# MICROSERVICES

##### Monolithic Application:--

A Project which holds all modules together and converted as one Service (one .war file).

=>All most every project in real time is implemented using this format only.

=>In this case, if no. of users are getting increased, then to handle multiple request (load) use LBS (Load Balancing Server).

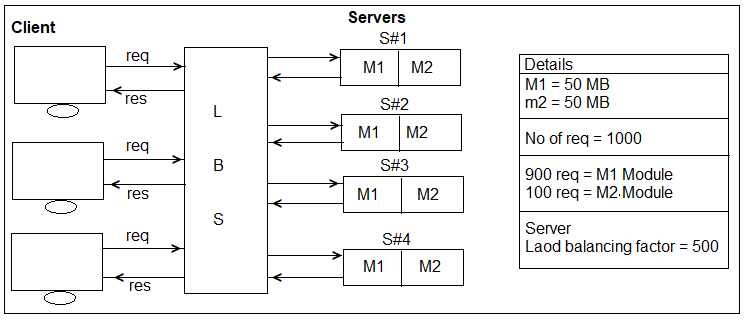
=>But few modules needs Extra load, not all. In this case other modules get memory which is waste (no use). Hence reduces performance of server (application).

\*\*Consider Project P1 is having 2 modules M1(Register), M2(Login) and their runtime memories are M1=50 MB, M2=50 MB.

=>Here, End user Register only onetime (first time) and login is mostly used module.

->Consider Application needs 100MB size in server as register 50MB and Login 50MB.

->To handle multiple client request multiple instances of sample App must be provided using Load balancing server with Load Register.

**Diagram:--** Load Balancing is done using Servers looks like.

=>In above example, M2 Module is getting fewer (100) requests from Client. So, max 2 instances are may be enough. Other 2 instances (memories) are no use. It means near 100MB memory is not used, which impacts server performance.

1. **Microservices:--** It is a independent deployment component.

=>It is a combination of one (or more) modules of a Projects runs as one Service.

=>To avaoid monolithic Limitations like memory and time (performance) problems use this design.

##### Nature of Microservices:--

1>Every Service must be implemented using **Webservices** concept. 2>One or multiples module as one Project.

3>Every service must be **independent** (One service should not effect another one, like for code modifications, version upgrades, downtime for few servers… etc).

4>It should able to communicate with any type of client (Mobile, Web based, 3rd party). 5>Every Services should able to communicate with each other Microservice, It is also called as **“Intra Communication**”.

6>Required Services must be supported for **Dynamic Load Balancing** (i.e. one service runs in multiple instances) based on click request.

7>Every microservice should support able to read input data from External **Configuration Server** [Config Server](Dynamic inputs using (\_.properties/\_.yml), with GIT/Config Server).

8>Service communication (Chain of execution) problems should be able to solve using

**Circuite Breaker** [Find other possible…].

9>All Servers must be accessed to Single Entry known as Gateway Service [ Proxy Gateway or API Gateway], It supports securing, **metering** and **Routing**.

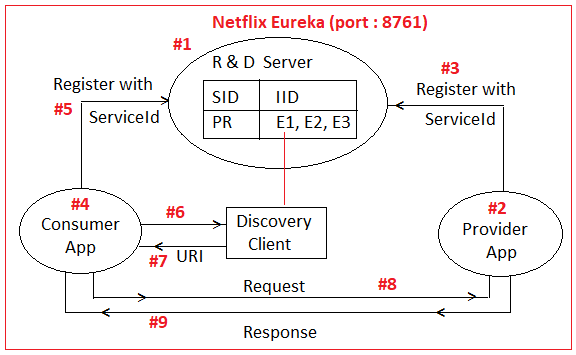
##### Microservice Generic Architecture:--

=>This design provides bility to communicate any kind of client application. Ex:-- Android, Angular, RFID, Web, 3rd party, Swings,… etc.

=>Eureka is a R & D server which supports microservices register, discover with each other for intra communication.

=>Every Microservice should get reistered with Eureka using one ServiceId (one or more instanceId).

=>For consumer application using Discovery client, get URI and make http Request to producer Application.



##### Component Names:--

1>Service Registry & Discovery = Eureka 2>Load Balancing Server = Spring

3>Circuite Breaker = Hystrix

4>API Gateway = Spring

5>Config Server = GitHub

6>Secure Server = OAuth2

7>Production Ready Endpoint = Actuator

8>Metrics UI = Admin (Server/Client)

9>Cloud Platform = AWS, Docker ,With Deploy services

##### SOA (Service Oriented Architecture):--

=>It is a Design Pattern used to create communication links between multiple services providers and Consumers (users).

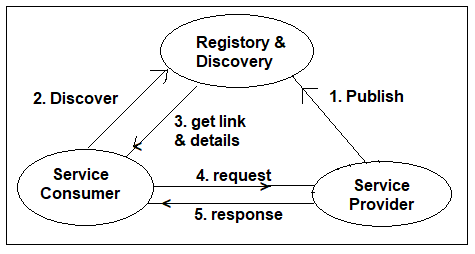
##### Components of SOA:--

a>Service Registry and Discovery [Eureka] b>Service Provider [Webservice Provider] c>Service Consumer [Webservice Client]

**Operations:--** 1>Publish 2>Discover

3>Link Details of Provider

4>Query Description (Make Http Request). 5>Access Service (Http Response).



##### Implementing MicroService Application Using Spring Cloud:--

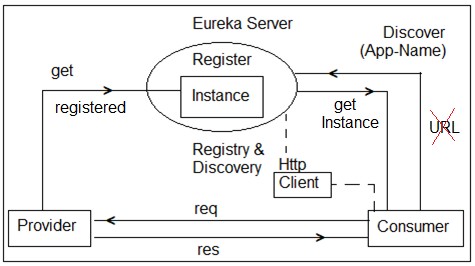
##### MicroService Design and Implementation using Spring Cloud (Netflix Eureka Registry & Discovery):--

=>Registory and Discovery server hold details of all Client (Consumer/Producer) with its serviced and Instance Id.

=>Netflix Eureka is one R & D Server.

=>Use default port no is 8761.

##### Design#1:- [Basic – No Load Balancing]



**Step #1: Create Eureka Server:--** Create one Spring Boot Starter Project with Dependencies : **Eureka Server**

##### Eureka Server Dependencies:--

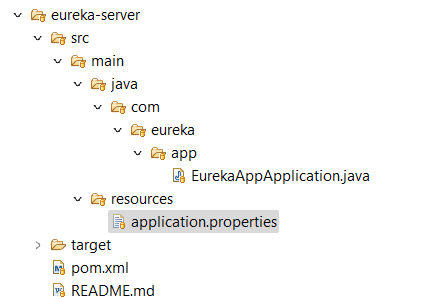
<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server </artifactId>

</dependency>

##### #3. Folder Structure of Eureka Server:--

****

**pom.xml of Eureka Server:--**

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

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

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

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>com.eureka.app</groupId>

<artifactId>eureka-app</artifactId>

<version>1.0.0</version>

</parent>

<groupId>com.eureka.app</groupId>

<artifactId>eureka-server</artifactId>

<version>1.0.0</version>

<name>eureka-server</name>

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

<properties>

<java.version>11</java.version>

<spring-cloud.version>2021.0.3-SNAPSHOT</spring-cloud.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>

</dependency>

<dependency>

<groupId>com.sun.jersey.contribs</groupId>

<artifactId>jersey-apache-client4</artifactId>

<version>1.19.4</version>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-snapshots</id>

<name>Spring Snapshots</name>

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

<releases>

<enabled>false</enabled>

</releases>

</repository>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

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

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

**Step#2:-** At Starter class level add Annotation **@EnableEurekaServer** Annotation:-- package com.eureka.app;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class EurekaAppApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaAppApplication.class, args);

}

}**Step #3:-**

In application.properties add keys server.port=8761

spring.application.name=eureka-discovery-service

eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

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

**Step #4:-** Run starter class and Enter URL http://localhost:8761 in browser

**NOTE:--** Default port no of Eureka Server is 8761.

##### Screen Short of Eureka Server Dashboard:--

* 1. **Load Balancing and API Gateway in Spring Cloud:--**

=>To handle multiple requests made by any HTTP client (or consumer) in less time, one provider should run as multiple instances and handle request in parallel. Such concept is called as Load Balancing.

=>Spring cloud has provided one Interface “Load Balance Client” which is used to define LBS (Load balancing server) Register.

=>This register holds Load factor over Instances and stores in a map format based on ServiceId. Load factor also called as current load over level over any instance.

=>Load balancing is to make request handling faster (reduce waiting time in queue).

##### Step to implement Load Balancing:--

a>Create one gateway application.

b>Register and this as multiple instances in Registry & Discovery [Eureka] server every instance with unique ID.

Ex:-- P1-58266, P2-2353424, P3-740986 etc.

c>Define consumer with any one Load Balancing Component using Rest Template

d>Define URL Mapping in Gateway Controller

Coding Steps:--

**Step#1:-** In Gateway , define detailsin pom.xml and application.yml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

**server:**

**port:** 9091 #port number

**eureka:**

**instance:**

**leaseRenewalIntervalInSeconds:** 1

**leaseExpirationDurationInSeconds:** 2

**client:**

**serviceUrl:**

**defaultZone:** http://127.0.0.1:8761/eureka/

**healthcheck:**

**enabled:** true

**lease:**

**duration:** 5

**spring:**

**application:**

**name:** gateway-service #service name

**logging:**

**level:**

**com.eureka.app:** DEBUG

**Step#2:-** In Config file annotate with LoadBalanced on RestTemaplate

package com.eureka.app.gateway.config;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.web.client.RestTemplate;

@Configuration

public class GatewayConfig {

@Bean

@LoadBalanced

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

##### Class to write GatewayConfigurations

package com.eureka.app.gateway.controller;

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

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.context.annotation.Bean;

import org.springframework.core.ParameterizedTypeReference;

import org.springframework.http.HttpMethod;

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

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

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

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

import org.springframework.web.client.RestTemplate;

@RestController

public class ServiceController {

@Autowired

RestTemplate restTemplate;

@RequestMapping(value = "/getSchoolDetails/{schoolname}", method = RequestMethod.GET)

public String getStudents(@PathVariable String schoolname) {

System.out.println("Getting School details for " + schoolname);

String response = restTemplate.exchange("http://school-service/getSchoolDetails/{schoolname}",

HttpMethod.GET, null, new ParameterizedTypeReference<String>() {

}, schoolname).getBody();

System.out.println("Response Received as " + response);

return "School Name - " + schoolname + " \n Student Details " + response;

}

@RequestMapping(value = "/getScore/{bId}", method = RequestMethod.GET)

public int getScore(@PathVariable int bId) {

System.out.println("Getting Borrower details for " + bId);

String response = restTemplate.exchange("http://scorecheck/scorecheck/score/{bId}",

HttpMethod.GET, null, new ParameterizedTypeReference<String>() {

}, bId).getBody();

System.out.println("Response Received as " + response);

return Integer.parseInt(response);

}

@RequestMapping(value = "/getScore", method = RequestMethod.GET)

public int gettinscore() {

String response = restTemplate.exchange("http://scorecheck/scorecheck/score/12",

HttpMethod.GET, null, new ParameterizedTypeReference<String>() {

}).getBody();

System.out.println("Response Received as " + response);

return Integer.parseInt(response);

}

@RequestMapping(value = "/hi", method = RequestMethod.GET)

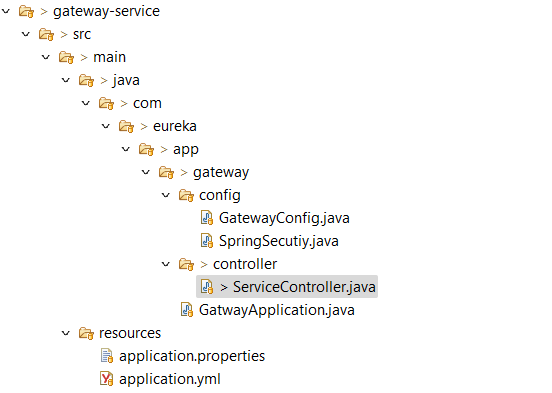
public String say() {

return "Hello";

}

}

**Folder Structure:**

****

##### Spring Security using Open Authorization (OAuth 2.x):--

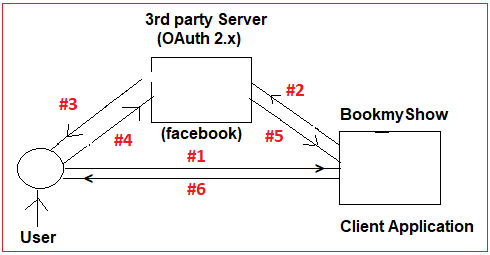
A & A = Authentication and Authorization

=>It is a process of secure one application user data. Here,

->Authentication means “IDENTITY OF USER”, like username, password, otp …etc.

=>Authorization means “ROLE OF USER” what you can do? Like ADMIN ROLE, END USER, EMPLOYEE ROLE… (It is like permissions/grant).

**OAuth 2.x:--** It is standard and framework which provides 3rd party security services to client application which are registered, on behalf of end user.



* 1. >Browser making request to client Application. 2.>Client asking permission to third party.

3.>Third party Application asking confirmation (Grant) to end user. 4.>User confirmation.

5.>Data shared from 3rd party Application to Client App. 6.>Client gives response to end user.

=>These 3rd party Applications are also called as **“Authorization and Resource Servers”.**

->Authorization and Resource Server examples are:- Google, Facebook, Githb, Twitter, Linkedin … etc.

