SpringBoot

Limitations with Spring Framework

- In spring framework a user is responsible for following things.
- Adding the dependencies / jar files
- Performing the configuration (applicationContext.xml)
- Need to Arrange physical server like tomcat.
- Need to Arrange physical database like oracle.

To overcome this limitations we need to use Spring Boot framework.

ex:

```
Developer
   |
   |
Spring Boot
   |
   |
Spring Framework
```

Spring Boot Framework

- It is a open source java based application framework which is developed by Pivotal Team.
- It provides RAD(Rapid Application Development) features to spring based applications.
- It is standalone , production ready grade sprin based applications with minimum configurations.
- In short spring boot is a combination of

  Spring Framework + Embedded Database + Embedded Server

- It does not support xml configuration instead we can use Annotations.

Advantages of Spring Boot

- It is used to create a standalone applications which can be run using java -jar.
- It provides production ready grade features like metrics , health check , externalization configuration and etc.
- It reduce development time and gives productivity.
- It provides optionate starters POM's to simply the maven configurations.
- It offers number of plugins.
- It test the application with various HTTP servers like Tomcat,Jetty and Undertow.
- There is no xml configurations in spring boot.
- It provides CLI tool for developing and testing spring boot applications.

Interview Question

Q) List out spring boot components?

There are four components in spring boot.

- AutoConfiguration
- Starter
- CLI Tool
- Actuators

Q) Where we will do configurations in spring boot?

We perform configurations in following files.

1) application.properties

2) application.yml

Q) List out some stereotype annotations?

@Controller

@Service

@Repository

@Component

and etc.

Q) How to create a controller in spring boot?

To create a controller we need to use @Controller annotation.

Q) How to create a service in spring boot?

To create a service we need to use @Service annotation.

Q) How to create a repository in spring boot?

To create a repository we need to use @Repository annotation.

Q) List out spring boot embedded databases?

H2

HSQL

Derby

Q) List out spring boot embedded servers?

Tomcat

Jetty

Undertow

Q)In how many ways we can create spring boot application?

There are two ways to create a spring boot application.

1) spring intializr (https://start.spring.io/)

2) Using IDE's like STS or IntelliJ

Q) Which component is used to add the dependencies in the spring boot?

Spring Boot starter is responsible to add dependencies in spring boot.

Q) What is spring initializr ?

It is a web-based tool to create spring boot project structure.

ex: https://start.spring.io

Q) What is @SpringBootApplication?

It is a combination of three anotations.

@EnableAutoConfiguration : enable Spring Boot's auto-configuration mechanism.

@ComponentScan : It scan on the package where the application is located.

@Configuration : allow to register extra beans in the context.

Steps to develop first spring boot application

step1:

Goto spring initializr

ex:

https://start.spring.io/

step2:

Create a spring boot project i.e FirstSB.

ex:

project	:	Maven
Language	:	Java
Dependencies	:	(don't add)
Spring Boot	:	3.1.1

Project Metadata

Group	:	com.ihub.www
Artifact	:	FirstSB
Name	:	FirstSB
Description	:	Demostration on Spring Boot
Package name	:	com.ihub.www
packaging	:	jar
Java	:	8

---> click on Generate button.

step3:

Download and Install STS IDE.

STS is a eclipse Based Environment.

ex:

https://spring.io/tools

step4:

Launch STS IDE by choosing workspace location.

step5:

Extract "FirstSB.zip" file in any loctation.

step6:

Open "FirstSB" project in STS IDE.

ex:

File --> Open project from file system --> click to directory button.

--> select FirstSB folder --> Finish.

step7:

Add custom message in FirstSBApplication.java file.

step8:

Run the spring boot appilication.

ex:

right click to the project --> run as --> spring boot application.

Spring Boot Starters

- Spring Boot provides a number of starters that allow us to add jars in the classpath.
- Spring Boot built-in starters make development easier and rapid.
- Spring Boot Starters are the dependency descriptors.
- In the Spring Boot Framework, all the starters follow a similar naming pattern:
- spring-boot-starter-*, where * denotes a particular type of application.

ex:

spring-boot-starter-test

spring-boot-starter-web

spring-boot-starter-validation (bean validation)

spring-boot-starter-security

spring-boot-starter-data-jpa

spring-boot-starter-data-mongodb

spring-boot-starter-mail

Third-Party Starters

We can also include third party starters in our project.

The third-party starter starts with the name of the project.

ex:

abc-spring-boot-starter.

Spring Boot Starter Web

- There are two important features of spring-boot-starter-web.
- It is compatible for web development
- AutoConfiguration
- If we want to develop a web application,we need to add the following dependency in pom.xml file.

ex:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>2.2.2.RELEASE</version>
</dependency>
```

- Spring web starter uses Spring MVC, REST and Tomcat as a default embedded server.
- The single spring-boot-starter-web dependency transitively pulls in all dependencies related to web development.
- By default, the spring-boot-starter-web contains the following tomcat server dependency:

ex:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <version>2.0.0.RELEASE</version>
        <scope>compile</scope>
</dependency>
```

The spring-boot-starter-web ,auto-configures the following things that are required for the web development:

1)Dispatcher Servlet

2)Error Page

3)Web JARs for managing the static dependencies

4)Embedded servlet container

Spring Boot + JSP Application

Project structure

```
MVCApp1
|
|----src/main/java
|       |
|       |----com.ihub.www
|               |
|               |--MVCApp2Application.java
|               |--HomeController.java
|---src/main/resources
|       |
|       |-----application.properties
|
|---src/test/java
|       |
|       |-----MVCApp1ApplicationTests.java
|
| --
| --
| --
|-----src
         |
         |----main
                  |
                  |----webapp
                           |
                           |----pages
                           |        |
                                    |----index.jsp
|---pom.xml
|
|
```

step1:

Create a spring starter project.

ex:

File --> new --> spring starter project -->

Name : MVCApp1

Group: com.ihub.www

Artifact: MVCApp1

Description: This is Spring Boot Application with JSP

package : com.ihub.www ---> next -->

Starter: Spring Web --> next --> Finish.

step2:

create a HomeController class inside "src/main/java".

ex:

Right click to package(com.ihub.www) --> new --> class --> Class: HomeController

-->finish.

step3:

Add @Controller annotation and "@RequestMapping" annotation inside HomeController class.

