Limitations with Spring Framework

==========

In spring framework , a developer is responsible for following things.

1) Add dependencies / jar files

2) Add Configurations i.e applicationContext.xml

3) Arranging the virtual server like Tomcat , Weblogic and etc to deploy the web application.

4) Managing the physical database like Oracle, MySQL and etc to communicate with database.

To overcome these limitations we need to use Spring Boot.

Spring Boot

==========

Spring Boot is a java based application framework developed by Pivotal Team.

It provides RAD(Rapid Application Development) features for spring based applications.

It is an open source , production ready grade spring based applications with minimum configurations.

In short , spring boot is a combination of

ex:

spring framework + embedded server + embedded database.

Diagram:

Developer

|

|

Spring Boot

|

|

Spring Framework

Advantages of Spring Boot

==========

> It creates standalone application which can be run by using java -jar.

> It provides production ready grade features like metrics, healthcheck, externalized configuration and etc.

> It provides optionate stater's POM's to simply the MAVEN configuration.

> It does not support XML configurations. Instead we will use Annotations.

> It allows us to test our web applications by using various HTTP servers like Tomcat.We don't need to convert to war file.

> It uses CLI (Command Line Interface) tool for creating and testing the spring boot project.

> It minimize the boiler plate codes and xml configurations.

> It supports number of plugin's.

> It increases productivity and reduces development time.

List of companies are working with spring boot

==========

We have following list of companies working with spring boot.

1. Netflix
2. Alibaba
3. LinkedIn
4. Uber
5. Groupon
6. Zillow
7. Intuit.
8. welmart and etc.

Interview Questions

==========

Q) What is spring boot?

It is a java based application framework developed by pivotal team.It provides RAD features for spring based applications.It is a standalone , production ready grade spring based applications with minimum configurations.

Q)How many components are there in spring boot?

We have four components in spring boot.

1) AutoConfiguration

2) Starter

3) CLI tool

4) Actuator

Q) List out embedded servers present in spring boot?

We have following embedded servers.

ex:

Tomcat , Jetty , Undertow

Q) List out embedded databases present in spring boot?

We have following embedded databases.

ex:

H2, HSQL , Derby

Q) Where we will do configurations in spring boot?

We can perform spring boot configurations in two files.

ex:

application.properties

application.yml

Q) List out some stereotype annotations?

We have following list of stereotype annotations.

ex:

@Controller

@Service

@Repository

@Component

and etc.

Q) Which annotation is used to create a controller in spring boot?

@Controller annotation

Q) Which annotation is used to create a service in spring boot?

@Service annotation

Q) Which annotation is used to create a repository in spring boot?

@Repository annotation

Q) How many ways we can develop spring boot project?

There are two ways to develop spring boot project.

1) Using Spring Initializr

2) Using IDE's (Spring Tool Suit(STS) / Intellij)

Spring Initializr

==========

Spring initializr is a based based tool provided by pivotal web services.

Using spring initializr we can generate create basic structure of a project but it won't add application code.

It offers extensiable API for JVM based projects.

To create a project structure using spring initializr we need to use below url.

ex:

https://start.spring.io/

Steps to create a spring boot project using spring initializr

---

step1:

Goto spring initializr web based tool.

ex:

https://start.spring.io/

step2:

Create a spring boot project i.e SBApp1.

ex:

Project : Maven

Language : Java

Dependencies : (no dependencies)

Spring Boot : 3.2.1

Project Metadata

Group : com.ihub.www

Artifact : SBApp1

Name : SBApp1

description : First Spring Boot Project

package : com.ihub.www

packaging : jar

Java : 17

---> click on generate button.

step3:

Download and Extract STS IDE.

step4:

Launch STS IDE by choosing workspace location.

step5:

Exract SBApp1 project and open the project in STS IDE.

ex:

import projects --> Maven --> Existing Maven project --> Next -->

Root directory --> SBApp1 --> select folder --> finish.

step6:

Add some code in SBApp1Application.java file.

ex:

System.out.println("I Love Spring Boot");

step7:

Run spring boot project.

ex:

right click on SBApp1 --> run as --> spring boot App.

