Limitations with Spring Framework
==========
In spring framework a programmer is responsible for following things.

1) Adding dependencies or jar files.

2) Performing configuration in applicationContext.xml file.

3) Arranging physical server like Tomcat.

4) Managing physical database like Oracle, MySQL and etc.

To overcome above limitations we need to use Spring Boot.

ex:

Developer
|
|
Spring Boot
|
|
Spring Framework

Spring Boot
==========
It is an open source java based application framework developed by Pivotal Team.

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

It is a standalone, production ready grade spring based applications with minimum configurations.

In short, **spring boot is a combination of**

ex:

spring boot ⇒ **Spring framework + Embedded Server + Embedded Database**

Spring Boot does not support xml configurations instead it will use annotations.

Advantages of Spring Boot
==========
> It is used to develop standalone applications which can run by using java -jar.

> It gives production ready grade features like metrics, Health check , Externalized configurations and etc.

> It provides optionate starters to simplify the maven development process.

> It allows us to test web applications by using HTTP servers like Tomcat, Jetty and Undertow.

> It provides in memory databases like H2, HSQL and Derby.

> It does not support xml configurations.

> It contains CLI (Command Line Interface) tool for developing and testing spring boot applications.

Interview Questions
==========

Q) What is the difference between Spring Framework and Spring Boot?

Spring Framework	Spring Boot
It is a leight weight open source framework widely used to develop enterprise applications.	It is built on top of spring framework widely used to develop REST API's.
A programmer is responsible to add dependencies.	A spring boot starter component is responsible to add dependencies.
The main feature of spring framework is dependency injection.	The main feature of spring boot is auto configuration.
It is used to develop loosely coupled applications.	It is used to develop standalone applications.
We need to arrange physical servers to test web applications.	It comes with embedded servers like Tomcat, Jetty and Undertow.
It does not provide in-memory databases.	It provides in-memory databases.

Q) How many components are there in spring boot?
We have four components in spring boot.

1) AutoConfiguration

2) Starters

3) Actuators

4) CLI Tool

Q) What is dependency injection?

A dependency injection is a programming technique which makes our class independent to it's dependencies.

In dependency injection one object gives dependencies of another.

To perform dependency injection in spring boot we will use @Autowired annotation.

If one class need dependency of another class then we need to use dependency injection.

ex:

```
class Recording
{
        -
        - // Recording related logic
        -
}
class Student
{
        @Autowired
        Recording r;
}
```

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

There are two ways to create spring boot project.

1) Using Spring Intializr

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

Q) What is Spring Initializr?

It is a web-based tool that helps developers to create and set up Spring Boot project or structure.

STS IDE
=======
STS stands for Spring Tool Suit.

Website : https://spring.io/tools

First Spring Boot application using STS IDE
==========
Project structure

FirstSB
|
|----src/main/java
	|
	|---com.ihub.www (base package)
		|
		|---FirstSBApplication.java
|
|----src/main/resources
	|
	|---application.properties
|
|----src/test/java
|
|----pom.xml

step1:

	Launch STS IDE by choosing workspace location.

step2:

	Create a spring starter project.
	ex:
		File --> new --> Spring starter project -->
		Name : FirstSB
		Type : Maven
		Packaging : jar
		Java version : 17
		language : java

Group : com.ihub.www
Artifact : FirstSB
Description : Demo project for Spring Boot
package : com.ihub.www

---> Next ---> Next ---> Finish.

step3:

Write a custom message inside FirstSbApplication.java file.

FirstSbApplication.java

```
package com.ihub.www;

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

@SpringBootApplication
public class FirstSbApplication {

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

		System.out.println("Welcome to Spring Boot");
	}

}
```

step4:

Run spring boot starter project.
ex:
Right click to FirstSB --> run as --> spring boot App.

Q) What is @SpringBootApplication annotation?

This annotation is a combination of three annotations.

1) @AutoConfiguration

It is used to enable auto configuration mechanism in spring boot.

2) @ComponentScan

It tells to Spring Boot in which packages we have annotated our classes and should manage by spring boot.

It is used to scan on packages in which our project is located.

3) @Configuration

It is use to register extra beans on context.

Spring Boot Staters
==========

Spring boot contains number of built-in starters to develop applications rapidly and easily.

Spring boot starters are also known as dependency descriptors.

Spring boot starters are used to add jar files in CLASSPATH.

Spring boot built-in starters following below naming patterns.

ex:

spring-boot-stater-*

Here '*' means name of the application.

ex:
---

spring-boot-starter-web
spring-boot-starter-test
spring-boot-starter-data-jpa
spring-boot-starter-security
spring-boot-starer-validation
and etc.

Spring Boot Web Dependency
==========

A spring web dependency is used to develop web applications in spring boot.

There are two important features of spring-boot-starter-web.

1) It is compatible with web applications.

2) It performs autoconfigurations.

Spring web dependency internally uses Spring MVC, REST and Tomcat (Embedded Server).

A spring-boot-starter-web performs following things as a autoconfigurations.