->Example client Applications that are using OAuth2 ares BookMyshow, redbus, yatra, makemytrip, avast, zomato.. etc.

=>OAuth 2.x standard is widely used in small/medium scale/daily used, business application.

=>OAuth2 Provide SSO (Single Sign on) for multiple applications acting as a one Service.

=>\*\*\*OAuth2 may not be used for high level and Large scale applications (Ex:- Banking Creditcard, Stock Market, finance… etc). These Spring Security ORM is used mostly.

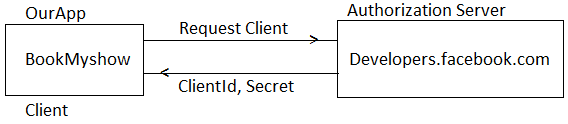
##### #1:-- Client Register with Authorization Server

Ex: BookMyShow---Register-with--->Facebook Server

=>Here, every Client Application needs to be register with **AuthorizationServers** First.

=>Client Application gets clientId and clientSecret (like Password) on Successful register at AuthorizationServer.

=>This is like one time setup between Client Application and Authorization Server. #1. Register Client Application with Authorization Server.

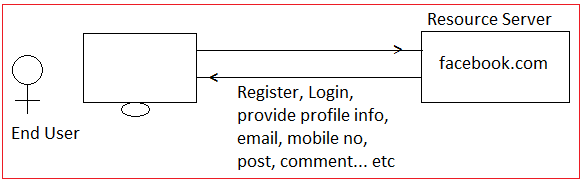


##### #2:- End User must be Register with Resource Server:--

=>User must be register and Login with 3rd party **Resource Servers** (Like facebook.com).

=>User need to provide profile information, basic data, comments, posts, photos… etc.

=>User Id and password will never be shared with Client Application. Only Users Public and General information is shared with Client Application.



##### Request Work flow of OAuth2:--

1. End user makes request using browser to client application (BookMyshow) request for “verify using 3rd party service” ex: Facebook, Google… etc.
2. Client Application will ask for Grant from end user, which confirms that access user data.
3. End user has to provide Grant (Permission) to access data.
4. Client makes request to Authorization server using ClientId, secret, user grant.
5. Auth server verifies details and goes to token Management process.

=A unique number is generated, called as Token which works for User+Client combination.

1. Now, client application makes request to resource Server using Access Token.
2. Resource server returns end user secure data to client.
3. Finally, Client App process the end user request and gives response.

##### OneTime setup for OAuth2:--

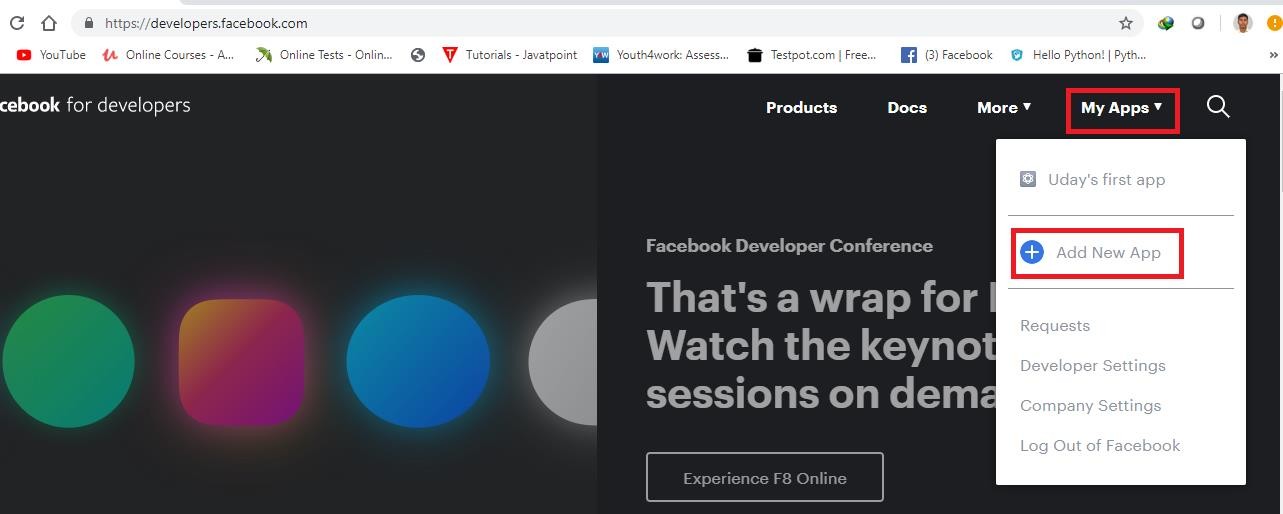
**Step#1:-** Choose any one (or more) 3rd party **“Authorization & Resource Server”**. Ex:-- facebook, Google cloud platform (GCP) Github, Linkedin, Twitter… etc.

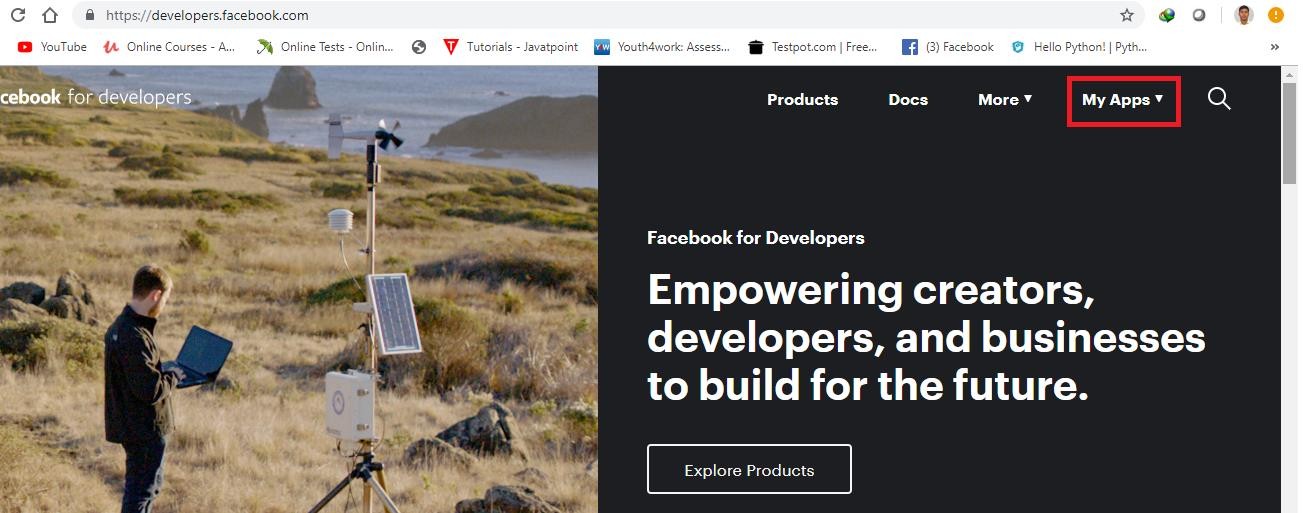
**Step#2:-** Here choosing Facebook as 3rd party server for open Authorization link is: <https://developers.facebook.com/>

**Step#3:-** Define one new (client) Application in FB Authorization server which generates ClientId (AppId) and secrete (App Secret)

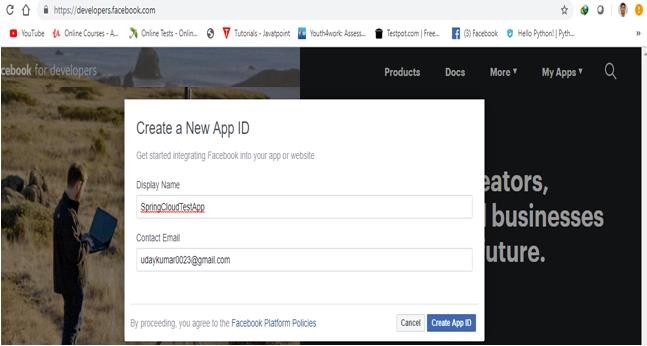
=>Goto facebook developer page

=>Click on top right corner ”My Apps”

=>Choose “Add New App”.



=>Provide Display name (ex: SpringCloudTestApp) and email id : [udaykumar0023@gmail.com](mailto:udaykumar0023@gmail.com).



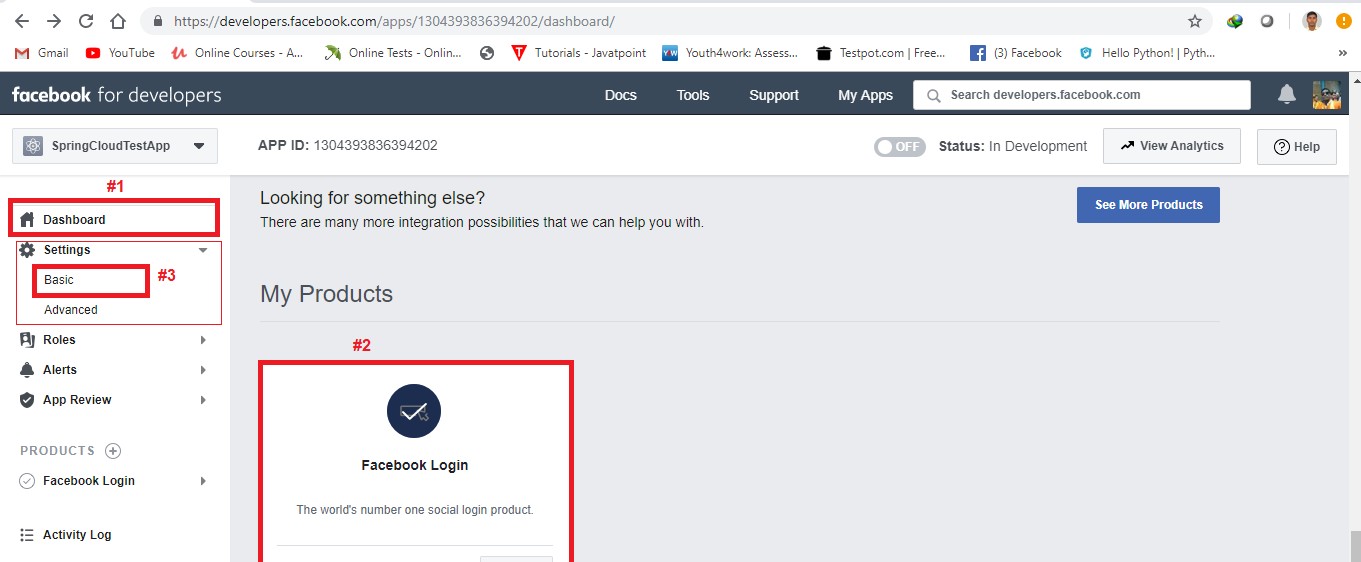
=>Click on “Create App ID”

=>Complete Security verification

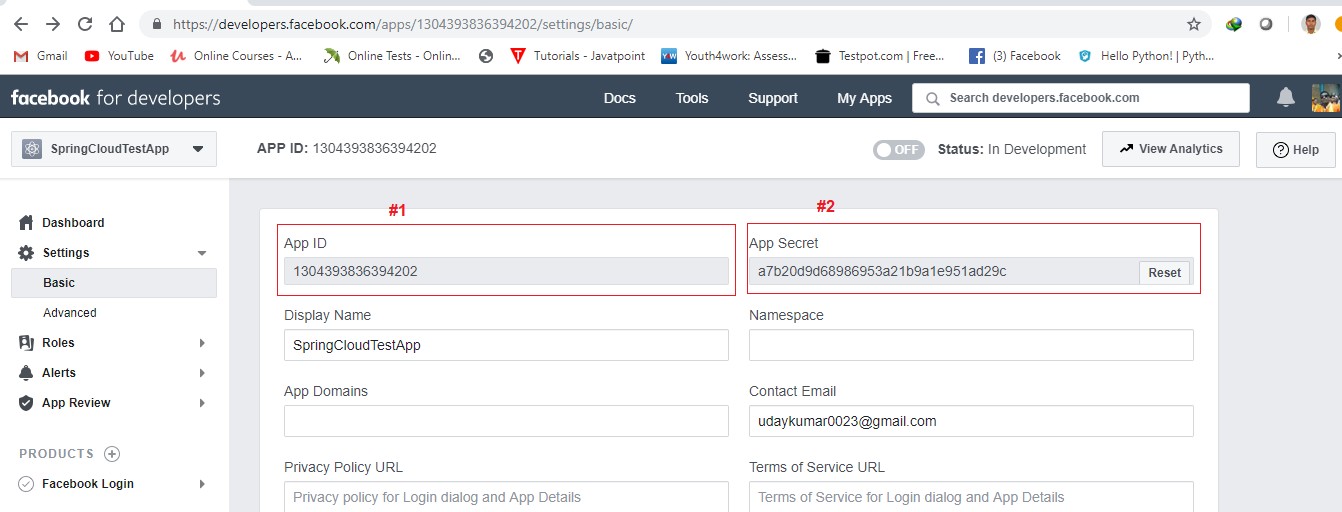
=>Click on “Dashboard”

=>Click on “facebook Login” setup button

=>Click on “Settings” >>Basic



=>Basic AppId (ClientId) and App Secret (ClientSecret).