STS - Spring Tool Suit

==========

STS is an IDE's to develop spring boot applications.

It provides eclipse-based environment.

It provides ready to use environment for implement, run, develop, deploy spring applications.

Steps to work with STS IDE

---

step1:

Launch STS IDE.

step2:

Create a spring boot starter project.

ex:

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

Name : SBApp2

Type : Maven

packaging : jar

Java version : 17

Language : JAVA

Group : com.ihub.www

Artifact : SBApp2

description : My second spring boot project

package : com.ihub.www

---> next --> next --> finish.

step3:

Write simple statement in SBApp2Application.java file.

ex:

System.out.println("This is my second project");

step4:

Run spring boot project.

ex:

Right click to SBApp2 --> run as --> spring boot application.

Q) What is @SpringBootAnnotation in spring boot?

@SpringBootAnnotation is a combination of three annotations.

1) @EnableAutoConfiguration : It enables the auto configuration mechanism.

2) @ComponentScan : It is used to scan the package where application is located.

3) @Configuration : It is used to register extra beans in context and add additional configrations.

Spring Boot Architecture

==========

Spring Boot follows layered architecture.

Before we goto spring boot layered architecture, we will see how many layers present in spring boot.

We have following four layers in spring boot where one layer interacts with another layer.

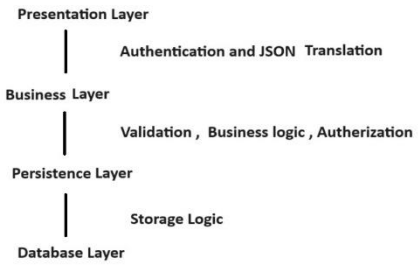
1) Presentation Layer

2) Business Layer

3) Persistence Layer

4) Database Layer

Diagram: sb2.1



1)Presentation Layer:

The presentation layer handles the HTTP requests, translates the JSON parameter to object, and authenticates the request and transfer it to the business layer.

2)Business Layer:

The business layer handles all the business logic. It consists of service classes and uses services provided by data access layers. It also performs authorization and validation.

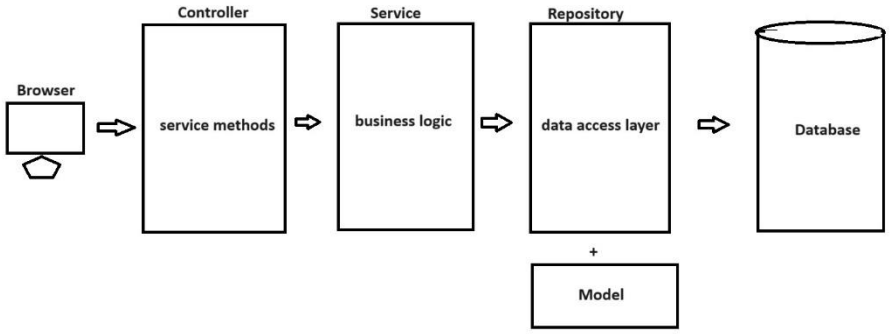
3)Persistence Layer:

The persistence layer contains all the storage logic and translates business objects from and to database rows.

4)Database Layer:

In the database layer, CRUD (create, retrieve, update, delete) operations are performed.

Diagram: sb2.2



With respect to the Diagram

The client makes the HTTP requests (PUT or GET).

The request goes to the controller, and the controller maps that request and handles it. After that, it calls the service logic if required.

In the service layer, all the business logic performs. It performs the logic on the data that is mapped to JPA with model classes.

A JSP page is returned to the user if no error occurred.

Spring Boot starters

==========

Spring Boot starters provides number of starters to add jar file in CLASSPATH.

