==========
RESTFul Services & Microservices
==========

Pre-Requisite : Spring Boot

==========
Course Content :
==========
Part- 1) RESTFul Services

=> What is Distributed Application
=> Distributed Technologies
=> REST Introduction
=> REST Architecture
=> XML & JAX-B API.
=> JSON & Jackson API
=> HTTP Protocol (Methods + Status Codes)

=> REST Architecture Principle
=> REST API Development (Provider Development)
=> What is RestController
=> GET + POST + PUT + DELETE Methods
=> Query Params & Path Params
=> Request Body & Response Body
=> REST API Testing using POSTMAN
=> Swagger

=> Rest Client Development (Consumer Development)
=> RestTemplate class (Sync)
=> WebClient (Sync & Async)

=> Exception Handling in REST API

Part-2) Spring Security

=> Authetication
=> Authorization
=> Basic Authentication
=> OAuth 2.0
=> JWT

Part-3) Microservices with Spring Cloud

=> Monolith Architecture
=> Pros & Cons of Monolith

=> Microservices Introduction
=> Microservices Architecture
=> Pros & Cons of Microservices

=> Service Registry (Eureka Server)
=> Admin Server + Admin Client
=> Distributed Log Tracing (zipkin + sleuth)
=> API Gateway (Filters + Rounting) - CloudGateway

=> Fiegn Client (Interservice Communication)
=> Load Balancer (Ribbon)

=> Cicuit Breaker with Reselliance

=> Config Server

Part-4 : SPring Boot - Integrations

=> Spring Boot with Redis Cache Integration

=> Spring Boot with Kafka Integration

=> Spring Boot with Docker Integration

Course Duration : 45 Days

Class Timings : 7:30 PM - 9:00 PM (IST) (Mon-Sat)

Course Start Date : Today

Course Fee : 8,000 INR (Live Classes + Backup Videos + ClassNotes)

==========
Spring Boot & Microservies - 10,000 INR (5:00 PM IST Batch started one week ago)
==========

Spring Core
SPring Boot
Spring Data JPA
Web MVC
RESTFulservices
Security
Microservices
Integration

==========
RESTFul Services & Microservies - 8000 INR (7:30 PM IST Batch started today)
==========

RESTFulservices
Security
Microservices
Integration

==========
What is Distributed Application ?
==========

-> If one application is communicating with another application then they are called as Distributed Applicat ions.

MakeMyTrip ----------> IRCTC

Passport App ---------> AAdhar App

Gpay / Phone Pay ------> Banking App

=> Distributed Applications are used for Business To Business Communication (B 2 B).

Note: Web Applications are used for Customer To Business Communication (C 2 B).

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

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

Provider : The application which is providing services to other application (Resource)

Consumer : The application which is accessing services from other application (Client)

==========
What is Intereoperability ?
==========

=> Irrespective of language & platform if applications are communicating then they are called as Intereop erable applications.

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

Python <------> .Net

.Net <-----> PHP

Note: By using RESTFUl Services we can develop Intereoperable applications.

=======
XML
======

-> XML stands for Extensible Markup Language

-> XML Govenred by w3c org

-> XML is platform independent and language independent

-> XML is used to exchange data between applications (webservices)

-> XML represents data in elements format

-> Every Element contains open tag and closed tag

```
<id>101</id>
<name>Ashok</name>
```

-> In XML we can use 2 types of elements

1) Simple element
2) Compound element

-> The element which represents data directley is called as simple element.

```
<id>101</id>
```

-> The element which represents child element(s) is called as Compound Element.

```
<person>
 <fname>Ashok</fname>
 <lname>Kumar</lname>
</person>
```

==========
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 java object to xml data.

Marshalling : Convert Java Object to XML data

Un-Marshalling : Convert XML data to Java Object

=> To perform Marshalling & Un-Marshalling we need to create Binding Classes.

=> Binding Class means the class which represents XML structure

Note: Upto JDK 1.8v JAX-B is part of JDK software. From JDK 1.9 version JAX-B removed from JDK.

Note: From java 1.9v or above versions should have jax-b dependency.

