**Rest Full Web Services & Micro services:**

**REST API & Micro Services:**

1) Type of Application.

2) What is distributed application?

3) Why we need to go for distributed applications

4) Distributed Technologies

5) Provider Development

6) Testing Using Postman

7) Swagger

8) Consumer Development

9) Exception Handling

10) Micro Services

11) Service Registry

12) Admin Server

13) Zipkin Server

14) API Gateway

15) Ribbon for lbr

16) Feign Client

17) Circuit Breaker

18) Kafka

19) Redis

20) Config Server

21) Security

**Types of Application:**

1. Standalone Application( Only one user can access at a time)
2. Web Application( Multiple users can access at a time)
3. Distribution Application (Web Services).

**What is Distribution Application?**

1. If one application communicate another application then it is called Distribution application.

Ex: 1. Make my Trip Application----->IRTC Application

2. Passport ------------> aadhar

3. gpay ----------------> banking

**Why to develop distribution application?**

=> for code reusability

=> for loosely coupling.

**Distribution Application Architecture:**

=> In distribution applications, 2 actions will be available

1) Provider: the app which is giving to service to other application

Ex: IRTC.

2) Consumer: the app which is accessing the service to other application.

Ex: MakeMyTrip, Yatra …

**What is Interoperability?**

=> Not language specification

=> Not platform specification

=> Irrespective of platform and language if two apps are communicating then those are called as interoperability applications.

Java -------------> .net

Java--------------> python

Note: To achieve intereoperability we will use xml/json format to transfer data from one application to another application.

=> XML & JSON formats are universal, all languages will understand these formats.

Note: To achieve intereoperability we will use xml/json format to transfer data from one application to another application.

=> XML & JSON formats are universal, all languages will understand these formats.

**Distribution Technologies:**

1) CORBA

2) RMI

3) EJB

4) SOAP Web services (Outdated)

- JAX-RPC

- JAX-WS

5) Restful Services (Trending)

- JAX-RS (outdated)

- Spring REST (trending)

**JSON:**

=> Json stands for **Java Script Object Notation**.

=> In Distributed applications we will use JSON as a mediator between Consumer & Provider.

=> JSON is intereoperable

=> JSON is very light weight

=> JSON will represent data in key-value format.

Syntax:

{

"Id": 101,

"Name”: "Ravi",

"Gender": "Male"

}

=> To convert java object to json data and json data to java object we will use "Jackson" API.

=> Jackson is a third party library.\

=> Jackson API provided "Object Mapper" class to perform conversions

Jackson API

Java obj <---------------------> json data

ObjectMapper mapper = new ObjectMapper ( );

writeValue(object); // convert java obj to json

readValue (String file); // convert json to java obj

**Java with Jackson API Example:**

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.16.1</version>

</dependency>

public class Customer {

private Integer cid;

private String cname;

private String cemail;

// setters & getters

}

|  |
| --- |
| public class ConvertJsonToObj {    public static void main(String[] args) throws Exception{    // json conversion to java obj    File f = new File("customer.json");  ObjectMapper mapper = new ObjectMapper();  Customer customer = mapper.readValue(f, Customer.class);  System.out.println(customer);    }  } |

**REST API development using Spring Boot:**

-> We need to use "web-starter" dependency in pom.xml

-> As part of REST API (Provider) development we will use below annotations

1) @RestController : To make our class as distributed component (B2B)

2) @RequestParam: To read query parameters from URL (K-V)

3) @PathVariable: To read path parameters

4) @RequestBody: To read data from request body

5) @GetMapping: To map rest controller method to HTTP GET request

6) @PostMapping: To map rest controller method to HTTP POST request

7) @PutMapping: To map rest controller method to HTTP PUT request

8) @DeleteMapping: To map rest controller method to HTTP DELETE request

|  |
| --- |
| @RestController  public class MsgRestController {  @GetMapping("/welcome")  public String getWelcomeMsg() {  String msg = "Welcome to REST API..!!";  return msg;  }  @GetMapping("/greet")  public String getGreetMsg(@RequestParam("name") String name) {  String msg = name + ", Good Morning..!!";  return msg;  }  } |

=> HTTP will act as mediator between consumer and provider

**HTTP Methods:**

1) GET: Method is responsible to send data to consumer

2) POST: Method is responsible to create a resource/record

3) PUT: Method is responsible to update a record

4) DELETE: Method is responsible to delete a record

Note: Rest Controller class methods will be mapped to HTTP methods

Note: GET method will not have request body.

