===================
Java Microservices
===================

Pre-Requisite : Spring Boot + Spring Web MVC

Course content

Module-1 : RESTFul Services
Module-2 : Microservices
Module-3 : Spring Security
Module-4 : Integrations

=================
RESTFul Services
=================
1) What is Distributed Application ?
2) Distributed Technologies
3) REST Architecture
- Provider
- Consumer
4) HTTP Protocol
- Request
- Response
- Methods
- Status Codes

5) XML & JAX-B API
- Binding Classes
- Marshalling
- Un-Marshalling

6) JSON & JACKSON / GSON API

7) Provider Development
- @RestController
- @RequestParam
- @PathVariable
- @RequestBody
- @GetMapping
- @PostMapping
- MediaTypes (consumes & produces)

8) Content-Type and Accept headers

9) Provider Testing using POSTMAN

10) Provider Documentation using SWAGGER & Swagger UI

11) Consumer Development (Sync & Async)

- RestTemplate (Sync)

- WebClient (Sync & Async)

12) Exception handling in REST api

- @RestControllerAdvice

- @ExceptionHandler

==============
Microservices
===============

1) What is Monolith architecture ?
2) Pros and Cons of Monolith
3) Microservices Introduction
4) Pros and Cons of Microservices
5) Microservices Architecture
6) Service Registry (Eureka Server)
7) Admin Server
8) Zipkin Server
9) Microservices Develoment
10) Interservice Communication (FeignClient)
11) APIGateway (Filters & Routers) (Spring Cloud Gateway)
12) Load Balancing (Ribbon)
13) Circuit Breaker
14) Config Server
15) Connecting Multiple DBs

================
Spring Security
=================

1) Basic Auth
2) OAuth 2.0
3) JWT

=============
Integrations
=============

1) Spring Boot + Kafka Integration
2) Spring Boot + Redis Integration
3) Spring Boot + Angular Integration

=================

RestFul Services
================

=> To develop distributed applications with intereoperability

App-1   <------------->   App-2

=> Intereoperability means platform indendent and language independent

java-app <-----------> .net app

.Net app <--------> Python

Python <-----------> Java

==
Why one application should communicate with another application?
==

=> To re-use business services (B 2 B)

=========================
Distributed Technologies
=========================

1) CORBA
2) RMI
3) EJB
4) SOAP Webservices
5) RESTFul Services (Trending)

==================
REST Architecture
===================

1) Provider / Resource
2) Consumer / Client

Provider: The application which is giving services to other applications is called as Provider application.

Consumer : The application which is accessing services from other applications is called as Consumer application.

==
How communication will happen between Provider & Consumer ?
==

-> HTTP protocol will act as mediator between Consumer and Provider

-> Consumer and Provider will exchange data in the form XML / JSON

Note: XML and JSON are intereoperable.

==============
HTTP Protocol
==============

1) Http Request
2) Http Response
3) HTTP Methods
4) HTTP Status Codes

=> HTTP will act as mediator between Client and Server
=> HTTP is stateless protocol (can't remember previous requests)

==============
HTTP Methods
==============

=> Every REST API method should be mapped to HTTP Method.

GET --> To get resource/data from server

POST --> To insert/create record at server

PUT --> To update data at server

DELETE --> To delete data at server

==================
HTTP Status Codes
===================

-> When client send request to server then server will process that request and server will send response to client with status code.

100 - 199 (1xx) ---> Information

200 - 299 (2xx) ---> Success (OK)

300 - 399 (3xx) ---> Redirection

400 - 499 (4xx) ---> Client Error

500 - 599 (5xx) ---> Server Error

=============
HTTP Request
==============

-> HTTP request contains below parts

1) Request Line (Request Type + URL)

2) Request Header (metadata)

3) Request Body (Payload)

===============
HTTP Response
===============

-> HTTP response contains below parts

1) Response Line (Status Code + Status Msg)

2) Response Header (metadata)

3) Response Body (Payload)

=====================================
JSON (Java Script Object Notation)
=====================================

=> JSON is used to represent data in key-value format

=> JSON is universal format to exchange data over internet

Synax:

```
{

"id" : 101,
"name" : "Ashok",
"gender" : "Male",
"phno"  : 463413

"address" : {
"city" : "Hyd",
"state" : "TG"
}

}
```

=> As part of REST API development, we need to convert Java Obj data to JSON format and JSON data to Java Object

Java Obj <---------------> JSON

=> In Java we don't have direct support to convert java to json and vice versa.