========== JAX-B Example ==========

```
@XmlRootElement
public class Person {
```

```
 private Integer id;
 private String name;
 private String email;
 private String gender;

 private Address addr;

 //setters & getters

}

public class Address {

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

 //setters & getters

}

public class MarshallDemo {

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

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

  Person p = new Person();
  p.setId(101);
  p.setName("Ashok");
  p.setGender("Male");
  p.setEmail("ashokitschool@gmail.com");
  p.setAddr(addr);

  JAXBContext context = JAXBContext.newInstance(Person.class);
  Marshaller marshaller = context.createMarshaller();
  marshaller.marshal(p, new File("person.xml"));

  System.out.println("done.....");

 }
}

public class UnmarshalDemo {

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

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

  Unmarshaller unmarshaller = context.createUnmarshaller();
```

```
  Person p  = (Person) unmarshaller.unmarshal(new File("person.xml"));

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

```
<dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.1</version>
</dependency>
```

=======
JSON
=======

=> JSON stands for Java Script Object Notation

=> JSON represents data in key-value format

=> JSON is light weight

=> JSON is Intereoperable (Platform & Language Independent)

=> We can use JSON structure to exchange data from one application to another application

Note: When compared with XML, JSON will take less memory

=> To work with JSON data we have below 3rd party APIs

   1) Jackson (It is default in spring boot)

   2) Gson (given by google)

=> Using above apis we can convert java object to json and json to java object

    Serialization : Convert Java Object to JSON

    De-Serialization : Convert JSON data to Java Object

=> Jackson API provided methods to perform operations java with json

```
    ObjectMapper mapper = new ObjectMapper ( ) ;

    mapper.writeValue(new File("person.json"), personObj);

    Person p = mapper.readValue(new File("person.json"), Person.class);
```

========== Jackson API Example ==========
========

```
public class Address {

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

 // setters  & getters
}

public class Passenger {

 private String name;
 private String from;
 private String to;
 private String gender;

 private Address addr;

 // setters & getters

}

public class JavaToJson {

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

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

  Passenger passenger = new Passenger();
  passenger.setName("Raju");
  passenger.setFrom("Hyd");
  passenger.setTo("Delhi");
  passenger.setGender("Male");
  passenger.setAddr(addr);

  ObjectMapper mapper = new ObjectMapper();
  mapper.writeValue(new File("passenger.json"), passenger);

  System.out.println("Done....");
 }
}

public class JsonToJava {

 public static void main(String[] args) throws Exception{
  ObjectMapper mapper = new ObjectMapper();
  Passenger passenger = mapper.readValue(new File("passenger.json"), Passenger.class);
  System.out.println(passenger);
 }
}
```

==========GSON API Example ==========

==========

```
Gson gson = new Gson();

String json = gson.toJson(passenger);
System.out.println(json);

Passenger p = gson.fromJson(new FileReader("passenger.json"), Passenger.class);
System.out.println(p);
```

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

=> HTTP stands for Hypertext Transfer Protocol

=> HTTP acts as mediator between Client & Server

=> HTTP is a stateless protocol. It will treat every request as a new request.

Note: To develop REST API we should know below details about HTTP Protocol.

1) HTTP Methods

2) HTTP Status Codes

3) Http Request

4) HTTP Response

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

GET ===> To get data from server to client
POST ===> To send data from client to server
PUT ===> To update data at server
DELETE ===> To delete data from server

Note: Every REST API method/endpoint should be binded to HTTP Protocol method

=> getTicketData ( ) ---> HTTP GET Method ---> @GetMapping

=> bookTicket(..) ---> HTTP POST Method ----> @PostMapping

=> updateProduct (..) --> HTTP PUT Method ----> @PutMapping

=> deleteBook (..) ---> HTTP DELETE Method ---> @DeleteMapping

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

=> Server will send HTTP Status code to client in the response

=> HTTP Status codes will indicate how server processed our request

1XX (100 to 199) => Informational status code

2XX (200 to 299) => SUCCESS status code (OK)

3XX (300 to 399) => Redirectional

4XX (400 to 499) => Client Error

5XX (500 to 599) => Server Error

==========
HTTP Request Packet
==========

1) Request Line (HTTP Method + Request URL)

Ex: GET www.irctc.com/ticket/13454

2) Request Header (Meta data)

Ex :
Content-Type = application/json
Accept = application/json
Authentication = uname:pwd
Token = sldfjdlsfyso

3) Request Body (Payload)

Ex: xml or json data

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

1) Response Line (Https Status Code + Status Msg)

Ex: 200 OK

2) Response Header (Meta data)

Content-Type : application/json
Content-Length : 100
Date: mm/dd/yyyy

3) Response Body (Payload)

Ex: xml data or json data

==========
REST API Development
==========

=> It is very simple to develop REST API using Spring Boot

=> Spring Boot provided 'web-starter' to develop both web & distributed apps

=> 'Web-Starter' will provide tomcat as default embedded server

Step-1) Create Spring-Starter Project with below dependency

*** springboot-starter-web

Step-2) Create Rest Controller class using @RestController

Step-3) Write the required methods & bind them to HTTP Protocol Request

Step-4) Run the boot application (it will run in embedded server)

Step-5) Test our REST Application using POSTMAN tool

==========
Media Types
==========

consumes : It represents in which format REST API method can take input

produces : It represets in which format REST API method can provide output

Content-Type Header : It represents in which format client sending data to REST API in req body

Accept Header : It represents in which format client expecting response from REST API.

========== Media Types ==========

