File Edit View

RESTFul Services
++++++++++++++

-> REST stands for 'Representational State Transfer'

-> RESTFul services are used to develop Distributed Applications with Intereoperabi

-> Distributed applications are used for B2B communications

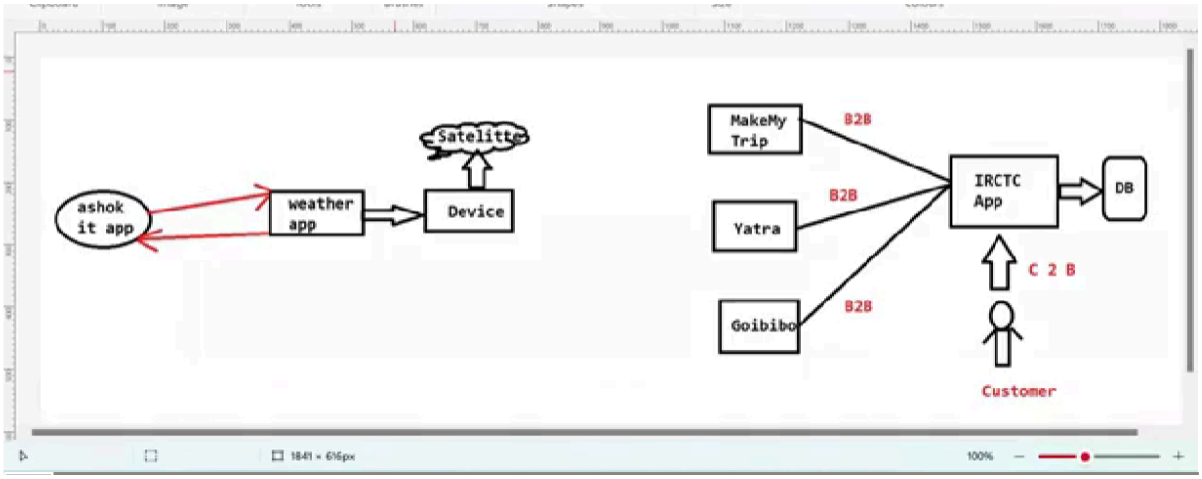
-> B2B means Business to Business Communication

-> If one application is communicating with another application then those are called as 'Distributed Applications'

-> Distributed Application will re-use services of one application in another application

-> RESTful Services are providing 'Intereoperability'

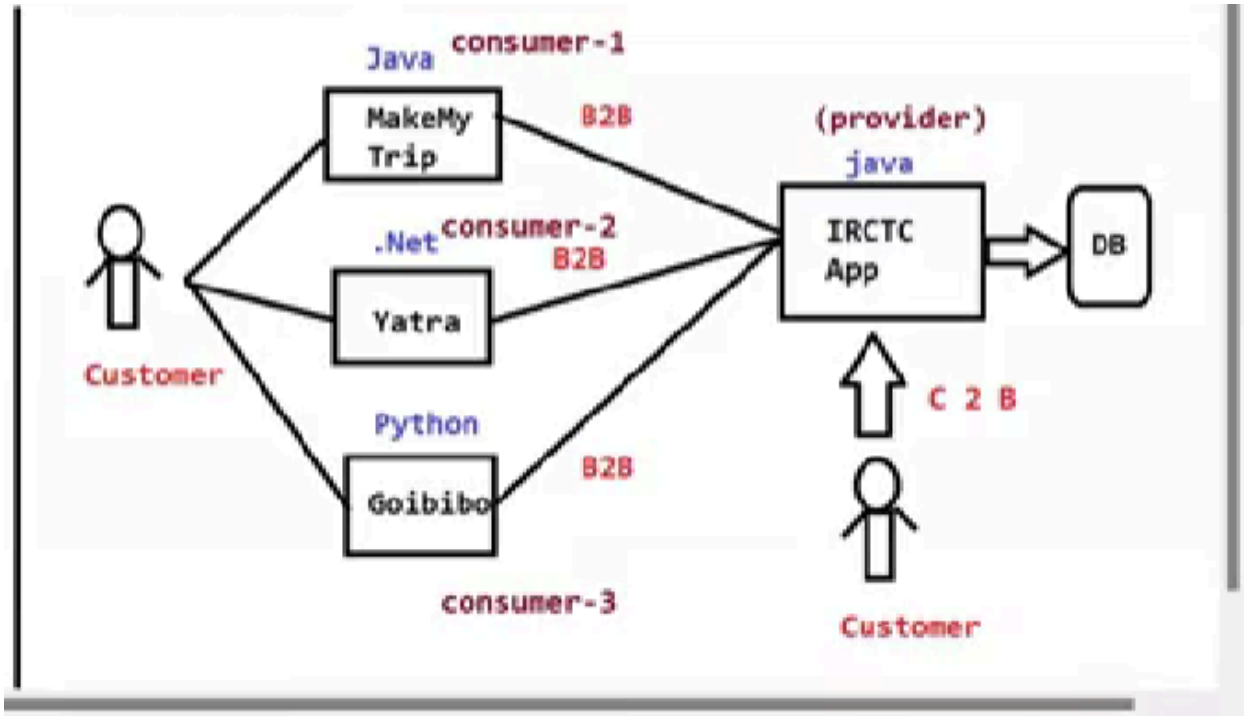
-> Intereoperability means irrespective of the platform and language applications can communicate