=> POST, PUT and DELETE methods will have request body.

=> Request Body is used to send payload from consumer to provider in the form xml or json.

=> Response Body is used to send payload from Provider to consumer.

Ex: 1) MakeMyTrip will send passenger data to IRCTC in request body

(Xml/json)

2) IRCTC will send ticket data to MakeMyTrip in response body

(Xml/json)

**HTTP Status Codes:**

=> Provider will send response to consumer using HTTP Status code.

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

4xx (400 - 499): Client Error

5xx (500 - 599): Server Error

|  |
| --- |
| @RestController  public class MsgRestController {    @GetMapping("/greet")  public String getGreetMsg(@RequestParam("name") String name) {  String msg = name+", Good Morning..!!";  return msg;  }  @GetMapping("/welcome/{name}")  public ResponseEntity<String> getWelcomeMsg(@PathVariable("name") String name) {  String msg = name+", Welcome to REST API..!!";  return new ResponseEntity<>(msg, HttpStatus.OK);  }    @GetMapping("/action")  public ResponseEntity<Void> doAction() {  System.out.println("doAction () metdod called...");  return new ResponseEntity<>(HttpStatus.NO\_CONTENT);  }  } |

**REST API with JSON Response:**

1) Create spring boot app with web-starter

2) Create Binding class to represent data

|  |
| --- |
| public class Customer {  private Integer cid;  private String cname;  private String cemail;  } |

3) Create Rest Controller with Required methods

Ex: GET, POST, PUT and DELETE

|  |
| --- |
| @RestController  public class CustomerRestController {  @DeleteMapping(value = "/customer/{cid}", produces = "text/plain")  public String deleteCustomer(@PathVariable Integer cid) {  // db logic to delete  return "Customer Deleted";  }  @PutMapping(value = "/customer", consumes = "application/json", produces = "text/plain")  public String updateCustomer(@RequestBody Customer c) {  System.out.println(c);  // TODO: DB logic to update the record  return "Customer Updated";  }  @PostMapping(value = "/customer", produces = "text/plain", consumes = "application/json")  public ResponseEntity<String> addCustomer(@RequestBody Customer c) {  System.out.println(c);  // TODO: DB logic to insert record  String body = "Customer Added";  return new ResponseEntity<>(body, HttpStatus.CREATED);  }  @GetMapping(value = "/customer", produces = "application/json")  public Customer getCustomer() {  // logic to get record from db  Customer c = new Customer(1, "john", "john@gmail.com");  return c;  }  @GetMapping(value = "/customers", produces = "application/json")  public List<Customer> getCustomers() {  // logic to get records from db  Customer c1 = new Customer(1, "john", "john@gmail.com");  Customer c2 = new Customer(2, "smith", "smith@gmail.com");  Customer c3 = new Customer(3, "david", "david@gmail.com");  List<Customer> customers = Arrays.asList(c1, c2, c3);  return customers;  }  } |

**Consumes**: It represents in which format our rest api method can take input data from request body

ex: consumes = "application/json"

**Produces**: It represents in which format our rest api method can provide response to consumer in response body.

**Swagger:**

> It is used to generate documentation for REST API.

=> Provider app dev team should provide swagger documentation to consumer app dev team.

=> Swagger documentation will show provider information

1) Where is provider (URL?)

2) How is provider (what operations available)

3) Input format

4) Output format...

Note: Using Swagger, we can test provider functionality also.

=> Consumer app team will understand provider details using swagger documentation.