```
@XmlRootElement
@Data
public class Customer {

 private String name;
 private String email;
 private String gender;

}

@RestController
public class CustomerRestController {
```

```
 @GetMapping(
   value="/customer",
   produces = {"application/xml" , "application/json"}
 )
 public Customer getCustomer() {
  Customer c = new Customer();
  c.setName("John");
  c.setEmail("john@gmail.com");
  c.setGender("Male");
  return c;
 }

 @PostMapping(
   value = "/customer",
   consumes = {"application/xml", "application/json"},
   produces = {"text/plain"}
 )
 public ResponseEntity<String> addCustomer(@RequestBody Customer customer) {
  System.out.println(customer);
  // logic to insert customer in db
  return new ResponseEntity<>("Customer Saved", HttpStatus.CREATED);
 }
}
```

==========

Requirement : Develop REST API to book train tickets. It should contain 2 below endpoints

1) BooK Ticket -- POST Request

   Input : Passenger Data
   Output : Ticket data

   consumes : xml & json
   produces : xml & json

2) Get Ticket -- GET Request

   Input : Ticket ID
   Output : Ticket Data

   consumes : N/A
   produces : xml & json

==========
Development Procedure For Above Requirement
==========

1) Create Spring Boot Application with below dependencies

   a) web-starter
   b) lombok
   c) devtools

2) Create Request and Response Binding classes

3) Create REST Controller class with Required Methods

4) Bind Rest controller methods to HTTP Request methods

5) Run our application with Embedded Server

6) Test our application with POSTMAN.

==========
Passenger Data in JSON Format
==========

```
{
    "fname": "ashok",
    "lname": "kumar",
    "from": "Hyd",
    "to" : "Delhi",
    "doj" : "30/03/2023",
    "trainNum": "8878"
}
```

==========
Swagger
==========

=> In Distributed applications two actors will be available

1) Provider
2) Consumer

=> Provider will be developed by one company
=> Consumer will be developed by another company

=> If consumer wants to access provider, consumer side dev team should know provider information

-> What is provider api url ?
-> What operations (methods) provider having ?
-> Operations are binded to which Request Type (GET or POST or PUT or DELETE)?
-> What input provider expecting from consumer ?
-> What output provider will give to consumer ?
-> Which data form provider will support for input and output ?

Note: If consumer side dev team having all the above information then only then can start consumer side development.

Note: Provider side dev team should provide API documentation to consumer side dev team.

=> Swagger is used to generate API documentation.

=> Swagger is a third party library which is used to generate REST API documentation.

Note: Using Swagger Documentation Consumer side dev team will understand Provider API information.

==========
Steps to add Swagger Documentation for REST API
==========

1) Add swagger & swagger-ui dependencies in project pom.xml

```
 <dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger2</artifactId>
   <version>2.6.1</version>
  </dependency>
  <dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger-ui</artifactId>
   <version>2.6.1</version>
 </dependency>
```

2) Create SwaggerConfig class to generate documentation

```
@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();
 }
}
```

3) Run the application and access Swagger documentation in browser.

Json Doc URL : http://localhost:8080/v2/api-docs

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

NOte: Using Swagger UI we can test REST API functionality just like POSTMAN.

1) What is Distributed Application
2) Distributed Technologies
3) What is Intereoperability ?
4) What is HTTP Protocol ?
5) HTTP Request Packet

6) HTTP Response Packet
7) HTTP Methods
8) HTTP Status Codes

9) XML & JAX-B API
10) JSON & JACKSON API / GSON api
11) REST API Development (@RestController)
12) REST API Testing with POSTMAN
13) MEDIA TYPES (conumes & produces)
14) Content-Type & Accept headers
15) Swagger Documentation

16) Query Parameter (@RequestParam)
17) Path Variable (@PathVariable)
18) @RequestBody
19) ResponseEntity (combine resp body + http status code)

20) REST API Deployment in AWS Cloud

==========
REST Client Development
==========

=> The application which is accessing other applications is called as REST Client.

=> In Spring Boot we can develop REST Client in 3 ways

1) RestTemplate class (Synchronus)

2) WebClient interface (Synchronus & Async) (introduced in spring 5.x version)

3) FeiginClient interface (spring cloud)

Note: From Spring 5.x version onwards we have to use WebClient instead of RestTemplate.

----------application.properties----------
```
spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp

irctc.endpoint.book.ticket=http://3.110.190.17:8080/ticket
irctc.endpoint.get.ticket=http://3.110.190.17:8080/ticket/{ticketId}
```

