**SpringBoot & MicroServices**

Pre-Requisites

Core Java

Adv Java (Jdbc, Servlets, Jsp, JSTL)

Hibernate (ORM)

Spring (Core)

**SpringBoot**

Spring vs SpringBoot

SpringBoot Advantages

IOC Container

Stereotype annotations

Auto Configuration

Starter POM

Standalone application development using SpringBoot

Introduction to Spring MVC

Spring MVC Architecture

Embedded Servers in SpringBoot

Webapp development using SpringBoot

application.properties file

application.yml file

Boot application deployment in third party server

SpringBoot profiles

SpringBoot Actuators

Exception Handling in SpringBoot

Logging in SpringBoot (SLF4j, Log4j, Logback, MDC)

TheamLeaf in SpringBoot

Apache Kafka

JAX-RS (JAVA API FOR XML RESTFUL SERVICE)

From Spring 3.0v we have support to develop rest application using spring.

Spring with Rest

SOAP vs REST

Rest Architecture Principles -- Roy Fielding

XML, XSD and JAX-B

JSON, JsonSchema and Jackson

HTTP Methods

HTTP Status Codes & messages

REST API Development

REST Client Development(RestTemplate, WebClient)

Swagger

Asynchronus Communication Services (Reactive Programming)

Exception Handling in REST API (ExceptionMapper)

SOAP UI & POSTMAN

Securying REST API(BasicAuth & OAuth2.0)

SpringData

Introduction to SpringData

Introduction to SpringData Starter POM

DataSource Configuration

CRUDRepository

JpaRepository

Custom Queries execution using @Query annotation

Sorting

Pagination

MicroServices

What is Monolithic Architecture

What is Load Balancer (Cluster) - LBR

What is MicroService Architecture

Advantages of MicroServices

Service Registry(Netflix server)

Gateway Service (Zuul proxy server)

MicroServices Intercommunication (FiegnClient)

Microservice deployment in Pivotal Cloud Foundry

MicroService deployment in AWS

Ribbon (Load Balancer)

Circuit Breaker (Hystrix)

Docker (Orchestration) | Kubernetes

Microservice application monitoring(JMETER)

Tools

Maven

Log4J

Junit & PowerMock

Jenkins (CI & CD)

Duration: 45-60 days

Timings: 9:00 AM - 10:30 AM (Monday-Monday)

No class notes

Facebook Group : TEK Leads

email id : tekleads4u@gmail.com or ashok@nareshit.com

3 expereience (Boot & MicroService)

Service based company - Accenture, Deloitte, TCS, Infy, TechM..

--------------------------------------------------------------

years of experience \* 3 lakhs

3\*3=9 lakhs

Product based company- Oracle, DeShaw, Pramati, JpMorgan etcc

-----------------------------------------------------

years of experience \* 4 lakhs

3 \* 4 = 12 lakhs

Note: DS, Algos, DP, System Design are very imp

(www.geeksforgeeks.com)

**Spring**

It is an opensource application development framework

Pivotal team

Latest version of spring is 5.x (Reactive Programming-webflux)

Spring is mainely using to develop below 3 types of applications

Web applications (C 2 B)

Distributed applications (B 2 B)

MicroService applications (B 2 B)

Spring Core

Spring AOP

Spring MVC & Spring Rest

Spring Data

SpringBoot

**Spring Boot** is an open source Java-based framework used to create a Micro Service. ... **Spring Boot** contains a comprehensive infrastructure support for developing a micro service and enables you to develop enterprise-ready **applications** that you can “just run”.

Spring is framework

SpringBoot is an approach to develop spring applications with minimal or less configuration.

Spring Boot Provided Starter Pom (Simplifying maven dependencies)

Spring Boot Provided Http Embedded Servers (Tomcat , Jetty, netty)

SpringBoot works based on Opinionated Configuration

SpringBoot supports Auto Configuration

SpringBoot supports Profiles (env specific configuration)

SpringBoot supports Actuators (Production Ready Features)

SpringBoot = Spring + Embedded Servers - xml or annotation config

Dis-Advantages of SpringBoot

Migrating spring project to spring boot will be very difficult

It is recommended for scratch development project

Un-necessary dependencies will be downloaded from starter pom. Manually we need to exclude them.

Creating SpringBoot Application

1) Spring Starter Wizard in STS IDE

2) Spring initializer website ( start.spring.io )

3) CLI

4) Maven project

pom.xml

spring-boot-starter-parent (version number) -- 2.2.0

spring-boot-starter (packaging type is jar)- standalone

spring-boot-starter-test --- unit testing (junit)

Note : If we are not using any dependency then it is recommended to exclude that from build path.

Note: Project specific dependencies we should add manually in pom.xml file.

ex : jdbc-driver, lombok, kaptcha etc.....

------------------------------------

Main class in springboot application

package com.example.demo;

@SpringBootApplication

public class SpringBootApp1Application{

public static void main(String... args){

SpringApplication.run (SpringBootApp1Application.class,args);

}

}

-----------------------------------------------------------

spring-boot-starter (standalone)

spring-boot-starter-web (web)

spring-boot-starter-webflux (reactive)

18-10-2019

-> Spring Starter Wizard in STS

-> Create project from start.spring.io website

-> download project zip file

-> import project into IDE

pom.xml

-> spring-boot-starter-parent

-> spring-boot-starter (standalone) - packaging - jar

-> spring-boot-starter-test

main class

package com.nit;

@SpringBootApplication

public class SpringBootApplication{

public static void main(String... args){

ApplicationContext ac=SpringApplication.*run*(DemoApplication.class, args);

System.*out*.println( ac.getBeanDefinitionCount()); }

}

Op:27

Predefined 27 beans are created when running main class only.

com.nit.service

com.nit.controller

com.nit.dao

com.util.nit

-> Starts StopWatch

-> Which is profile is activated (to load properties)

-> Type of application (web | reactive | default)

-> Starts IOC Container

-> ClassPath Scan (to identify spring beans)

-> Bean Objects creation & dependency injection

-> Calculate time taken to start application and print it

-> Return IOC Container reference

properties file

application.properties or application.yml

presentation

web layer ==> (com.ibm.dbs.admin.controller)

business layer ==> (com.ibm.dbs.admin.service)

persistence layer ==> (com.ibm.dbs.admin.repository)

base package ===== > com.ibm.dbs

21-Oct-19

Stereotype annotations

@Component

@Service

@Repository

@Controller (C 2 B)-- Spring 3.0

@RestController (B 2 B) -- Spring 4.0

@Configuration

@Bean (Method level)

Refer link: <https://howtodoinjava.com/spring-boot-tutorials/>

22-Oct-19

Beans Colloboration

SI : Injecting dependent object into target object by calling target class exposed setter method.

public class ContactController{

private ContactService contactService;

public void setContactService(Contactservice cs){

this.contactService = cs;

}

//methods

}

<bean id="controller" class="pkg.ContactController">

<property name="contactService" ref="cs"/>

</bean>

----------------------------------------------------------------

CI: Injecting dependent bean into target objan ect by calling target bean constructor.

public class ContactController{

private ContactService contactService;

public ContactController(ContactService cs){

this.contactService = cs;

}

//methods

}

<bean id="controller" class="pkg.ContactController">

<constructor-arg name="contactService" ref="cs"/>

</bean>

----------------------------------------------------------------

@Autowired

byName

byType

constructor (internally byType mode will be used)

When we use this Annotation, internally spring uses AutowiredAnnotationBeanPostProcessor.

This annotation we can use at below 3 levels

1) field

2) setter method

3) constructor (if we have only one parameterized constructor with beans as parameters @Autowired is optional)

@Controller

public class ContactController{

@Autowired

private ContactService contactService;

}

@Primary :

If we have multiple beans qualified for Autowiring then to choose one bean we can use @Primary annotation.

The bean which contains @Primary will be considered for Autowiring.

@Qualifier :

If we want to inject dependent object based on name of the bean then we can use @Qualifier annotation.

Note: We can use @Primary & @Qualifier only in Autowiring in situation.

public String toString(){

return

this.getClass().getName()+"@"+Integer.toHexString(this.hashCode());

}

SpringApplication.run(..)

@SpringBootApplication

How IOC will start in SpringBoot

ComponentScan

Stereotype annotation

Base Package Naming Convention

Bean Colloboration (Autowiring)

24-Oct-2019

SpringBoot Web Application Creation

spring-boot-starter-web

- Spring MVC

- Restful App

- Embedded Tomcat Container

Note: When we run this SB web application it Starts IOC container by using WebApplicationContext.

---------------------------------------------------------------------------------------------

Web-Content

# Spring boot change default port of embedded server

By default, Spring boot applications start with embedded tomcat server start at **default port** 8080

## 1. Change default server port from properties file

We can do lots of wonderful things by simply making few entries in application properties file in any spring boot application. Changing server port is one of them.

#### 1.1. application.properties

|  |
| --- |
| application.properties |
| server.port=9000 |

#### 1.1. application.yml

|  |
| --- |
| application.yml |
| server:    port : 9000 |

YAML : Yet Another Markup Language

Note: Indent spaces plays very important in yml file

## 2. Change the server port programatically

[EmbeddedServletContainerCustomizer](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/context/embedded/EmbeddedServletContainerCustomizer.html) interface is used to **customize embedded tomcat configuration**. Any beans of this type will get a callback with the container factory before the container itself is started, so we can set the port, address, error pages etc.