**Spring Boot 3.x + Swagger:**

=> Add below dependency in pom.xml file

<dependency>

<groupId>org.springdoc</groupId>

<artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

<version>2.1.0</version>

</dependency>

=> Run the application and access swagger-ui page with below url

URL : <http://localhost:8080/swagger-ui.html>

**IRCTC REST API development:**

=> Develop IRCTC api to book train tickets for passengers

=> We need to have below 2 functionalities in IRCTC API

1) book-ticket (POST Request method)

input : passenger data

output : ticket data

2) get all tickets (GET request method)

input : N/A

outupt : all tickets data

### Step-1 ) Identify request structure and create req binding class

Request : passenger info

name

email

doj

from

to

trainNum

### Step-2 ) Identify response structure and create resp binding class

Response : Ticket info

ticket-num

name

doj

from

to

trainNum

status

### Step-3 ) Creae Boot application with required classes

1) Passenger.java

2) Ticket.java

3) TicketService.java

4) TicketServiceImpl.java

5) TicketRestController.java

### Step-4) Run the application and test it using swagger ui

**Deploying Provider API in AWS cloud:**

## Reference Video : https://youtu.be/cRQPgbwOWq0?si=9b0Yuye0iFPoUB1W

1) Login into aws cloud account

2) Create Linux VM in aws cloud using EC2 service

3) Connect with Linux VM using MobaXterm

4) Upload our api jar file

5) Install java in linux vm

$ sudo yum install java

6) Run jar file

nohup java -jar <jarfile> &

7) Access our application documentation like below

URL : http://public-ip:8080/swagger-ui/index.html

Provider URL : <http://43.205.144.253:8080/swagger-ui/index.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 (synchronus - outdated)

2) WebClient (Sync & Async) - Spring 5.x

3) FeignClient (Interservice communication)

|  |
| --- |
| @Service  public class MakeMyTripService {  public Ticket bookTicket(Passenger p) {  String apiUrl = "http://43.205.144.253:8080/ticket";  RestTemplate rt = new RestTemplate();    ResponseEntity<Ticket> forEntity =  rt.postForEntity(apiUrl, p, Ticket.class);    Ticket body = forEntity.getBody();  return body;  }  public List<Ticket> getAllTickets(){  String apiUrl = "http://43.205.144.253:8080/tickets";  RestTemplate rt = new RestTemplate();    ResponseEntity<Ticket[]> forEntity =  rt.getForEntity(apiUrl, Ticket[].class);  Ticket[] body = forEntity.getBody();  List<Ticket> tickets = Arrays.asList(body);  return tickets;  }  } |

|  |
| --- |
| @Controller  public class MakeMyTripController {  @Autowired  private MakeMyTripService service;  @PostMapping("/ticket")  public String ticketBooking(@ModelAttribute("p") Passenger p, Model model) {  Ticket bookTicket = service.bookTicket(p);  model.addAttribute("msg", "Your Ticket Booked, ID : "+ bookTicket.getTicketNum());  return "bookTicket";  }  @GetMapping("/book-ticket")  public String bookTicket(Model model) {  model.addAttribute("p", new Passenger());  return "bookTicket";  }  @GetMapping("/")  public String index(Model model) {  List<Ticket> allTickets = service.getAllTickets();  model.addAttribute("tickets", allTickets);  return "index";  }  } |

|  |
| --- |
| <!doctype html>  <html lang="en">  <head>  <meta charset="utf-8">  <meta name="viewport" content="width=device-width, initial-scale=1">  <title>Bootstrap demo</title>  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet"  integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN" crossorigin="anonymous">  </head>  <body>  <div class="container">  <h1>Book Ticket Here</h1>  <font color='green'>  <p th:text="${msg}"></p>  </font>  <form th:action="@{/ticket}" th:object="${p}" method="post">  <table>  <tr>  <td>Name</td>  <td><input type="text" th:field="\*{name}" /></td>  </tr>  <tr>  <td>From</td>  <td><input type="text" th:field="\*{from}" /></td>  </tr>  <tr>  <td>To</td>  <td><input type="text" th:field="\*{to}" /></td>  </tr>  <tr>  <td>DOJ</td>  <td><input type="text" th:field="\*{doj}" /></td>  </tr>  <tr>  <td>Train Num</td>  <td><input type="text" th:field="\*{trainNum}" /></td>  </tr>  <tr>  <td></td>  <td><input type="submit" value="Book Ticket" class="btn btn-primary" /></td>  </tr>  </table>  </form>    <a href="/" clas="btn btn-danger">View Tickets</a>  </div>  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"  integrity="sha384-C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNXj1NW7RuBCsyN/o0jlpcV8Qyq46cDfL"  crossorigin="anonymous"></script>  </body>  </html> |