1) DispatcherServlet

2) ErrorPages

3) Static dependent dependencies

4) Servlet container

Second Spring Boot Application Development
==========
SecondSB
|
|---src/main/java
|	|
	|---com.kits.www
		|
		|---SecondSBApplication.java
		|---HomeController.java

|---src/main/resources
	|
	|---application.properties
|
|---src/test/java
|
|
|---pom.xml

step1:

	Create a spring boot starter project i.e SecondSB.
	ex:
		starter:
				Spring Web

step2:

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

HomeController.java

```
package com.ihub.www;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HomeController
{
	//handler methods
	@RequestMapping("/home")
	@ResponseBody
	public String home()
	{
		return "I love spring boot";
	}
}
```

step3:

Configure tomcat server port number in application.properties.

application.properties

```
server.port=9090
```

step4:

Run the spring boot starter project.

step5:

Test the spring boot starter project.

ex:

http://localhost:9090/home

Interview Questions
==========

Q) What is @Controller annotation ?

It is a class level annotation.
It is a stereotype annotation.
This annotation indicates that a particular class serves the role of a controller.

Q) What is @ResponseBody annotation?

It is a method level annotation.
This annotation indicates that spring boot should serialize java object into JSON or XML or Text.

Q) What is @RequestMapping annotation?

It is a class level and method level annotation.

It is used for all kinds of HTTP methods.

It is used to map the request to controller and handler methods.

Spring Data JPA
==========
Spring Data JPA handles most of the complexity of JDBC based database acccess and ORM (Object Relational Mapping).

It reduces the boiler place 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 layer by reducing the effort to the amount that is needed.

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

ex:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-stater-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 follow.

1) CrudRepository

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

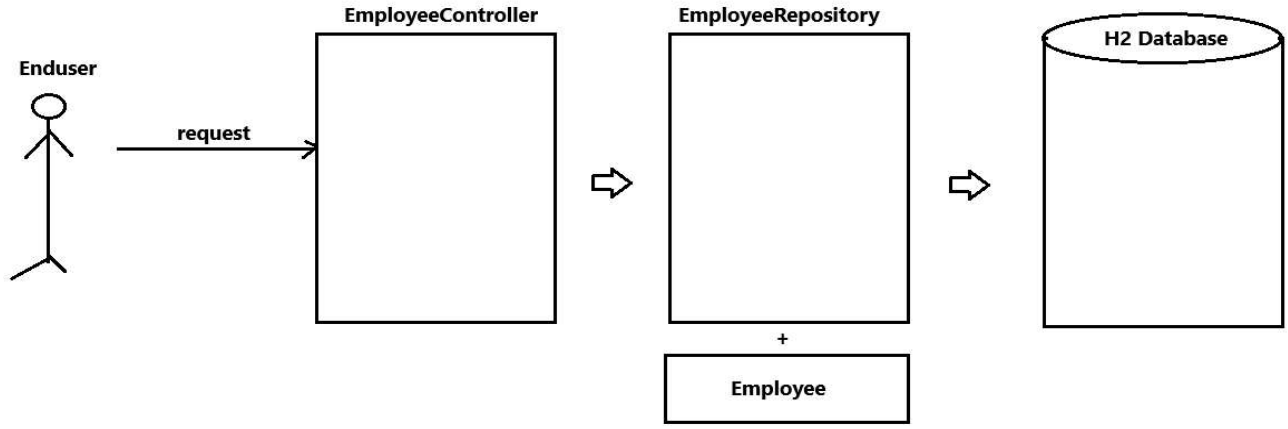
2) PagingAndSortingRepository

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

3) JpaRepository

It is a JPA specific repository and It is defined in spring data jpa.
It extends both CrudRepository and PagingAndSortingRepository.
It adds the JPA-specific methods like flush() to trigger a flush on the persistence context.

Diagram 3.1



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.repository
                |
                |---EmployeeRepository.java (interface)
        |
        |--com.ihub.www.model
                |
                |---Employee.java

|---src/main/resources
        |
        |---application.properties
|
|---src/test/java
|
|------src
        |
        |-----main
                |
                |---webapp (folder)
                        |
                        |---index.jsp
|
|---pom.xml
```

step1:

Launch STS IDE by choosing workspace location.

step2:

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

ex:

staters:

Spring Web
Spring Data JPA
H2 Database

step3:

Add Tomcat Jasper dependency inside pom.xml file.

ex:

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

step4:

Create a "index.jsp" file inside "src/main/webapp" folder.

index.jsp

```
<center>
    <h1 style="text-decoration:underline"> Enter the Details </h1>

    <br>

    <form action="addEmp">

        <table>
            <tr>
                <td>Id:</td>
                <td><input type="text" name="t1"/></td>
            </tr>

            <tr>
                <td>Name:</td>
                <td><input type="text" name="t2"/></td>
            </tr>
            <tr>
                <td>Address:</td>
                <td><input type="text" name="t3"/></td>
            </tr>
            <tr>
                <td><input type="reset" value="reset"/></td>
                <td><input type="submit" value="submit"/></td>
            </tr>