#### Spring boot2 – WebServerFactoryCustomizer interface

Change default server port in spring boot2 applications by implementing ConfigurableWebServerFactory interface.

|  |
| --- |
| **package** com.example.demo; |
| @Component  **public** **class** ProgrameticPort **implements** WebServerFactoryCustomizer<ConfigurableServletWebServerFactory>{  @Override  **public** **void** customize(ConfigurableServletWebServerFactory factory) {  // **TODO** Auto-generated method stub  //factory.addErrorPages(errorPages);  factory.setPort(9090);    }  } |
|  |

## 3. Spring boot change default port from command line

If the application is built as [uber jar](https://howtodoinjava.com/maven/maven-shade-plugin-create-uberfat-jar-example/), we may consider this option as well. In this technique, we will pass **‘server.port’** argument during application run command.

|  |
| --- |
| $ java -jar -Dserver.port=9000 spring-boot-demo.jar |
|  |

----------------------------------------------------------------------------------------

In SpringBoot web application context-path is empty by default.

Note: We can specifiy context-path in properties file or yml file

application.properties

server.servlet.context-path=/App

application.yml

server:

servlet:

context-path: /App

With Context-Path : http://localhost:7070/App/welcome

Without Context-Path : http://localhost:7070/welcome

Note:

Context Path should start with '/' and should not end with '/'

If we have spring-boot-starter-web then by default we will get

Embedded Tomcat & Jetty container.

If we don't want to use Embedded Tomcat container we should remove that from build path.

To remove dependency in maven we can use exclusions option

Making Jetty Serve as default embedded server

1) Exclude starter-tomcat from spring-boot-starter-web

<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>

2) Add Jetty starter dependency in pom.xml

<dependency>

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

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

</dependency>

We see like this in banner: 2019-11-04 18:54:48.712 INFO 6636 --- [ main] o.s.b.web.embedded.jetty.JettyWebServer : Jetty started on port(s) 5555 (http/1.1) with context path '/'

Deploying SpringBoot Webapplication in External Tomcat server

1) Specify embedded server scope as 'provided' in pom.xml

<dependency>

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

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

<scope>provided</scope>

</dependency>

2) Add <start-class> in <properties /> section of pom.xml

<properties>

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

<start-class>com.nare.ExternalTomCatServerApplication</start-class>

</properties>

and next add tomcat to build path

**Http Status and Codes:**

**1.1XX: Information**

**2.2XX: Success**

**3.3XX: Redirect**

**4: 4XX: Client Side**

**5: 5XX: Server Side**

**1.405** method not allowed **: HttpRequestMethodNotSupportedException**

- If we give request mismatch then get this status

Actual request is Post we send get error will get Like this

"error": "Method Not Allowed",

"message": "Request method 'GET' not supported",

Ex : Actual Request is Get, if we provide Post then we get this status code

**2. 415 Unsupported Media Type: HttpMediaTypeNotSupportedException**

- if we provide @GetMapping(value="/get",consumes="application/json") is invalid media Type ’ Get’ is used the purpose of getting data form server or provider.

We can use this GetMapping(value="/get",produces="application/json")

-If You notgive Content Type in post request is wrong it will come

**3. 406 Not Acceptable :**

**-** Actually provider sending “ json data ” client is set header accept is xml then 406 status will shown.

**4.200 OK :**

Success Message

**5.404 Not Found :**

Success Requested url is not correct.

Url Is case Sensitive.

**Rest API**

1) SOAP vs REST

2) Rest Architecture Principles

3) RestController

4) Http Annotations

- @GetMapping

- @PostMapping

- @PutMapping

- @DeleteMapping

5) @RequestBody & @ResponseBody

6) @RequestParam & @PathParam

7) Consumes

8) Produces

9) Content-Type & Accept headers

10) POSTMAN

11) Basic Authentication

12) XML & JAX-B

13) JSON & Jackson

14) Swagger

15) Exception Handling

16) Rest Client Development (RestTemplate & WebClient)

SOAP Webservices

Provider : Application which is providing services

Consumer : Application which is accessing services

WSDL : Webservice description language (xml)

SOAP : Simple Object Access Protocol (xml)

SOAP UI : Provider functionality testing tool

Note: Webservices is the first technology which supported for Distributed application development with Intereoperability.

Roy fielding identified some loop holes available in SOAP Based Services. They are below

1) Always we should exchange data in soap xml format only

(Real intereoperability is not available)

2) Everybody can't access soap webservices

(Real adoptability is not available)

To avoid problems with SOAP development Roy Fielding Provided Rest Architecture principles

1) Unique Addressbility

2) Uniform Constraint Interfaces

3) Message Oriented Representation

4) Communication Stateless

5) HATEOS

JAX-RS API (java api for xml - Restful services)

- Jersey (Sun)

- Rest Easy (Jboss)

Spring also supporting for Restful services development from Spring 3.0 version onwards

3.0 =====> @Controller + @ResponseBody

4.0 ===> @RestController

Unique Addressability

Every distributed component should bind to unique address.

Note: Class level & method level address mapping.

Uniform Constraint interfaces

To achieve easy adoptability we should bind our rest controller method to HTTP methods

GET method ----> @GetMapping

POST method ---> @PostMapping

PUT method ----> @PutMapping

DELETE method ---> @DeleteMapping

Message Oriented Representation

Rest api supports for multiple data formats

Ex: xml, json, yml, text, html etc....

Communication Stateless

In Rest api communication, every client request will be considered as new request. Nothing will be stored at server.

HATEOS

Hypermedia as an engine for application state

Working with JSON

JSON - Java Script Object Notation

JSON is platform independent & language independent

JSON is light weight when compared with XML

JSON represents data in key-value format

To work with JSON data in Java applications we can use below apis

1) Jackson

2) Gson

Jackson API

ObjectMapper mapper = new ObjectMapper();

//converting java obj to json data

String personJson = mapper.writeValueAsString(personObj);

//converting json data to java obj

Person p = mapper.readValue(personJson,Person.class);

Gson API

Gson gson = new Gson();

//convert java obj to json

String personJson = gson.toJson(personObj);

//convert json to java object

Person p = gson.fromJson(personJson,Person.class);

------------------------------------------------------------------------------------------------------------

**RestController with Json Data**

@RestController ---> Class will become distributed component

@GetMapping --> To bind method to HTTP GET request

GET : If server wants to send data to client then use GET request method (client will send GET request to get the data)

- GET request will not contain body

@PostMapping --> To bind method to HTTP Post request

POST : If server wants to recieve the data to create new record then use POST request method (Client will send data in POST request body).

- POST request will contain body

**Consumes & Produces**

Consumes represents in which format rest api method can recieve the data

Produces represents in which format rest api method can provide the response

Note: one method can support for multiple consume and produces formats.

GET request with Produces syntax

@GetMapping(value="/getCustomer",produces={"application/json"})

public Customer getCustomer(){

//logic

return customerObj;

}

Ex:

**package** com.example.demo;

@Component

**public** **class** Product {

**private** **int** id;

**private** String pname;

**private** **double** d;

//setter and to string methods

}

@RestController

**public** **class** ProducerApplication {

@GetMapping(value="/get",produces= {"application/json"})

**public** Product getDetails() {

Product p=**new** Product();

p.setId(8);

p.setPname("tv");

p.setD(50000.5);

System.***out***.println(p);

**return** p;

}

}

Op: {

"id": 8,

"pname": "tv",

"d": 50000.5

}

Note: When client sending GET request to above method he should send

Accept header.

syntax : Accept=application/json

Note: Accept header represents client expecting response format from server.

POST Request method with JSON

@PostMapping(value="/addCustomer",consumes= {“application/json","application/xml" })

public String addCustomer(Customer c){

//logic

return "Success";

}

Ex:

**package** com.example.demo;

@Component

//without these two annotations we can't convert java obj to xml rest api will give annotation to convert java obj to xml

Other wise we have api jax-B

@XmlRootElement

@XmlAccessorType(XmlAccessType.***FIELD***)

**public** **class** Product {

**private** **int** id;

**private** String pname;

**private** **double** d;

//setter and to string methods

}

@RestController

**public** **class** ProducerApplication {

@PostMapping(value="/set",consumes={"application/json","application/xml"})

**public** Product getDetailsFromClient(@RequestBody Product p) {

System.***out***.println("client send data"+p);

**return** p;

}

}

When Client is sending POST request with data in request body, he should send one header called 'Content-Type'

Content-Type header represents in which format client is sending data to server in request body.

Syntax: Content-Type=application/json

Client give data Like: :

{

"id": 8,

"pname": "tv",

"d": 50000.5

}

Syntax: Content-Type=application/xml

Client give data Like: :

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

<product>

<id>85</id>

<pname>tv</pname>

<d>50000.5</d>

</product>

If you get and provide data in Xml format to producer we must Bind the data to the Xml

Otherwise we get 406 Not Acceptable . In this converting purpose we have a api JAX-B

**JAX-B;**

-Java Architecture for XML Binding (JAXB)

- We must bind the data using rest annotation or Binding Model class to Xsd using

-1. converting xml to xsd using google

-2.in cmd promt

Xjc Model ClassName.java

Then we get Xml Binding Class we will get

-Converting java Object to Xml and vice -versa

Java Architecture for XML Binding (JAXB) is a Java standard that defines how Java objects are converted from and to XML. It uses a standard set of mappings.

JAXB defines an API for reading and writing Java objects to and from XML documents.