java app <----------> .Net app

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

Python app <--------> Java App



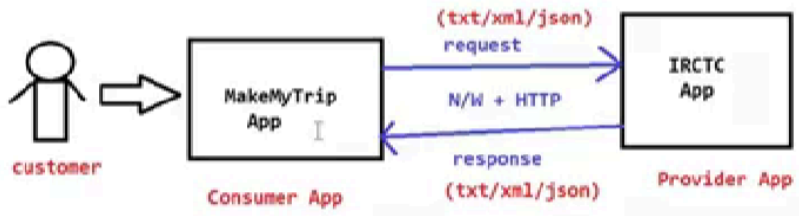
-> Two Actors will be involved in Distributed Applications development

1) Provider
2) Consumer

-> The application which is providing business services is called as Provider Application

-> The application which is accessing business services is called as Consumer Application

Note: One provider application can have multiple consumer applications



Note: One provider application can have multiple consumer applications

-> Provider and Consumer will communicate using HTTP as a mediator

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

Note: In Industry we will use JSON format to exchange data from one application to another application

-> To start our journey with RESTFul services development we should be good in below areas

1) HTTP Protocol (Methods, Status Codes, Req structure & Res Structure)

2) XML and JAX-B Api

3) JSON and Jackson Api

File Edit View

```
################
What is HTTP ?
################

-> HTTP stands for Hyper Text Transfer Protocol

-> It acts as a mediator between client & server   ( Consumer & Provider )

-> Consumer application will send HTTP Req to Provider application

-> Provider application will process the request and it will send HTTP response to Consumer


###############
HTTP Methods
###############

-> HTTP methods are used to send request from Consumer application to Provider application

GET
POST
PUT
DELET
```

-> GET request is used to retrieve data from Server / Provider application.
-> GET request will not have body to send data in the request
-> To send any data using GET request then we will use Path Params & Query Params

Ex: https://www.youtube.com/watch?v=VO818de8sdk

Note: Path Params & Query Params data will be displayed in the URL

Note: It is not recommended to send sensitive / secret data using Path Params & Query Params

-> GET request is Idempotent (means if you send same GET request for multiple times also nothing will change at server)

- > POST request is used to create a new record at server
-> When consumer wants to send huge data/ sensitive data then Consumer will use POST request
-> POST request contains request body
-> POST request is Non-Idempotent

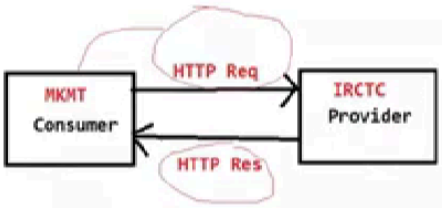
Note: In POST request we can send in URL and in Request Body.

Note: Request Body is the recommended approach to send sensitive data to server

-> PUT request is used to update a record at server
-> When consumer wants to update a record at then consumer will send PUT request to Provider
-> POST request contains request body
-> POST request is Idempotent

Note: In PUT request we can send in URL and in Request Body.

Note: Request Body is the recommended approach to send sensitive data to server



HTTP Request Format

Initial Request Line	HTTP METHOD + URL
Header	Meta data (key-value)
Blank Line	empty line
Request Body	contains payload (business data)

HTTP Response Structure

Initial response line	HTTP Method + Status Code + Status MSG
Response headers	meta data (key-value)
Blank line	Empty Line
Response Body	(Payload) (business data)

##################
HTTP Status Codes
##################

-> HTTP Status codes will represent the status of our request & response

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

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

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

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

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

-> XML is intereoperable (Language independent & Platform indenpendent)

-> XML we can use to transfer data from one application to another application

-> XML introduced by w3c org

-> The initial version of xml is 1.0 and the current version of xml is also 1.0

-> XML will represent data in the form of elements

-> An element is the combination of start tag and end tag

Ex: <name> Ashok IT </name>

-> We will have 2 types of elements in the XML

1) Simple Element

2) Compound Element

-> The element which contains data directley is called as Simple Element

```
<name> Ashok IT </name>

<type> Educational </type>
```