        </table>
```

```
	</form>
</center>
```

step5:

Create a model class i.e Employee inside "com.kits.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 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;
	}
}
```

step6;

	Create "EmployeeRepository" inside "com.kits.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:

	Create a EmployeeContrller inside "com.kits.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 addEmployee(Employee employee)
	{
		employeeRepository.save(employee);

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

step8:

Add Tomcat port number and configure H2 database and hibernate properties inside application.properties file.

application.properties

```
server.port=9090

spring.datasource.url= jdbc:h2:~/test
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:9090

http://locahost:9090/h2-console

RestController
==========

RestController is used to develop resful web services using @RestController annotation.

@RestController annotation is a class level annotation and it allows a class to handle all the request which are made by the client.

@RestController annotation in introduced in spring 4.0.

@RestController annotation is a combination of two annotations i.e @Controller and @ResponseBody.

We have following HTTP methods along with REST annotations.

ex:

HTTP Method	Annotation
GET	@GetMapping
POST	@PostMapping
PUT	@PutMapping
DELETE	@DeleteMapping

and etc.

Diagram: sb4.1

```
@Controller
class  HomeController
{
        @RequestMapping("/home")
        @ResponseBody
        public String home()
        {
            return "Controller Example";
        }
}
```

```
@RestController
class  HomeController
{
        @GetMapping("/home")
        public String home()
        {
            return "Controller Example";
        }
}
```

Project Structure

FourthSB

```
|
|---src/main/java
        |
        |---com.ihub.www (base package)
                |
                |--FourthSBApplication.java
        |
        |---com.ihub.www.controller
                |
                |---HomeController.java
|
|---src/main/resources
        |
        |---application.properties
|
|---src/test/java
|
|---pom.xml
```

step1:

Create a spring boot starter project i.e FourthSB.

ex:

starter:

Spring Web

step2:

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

step3:

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

HomeController.java

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

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

@RestController
```

```
@RequestMapping("/ihub")
public class HomeController
{
	@GetMapping("/home")
	public String home()
	{
		return "RestController Example";
	}
}
```

step4:

Configure server port number inside application.properties file.

application.properties

```
server.port=9090
```

step5:

Run spring boot starter project.

step6:

Test the application by using below request.
ex:
http://localhost:9090/ihub/home

Q) What is the difference between @Controller and @RestController annotation?

@Controller	@RestController
It is used to create spring MVC based web applications.	It is used to create restful web services.
It is a specialized version of @Component annotation.	It is a specialized version of @Controller annotation.
We need to use @ResponseBody annotation for every handler method.	It is a combination of @Controller and @RestController annotation.
It returns view in MVC.	It does not return view.

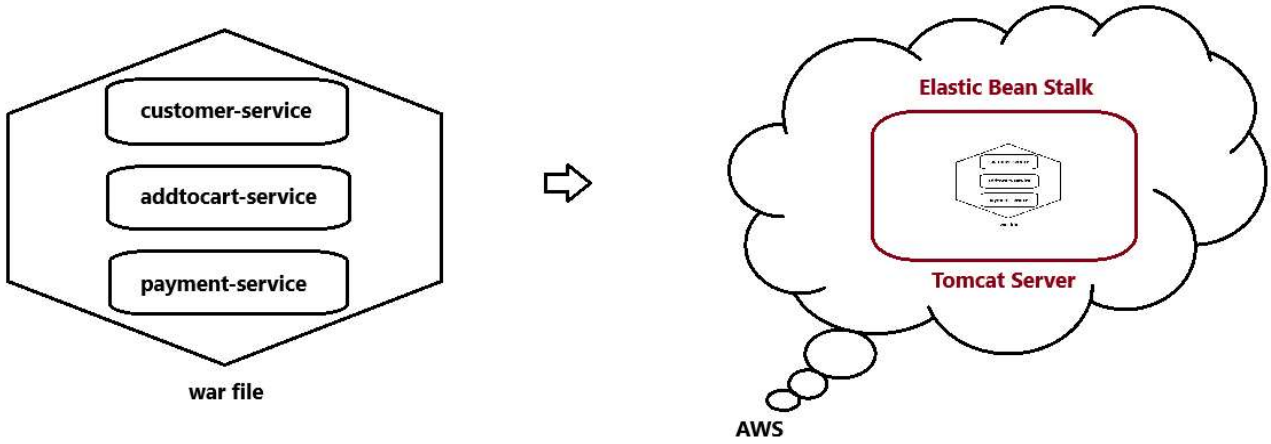
It is add in spring 2.5 version. It is add in spring 4.0 version.

Monolithic Architecture vs Microservice Architecture
==========
A monolithic means compose all in one place.

In monolithic architecture , we will develop independent services and at the end of the development we convert to war file and deploy in a server.

Advanced java follows monolithic architecture.