```
@Service
public class MakeMyTripService {

 @Value("${irctc.endpoint.book.ticket}")
```

```
 private String IRCTC_BOOK_TICKET_URL;

 @Value("${irctc.endpoint.get.ticket}")
 private String IRCTC_GET_TICKET_URL;

 public Ticket getTicketInfo(String ticketId) {

  RestTemplate rt = new RestTemplate();

  ResponseEntity<Ticket> responseEntity =
    rt.getForEntity(IRCTC_GET_TICKET_URL, Ticket.class, ticketId);

  int status = responseEntity.getStatusCodeValue();
  if (status == 200) {
   Ticket ticket = responseEntity.getBody();
   return ticket;
  }

  return null;
 }

 public Ticket processTicketBooking(Passenger passenger) {

  RestTemplate rt = new RestTemplate();

  ResponseEntity<Ticket> responseEntity =
     rt.postForEntity(IRCTC_BOOK_TICKET_URL, passenger, Ticket.class);

  int statusCode = responseEntity.getStatusCodeValue();

  if (statusCode == 200) {
   Ticket ticket = responseEntity.getBody();
   return ticket;
  }
  return null;
 }
}
```

IRCTC API Doc URL : http://3.110.190.17:8080/swagger-ui.html

==========
WebClient
==========

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

-> Using WebClient interface we can develop REST Client logics

-> WebClient supports both sync & async communication.

Sync : Blocking Thread (After sending request we have to wait for response)

Async : Non Blocking Thread (After sending request we no need to wait for response)

RestTemplate (C) : spring-boot-starter-web

WebClient (I) : spring-boot-starter-webflux

============================== RestClient development with WebClient ================
================

```
@Service
public class MakeMyTripService {

 @Value("${irctc.endpoint.book.ticket}")
 private String IRCTC_BOOK_TICKET_URL;

 @Value("${irctc.endpoint.get.ticket}")
 private String IRCTC_GET_TICKET_URL;

 public Ticket getTicketInfo(String ticketId) {

   WebClient webClient = WebClient.create(); // get WeClient instance

   Ticket ticket = webClient.get() // represents HTTP GET request
         .uri(IRCTC_GET_TICKET_URL, ticketId) // ENDPOINT URL
         .accept(MediaType.APPLICATION_JSON)
         .retrieve() // take resp from response body
         .bodyToMono(Ticket.class) // bind resp body data to java obj
         .block(); // make sync call
   if(ticket!=null) {
    return ticket;
   }

  return null;
 }

 public Ticket processTicketBooking(Passenger passenger) {

   WebClient webClient = WebClient.create(); // get WeClient instance

   Ticket ticket = webClient.post()
            .uri(IRCTC_BOOK_TICKET_URL)
            .body(BodyInserters.fromValue(passenger))
            .header("Content-Type","application/json")
            .accept(MediaType.APPLICATION_JSON)
            .retrieve()
            .bodyToMono(Ticket.class)
            .block();
```

```
   if(ticket!=null) {
    return ticket;
   }
  return null;
 }
}
```

==========
Async Client Development
==========

```
package in.ashokit;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;

@Service
public class MakeMyTripService {

 @Value("${irctc.endpoint.book.ticket}")
 private String IRCTC_BOOK_TICKET_URL;

 @Value("${irctc.endpoint.get.ticket}")
 private String IRCTC_GET_TICKET_URL;

 public void getTicketInfoSync(String ticketId) {

   System.out.println("Sync - method started....");

   WebClient client = WebClient.create();

   String response = client.get()
         .uri(IRCTC_GET_TICKET_URL, ticketId)
         .accept(MediaType.APPLICATION_JSON)
         .retrieve()
         .bodyToMono(String.class)
         .block(); // wait for response

   System.out.println(response);

   System.out.println("Sync - method ended....");
 }


 public void getTicketAsync(String ticketId) {

  System.out.println("Async method execution started.....");

  WebClient client = WebClient.create();

     client.get()
        .uri(IRCTC_GET_TICKET_URL, ticketId)
        .accept(MediaType.APPLICATION_JSON)
```

```
        .retrieve()
        .bodyToMono(String.class)
        .subscribe(response -> handleResponse(response));

    System.out.println("Async method execution ended.....");
  }

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

==========

==========

Spring Data REST

==========

=> It is used to simplfiy REST API development

=> We no need to create REST Controllers to perform CRUD operations with DB table when we use Spri ng Data REST.