-> The element which contains child element is called as Compound Element

```
<person>
    <id> 101 </id>
    <name> raju </name>
</person>
```

Note: here <person> is compound element

Note: XML should have only one root element. Inside the root element we can have multiple child elements

```
<persons>
    <person>
        <id> 101 </id>
        <name> raju </name>
    </person>

    <person>
        <id> 101 </id>
        <name> raju </name>
    </person>
</persons>
```

#############

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

-> JAX-B is used to convert Java object to xml and xml to java object

-> JAX-B is free and open source

-> JAX-B given by sun microsystem

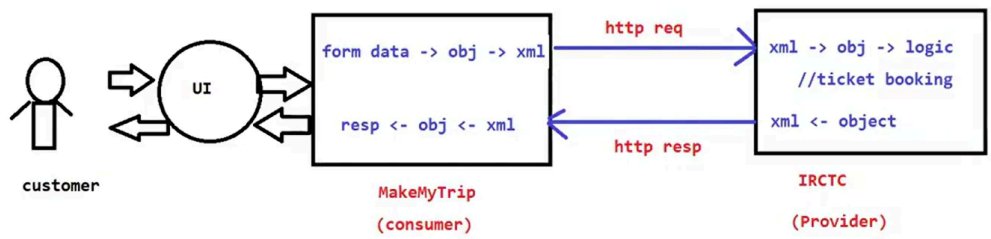
-> JAX-B is part of JDK upto 1.8v

-> If you are using JDK 1.8+ version of java then you need to add JAX-B dependency in pom.xml file

-> The process of converting Java Object into xml is called as "Marshalling"

-> The process of converting XML data to Java Object is called as "Un-Marshalling"

```java
import in.ashokit.bindings.Person;

public class ConverJavaToXml {

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

        Person person = new Person();
        person.setId(101);
        person.setName("John");
        person.setAge(25);
        person.setPhno(125757571);

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

        Marshaller marshaller = instance.createMarshaller();

        marshaller.marshal(person, new File("Person.xml"));

        System.out.println("Marshalling Completed....");

    }
```




```
########
JSON
########

-> JSON stands for Java Script object notation

-> JSON will represent data in key-value format

Ex :

{
   "id" : 101,
   "name: "raju",
   "age" : 20
}

-> JSON is intereoperable (language in-dependent & platform independent)

-> JSON is light wei
```

XML data

```
<person>
   <id> 10 </id>
   <name> Raju </name>
   <age> 20  </age>
</person>
```

JSON data

```
{
  "id" : 10,
  "name" : "Raju",
  "age" : 20
}
```

-> In xml meta-data (tags) is more than actual data
-> JSON represents data in the form key-value

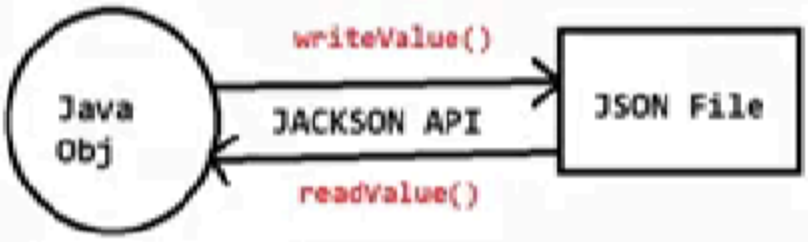
-> XML requires more memory bcz of its metadata
-> JSON is light weight

30-JSON-JACKSON-App/pom.xml Book.java JavaToJsonConverter.java

```java
package in.ashokit.binding;

import java.io.File;

import com.fasterxml.jackson.databind.ObjectMapper;

public class JavaToJsonConverter {

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

        Book book = new Book();
        book.setId(101);
        book.setName("Spring");
        book.setPrice(450.00);

        ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(new File("book.json"), book);
        System.out.println("Conversion Completed....");

    }
}
```

Background fill

JSON with JACKSON



-> Jackson API provided ObjectMapper class to perform conversions

-> writeValue() method converts java to json
-> readValue() method converts json to java

File Edit View

-> JSON will take less memory
-> JSON is light weight

-> To work with JSON data in Java Applications we have below 3rd party APIs

1) JACKSON API

2) GSON API

-> By using above apis we can convert JSON data to Java Object and vice versa

1) Create Maven project with below dependencies

```
<dependencies>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.13.3</version>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
```

2) Create Java classes to represent data (Use lombok)

```
@Data
public class Author {

    private String authorName;
    private String authorEmail;
    private Long authorPhno;

}

@Data
public class Book {

    private Integer id;
    private String name;
    private Double price;
    private Author author;
```