Diagram: sb4.2



Advantages:

1) Simple Develop

2) Simple Deploy

3) Simple Test

4) Simple Scale

Disadvantages:

1) Large and complex application

2) Blocks contineous development

3) Slow development

4) It is inflexible

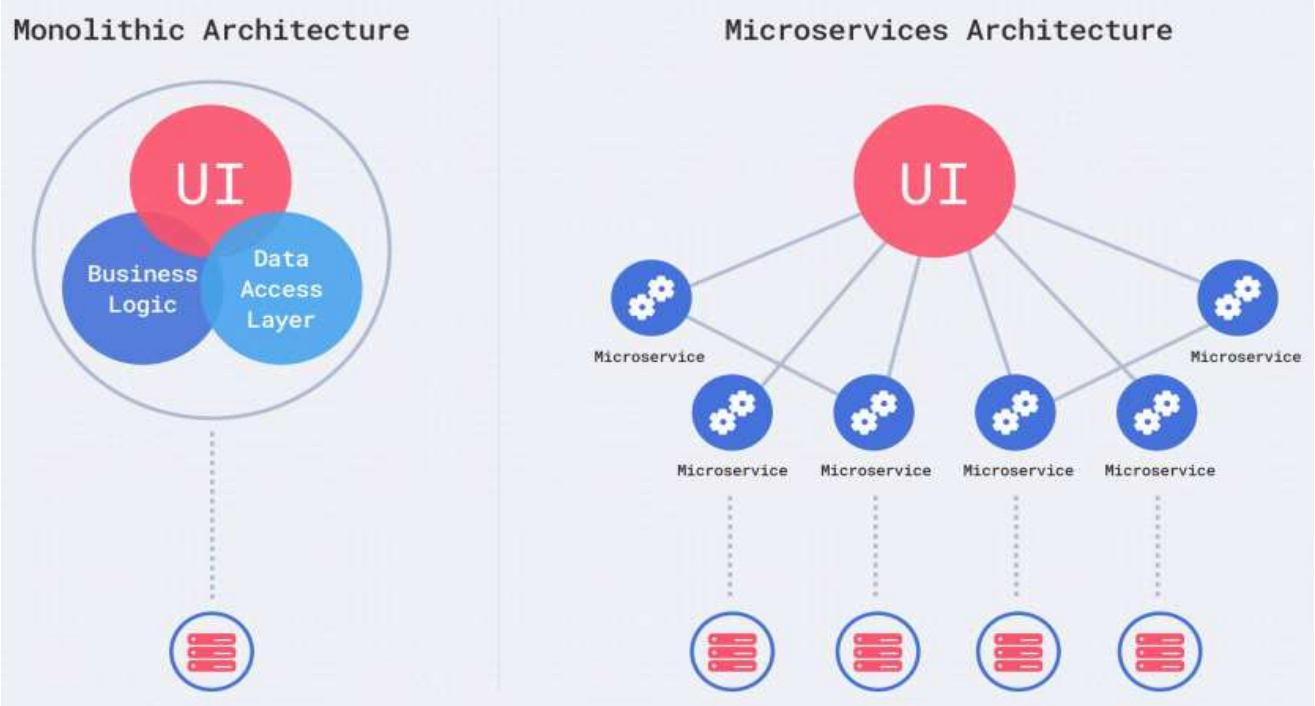
5) It is Inreliable

Microservice Architecture
==========
Microservice Architecture
==========
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.

Diagram: sb4.2



Advantages:
---

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.

Project Lombok
==========

The project Lombok is a popular and widely used Java library that minimizes or removes boilerplate code. It saves both time and effort. Just by using the annotations, we can save space and improve the readability of the source code. It automatically plugs into IDEs and builds tools to spice up our Java application.

It is achieved by introducing annotations that create getters, setters, constructors, equals(), hashCode(), and toString() methods-all typical Java code constructs-automatically.

