="example" class="table table-striped table-bordered" style="width: 100%">
```

```

<thead>
  <tr>
    <th scope="col">ID</th>
    <th scope="col">Name</th>
    <th scope="col">Address</th>
    <th scope="col">Salary</th>
    <th scope="col">Action</th>
  </tr>
</thead>
<tbody>

  <tr *ngFor="let employee of employeeList">
    <td {{employee.id}} </td>
    <td {{employee.name}} </td>
    <td {{employee.address}} </td>
    <td {{employee.salary}} </td>
    <td>
      <button class="btn btn-primary"><i class="fas fa-eye"></i> </button> <span>&nbsp;</span>
      <button class="btn btn-warning"><i class="fas fa-edit"></i> </button> <span>&nbsp;</span>
      <button class="btn btn-danger"><i class="fas fa-trash-alt"></i> </button>

    </td>
  </tr>

</tbody>
</table>
</div>
</div>
</div>
</div>
</div>

```

Angular -EmployeeServices™ Home Add Employee All Employees

Employees List

Show entries Search:

ID	Name	Address	Salary	Action
101	Satya	Init-data	18000	👁 ✎ 🗑
102	RAM	Init-data	11000	👁 ✎ 🗑

Showing 1 to 2 of 2 entries Previous **1** Next

EmployeeService – To communicate with SpringBoot Services

```

C:\Git\books\Codes\employee-angular-app>ng g s employee
CREATE src/app/employee.service.spec.ts (367 bytes)
CREATE src/app/employee.service.ts (137 bytes)

```

```
//employee.service.ts (Default generated code)
```



```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class EmployeeService {

  constructor() { }
}
```

We need to use `HttpClientModule` to call Services. We need import that module in `app.module.ts`

```
//employee.service.ts (CURD Operations Rest Call)
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http'
import { Observable } from 'rxjs';
import { Employee } from './employee';

@Injectable({
  providedIn: 'root'
})
export class EmployeeService {

  private baseUrl = "http://localhost:8080/api/v1";

  constructor(private httpClient: HttpClient) { }

  getEmployeesList(): Observable<Employee[]>{
    return this.httpClient.get<Employee[]>(`${this.baseUrl}/all`);
  }

  createEmployee(employee: Employee): Observable<Object>{
    return this.httpClient.post(`${this.baseUrl}/addEmployee`, employee);
  }

  getEmployeeById(id: number): Observable<Employee>{
    return this.httpClient.get<Employee>(`${this.baseUrl}/getEmployee/${id}`);
  }

  updateEmployee(id: number, employee: Employee): Observable<Object>{
    return this.httpClient.put(`${this.baseUrl}/updateEmployee/${id}`, employee);
  }

  deleteEmployee(id: number): Observable<Object>{
    return this.httpClient.delete(`${this.baseUrl}/deleteEmployee/${id}`);
  }
}
```

Now call `getEmployeesList()` method in `EmployeeListComponent` to get employees data.

```
// employee-list.component.ts
import { Component, OnInit } from '@angular/core';
import { Employee } from '../employee';

//1.Imported EmployeeService
import { EmployeeService } from '../employee.service';

@Component({
  selector: 'app-employee-list',
  templateUrl: './employee-list.component.html',
  styleUrls: ['./employee-list.component.css'],
})
```

```

})
export class EmployeeListComponent implements OnInit {
  //List of employee objects created, initilaized to EMpty.
  employeeList!: Employee[];

  //2. Created EmployeeService Object to call methods
  constructor(private employeeService: EmployeeService) {}

  //3.at the time of initilaizing calling getEmployees
  ngOnInit(): void {
    this.getEmployees();
  }

  //4.Finally, employeeList object contains all empdata coming from SpringBoot
  private getEmployees() {
    this.employeeService.getEmployeesList().subscribe((data) => {
      this.employeeList = data;
    });
  }
}

```

Angular - EmployeeServices™ Home Add Employee All Employees

Employees List

Show 10 entries Search:

ID	Name	Address	Salary	Action
109	Satya	Killaruru	1111	
110	Vijay	Kakinada	20000	
111	Satya	Killaruru	122000	
112	Johnny	Vizag	30000	
113	Kiran	NewZland	100000	
114	Ronaldo	Mirzapur	50000	

Showing 1 to 6 of 6 entries Previous 1 Next

Angular Routing