Refer link: <https://howtodoinjava.com/spring-boot/spring-boot-jersey-example/>

Marshalling and Un marshalling search using this link:

<https://www.javatpoint.com/jaxb-tutorial>

4/11/2019

**@QueryParam and @PathParam**

Both Query param and Path param using passing value with requestUrl

If there is a scenario to retrieve a record based on id, for example you need to get the details of the employee whose id is 15, then you can have resource with @PathParam.

GET /employee/{id}

@GetMapping(value="/add/{id}/{name}")

If there is a scenario where you need to get the details of all employees but only 10 at a time, you may use @Query param

GET /employee?start=1&size=10

We can send data from anchor tag like this:

<a href=*"delete?uid=10"*>

<a href=*"delete?{10}"*>

@Request Param

@RestController

**public** **class** ExController {

@GetMapping(value="/setparam")

**public** Map<Integer, Employee> setParam(@RequestParam Integer i) {

Map<Integer,Employee> m=**new** HashMap();

m.put(i,**new** Employee("narenrdra"));

System.***out***.println(m);

**return** m ;

}

url like: <http://localhost:1212/setparam?i=10>

op: {"10":{"n":"narenrdra"}}

@Query Param

@RestController

**public** **class** ExController {

@GetMapping(value="/setparam/{param}")

**public** Map<Integer, Employee> setParam(@PathVariable Integer param) {

Map<Integer,Employee> m=**new** HashMap();

m.put(param,**new** Employee("narenrdra"));

System.***out***.println(m);

**return** m ;

}

}

url like: <http://localhost:8585/setparam/55>

op:{"55":{"n":"narenrdra"}}

5/11/2019

***Exception Handling***

We can handle Exception in Two ways

*1) Local Exception Handling*

We can handle exception in only one rest controller

@ExceptionHandler(value = EmployeeNotFoundException.**class**)

Using handle the exception in controller and send response to the client

*For more information:*[*https://dzone.com/articles/spring-rest-service-exception-handling-1*](https://dzone.com/articles/spring-rest-service-exception-handling-1)

@Component

**public** **class** Employee {

**private** **int** id;

**private** String name;

}

@Service

**public** **class** EmployeeServiceImpl **implements** IEmployeeService {

@Autowired

**private** Employee e;

Map<Integer,Employee> m=**new** HashMap();

**public** String save(Integer i,String s) {

System.***out***.println("save");

**if** (!m.containsKey(i)) {

e.setId(i);

e.setName(s);

m.put(i,e);

}**else**

{

**return** "employee already added";

}

**return** " employee added succces fully";

}

@Override

**public** Employee get(Integer i) **throws** EmployeeNotFoundException {

**if**(m.containsKey(i)) {

Employee e=m.get(i);

**return** e;

} **else** {

**throw** **new** EmployeeNotFoundException("Employee not found");

}

}

@RestController

**public** **class** EmployeeController {

@Autowired

**private** IEmployeeService s;

@GetMapping(value="/add/{id}/{name}")

**public** String add(@PathVariable Integer id,@PathVariable String name){

System.***out***.println("add method");

String msg=s.save(id, name);

**return** msg;

}

@GetMapping(value="/get/{id}",produces= {"application/json"})

**public** Employee g(@PathVariable Integer id) **throws** EmployeeNotFoundException{

Employee m=s.get(id);

**return** m;

}

@ExceptionHandler(value = EmployeeNotFoundException.**class**)

**public** ResponseEntity<ErrorHanlingClass> handleCustomerNotFoundException() {

ErrorHanlingClass e=**new** ErrorHanlingClass(400,"employye id not found");

**return** **new** ResponseEntity<ErrorHanlingClass>(e,HttpStatus.***BAD\_REQUEST***);

}

*2) Global Exception Handling*

This is a centralized place to handle all the application level exceptions.

We can handle exception throughout (all controllers) application we can write Exception

Whenever exception is raised controller call appropriate method it acts as throws keyword.

@ControllerAdvice

**public** **class** GlobalExceptionController {

@ExceptionHandler(value = EmployeeNotFoundException.**class**)

**public** ResponseEntity<ErrorHanlingClass> handleCustomerNotFoundException() {

ErrorHanlingClass e=**new** ErrorHanlingClass(400,"employye id not found");

**return** **new** ResponseEntity<ErrorHanlingClass>(e,HttpStatus.***BAD\_REQUEST***);

}

}

## @ResponseEntity

ResponseEntity represents an HTTP response, including headers, body, and status. While @ResponseBody puts the return value into the body of the response, ResponseEntity also allows us to add headers and status code.

@**ControllerAdvice**

The @**ControllerAdvice** annotation was first introduced in Spring 3.2. It allows you to handle exceptions across the whole application, not just to an individual controller. You can think of it as an interceptor of exceptions thrown by methods annotated with @RequestMapping or one of the shortcuts

**Difference between** @**RestController**

**Difference between** @**RestController** and @**Controller** in Spring. ... The @**Controller** is a common annotation which is used to mark a class as Spring MVC **Controller** while @**RestController** is a special **controller** used in RESTFul web services and the equivalent of @**Controller** + @ResponseBody

**Swagger**

Sagger is plugin to develop a document to our Rest API

Swagger 2 is an open-source project used to describe and document RESTful APIs

Swagger is used to create a document to the rest API

Without document client cannot implement the consumer Application

Using Swagger we can add details of Provider contact details

Step 1:We need add dependencies topom.xml

1) swagger documentation

2) swagger ui

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.6.1</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.6.1</version>

</dependency>

Step 2: Write a swagger Configuration class

Create Swagger Configuration class using @Configuration annotation and write a method to create Docket object using @Bean annotation and enable swagger using @EnableSwagger2 annotation.

- A Springfox Docket instance provides the primary API configuration.

-@EnableSwagger2 annotation enables Swagger support in the class.

-The select() method called on the Docket bean instance returns an ApiSelectorBuilder

-which provides the apis() and paths() methods that are used to filter the controllers and methods that are being documented using String predicates.

-.apiInfo(metaData()): describes Contact details of provider

package com.nare.config;

@Configuration

@EnableSwagger2

public class SwaggerConfig {

@Bean

public Docket productApi() {

return new Docket(DocumentationType.SWAGGER\_2)

.select()

.apis(RequestHandlerSelectors.basePackage("com.nare.controller"))

.paths(PathSelectors.any())

.build()

.apiInfo(metaData());

}

private ApiInfo metaData() {

ApiInfo apiInfo = new ApiInfo(

"Spring Boot REST API ",

"Spring Boot REST API for Online Store",

"1.0",

"Terms of service",

new Contact("Ashok ", "https://dzone.com/articles/spring-boot-restful-api-documentation-with-swagger", "john@springfrmework.guru"),

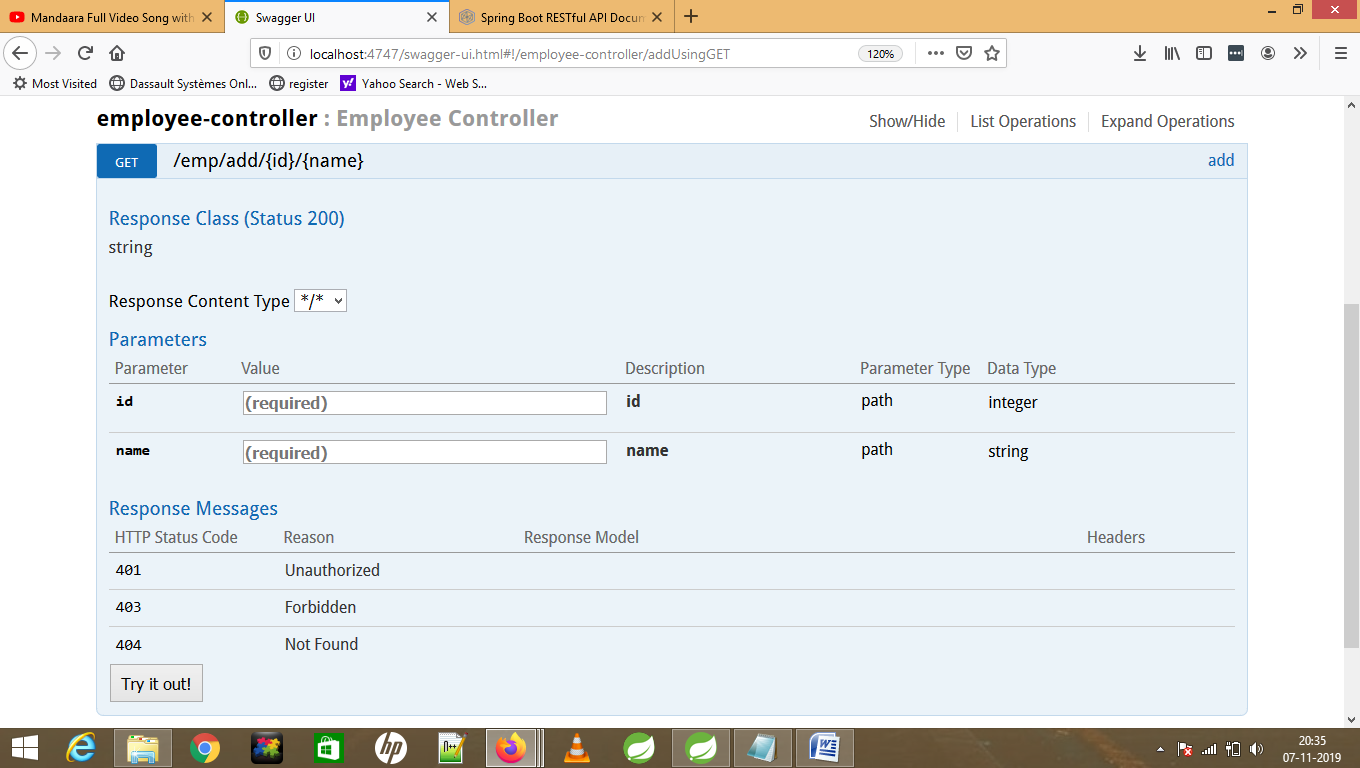
"Apache License Version 2.0",

"https://www.apache.org/licenses/LICENSE-2.0");

return apiInfo;