Download link: https://projectlombok.org/download

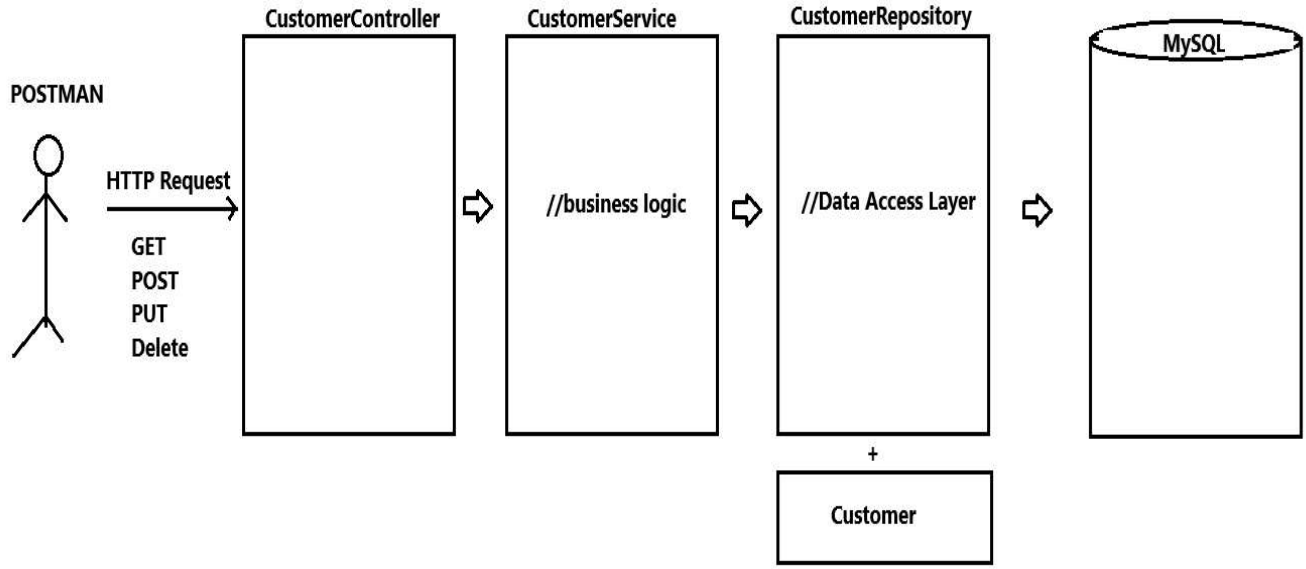
ex:

```
@Data
@AllArgumentConstructor
@NoArgumentConstructor
class Product
{
	private int prodId;
	private String prodName;
	private double prodPrice;
}
```

Microservice
==========
Using spring boot we can create micro services.

To build a microservice we need to use spring boot flow layered architecture.

Diagram: sb5.1



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

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

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

step1:

Create a "demo" schema in mysql database.

ex:

```
create schema demo;
use demo;
```

step2:

Launch STS IDE by choosing workspace location.

step3:

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

ex:

starters :

- Spring Web
- Spring Data JPA
- MySQL Driver
- Lombok

step4:

Crate a Customer model class inside "com.ihub.www.model" package.

Customer.java

```
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(name="customers")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Customer
{
	@Id
	private int custId;

	@Column
	private String custName;

	@Column
	private String custAddress;
}
```

step5:

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

CustomerRepository.java

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

import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import com.ihub.www.model.Customer;

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

}
```

step6:

Create a CustomerController 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.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
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 Iterable<Customer> getAllCustomers()
	{
		return customerService.getAllCustomers();
	}

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

step7:

Creat a CustomerService 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.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
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 Iterable<Customer> getAllCustomers()
	{
		return customerRepository.findAll();
	}

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

step8:

Convert application.properties file to application.yml file.

step9:

Configure server port , mysql database properties and hibernate properties inside application.yml file.

application.yml

```
server:
  port: 9191

spring:
  application:
    name: CUSTOMER-SERVICE

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

step10:

Run customer service

step11:

Download and install postman.
ex:
https://www.postman.com/downloads/

step12:

Open the postman and check the given request urls.

ex:

HTTP Method	Request url

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

Body

Raw (JSON)

```
{
	"custId":101,
	"custName":"Alan",
	"custAddress":"Florida"
}
```

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

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

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 Iterable<Customer> getAllCustomers()
	{
		return customerService.getAllCustomers();
	}

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

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

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

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 Iterable<Customer> getAllCustomers()
	{
		return customerRepository.findAll();
	}

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

	public String updateCustomer(Customer customer)
	{
		Customer cust=customerRepository.findById(customer.getCustId()).get();
		cust.setCustName(customer.getCustName());
		cust.setCustAddress(customer.getCustAddress());
		customerRepository.save(cust);
		return "Record Updated";
	}


	public String deleteCustomer(int custId)
	{
		Customer customer=customerRepository.findById(custId).get();
		customerRepository.delete(customer);
		return "Record Deleted";
	}
}
```

Ex:

HTTP Method	Request url

PUT http://localhost:9191/customer/update

Body

RAW (json)

```
{
    "custId":102,
    "custName":"Jack",
    "custAddress":"Chicago"
}
```

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

Exception handling in spring boot

==========

If we give wrong request to the application then we will get exception.

ex:

http://localhost:9191/customer/fetch/102

Here 102 record is not available then immediately our controller will throw one exception.

Ex:

```
{
    "timestamp": "2024-11-23T09:33:21.142+00:00",
    "status": 500,
    "error": "Internal Server Error",
    "path": "/customer/fetch/102"
}
```

Handing exceptions and sending errors in API's is always good for enterprise application.

There are two annotations we will use to handle the exceptions in spring boot.

1) @ControllerAdvice

It is used to handle the exceptions globally.

2) @ExceptionHandler

The @ExceptionHandler is an annotation is used to handle specific exceptions and sending custom response 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.model
		|
		|---Customer.java
	|
	|------com.ihub.www.repo
		|
		|---CustomerRepository.java (interface)
	|
	|------com.ihub.www.exception
		|
		|---ErrroDetails.java
		|---ResourceNotFoundException.java
		|---GlobalExceptionHandler.java

|
|---src/main/resources
	|
	|---application.yml
|
|---src/test/java
|
|---pom.xml
|

step1:

	Make sure customer-service project is ready.

step2:

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

step3:

	Create a ErrorDetails class inside "com.ihub.www.exception" package.

ErrorDetail.java

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

import java.util.Date;

public class ErrorDetails
{
	private Date timestamp;
	private String message;
	private String 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 class inside "com.kits.www.exception" package.

ResourceNotFoundException.java

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

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

step5:

Create a GlobalExceptionHandler class inside "com.kits.www.exception" package.

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);
	}

}
```

step6:

Now add ResourceNotFoundException to CustomerService.java file.