HomeController.java

```
package com.ihub.www;


import org.springframework.stereotype.Controller;

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


@Controller

public class HomeController {


	@RequestMapping("home")

	public String home()

	{

		return "index";

	}

}
```

step4:

create a "webapp" and "pages" folder inside "src/main" folder for adding JSP files.

ex:

```
|-----src
        |
        |----main
                |
                |----webapp
                        |
                        |-----pages
```

step5:

create "index.jsp" file inside "src/main/webapp/pages/" folder.

ex:

Right click to pages folder--> new --> file --->

File Name: index.jsp --> finish.

index.jsp

```
<center>
    <h1>
        I love Spring Boot Programming
    </h1>
</center>
```

step6:

Add "Tomcat Embed Jasper" dependency to read the jsp file.

ex:

```
<dependency>
          <groupId>org.apache.tomcat.embed</groupId>
          <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

Note:

Embedded Tomcat server does not have Jasper. So we need to add above dependency.

step7:

Configure tomcat server port number and jsp file.

application.properties

```
server.port=9191
spring.mvc.view.prefix=/pages/
spring.mvc.view.suffix=.jsp
```

step8:

Run Spring Boot application.

ex:

Right click to MVCApp2 --> run as --> spring boot application.

step9:

Test the application with below request url.

ex:

http://localhost:9191/home

Interview question

Q)In spring boot mvc based web application who will pass HTTP request to controller?

ans) DispatcherServlet

Q)Tomcat embedded server by default runs under which port no?

8080

Q)To create a spring mvc based web application we need to add which starter?

spring-boot-stater-web

Q)How to change tomcat embedded server port no?

ans)

application.properties

```
server.port = 9191
```

Spring Data JPA

- Spring Data JPA handles most of the complexity of JDBC-based database access and ORM (Object Relational Mapping).
- It reduces the boilerplate code required by JPA(Java Persistence API).
- It makes the implementation of your persistence layer easier and faster.
- Spring Data JPA aims to improve the implementation of data access layers by reducing the effort to the amount that is needed.

- Spring Boot provides spring-boot-starter-data-jpa dependency to connect Spring application with relational database efficiently.

ex:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
        <version>2.2.2.RELEASE</version>
        </dependency>
```

The spring-boot-starter-data-jpa internally uses the spring-boot-jpa dependency.

Spring Data JPA provides three repositories are as follows:

CrudRepository:

It offers standard create, read, update, and delete It contains method like findOne(), findAll(), save(), delete(), etc.

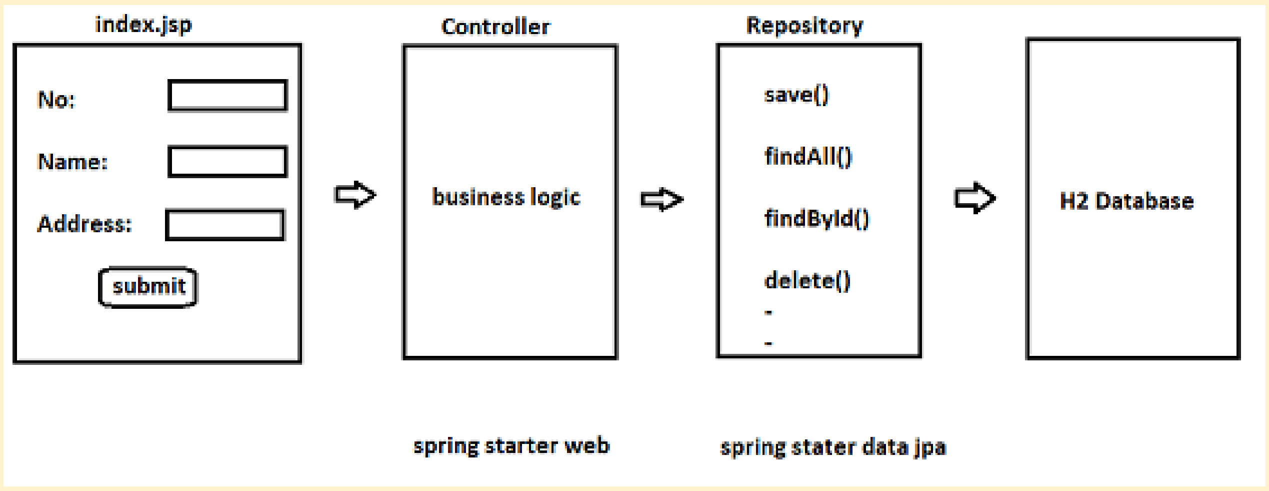
PagingAndSortingRepository:

It extends the CrudRepository and adds the findAll methods. It allows us to sort and retrieve the data in a paginated way.

JpaRepository:

It is a JPA specific repository It is defined in Spring Data Jpa. It extends the both repository CrudRepository and PagingAndSortingRepository. It adds the JPA-specific methods, like flush() to trigger a flush on the persistence context.

Diagram: sb2.1



Spring Boot application interact with H2 Database

Project structure

```
MVCApp2
|
|---src/main/java
        |
        |---com.ihub.www
                |
                |----MVCApp2Application.java
                |
        |---com.ihub.www.controller
                |
                |----EmployeeController.java
        |
        |---com.ihub.www.model
                |
                |----Employee.java
        |---com.ihub.www.repository
                |
                |----EmployeeRepository.java (interface)

|---src/main/resources
        |
        |---aplication.properties

|---src/test/java
        |
        |---com.ihub.www
                |
                |----MVCApp2ApplicationTests.java
|
|-----src
        |
        |-----main
                |
                |----webapp
                        |
                        |---index.jsp
                        |
                        |---view.jsp
|
|-----pom.xml
```

step1:

Launch STS IDE by choosing workspace location.

step2:

create a spring boot project i.e "MVCApp2".

ex:

starters:

Spring Web

Spring Data Jpa

H2 Database

step2:

Add Tomcat Embed Jasper dependency inside "pom.xml" file.

ex:

```
<dependency>
<groupId>org.apache.tomcat.embed</groupId>
<artifactId>tomcat-embed-jasper</artifactId>
</dependency>
```

step3:

create a "index.jsp" file inside "src/main" folder.

ex:

index.jsp

```
<form action="addEmp">
```

```
        <table align="center">
            <tr>
                <td>Id:</td>
                <td><input type="text" name="empId"/></td>
            </tr>
            <tr>
                <td>Name:</td>
                <td><input type="text" name="empName"/></td>
            </tr>
            <tr>
                <td>Address:</td>
                <td><input type="text" name="empAdd"/></td>
            </tr>
            <tr>
                <td><input type="reset" value="reset"/></td>
                <td><input type="submit" value="submit"/></td>
            </tr>
        </table>
</form>
```

step4:

Create a "Employee.java" file inside "com.ihub.www.model" package.

Employee.java

```
package com.ihub.www.model;


import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name="employees")
public class Employee
{
	@Id
	private int empId;

	@Column
	private String empName;

	@Column
	private String empAdd;

	public int getEmpId() {
		return empId;
	}

	public void setEmpId(int empId) {
		this.empId = empId;
	}

	public String getEmpName() {
		return empName;
	}

	public void setEmpName(String empName) {
		this.empName = empName;
	}

	public String getEmpAdd() {
		return empAdd;
	}

	public void setEmpAdd(String empAdd) {
		this.empAdd = empAdd;
	}
}
```

step5:

Create a "EmployeeRepository.java" file inside "com.ihub.www.repostory" pkg.

EmployeeRepository.java

```
package com.ihub.www.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.ihub.www.model.Employee;


@Repository
public interface EmployeeRepository extends
JpaRepository<Employee,Integer>
{


}
```

step6:

Create a "HomeController.java" file inside "src/main/java/com.ihub.www" package.

EmployeeController.java

```
package com.ihub.www.controller;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;


import com.ihub.www.model.Employee;
import com.ihub.www.repository.EmployeeRepository;


@Controller
public class EmployeeController
{
	@Autowired
	EmployeeRepository employeeRepository;

	@RequestMapping("/")
	public String home()
	{
		return "index.jsp";
	}

	@RequestMapping("/addEmp")
	public String addEmployee(Employee employee)
	{
		employeeRepository.save(employee);

		return "view.jsp";
	}
}
```

step7:

create a view.jsp file .

view.jsp

```
<center>
<h1>
			Record Inserted Successfully..
</h1>
```

	</center>

step8:

	Add port number and database configurations inside

	application.properties file.

application.properties

```
server.port=9191

spring.datasource.url= jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
```

step9:

	Test the spring boot application.

	ex:

		http://localhost:9191

		http://localhost:9191/h2-console

Spring Boot Application using @RestController

Spring RestController annotation is a convenience annotation that is itself annotated with @Controller and @ResponseBody

Spring RestController annotation is used to create RESTful web services using Spring MVC.

Project structure

```
MVCApp3
|
|----src/main/java
|        |
|        |----com.ihub.www
|                  |
|                  |--MVCApp3Application.java

|        |----com.ihub.www.controller
                   |
                   |--HomeController.java

|---src/main/resources
|        |
|        |-----application.properties
|
|---src/test/java
|        |
|        |-----MVCApp3ApplicationTests.java
|
| --
| --
| --

|---pom.xml
```

|

|

step1:

Create a spring starter project i.e MVApp3.

ex: **starter:**

spring web

step2:

create a HomeController class inside "src/main/java".

ex:

Right click to package(com.ihub.www.controller) --> new --> class --> Class: HomeController -->finish.

step3:

Add @RestController annotation and "@GetMapping" annotation inside HomeController class.

HomeController.java

```
package com.ihub.www;

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

@RestController
public class HomeController {

        @GetMapping("/msg")
        public String home()
        {
                return "I Love Spring Boot concepts";
        }
}
```

step4:

Configure tomcat server port number and jsp file.

application.properties

```
server.port=9191
```

step5:

Run Spring Boot application.

ex:

Right click to RestApp --> run as --> spring boot application.

step6:

Test the application with below request url.

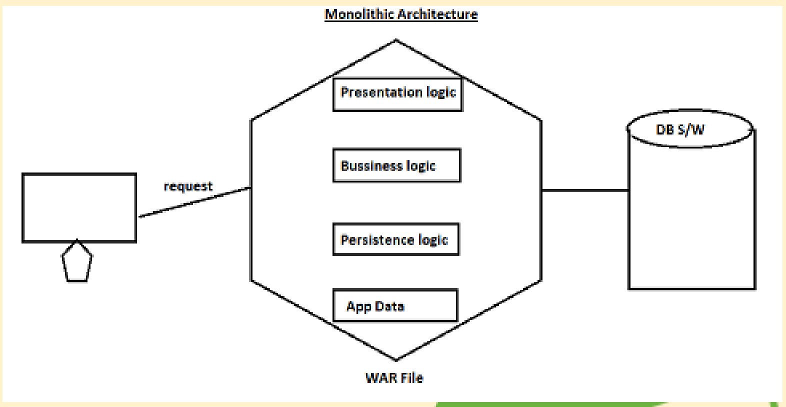
ex:

http://localhost:9191/

Monolithic Architecture

In Monolithic Architecture we are developing every service individually and at end of the development we are packaging all services as single war file and deploying in a server.