}

}



3) Use Swagger annotations to provide more details about Rest API

@Api: APi is used to describe our RestController api

@Api(value="employee details",~~description~~="this is sending data to employee controller ")

@ApiOperation : Annotation to describe the endpoint and its response type

@ApiOperation(value="this is used to add the details of employee")

@ApiParam: describe parameters of methods

@ApiModel, @ApiModelProperty: ApiModelProperty annotation to describe the properties of the model

@ApiModelProperty(notes="id is emp integer")

@ApiResponses, @ApiResponse: Swagger 2 also allows overriding the default response messages of HTTP methods. You can use the @ApiResponse annotation to document other responses

**package** com.nare.controller;

@RestController

@RequestMapping("/emp")

@Api(value="employee details",~~description~~="this is sendind data to employee controller ")

**public** **class** EmployeeController {

@Autowired

**private** IEmployeeService s;

@ApiResponses(value = {

@ApiResponse(code = 200, message = "Successfully retrieved added"),

@ApiResponse(code = 401, message = "You are not authorized to view the resource"),

@ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),

@ApiResponse(code = 404, message = "The resource you were trying to reach is not found")

}

)

@ApiOperation(value="this is used to add the details of employee")

@GetMapping(value="/add/{id}/{name}")

@ApiParam("send id and name")

**public** String add(@PathVariable Integer id,@PathVariable String name){

System.***out***.println("add method");

String msg=s.save(id, name);

**return** msg;

}

@ApiOperation(value="this is used toget the details of employee")

@GetMapping(value="/get/{id}",produces= {"application/json"})

**public** Employee g(@PathVariable Integer id) **throws** EmployeeNotFoundException{

Employee m=s.get(id);

**return** m;

}

@ExceptionHandler(value = EmployeeNotFoundException.**class**)

**public** ResponseEntity<ErrorHanlingClass> handleCustomerNotFoundException() {

ErrorHanlingClass e=**new** ErrorHanlingClass(400,"employye id not found");

**return** **new** ResponseEntity<ErrorHanlingClass>(e,HttpStatus.***BAD\_REQUEST***);

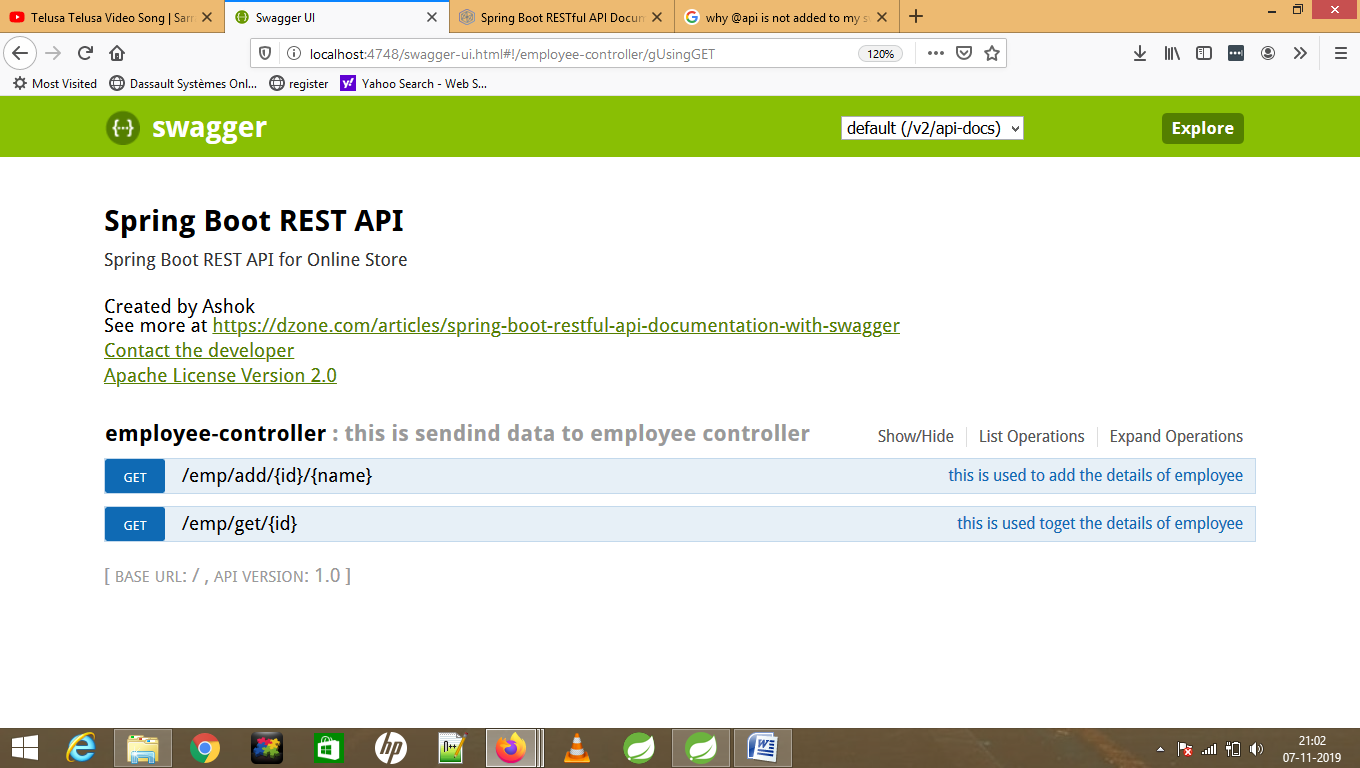
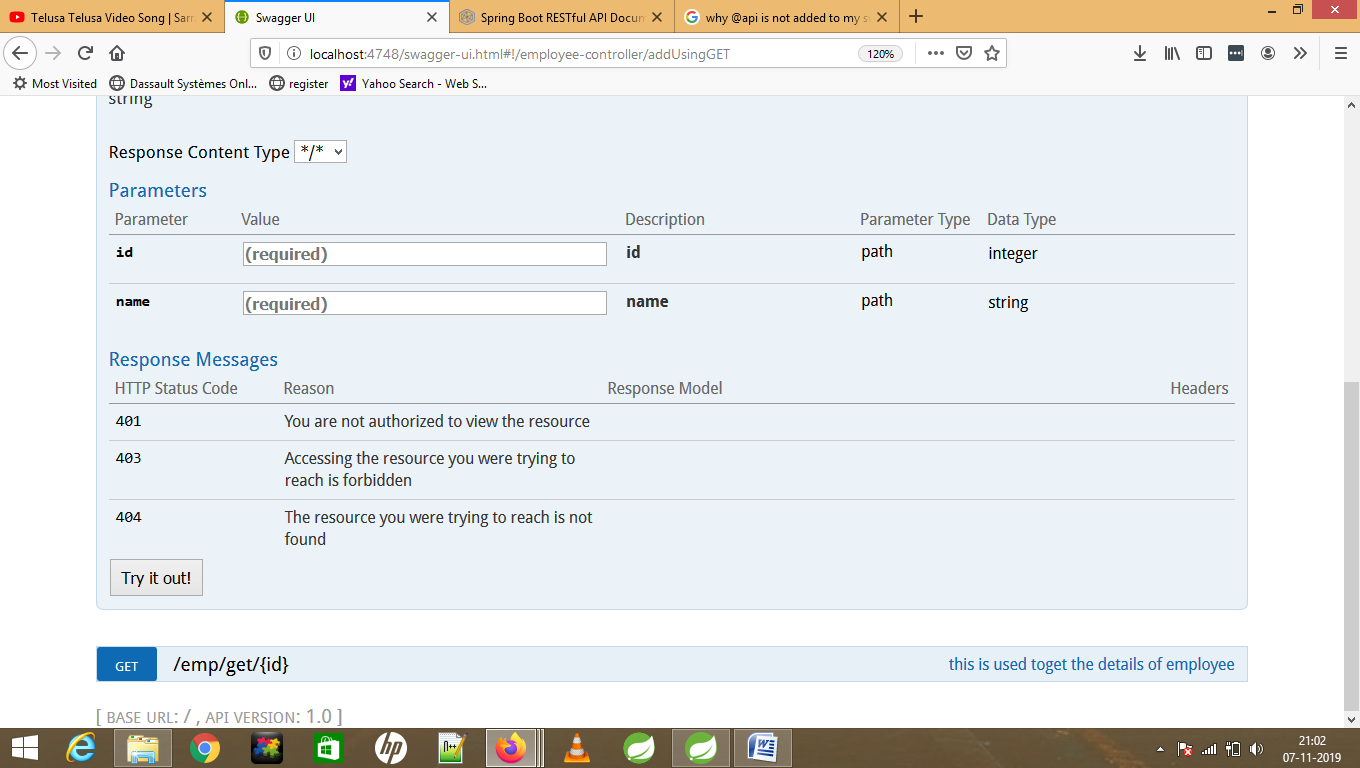
}

}

4) Run SpringBoot application and access documentation & UI using below URLs.

http://localhost:9090/swagger-ui.html --> for UI screen

http://localhost:9090/v2/api-docs --> for documentation in json

<http://localhost:4748/v2/api-docs>

* We can get like this documentation if you want see perfectly create a file in our application and copy this doc.

