Yesterday's session : First REST App Development

Request Mappings

-> We can develop REST API using Spring with REST

-> Spring With REST is part of Spring Web MVC

-> If we add Spring Web MVC Jars then we can develop REST API

-> To Develop REST API in Spring Boot Project we need to add spring-boot-starter-web in pom.xml file.

-> To develop REST component we will use below annotation

@RestController

Note: This annotation is available from Spring version 4.o

-> Before @Restcontroller annotation people used to work with @Controller + @ResponseBody.

-> Rest Controller methods we have to bind with Http Protocol methods.

-> To bind our methods to HTTP protocol method we will use below annotations

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping

->RestController methods will return Raw Response in Response Body.

-> To check apps in google we can use below URL

chrome://apps/

What is Rest Resource

REST Resource is a distributed component which provides business services to other applications over a network.

What is Rest Client

Rest Client can be a human, can be a device, can be an application and can be a tool...

Anybody/anything which is capable of sending Http request can be treated as Rest Client.

-> In RestController, If we have more than one method binded same HTTP Request method then we should unique urls for both methods.

-> If we have more than one RestController in Project, it is higly recommended to specify class level URL pattern.

```
@RestController
@RequestMapping(value="/demo")
public class DemoController {
}
```

Note: Writing class level url-pattern is optional

-> Writing method level url pattern is optional if we have only one method which is binded to particular Http request method.

-> If we have more than one method binded to particular HTTP Request then we should choose unique url pattern for all methods.

Note: It is recommended to write both class level and method level url patterns so that we can avoid ambiguity problems in url-patterns.

Today's session : Request Parameters

-> To develop Distributed component in Spring MVC/Spring Boot we will use @RestController annotation

-> If we want to make our method as distributed method we should bind that to Http Request Methods

-> To bind our methods to Http Request methods we will use below annotations

@GetMapping

@PostMapping

@PutMapping

@DeleteMapping

Http GET Request

When we should bind our method to HTTP GET Request?

->Based on Client Request, if Resource wants to send data to client then we should bind that method to GET Request method.

-> To bind the method to GET request method we will use @GetMapping annotation

```
@GetMapping(value="/hi")
public String sayHi(){
  return "Hi, Ashok";
}
```

Ex:

Client wants to get Product Info

Client wants to one employee info

Client wants to get book info

-> HTTP GET method is idempotent method (Safe)

-> Idempotent methods are safe methods, if we send same request for multiple times also they won't change resources(data) at the server

-> For GET Request method Request Body will not be available

How to send data to server in GET Request?

To send data in GET Request we can use below things

1) Query Parameters

2) Path Parameters

Notes: Query Parameters' & Path Parameters will be present in URL.

Query Parameters

-> Query Parameters are used to send data from client to server in URL.

-> Query Parameters will represent data in Key-Value pair format

Ex : ?id=101

Ex: ?id=101&name=Ashok

Note: Query Parameters should present only at end of the URL.

REST Endpoint URI :http://www.usa.gov.in/getFpdScores

Input : SSN Number as Query Parameter (key is ssn)

http://www.usa.gov.in/getFpdScores?ssn=797979799

-> To read Query Parameter in Resource method we will use @RequestParam annotation

Syntax

```
@GetMapping("/hello")
public String sayHello(@RequestParam("name") String name){
        return "Hello, "+name;
}
```

Query Parameter application

```
@RestController
@RequestMapping("/fpd")
public class FpdScoresRestController {

    @GetMapping(value="/score")
    public String getFpdScores(@RequestParam("ssn") Long ssn) {
        //logic retrive from db
        return "FPD Score Is : 75";
    }
}
```

REST Endpoint URL : http://localhost:8081/fpd/score?ssn=897878897

http://localhost:8081/fpd/score

Note: When we are using @RequestParam annotation, Query Parameters are mandatory in URL to access Resource.

-> We can make Query Parameters as Optional using below approach

```
    @GetMapping(value = "/score")
    public String getFpdScores(
            @RequestParam(
                    value = "ssn",
                    required = false,
```

```
                defaultValue = "7797997979") Long ssn) {

        System.out.println("SSN :: " + ssn);
        // logic retrive from db
        return "FPD Score Is : 75";
    }
```

-> When we make Query Parameter is optional, we can consider defaultValue like above

->string is converted to long how?? dispatcher servlet is everything handle.

Path Parameters in REST API

-> Request Params are called as Query Parameters

-> Query Params are used to send data to server in the URL using key-value pair format

-> Query Params should be present only at end of the URL

-> Query Params starts with ? and seperated by &

Ex: www.ashokit.in?course=jrtp&trainer=ashok

How to access Query Parameters in REST Controller ?

-> By using @RequestParam annotation we can access Query Parameters coming in the URL

```
@GetMapping("/hello")
public String sayHello(@RequestParam("name") String s){
        return "Hello, "+name;
}
```

-> By Default Query Params are mandatory in Spring MVC

Working with Multiple Query params in REST API

```
@GetMapping("/course")
    public String getCourseDetails(@RequestParam("cname") String courseName,
                @RequestParam("tname") String trainerName) {
```

```
        if (courseName.equals("JRTP") && trainerName.equals("Ashok")) {
            return "Starting from 20-July-2020 @11:30 AM IST";
        }
        return "Please Check Our Website http://www.test.in For More Details";
    }
```

Path Parameter

-> Path Parameters are called as URI parameters

-> Path Parameters are used to send data from client to server in URL

Ex: www.ashokit.in/course/{JRTP}/{Ashok}

-> Path Parameters Can Present Anywhere in the URL

Ex : www.ashokit.in/course/{JRTP}/fastrack/{Ashok}

Working with Path Parameters in Rest Controller

-> To Read Path Parameters in RestController we will use @PathVariable annotation.

We will work with below 3 examples

1) Working with Single Path Parameter

2) Working with more than one path parameter

3) Working with Path Params in Middle Of URL

URL pattern : www.ashokit.in/course/name/{course-name}

Actual URL : www.ashokit.in/course/name/jrtp

Single Path Parameter

```
@GetMapping(value = "/course/{name}")
    public String getCourseDuration(@PathVariable("name") String courseName) {
        if ("JRTP".equals(courseName)) {
            return "Duration : 3 Months";
        } else if ("SBMS".equals(courseName)) {
            return "Duration : 75 Days";
        } else {
```

```
            return "Please check our website for more details : www.ashokit.in";
        }
    }
```

Multiple Path Parameters

```
@GetMapping(value="/course/{cname}/{tname}")
    public String getCourseDetails(
                        @PathVariable("cname") String cname,
                        @PathVariable("tname") String tname) {

        if ("JRTP".equals(cname) && "Ashok".equals(tname)) {
            return "Starting new batch From 20-Jul-2020 @11:30 AM IST";
        } else if ("SBMS".equals(cname) && "Ashok".equals(tname)) {
            return "Starting new batch from 29-Jul-2020 @7:15 AM IST";
        } else {
            return "Please check our website for more details : www.ashokit.in";
        }
    }
```

Using Path Params in Middle Of URL

```
@GetMapping(value = "/course/{cname}/fastrack/{tname}")
    public String getDetails(@PathVariable("cname") String cname, @PathVariable("tname")
String name) {
        return "Registration Process Is Started";
    }
```

Today' session : Produces formats

-> What is Path Parameter?