Diagram : sb4.1



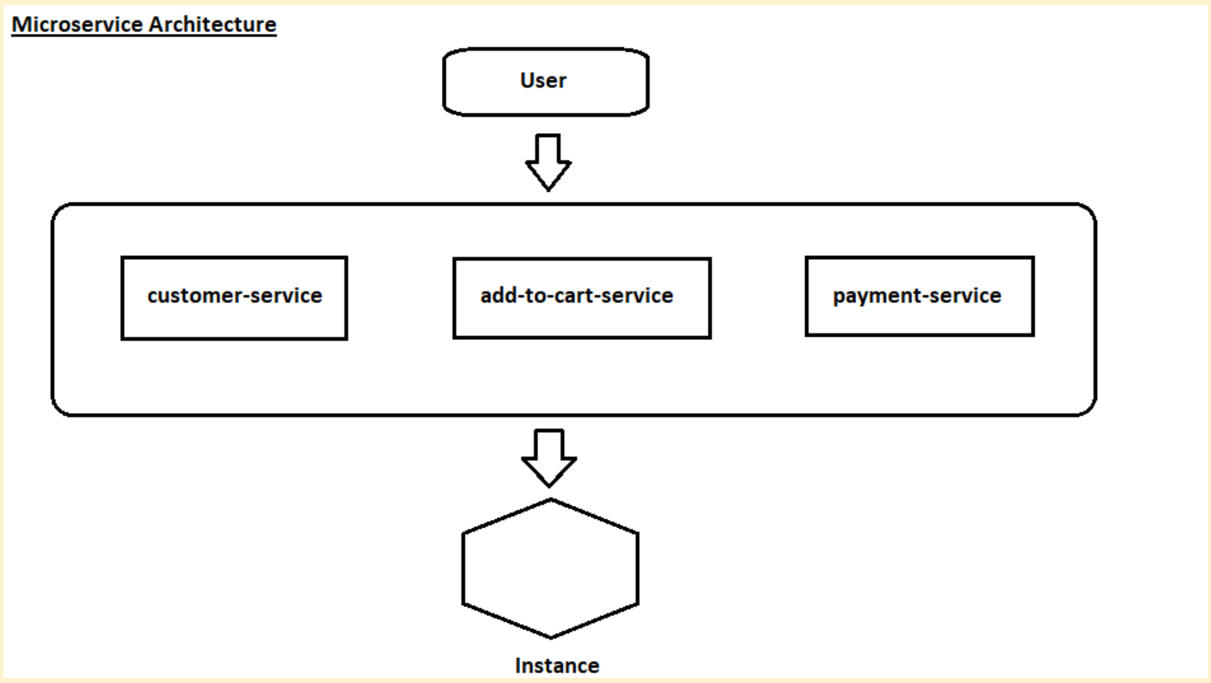
Drawbacks of Monolithic Architecture

- Not suitable for Large and Complex Application
- Slow Development
- Blocks Continenous development
- Unscalable
- Inflexible

Microservice architecture

- The microservice architectural style is an approach to develop a single application as a suite of small services.It is next to Service-Oriented Architecture (SOA).
- Each microservice runs its process and communicates with lightweight mechanisms.
- These services are built around business capabilities and independently developed by fully automated deployment machinery.

Diagram: sb4.2



Advantages of Microservice Architecture

- Independent Development
- Independent Deployment
- Fault Tolerance
- Mixed Technology Stack
- Granular Scaling

List of companies which are working with Microservices

Amazon

NetFlix

SoundCloud

Twitter

Uber

PayPal

ebay

GILT

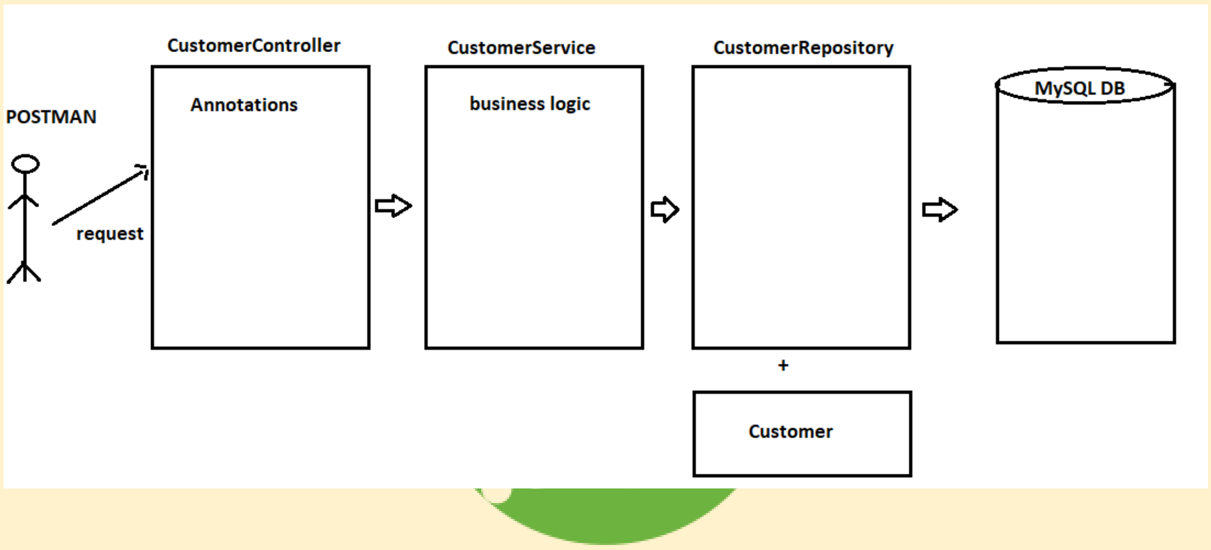
theguardian

NORDSTROM

and etc.