{"swagger":"2.0","info":{"description":"Spring Boot REST API for Online Store","version":"1.0","title":"Spring Boot REST API ","termsOfService":"Terms of service","contact":{"name":"Ashok ","url":"https://dzone.com/articles/spring-boot-restful-api-documentation-with-swagger","email":"john@springfrmework.guru"},"license":{"name":"Apache License Version 2.0","url":"https://www.apache.org/licenses/LICENSE-2.0"}},"host":"localhost:4748","basePath":"/","tags":[{"name":"employee-controller","description":"this is sendind data to employee controller "}],"paths":{"/emp/add/{id}/{name}":{"get":{"tags":["employee-controller"],"summary":"this is used to add the details of employee","operationId":"addUsingGET","consumes":["application/json"],"produces":["\*/\*"],"parameters":[{"name":"id","in":"path","description":"id","required":true,"type":"integer","format":"int32"},{"name":"name","in":"path","description":"name","required":true,"type":"string"}],"responses":{"200":{"description":"Successfully retrieved list","schema":{"type":"string"}},"401":{"description":"You are not authorized to view the resource"},"403":{"description":"Accessing the resource you were trying to reach is forbidden"},"404":{"description":"The resource you were trying to reach is not found"}}}},"/emp/get/{id}":{"get":{"tags":["employee-controller"],"summary":"this is used toget the details of employee","operationId":"gUsingGET","consumes":["application/json"],"produces":["application/json"],"parameters":[{"name":"id","in":"path","description":"id","required":true,"type":"integer","format":"int32"}],"responses":{"200":{"description":"OK","schema":{"$ref":"#/definitions/Employee"}},"401":{"description":"Unauthorized"},"403":{"description":"Forbidden"},"404":{"description":"Not Found"}}}}},"definitions":{"Employee":{"type":"object","properties":{"id":{"type":"integer","format":"int32"},"name":{"type":"string"}}}}}

**Spring Data:**

**Ref link:**[**https://github.com/TEK-Leads/ClassNotes**](https://github.com/TEK-Leads/ClassNotes)

Spring Data is used to develop persistent logic for our application.

This is mainly used to perform only CURD operations.

We no need to write any curd operations.

TO Avoid boiler plate coding spring data . In previous we write one controller for four curd operations(Save, update, delete,retrieve) (ht.save, ht.update, ht.delete, ht.get, ht.loadAll) if we have 100 controllers in project we write 500 curd operations. We no need to write any curd operation in spring data. That’s why Spring Data comes into the picture......

Similarly Spring Orm But in orm we write curd Operation in this we no need to write curd operation only implementing two interfaces given below..

As part of Spring Data, we have below 2 Repositories

1) CrudRepository (curd operations)

2) JpaRepository (curd operation + pagination + sorting)

**Curd Operations** :

**public Integer saveUom(Uom u);**

**public void updateUom(Uom u);**

**public void deleteUom(Integer id);**

**public Uom getUomById(Integer id);**

**public List<Uom> getAllUoms();**

**Pagination** is a technique for splitting a list of multiple records into sublists. For example, you search with a keyword on Google and receives tens of thousands of results. However, every Google page displays only 10 results for you. Other results will show on next pages.

**Sorting** : we can get Result in Sorting order

Database Setup

We need to install below 2 softwares to work with DB operations

1) DB Server (Oracle, MySql, SqlServer, DB2 etc)

-Install appropriate database in your pc, real we no need to down load any one we have a Data Repository we can get the port, username like we get

2) DB Client (SQL Developer, Toad)

-In industry used only these tools ..

-> First install Oracle Database 10g

-> Download SQL developer from Oracle website.

-> Create Connection in SQL developer using DB details.

Note: To Connect with Oracle Database we need Oracle Database Driver (OracleDriver). Oracle Driver will be available in ojdbc6.jar file.

**As Oracle is Proprietary software we can't get Maven Dependency From mvnrepository.com.**

To use ojdbc6.jar we have below 2 options

1) Add ojdbc6.jar file to project build path

2) Install ojdbc6.jar file to maven local repository and add local repository dependency in project pom.xml

Installing Third Party Jar to Maven Local Repository

1) Install Maven from <https://maven.apache.org/download.cgi>

2) Add MAVEN\_HOME to environment variable

Microsoft Windows [Version 6.3.9600]

(c) 2013 Microsoft Corporation. All rights reserved.

C:\Windows\system32>mvn -version

Apache Maven 3.6.2 (40f52333136460af0dc0d7232c0dc0bcf0d9e117; 2019-08-27T20:36:1

6+05:30)

Maven home: C:\apache-maven-3.6.2\bin\..

Java version: 1.8.0\_161, vendor: Oracle Corporation, runtime: C:\Program Files\J

ava\jdk1.8.0\_161\jre

Default locale: en\_IN, platform encoding: Cp1252

OS name: "windows 8.1", version: "6.3", arch: "amd64", family: "windows"

3) Use maven install command to install jar to local repo.

> mvn install:install-file -Dfile=ojdbc6.jar

-DgroupId=com.oracle

-DartifactId=ojdbc6

-Dversion=2.2.2

-Dpackaging=jar

Once above command is execute, ojdbc6.jar is installed to maven local repository. We can use below dependency in pom.xml to get ojdbc6.jar file to project build path.

<dependency>

<groupId>com.oracle</groupId>

<artifactId>ojdbc6</artifactId>

<version>2.2.2</version>

</dependency>

Note: This dependency applicable only for the system where above command is executed.

Working with SpringData in Project

1) Add Spring Data Jpa dependency in pom.xml file

<dependency>

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

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

</dependency>

<dependency>

<groupId>javax.persistence</groupId>

<artifactId>javax.persistence-api</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

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

</dependency>

2) Configure Data Source Properties in application.properties file or application.yml file

server.port=1010

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

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

spring.datasource.username=root

spring.datasource.password=admin

spring.jpa.database-platform=org.hibernate.dialect.MySQL55Dialect

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true

3) Create Entity Class (mapping with database table)

@Entity

@Table(name="emptab")

**public** **class** EmployeeEntity {

@Id

**private** **int** id;

**private** String name;

//setters and getters

}}

4) Create Repository interface by extending from SpringData Repository

@Repository

@Component

**public** **interface** IEmployeeRepository **extends** CrudRepository<EmployeeEntity,Integer>{

}

6) Inject Repository interface into service class and call methods to perform curd operations.

@Service

**public** **class** EmployeeServiceImpl **implements** IEmployeeService {

@Autowired

**private** Employee e;

@Autowired

**private** IEmployeeRepository emp;

@Override

**public** **void** save(Employee e) {

EmployeeEntity e1=**new** EmployeeEntity();

BeanUtils.*copyProperties*(e,e1);

//BeanUtils.copyProperties(source, target);

emp.save(e1);

}

@Override

**public** EmployeeEntity getData(Integer i) {

// **TODO** Auto-generated method stub

Optional<EmployeeEntity> o=emp.findById(i);

**if**(o.isPresent()) {

**return** o.get();

}**else** {

**throw** **new** EmployeeNotFoundException();

}

}

}

7)Rest Controller

@RestController

@Api(value="employee details",~~description~~="this is sendind data to employee controller ")

**public** **class** EmployeeController {

**public** EmployeeController() {

System.***out***.println("rest controller"); }

@Autowired

**private** IEmployeeService s;

@ApiResponses(value = {

@ApiResponse(code = 200, message = "Successfully retrieved added"),

@ApiResponse(code = 401, message = "You are not authorized to view the resource"),

@ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),

@ApiResponse(code = 404, message = "The resource you were trying to reach is not found")

}

)

@GetMapping(value="/insert",produces= {"application/json"})

**public** Employee addEmpby() {

Employee e=**new** Employee();

e.setId(100);

e.setName("nare");

String s="employee added Succesfully";

**return** e;

}

@PostMapping(value="/add",consumes= {"application/json"})

**public** String addEmp(@RequestBody Employee e) {

String msg="employee added Succesfully";

s.save(e);

**return** msg;

}

@GetMapping(

value="getemp/{id}",

produces= {"application/json"}

)

**public** Employee getEmployee(@PathVariable Integer id) {

EmployeeEntity e=s.getData(id);

Employee emp= **new** Employee();

BeanUtils.*copyProperties*(e,emp);

**return** emp;

}

**Ref:**[**https://dzone.com/articles/spring-boot-with-spring-data-jpa**](https://dzone.com/articles/spring-boot-with-spring-data-jpa)

|  |
| --- |
| Methods available in CrudRepository |

T save(T entity) --- polymorphic method (save and update)

Optional findById(Serializable id)

Iterable<T> findAll( )

Iterable<T> findAllById(Iterable<Serializable> ids)

delete(T entity)

deleteById(Serializable id)

deleteAll( )

deleteAllById(Iterable<Serializable> ids)

long count()

existById(Serializable id)

Custom Queries in Spring Data

We can execute Custom queries also in Spring data

Custom Queries are fail-fast

Custom queries will be compiled at the time of application startup

To write the custom queries we will use @Query annotation

*Custom Queries Examples*

@Query(value = "select name from EmployeeEntity")

**public** List<String> findAllEmpNames();

Application environments

Every applicaiton contains below environments to test properly.