**Step#4:-** Create one SpringBoot app with dependencies “Security” & “Cloud OAuth2”.

<dependency>

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

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

</dependency>

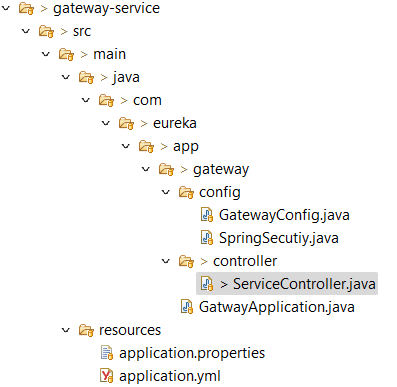
<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-oauth2</artifactId>

</dependency>

##### #23. Folder Structure of OAuth2 Application:--

****

**pom.xml:--**

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

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

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

<modelVersion>4.0.0</modelVersion>

<groupId>com.cg.eureka.school</groupId>

<artifactId>gateway-service</artifactId>

<version>1.0.0</version>

<packaging>jar</packaging>

<name>gateway-service</name>

<description>Demo project for Spring Boot eureka gateway client</description>

<!--<parent>

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

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

<version>2.7.0</version>

</parent> -->

<parent>

<groupId>com.eureka.app</groupId>

<artifactId>eureka-app</artifactId>

<version>1.0.0</version>

</parent>

<properties>

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

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

<java.version>11</java.version>

<spring-cloud.version>2021.0.2</spring-cloud.version>

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-oauth2-client</artifactId>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>spring-security-oauth2-jose</artifactId>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring-cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

<repositories>

<repository>

<id>spring-milestones</id>

<name>Spring Milestones</name>

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

<snapshots>

<enabled>false</enabled>

</snapshots>

</repository>

</repositories>

</project>

**Step#5:-** Create application.properties / application.yml file using below key value pairs.

##### application.properties:--

spring.security.oauth2.client.regisration.oauth2-client-credentials.client-id=oauth2-client-credntials

spring.security.oauth2.client.regisration.oauth2-client-credentials.client-secret=HnLz6doIrWZIsBDzrpeI9sRTyyaCBwY3

spring.security.oauth2.client.regisration.oauth2-client-credentials.scope=openid,profile,roles

spring.security.oauth2.client.regisration.oauth2-client-credentials.authorization-grant-type=client\_credentials

spring.security.oauth2.client.provider.oauth2-client-credentials.issuer-uri:http://localhost:8080/realms/oauth2-demo-realm

spring.security.oauth2.resourceserver.jwt.jwk-set-uri=http://localhost:8080/realms/oauth2-demo-realm/protocol/openid-connect/certs

**NOTE:--** AppInit is provided by Boot, no need to write.

Components Involved : ClientApp, User(Browser), Auth Server (with Token Generator and Resource Server (User data in XML/JSON).

**Step#6:-** Define SecurityConfig class (SecurityInit is not required, Handled by spring Boot only).

=>Here Authentication Details not required configuring as we are using 3rd party security services.

=>In Authorization config specify which URLs needs type **“Every One Access“**

[PermitAll].

**SecurityConfig.java:--**

@Configuration

**public** **class** SpringSecutiy **extends** WebSecurityConfigurerAdapter{

@Override

**public** **void** configure(HttpSecurity httpSecurity) **throws** Exception {

httpSecurity.authorizeRequests().anyRequest().authenticated().and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and().

cors().and().csrf().disable().oauth2ResourceServer().jwt();

}

}

Download Key Cloak using docker-compose Keycloak:

---

version: '3.7'

services:

## Keycloak Config with Mysql database

keycloak-mysql:

container\_name: keycloak-mysql

image: mysql:5.7

volumes:

- ./mysql\_keycloak\_data:/var/lib/mysql

environment:

MYSQL\_ROOT\_PASSWORD: root

MYSQL\_DATABASE: keycloak

MYSQL\_USER: keycloak

MYSQL\_PASSWORD: password

ports:

- "3307:3307"

keycloak:

container\_name: keycloak

image: quay.io/keycloak/keycloak:18.0.0

command: [ "start-dev", "--import-realm" ]

environment:

DB\_VENDOR: MYSQL

DB\_ADDR: mysql

DB\_DATABASE: keycloak

DB\_USER: keycloak

DB\_PASSWORD: password

KEYCLOAK\_ADMIN: admin

KEYCLOAK\_ADMIN\_PASSWORD: admin

ports:

- "8080:8080"

volumes:

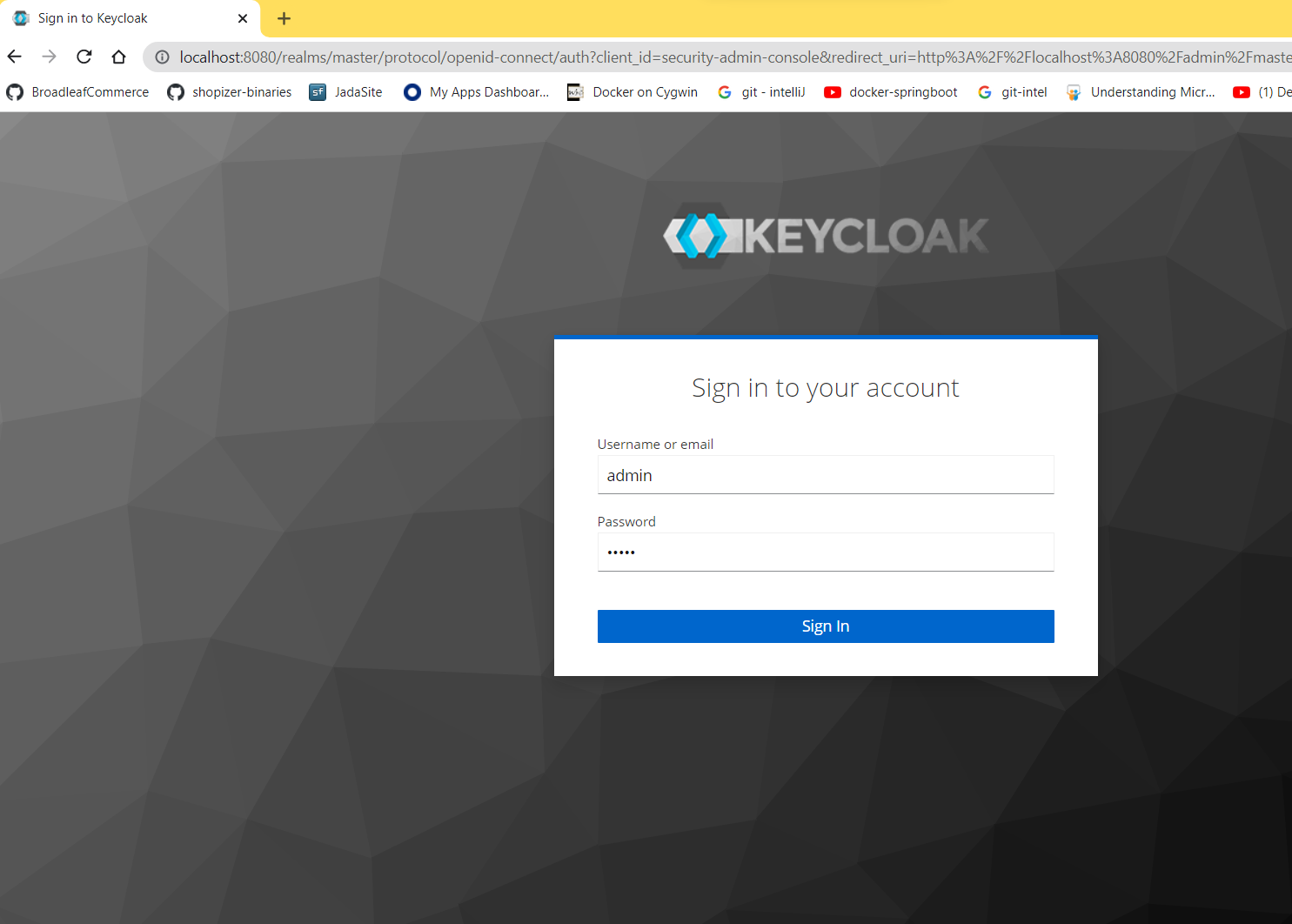
- ./realms/:/opt/keycloak/data/import/

depends\_on:

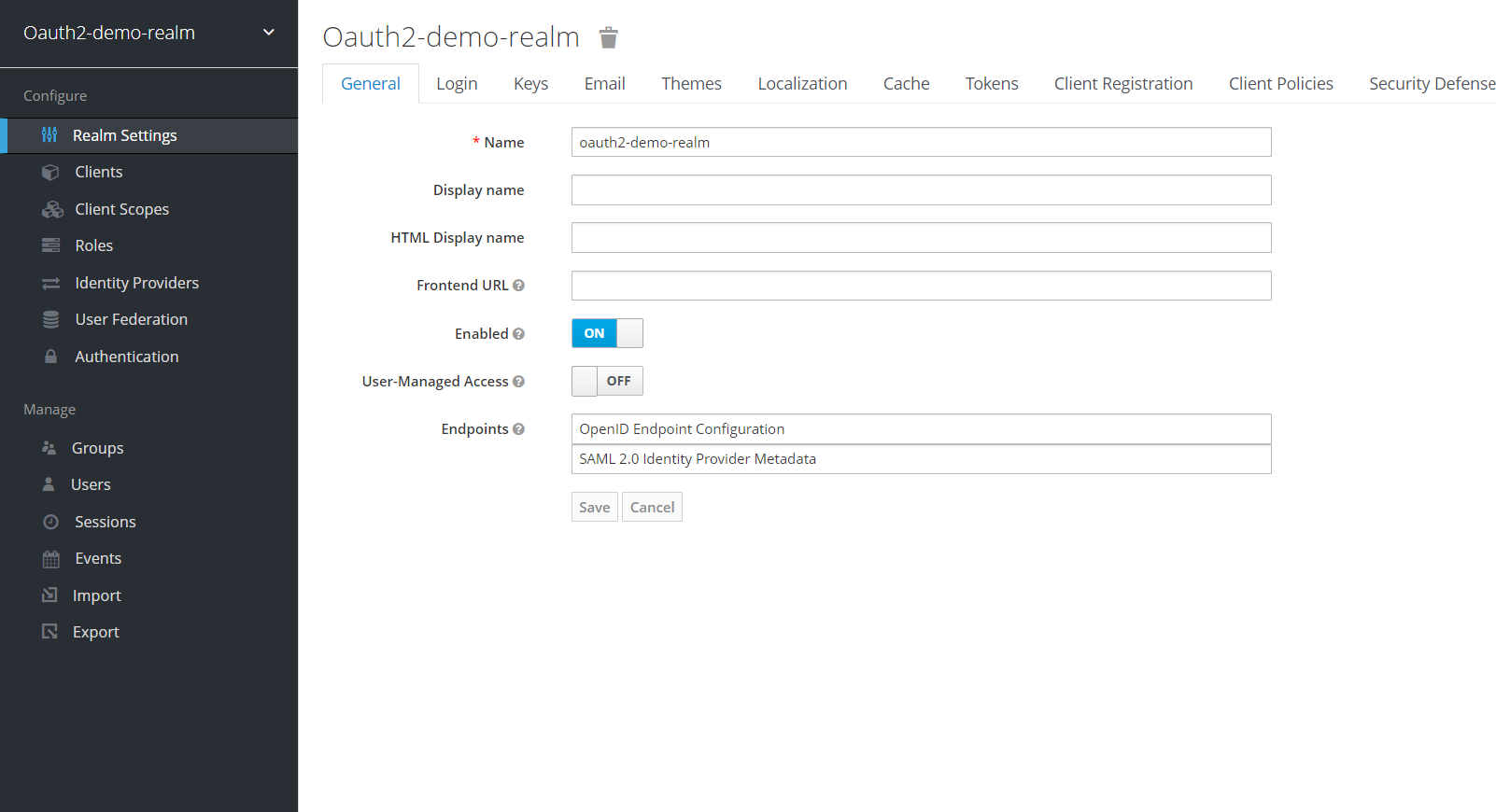
- keycloak-mysql

Run the container in Docker and login to Keycloak via port number localhost:8080

Usrername:admin and password:admin

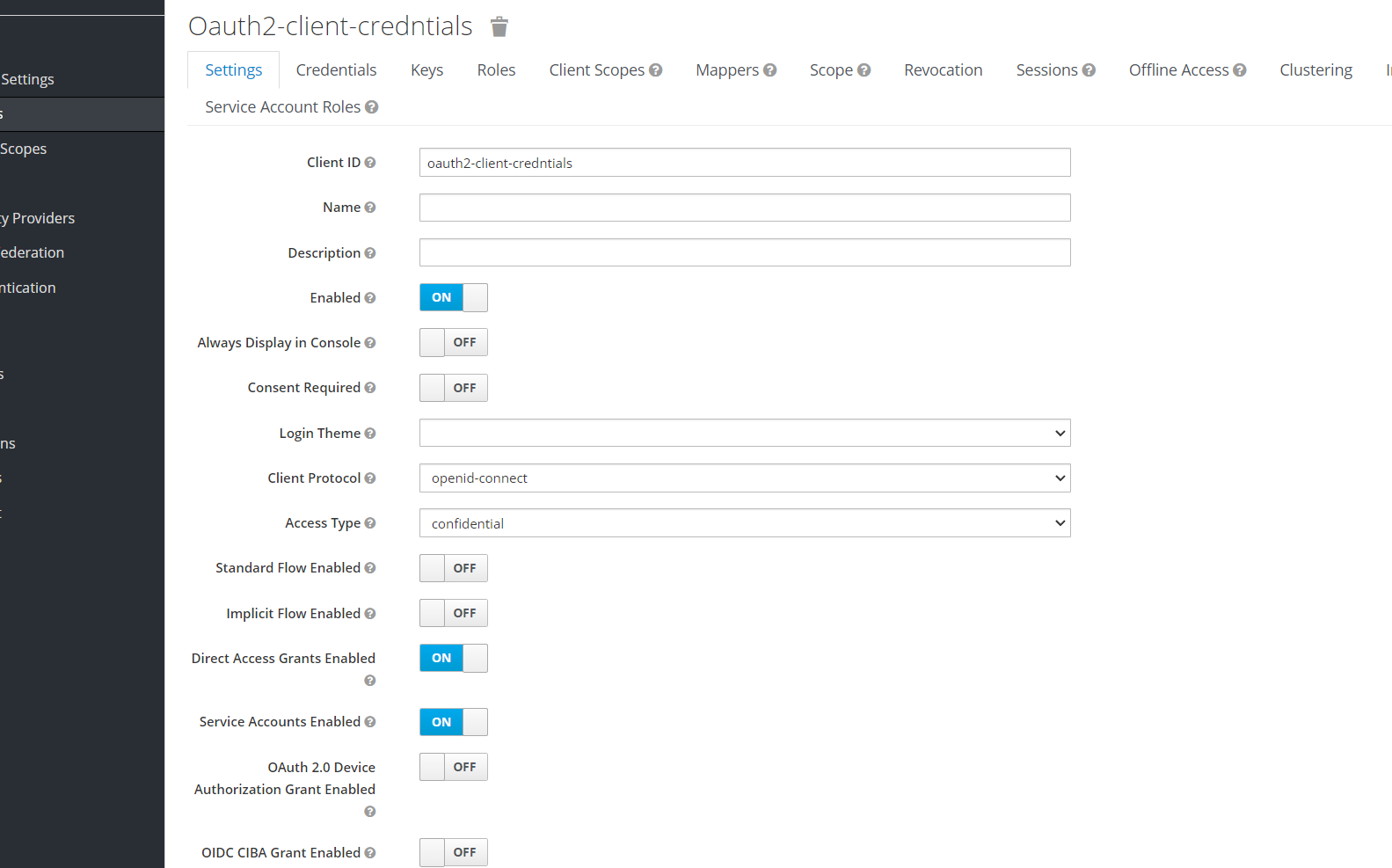


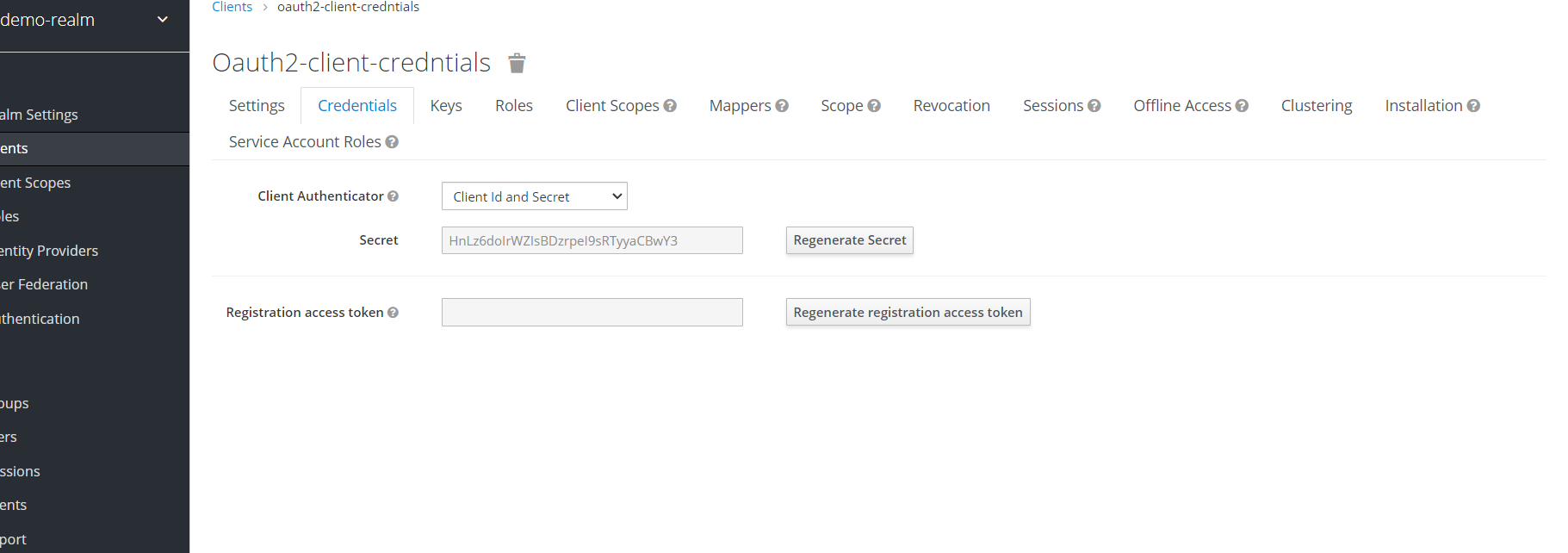
Create a new Realm to have all Security settings grouped together



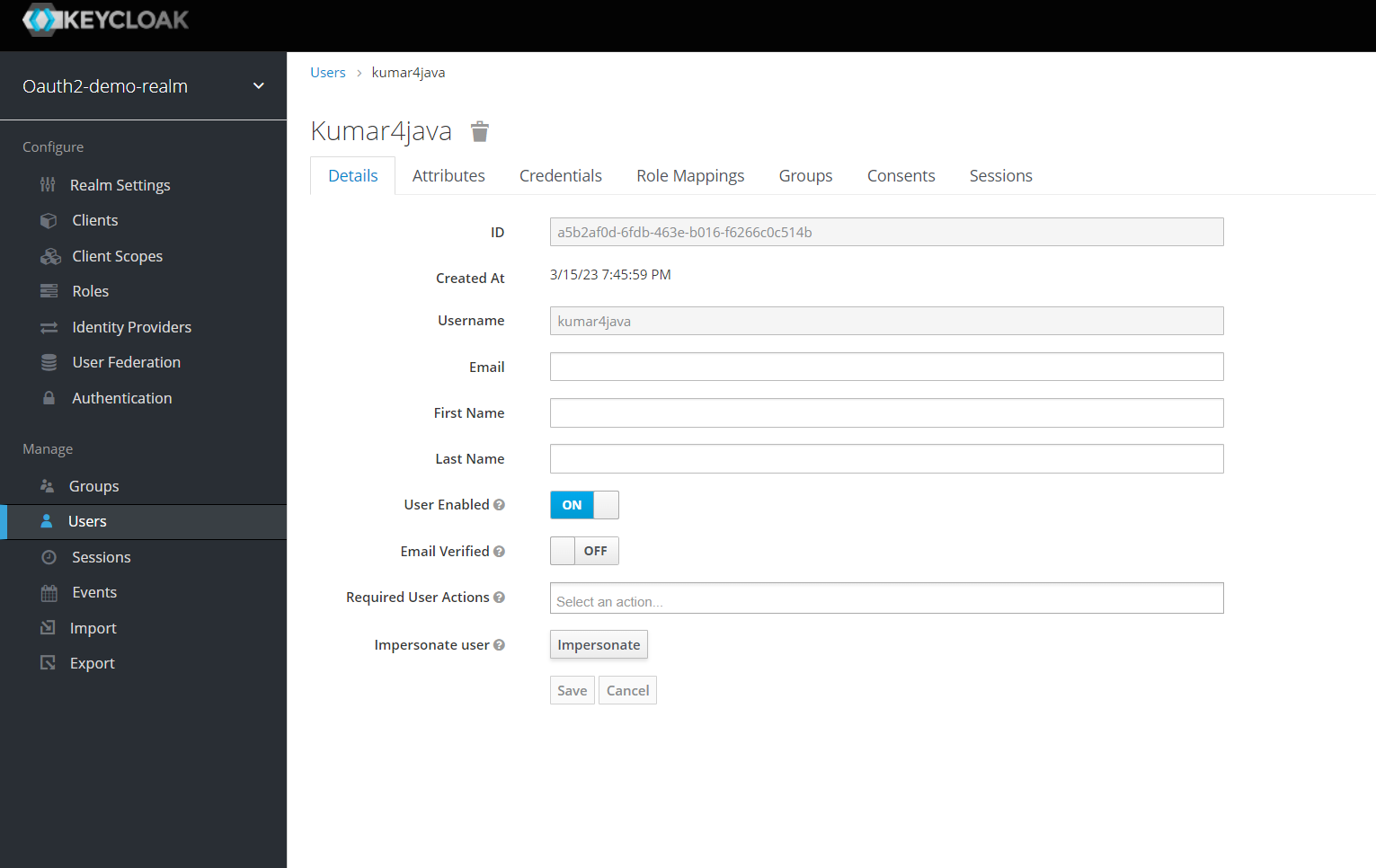
Create new Clients in Client ID, Generate Client Secret and update these details in

application.properties accordingly



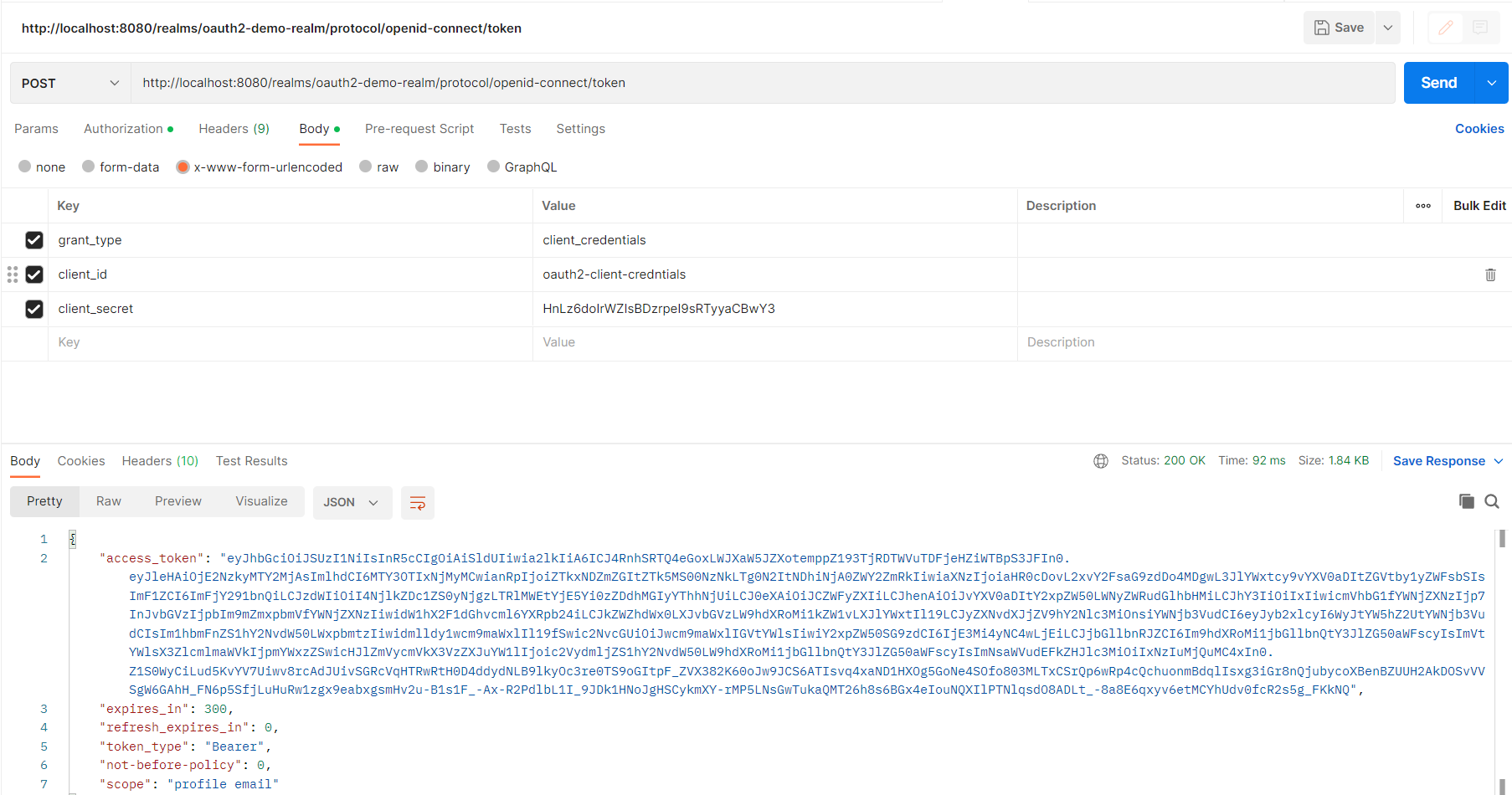


Create a new User details:



To test these in PostMan:

To get Access token from Resource Server:



To Autoconfigure these resource details via Authorization

##### 

# \*\*Deployment in AWS\*\*\*

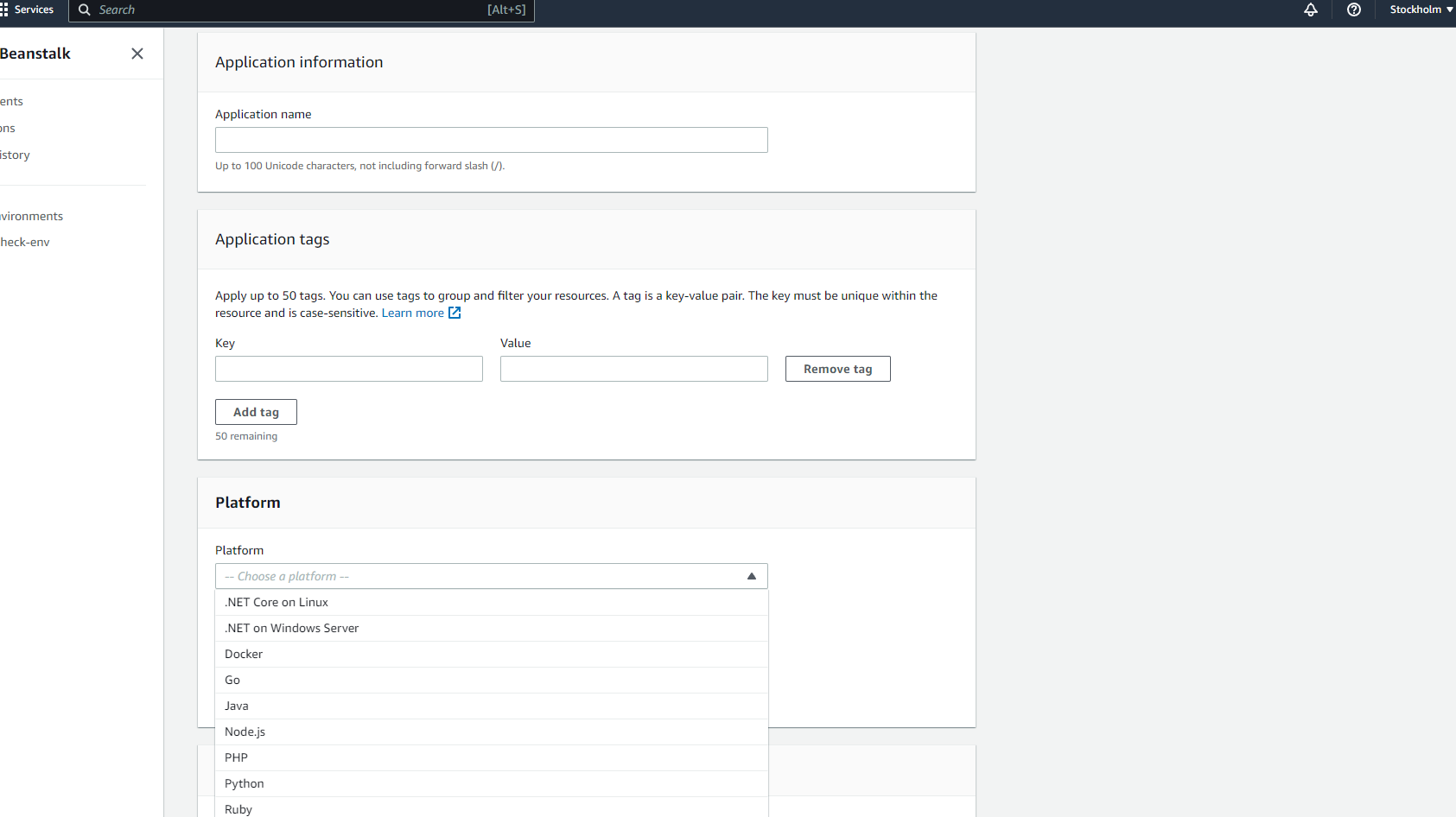
Login to AWS using credentials

Click on Compute then Elastic Beanstalk

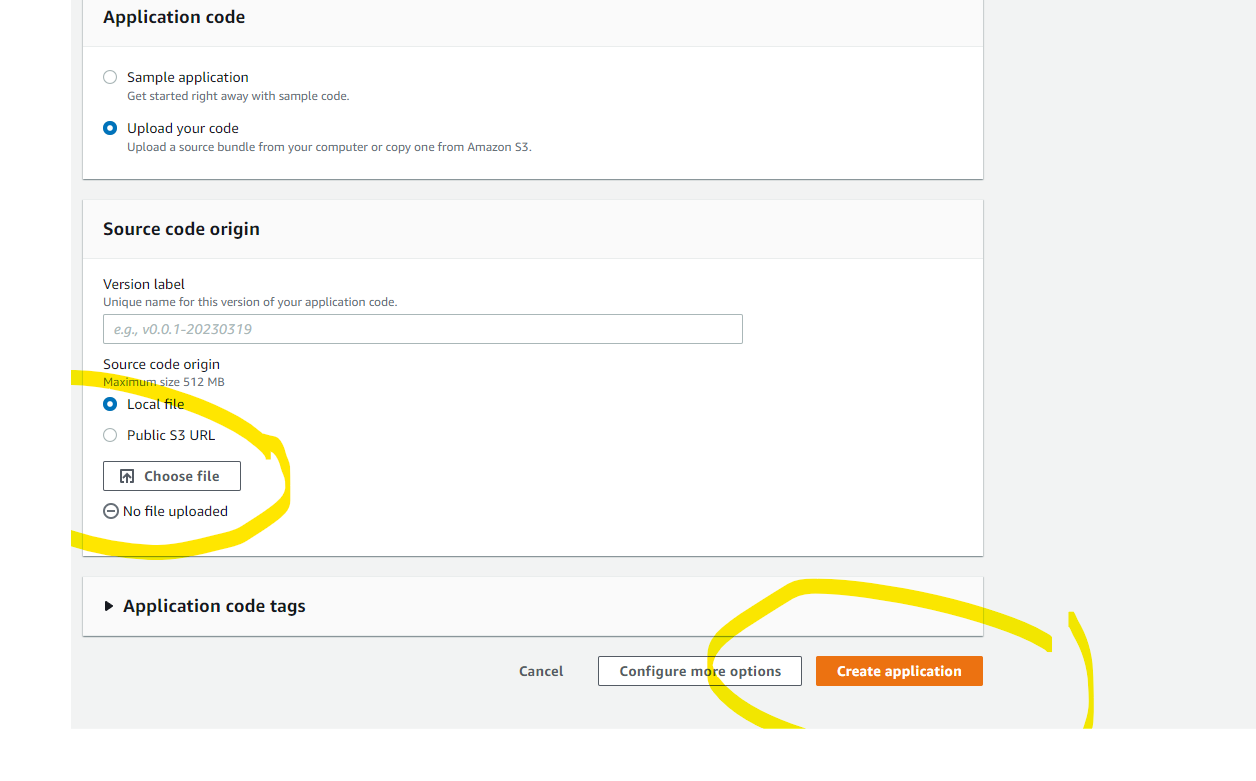
Click on Create Application

# 

Provide Application Name and Platform details



Upload the Local jar from target folder of your Application and Click on Create



After Few Minutes, It will show as Loading, then we have to

# 

We can test above link from POSTMAN/SWAGGER use above link instead of localhost:8080

# \*\*\*DOCKER\*\*\*

##### Docker:--

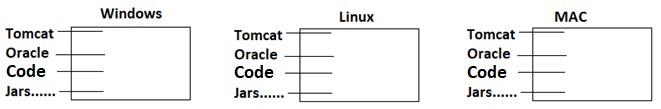
Docker is “CONTAINER SYSTEM” which includes all software’s a unit to run application, On any Platform (Windows, Linux, Mac…).

=>Docker supports running application on cloud servers also.

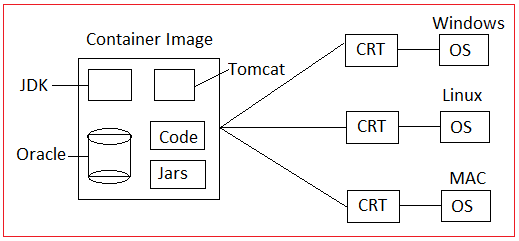
=>Docker supports follow of “CROSS-OS”. It means docker behaves a middleware between our runtime software and actual operating System.

=>Docker tool is used for Application Deployment (Running Application).

##### Before Using Container System:--

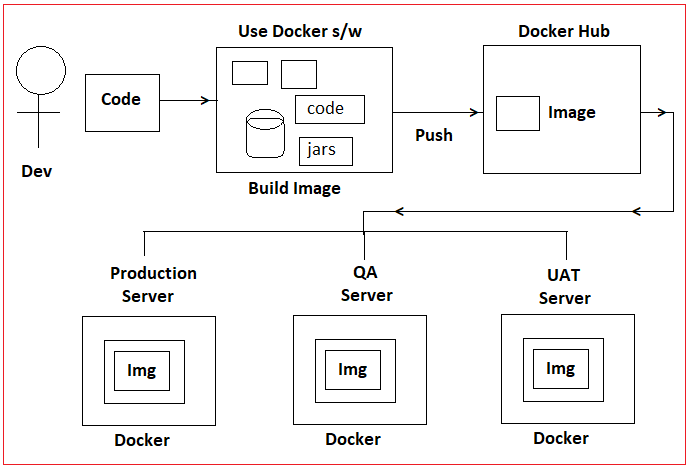


**After Container System (Docker):--**



=>CRT = Container Run time.

##### Docker Workflow:--



**Working with Docker:--**

##### #1:- DOWNLOAD DOCKER For DeskTop for Docker website.

##### #2. INSTALL DOCKER For DeskTop :--

#3:- CREATE ACCOUNT IN DOCKER HUB

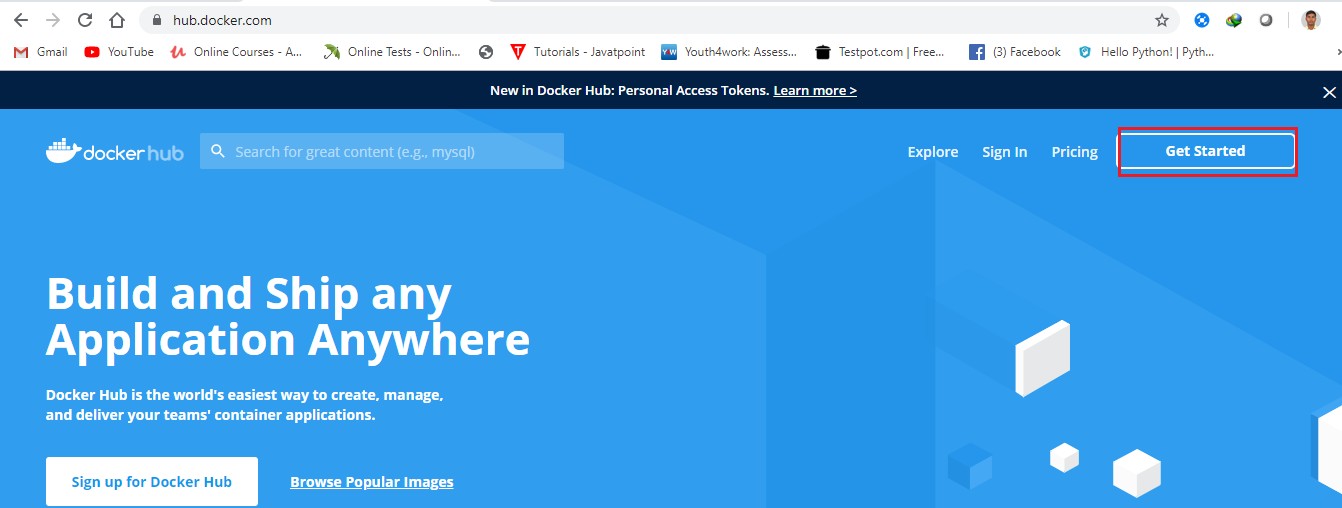
>Goto [https://hub.docker.com](https://hub.docker.com/)

>click on signup up for Docker Hub / Get Started option and register once

>enter details

>verify email

>Login here



##### =>After click on Get Started/Sign up for Docker Hub.

=>Complete the Registration process after Gmail verification.

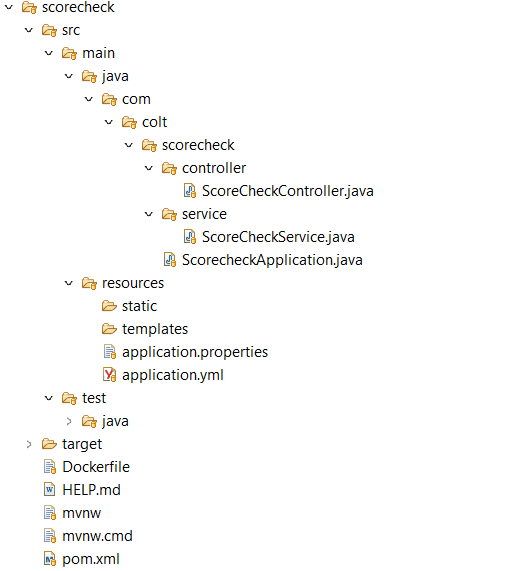


=>Login to Docker hub.

##### #4:- Dockerize score check App:--

=>Spring Boot Application with RestController

##### FOLDER STRUCTURE OF DOCKER-SPRING-BOOT APPLICATION:--

****

##### 3:- Update pom.xml with plugin

<build>

<plugins>

<plugin>

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

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

</plugin>

<plugin>

<groupId>com.spotify</groupId>

<artifactId>dockerfile-maven-plugin</artifactId>

<version>1.4.9</version>

<configuration>

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

<buildArgs>

<JAR\_FILE>target/${project.build.finalName}.jar</JAR\_FILE>

</buildArgs>

<tag>${project.version}</tag>

</configuration>

</plugin>

</plugins>

</build>

**DOCKERFILE:--** Create/Add one Dockerfile under project

FROM openjdk

VOLUME /tmp

ARG JAR\_FILE

COPY ${JAR\_FILE} app.jar

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

=>Right click on Project and click on terminal then

=>mvn clean install -DskipTests

To Build the docker images:

Mvn install dockerfile:build

##### c>check all images:--

$ docker images

$ docker image ls

##### d. run image:--

docker run -p 9090:8080 scorecheck

=>Here 9090 is docker container port no and 8080 is application port no.