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

```
	public Customer getCustomerById(int custId)
	{
		return customerRepository.findById(custId).orElseThrow(()-> new
ResourceNotFoundException("Id Not Found"));
	}

	public String updateCustomer(Customer customer)
	{
		Customer
cust=customerRepository.findById(customer.getCustId()).orElseThrow(()-> new
ResourceNotFoundException("Id Not Found"));
		cust.setCustName(customer.getCustName());
		cust.setCustAddress(customer.getCustAddress());
		customerRepository.save(cust);
		return "Record Updated";
	}


	public String deleteCustomer(int custId)
	{
		Customer customer=customerRepository.findById(custId).orElseThrow(()-> new
ResourceNotFoundException("Id Not Found"));
		customerRepository.delete(customer);
		return "Record Deleted";
	}
}
```

step7:

Relaunch the spring boot application

step8:

Test the spring boot application by using below request url.

ex:

http://localhost:9191/customer/fetch/102

Note:

Here exception will display in below format.

ex:

```
{
	"timestamp": "2024-06-28T03:08:34.856+00:00",
	"message": "Id Not Found",
```

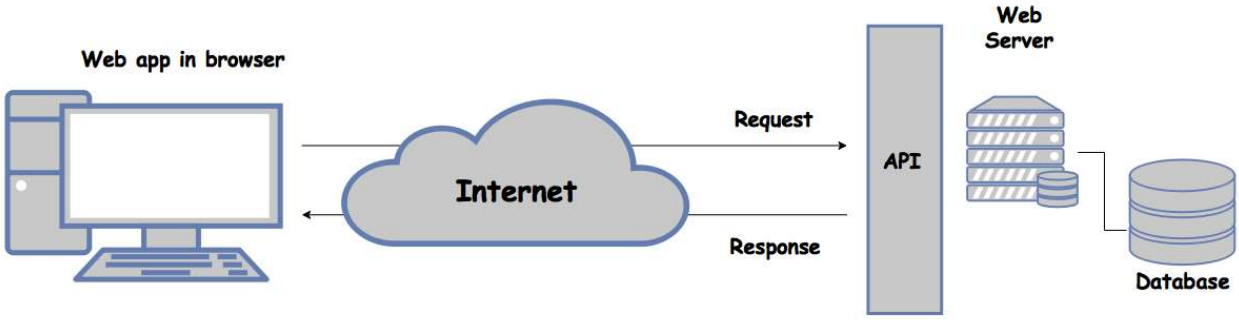
"details": "uri=/customer/fetch/102"
}

What is API
==========
API stands for Application Programming Interface.

API is a mechanism that enables two software components to communicate with each other using set of rules and protocols.

It acts like a interface between two software applications to exchange the data.

Diagram: sb6.1



We have four types of API's.

1) public API

It is open and available for use by any outside developers.

2) private API

It is also known as internal API.
It is available for use within the enterprise to connect the systems.

3) partner API

It is available to specficially selected and authorized outside developers.

4) composite API

It is generally combination of two or more API's.

Eureka Server
==========
Eureka server is also known as Discovery Server.

Eureka server is used to register client server applications (Microservices).

Eureka server runs on default port number i.e 8761.

Each micro service register with Eureka server and It contains each micro service port number and IP address.

Diagram: sb7.1

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
> Spring Web

> 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: 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

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.