Local : Development will happen

DEV : Developer Integration Testing

QA : System Integration Testing

UAT : User acceptance Testing

Pilot/Staging : Similar to prod to test with real data

Production : Go live

From environment to environment few properties values are going to change like DB, smtp, logging, webservices endponints etc....

In SpringBoot application we will configure below properties in application.properties file or application.yml file

1) DB Config Properties

2) ORM properties

3) SMTP properties

4) Webservice Endpoint URLs

5) Application messages

At the time of deployment changing environment specific properites everytime is difficult and time waste process.

To avoid this problem we will maintain envrinoment specific configuration files using Profiles concept.

SpringBoot profiles

Profiles are used to main environment specific configuration.

we will create profiles like below

application-dev.yml --> DEV profile file

server.port=1010

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

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

spring.datasource.username=narendra

spring.datasource.password=143saida

spring.jpa.database-platform=org.hibernate.dialect.MySQL55Dialect

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true

application-qa.yml --> QA profile file

server.port=1010

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

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

spring.datasource.username=root

spring.datasource.password=admin

spring.jpa.database-platform=org.hibernate.dialect.MySQL55Dialect

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true

application-uat.yml --> UAT profile file

server.port=1010

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

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

spring.datasource.username=uatadmin

spring.datasource.password=admin

spring.jpa.database-platform=org.hibernate.dialect.MySQL55Dialect

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true

application.yml ------ base file

Note : in base yml we need to activate profile like below

spring.profiles.active=dev

spring:

profiles:

active: qa

Note: As per above configuration application application will load

qa yml file.

*Creating User in Oracle Database*

1) Connect to DB as a SysDba in SQL command line

Open sql command line and execute below command

Command : connect as sysdba;

Username : dba

password : abc

2) Execute below queries

create user <uname> identified by <pwd>

grant connect, resource to <uname>

3) Commit

**Junit:**

The amount code will be covered by the testing ..

In spring boot we no need add any dependencies it comes along with starter web...

Mockito also comes with starter web..

@Mock: its creating dummy object to the our providing class

We are Using Annotation with junit:

@SpringBootTest

@InjectMocks

@Mock

@Test

***Mokito when().thenreturn()***

Enables stubbing methods. Use it when you want the mock to return particular value when particular method is called.

Simply put: "**When** the x method is called **then** return y".

Examples:

**when**(mock.someMethod()).**thenReturn**(10);

//you can use flexible argument matchers, e.g:

when(mock.someMethod(**anyString()**)).thenReturn(10);

//setting exception to be thrown:

when(mock.someMethod("some arg")).thenThrow(new RuntimeException());

**For all code available in this link up to junit:** [**https://github.com/narendradasara99/SpringBootRestProviderApp**](https://github.com/narendradasara99/SpringBootRestProviderApp)

**How to Use gitHub:**

* Open with credencials...
* Create new Repository with name and click on public
* Save code repo link
* Open STS
* Add git views-> windows->show view->search with git->select Git repositories and staging->open
* After completion of project
* Share Project : right click on project->team>share project
* Click on check box create or use parent folder of project
* Click on create Repository
* Finish
* Come to git repositories
* Add unstaged to to staged and write some desc in commit message
* Commit
* Give repository in uri
* And give credentials
* Click on next
* Finish
* Finally got success message

**Reactive Programming in spring api**

Spring 5 has embraced reactive programming paradigm by introducing a brand new reactive framework called **Spring WebFlux**.

Spring WebFlux uses a library called Reactor for its reactive support. Reactor is an implementation of the [Reactive Streams](https://github.com/reactive-streams/reactive-streams-jvm#reactive-streams) specification.

Reactor Provides two main types called Flux and Mono.

Both of these types implement the Publisher interface provided by Reactive Streams.

Flux is used to represent a stream of 0..N elements and

Mono is used to represent a stream of 0..1 element.

Although Spring uses Reactor as a core dependency for most of its internal APIs, It also supports the use of RxJava at the application level.

-we need to add dependency spring web reactive

<dependency>

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

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

    <version>2.1.6.RELEASE</version>

</dependency>

<dependency>

<groupId>io.projectreactor</groupId>

<artifactId>reactor-test</artifactId>

<scope>test</scope>

</dependency>

Link: -<https://www.callicoder.com/reactive-rest-apis-spring-webflux-reactive-mongo/>

 **Mono**: Returns 0 or 1 element.

|  |
| --- |
| Mono<String> mono = Mono.just("Alex");  Mono<String> mono = Mono.empty();  @GetMapping(value="/getmono", produces= {"Application/json"})  **public** Mono<Customer> monoEx() {  Customer c=**new** Customer();  c.setId(85);  c.setName("narendra");  Mono<Customer> m=Mono.*just*(c);  **return** m;  } |

 **Flux**: Returns 0…N elements. A Flux can be endless, meaning that it can keep emitting elements forever. Also it can return a sequence of elements and then send a completion notification when it has returned all of its elements.

|  |
| --- |
| Flux<String> flux = Flux.just("A", "B", "C");  Flux<String> flux = Flux.fromArray(new String[]{"A", "B", "C"});  Flux<String> flux = Flux.fromIterable(Arrays.asList("A", "B", "C"));    //To subscribe call method    flux.subscribe();  @GetMapping(value="/getflux", produces= {"Application/json"})  **public** Flux<Customer> fLuxEx() {  Flux<Customer> f=Flux.*fromStream*(Stream.*generate*(()-> **new** Customer(1,"narendra")));  Flux<Long> d=Flux.*interval*(Duration.*ofSeconds*(4));  **return** Flux.*zip*(f,d).map(Tuple2::getT1);  } |

**Client App:**

We can create client using below three ways

1)RestTemplate : Rest Template is synchronous communication

2)WebClient : Web client is both Synchronous and asynchronous communication

3)Fiegn client(Microservices)

-In real time we can the provider details from the Swagger Ui

1) getForEntity (GetMapping)

**public** **void** getAllNames() {

String url="http://localhost:1010/getname";

ResponseEntity<String> s=r.getForEntity(url,String.**class**);

**int** i=s.getStatusCode().value();

String msg=s.getBody();

System.***out***.println(msg);

}

2) getForEntity (GetMappingwithqueryparam )

**public** **void** getempbyid(Integer id) {

String url="http://localhost:1010/getemp/"+id;

ResponseEntity<String> s=r.getForEntity(url,String.**class**);

**int** i=s.getStatusCode().value();

**if**(i==200) {

String msg=s.getBody();

System.***out***.println(msg);

}

}

3) postForEntity (PostMapping)

**public** **void** addEmployee() {

String url="http://localhost:1010/add";

Employee e=**new** Employee(42,"hari");

ResponseEntity<String> s=r.postForEntity(url,e,String.**class**);

If u user requirement we can give header to the request using ResponsetEntity

**int** i=s.getStatusCode().value();

**if**(i==200) {

String msg=s.getBody();

System.***out***.println(msg);

}

Git:<https://github.com/narendradasara99/RestTemplateClientApp>

2) Web Client : Web client is both Synchronous and Asynchronous communication

Web client is a monolithic and flux architecture

1) bodyToMono :

Request is created one time with Synchronous and Asynchronous communication

Where as RestTemplate is only perform synchronous only . it effects performance

So Rest template is deprecated.WebClient is introduced Spring 5.0

It requests only one time

Block(): is a synchronous the request

Subscribe: is a asynchronous request it process the next request its not wait for response

2) bodyToFlux :

It requests only given period of time

Requested is created in only asynchronous communication

CommandLineRunner in spring boot :

Spring Boot we have a functional interface CommandLinerunner

This is used to perform whenever beans created command line Runner will executed

Here we can write the delete the database data like

After bean creation it will executed whether beans are created or not we check with cmdRunner

@SpringBootApplication

**public** **class** P5SbclrappApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(P5SbclrappApplication.**class**, args);

}

@Bean

**public** CommandLineRunner cmd() {

**return** args->{

System.***out***.println("comd runner");

};

}

@FunctionalInterface

**public** **interface** CommandLineRunner {

v**oid** run(String... args) **throws** Exception;

}

**ApplicationRunner**

Same as CommandLineRunner but difference is argument is Application arguments

@Bean

**public** ApplicationRunner ar() {

**return** args->{

System.***out***.println("hshd");

args.getSourceArgs();

};

}

@FunctionalInterface

**public** **interface** ApplicationRunner {

**void** run(ApplicationArguments args) **throws** Exception;

}

**In Line Memory Database In spring Boot**

Spring boot provides a inline memory databases for perform operation of database without real data base before going to add real database we can check with inline memory database.

-and We can use in the unit testing instead mocking we use inline database

-Use in memory database for your unit tests

An in memory database is created when an application starts up and destroyed when the application is stopped.

Advantages

* Zero project setup or infrastructure
* Zero Configuration
* Zero Maintainance
* Easy to use for Learning, POCs and Unit Tests
* Spring Boot provides Simple Configuration to switch between a real database and an in memory database like H2

## H2

H2 is one of the popular in memory databases. Spring Boot has very good integration for H2.