```

app-routing.module.ts (Default generated code)
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }

```

we need to configure **path** for each **component** like below

```
import { EmployeeListComponent } from './employee-list/employee-list.component';

const routes: Routes = [
  {path: 'all', component: EmployeeListComponent}
];
```

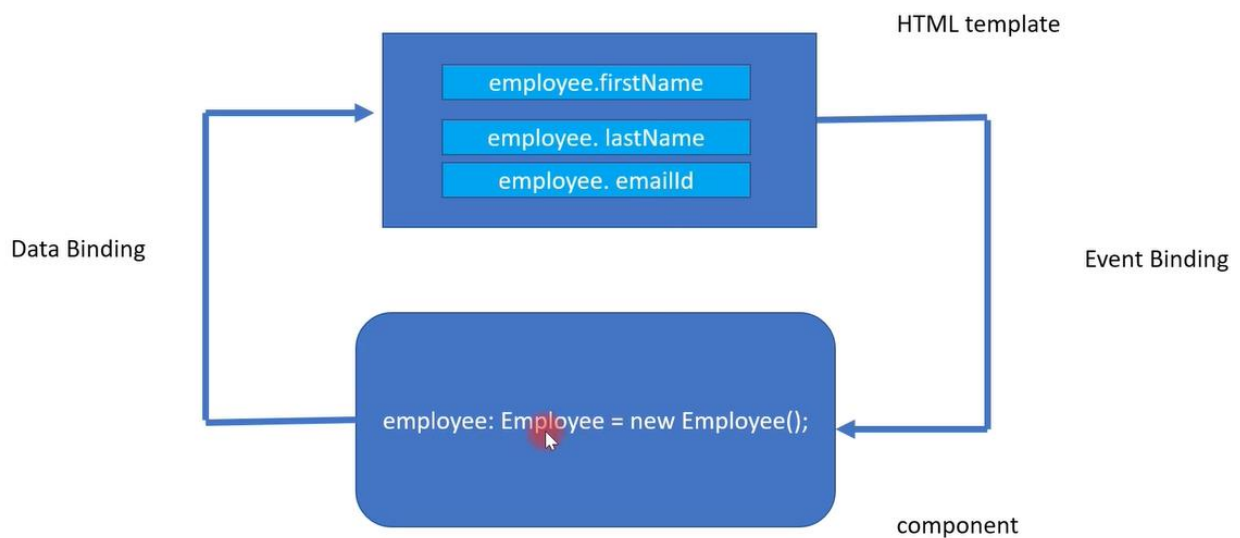
Next, in `app.component.html` – we need to place `<route-outlet>` to display the content based on `path`

```
<!--This is used directly to display all Employees
  <app-employee-list></app-employee-list> -->

<!-- Based on Route it will display content -->
<router-outlet></router-outlet>
```

Similarly, we need to create Components for ADD, UPADATE, DELETE employee

Two Way Binding



```
C:\Git\books\Codes\employee-angular-app>ng g c add-employee
CREATE src/app/add-employee/add-employee.component.html
CREATE src/app/add-employee/add-employee.component.ts
```

Complete Code. Github

<https://github.com/smlcodes/Books-Sync-Gitlab/tree/main/Codes/employee-angular-app>

Ref.

<https://github.com/RameshMF/Angular-10-Spring-Boot-CRUD-Full-Stack-App>

<https://www.youtube.com/watch?v=tLBX9fq813c&list=PLGRDMO4rOGcNzi3CpBWsCdQSZbjdWWy-f>

Ref.

<https://ordina-jworks.github.io/security/2020/08/18/Securing-Applications-Azure-AD.html>

<https://docs.microsoft.com/en-us/azure/active-directory/develop/tutorial-v2-react>

<https://blog.baeke.info/2019/01/15/simple-azure-ad-authentication-in-single-page-application-spa/>

New

<https://stackoverflow.com/questions/45593002/how-to-validate-access-tokenazure-ad-oauth-2-0-in-web-api>

<https://stackoverflow.com/questions/71321336/getting-401-error-while-trying-to-validate-a-azure-token>

<https://stackoverflow.com/questions/65182906/how-to-authenticate-spring-boot-rest-api-having-post-method-using-azure-ad>

<https://github.com/calvear93/react-azure-msal-security>