Spring Boot built-in starters makes our development easier and rapid.

Spring Boot starters are 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

SBApp3

|

|----src/main/java

| |

| |----com.ihub.www (base package)

| |

```
|                   |--SBApp3Application.java
|                   |--HomeController.java
|---src/main/resources
|         |
|         |-----application.properties
|
|---src/test/java
|         |
|         |-----SpringBootApp3ApplicationTests.java
|
| --
| --
| --
|-----src
          |
          |----main
                    |
                    |----webapp
                              |
                              |----pages
                              |         |
                                        |----index.jsp
|---pom.xml
|
|
```

step1:

Create a spring starter project.

ex:

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

Name : SBApp3

Group: com.ihub.www

Artifact: SBApp3

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=9090

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

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.

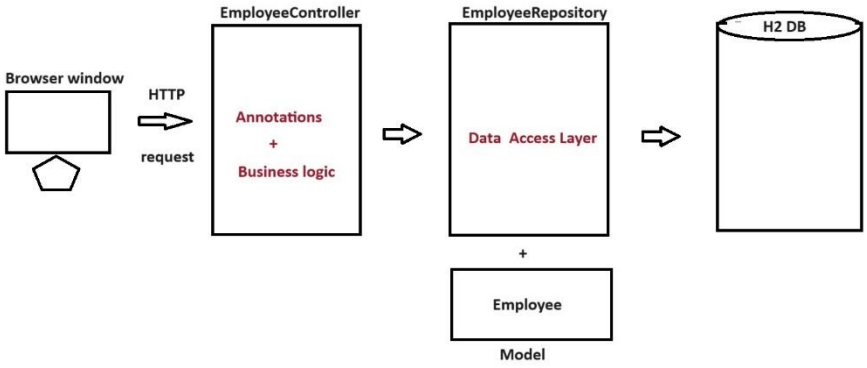
JpaRepository:

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.

Spring Boot application to interact with H2 Database

==========

Diagram: sb3.1



project structure

SBApp4

|

|----src/main/java

|

|---com.ihub.www

| |

|---SBApp4Application.java

|

|---com.ihub.www.controller

|

|---EmployeeController.java (Class)

|

|---com.ihub.www.repository

|

|---EmployeeRepository.java (Interface)

```
        |
        |---com.ihub.www.model
                |
                |---Employee.java (Class)
|
|
|----src/main/resources
        |
        |---application.properties
        |

|-----src
        |
        |-----main
                |
                |---webapp
                        |
                        |----index.jsp
```

step1:

Create a spring boot starter project i.e SBapp4.

ex:

starters:

spring web

spring data jpa

H2 Database

step2:

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

ex:

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

step3:

Create a EmployeeController inside "com.ihub.www.controller" 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 addEmployeeDetails(Employee e)
	{
		employeeRepository.save(e);

		return "index.jsp";
	}
}
```

step4:

Create index.js file inside "src/main/webapp" folder.

index.js

```
<form action="addEmp">
	<table align="center">
		<caption>Enter the Details</caption>
		<tr>
			<td>Employee Id </td>
			<td><input type="text" name="empId"/></td>
		</tr>
		<tr>
			<td>Employee Name </td>
			<td><input type="text" name="empName"/></td>
		</tr>
		<tr>
			<td>Employee Salary </td>
			<td><input type="text" name="empSal"/></td>
		</tr>
		<tr>
			<td><input type="reset" value="reset"/></td>
			<td><input type="submit" value="submit"/></td>
		</tr>
	</table>
</form>
```

step5:

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
public class Employee
{
	@Id
	private int empId;
	@Column
	private String empName;
	@Column
	private double empSal;

	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 double getEmpSal() {
		return empSal;
	}
	public void setEmpSal(double empSal) {
		this.empSal = empSal;
	}
}
```

step6:

Create a EmployeeRepository.java interface inside "com.ihub.www.repository" package.