-> How to represent Path Param in Rest Controller?

```
@GetMapping(value="/course/{id}/details")
```

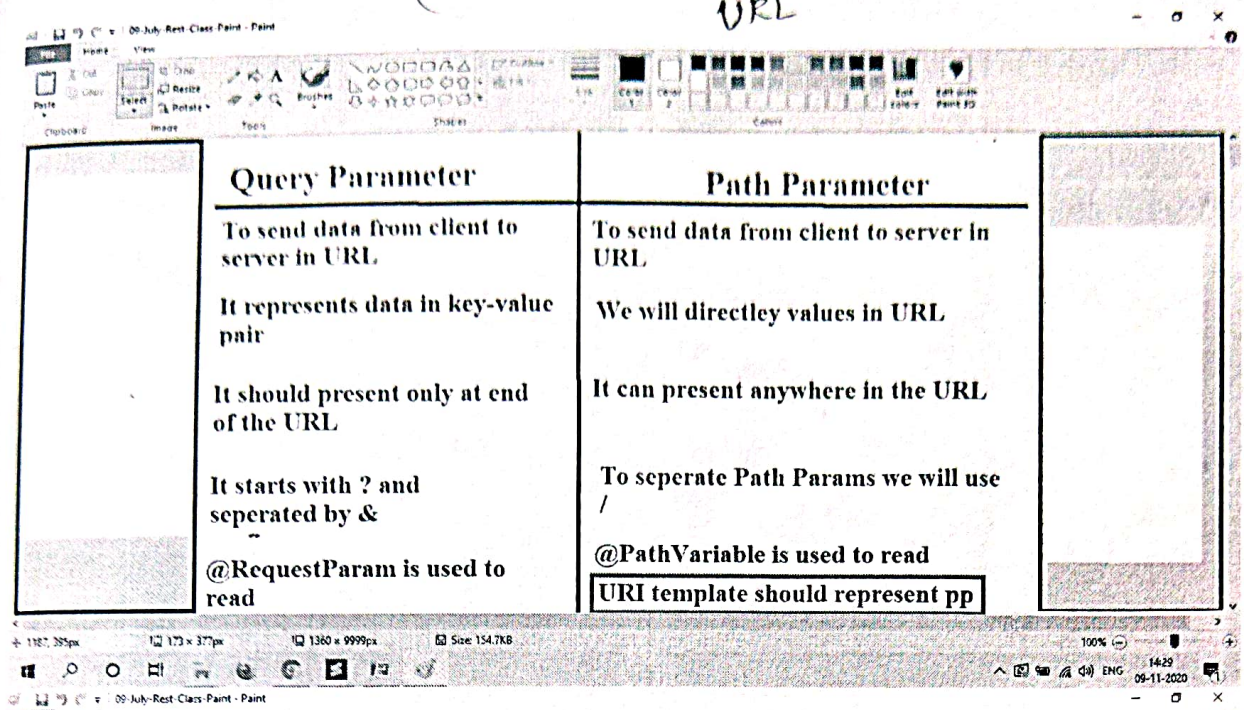
/course/jrtp/details

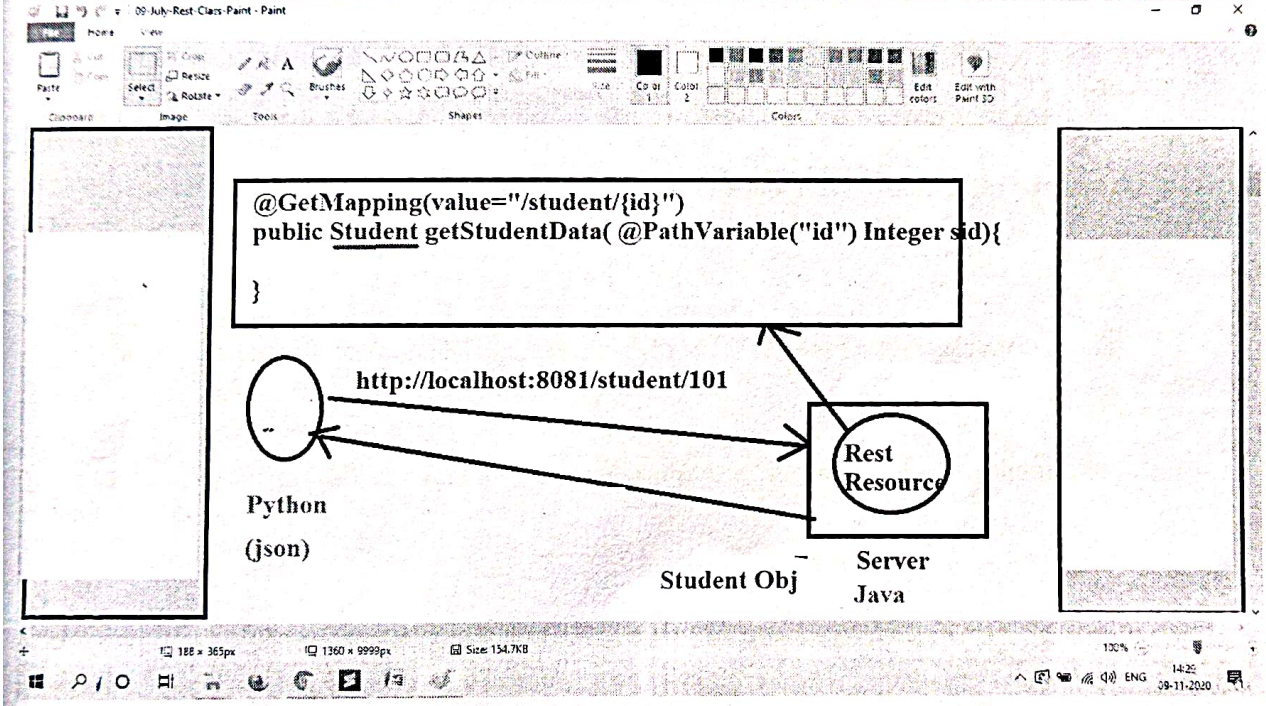
```
public String getDetails(@PathVariable("id") String s){
}
```

Taking Object as Return Type For Controller method

```
@RestController
public class StudentRestController {

    @GetMapping(value = "/student/{sid}", produces = "application/json")
    public Student getStudentDetails(@PathVariable("sid") Integer studentId) {
        Student s = new Student();

        if (studentId == 101) {
            s.setStudentId(101);
            s.setStudentName("Raju");
            s.setStudentSkill("Java");
        } else if (studentId == 102) {
            s.setStudentId(102);
            s.setStudentName("Rani");
            s.setStudentSkill("Testing");
        }

        return s;
    }
}
```

(Uniform Resource Locator)
URL





Today's session : JSON

-> JSON stands for Java Script Object Notation

-> JSON format is universal format to exchange data over a network

-> JSON is intereoperable (Platform Indepdent & Language Independent)

-> When we compare JSON with XML, JSON is light weight

(it occupies less memory)

-> JSON will represent the data in Key-Value Pair Format

Syntax:

```
{
  "sid" : 101,
  "sname" : "Raju",
  "skill" : "Java
}
```

```
<?xml version="1.0" encoding="UTF-8">
<student>
<sid>101</sid>
<sname>Raju</sname>
<skill>Java</skill>
</student>
```

Intro To Http Msg Converters

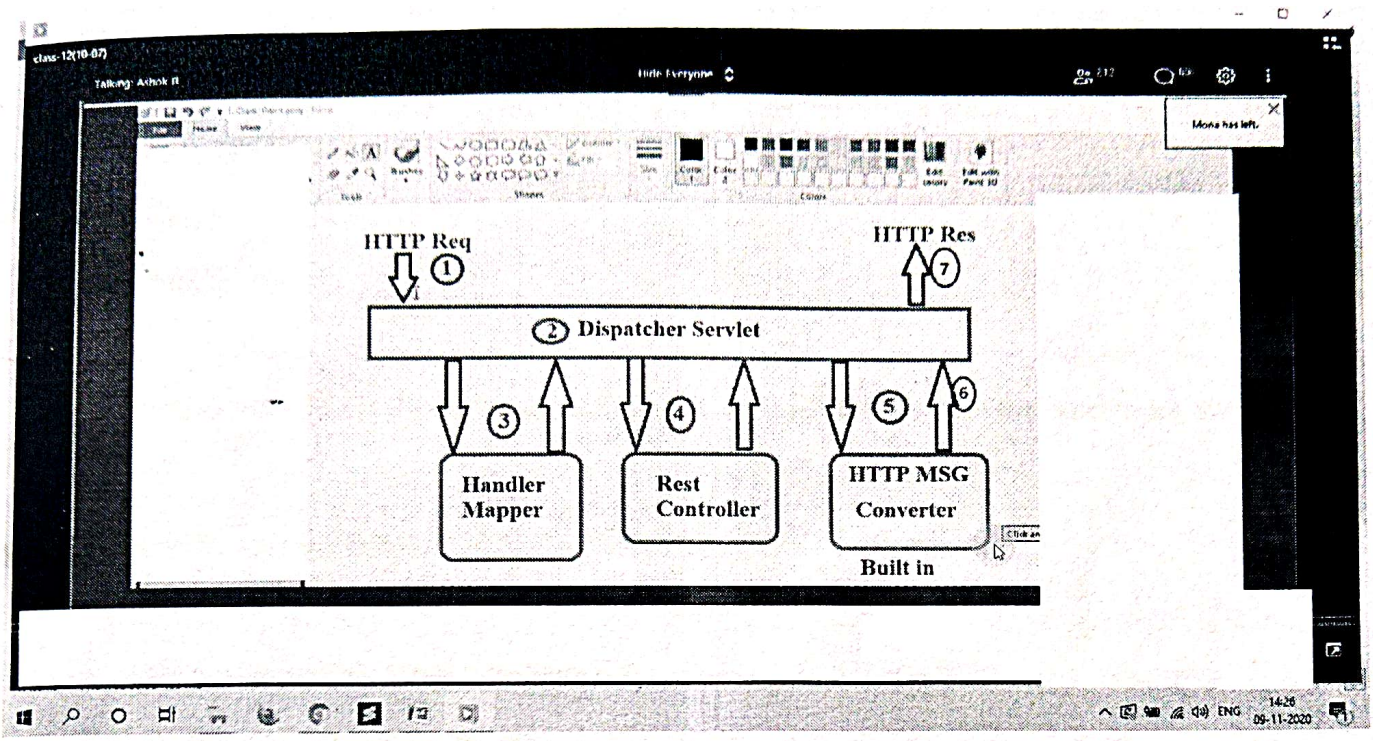
-> When we are developing REST Controller method we should not write logic to convert our object to json directley because it will be tightly coupled with only json format.