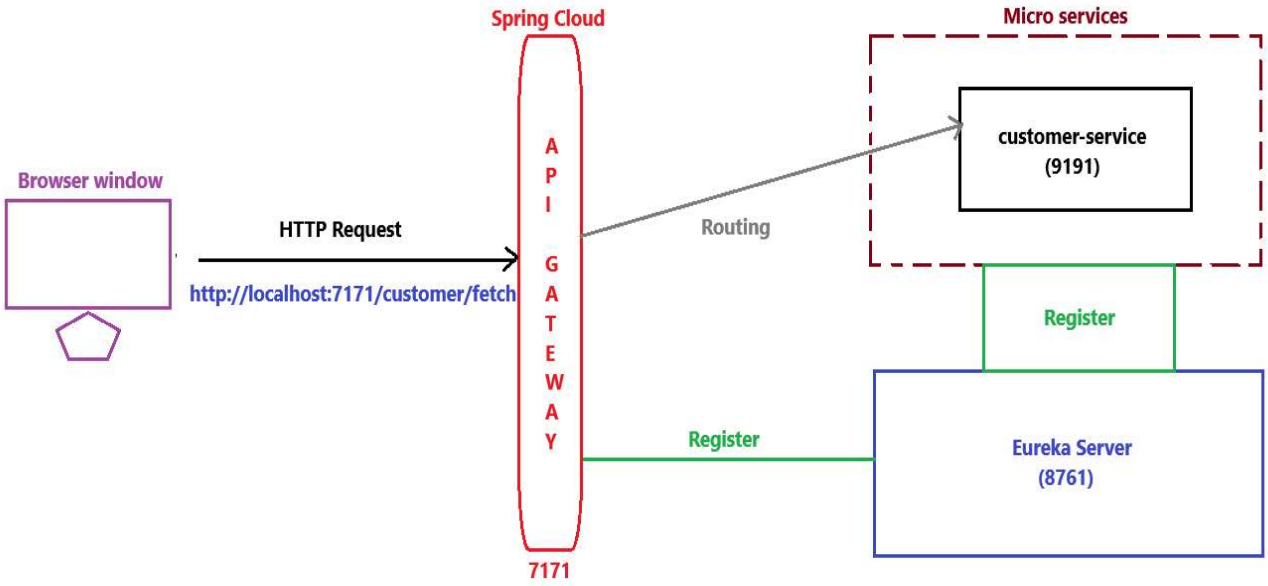
step7:

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: sb7.2



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 --> Request to customer
http://localhost:7171/customer/fetch --> Request to API gateway

Spring Cloud Hystrix
==========

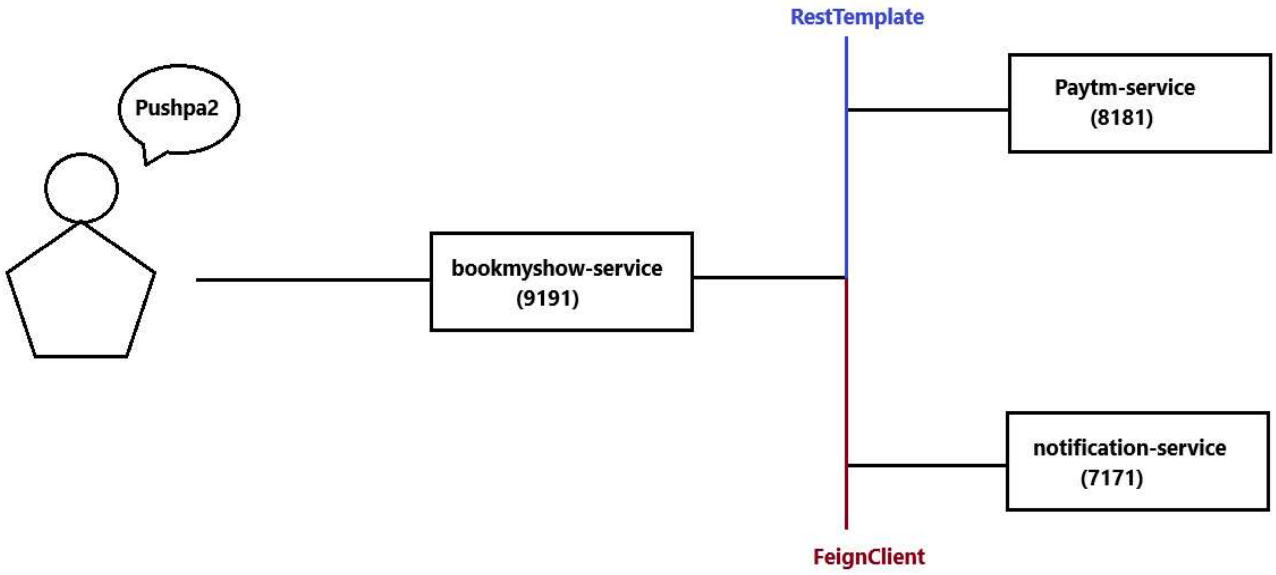
Hystrix is a fault tolerance library provided by Netflix.

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: sb8.1



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.

@Query Annotation
==========

@Query annotation is used to define custom queries using JPQL (Java Persistence Query Language) and native SQL.