Choose following dependencies

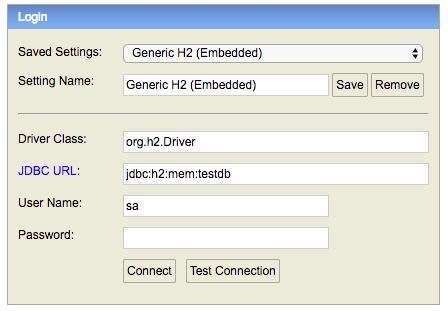
* Web
* JPA
* H2
* DevTools

Enabling H2 Console

spring.h2.console.enabled=true// default value is true

When you reload the application, you can launch up H2 Console at

http://localhost:8181/h2-console.



**Application.properties**

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

create table in :/src/main/resources/schema.sql

CREATE TABLE Persons (

ID int,

Name varchar(255),

City varchar(255)

);

Inser data in to table :/src/main/resources/data.sql

insert into persons values(8,'naren','vja');

insert into persons values(9,'naren','vja');

select \* from persons;

|  |  |  |
| --- | --- | --- |
| [**ID**](http://localhost:8585/h2-console/query.do?jsessionid=6543375ce2823b176466f4634ffe4d7c) | [**NAME**](http://localhost:8585/h2-console/query.do?jsessionid=6543375ce2823b176466f4634ffe4d7c) | [**CITY**](http://localhost:8585/h2-console/query.do?jsessionid=6543375ce2823b176466f4634ffe4d7c) |
| 8 | Naren | Vja |
| 9 | Naren | Vja |

Microservices:

[Spring Boot](https://dzone.com/articles/what-is-spring-boot) enables building production-ready applications quickly and provides non-functional features:

* Embedded servers which are easy to deploy with the containers.
* It helps in monitoring the multiples components.
* It helps in configuring the components externally.

Monolithic Architecture:

Developing all the business logic and all the functionalities in a single application is called monolithic architecture.

Dividing small scale applications is called micro service

Examples :

1)MoneyExchangeApp using h2db

* Dependencies

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

<dependency>

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

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

</dependency>

1)Model.java

package com.nare.bean;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

import org.springframework.stereotype.Component;

import lombok.Data;

@Data

@Component

public class Model {

private int id;

private String from;

private String to;

private int value;

}

2)Entity.java

package com.nare.entity;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

import org.springframework.stereotype.Component;

import lombok.Data;

@Component

@Data

@Entity

@Table(name="CURRENCYEXCHANGE")

public class CurrencyEntity {

@Id

@Column(name="id")

private int id;

@Column(name="currencyfrom")

private String cfrom;

@Column(name="currencytto")

private String to;

@Column(name="value")

private int value;

}

3) CurrencyExchangedao.java

package com.nare.dao;

import org.springframework.data.jpa.repository.Query;

import org.springframework.data.repository.CrudRepository;

import org.springframework.stereotype.Repository;

import com.nare.entity.CurrencyEntity;

@Repository

public interface CurrencyExchangedao extends CrudRepository<CurrencyEntity, Integer>{

@Query(value="from CurrencyEntity where cfrom=:f and to=:t")

public CurrencyEntity exchange(String f, String t);

}

4) CurrencyService.java

package com.nare.service;

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

import org.springframework.stereotype.Service;

import com.nare.bean.Model;

import com.nare.dao.CurrencyExchangedao;

import com.nare.entity.CurrencyEntity;

@Service

public class CurrencyService {

@Autowired

private CurrencyExchangedao dao;

public CurrencyEntity currencyexchange(String from,String to) {

System.out.println("djjfjfjj");

CurrencyEntity c=dao.exchange(from,to);

System.out.println(c+"djjfjfjj");

System.out.println();

return c;

}

}

5) CurrencyController.java

package com.nare.controller;

import org.springframework.beans.BeanUtils;

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

import org.springframework.context.annotation.Bean;

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

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

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

import com.nare.bean.Model;

import com.nare.entity.CurrencyEntity;

import com.nare.service.CurrencyService;

@RestController

public class CurrencyController {

@Autowired

private CurrencyService s;

@GetMapping(value="/exchange/from/{from}/to/{to}",produces= {"application/json"})

public CurrencyEntity exchangem(@PathVariable("from")String from,@PathVariable("to")String to) {

Model m=new Model();

System.out.println("before"+from+""+to);

CurrencyEntity e=s.currencyexchange(from, to);

BeanUtils.copyProperties(e, m);

return e;

}

}

6)Schema.sql

CREATE TABLE CURRENCYEXCHANGE(

ID int,

currencyfrom varchar(255),

currencytto varchar(255),

value int

);

7)data.sql

insert into currencyexchange values(1,'Aus','inr',48.60);

insert into currencyexchange values(2,'usa','inr',71.74);

insert into currencyexchange values(3,'uero','inr',79.04);

8)application.properties

server.port=8888

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

2)MoneyConversionApp using MoneyExchangeApp as provider

* Dependencies:

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

1)CurrencyConversionBean.java

**package** com.nare.bean;

**public** **class** CurrencyConversionBean {

**private** String from;

**private** String to;

**private** **int** v;

}

}

2) CurrencyEntity.java

**package** com.nare.bean;

**public** **class** CurrencyEntity {

**private** **int** id;

**private** String cfrom;

**private** String to;

**private** **int** value;

}

3) CurrencyController.java

package com.nare.controller;

import javax.xml.bind.ParseConversionEvent;

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

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

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

import org.springframework.web.reactive.function.client.WebClient;

import com.nare.bean.CurrencyConversionBean;

import com.nare.bean.CurrencyEntity;

@RestController

public class CurrencyController {

@GetMapping(value="/get/from/{from}/to/{to}/{value}")

public CurrencyConversionBean conversion(@PathVariable("from")String from ,@PathVariable("to")String to,@PathVariable("value")String val) {

String url="http://localhost:8888/exchange/from/"+from+"/to/"+to;

WebClient.Builder b=WebClient.builder();

WebClient w=b.build();

CurrencyEntity e=w.get()

.uri(url)

.retrieve()

.bodyToMono(CurrencyEntity.class).block();

int v=e.getValue()\*(Integer.parseInt(val));

CurrencyConversionBean cc=new CurrencyConversionBean();

cc.setFrom(from);

cc.setTo(to);

cc.setV(v);

return cc;

}

}

Pivotal Cloud Foundry:

[Pivotal Cloud Foundry (PCF)](https://pivotal.io/platform) is a multi-cloud platform for the deployment, management, and continuous delivery of applications, containers, and functions. PCF is a distribution of the [open source Cloud Foundry](https://www.cloudfoundry.org/) developed and maintained by Pivotal Software, Inc. PCF is aimed at enterprise users and offers additional features and services—from Pivotal and from other third parties—for installing and operating Cloud Foundry as well as to expand its capabilities and make it easier to use. Major cloud platforms such as Amazon Web Services and Google Cloud also provide templates and quick starts that automate large portions of the PCF deployment process.

Pivotal Cloud Foundry abstracts away the process of setting up and managing an application runtime environment so that developers can focus solely on their applications and associated data. Running a single command—cf push—will create a scalable environment for your application in seconds, which might otherwise take hours to spin up manually. PCF allows developers to deploy and deliver software quickly, without needing to manage the underlying infrastructure.

Steps:

1. create account in <https://run.pivotal.io/>

2. choose option for free and create account

3. choose pivotal web services

4. Claim Your Free Trial we need to enter phone no Create Org

5. how to push we can check <https://docs.run.pivotal.io/devguide/deploy-apps/deploy-app.html#push>

https://run.pivotal.io

To use the cf CLI installer for the Windows command line, perform the following steps:

1. Download [the Windows installer](https://cli.run.pivotal.io/stable?release=windows64&source=github).
2. Unpack the zip file.
3. Right click on the ‘cf\_installer’ executable and select “Run as Administrator”
4. When prompted, click **Install**, then **Finish**.
5. To verify your installation, open a command prompt and type cf. If your installation was successful, the cf CLI help listing appears. You may need to restart the command prompt to see the cf CLI help listing appear.

Maveh build goals:

* validate - validate the project is correct and all necessary information is available
* compile - compile the source code of the project
* test - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
* package - take the compiled code and package it in its distributable format, such as a JAR.
* verify - run any checks on results of integration tests to ensure quality criteria are met
* install - install the package into the local repository, for use as a dependency in other projects locally
* deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

Deployment

Login in cmd:cf login https://api.run.pivotal.io

Credentials:dasaranarendra99@gmail.com

D:\Narendra Java\SpringBoot\P6SBCurrencyExchangeApp>cf push ceapp -p target/curr

encyexchange.jar

Pushing app ceapp to org naren / space development as dasaranarendra99@gmail.com

...

Getting app info...

Creating app with these attributes...