-> To achieve this loosely copuling Spring MVC provided HTTP Msg Converters for us.

-> These msg converters will convert our controller method returned object to client understable format (json or xml)

-> In Spring MVC, by default object will be converted JSON format (if required we can choose xml also)

-> To convert Java Object to JSON, internally MSG Converter will use JACKSON API.



Today's session : Working with Jackson API

-> When we develop a REST api we will design our methods with Objects

-> If we expect objects from clients as input or if we return objects to clients directley then we can't achieve intereoperability

-> To achieve intereoperability can we directley take/provide json or xml for input and ouput?

-> If we directley deal with json data or xml data in REst api methods then our logic will be tightly coupled with format of data.

-> To avoid this problem, Spring MVC provided Http Msg Converters

-> These Msg Converters are responsible to convert objects of data into intereoperable format and vice versa.

Json

-> Java Script Object Notation

-> It is light weight when compared with xml

-> Now a days it is the defacto standard to exchange data over a network.

-> Json will represent data in key value format

syntax:

```
{
  "student-id" : 101,
  "student-name" : "Raju",
  "student-skill" : "Java
}
```

-> While working with REST API development we will deal with JSON data for both Input and Ouput.

How JSON data can be used in Java Applications ?

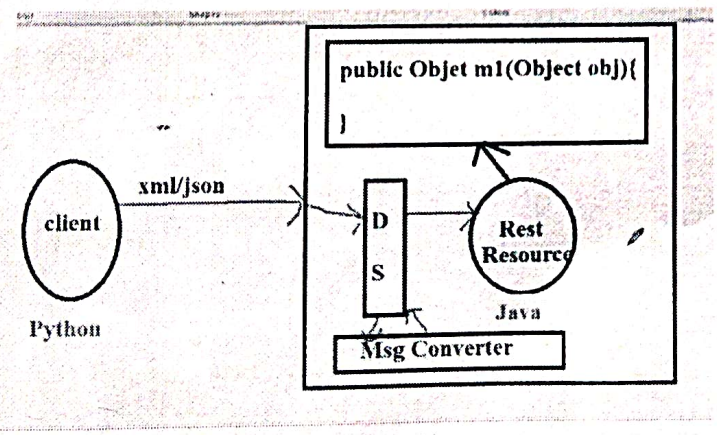
-> To work with JSON data we have Jackson API

-> Jackson API is JSON Processor For Java

-> Using Jackson, we can convert java object to json and json data to java object.

Note: JSON is universal format, where as Jackson is java specific api.

-> Every programming language will have their own apis to deal with JSON data.



Requirement

Develop a standalone application and perform below operations

1) Convert Java Object to Json format

2) Convert JSON Data to Java Object

Steps to develop application using Jackson

1) Create Standalone Maven Application

2) Add Below dependencies in pom.xml file

a) Jackson

b) Project Lombok

3) Create Pojo (to represent data)

4) Create Converter class to convert obj to json and vice versa.

```
<dependency>
                <groupId>com.fasterxml.jackson.core</groupId>
                <artifactId>jackson-databind</artifactId>
                <version>2.11.1</version>
        </dependency>
```

```
@Data
public class Student {

        private Integer studentId;
        private String studentName;
        private String studentSkill;
}
```

```
public class JavaToJsonConverter {
        public static void main(String[] args) throws JsonProcessingException {
                Student s = new Student();
                s.setStudentId(201);
```

```
        s.setStudentName("John");
        s.setStudentSkill("Java");

        ObjectMapper mapper   = new ObjectMapper();
        String jsonString = mapper.writerWithDefaultPrettyPrinter()
                                                    .writeValueAsString(s);
        //String jsonString = mapper.writeValueAsString(s);
        System.out.println(jsonString);
    }
}
```

Last session : working with Jackson API

Today' session : Jackson Examples

-> Jackson api provided JSON parser for java applications

-> Using JSON Parser we can convert java object to json and vice versa

```
ObjectMapper mapper = new ObjectMapper( );
```

-> ObjectMapper class provided methods to perform conversions.

Converting Object To Json

```
String json = mapper.writeValueAsString(obj);
```

-> When we convert java object to json, it considers class variable names as json key names.

-> If we want to bind our variables to custom keys then we can use @JsonProperty annotation

---

```
@Data
public class Student {

    @JsonProperty("student-id")
    private Integer studentId;

    @JsonProperty("student-name")
    private String studentName;
```

```
    @JsonProperty("student-skill")
    private String studentSkill;

}
```

-> At the time of serializing java object to json, if we want to ignore one field then we can use @JsonIgnore annotation at the field.

```
@JsonIgnore
private Integer studentAge;
```

-> To ignore multiple fields we can use below annotation

```
@Data
@JsonIgnoreProperties({ "student-skill", "studentAge" })
public class Student {
        @JsonProperty("student-id")
        private Integer studentId;

        @JsonProperty("student-name")
        private String studentName;

        @JsonProperty("student-skill")
        private String studentSkill;

        private Integer studentAge;
}
```

Note: @JsonIgnoreProperties will ignore properties irrespective of data present in properties (always those properties will be ignored)

-> At the time of serializing java obj to json we can include only the fields which contains data using eblow annotation

@JsonInclude

```
@Data
@JsonInclude(value = Include.NON_NULL)
public class Student {
	@JsonProperty("student-id")
	private Integer studentId;

	@JsonProperty("student-name")
	private String studentName;

	@JsonProperty("student-skill")
	private String studentSkill;

	private Integer studentAge;
}
```

@JsonProperty : To bind field with json key

@JsonIgnore : To ignore particular field from JSON

@JsonIgnoreProperties : To ignore more than one field

@JsonInclude : To specifiy which fields should include in json

Converting Json File Data To Java Object

----------Binding class----------

```
@Data
@JsonInclude(value = Include.NON_NULL)
public class Student {
        @JsonProperty("student-id")
        private Integer studentId;

        @JsonProperty("student-name")
        private String studentName;

        @JsonProperty("student-skill")
        private String studentSkill;

        private Integer studentAge;
}
```

----------json file----------

```
{
 "studentAge" : 20,
 "student-id" : 201,
 "student-name" : "John",
 "student-skill" : "Java"
}
```

---------- Converter class method----------

```
public static void convertJsonToObj() throws Exception{
                File f = new File("student.json");
                ObjectMapper mapper = new ObjectMapper();
                Student student = mapper.readValue(f, Student.class);
                System.out.println(student);
 }
```

Yesterday's session : Jackson annotations

Today' session : Working with JSON

-> Jackson provided json parser to work with json data in java applications

-> Using JSON Parser we can convert java objects to json and vice versa.