=> To use Data-REST in our project we need to add below dependency in pom.xml (REST REpositories)

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

```
@Entity
@Table(name = "book_tbl")
@Data
public class Book {

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

}
```

```
@RepositoryRestResource(path = "books")
public interface BookRepository extends JpaRepository<Book, Integer> {
  public List<Book> findByNameContaining(@Param("name") String name);
}
```

```
@Configuration
public class MyDataRestConfig implements RepositoryRestConfigurer {
```

```
 @Override
 public void configureRepositoryRestConfiguration(RepositoryRestConfiguration config, CorsRegistry cors
) {

  HttpMethod[] unsupportedMethods = { HttpMethod.PUT, HttpMethod.DELETE };

  config.getExposureConfiguration()
     .forDomainType(Book.class)
     .withItemExposure((metadata, http) -> http.disable(unsupportedMethods))
     .withCollectionExposure((metadata,http) -> http.disable(unsupportedMethods));

 }
}
```

==========
REST API Exception Handling
==========

-> Exception means un-expected and un-wanted situation

-> Exception will cause abnormal termination of our program

-> To achieve graceful termination, we need to handle exceptions in our application

-> In springboot, we can handle exceptions in 2 ways

  1) Local Exception Handling
  2) Gloabl Exception Handling

==========
Steps to implement Exception Handling
==========

1) Create boot application with web starter

2) Create RestController with required method

3) Create User Defined Exception class

4) Create ExceptionInfo binding class

5) Create Rest Controller Advice to handle global exceptions in our application.

```
package in.ashokit.exception;

public class CustomerNotFoundException extends RuntimeException {

 public CustomerNotFoundException() {
  // TODO Auto-generated constructor stub
 }
```

```
 public CustomerNotFoundException(String msg) {
  super(msg);
 }

}
```

```
@Data
public class ExceptionInfo {

 private String code;
 private String msg;
 private LocalDateTime date;
}
```

```
@RestControllerAdvice
public class AppExceptionHandler {

 @ExceptionHandler(value = CustomerNotFoundException.class)
 public ResponseEntity<ExceptionInfo> handleCnfe(CustomerNotFoundException cnfe) {

  ExceptionInfo info = new ExceptionInfo();
  info.setCode("EX0011");
  info.setMsg(cnfe.getMessage());
  info.setDate(LocalDateTime.now());

  return new ResponseEntity<>(info, HttpStatus.BAD_REQUEST);

 }
}
```

```
@Service
public class CustomerService {

 public String getCustomerNameById(Integer customerId) {
  if (customerId >= 100) {
   return "John";
  } else {
   throw new CustomerNotFoundException("Invalid customer id");
  }
 }
}
```

```
@RestController
public class CustomerRestController {

 @Autowired
 private CustomerService service;

 @GetMapping("/customer/{customerId}")
 public String getCustomerName(@PathVariable Integer customerId) throws Exception {
  return service.getCustomerNameById(customerId);
 }
}
```

==========
REST Architecture Principles
==========

REST : Representation State Transfer

1) Client Server Architecture (B 2 B)

2) No State / Session Management

3) Unique Addressability

4) Map REST API endpoints to HTTP Methods

5) MediaType Representation (consumes, produces & Content-Type, Accept)

6) HATEOS (Hypermedia as the Engine of Application State)

==========
HATEOS
========

-> It is one of the REST Architecture Principle

-> It is used to send response along with hyperlinks for related data

Ex:

URL : http://localhost:8080/customer/101

```
{
 "id" : 101,
 "name" : "John",
 "email" : "john@gmail.com",

 "links" :{
  "url" : "http://localhost:8080/customers"
 }
}
```

=> In springboot we have HATEOS starter to develop REST API with HATEOS concept.

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

@Data
@AllArgsConstructor
@NoArgsConstructor

```
public class User extends RepresentationModel<User> {

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

}
```

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import static org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*;

import in.ashokit.binding.User;

@RestController
public class UserRestController {

 @GetMapping("/user")
 public ResponseEntity<User> getUser() {

  User user = new User(101, "John", "john@gmail.com");

  user.add(linkTo(methodOn(UserRestController.class).getUser()).withSelfRel());

  return new ResponseEntity<>(user, HttpStatus.OK);
 }
}
```

PUT -> To update complete resource data (user : name, age, dob)

PATCH -> To update particular fields in resource (order : status = Delivered)