Ln 50, Col 32

3) Create Java class to convert Java Obj to JSON file

```
public class JavaToJsonConverter {

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

        Author author = new Author();
        author.setAuthorName("Rod Johnson");
        author.setAuthorEmail("r.john@gmail.com");
        author.setAuthorPhno(868686861);

        Book book = new Book();
        book.setId(101);
        book.setName("Spring");
        book.setPrice(450.00);
        book.setAuthor(author);

        ObjectMapper mapper = new ObjectMapper();

        // converting java obj to json and store into a file
        mapper.writeValue(new File("book.json"), book);

        System.out.println("Conversion Completed....");
```

4) Create Java Class To Convert JSON to Java Object

```
public class JsonToJavaConverter {

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

        File jsonFile = new File("book.json");

        ObjectMapper mapper = new ObjectMapper();

        Book book = mapper.readValue(jsonFile, Book.class);

        System.out.println(book);

    }
}
```

File Edit View

++++++++++++++++++++++++
Working with GSON API
++++++++++++++++++++++++

1) Create a maven project with below dependency

```
<dependency>
   <groupId>com.google.code.gson</groupId>
   <artifactId>gson</artifactId>
   <version>2.9.0</version>
</dependency>
```

-> GSON api provided by google

-> In GSON api we have 'Gson' class to perform conversions

toJson ( ) -> to convert java object to JSON

fromJson ( ) -> to convert json data to java object

++++++++++++++++++++++++
RESTFul Services Architecture
++++++++++++++++++++++++

-> We will have 2 actors in RESTful services

1) Provider / Resource

2) Consumer / Client

-> The application which is providing services to other applications is called as Provider or Resource application

-> The application which is accessing services from other applications is called as Consumer or Client application

++++++++++++++++++++++++++++++++
How to develop REST API using Java
++++++++++++++++++++++++

-> To develop RESFul Services/ REST APIs using java SUN Microsystem released 'JAX-RS' API

-> JAX-RS api having 2 implementations

1) Jersey (Sun Microsystems)
2) REST Easy (JBOSS)

Note: We can develop RESTFul Services using any one of the above implementation

-> Spring framework also provided support to develop RESTFul Services using 'Spring Web MVC' module.

-> Client application and Resource application will exchange data in intereoperable format (like XML & JSON)

request
client app <----------> resource app
response

Note: RESTful Services are used to develop B2B communications (No presentation logic, No view Resolver)

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class WelcomeRestController {

    @GetMapping("/welcome")
    public ResponseEntity<String> getWelcomeMsg() {
        String respPayload = "Welcome to Ashok IT";
        return new ResponseEntity<>(respPayload, HttpStatus.OK);
    }

    @GetMapping("/greet")
    public String getGreetMsg() {
        String respPayload = "Good Morning..!!";
        return respPayload;
    }
}
```

Note: Download postman tool to test our REST API functionality

++++++++++++++++++++++++++++++++++++

```
@RestController
public class WelcomeRestController {

	@GetMapping("/welcome")
	public ResponseEntity<String> getWelcomeMsg() {
		String respPayload = "Welcome to Ashok IT";
		return new ResponseEntity<>(respPayload, HttpStatus.OK);
	}

	@GetMapping("/greet")
	public String getGreetMsg() {
		String respPayload = "Good Morning..!!";
		return respPayload;
	}
}
```

Above are my first rest api

Note: GET Request will not contain Body to send data

-> We can use Query Params and Path Params to send data in GET Request

-> Query Params & Path Params will represent data in URL directlry

++++++++++++++
Query Params
++++++++++++++

-> Query Params are used to send data to server in URL

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

-> Query Params will start with '?'

-> Query Parameters will be seperated by '&'

-> Query Parameters should present only at the e

-> Query Parameters should present only at the end of the URL

Ex: www.ashokit.in/courses?name=SBMS&trainer=Ashok

-> To read Query Parameters from the URL we will use @RequestParam annotation

WelcomeRestcontroller.java | CourseRestController.java

```java
package in.jyoti.rest;

import org.springframework.http.HttpStatus;

@RestController
public class WelcomeRestcontroller {
    @GetMapping("/welcome")
    public ResponseEntity<String> getWelcomeMsg(@RequestParam("name") String name){
        String respPayload= name +"welcome to my 1st rest api";
        return new ResponseEntity<>(respPayload,HttpStatus.OK);
    }
    //2nd approch to send msg
    @GetMapping("/greet")
    public String getGreetmsg() {
        String respPayload="good morning";
        return respPayload;
    }

}
```

```java
package in.jyoti.rest;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class CourseRestController {
    @GetMapping("/course")
    public ResponseEntity<String> getcourse(@RequestParam("name") String name, @RequestParam("course") String c
        String payload= name +" ,welcome to ," + course;
        return new ResponseEntity<>(payload,HttpStatus.OK);
    }