EmployeeRepository.java

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

import org.springframework.data.repository.CrudRepository;
```

```
import org.springframework.stereotype.Repository;

import com.ihub.www.model.Employee;

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

}
```

step7:

Configure server port and h2 database properties inside application.properties file.

application.properties

```
server.port=9090

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

step8:

Run the spring boot starter project.

step9:

Test the application by using below request url.

ex:

http://localhost:9090

http://localhost:9090/h2-console

RestController:

==========

RestController is used for making restful web services with the help of the @RestController annotation.

This annotation is used at the class level and allows the class to handle the requests made by the client.

The main difference between the @RestController and the @Controller is that the @Restcontroller

is a combination of the @controller and @ResponseBody annotation.

We have following HTTP methods along with rest annotations.

@RestController

HTTP Methods	Annotations
GET	@GetMapping
POST	@PostMapping
PUT	@PutMapping
Delete	@DeleteMapping
and etc.	

@Controller

Http methods	Annotations
GET	@RequestMapping
POST	@RequestMapping
PUT	@RequestMapping
DELETE	@RequestMapping

Spring Boot Application using @RestController

==========

Project structure

RestApp

|

|----src/main/java

| |

| |----com.ihub.www

| |

| |--RestAppApplication.java

| |--HomeController.java

|---src/main/resources

| |

| |-----application.properties

|

|---src/test/java

| |

| |-----RestAppApplicationTests.java

|

| --

| --

| --

|---pom.xml

|

|

step1:

Create a spring starter project.

ex:

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

Name : RestApp

Group: com.ihub.www

Artifact: RestApp

Description: This is Spring Boot Application

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

@RestController
public class HomeController {

	@GetMapping("/")
	public String home()
	{
		return "Rest Controll Example";
	}
}
```

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/

Q)Difference between Monolethic Architecture vs Microservice Architecture?

Monolithic Architecture

==========

Monolith means composed all in one piece.

The Monolithic application describes a single-tiered software application in which different components combined into a single program from a single platform.

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.

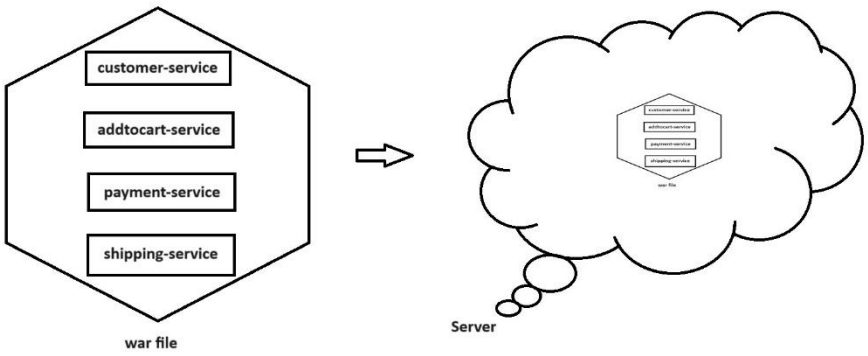
Lets take an example of E-commerce website where we have basic and common option of Customer

Service, Product Service and Cart Service, which a customer can access through browser.When we

launch the application It is deployed as a single monolithic application.It means we will have

only one single instance.

Diagram: sb4.1



Advantages

==========

1)Simple to develop

2)Simple to test

3)Simple to deploy

4)Simple to scale

Drawbacks of Monolithic Architecture

==========

1)Large and Complex Application

2)Slow Development

3)Blocks Continenous development

4)Unscalable

5)Unreliable

6)Inflexible

MicroService Architecture

==========

Microservices are the small services that work together

The microservice defines an approach to the architecture that divides an application into a pool of loosely coupled services that implements business requirements.

In Microservice architecture,Each service is self contained and implements a single bussiness capability.

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.

Advantages of Microservice Architecture

==========

1)Independent Development

Each microservice can be developed independently.

A single development team can build test and deploy the service.

2)Independent Deployment

---

we can update the service without redeploying the entire application.

Bug release is more managable and less risky.

3)Fault Tolerance

If service goes down ,It won't take entire application down with it.

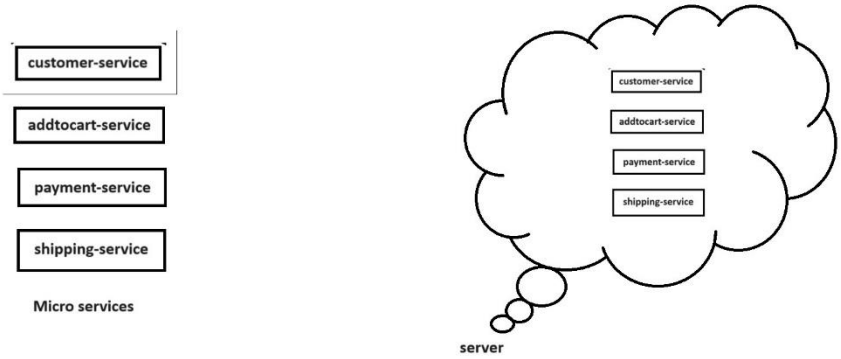
4)Mixed Technology Stack

It is used to pick best technology which best suitable for our application.

5)Granular Scaling

In Granular scaling ,services can scaled independently.Instead of entire application.

Diagram: sb4.2



List of companies working with micro-services

==========

We have following companies working with microservices.

1. Netflix
2. Amazon
3. Uber
4. eBay
5. SoundCloud
6. Karma

7. Groupon.

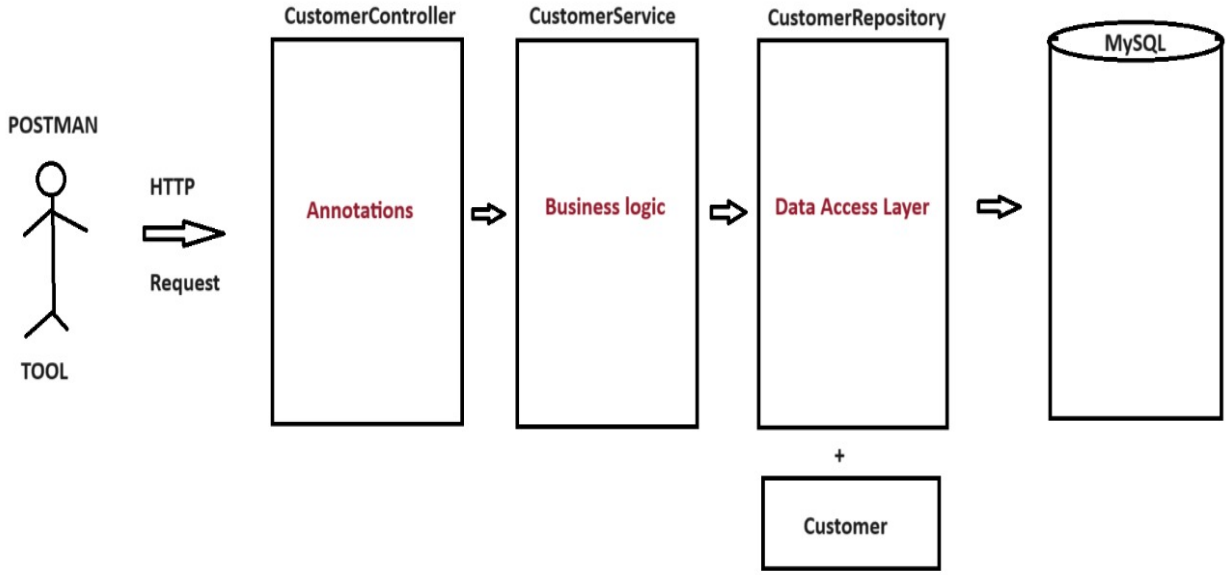
and etc.

customer micro service

==========

To develop any micro service we need to follow spring boot flow layered architecture.

Diagram: sb4.3



Project structure

---

customer-service

|

|-----src/main/java

| |

  |-----com.ihub

  | |

  | |----CustomerMicroserviceApplication

  |

  |-----com.ihub.controller

  | |

  | |----CustomerController.java (controller class)

  |

  |-----com.ihub.entity

  | |

  | |----Customer.java (POJO class)

  |

  |-----com.ihub.service

  | |

  | |----CustomerService.java (service class)

  |

  |-----com.ihub.repository

  | |

  | |----CustomerRepository.java (interface)

  |

|-------src/main/resources

| |

| |-----application.yml

| |

|

|-------pom.xml

|

step1:

Create a "customer-service" project.

starters:

spring reactive web

spring Data JPA

Lombok

mysql driver

step2:

Download and Install project lombok.

ex:

https://projectlombok.org/download

step3:

Create a Customer Model class.

Customer.java