|  |
| --- |
| <!doctype html>  <html lang="en">  <head>  <meta charset="utf-8">  <meta name="viewport" content="width=device-width, initial-scale=1">  <title>Bootstrap demo</title>  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/css/bootstrap.min.css" rel="stylesheet"  integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN" crossorigin="anonymous">  </head>  <body>  <div class="container">  <a href="book-ticket" class="btn btn-primary">Book Ticket</a>  <h1>View Tickets</h1>  <table class="table table-bordered table-striped">  <thead>  <tr>  <th>Ticket Num</th>  <th>Passenger Name</th>  <th>From</th>  <th>To</th>  <th>DOJ</th>  <th>Train Num</th>  <th>Status</th>  </tr>  </thead>  <tbody>  <tr th:each="t : ${tickets}">  <td th:text="${t.ticketNum}"></td>  <td th:text="${t.name}"></td>  <td th:text="${t.from}"></td>  <td th:text="${t.to}"></td>  <td th:text="${t.doj}"></td>  <td th:text="${t.trainNum}"></td>  <td th:text="${t.status}"></td>  </tr>  </tbody>  </table>  </div>  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.2/dist/js/bootstrap.bundle.min.js"  integrity="sha384-C6RzsynM9kWDrMNeT87bh95OGNyZPhcTNXj1NW7RuBCsyN/o0jlpcV8Qyq46cDfL"  crossorigin="anonymous"></script>  </body>  </html> |

**Rest Template:**

=> Predefined class in spring web mvc

=> We can send http requests using this class

=> It is used for consumer development

=> we have below methods

1) getForEntity(..)

2) postForEntity(..)

3) putForEntity(..)

4) DeleteForEntity (..)

=> It supports only synchronous communication.

=> Synchronous means, after sending request, consumer side thread will wait until we get response from provider.

Note: Asynchronous means after sending request, consumer side thread will not wait for provider response.

=> To work with Asynchronous communication we will use Web Client which introduced in spring 6.x version.

Note: Web client supports both sync & async communications.

=> Web Client is an interface which is part of 'web-flux' starter.

=> Web Flux starter will provide 'netty' as default embedded container.

|  |
| --- |
| @Service  public class MakeMyTripService {  public Mono<Ticket> bookTicket(Passenger p) {  String apiUrl = "http://43.205.144.253:8080/ticket";    WebClient webClient = WebClient.create();    Mono<Ticket> bodyToMono = webClient.post()  .uri(apiUrl) .body(BodyInserters.fromValue(p))  .retrieve()  .bodyToMono(Ticket.class);  return bodyToMono;  }  public Mono<Ticket[]> getAllTickets() {  String apiUrl = "http://43.205.144.253:8080/tickets";  WebClient webClient = WebClient.create();    Mono<Ticket[]> bodyToMono = webClient.get()  .uri(apiUrl)  .retrieve()  .bodyToMono(Ticket[].class);  return bodyToMono;  }  } |

|  |
| --- |
| @Controller  public class MakeMyTripController {  @Autowired  private MakeMyTripService service;  @PostMapping("/ticket")  public String ticketBooking(@ModelAttribute("p") Passenger p, Model model) {  Mono<Ticket> bookTicket = service.bookTicket(p);  model.addAttribute("ticket", bookTicket);  model.addAttribute("msg", "Your Ticket Booked");  return "bookTicket";  }  @GetMapping("/book-ticket")  public String bookTicket(Model model) {  model.addAttribute("p", new Passenger());  model.addAttribute("ticket", new Ticket());  return "bookTicket";  }  @GetMapping("/")  public String index(Model model) {  Mono<Ticket[]> allTickets = service.getAllTickets();  model.addAttribute("tickets", allTickets);  return "index";  }  } |