}
```

localhost:8080/course?name=raju&course=ashokit

Save | Share

GET | localhost:8080/course?name=raju&course=ashokit | Send

Params | Auth | Headers (7) | Body | Scripts | Tests | Settings | Cookies

Query Params

Key	Value	Description
name	raju	
course	ashokit	
Key	Value	Description

Bulk Edit

Body | 200 OK | 11 ms | 189 B

Raw | Preview | Visualize

```
raju ,welcome to ,ashokit
```

++++++++++++++++++++++++++++++
Path Parameter or URI variables
++++++++++++++++++++++++++++++

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

-> Path Params will represent data directley in URL (no keys)

-> Path Params can present anywhere in the URL

-> Path Params will be seperated by / (slash)

-> Path Params should be represented in Method URL pattern (Template Pattern)

Ex: www.ashokit.in/courses/{cname}/trainer/{tname}

```
2
3 ort org.springframework.http.HttpStatus;
4 ort org.springframework.http.ResponseEntity;
5 ort org.springframework.web.bind.annotation.GetMapping;
6 ort org.springframework.web.bind.annotation.PathVariable;
7 ort org.springframework.web.bind.annotation.RestController;
8
9 stController
10 lic class bookRestcontroller {
11  // get data into server through pathvariable
12  @GetMapping("/book/{bname}")
13  public ResponseEntity<String> getoneparameter(@PathVariable("bname") String bname){
14
15      String payload= "book name is ," +bname;
16      return new ResponseEntity<>(payload,HttpStatus.OK);
17
18  }
19
20  @GetMapping("/book/{bname}/{aname}")
21  public ResponseEntity<String> getTwoparameter(@PathVariable("bname") String bname, @PathVariable("aname") Stri
22
23      String payload= "book name is ," +bname + "2nd book name is ," + aname;
24      return new ResponseEntity<>(payload,HttpStatus.OK);
25
26  }
27
28
```

localhost:8080/book/math/science

Save Share

GET localhost:8080/book/math/science Send

Params Auth Headers (7) Body Scripts Tests Settings Cookies

Query Params

Key	Value	Description	Bulk Edit
Key	Value	Description	

Body 200 OK 21 ms 207 B

Raw Preview Visualize

```
book name is ,math2nd book name is ,science
```

```
#################
What is Produces
##################

-> "produces" is a media type
-> It represens the response formats supported by REST Controller Method
-> One Rest Controller method can support multiple response formats (xml and json)

-> Client should send a request with "Accept" http header
-> Accept header represents in which format client expecting response
-> Based on Accept header Message Converter will convert the response and it will send to client as Http Response obj
```

Product.java | ProductRestController.java

```java
6 import org.springframework.web.bind.annotation.RestController;
7
8 import in.ashokit.binding.Product;
9
10 @RestController
11 public class ProductRestController {
12
13     @GetMapping(
14             value = "/product",
15             produces = { "application/xml", "application/json" }
16     )
17     public ResponseEntity<Product> getProduct() {
18
19         Product productObj = new Product();
20         productObj.setPid(101);
21         productObj.setPname("Mouse");
22         productObj.setPrice(450.00);
23
24         return new ResponseEntity<>(productObj, HttpStatus.OK);
25     }
26 }
27
```

GET localhost:8080/product Params

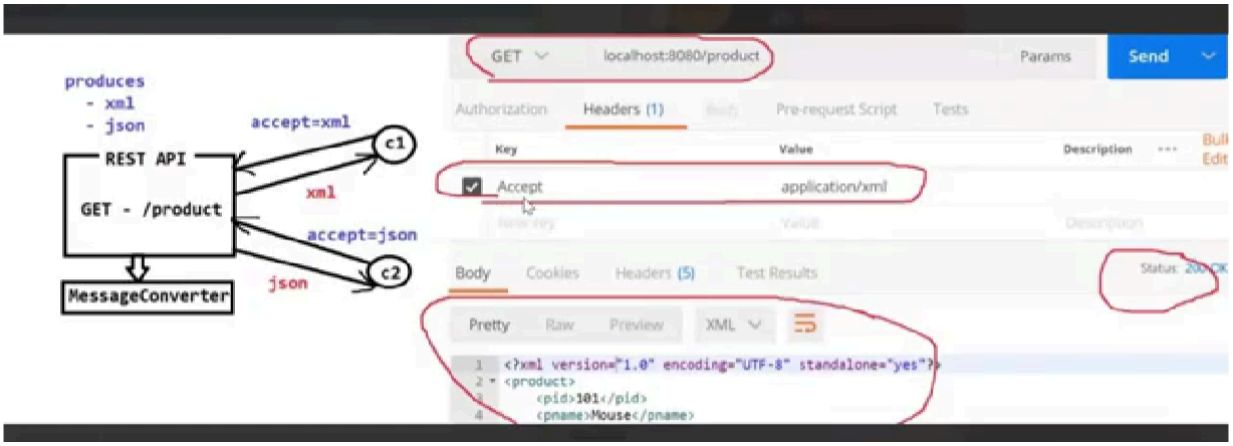
Authorization | Headers (1) | Body | Pre-request Script | Tests

Key	Value	Descripti
Accept	application/xml	
New key	Value	Descrip

Body | Cookies | Headers (5) | Test Results

Pretty | Raw | Preview | XML