Spring boot flow layered Architecture

Diagram: sb4.3



Note:

create a schema in mysql database.

ex:

create schema demo18;

project structure

```
customer-service
|
|----src/main/java
|        |
          |---com.ihub.www(base package)
                    |
                    |---CustomerServiceApplication.java

          |
          |---com.ihub.www.controller
                    |
                    |---CustomerController.java

          |---com.ihub.www.service
                    |
                    |---CustomerService.java

          |---com.ihub.www.repo
                    |
                    |----CustomerRepository.java(Interface)

          |---com.ihub.www.model
                    |
                    |----Customer.java

|-----src/main/resources
|                   |
                    |---application.yml
|
|----pom.xml
```

step1:

Create a spring starter project i.e customer-service.

ex:

```
        staters:  spring web
                  spring data jpa
                  mysql driver
                  lombok
```

step2:

Create Customer model class inside "com.ihub.www.model" package

ex:

```
package com.ihub.www.model;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Table
@Data
```

```
@NoArgsConstructor
@AllArgsConstructor
public class Customer
{
        @Id
        private int custId;

        @Column
        private String custName;

        @Column
        private String custAdd;
}
```

step3:

Create CustomerRepository interface inside "com.ihub.www.repo" package

ex:

```
package com.ihub.www.repo;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import com.ihub.www.model.Customer;
```

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Integer>
{


}
```

step4:

Create a "CustomerController" inside "com.ihub.www.controller" package.

ex:

```
package com.ihub.www.controller;

import java.util.List;

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

import com.ihub.www.model.Customer;
import com.ihub.www.service.CustomerService;
```

```
@RestController
@RequestMapping("/customer")
public class CustomerController
{

        @Autowired
        CustomerService customerService;

        @PostMapping("/add")
        public Customer addCustomer(@RequestBody Customer customer)
        {
                return customerService.addCustomer(customer);
        }

        @GetMapping("/fetch")
        public List<Customer> getAllCustomer()
        {
                return customerService.getAllCustomer();
        }

        @GetMapping("/fetch/{custId}")
        public Customer getCustomer(@PathVariable int custId)
        {
                return customerService.getCustomer(custId);
        }
```

```
	@PutMapping("/edit")
	public String updateCustomer(@RequestBody Customer customer)
	{
		return customerService.updateCustomer(customer);
	}

	@DeleteMapping("/delete/{custId}")
	public String deleteCustomer(@PathVariable int custId)
	{
		return customerService.deleteCustomer(custId);
	}
}
```

step5:

Create a "CustomerService" inside "com.ihub.www.service" package.

ex:

```
package com.ihub.www.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;

import com.ihub.www.model.Customer;
import com.ihub.www.repo.CustomerRepository;

@Service
public class CustomerService
{
        @Autowired
        CustomerRepository customerRepository;

        public Customer addCustomer(Customer customer)
        {
                return customerRepository.save(customer);
        }
        public List<Customer> getAllCustomer()
        {
                return customerRepository.findAll();
        }

        public Customer getCustomer(int custId)
        {
                return customerRepository.findById(custId).get();
        }

        public String updateCustomer(Customer customer)
        {
        Customer
cust=customerRepository.findById(customer.getCustId()).get();
```

```
        cust.setCustName(customer.getCustName());

        cust.setCustAdd(customer.getCustAdd());

        customerRepository.save(cust);

        return "Record is updated";
        }

        public String deleteCustomer(int custId)
        {
                customerRepository.deleteById(custId);
                return "Record is deleted";
        }
}
```

step6:

Configure msql driver and server port inside application.yml.

ex:

application.yml file

```
server:
  port: 9191



spring:
  application:
    name: CUSTOMER-SERVICE

  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/demo17
    username: root
    password: root
  jpa:
    hibernate.ddl-auto: update
    generate-ddl: true
    show-sql: true
```

step7:

Test the applications by using below rest api's.

REST API **HTTP Method**

http://localhost:9191/customer/add POST

```
{

  "custId": 101,
  "custName": "Lisa",
  "custAdd":"Chicago"
}
```

http://localhost:9191/customer/fetch GET

http://localhost:9191/customer/fetch/101GET

http://localhost:9191/customer/edit PUT

```
{

   "custId": 102,
   "custName": "Linda",
   "custAdd":"Vegas"
}
```

http://localhost:9191/customer/delete/102 DELETE

<u>Exception Handling in Spring Boot</u>

If we give/pass wrong request to our application then we will get Exception.

ex:

http://localhost:9090/fetch/102

Here '102' record is not available so immediately our controller will throw below exception.

ex:

```
{
   "timestamp": "2021-02-14T06:24:01.205+00:00",
   "status": 500,
   "error": "Internal Server Error",
   "path": "/fetch/102"
}
```

Handling exceptions and errors in APIs and sending the proper response to the client is good for enterprise applications.

In Spring Boot Exception handling can be performed by using Controller Advice.

@ControllerAdvice

The @ControllerAdvice is an annotation is used to to handle the exceptions globally.

@ExceptionHandler

The @ExceptionHandler is an annotation used to handle the specific exceptions and sending the custom responses to the client.

project structure

```
customer-service
|
|----src/main/java
|       |
        |---com.ihub.www
                |
                |---CustomerServiceApplication.java


        |
        |---com.ihub.www.controller
                |
                |---CustomerController.java
```

```
          |---com.ihub.www.service
                    |
                    |---CustomerService.java

          |---com.ihub.www.repo
                    |
                    |----CustomerRepository.java(Interface)

          |---com.ihub.www.model
                    |
                    |----Customer.java


|         |---com.ihub.www.exception
|                   |
|                   |---ErrorDetails.java(POJO)
|                   |---ResourceNotFoundException.java
|                   |---GlobalExceptionHandler.java

|-----src/main/resources
|                   |
                    |---application.properties
|
|----pom.xml
```

step1:

Use the existing project i.e customer-service.

step2:

Create a com.ihub.www.exception package inside "src/main/java".

step3:

Create ErrorDetails.java file inside "com.ihub.www.exception" pkg.

ErrorDetails.java

```
package com.ihub.www.exception;
import java.util.Date;
public class ErrorDetails
{
	private Date timestamp;
	private String message;
	private String details;


	public ErrorDetails(Date timestamp, String message, String details) {
		super();
		this.timestamp = timestamp;
		this.message = message;
		this.details = details;
	}

	public Date getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(Date timestamp) {
        this.timestamp = timestamp;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public String getDetails() {
        return details;
    }
    public void setDetails(String details) {
        this.details = details;
    }
}
```

step4:

Create ResourceNotFoundException.java file inside "com.ihub.www.exception" pkg.

ResourceNotFoundException.java

```
package com.ihub.www.exception;

public class ResourceNotFoundException extends RuntimeException
{
        public ResourceNotFoundException(String msg)
        {
                super(msg);
        }
}
```

step5:

Create a GlobalExceptionHandler.java file inside "com.ihub.www.exception" pkg.

GlobalExceptionHandler.java