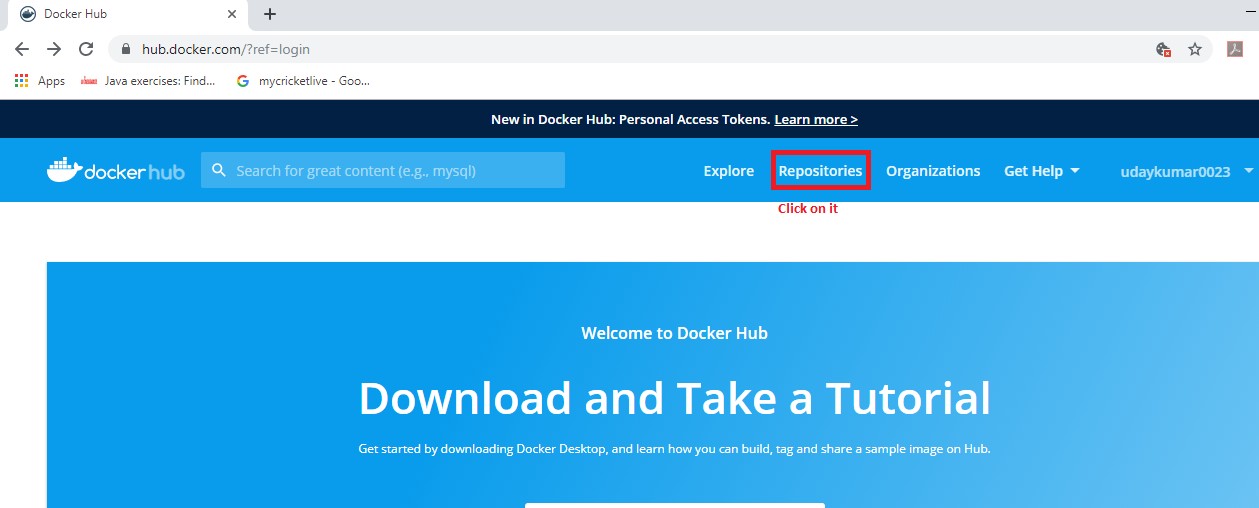
=> Goto browser and enter URL:

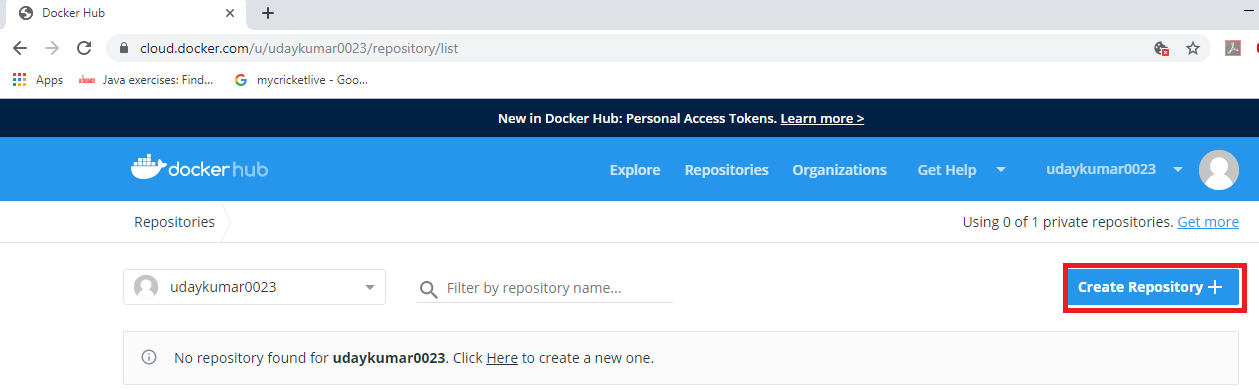
Example URL: http://192.168.99.100:9090/score

##### #7. CREATE REPOSITORY IN DOCKER HUB:--

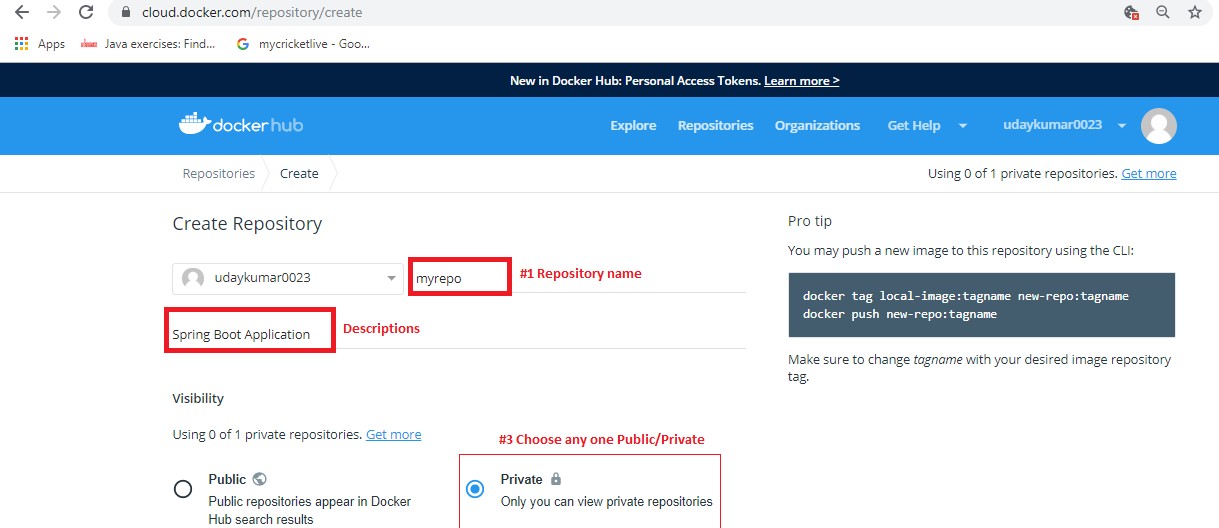
=>Login Docker hub and create one repository [Click on button “Create repository +”] Ex:--myrepo > Create

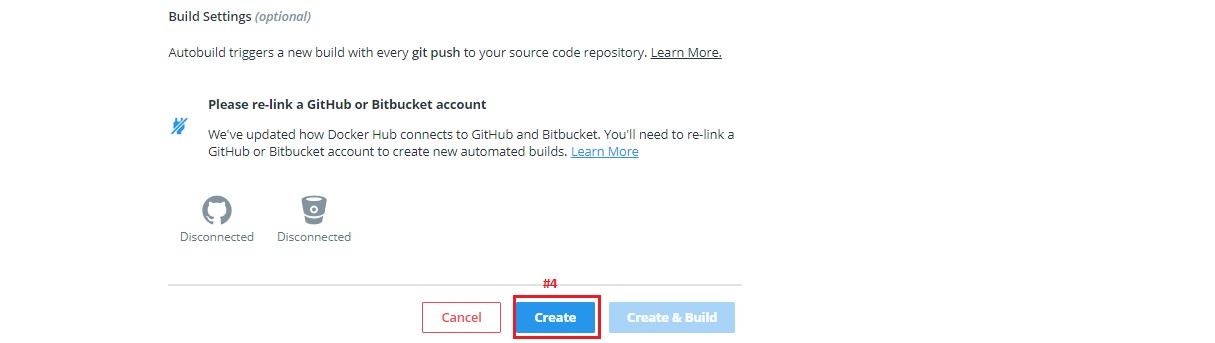
**Step#1:-** Click on Repositories



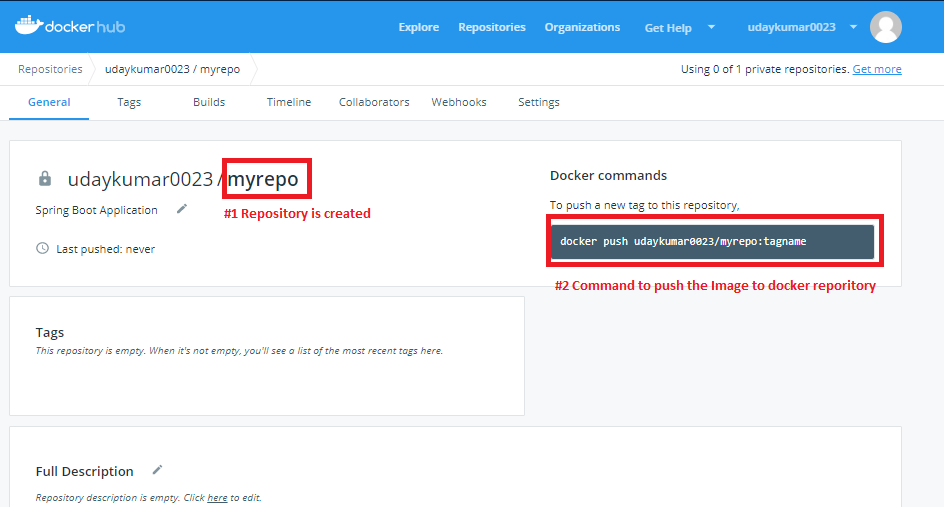
=>Click on “Create Repository +”.

=>Enter the repository name and choose any one visibility like (Public/Private).





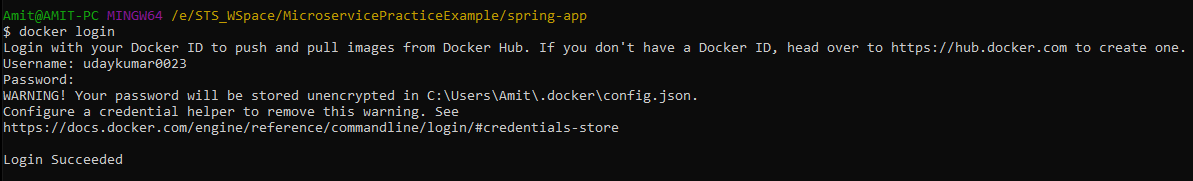
=>Final Screen of Repository.



##### #8. LOGIN TO DOCKER HUB:--

docker login

UserName : docker account username (udaykumar0023) password: docker password (Uday123)



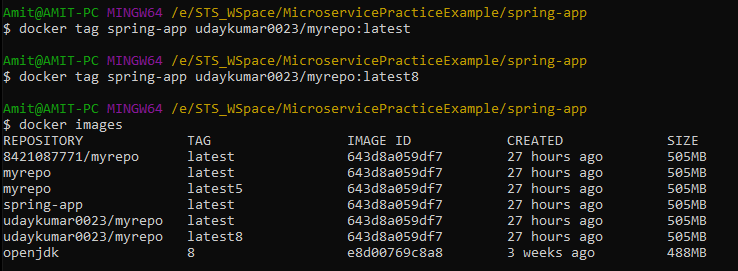
##### #9:- CREATE NEW TAG:--

=>Come to docker terminal and create one tag between docker and hub-repository

=>Format to create tag name is

=>docker tag local-image:tagname reponame:tagname

=>docker tag <imageName> <username>/<repoName>:<tagname> docker tag spring-app udaykumar0023/myrepo:latest