```
package com.ihub.entity;

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

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Customer {

	@Id
	@Column(length =6)
	private int custId;

	@Column(length =12)
	private String custName;
```

```
	@Column(length=12)
	private String custAddress;

}
```

step2:

Create a "demo" schema inside mysql database.

ex:

```
MYSQL> create schema demo;

MYSQL> use demo;
```

step3:

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

CustomerRepository.java

```
package com.ihub.repository;

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

import com.ihub.entity.Customer;
```

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

}
```

step4:

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

CustomerController.java

```
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.RequestMapping;
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> getAllCustomers()
	{
		return  customerService.getAllCustomers();
	}

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

```
	@PutMapping("/update")
	public Customer 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.java" file inside "com.ihub.www.service" package.

CustomerService.java

```
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.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 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> getAllCustomers()
	{
		return  customerRepository.findAll();
	}
```

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

	public Customer updateCustomer(Customer customer)
	{
		Customer
oldCustomer=customerRepository.findById(customer.getCustId()).get();

		oldCustomer.setCustName(customer.getCustName());
		oldCustomer.setCustAdd(customer.getCustAdd());

		customerRepository.save(oldCustomer);

		return customerRepository.findById(customer.getCustId()).get();
	}


	public String deleteCustomer(int custId)
	{
		Customer customer=customerRepository.findById(custId).get();

		customerRepository.delete(customer);

		return "Record Deleted ";
	}
```

}

step6:

Configure server port , database properties and hibernate properties in application.yml.

application.yml

```
server:
  port: 9090


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
```

step7:

Run the spring boot application.

step8:

Test the application by using below request url.

METHODS	URL
GET	http://localhost:9001/customer/fetch
GET	http://localhost:9001/customer/fetch/101
POST	http://localhost:9001/customer/add

> body

>raw