```xml
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <product>
3     <pid>101</pid>
4     <pname>Mouse</pname>
5     <price>450.0</price>
6 </product>
```




```java
package in.ashokit.binding;

import javax.xml.bind.annotation.XmlRootElement;

import lombok.Data;

@Data
@XmlRootElement
public class Product {

    private Integer pid;
    private String pname;
    private Double price;

}
```

For single and how to deal with multiple object also

```
                produces = { "application/xml", "application/json" }
        )
        public ResponseEntity<Product> getProduct() {

            Product p1 = new Product(101, "Monitor", 8000.00);

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

        @GetMapping("/products")
        public ResponseEntity<List<Product>> getProducts(){

            Product p1 = new Product(101, "Monitor", 8000.00);
            Product p2 = new Product(102, "RAM", 6000.00);
            Product p3 = new Product(103, "CPU", 15000.00);

            List<Product> products = Arrays.asList(p1,p2,p3);

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

##############################
Working with HTTP POST Request
##############################

-> HTTP POST request is used to create new resource/record at server

-> POST request contains request body

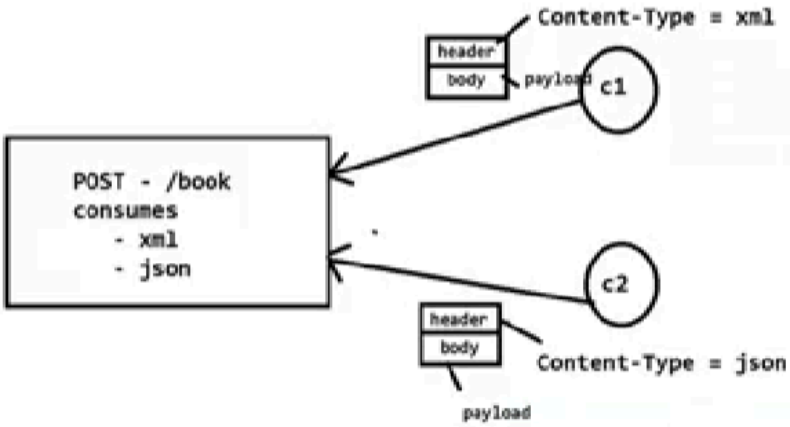
-> Client can send data to server in Request Body

-> To bind Rest Controller method to POST request we willl use @PostMapping

-> To read data from Requet body we will use @RequestBody annotation

-> "consumes" represents in which formats method can take input

-> "Content-Type" header represents in which format client sending data in request body.



```java
package in.jyoti.binding;

import lombok.Data;

@Data
public class book {
    Integer id;
    String name;
    Double price;


}

```

```java
package in.jyoti.rest;

import in.jyoti.binding.book;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;


@RestController
public class bookcontroller {

    @PostMapping(
            value="/book",
            consumes= {"application/json" , "application/xml"}
            )
    public ResponseEntity<String> addbook(@RequestBody book b){
        // msg for console
        System.out.println(b);
        // now msg for server
        String msg="book added succesfully ";
        return new ResponseEntity<>(msg, HttpStatus.CREATED);
    }

}

```

POST localhost:8080/book

Params Authorization Headers (9) Body Scripts Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
    "id":101,
    "name":"jyoti",
    "price":103.4
}
```

Body Cookies Headers (5) Test Results

Raw Preview Visualize

```
book added succesfully
```

-> produces vs consumes

-> Content-Type vs Accept

-> produces attribute represents in which formats Method can provide response data to clients

-> consumes attribute represents in which formats Method can take request data from clients

-> Accept header represents in which format client expecting response from REST API

-> Content-Type header represents in which format client is sending request data to REST API

++++++

```java
package in.jyoti.binding;

import lombok.Data;

@Data
public class Responsedata {
    private String bookingId;
    private String status;
    private String passengerName;
    private String trainNumber;
    private String journeyDate;
    private String source;
    private String destination;
    private String classType;

}

```

```java
package in.jyoti.rest;

import java.util.UUID;

@RestController
public class restController {
    @PostMapping("/ticket")
    ResponseEntity<Responsedata> BookTicket(@RequestBody Requestdata request){