```
package com.ihub.www.exception;

import java.util.Date;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.context.request.WebRequest;

@ControllerAdvice
public class GlobalExceptionHandler
{

        @ExceptionHandler(ResourceNotFoundException.class)
        public ResponseEntity<?> handleResourceNotFoundException
        (ResourceNotFoundException exception,WebRequest request )

        {

                ErrorDetails errorDetails=new ErrorDetails(new
Date(),exception.getMessage(),request.getDescription(false));

                return new
ResponseEntity<>(errorDetails,HttpStatus.NOT_FOUND);

        }


        //handle global exception

                @ExceptionHandler(Exception.class)

                public ResponseEntity<?> handleException

                (Exception exception,WebRequest request )

                {

                        ErrorDetails errorDetails=new ErrorDetails(new
Date(),exception.getMessage(),request.getDescription(false));

                        return new
ResponseEntity<>(errorDetails,HttpStatus.INTERNAL_SERVER_ERROR);

                }

}
```

step6:

Now add ResourceNotFoundException to CustomerService.

CustomerService.java

```
package com.ihub.www.service;

import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.ihub.www.exception.ResourceNotFoundException;
import com.ihub.www.model.Customer;
import com.ihub.www.repo.CustomerRepository;

@Service
public class CustomerService
{
	@Autowired
	CustomerRepository customerRepository;

	public Customer addCustomer(Customer customer)
	{
		return customerRepository.save(customer);
	}
	public List<Customer> getAllCustomer()
	{
		return customerRepository.findAll();
	}

	public Customer getCustomer(int custId)
	{
		return customerRepository.findById(custId)
```

```
                                        .orElseThrow(()-> new
ResourceNotFoundException("ID NOT FOUND"));

          }

          public Customer updateCustomer(Customer customer)
          {
                    customer.setCustName("Lisa");
                    customer.setCustAddrs("Chicago");
                    return customerRepository.save(customer);
          }

          public String deleteCustomer(int custId)
          {
                    Customer customer=customerRepository.findById(custId)
                    .orElseThrow(()->new ResourceNotFoundException("Id Not
Found for Delete"));

                    customerRepository.delete(customer);

                    return "Record is deleted";
          }
}
```

step7:

Relaunch the spring boot application.

step8:

Test the application by using below request url.

ex:

http://localhost:9090/fetch/102

step9:

Here exception will display in below format.

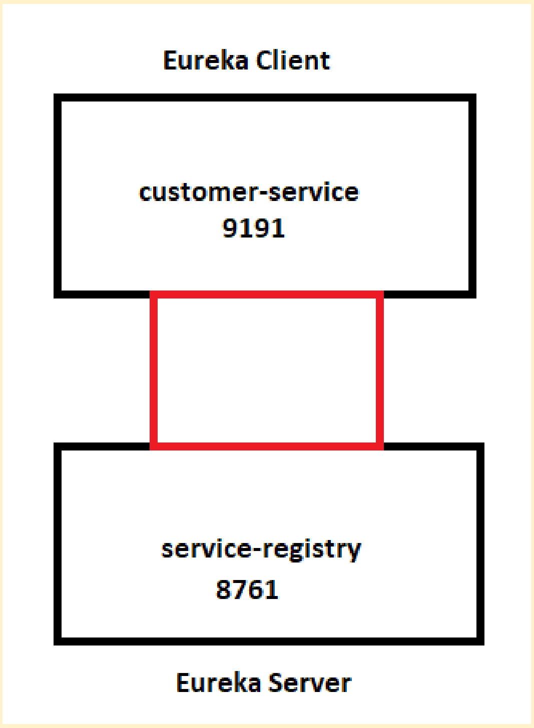
ex:

```
{
        "timestamp": "2023-03-27T23:04:03.181+00:00",
        "message": "ID NOT FOUND",
        "details": "uri=/fetch/102"
}
```

Eureka Server

- This server holds information about the client service applications.
- Each microservice registers into Eureka server and eureka server knows all client applications running on each port and IP address.
- This server is also known as discovery server.

Diagram: sb5.1



step1:

Add Eureka Client dependency in "customer-service" project.

ex:

starter

Eureka Discovery client.

step2:

Create a "service-registry" project to register all microservices.

Here "service-registry" is a Eureka Server and microservices are Eureka Clients.

> service-registry

starter

> Eureka Server.

step3:

Add "@EnableEurekaServer" annotation in main spring boot application.

ServiceRegisterApplication.java

```
package com.ihub;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer
public class ServiceRegisterApplication {

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

}
```

step4:

Add port number and set register for Eureka service as false.

application.yml

```
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
```

step5:

Open the "customer-service" application.yml and add register with eureka as true.

application.yml