```
{
    "custId":101,
    "custName":"Alex",
    "custAdd":"Chicago"
}
```

PUT	http://localhost:9001/customer/update
DELETE	http://localhost:9001/customer/delete/101

Exception Handling in Spring Boot

==========

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

                cust.setCustName(customer.getCustName());
                cust.setCustAdd(customer.getCustAdd());

                customerRepository.save(cust);

                return "Record updated";
        }

        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

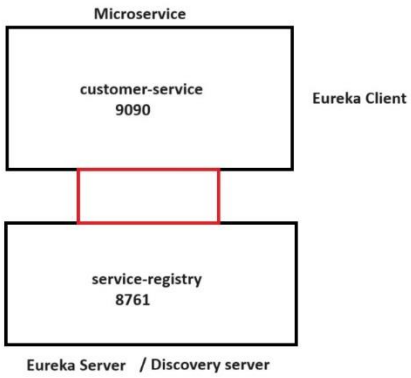
==========

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.

Eureka server is also known as discovery server.

Diagram: sb6.1



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

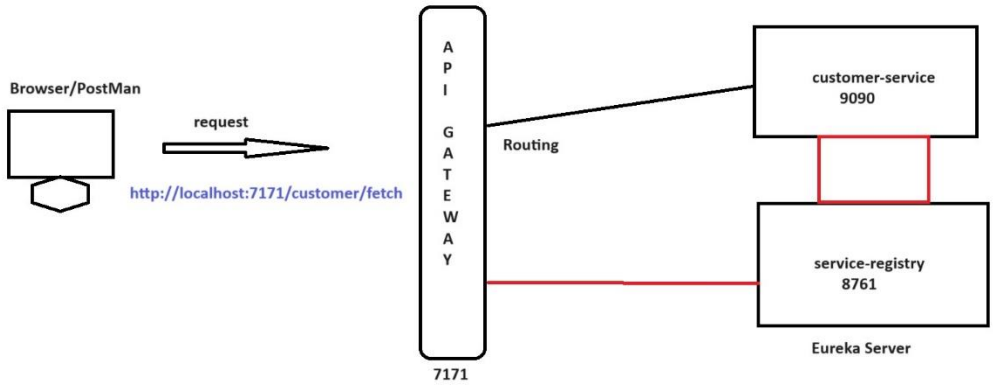
==========

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: sb6.2



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

```
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

```
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

==========

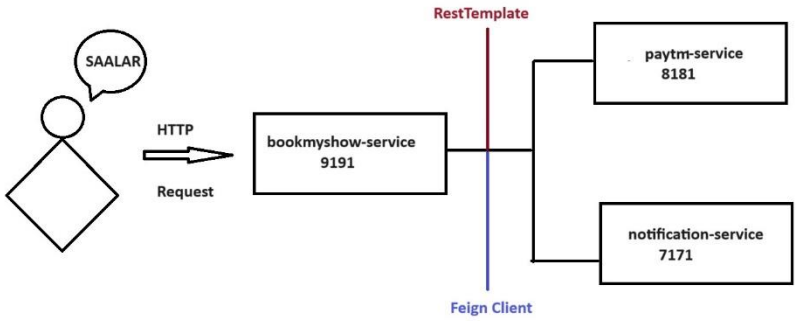
Hystrix is a fault tolerance library provided by Netflix.

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: sb7.1



notification-service

==========

step1:

---

create a "notification-service" project in STS.

Starter:

Spring Web.

step2:

Add the following code in main sprping boot application.

NotificationServiceApplication.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("/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/paytm/pay", String.class);
		String notificationServiceResponse=restTemplate.getForObject("http://localhost:7171/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.

It is a sub-project of Spring framework which was started in 2003 by Ben Alex.

Later on, in 2004, It was released under the Apache License as Spring Security 2.0.0.

This framework targets two major areas of application

1)Authentication

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

2)Authorization

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

Project structure

SpringSecurityApp

```
|
|----src/main/java
|       |
|       |----com.ge.www
|               |
|               |--SpringSecurityAppApplication.java
|       |
|       |----com.ge.www.controllller
|               |
|               |--HomeController.java
|
|---src/main/resources
|       |
|       |-----application.yml
|
|---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.

HomeController.java

```
package com.ge.www.controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HomeController {

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

step3:

Configure server port number in application.properties file.

application.yml

```
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 properties in application.properties file.

application.yml

```
server:
  port: 9191

spring
  security:
    user:
      name=raja
```

password=rani

step8:

Relaunch the spring boot application.

step9:

Test the application by using below url.

ex:

http://localhost:9191/msg

How can we convert spring boot project to jar file

==========

step1:

Make sure spring boot project is ready.

step2:

Create a jar file for spring boot project.

ex:

right click to project --> run as --> Maven build -->

Goals: package --> run.

step3:

Check the jar file inside target folder of a spring boot project.