-> To perform these conversions we have ObjectMapper class.

```
ObjectMapper mapper = new ObjectMapper( );

//convert object to json
String json = mapper.writeValueAsString(obj);
```

Note: We can convert single object and we can convert list of objects also

-> When we have more than one object json represents in array format.

```
//convert json data to java object
ObjectMapper mapper = new ObjectMapper( );
Object obj = mapper.readValue(jsonFile, Class type);
```

For Single Student : http://localhost:8081/student/101

For All Students : http://localhost:8081/students

```
@Data
@JsonInclude(value=Include.NON_NULL)
public class Student {
```

```
    @JsonProperty("student-id")
    private Integer studentId;

    @JsonProperty("student-name")
    private String studentName;

    @JsonProperty("student-skill")
    private String studentSkill;

}
```

```
@Data
public class Students {
        private List<Student> students;
}
```

```
package com.ashokit.resources;

import java.util.ArrayList;
import java.util.List;

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

import com.ashokit.response.Student;
import com.ashokit.response.Students;

@RestController
public class StudentRestController {
```

```
@GetMapping(value = "/student/{sid}")
public Student getStudentDetails(@PathVariable("sid") Integer studentId) {
        Student s = new Student();
        if (studentId == 101) {
                s.setStudentId(101);
                s.setStudentName("Raju");
                s.setStudentSkill("Java");
        } else if (studentId == 102) {
                s.setStudentId(102);
                s.setStudentName("Rani");
                s.setStudentSkill("Testing");
        }
        return s;
}

@GetMapping(value = "/students")
public Students getStudentsDetails() {
        List<Student> slist = new ArrayList<Student>();

        Student s1 = new Student();
        s1.setStudentId(101);
        s1.setStudentName("Raju");
        s1.setStudentSkill("Java");

        Student s2 = new Student();
        s2.setStudentId(102);
        s2.setStudentName("Rani");
        //s2.setStudentSkill("Testing");

        slist.add(s1);
        slist.add(s2);
```

```
        Students students = new Students();
        students.setStudents(slist);

        return students;
    }
}
```

Yesterday's session : Jackson API

Today's session : JAX-B API

-> XML stands for Extensible Markup Language

-> XML governed by w3c

-> The initial version of xml is 1.0 & current version of xml is also 1.0

-> XML will reprsent the data in elements

ex: <id>101</id>

-> One element contains start tag and end tag

-> Every xml will start with prolog

<?xml version="1.0" encoding="UTF-8">

-> Prolog represents processing instructions

-> Every XML should contain only one root element

```
<?xml version="1.0" encoding="UTF-8">
<student>
    <id>101</id>
    <name>John</name>
    <rank>101</rank>
</student>
```

-> In the above xml we have 2 types of elements

1) Simple Element

2) Compound Element

What is Simple Element?

The element which represents data directley is called as Simple Element

Ex: <id>101</id>

What is Compound Element?

The Element which represents child elements is called as Compound Element.

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

-> We can use attributes also in the xml

ex: <book-name type="java">Spring</book-name>

-> XML is intereoperable

What is intereoperability?

Language Independent + Platform Independent

->XML is the defacto standard to exchange data among applications in Webservices.

-> SOAP Webservices will support only xml format for exchanging data

-> RESTful services supports for xml and some other formats also like json.

JAX-B

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

-> Using JAX-B we can convert java object into xml and xml into java object

Using JAX-B api we will perform mainly 2 operations

---

1) One Time operation

2) Runtime operations

What is Onetime operations?

Onetime Operation means desigining binding classes

Binding class will represent structure of the xml

The java class which represents structure of the xml is called as Binding class.

Note: We will create Binding class only one time.

What are Runtime Operations?

-> Marshalling & Un-Marshalling are called as Runtime operations.

-> The process of converting java object into xml is called as Marshalling

-> The process of converting xml into java object is called as Un-Marshalling.

Note: These operations will happen when the application is running.

-> To perform Runtime operations Binding classes are mandatory.

Working with JAX-B

-> JAX-B api & its implementation is part of JDK only

-> We no need to add any additional dependencies to work with jax-b.

----------Binding Class----------

```
@Data
@XmlRootElement(name = "student")
public class Student {

        private Integer id;
        private String name;
        private Integer rank;

}
```

```
package com.ashokit.converters;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;

import com.ashokit.bindings.Student;

public class JavaToXmlConverter {

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

                Student s = new Student();
                s.setId(201);
                s.setName("Peter");
                s.setRank(80);

                JAXBContext jaxbContext = JAXBContext.newInstance(Student.class);
                Marshaller marshaller = jaxbContext.createMarshaller();
                marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
```

```
        marshaller.marshal(s, System.out);
    }
}
```

Last session : JAX-B examples

Today' session : Un-Marshalling

-> JAX-B stands for java architecture for xml binding

-> Using JAX-B api, we can convert java object to xml and vice versa

-> In jax-b we will perform 2 operations

a) One Time Operation

- Designing binding classes

b) Runtime operations

- Marshalling (Java obj ---> XML)
- Un-Marshalling (XML --> Java Obj)

-> What is Un-Marshalling ?

The process of converting xml data to java object is called as Un Marshalling.

Note: To perform Marshalling/Un-Marshalling we need Binding class.

What is Binding class ?

The class which is representing structure of xml is called as Binding class.

----------Student.java----------

```
@Data
@XmlRootElement(name = "student")
@XmlAccessorType(XmlAccessType.FIELD)
public class Student {

    @XmlElement(name = "student-id")
    private Integer id;
```

```
    @XmlElement(name = "student-name")
    private String name;

    @XmlElement(name = "student-rank")
    private Integer rank;

}
```

----------Marshaller----------

```
package com.ashokit.converters;

import java.io.FileOutputStream;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Marshaller;

import com.ashokit.bindings.Student;

public class JavaToXmlConverter {
    public static void main(String[] args) throws Exception {
        Student s = new Student();
        s.setId(201);
        s.setName("Peter");
        s.setRank(80);

        JAXBContext jaxbContext = JAXBContext.newInstance(Student.class);
        Marshaller marshaller = jaxbContext.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        marshaller.marshal(s, System.out);
        marshaller.marshal(s, new FileOutputStream("student.xml"));
    }
```

```
}
```

----------Un-Marshaller----------

```
package com.ashokit.converters;

import java.io.File;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;

import com.ashokit.bindings.Student;

public class XMLToJavaConverter {

	public static void main(String[] args) throws Exception {
		File xmlFile = new File("student.xml");

		JAXBContext jaxBContext = JAXBContext.newInstance(Student.class);
		Unmarshaller unmarshaller = jaxBContext.createUnmarshaller();
		Student student = (Student) unmarshaller.unmarshal(xmlFile);
		System.out.println(student);

	}
}
```

To work attributes in xml we will use below annotation in binding class

```
@XmlAttribute
private String standard;
```

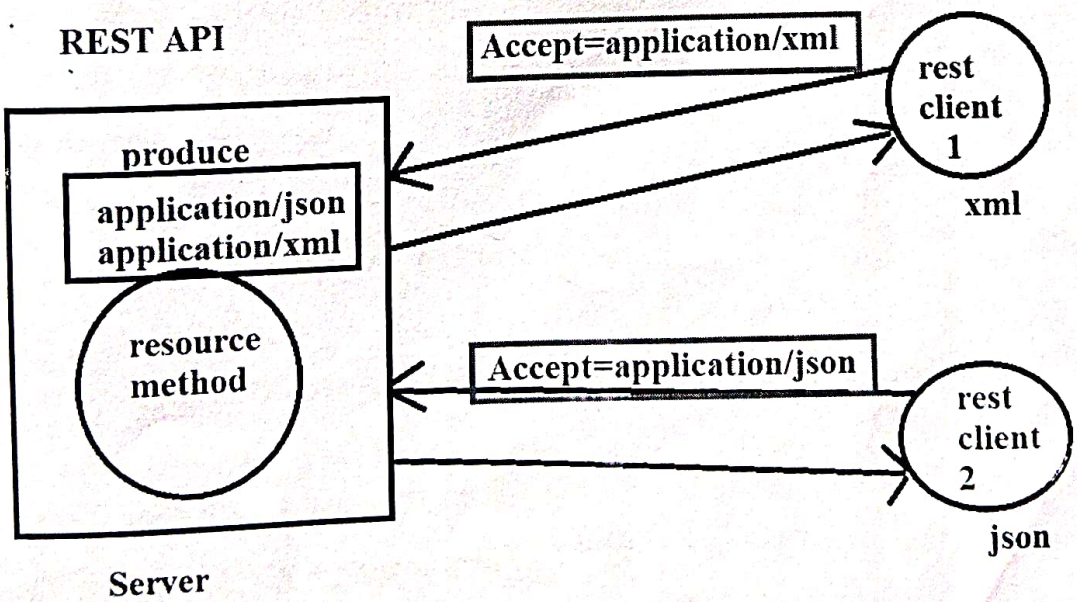
Note: If we don't use @XmlAttribute annotation for variable, then it will be considered as child element.