```
server:
  port: 9001


spring:
  application:
    name: CUSTOMER-SERVICE

  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/demo
    username: root
    password: root
  jpa:
    hibernate.ddl-auto: update
    generate-ddl: true
    show-sql: true
eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    hostname: localhost
```

step6:

Now run all two projects.

First run service-registry then customer-service.

First run eureka server then eureka client.

step7:

Check the output in below url's.

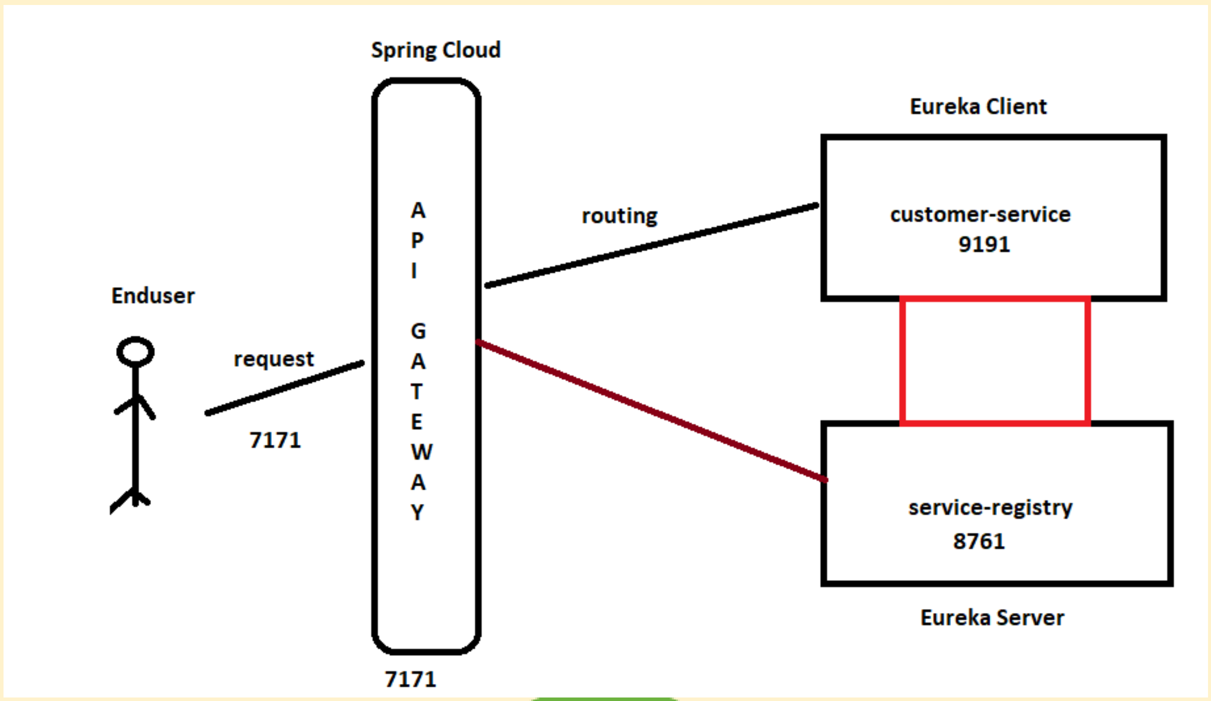
ex:

http://localhost:8761/

Spring Cloud API Gateway

- **Spring Cloud Gateway aims to provide a simple, effective way to route to API's and provide cross cutting concerns to them such as security,monitoring/metrics , authentication, autherization ,adaptor and etc.**

Diagram: sb5.2



step1:

Create a "cloud-apigateway" project in STS.

starters:

eureka Discovery client

Spring boot actuators

spring reactive web

step2:

Add spring cloud dependency in pom.xml file.

ex:

```
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-gateway</artifactId>
	<version>3.1.1</version>
	</dependency>
```

step3:

Add "@EnableEurekaClient" annotation on main spring boot application.

CloudApigatewayApplication.java

```java
package com.ge;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

@SpringBootApplication
@EnableEurekaClient
public class CloudApigatewayApplication {

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

}
```

step4:

Register port number, set application name,and configure

all microservices for routing in application.yml file.

application.yml

```yaml
server:
  port: 7171

eureka:
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    hostname: localhost

spring:
  application:
    name: API-GATEWAY
  cloud:
    gateway:
      routes:
        - id: CUSTOMER-SERVICE
          uri: lb://CUSTOMER-SERVICE
          predicates:
            - Path=/customer/**
```

step5:

Now Run the following applications sequentially.

"service-registry"

"customer-service"

"cloud-apigateway".

step6:

Test the applications by using below urls.

ex:

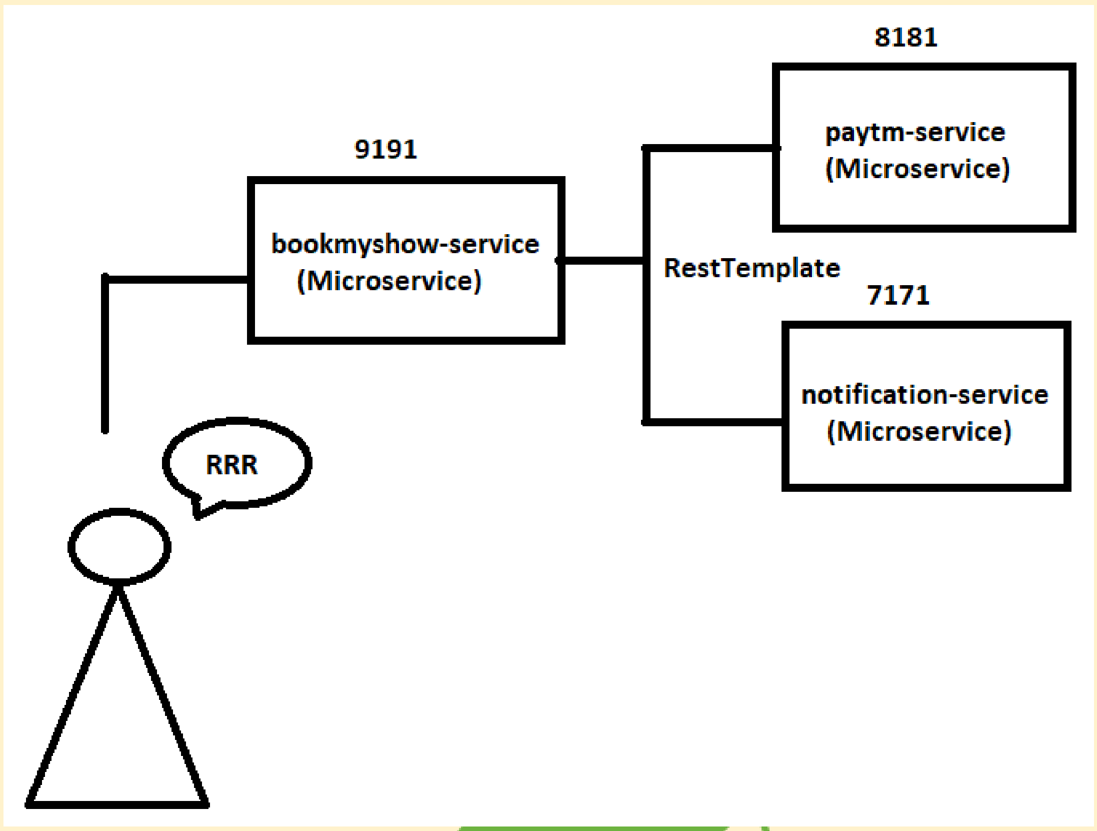
http://localhost:9191/customer/fetch/101

http://localhost:9191/customer/fetch

Spring Cloud Hystrix

- Hystrix is a fault tolerance library provided by Netflix.
- Using Hystrix we can prevent Deligation of failure from one service to another service.
- Hystrix internally follows Circuit Breaker Design pattern.
- In short circuit breaker is used to check availability of external services like web service call,database connection and etc.

Diagram: sb6.1


8181
paytm-service (Microservice)
9191
bookmyshow-service (Microservice)
RestTemplate
7171
notification-service (Microservice)
RRR

notification-service

step1:

create a "notification-service" project in STS.

Starter:

Spring Web.

step2:

Add the following code in main sprping boot application.

<u>NotificationServiceApplication.java</u>