=> We have below third party apis to work with JSON data in Java applications

1) Jackson api

2) Gson api

==============
Jackson API
==============

=> ObjectMapper class provided methods to convert java to json and vice versa

========================
Working with JACKSON API
========================

1) Create maven Project (quick-start)

2) Add Jackson dependency in pom.xml file

```
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<version>1.18.26</version>
</dependency>

<dependency>
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-databind</artifactId>
<version>2.14.2</version>
</dependency>
```

3) Create Binding class to represent data

```
@Data
public class Address {

private String city;
private String state;
private String country;

}

@Data
public class Customer {
```

```
private Integer id;
private String name;
private String email;
private Long phno;

private Address addr;
}
```

4) Create Converter classes

```
public class JavaToJsonConverter {

public static void main(String[] args) throws Exception{

Address addr = new Address();
addr.setCity("Hyd");
addr.setState("TG");
addr.setCountry("India");

Customer c = new Customer();
c.setId(1);
c.setName("Robert");
c.setEmail("robert@gmail.com");
c.setPhno(76413132l);
c.setAddr(addr);

ObjectMapper mapper = new ObjectMapper();
mapper.writeValue(new File("customer.json"), c);
System.out.println("Json file created");

}
}
```

```
public class JsonToJavaConverter {

public static void main(String[] args) throws Exception {

File f = new File("customer.json");

ObjectMapper mapper = new ObjectMapper();

Customer c = mapper.readValue(f, Customer.class);

System.out.println(c);

}

}
```

==========
GSON API
=========

-> Provided by Google

```xml
<dependency>
<groupId>com.google.code.gson</groupId>
<artifactId>gson</artifactId>
<version>2.8.5</version>
</dependency>
```

-> In this api we have predefined class i.e 'Gson'

```java
Gson gson = new Gson ( );

gson.toJson(file, obj); // convert java obj to json

gson.fromJson(file, Type); // convert json to java obj
```

===============
XML and JAX-B
===============

-> XML stands for Extensible Markup Language

-> XML is intereoperable

-> XML will represent data in element format

Ex: <id>101</id>

-> Every element is combination of start tag and end tag

-> In XML we have 2 types of elements

1) Simple Elements
2) Compound Elements

```xml
<person>
<id>101</id>
<name>smith</name>
<address>
<city>Hyd</city>
<state>TG</state>
</address>
</person>
```

-> Elements which contains data directley are called as Simple Elements

```
<id>101</id>
<name>smith</name>
<city>Hyd</city>
<state>TG</state>
```

-> Elements which contains child elements are called as compound elements

```
<person>
<address>
```

==========
JAX-B API
===========

-> JAX-B Stands for Java Architecture For XML Binding

-> Using JAX-B API we can convert xml data to java object and vice versa

Marshalling : Converting java obj to xml

Un-Marshalling : Converting xml to java obj

Note: To perform marshalling or Un-marshalling we need to create Binding class first.

Note: Upto JDK 1.8v, JAX-B is part of JDK itself. But from Java 1.9 version it is not part of JDK.

-> If we want to work with JAX-B api from java 1.9v then we have to add dependency in pom.xml file

=======================
Working with JAX-B API
=======================

1) Create maven quick-start project

2) Add below dependencies

```
<dependency>
<groupId>org.projectlombok</groupId>
<artifactId>lombok</artifactId>
<version>1.18.26</version>
</dependency>
<dependency>
<groupId>com.sun.xml.bind</groupId>
<artifactId>jaxb-core</artifactId>
<version>2.3.0.1</version>
</dependency>
```

```
<dependency>
<groupId>javax.xml.bind</groupId>
<artifactId>jaxb-api</artifactId>
<version>2.3.1</version>
</dependency>
<dependency>
<groupId>com.sun.xml.bind</groupId>
<artifactId>jaxb-impl</artifactId>
<version>2.3.1</version>
</dependency>
<dependency>
<groupId>org.javassist</groupId>
<artifactId>javassist</artifactId>
<version>3.25.0-GA</version>
</dependency>
```

3) Create binding class (represent xml structure)

```
@Data
@XmlRootElement
public class Customer {

private Integer id;
private String name;
private String email;
private Long phno;

}
```