In Rest Controller, How to return response in Multiple Formats?

To represent response formats in Rest Controller method we will use produces concept.

```
@GetMapping(
        value="/student/{sid}',
        produces = {
           "application/json",
           "application/xml"
        }
)
public Student getStudentById(Integer sid){
        //logic
        return studentObj;
}
```

When method is supporting for multiple response formats like above then clients can choose response format using 'Accept' header.



Yesterday's session : produces & Accept

Today's session : POST Request

what is produces in Rest api ?

produces attribute represents REST API method supported response formats.

What is Accept ?

Rest client will send Accept header in request with desired response format

Note: Client sending Accept header value should match with produces attribute value in Rest api method.

-> When we are developing REST Api, if our rest controller method is responsible to send data to client then we will bind that method to GET Request method.

-> To bind the method to GET request, we will use @GetMapping annotation.

HTTP Post

If our rest api method wants to take huge data/sensitive data from client then we will bind our method to HTTP Post Request method.

To Bind Our method to HTTP Post Request then we will use

@PostMapping annotation.

Rest Resource

@GetMapping

m1() - This is responsible to send data to client

@PostMapping

m2() - This is responsible to capture data from client (to create a record at server)

HTTP Status Codes & Status Messages

Today's session : POST Request method

-> To bind Rest resource method to Http Post request we will use @PostMapping annotation

@PostMapping(value="/url")

```
public String addBook(){
        return str;
}
```

-> If the method is responsible to create a record at server by taking data from client then that method we will bind to POST request method.

-> For Post request method we can take method parameter and method return type

```
@PostMapping(value="/bookTicket")
public Ticket bookTicket(PassengerInfo pinfo){
   //logic
   return ticket;
}
```

Note: For Post request we will have request body.

-> consumes attribute will represent input formats supported by Rest Resource method

-> produces attribute will represent output formats supported by Rest Resource method

---

```
@PostMapping(
                value="/bookTicket",
                produces= {
                        "application/xml",
                        "application/json"
                },
                consumes= {
                        "application/json",
                        "application/xml"
                }
        )
        public Ticket bookTicket(PassengerInfo pinfo) {
                // logic
                Ticket t = new Ticket();
                t.setTicketId("TA02899");
```

```
        t.setTicketStatus("CONFIRMED");
        t.setJourneyDate(pinfo.getJoureyDate());
        t.setPassengerName(pinfo.getFirstName() + " " + pinfo.getLastName());
        t.setTrainNumber(pinfo.getTrainNumber());
        t.setTicketPrice(350.00);
        return t;
    }
```

```
<?xml version="1.0" encoding="UTF-8">
<passenger-info>
<firstName>John</firstName>
<lastName>Peter</lastName>
<from>Hyd</from>
<to>Chennai</to>
<journeyDate>25-July-2020</journeyDate>
<trainNumber>KH668</trainNumber>
</passenger-info>
```

-> Client will send Content-Type header to represent in which format client is sending data to server.

produces

consumes

Accept

Content-Type

Today's session : PUT & DELETE Request

-> To bind our rest api method to POST request we will use @PostMapping annotation

```
@PostMapping(
    value="/bookTicket",
    consumes={"application/xml", "application/json"},
    produces={"application/xml", "application/json"}
)
```

```
public Ticket bookTicket(Passenger pinfo){
    //logic
        return ticket;
}
```

-> consumes attribute represents input data formats supported by method

-> produces attribute represents output data formats supported by method

-> When client is sending request to above method, he should sent two HTTP Headers in request

a)Content-Type

b)Accept

-> Content-Type header represents input data format sending by client.

-> Accept Header represents output data format expecting by client.

-> POST Request Means Creating Record/Resource at server. If POST Request operation is successful it should represent with HTTP 201 status code.

-> By default, rest controller methods response sending to cient 200 as status code which means OK.

-> If we want to return response with different status code then we have to use ResponseEntity concept.

Who is construcing response at API side?

-> DispatcherServer

-> If there is no problem in request execution, then dispatcher servlet will construct response with 200 as status code and it will send to client.

GET Requet

POST Request

PUT Request

DELETE Request

When to bind our rest controller method to PUT request?

If our rest controller method is responsible for updating existing record data then we will bind that method to PUT request method

-> To bind the method to put request method, we will use @PutMapping annotation

```
@PutMapping(
        value="/updateTicket",
        consumes={"application/json", "application/xml" },
        produces={"application/json" , "application/xml" }
)
public ResponseEntity<Ticket> updateTicket(){
}
```

When we should bind our Rest Controller method to DELETE request?

If our rest controller method is responsible to delete a record at server then we will bind that method to HTTP DELETE method.

-> @DeleteMapping we can use to bind method to delete request.

Note : Only for GET, request body is not available. For POST, PUT and DELETE we have request body.

-> If we want to get small piece of information from client to server then we can go for query params or path params.

-> If we want to recieve huge amount of data from client then we can use request body which is available for POST, PUT and DELETE.

HTTP Methods

GET

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

HEAD

The HEAD method asks for a response identical to that of a GET request, but without the response body.

POST

The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.

PUT

The PUT method replaces all current representations of the target resource with the request payload.

DELETE

The DELETE method deletes the specified resource.

CONNECT

The CONNECT method establishes a tunnel to the server identified by the target resource.

OPTIONS

The OPTIONS method is used to describe the communication options for the target resource.

TRACE

The TRACE method performs a message loop-back test along the path to the target resource.

PATCH

The PATCH method is used to apply partial modifications to a resource.

Today's session : HTTP Protocol

-> In RESTFul services two actors will be available

1) Resource
2) Client

-> To establish communication between Client and Resource we will use HTTP Protocol

-> HyperText Transfer Protocol

-> For HTTP Protocol we have 2 version they are

1) HTTP 1.0 (status codes support is not available)

2) HTTP 1.1

-> In HTTP Protocol we have several methods like below

1)GET

2)POST

3)PUT

4)DELETE

5)HEAD

6)OPTIONS

7)CONNECT

8)TRACE

9)PATCH

-> GET method is used to retrieve information from the server using given URI. (It doesn't contain request body). The data will be appended to URL in GET request (path params & query params)

-> HEAD method is same as GET method it is used to transfer the status line and header section only. It will not return message body.

-> POST method is used to send information to server in request body (File upload, HTML forms) (It creates new record at server)

-> PUT method is used to replace all the current representation of record/resource (current data will be updated with new data)

-> DELETE method is used to remove current representation of the target record/resource from the server which is given by URI.

-> CONNECT method establishes connection with Server

-> OPTIONS method is used by client to findout http methods and other options that are supported by server.

-> PATCH method is used to update partial content of target resource at server.

-> TRACE method purpose is to return request back to client. It is used to debug the problem

What is idempotency in HTTP Protocol?

When we send multiple identical(same) requests to server if it is producing same result without any side effects then it is called as Idempotency.

-> GET, PUT, HEAD and TRACE methods are called as idempotent methods.