JPQL

```
@Query(value = "SELECT u FROM User u")
public List<User> findAllUsers();
```

Native SQL

```
Query(value="select * from user u",nativeQuery=true)
public List<User> findAllUsers();
```

Project structure

```
QueryApp
|
|---src/main/java
|       |
        |---com.ihub.www
                |
                |---QueryAppApplication.java
        |
        |---com.ihub.www.controller
                |
                |---UserController.java

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

        |---com.ihub.www.repo
                |
                |---UserRepository.java (interfae)
```

```
        |---com.ihub.www.model
                |
                |---User.java
|
|---src/main/resources
        |
        |---application.properties
|
|---src/test/java
|
|---pom.xml
```

step1:

Create a spring starter project i.e QueryApp.

ex:

starters:

Spring Web
Spring data JPA
Project lombok
H2 Database

application.properties

```
server.port=9090

spring.datasource.url= jdbc:h2:~/test
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
```

User.java

```
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(name="users")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User
{
	@Id
	private int userId;

	@Column
	private String userName;

	@Column
	private String userAddress;
}
```

UserRepository.java

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

import java.util.List;

import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import com.ihub.www.model.User;

@Repository
public interface UserRepository extends CrudRepository<User, Integer>
{
	@Query(value = "select * from users",nativeQuery = true)
	public List<User> findAllUsers();


}
```

UserService.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.model.User;
import com.ihub.www.repo.UserRepository;

@Service
public class UserService
{
	@Autowired
	UserRepository userRepository;


	public List<User> getAllUsers()
	{
		return userRepository.findAllUsers();
	}
}
```

UserController.java

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

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.ihub.www.model.User;
import com.ihub.www.service.UserService;

@RestController
@RequestMapping("/user")
public class UserController
{
		@Autowired
		UserService userService;
```

```
	@GetMapping("/all")
	public List<User> getAllUsers()
	{
		return userService.getAllUsers();
	}

}
```

Request url
---
http://localhost:9090/h2-console (First insert the data)
http://localhost:9090/user/all

Spring Security
==========
Spring security is a framework which provides various security features to create secure enterprise application.

It is a sub project of spring framework which is developed in 2003 by Ben Alex.
Late on , in 2004 it was released under the Apache licence with spring 2.0.0.

It targets two major areas of our application.

1) Authentication

It is a process of knowing or identifying a user.

2) Autherization

It is a process of giving the autherization to the user to perform actions in our application.

Project Structure

```
SpringSecurity
|
|---src/main/java
	|
	|---com.ihub.www
		|
		|---SpringSecurityApplication.java
	|
	|---com.ihub.www.controller
		|
```

```
                |---HomeController.java

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

|---src/test/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("/home")
        public String home()
        {
                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/home

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.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/home

Project structure

QueryApp
|
|---src/main/java

```
|       |
        |---com.ihub.www
                |
                |---QueryAppApplication.java
        |
        |---com.ihub.www.controller
                |
                |---UserController.java

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

        |---com.ihub.www.repo
                |
                |---UserRepository.java (interfae)

        |---com.ihub.www.model
                |
                |---Users.java
|
|---src/main/resources
        |
        |---application.properties
|
|---src/test/java
|
|---pom.xml
```

step1:

Create a spring starter project i.e QueryApp.

ex:

starters:

Spring Web
Spring data JPA
Project lombok
H2 Database

application.properties

```
server.port=9090

spring.datasource.url= jdbc:h2:~/test
```

```
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
```

Users.java

```
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(name="users")
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Users
{
	@Id
	private int userId;

	@Column
	private String userName;

	@Column
	private String userAddress;
}
```

UserRepository.java

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

import java.util.List;
```

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

import com.ihub.www.model.Users;

@Repository
public interface UserRepository extends JpaRepository<Users, Integer>
{
	@Query("SELECT u FROM Users u")
	List<Users> findAllUsers();

	Users findByUserName(String userName);
}
```

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

import com.ihub.www.model.Users;
import com.ihub.www.repo.UserRepository;

@Service
public class UserService
{
	@Autowired
	UserRepository userRepository;


	public List<Users> getAllUsers()
	{
		return userRepository.findAllUsers();
	}
```

```
	public Users getUser(String userName)
	{
		return userRepository.findByUserName(userName);
	}
}
```

UserController.java

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

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.ihub.www.model.Users;
import com.ihub.www.service.UserService;

@RestController
@RequestMapping("/user")
public class UserController
{
		@Autowired
		UserService userService;

		@GetMapping("/all")
		public List<Users> getAllUsers()
		{
			return userService.getAllUsers();
		}

		@GetMapping("/fetch/{userName}")
		public Users getUser(@PathVariable String userName)
		{
			return userService.getUser(userName);
		}

}
```

Request url

---

http://localhost:9090/h2-console (First insert the data)
http://localhost:9090/user/all
http://localhost:9090/user/fetch/Alan

Q) What is the difference between application.properties and application.yml?

application.properties	application.yml
It follows non-hierarchical structure.	It follows hierarchical structure.
We can configure only one spring profile.	We can configure multiple spring profiles.
It is primarily used in java.	It is used in many languages like Java, python, Ruby and etc.
Supports key/val, but doesn't support values beyond the string.	Supports key/val, basically map, List and scalar types (int, string etc.)

Q) What is the difference between Spring Bean and POJO class?

Spring Bean	POJO
An object that is managed by the spring IoC container is called spring bean.	An object that is managed by the user is called pojo. Any java object is a pojo.
Spring beans can be inject to other beans using dependency injection mechanism.	POJOs are not managed by the spring so they are not eligible for automatic dependency injection mechanism.
Spring beans have restrictions.	POJOs don't have restrictions.