4) Create Converter classes

```
public class MarshalDemo {

public static void main(String[] args) throws Exception {

Customer c = new Customer();
c.setId(101);
c.setName("John");
c.setEmail("john@gmail.com");
c.setPhno(64131313l);

JAXBContext context = JAXBContext.newInstance(Customer.class);

Marshaller marshaller = context.createMarshaller();

marshaller.marshal(c, new File("customer.xml"));

System.out.println("xml created....");
}

}
```

```
public class UnMarshallDemo {

public static void main(String[] args) throws Exception {

File f = new File("customer.xml");

JAXBContext context =
JAXBContext.newInstance(Customer.class);

Unmarshaller unmarshaller = context.createUnmarshaller();

Object object = unmarshaller.unmarshal(f);

Customer c = (Customer) object;

System.out.println(c);
}
}
```

=====================
Provider Development
=====================

-> The app which is providing services to other apps is called as Provider

-> Provider is also called as REST API.

1) Create Spring Boot application with below dependencies

a) web-starter

2) Create REST Controller class using @RestController annotation

3) Write the Required methods and map them to URL + HTTP protocol methods

4) Run the application and test it using POSTMAN

==============================First Rest Controller====================

```
@RestController
public class MsgRestController {

@PostMapping("/msg")
public ResponseEntity<String> saveMsg() {
// logic to save msg
String responseBody = "Msg Saved Successfully";
return new ResponseEntity<String>(responseBody, HttpStatus.CREATED);
}

@GetMapping("/welcome")
public ResponseEntity<String> getWelcomeMsg() {
String msg = "Welcome to REST API..!!";
return new ResponseEntity<String>(msg, HttpStatus.OK);
}

@GetMapping("/greet")
public String getGreetMsg() {
return "Good Evening";
}
}
```

==

```
@Data
public class User {

private Integer id;
private String name;
private String email;

}


@RestController
public class UserRestController {

private Map<Integer, User> dataMap = new HashMap<>();

@PostMapping("/user")
public ResponseEntity<String> addUser(@RequestBody User user) {
System.out.println(user);
dataMap.put(user.getId(), user);
return new ResponseEntity<String>("User Saved", HttpStatus.CREATED);
}

}
```

===

```
{
"id" : 202,
"name" : "John",
"email" : "john@gmail.com"
}
```

===

@RestController : To represent java class as Distributed Component

@RestController = @Controller + @ResponseBody

@GetMapping : Map the method to HTTP GET Request

@PostMapping : Map the method to HTTP POST Request

@RequestBody : To read payload from HTTP Request Body

ResponseEntity : To set custom HTTP Status Code in Response

Postman : To test REST API functionality

====================================
Query Parameters & Path Parameters
====================================

=> Query Parameters & Path Parameters are used to send data in URL

QP Ex : https://www.youtube.com/watch?v=8eVaci9WvP8

PP Ex : www.ashokitech.com/courses/java

Note: When client is sending GET request then client can use Query Params or Path Params to send data to Server

Ex: ticket-number, emp-id, book-id, customer-id etc..

Note: GET request will not contain Request Body so we have to use either Query Param or Path Param to send data to server.

================
Query Parameters
================

=> Query Params will represent data in key - value format

=> Query Params will start with '?' symbol

=> Query Params will be seperated using '&' symbol

=> Query Params should present only at end of the URL

=> To read Query Params from URL we will use @RequestParam annotation

```
@GetMapping("/user")
public User getUser(@RequestParam("userid") Integer userId) {
User user = dataMap.get(userId);
return user;
}
```

URL : http://localhost:8080/user?userid=202

Path Parameters

-> To send data to server in the URL

-> Path Param will represent data directley

-> Path Params can present anywhere in the URL

-> Path Param will start with '/' and will be seperated by '/'

-> We need to represent Path Parameters position in the URL pattern like below

Ex: @GetMapping("/user/{id}/data")

-> To read Path Parameters we will use @PathVariable annotation

```
@GetMapping("/user/{id}/data")
public User getUser(@PathVariable("id") Integer userId) {
User user = dataMap.get(userId);
return user;
}
```

URL : URL : http://localhost:8080/user/202/data

====================
Consumes & Produces
===================

consumes : It represents in which format REST API method can accept input data from client

produces : It represents in which format REST API method can provide response to clients

Content-Type : This header will represent in which format client sending data to server in request body

Accept : This header will represent in which format client expecting response from server