    Responsedata response = new Responsedata();
    response.setBookingId(UUID.randomUUID().toString());
    response.setStatus("CONFIRMED");

    // ☑ Use getters from Lombok
    response.setPassengerName(request.getPassengerName());
    response.setTrainNumber(request.getTrainNumber());
    response.setJourneyDate(request.getJourneyDate());
    response.setSource(request.getSource());
    response.setDestination(request.getDestination());
    response.setClassType(request.getClassType());

    return new ResponseEntity<>(response, HttpStatus.CREATED);
    }
}

```

HTTP PUT Request
##################

-> PUT request is used to update an existing record / resource at server

-> PUT Request can take data in URL and in Request Body

-> To bind our method to PUT request we will use @PutMapping

#####################
HTTP DELETE Request
#####################

-> DELETE request is used to delete an existing record / resource at server

-> DELETE Request can take data in URL and in Request Body

-> To bind our method to DELETE request we will use @DeleteMapping

```
@PutMapping("/ticket")
public ResponseEntity<String> updateTicket(@RequestBody PassengerInfo request){
    System.out.println(request);
    //logic to update ticket
    return new ResponseEntity<>("Ticket Updated", HttpStatus.OK);
}

public ResponseEntity<String> deleteTicket(@PathVariable Integer ticketId){
    //logic to delete the ticket
    return new ResponseEntity<>("Ticket Deleted", HttpStatus.OK);
}
```

```
@GetMapping("/ticket/{ticketId}")
public ResponseEntity<TicketInfo> getTicket(@PathVariable("ticketId") Integer ticketId){

    //logic to get ticket data

    TicketInfo tinfo = new TicketInfo();
    tinfo.setTicketId(ticketId);
    tinfo.setPnr("JLJL6868");
    tinfo.setTicketStatus("CONFIRMED");

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

@PostMapping(
        value = "/ticket",
        produces = {"application/json"},
        consumes = {"application/json"}
)
public ResponseEntity<TicketInfo> bookTicket(@RequestBody PassengerInfo request){
```

What is RestController ?

REST Controller Methods

@GetMapping
@PostMapping
@PutMapping
@DeleteMapping

Query Params
Path Params
Request Body
Response Body

@RequestParam
@PathVariable
@RequestBody

produces
consumes
Accept
Content-Type

+++++++++
Swagger
+++++++

-> Swagger is used to generate documentation for REST APIs

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

Assignment : https://youtu.be/ARlz2-Twm-g (watch Swagger video & practise it)

SOAP Webservices

1) App 1 is webservice provider
2) App2 is webservice consumer

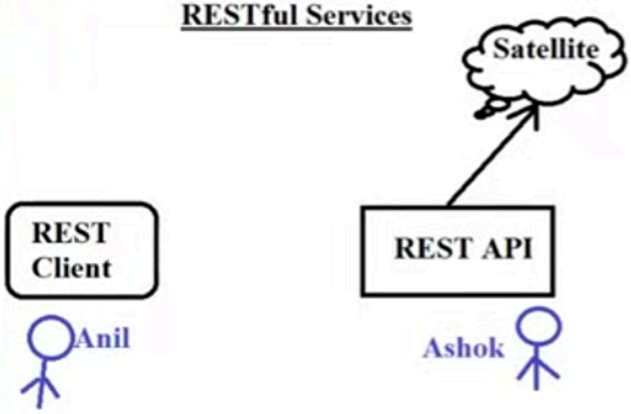
How App2 can access App1?

If App2 knows the details of App1 components then it can access.

How App1 provides component details to App2?

By using WSDL
Webservices description language

RESTful Services



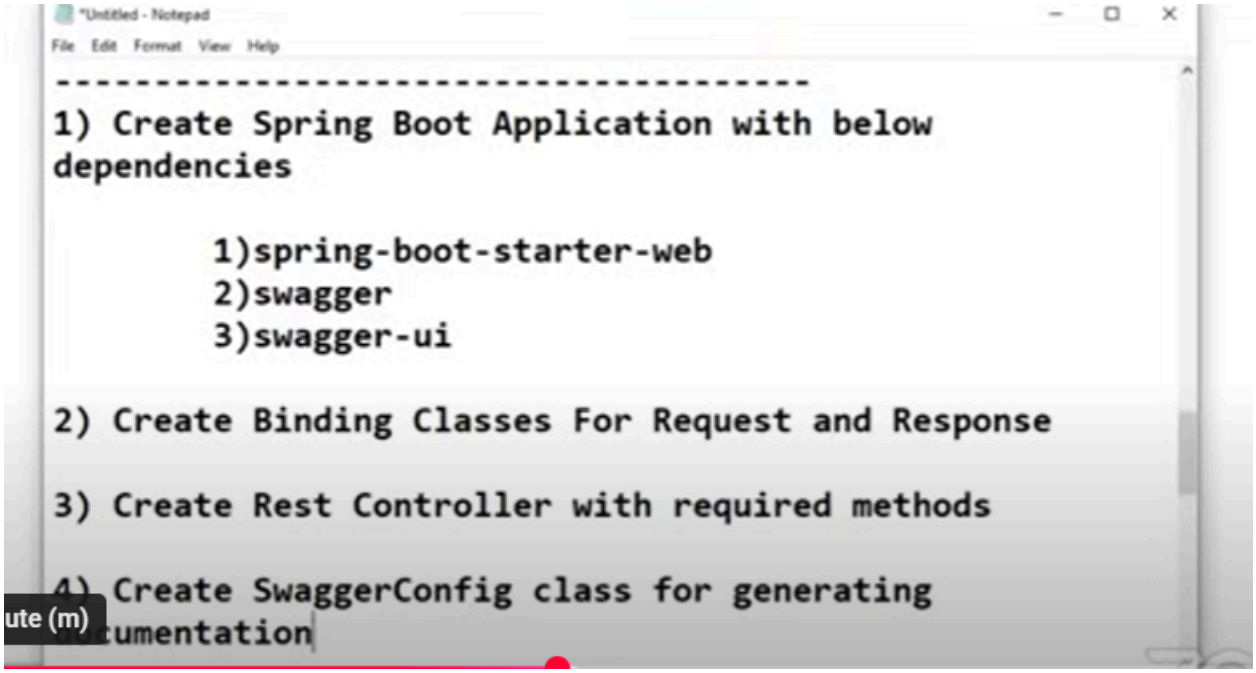
1) Rest api provideing services
2) Rest Clients can access services

How Rest Client can be developed?

If we know the details of REST API then we can develop REST clients.

How we will get REST API details?

Rest api developer should provide documentation about api using Swagger.




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

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

SO in the same IIRCTC Api we will configure our swagger and will test my api using swagger instead of postman

So

1.will create binding class requestdata and responsedata

2.create our restcontroller

3.and in my config package will create SwaggerConfig.java for configuration purpose

4.main is above 3 version we cant use swagger2 dependency instead of that i can use

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.3.0</version>
</dependency>
```