Today's session : HTTP Req & Http Response

-> HTTP stands for HyperText Transfer Protocol

-> This protocol acts as a mediator between client & server

-> This protocol acts as a mediator between distributed applications (client & resource applications)

-> Http Protocol divided into 2 parts

a) Http Request

b) Http Response

Http Request Again Divided into 5 parts

1) Initial Request Line

2) Host Address

3) Request Headers

4) Blank Line

5) Request Body

HTTP Response Divided into 4 Parts

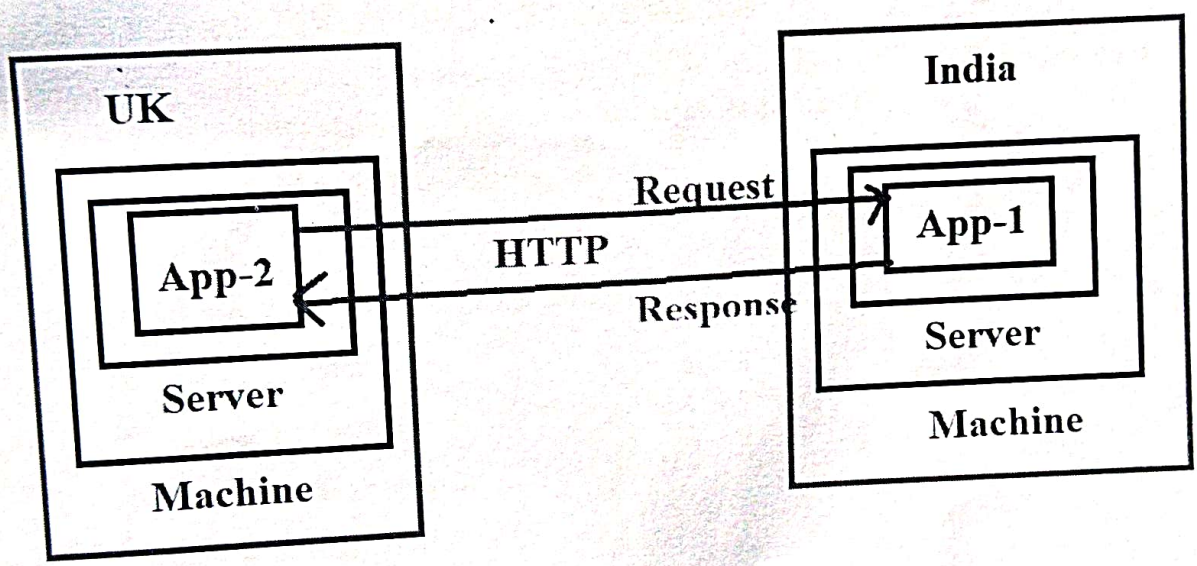
1) Initial Response Line

2) Response Headers

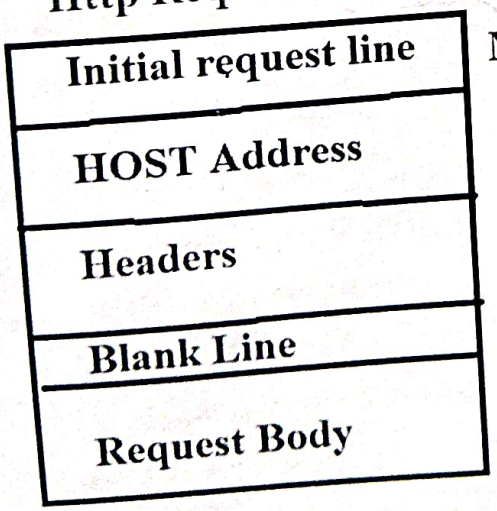
3) Blank Line

4) Response Body

Distributed Applications Architecture



Http Req Structure



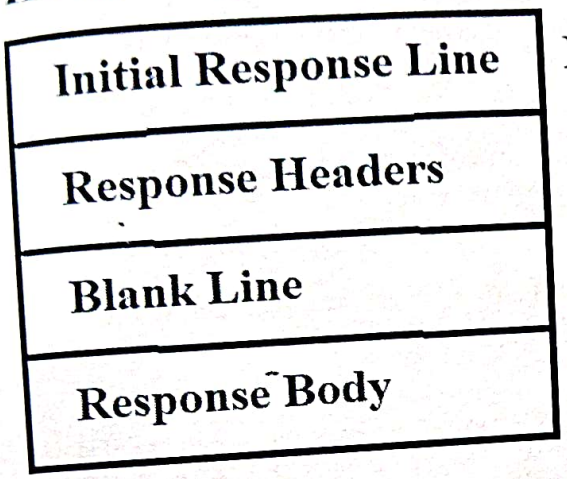
HTTP
METHOD URI Protocol-Version

SERVER IP

HTTP Headers in Key-Value
(Content-Type & Accept)

Payload (Business data)
text/json/xml etc...

HTTP Response Structure



HTTP Protocol_version HTTP Status CD HTTP MSG

Headers in Key-Value (Content-Length)

Payload (Business data) (text/xml/json etc..)

Today's session : Http Status Codes & Messages

-> When client sends http request to server, it will process that request and it will send response back to client with status code, status message, headers & response body

status-code status-msg protocol-version

-> HTTP status codes are divided into 5 categories

1xx (100-199) : Information

2xx (200-299) : Success

3xx (300-399) : Redirectional

4xx (400-499) : Client Error

5xx (500-599) : Server Error

-> 200 OK

-> 201 Created

-> 204 NO CONTENT

-> 400 BAD Request

-> 404 Not Found

-> 405 Method Not Allowed

-> 406 Not Acceptable

-> 415 Unsupported Media Type

-> 500 Internal Server Error

Today's session : HATEOS example

REST Architecture Principles

1) Unique Addressability

2) Uniform Contraint Interfaces

3) Message Oriented Representation

4) Communication Stateless

5) HATEOS (Hypermedia as the engine of application state)

-> HATOES is next level component in REST API.

-> Using this HATEOS Rest Resources will provide information dynamically through Hyper media

Scenario

Client sent request to REST API to get Book Information based on ID

Client Requested URL : www.amazon.in/book/101

Then Server sent Book Information as Response like below

```
{
   "id"  : 101,
   "name" : "Spring",
   "price" : "500$",
   "author" : "Rod Johnson"
}
```

USING HATEOS

Server can send response like below

```
{
   "id"  : 101,
   "name" : "Spring",
   "price" : "500$",
   "author" : "Rod Johnson",
    "all-books" {
          "href" : "www.amazon.in/books/rodjohnson"
        }
}
```

Develop a rest api which provides response using HATEOS

Note: To work with HATEOAS, spring boot provided starter pom

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

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Book extends RepresentationModel<Book> {

        private String isbn;
        private String name;
        private Double price;
        private String author;

  }
```

```
package com.ashokit.rest;
import java.util.ArrayList;
import java.util.List;
```

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.hateoas.Link;
import org.springframework.hateoas.RepresentationModel;
import org.springframework.hateoas.server.mvc.WebMvcLinkBuilder;

import com.ashokit.binidings.Book;

@RestController
public class BookRestController {

	@GetMapping(value = "/book/{isbn}", produces = "application/json")
	public Book getBookInfo(@PathVariable("isbn") String isbn) {
		// logic
		Book b = new Book(isbn, "Spring", 300.00, "Rod Johnson");

		Link withRel = WebMvcLinkBuilder.linkTo(
WebMvcLinkBuilder.methodOn(BookRestController.class)
.getAllBooks())
.withRel("All-Books");

		b.add(withRel);

		return b;
	}

	@GetMapping(value="/books", produces="application/json")
	public List<Book> getAllBooks() {
		Book b1 = new Book("ISBN001", "Spring", 300.00, "Rod Johnson");
```

```
        Book b2 = new Book("ISBN002", "Spring Boot", 400.00, "Rod Johnson");
        Book b3 = new Book("ISBN003", "Spring Cloud", 200.00, "Rod Johnson");

        List<Book> books = new ArrayList<Book>();
        books.add(b1);
        books.add(b2);
        books.add(b3);

        return books;
    }
}
```

Project Class : 7 AM - 9:30 AM IST

Tomorrow Class Timing : 10 AM - 12 PM IST

Topic : Swagger Documentation

Q) What is Swagger?

Swagger is an ecosystem which is used to generate documentation for REST apis.

Using Swagger-UI , we can test our REST Apis also

-> Swagger Configuration we will add in Rest Api project.

-> All our api details will be available in swagger. Using documentation provided by swagger, client side team can start their development.