-------------------- Consumes & Produces Example ---------------

```
@Data
@XmlRootElement
public class Book {

private Integer id;
private String name;
private Double price;

}
```

```
@RestController
public class BookRestController {

@PostMapping(
value="/book",
consumes = {"application/xml", "application/json"}
)
public ResponseEntity<String> addBook(@RequestBody Book b){
System.out.println(b);
//logic to save in db
String msg = "Record Saved";
return new ResponseEntity<>(msg, HttpStatus.CREATED);
}

@GetMapping(
value="/book",
produces = {"application/xml", "application/json"}
)
public Book getBook() {
Book b = new Book();
b.setId(101);
b.setName("Java");
b.setPrice(130.00);
return b;
}
}
```

=============
Requirement
=============

Develop an IRCTC REST API to book train ticket

Input : Passenger Data
- name
- from
- to
- doj
- trainNumber

Output : Ticket Data
- ticketNum
- name
- cost
- from
- to
- doj
- status

consumes : application/json

produces : application/json

======================
Development Procedure
======================

1) Create Spring Boot application with below starters

a) web-starter
b) lombok
c) devtools

2) Create Request Binding class (Passenger.java)

3) Create Response binding class (Ticket.java)

4) Create Service Interface & Impl class-

5) Create Rest Controller with below 2 operations

POST : To book ticket

GET : To get ticket

6) Run the application and test it using POSTMAN

Request data

```
{
"name": "John",
"from": "Hyd",
"to": "Delhi",
"doj" : "15-May-2023",
"trainNumber" : "46464"
}
```

======================
Swagger Configuration
======================

=> Swagger is used to generate REST API documentation

=> Swagger is a third party Library (we need to add in our app)

=> Swagger UI is used to test REST API with user interface

1) Add below dependencies in pom.xml file

```
<dependency>
<groupId>io.springfox</groupId>
<artifactId>springfox-swagger2</artifactId>
<version>2.4.0</version>
</dependency>

<dependency>
<groupId>io.springfox</groupId>
<artifactId>springfox-swagger-ui</artifactId>
<version>2.4.0</version>
</dependency>
```

2) Create SwaggerConfig class

```
@Configuration
@EnableSwagger2
public class SwaggerConfig {

@Bean
public Docket apiDoc() {
return new Docket(DocumentationType.SWAGGER_2)
.select()
.apis(RequestHandlerSelectors.basePackage("in.ashokit.rest"))
.paths(PathSelectors.any())
.build();
}
}
```

Note: If we are getting NPE when we run the application, then add below property in application.properties file

spring.mvc.pathmatch.matching-strategy = ANT_PATH_MATCHER

3) Run the application and access SWAGGER DOC and SWAGGER UI

Swagger DOC URL : http://localhost:8080/v2/api-docs

Swagger UI URL : http://localhost:8080/swagger-ui.html

==
IRCTC CLOUD API URL : http://13.232.253.164:8080/swagger-ui.html
==

====================
Consumer Development
=====================

=> The application which is accessing services from other applications is called as Consumer application.

=> In Spring Boot we can develop Consumer in 3 ways

1) RestTemplate (out dated)

2) WebClient (From Spring 5.x)

3) FeignClient (Spring Cloud)

===
Steps To develop Make My Trip Application (Consumer)
===

1) Create Spring Boot app with below dependencies

a) web-starter
b) thymeleaf-starter
c) lombok
d) devtools

2) Create Request and Response Binding classes

3) Create Service class with Integration Logic

4) Create Controller with Required methods

a) GET - load form
b) POST - Book ticket
c) GET - Get Ticket

5) Create View Pages

6) Run the application

==

```
@Service
public class MakeMyTripServiceImpl implements MakeMyTripService {

private String BOOK_TICKET_URL="http://13.232.253.164:8080/ticket";

private String GET_TICKET_URL="http://13.232.253.164:8080/ticket/{ticketNum}";

@Override
public Ticket bookTicket(Passenger passenger) {

RestTemplate rt = new RestTemplate();
ResponseEntity<Ticket> respEntity =
rt.postForEntity(BOOK_TICKET_URL, passenger, Ticket.class);

Ticket ticket = respEntity.getBody();

return ticket;
}

@Override
public Ticket getTicketByNum(Integer ticketNumber) {

RestTemplate rt = new RestTemplate();

ResponseEntity<Ticket> respEntity =
rt.getForEntity(GET_TICKET_URL, Ticket.class, ticketNumber);

Ticket ticket = respEntity.getBody();

return ticket;
}

}
```