+ name: ceapp

path: D:\Narendra Java\SpringBoot\P6SBCurrencyExchangeApp\target\Currenc

yExchange.jar

routes:

+ ceapp.cfapps.io

Then we execute our app with:

<https://ceapp.cfapps.io/exchange/from/Aus/to/inr>

**Cf Cli Commands**

cf - A command line tool to interact with Cloud Foundry

## GETTING STARTED

[help](http://cli.cloudfoundry.org/en-US/cf/help.html) Show help

[version](http://cli.cloudfoundry.org/en-US/cf/version.html) Print the version

[login](http://cli.cloudfoundry.org/en-US/cf/login.html) Log user in

[logout](http://cli.cloudfoundry.org/en-US/cf/logout.html) Log user out

[passwd](http://cli.cloudfoundry.org/en-US/cf/passwd.html) Change user password

[target](http://cli.cloudfoundry.org/en-US/cf/target.html) Set or view the targeted org or space

[api](http://cli.cloudfoundry.org/en-US/cf/api.html) Set or view target api url

[auth](http://cli.cloudfoundry.org/en-US/cf/auth.html) Authenticate non-interactively

## APPS

[apps](http://cli.cloudfoundry.org/en-US/cf/apps.html) List all apps in the target space

[app](http://cli.cloudfoundry.org/en-US/cf/app.html) Display health and status for an app

[push](http://cli.cloudfoundry.org/en-US/cf/push.html) Push a new app or sync changes to an existing app

[scale](http://cli.cloudfoundry.org/en-US/cf/scale.html) Change or view the instance count, disk space limit, and memory limit for an app

[delete](http://cli.cloudfoundry.org/en-US/cf/delete.html) Delete an app

[rename](http://cli.cloudfoundry.org/en-US/cf/rename.html) Rename an app

[start](http://cli.cloudfoundry.org/en-US/cf/start.html) Start an app

[stop](http://cli.cloudfoundry.org/en-US/cf/stop.html) Stop an app

[restart](http://cli.cloudfoundry.org/en-US/cf/restart.html) Stop all instances of the app, then start them again. This causes downtime.

[logs](http://cli.cloudfoundry.org/en-US/cf/logs.html) Tail or show recent logs for an app

[create-app-manifest](http://cli.cloudfoundry.org/en-US/cf/create-app-manifest.html) Create an app manifest for an app that has been pushed successfully

[get-health-check](http://cli.cloudfoundry.org/en-US/cf/get-health-check.html) Show the type of health check performed on an app

[set-health-check](http://cli.cloudfoundry.org/en-US/cf/set-health-check.html) Change type of health check performed on an app

## ORGS

[orgs](http://cli.cloudfoundry.org/en-US/cf/orgs.html) List all orgs

[org](http://cli.cloudfoundry.org/en-US/cf/org.html) Show org info

[create-org](http://cli.cloudfoundry.org/en-US/cf/create-org.html) Create an org

[delete-org](http://cli.cloudfoundry.org/en-US/cf/delete-org.html) Delete an org

For More commands : <http://cli.cloudfoundry.org/en-US/cf/>

**Deploying CurrencyConversionApp**

D:\Narendra Java\SpringBoot\P7SBCurrencyConversionApp\target>cf push currenycapp

-p currencyconvertion.jar

Pushing app currenycapp to org naren / space development as dasaranarendra99@gma

il.com...

Getting app info...

Creating app with these attributes...

+ name: currenycapp

path: D:\Narendra Java\SpringBoot\P7SBCurrencyConversionApp\target\Curre

ncyConvertion.jar

routes:

+ currenycapp.cfapps.io

Route: <https://currenycapp.cfapps.io/get/from/usa/to/inr/10>

3.Developing Currency conversion UI APP

Developing C To B

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>jasper</artifactId>

<version>6.0.32</version>

</dependency>

**public** **class** CurrencyappBean {

**private** String from;

**private** String to;

**private** **int** v;

}

**public** **class** CurrencyResponse {

**private** String from;

**private** String to;

**private** **int** v;

}

@Service

**public** **class** CurrencyService {

**public** CurrencyResponse currency( CurrencyappBean cbean) {

String url="https://currenycapp.cfapps.io/get/from/{from}/to/{to}/{value}" ;

WebClient.Builder w=WebClient.*builder*();

WebClient wb= w.build();

CurrencyResponse c=wb.get()

.uri(url,cbean.getFrom(),cbean.getTo(),cbean.getV())

.retrieve()

.bodyToMono(CurrencyResponse.**class**)

.block();

**return** c;

}

@Controller

@RequestMapping(value="/c")

**public** **class** CurrencyController {

@Autowired

**private** CurrencyService ser;

@RequestMapping(value="/nar")

**public** String sho(@ModelAttribute("currencyappBean") CurrencyappBean currencyappBean,Model m) {

m.addAttribute("cbean",currencyappBean );

**return** "CurrencyApp";

}

@RequestMapping(value="/save",method=RequestMethod.***POST***)

**public** String postdata(@ModelAttribute("currencyappBean") CurrencyappBean currencyappBean,Model m) {

CurrencyResponse c=ser.currency(currencyappBean);

m.addAttribute("cbean",c );

**return** "CurrencyApp";

}

}

app.properties

server.port=2222

spring.mvc.view.suffix=.jsp

spring.mvc.view.prefix=/views/

CurrencyApp.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=ISO-8859-1"*

pageEncoding=*"ISO-8859-1"*%>

<%@ taglib uri=*"http://www.springframework.org/tags/form"* prefix=*"f"*%>

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

<!DOCTYPE html>

<html>

<head>

<meta charset=*"ISO-8859-1"*>

<title>Insert title here</title>

</head>

<body>

<h3>Currency Conversion api</h3>

<f:form method=*"post"* action=*"/c/save"* modelAttribute=*"currencyappBean"*>

<table>

<tbody>

<tr><td>enter from value</td>

<td><f:input path=*"from"*/>

</tr>

<tr><td>enter to value</td>

<td><f:input path=*"to"*/>

</tr>

<tr><td>enter value to convert</td>

<td><f:input path=*"v"*/>

</tr>

<tr><td></td>

<td><input type=*"submit"* value=*"converted"*/></td>

</tr>

</tbody>

</table>

<c:if test=*"*${cbean.v!=0}*"*>

<h3></h3>

<c:out value=*" Converted amount :*${cbean.v}*"* />

</c:if>

</f:form>

</body>

</html>

Spring MVC Flow

---------------

DispatcherServlet : FrontController

HandlerMapper : mapping between request and request-handler

Controller : Request-Handler (@Controller)

ModelAndView : Model represents data, view represnets UI page

ViewResolver : To locate view files, to understand view technology

View : To render Model Data on View File

Service Registry For MicroServices

----------------------------------

Service Registry is used to maintain project related microservices details.

We can use Netflix-Eureka Server as a Service Registry.

Eureka Server is an application that holds the information about all client-service applications. Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

Eureka Server comes with the bundle of Spring Cloud. For this, we need to develop the Eureka server and run it on the default port 8761.

Working with Service Registry

------------------------------

1) Create SpringBoot Web application

2) Add Eureka-Server dependency in pom.xml file

<dependency>

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

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

</dependency>

<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>

<properties>

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

<spring-cloud.version>Hoxton.RELEASE</spring-cloud.version>

</properties>

3) Use @EnableEurekaServer at SpringBoot start class

@SpringBootApplication

@EnableEurekaServer

**public** **class** P6MSpringEurekaServerApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(P6MSpringEurekaServerApplication.**class**, args);

}

}

4) ServiceRegistry project should not register with Eureka

eureka.client.register-with-eureka=false

spring.application.name="SpringUrekaServer"

eureka.client.register-with-eureka=false

server.port = 8761

5) After Running this application, eureka dashboard using below url

localhost:<port-num>

http://localhost:8761/

Creating MicroService which acts as Eureka Client

--------------------------------------------------

1) Create SpringBoot Web application

2) Add Eureka-Client dependency in pom.xml file

<dependency>

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

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

</dependency>

<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>

<properties>

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

<spring-cloud.version>Hoxton.RELEASE</spring-cloud.version>

</properties>

3) Use @EnableDiscoveryClient at SpringBoot main class

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** P10MSpringEurekaclient1Application {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(P10MSpringEurekaclient1Application.**class**, args);

}

}

4) Create Required RestControllers, Services etc...

Ex:

@RestController

**public** **class** Greetcontroller {

@RequestMapping(value="/greet/{name}")

**public** String greet(@PathVariable("name") String name) {

String s="hello "+name;

**return** s;

}

}

5) Change port number in application.yml file

6) Add Application Name application.yml file (It is vvvvimp)

server.port=4568

spring.application.name="Spring Eureka Client1 Greet Message"

7) Run application, then check Eureka Dashboard

Note: If Eureka Service Registry Project running in diff machine, then we should add below property in client application yml file.

eureka:

client:

service-url:

defaultZone: http://IP:PORT-NUM/eureka

Creating API Gateway

---------------------------

An API Gateway is a server that is the single entry point into the system. It is similar to the [Facade](http://en.wikipedia.org/wiki/Facade_pattern) pattern from object-oriented design. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

1) Create SpringBoot Web application

2) Add Eureka Client Dependency in pom.xml

3) Add Zuul dependency in pom.xml

<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-zuul</artifactId>

</dependency>

4) Use @EnableDiscoveryClient at SpringBoot start class

@SpringBootApplication

@EnableDiscoveryClient

@EnableZuulProxy

**public** **class** P12ApiGatewayApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(P12ApiGatewayApplication.**class**, args);

}

}

5) Change port number and configure Application name in yml file

server:

port: 9999

spring:

application:

name: APIGateWay

zuul:

prefix: /api

routes:

welcome:

path: welcome/\*\*

service-id: Spring Eureka Client2 Welcome Message

greet:

path: greet/\*\*

service-id: Spring Eureka Client1 Greet Message

6) Run this application and check Eureka dashboard.

Link: <https://howtodoinjava.com/spring-cloud/spring-cloud-api-gateway-zuul/>

Run this application and check Eureka dashboard.

# Fiegn Client: Microservices Communication: Feign as REST Client