Swagger documentation Contains below details

---

1) Resource method Types

2) Resource method URLs

2) Resource Method Consumes types

3) Resource Method Produces types

4) Resource Method Input Data Structure

5) Resource method Output Data Structure

1) What is Distributed application?
2) Distributed Technologies
3) Why Webservices
4) Webservices Evolution
5) SOAP Webservice Apis
6) SOAP Architecture
7) REST Introduction
8) REST Architecture Principles
9) HTTP Protocol
10) HTTP Methods
11) HTTP Status Codes
12) HTTP Request & HTTP Response Parts
13) What is Rest Controller?
14) Request URL Mappings
15) GET , POST, PUT and DELETE methods
16) Consumes
17) Produces
18) JSON and Jackson
19) XML and JAX-B

20) ResponseEntity

21) Swagger

What is Swagger?

Swagger is used to generate documentation for REST API

Swagger UI is used to test REST API functionality

Using this swagger documentation, rest client can be developed.

WSDL	Swagger
-> It contains soap provider details	-> Swagger is an API
-> It is xml file	-> Using swagger we can generate documentation for rest api
-> It is used in SOAP Webservices	-> Swagger will generate documentation in JSON format
	-> Swagger provided UI component to test REST API (alternate for POSTMAN)
	-> It will be used in RESTful services

- Introduction
- What is Swagger
- Why Swagger
- Swagger Annotations
- Configuring Swagger in Spring Boot Application

Introduction.. Contd..

- As a API developer we should provide some documentation for Clients regarding our API.
- There are multiple approaches to documenting your REST API
  - WADL
  - RESTDocs
  - Swagger or OpenDocs

What is Swagger ?

- Swagger is an open source tooling ecosystem which is used to generate Documentation for Rest APIs.
- The goal of Swagger is to define a standard interface to REST APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code.

Why Swagger ?

- Swagger is one of the most popular specifications for REST APIs for a number of reasons :-
- Swagger generates an interactive API console for people to quickly learn about and try the API.
- The Swagger file can be auto-generated from code annotations or using case class definitions on a lot of different platforms.
- Swagger has a strong community with helpful contributors.

Requirement :

Develop a REST API using Spring Boot and Provide Documentation for REST API using Swagger.

Steps to Develop REST API With Swagger
---
1) Create Spring Boot

mvn repository swagger=springfox swager2, Swager UI=springfox swager UI

steps to Develop REST API With Swagger

1) Create Spring Boot Application with below dependencies

1)spring-boot-starter-web
2)swagger
3)swagger-ui

2) Create Binding Classes For Request and Response

3) Create Rest Controller with required methods

---

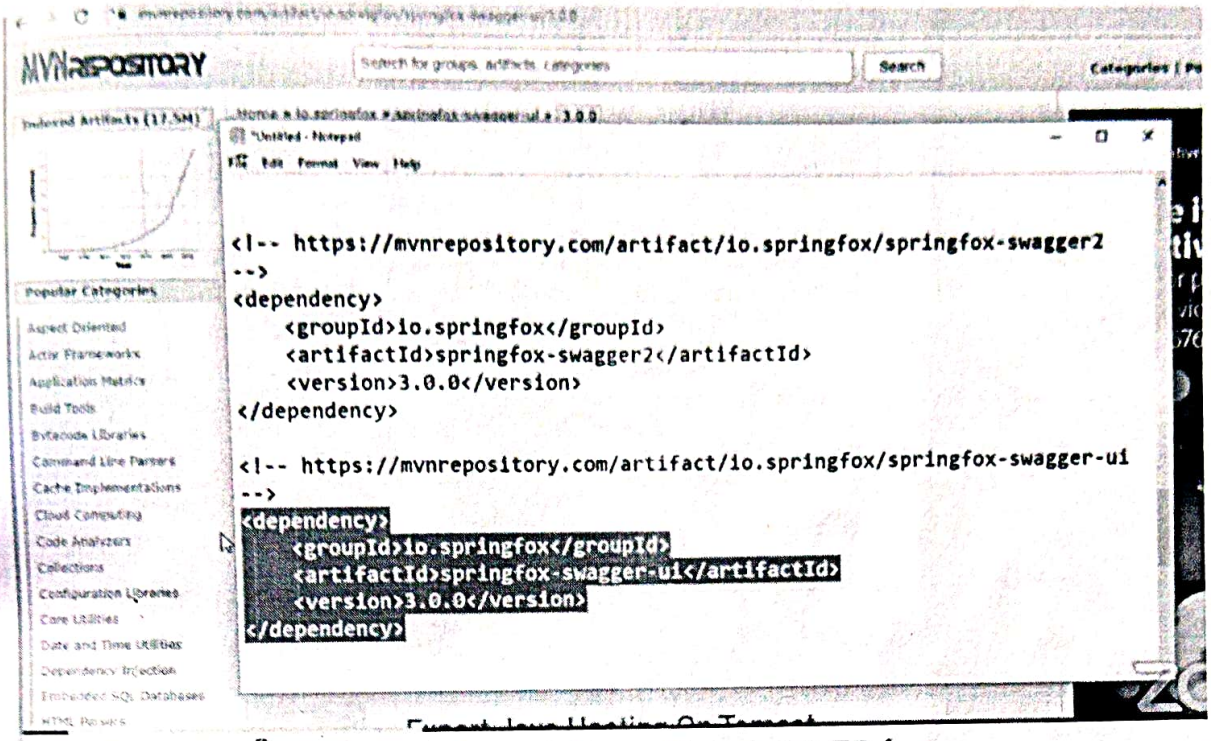
1) Create Spring Boot Application with below dependencies

1)spring-boot-starter-web
2)swagger
3)swagger-ui

2) Create Binding Classes For Request and Response

3) Create Rest Controller with required methods

4) Create SwaggerConfig class for generating documentation




```
Application URL : http://localhost:7071/

Swagger-UI URL : http://localhost:7071/swagger-ui.html

Swagger Documentation URL :
-------------------------------
http://localhost:7071/v2/api-docs

This URL will generate the documentation in JSON
format.
```

Docket object:

A builder which is intended to be the primary interface into the swagger-springmvc framework. Provides sensible defaults and convenience methods for configuration.

Docket object is represent For which component we want to generate the documentation.

Docket is a bean which is represent which contains the onformation about our API. For which API it will generate the documentation. we need to specify for which class or method it has to create the documentation.

Configuration class

```
@Configuration
public class SwaggerConfig {
```

```
@Bean
public Docket customerApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                                .select()

.apis(RequestHandlerSelectors.basePackage("com.ashokit.rest"))
                                .paths(PathSelectors.any())
                                .build();
    }
}
```

Controller class

```
@RestController
@Api("This is AirIndia Distributed Component")
public class AirIndiaRestController {


    @ApiResponses(value= {
                @ApiResponse(code=201, message="Resource Created"),
                @ApiResponse(code=500, message="Server Error")
    })
    @ApiOperation("This is used to book Flight Ticket")
    @PostMapping(
                        value = "/bookFlightTicket",
                        consumes = {
                                    "application/json",
                                    "application/xml"
                        },
                        produces = {
                                    "application/json",
                                    "application/xml"
                        }
     )
```

```
public ResponseEntity<TicketInfo> bookTicket(@RequestBody PassengerInfo pinfo) {
        // logic to book ticket
        // send ticket info
        TicketInfo ticket = new TicketInfo();
        ticket.setName(pinfo.getFname() + " " + pinfo.getLname());
        ticket.setFrom(pinfo.getFrom());
        ticket.setTo(pinfo.getTo());
        ticket.setTicketStatus("CONFIRMED");
        ticket.setTicketPrice("4500.00 INR");
        ticket.setJourneyDate(pinfo.getJourneyDate());
        return new ResponseEntity<>(ticket, HttpStatus.CREATED);
    }

    @ApiOperation("This is used to get PNR status")
    @GetMapping(value="/pnrstatus/{pnr}")
    public String getPNRStatus(@PathVariable("pnr") String pnrNo) {
        return "CONFIRMED";
    }
}
```

Q-1) Swagger @ApiParam vs @ApiModelProperty

Yesterday's session : Swagger

Today's session : REST Clients Introduction

REST Clients

-> Any application/Tool/Human which is cable of Sending HTTP Request can act as REST CLient.