**Exception Handling in REST API:**

1) Local Exception Handling: Specific to particular controller class

2) Global Exception Handling: Throughout application

|  |
| --- |
| public class ProductNotFoundException extends RuntimeException{    public ProductNotFoundException() {    }    public ProductNotFoundException(String msg) {  super(msg);  }  } |

|  |
| --- |
| @RestController  public class ProductRestController {  @GetMapping("/product/{pid}")  public String getProductInfo(@PathVariable Integer pid) {  if( pid > 100) {  throw new ProductNotFoundException("Invalid Product ID");  }    return "Keyboard - 2500 INR";  }  } |

|  |
| --- |
| @RestControllerAdvice  public class AppExceptionHandler {  @ExceptionHandler(value = Exception.class)  public ResponseEntity<ExInfo> handleEx(Exception e) {  ExInfo info = new ExInfo("EX001", e.getMessage(), LocalDate.now());  return new ResponseEntity<>(info, HttpStatus.INTERNAL\_SERVER\_ERROR);  }    @ExceptionHandler(value = ProductNotFoundException.class)  public ResponseEntity<ExInfo> handleProductNFEx(ProductNotFoundException e) {  ExInfo info = new ExInfo("EX002", e.getMessage(), LocalDate.now());  return new ResponseEntity<>(info, HttpStatus.BAD\_REQUEST);  }  } |

**Working with XML data in REST API:**

-> Add below dependency in pom.xml file

<dependency>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

</dependency>

Consumes: It represents input data format of provider (request body data)

Consumes = {"application/xml", "application/json"}

Produces: It represents output data format of provider (response body data)

Produces = {"application/xml", "application/json"}

Accept: It represents in which format consumer expecting response from provider

Ex: Accept = application/xml

Accept = application/json

Content-Type: It represents in which format consumer sending data to provider

Ex: Content-Type = application/xml

Content-Type = application/json

|  |
| --- |
| @RestController  public class BookRestController {  @PostMapping(  value = "/book",  consumes= {"application/xml", "application/json"},  produces = "text/plain"  )  public String addBook(@RequestBody Book b) {  System.out.println(b);  // logic to save in db  return "Book Added";  }  @GetMapping(  value = "/book",  produces = {"application/xml", "application/json"}  )  public Book getBook() {  Book book = new Book(101, "Java", 543.00);  return book;  }  } |

**Spring Data REST:**

=> It is used to simplify REST API development

RestController + JpaRepository = RestRepositoryResource

1) Create Spring starter project with below dependencies

a) rest-repositories (data-rest)

b) data-jpa

c) mysql-connector-j

d) devtools

2) Configure datasource properties

3) Create Entity class for table mapping

@Entity

@Table(name = "book\_tbl")

public class Book {

@Id

private Integer id;

private String name;

private Double price;

}

4) Create Rest Repository to expose REST API methods

@RepositoryRestResource(path = "books")

public interface BookRepository extends JpaRepository<Book, Integer>{

public List<Book> findByName(@Param("name") String name);

}

5) Run the application and test it using POSTMAN

POST : http://localhost:9090/books

{

"id": 103,

"name": "DevOps",

"price": 1500.20

}

PUT : http://localhost:9090/books/103

{

"id": 103,

"name": "DevOps",

"price": 2500.20

}

**How to disable HTTP Requests in Data REST application?:**

=> Add below class in Data REST Application to disable PUT and DELETE methods.

Note: With below configuration our api will not accept PUT and DELETE methods.

|  |
| --- |
| @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));    }  } |

Assignment: How to send Async request using WebClient.

=> In below class getQuoteV1() method will act as synchronus client and getQuoteV2() method will act as asynchronus client.

|  |
| --- |
| @Service  public class QuoteService {    private static final String QUOTE\_ENDPOINT = "https://type.fit/api/quotes";  public void getQuoteV2() {  WebClient client = WebClient.create();  System.out.println("Rest api call started...");  client.get()  .uri(QUOTE\_ENDPOINT)  .header("Accept", "application/json")  .retrieve()  .bodyToMono(String.class)  .subscribe(QuoteService::handleResponse);    System.out.println("Rest api call completed...");  }  public static void handleResponse(String s) {  System.out.println(s);  // logic  }  public void getQuoteV1() {  WebClient client = WebClient.create();  System.out.println("Rest api call started...");  Mono<String> bodyToMono = client.get()  .uri(QUOTE\_ENDPOINT)  .retrieve()  .bodyToMono(String.class);  System.out.println(bodyToMono.block());  System.out.println("Rest api call completed...");  }  } |