```
package com.ihub.www;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
@RequestMapping("/notification")
public class NotificationServiceApplication {

	@GetMapping("/send")
	public String sendEmail()
	{
		return "Email sending method is called from notification-
service";
	}
	public static void main(String[] args) {
		SpringApplication.run(NotificationServiceApplication.class, args);
	}
```

}

step3:

convert application.properties file to application.yml file.

step4:

configure server port number in application.yml file.

application.yml

```
server:
  port: 7171
```

step5:

Run "notification-service" project as spring boot application.

step6:

Test the application with below request url.

ex:

http://localhost:7171/notification/send

paytm-service

step1:

create a "paytm-service" project in STS.

Starter:

Spring Web.

step2:

Add the following code in main sprping boot application.

PaytmServiceApplication.java

```
package com.ihub.www;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
@RestController
@RequestMapping("/paytm")
public class PaytmServiceApplication {

	@GetMapping("/pay")
	public String paymentProcess()
	{
		return "Payment Pocess method called in paytm-service";
	}



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

```
    }

}
```

step3:

convert application.properties file to application.yml file.

step4:

configure server port number in application.yml file.

application.yml

```
server:
  port: 8181
```

step5:

Run "paytm-service" project as spring boot application.

step6:

Test the application with below request url.

ex:

http://localhost:8181/paytm/pay

bookmyshow-service

step1:

create a "bookmyshow-service" project in STS.

Starter:

Spring Web

step2:

Add Spring Cloud Hystrix dependency in pom.xml file.

ex:

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

step3:

Change <parent> tag inside pom.xml file for hystrix compatability.

ex:

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.3.RELEASE</version>
        <relativePath /> <!-- lookup parent from repository -->
</parent>
```

step4:

Add the following code in main spring boot application.

BookmyshowServiceApplication

```
package com.ihub.www;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.hystrix.EnableHystrix;
import org.springframework.context.annotation.Bean;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;

@SpringBootApplication
@RestController
@EnableHystrix
public class BookmyshowServiceApplication {

	@Autowired
	RestTemplate restTemplate;

	@HystrixCommand(groupKey = "ihub" , commandKey = "ihub"
,fallbackMethod = "bookMyShowFallBack")
	@GetMapping("/book")
	public String bookShow()
	{
```

```
                String
paytmServiceResponse=restTemplate.getForObject("http://localhost:8181/p
aytm/pay", String.class);

                String
notificationServiceResponse=restTemplate.getForObject("http://localhost:71
71/notification/send",String.class);


                return
paytmServiceResponse+"\n"+notificationServiceResponse;

        }



        public static void main(String[] args) {

                SpringApplication.run(BookmyshowServiceApplication.class,
args);


        }


        public String  bookMyShowFallBack()

        {

                return "service gateway failed";

        }


        @Bean

        public  RestTemplate    getRestTemplate() {


                return new RestTemplate();
```

```
          }

}
```

step5:

convert application.properties file to application.yml file.

step6:

configure server port number inside application.yml file.

application.yml

```
server:
  port: 9191
```

step7:

Add spring core dependency inside pom.xml file.

ex:

```
<dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-core</artifactId>
          <version>5.3.17</version>
</dependency>
```

step8:

Run the "bookmyshow-service" application as spring boot application.

step9:

Test the application by using below request url.

ex:

http://localhost:9191/book

step10:

Now stop any micro service i.e notification-service or paytm-service.

step11:

Test the "bookmyshow-service" application by using below url.

ex:

http://localhost:9191/book

Note:

Here fallback method will execute with the help of Hystrix.

Spring Security

- Spring Security is a framework which provides various security features like authentication, authorization to create secure Java Enterprise Applications.
- This framework targets two major areas of application i.e authentication and authorization.

Authentication

It is a process of knowing and identifying the user that wants to access.

Authorization

It is a process to allow authority to perform actions in the application.

Project structure

```
SBSpringSecurity
|
|----src/main/java
|       |
|        |----com.ge.www
|                |
|                |--SBSpringSecurityApplication.java
|        |
|        |----com.ge.www.controlller
|                |
|                |--MyController.java
|
|---src/main/resources
|        |
|        |-----application.properties
|
|---src/test/java
|        |
|        |-----SBSpringSecurityApplicationTests.java

|---pom.xml
|
|
```

step1:

create a spring starter project.

starters:

spring web

spring security.

step2:

create a Controller to accept the request.

MyController.java

```
package com.ge.www.controller;

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

@RestController
public class MyController {

	@GetMapping("/msg")
	public String msg()
	{
		return "Welcome to Spring Security";
	}
}
```

step3:

Configure server port number in application.properties file.

application.properties

```
server.port=9191
```

step4:

Run the application as spring boot application.

step6:

Test the application by using below url.

ex:

http://localhost:9191/msg

Note:

When we hit the request ,we will get login page.

Default username is "user" and password we can copy from STS console.

step7:

To change the default user and password we can use below propertiesin application.properties file.

application.properties

```
server.port=9191
spring.security.user.name=raja
spring.security.user.password=rani
```

step8:

Relaunch the spring boot application.

step9:

Test the application by using below url.

ex:

http://localhost:9191/msg