-> Python project can act as REST Client

-> .Net Project can act as REST client

-> Java Project Can act as REST Client

-> Angular & React Projects also can act as REST Clients

-> POSTMAN & Swagger-UI tools are acting as REST Clients.

How to Develop REST Client using JAVA

There are several ways available to develop REST Clients in Java applications

1) java.net client

2) Apache HTTP Utils

3) JAX-RS api with JERSEY client

4) JAX-RS api with REST Easy Client

5) Spring RestTemplate (it is going to be deprecated)

6) Spring WebClient (Available from Spring5.x, and famous now)

Yesterday's session : REST Client Introduction

Today's session : java.net client examples

-> Once Rest Resource Development is completed, Resource team should provide documentation

-> That documentation should contains details of Rest API

(Request Type, Request URL, Input format, Output format)

-> They can provide these details by using Swagger.

-> Using Swagger Documentation We can start Rest Client Development

REST Endpoint URL : http://192.168.3.1:8083/

Note: Swagger is enabled

Swagger UI URL : http://192.168.3.1:8083/swagger-ui.html

Api Docs Json URL :

http://192.168.3.1:8083/v2/api-docs

-> Any application which can send http to rest api can be called as Rest Client.

-> Using Java, we can develop Rest Client in several ways

1)java.net client (jdk)

2)Apache Http Client

3)Jax-rs api jersey client

4)Jax-rs api RestEasy client

5)Spring RestTemplate

6)Spring WebClient (Spring 5.x)

Developing Rest Client Using java.net client

Rest Endpoint : http://localhost:8080/

Resource URL : /welcome

Resource Method : HTTP GET

```
package com.ashokit.rest.client;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class WelcomeClient {

	public static void main(String[] args) {
		try {
			URL url = new URL("http://localhost:8083/welcome");
			HttpURLConnection conn = (HttpURLConnection) url.openConnection();
			conn.setRequestMethod("GET");

			int httpStatusCd = conn.getResponseCode();
			if(httpStatusCd==200) {
				InputStream is = conn.getInputStream();
				InputStreamReader isr = new InputStreamReader(is);
				BufferedReader br = new BufferedReader(isr);
				String line = br.readLine();
				while(line!=null) {
					System.out.println(line);
					line = br.readLine();
```

```
                    }
                    conn.disconnect();
                }
            }catch(Exception e) {
                e.printStackTrace();
            }
        }

}
```

Yesterday's session : java.net client

Today's session : java.net client for POST request

What is Rest client?

Any application which is capable of sending Http GET request is called as Rest Client.

How many ways we can develop Rest Client ?

There are several ways available to develop Rest clients using java

1)java.net client

2) apache http client

3) jersey client

4) rest easy client

5) RestTemplate

6) WebClient

----------Client sending HTTP GET request----------

```
URL url = new URL(restEndpointUrl);

HttpUrlConnection con = (HUC) url.openConnection();

con.setRequestMethod("GET");

int scd = con.getResponseStatusCode();

if(scd==200){
        //read response from inputstream
}
```

---

localhost:8084/swagger-ui.html

Client sending HTTP Post Request

---

https://documenter.getpostman.com/view/8854915/SzS8rjHv?version=latest

RESTful Services & Microservices

Duration : 40 Days

Class Timings : 7:30 AM - 9:00 AM IST

Pre-Requisites : Basics of Spring Boot & Spring MVC

RESTful servies (Spring with REST)

---

1) Distributed Application

2) What are the technologies available to develop Distributed applications

3) Distributed Application Architecture

4) Realtime use cases for DS application

5) XML, XSD & JAX-B

6) JSON, Jackson and Gson

7) Spring with REST Introduction

8) HTTP Protocal Methods

9) HTTP Protocal Status Codes & Status Messages

10) REST Controller

11) Request Mapping Annotations

- -@GetMapping
- -@PostMapping
- -@PutMapping
- -@DeleteMapping

12)REST API Development

13) Query Parameters & Path Parameters

- -@RequestParam
- -@PathVariable

14)@RequestBody

15)@ResponseBody

16)Working with Complex Response

17)ResponseEntity

18)POSTMAN (To test REST API)

19)Swagger & Swagger UI

20)Exception Handling In REST API

21) REST Client Development

- - RestTemplate
- - WebClient (Spring v5.0)

22) Mono Object & Flux Object

23) Async Rest Calls

24) Security (OAuth 2.0 & JWT)

25) SOAP Provider using Boot

26) SOAP Consumer using Boot

Microservices

---

1) What is Monolith Architecture
2) Drawbacks of Monolith Architecture
3) What is Load Balancer
4) Load Balancing Algorithms
    - Sticky Session
    - IP Hashing
    - Round Robbin
5) Microservices Introduction
6) Challenges of working with Microservices
7) Advantages of Microservices
8) Microservices Architecture
9) Service Registry (Eureka)
10) Gateway (Zuul Proxy)
11) 2 Microserviecs Development with H2 DB
12) Interservice Communication (Feign Client)
13) Circuit Breaker with Hystrix
14) Distributed Logging
    - Sleuth
    - Zipkin
15) Actuators
16) Boot Admin Server & Admin Client
17) Cloud Platforms Introduction
    - IaaS
    - SaaS
    - PasS
18) PCF Account Creation
19) PCF CLI
20) PCF Commands

21) Deploying Microservices to Cloud (PCF)

22) Deploying Microservices to AWS

23) What is Scaling ?

- Verticle Scaling
- Horizontal Scaling
- AutoScaling

24) Apache Kafka

25) MongoDB

26) ConfigServer

27) RedisCache

28) LDAP

29) Mini Project with Angular Frontend

Yesterday's session : java.net client with GET Request

Today's session : java.net client with POST request

-> We are trying to develop a REST client to send HTTP Post request

-> To develop client we are using java.net client package

-> We have created client side binding classes

```
package com.ashokit.client;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.net.HttpURLConnection;
```

```
import java.net.URL;

import com.ashokit.request.PassengerInfo;
import com.fasterxml.jackson.databind.ObjectMapper;

public class ERailClient {

	public static void main(String[] args) throws Exception {
		URL url = new URL("http://localhost:8084/bookTicket");
		HttpURLConnection con = (HttpURLConnection) url.openConnection();

		con.setRequestMethod("POST");

		// Setting HTTP Headers
		con.setRequestProperty("Content-Type", "application/json");
		con.setRequestProperty("Accept", "application/xml");

		// set data to request body
		PassengerInfo p = new PassengerInfo();
		p.setFirstName("John");
		p.setLastName("Smith");
		p.setFrom("HYD");
		p.setJourneyDate("15-Aug-2020");

		ObjectMapper mapper = new ObjectMapper();
		String jsonStr = mapper.writeValueAsString(p);

		con.setDoOutput(true);
		OutputStream outputStream = con.getOutputStream();
		outputStream.write(jsonStr.getBytes());
		outputStream.flush();
```

```
        //Get Response status code
        int responseCode = con.getResponseCode();
        if(responseCode==201) {
                InputStream inputStream = con.getInputStream();
                InputStreamReader isr = new InputStreamReader(inputStream);
                BufferedReader br = new BufferedReader(isr);
                String line = br.readLine();
                while(line!=null) {
                        System.out.println(line);
                        line = br.readLine();
                }

                con.disconnect();
        }

    }

}
```

Yesterday's session : java.net client

Today's session : RestTemplate

-> RestTemplate is a predefined class provided by Spring as part of Spring Web MVC Module

-> This class is available from Spring 3.x version

-> This class is used to send Http request

-> We can use this RestTemplate to develop Rest Clients in our applications

-> This class internally uses java.net package components to send http request.

-> It supports Synchronus communication

Note: Spring provided WebClient as an alternate for RestTemplate. (In future RestTemplate may be deprecated)

What is Synchronus Communication?

After sending our request, if we our thread is blocked until we response then it is called Sync communication.

Note: RestTemplate supports only Sync communication.

```
package com.ashokit;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class Application {

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

		RestTemplate restTemplate = new RestTemplate();

		String url = "http://localhost:8083/welcome";
```

```
        ResponseEntity<String> responseEntity =
                                restTemplate.getForEntity(url, String.class);

        int statusCd = responseEntity.getStatusCode().value();

        if(statusCd==200) {
                System.out.println(responseEntity.getBody());
        }

    }

}
```