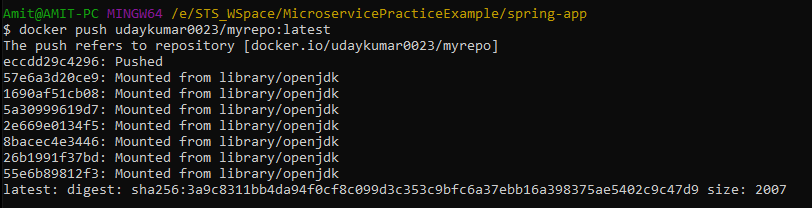


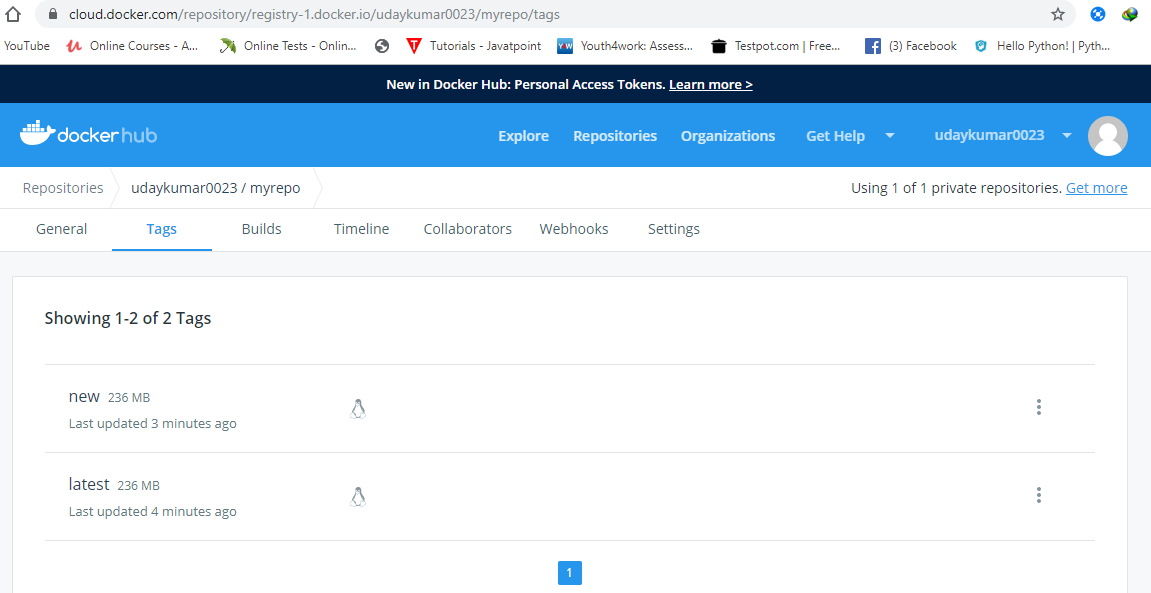
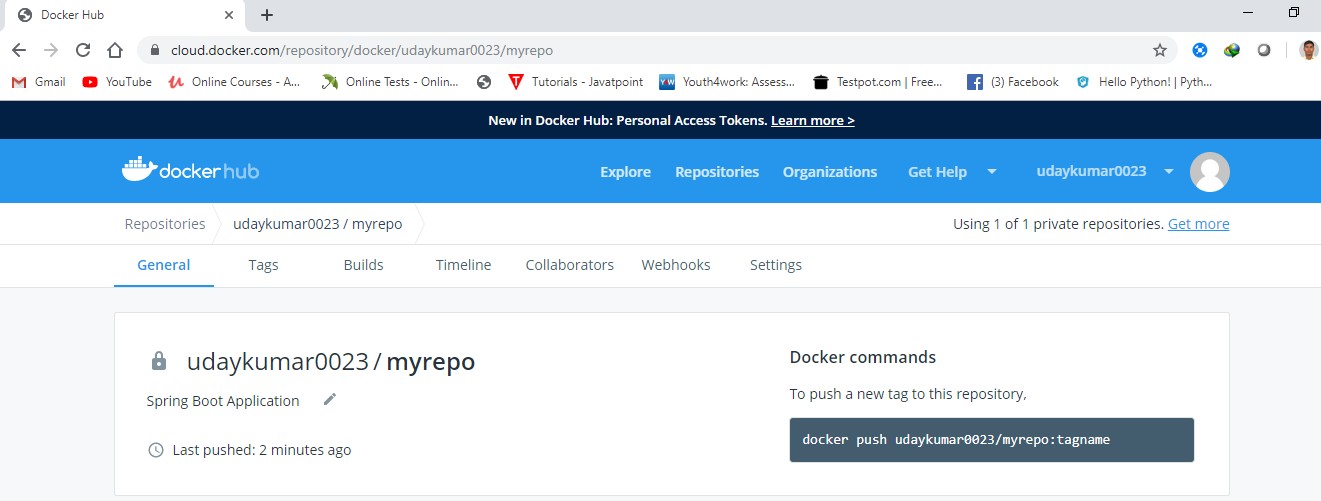
##### #10:- PUSH IMAGE INTO DOCKER HUB:--

Syntax:- docker push <username></repoName>:tagname

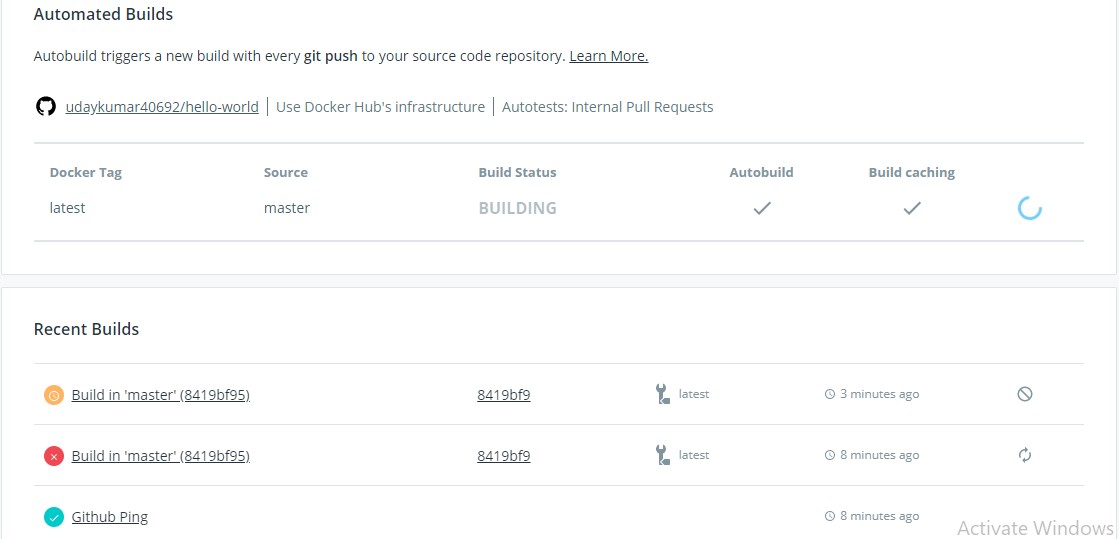
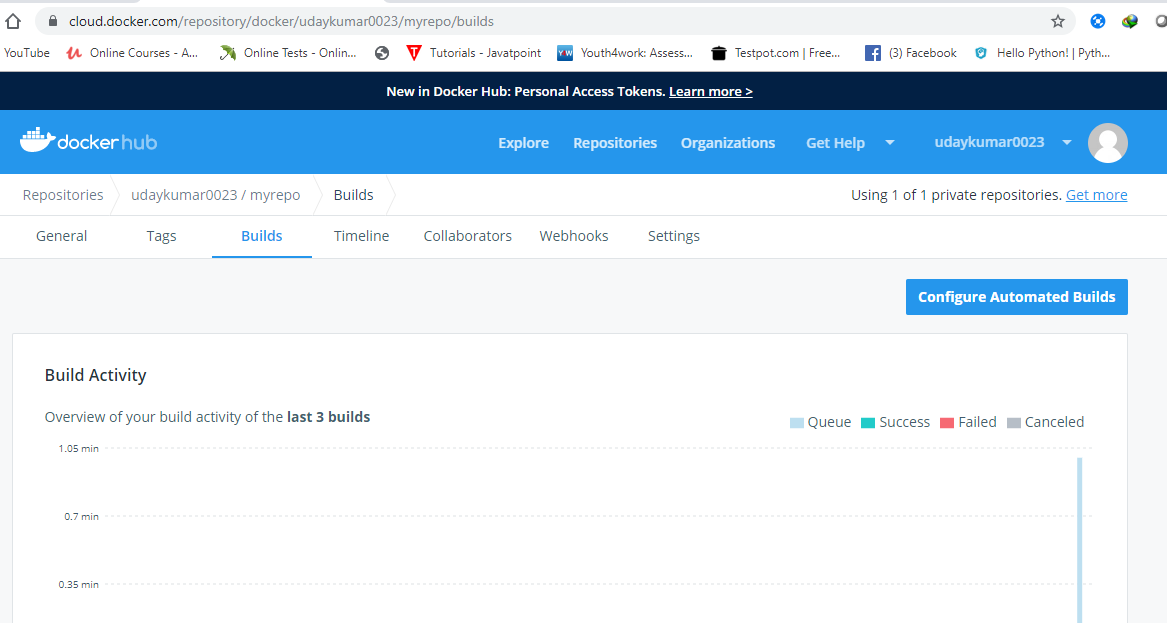
=>docker push udaykumar0023/myrepo:latest

=>Goto docker hub and refresh to see the latest update:--



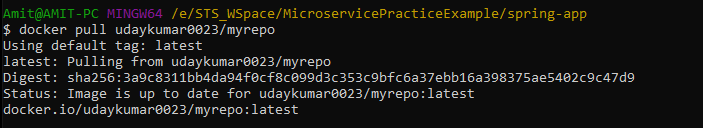


##### #13:- LINK DOCKER HUB WITH GITHUB/BIGBUKET ETC…



**#14:- PULL THE IMAGE FROM DOCKER HUB:--**

Syntax:-- docker pull <username>/myrepo docker pull udaykumar0023/myrepo



**NOTE:-** Press ctrl+c to shutdown the docker container

# \*\*\*MOCKITO\*\*\*

### Spring Boot UnitTesting using JUnit+Mocking:--

**Mock:--** It is a component which acts like Client to make request.

=>Behaves as Container.

=>Supports object creation and Destroy.

=>Uses Proxy design Pattern.

=>Supports HTTP calls, No need of using any client (**RestTemplate, HttpClient**… etc).

Mocking is implemented using

1. EasyMock
2. Mockito

=>Spring boot uses JUnit 4 + Mockito 2 Integration for UnitTesting of Applications.

=>Here Mockito supports,

* 1. HTTP Request Creation.
  2. Making HTTP Client (GET/POST…).
  3. Execute code using Proxies.
  4. GetResult and store in HTTP Response objects.

=>Here JUnit is used for

1. Writing Test cases with annotations.
2. Verify Result using assert methods (Return PASS/FALL).

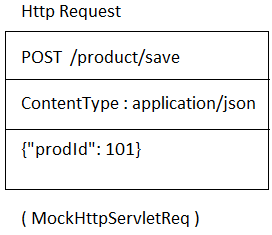
**Step#1:-** Create one Spring Starter Project and define RestController with methods and URLs added.

**Step#2:-** Define UnitTesting class (Test case) under src/test/java.

1. Autowire MockMvc (C) Object[acts as Http Client and supports container communication].
2. Call perform method to make Http Request but construct Request object at same time that returns as **MockHttpServletRequest** (use **RequestBuilder** static methods and call).

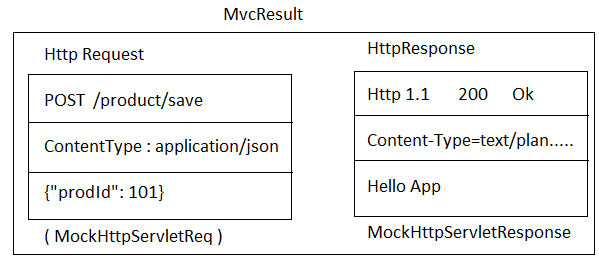
**Ex:-** mockMvc.perform(post("/product/save").header("Content-Type", "application/json").content("{\"prodId\”:101,….}")).andReturn();

1. Here andReturn() submits HttpRequest above one looks like.



1. Both Request and Response objects are stored in “MvcResult” as,
   1. MockHttpServletRequest
   2. MockHttpServletResponse.

It looks like



1. Enable this Mock and Test process using Annotations : @RunWith(SpringRunner.class)

@WebMvcTest

##### Code:-- RestController:--

package com.app.controller;

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

@RestController @RequestMapping(name="/emp") public class EmployeeRestController {

@GetMapping("/data") public String getData() { return "Hello";

}

}

**Code:- Test class:-- package** com.app;

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

**import static** org.springframework.test.web.servlet.request. MockMvcRequestBuilders.*get*;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

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

**import** org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;

**import** org.springframework.boot.test.context.SpringBootTest; **import** org.springframework.mock.web.MockHttpServletResponse; **import** org.springframework.test.context.junit4.SpringRunner; **import** org.springframework.test.web.servlet.MockMvc;

**import** org.springframework.test.web.servlet.MvcResult;

@RunWith(SpringRunner.**class**) @WebMvcTest @SpringBootTest

**public class** JUnitMockitoApplicationTests {

@Autowired

**private** MockMvc mockMvc;

@Test

**public void** testEmpData() **throws** Exception {

MvcResult result=mockMvc.perform(*get*("/emp/data")).andReturn();

//MockHttpServletRequest req =result.getRequest(); MockHttpServletResponse resp =result.getResponse(); *assertEquals*("Hi", resp.getContentAsString());

}

}

=>Right click => Run as => JUnit Test

**Spring Boot with JUnit and Mockito:--**

##### ##Testing : ReST CURD Application ##

=>To implement Unit Testing, we need to follow 4 steps. Given below,

1. Construct Http Request using **RequestBuilders**.
2. Execute Http Request using **MockMvc**
3. Store Response along with Request in **MvcResult**.
4. Use assert API (JUnit) to verify actual details with expected values.

**Step#1:-** To construct Request Object use RequestBuilder and call static method (Http Method Type).

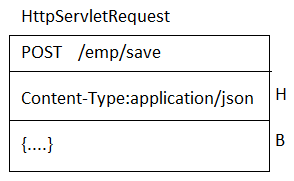
MockMvcRequestBuilders.post(…)….;

=>It returns HttpServletRequest object. MockMvcServletRequestBuilder

##### Ex#1 (GET):-

MockHttpServletRequestBuilder request = MockMvcRequestBuilders.get ("/emp/findOne/21");

##### Ex#2 (POST):--



=>Equal code of above Diagram is

MockHttpServletRequestBuilder request = MockMvcRequestBuilders

.*post*("/emp/save")

.header("Content-Type", "application/json")

//.contentType("application/json")

//contentType(MediaType.APPLICATION\_JSON)

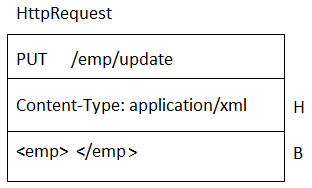
.content("{…}");

##### Ex#3 (DELETE):-

=>Equal code MockHttpServletRequestBuilder request =

MockMvcRequestBuilders.*delete*("/emp/remove/101");

##### EX#4 (PUT):-



=>Equal code

MockHttpServletRequestBuilder request = MockMvcRequestBuilders

.*post*("/emp/save")

.contentType("application/xml")

.content("<emp>…</emp");

**Step#2:-** Execute Request call using MockMvc.

MvcResult result = mockMvc.perform(request).andReturn();

**Step#3:-** Get Request/Response Object from Mvc.

MockHttpServletRequest req = result.getRequest(); MockHttpServletResponse resp = result.getResponse();

##### Project Creation step by Step:--

Step#1:- Define one CURD application using one database and RestController. Step#2:- Check all operations using POSTMAN Screen.