==
===

```
private String BOOK_TICKET_URL="http://13.232.253.164:8080/ticket";

private String GET_TICKET_URL="http://13.232.253.164:8080/ticket/{ticketNum}";
```

==

=> WebClient is a predefined interface introduced in Spring 5.x version

=> Using WebClient we can send HTTP Requests (GET, POST, PUT, DELETE)

=> WebClient supports both Synchronus & Asynchronus communications

=> To use WebClient, we need to add "web-flux-starter" in pom.xml file

```
@Service
public class MakeMyTripServiceImpl implements MakeMyTripService {

private String BOOK_TICKET_URL="http://13.232.253.164:8080/ticket";

private String GET_TICKET_URL="http://13.232.253.164:8080/ticket/{ticketNum}";

@Override
public Ticket bookTicket(Passenger passenger) {

// get the instance of webclient (impl class)
WebClient webClient = WebClient.create();

// send POST request with passenger data
//and map response to Ticket Obj

Ticket ticket = webClient.post()
.uri(BOOK_TICKET_URL)
.bodyValue(passenger)
.retrieve()
.bodyToMono(Ticket.class)
.block();

return ticket;


}

@Override
public Ticket getTicketByNum(Integer ticketNumber) {

// get the instance of webclient (impl class)
WebClient webClient = WebClient.create();

// send get request and map response to Ticket Obj

Ticket ticket = webClient.get()
.uri(GET_TICKET_URL, ticketNumber)
.retrieve()
.bodyToMono(Ticket.class)
.block(); // sync call
return ticket;

}
}
```

============================
Sync & Async Communication
============================

Sync Communication : After sending the request thread will wait for Response

ASync Communication : After sending the request thread will not wait for response

```
@SpringBootApplication
public class Application {

static String url = "http://13.232.253.164:8080/ticket/{ticketNum}";

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

WebClient webClient = WebClient.create();

System.out.println("request sending start ......");

webClient.get()
.uri(url,6)
.retrieve()
.bodyToMono(String.class)
.subscribe(Application::handleResponse);

System.out.println("request sending end ......");
}

public static void handleResponse(String response) {
System.out.println(response);
}

}
```

RestTemplate --> Class ---> Sync

WebClient --> Interface --> Sync & Async

How to send Request Header and Body using WebClient

```
@Override
public Ticket bookTicket(Passenger passenger) {

// get the instance of webclient (impl class)
WebClient webClient = WebClient.create();

// send POST request with passenger data
//and map response to Ticket Obj

Ticket ticket = webClient.post()
.uri(BOOK_TICKET_URL)
.header("Accept", "application/json")
.bodyValue(passenger)
.retrieve()
.bodyToMono(Ticket.class)
.block();

return ticket;


}
```

===
application.properties file Vs application.yml file
===

-> In Spring Boot we will use .properties or .yml file to configure application properties

Ex: DataSource, SMTP, PORT, Kafka, Redis etc...

-> Properties file will represent data in key value format
-> YML file will represent data in hierarchical format

-> .properties will be used only in java applications
-> YML is universal format (java, .Net, Python, ansible, k8s)

Note: YML stands for YET ANOTHER MARKUP Language

-> Indent spacing is very important in yml file

=> Approach to develop Spring Based Applications with less configurations.

1) POM starters
2) Dependency Version management
3) Auto Configuration
4) Embedded Server
5) Actuators

============
Actuators
============

-> Actuators are used to provide production-ready features for our application

(Monitor and manage our application)

-> To work with Actuators spring boot provided below starter

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

-> We can write below configuration in application.yml file to expose actuator endpoints

```
-----------------------------application.yml---------------------
management:
endpoints:
web:
exposure:
include: '*'
exclude: 'beans'

endpoint:
shutdown:
enabled: true

-----------------------------------------------------------------
```

############ URL : http://localhost:8080/actuator/ ############

health : http://localhost:8080/actuator/health

mappings : http://localhost:8080/actuator/mappings

beans : http://localhost:8080/actuator/beans

heapdump : http://localhost:8080/actuator/heapdump

threaddump : http://localhost:8080/actuator/threaddump

Shutdown : http://localhost:8080/actuator/shutdown

Note: Shutdown is a special endpoint which is used to stop our application and it is mapped to POST request.
==