Step#3:- Define one test profile for properties or yml.

=>application-test.properties

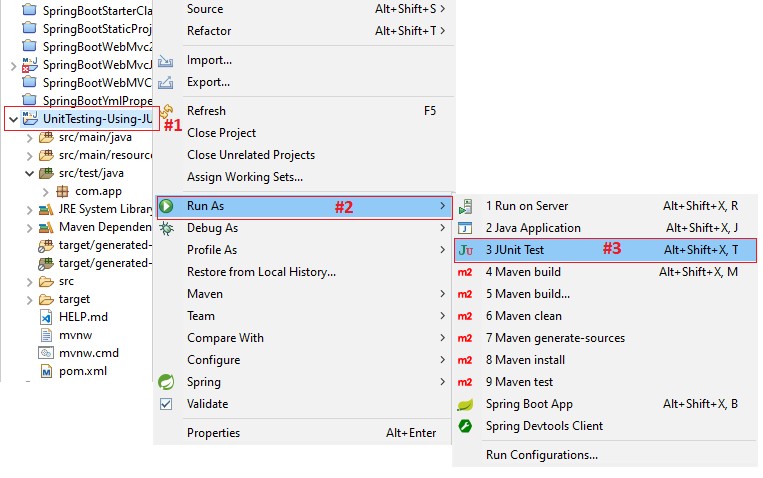
Step#4:- Define one class under src/test/java folder and apply annotations.

Step#5:- Apply below annotations over test class. @SpringBootTest(webEnvironment=WebEnvironment.MOCK) @AutoConfigureMockMvc @TestPropertySource("classpath:application-test.properties")

Step#6:- Use MockMvc dependency (autowire) in Test class. Step#7:- Define @Test methods under Test class.

Step#8:- Run Test class using JUnit Test.

=>Right click on Project > Run As > JUnit Test.



##### NOTE:--

**1>@TestPropertySource:-** It is used to load properties/yml file into UnitTest

**2>@AutoConfigureMockMvc:-** It Define all Mock Beans related to environment (Ex: Datasource, ConnectionPool, Cache…).

**3>@SpringBootTest:-** It define beans and injects them based on relations (Objects for: RestControllers, Services, Repos…etc).

**4>@WebMvcTest:-** Works only for @RestControllers without service and other dependencies.

##### 1. Folder Structure of JUnit + Mockito Testing:--

**pom.xml:--**

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

<project xmlns=*"*[*http://maven.apache.org/POM/4.0.0*](http://maven.apache.org/POM/4.0.0)*"* xmlns:xsi=*"*[*http://www.w3.org/2001/XMLSchema-instance*](http://www.w3.org/2001/XMLSchema-instance)*"* xsi:schemaLocation=*"*[*http://maven.apache.org/POM/4.0.0*](http://maven.apache.org/POM/4.0.0)

*https://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

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

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

<version>2.1.8.RELEASE</version>

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

</parent>

<groupId>com.app</groupId>

<artifactId>UnitTesting-Using-JUnitAndMockito</artifactId>

<version>1.0</version>

<name>UnitTesting-Using-JUnitAndMockito</name>

<description>Spring Boot Test Application</description>

<properties>

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

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<!-- <optional>true</optional> -->

<scope>provided</scope>

</dependency>

<dependency>

<groupId>com.oracle</groupId>

<artifactId>ojdbc6</artifactId>

<version>11.2.0</version>

</dependency>

<!-- <dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-all</artifactId>

<scope>test</scope>

</dependency> -->

<dependency>

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

<artifactId>spring-boot-devtools</artifactId>

<scope>runtime</scope>

<optional>true</optional>

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

</project>

##### Coding Step by Step:--

**Step#1: In “application.properties” & “application-test.properties” add bellow code:-**

server.port=2019 ##DataSource##

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe spring.datasource.username=system spring.datasource.password=system

#Spring Data JPA spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=create spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.Oracle10gDialect spring.jpa.properties.hibernate.format\_sql=true

spring.jpa.open-in-view=true

##### Step#2: Model class (Product.class):--

package com.app.model; import javax.persistence.Entity;

import javax.persistence.GeneratedValue; import javax.persistence.Id;

import javax.persistence.Table; import lombok.Data;

@Entity @Data @Table

public class Product {

@Id @GeneratedValue private Integer prodId;

private String prodCode; private Double prodCost; private String vendorCode;

}

##### Step#3: Repository Interface:--

package com.app.repo;

import org.springframework.data.jpa.repository.JpaRepository; import com.app.model.Product;

public interface ProductRepository extends JpaRepository<Product, Integer> { }

**Step#4: Service Interface:--** package com.app.service; import java.util.List;

import com.app.model.Product; public interface IProductService {

public Integer saveProduct(Product p);

public void deleteProduct(Integer prodId); public Product getProductById(Integer prodId); public List<Product> getAllProducts();

public boolean isProductExist(Integer id);

}

**Step#5: ServiceImpl class:--** package com.app.service.impl; import java.util.List;

import java.util.Optional;

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

import com.app.model.Product;

import com.app.repo.ProductRepository; import com.app.service.IProductService;

@Service

public class ProductServiceImpl implements IProductService{

@Autowired

private ProductRepository repo;

public Integer saveProduct(Product p) { p=repo.save(p);

Integer prodId=p.getProdId(); return prodId;

}

public void deleteProduct(Integer prodId) { repo.deleteById(prodId);

}

public Product getProductById(Integer prodId) { Optional<Product> p=repo.findById(prodId); if(p.isPresent()) {

return p.get();

}else {

}

}

return new Product();

public List<Product> getAllProducts() { List<Product> prods=repo.findAll(); return prods;

}

@Override

public boolean isProductExist(Integer id) { return repo.existsById(id);

}

}

**Step#6: RestController:--** package com.app.controller; import java.util.List;

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

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.DeleteMapping; import org.springframework.web.bind.annotation.GetMapping; import org.springframework.web.bind.annotation.PathVariable; import org.springframework.web.bind.annotation.PostMapping; import org.springframework.web.bind.annotation.PutMapping; import org.springframework.web.bind.annotation.RequestBody; import org.springframework.web.bind.annotation.RequestMapping; import org.springframework.web.bind.annotation.RestController; import com.app.model.Product;

import com.app.service.IProductService;

@RestController @RequestMapping("/product") public class EmployeeRestController {

@Autowired

private IProductService service;

##### //1. Post Method

@PostMapping("/register")

public ResponseEntity<?> saveProduct(@RequestBody Product product) { ResponseEntity<?> response=null;

try {

Integer stdId=service.saveProduct(product);

response = new ResponseEntity<String>(stdId+"-Inserted", HttpStatus.OK);

} catch (Exception e) { e.printStackTrace();

response = new responseEntity<>(HttpStatus.INTERNAL\_SERVER\_ERROR);

}

return response;

}

##### //2. Get Method

@GetMapping("/get/{id}")

public ResponseEntity<?> showOneProducts(@PathVariable Integer id) { ResponseEntity<?> response=null;

boolean exist=service.isProductExist(id); if(exist) {

Product s=service.getProductById(id);

response=new ResponseEntity<Product>(s, HttpStatus.OK);

} else {

response=new ResponseEntity<>(HttpStatus.NO\_CONTENT);

}

return response;

}

##### //3. Get Method for all data

@GetMapping("/all")

public ResponseEntity<?> showAllProducts() { ResponseEntity<?> response=null;

List<Product> Products=service.getAllProducts(); if(Products!=null && !Products.isEmpty()) {

response=new ResponseEntity<List<Product>>(Products, HttpStatus.OK);

} else {

response=new ResponseEntity<>(HttpStatus.NO\_CONTENT);

}

return response;

}

##### //4. Delete Method

@DeleteMapping("/delete/{id}")

public ResponseEntity<?> deleteProduct(@PathVariable Integer id) { ResponseEntity<?> response=null;

boolean exist=service.isProductExist(id); if(exist) {

service.deleteProduct(id);

response=new ResponseEntity<String>(id+"-Removed", HttpStatus.OK);

} else {

response=new ResponseEntity<String>("Product NOT FOUND", HttpStatus.BAD\_REQUEST);

}

return response;

}

##### //5. Edit method

@PutMapping("/edit")

public ResponseEntity<?> editProduct(@RequestBody Product product) { ResponseEntity<?> response=null;

Integer id=product.getProdId();

boolean exist=service.isProductExist(id); if(exist) {

service.saveProduct(product);

response = new ResponseEntity<String>(id+"-Updated", HttpStatus.OK);

}else {

response = new ResponseEntity<String>("Product NOT FOUND", HttpStatus.BAD\_REQUEST);

}

return response;

}

}

##### Step#7:JunitAndMockitoTests class:-- package com.app;

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

**import static** org.junit.Assert.*assertNotNull*;

**import static** org.springframework.test.web.servlet.request. MockMvcRequestBuilders.*delete*;

**import static** org.springframework.test.web.servlet.request. MockMvcRequestBuilders.*get*;

**import static** org.springframework.test.web.servlet.request. MockMvcRequestBuilders.*post*;

**import static** org.springframework.test.web.servlet.request. MockMvcRequestBuilders.*put*;

**import** org.junit.Test;

**import** org.junit.runner.RunWith;

**import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.boot.test.autoconfigure.web.servlet. AutoConfigureMockMvc;

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

**import** org.springframework.http.HttpStatus;

**import** org.springframework.http.MediaType;

**import** org.springframework.mock.web.MockHttpServletResponse; **import** org.springframework.test.context.TestPropertySource; **import** org.springframework.test.context.junit4.SpringRunner; **import** org.springframework.test.web.servlet.MockMvc;

**import** org.springframework.test.web.servlet.MvcResult;

@RunWith(SpringRunner.**class**)

//@WebMvcTest

@SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.***MOCK***) @AutoConfigureMockMvc

@TestPropertySource(locations = "classpath:application-test.properties")

**public class** UnitTestUsingJUnitAndMockitoTests {

@Autowired

**private** MockMvc mockMvc;

##### //1. Post Method case

@Test

**public void** testProductSave() **throws** Exception {

MvcResult result = mockMvc.perform(*post*("/product/register")

.contentType(MediaType.***APPLICATION\_JSON***)

.content("{\"prodCode\":\"ABCD\",\"prodCost\":88.55,\"vendorCode\": \"V11\"}"))

.andReturn();

MockHttpServletResponse resp = result.getResponse(); *assertEquals*(HttpStatus.***OK***.value(), resp.getStatus()); System.***out***.println(resp.getContentAsString()); *assertNotNull*(resp.getContentAsString());

}

##### //3. Put Method Test Case

@Test

**public void** testProductPut()**throws** Exception {

MvcResult result=mockMvc.perform(*put*("/product/edit")

.contentType(MediaType.***APPLICATION\_JSON***)

.content("{\"prodId\":1, \"prodCode\":\"ABCDE\",\"

prodCost\":38.55,\"vendorCode\": \"V14\"}")).andReturn(); MockHttpServletResponse resp=result.getResponse(); *assertEquals*(HttpStatus.***OK***.value(), resp.getStatus()); System.***out***.println(resp.getContentAsString()); *assertNotNull*(resp.getContentAsString());

}

##### //2. Get Method Test Case

@Test

**public void** testProductGet() **throws** Exception {

MvcResult result = mockMvc.perform(*get*("/product/get/4")).andReturn(); MockHttpServletResponse resp=result.getResponse(); *assertEquals*(HttpStatus.***OK***.value(), resp.getStatus()); *assertNotNull*(resp.getContentAsString());

}

##### //4. Delete Method Test Case

@Test

**public void** testProductDelete()**throws** Exception {

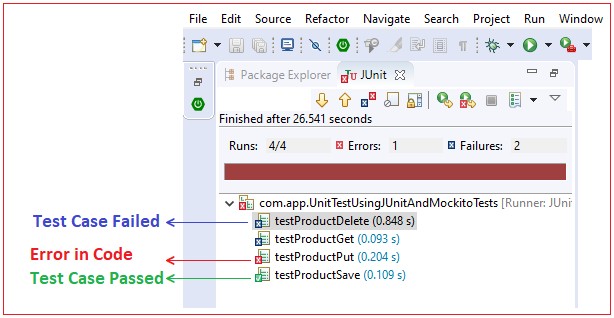
MvcResult result=mockMvc.perform(*delete*("/product/delete/5")).andReturn(); MockHttpServletResponse resp=result.getResponse(); *assertEquals*(HttpStatus.***OK***.value(), resp.getStatus()); System.***out***.println(resp.getContentAsString()); *assertNotNull*(resp.getContentAsString());

}

}

##### 1>Output SCRENN Short of JUnit:--

**2>Output Screen for all test cases.**



**NOTE:--** 1>Green Color(testProductSave) indicate Test cases passed. 2>Blue color(testProductDetele,testProductGet) indicates test cases failed.

3>Red color(testProductPut) indicate error in test class specific method code.