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

```java
package in.jyoti.config;

import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;

@Configuration
public class SwaggerConfig {

    public Docket restApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.basePackage("in.jyoti.rest"))
                .build();
    }


}
```

localhost:8080/swagger-ui/index.html#/

Untitled form - Goo... Gmail YouTube Maps

OpenAPI definition v0 OAS 3.0

/v3/api-docs

Servers

http://localhost:8080 - Generated server url

rest-controller

POST /ticket

Schemas

Requestdata >

Responsedata >

```
http://localhost:7071/v2/api-docs

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

++
CRUD Operations using REST API with MySQL DB
++

-> Download & install MySQL Database (DB Server)

-> Download & install MySQL Workbench (DB client)

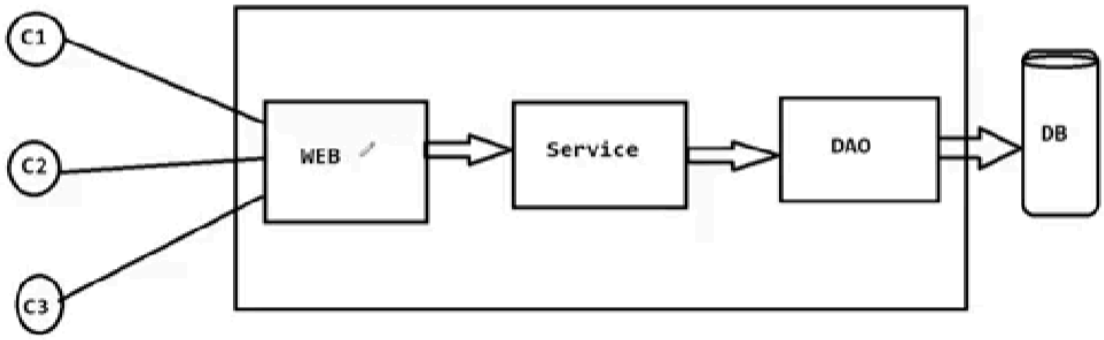
SQL Queries
DB client ----------> DB server

-> Develop REST API using Layered Architecture

1) Web Layer

2) Business / Serice Layer

3) DAO Layer



package in.jyoti.Entity;

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.Data;

@Entity
@Table(name="Book_Tbl")
@Data
public class Book {
        @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name="BOOK_ID")
        private Integer bookId;

        @Column(name="BOOK_NAME")
        private String bookName;

        @Column(name="BOOK_PRICE")
        private Double bookPrice;


}
```

```java
package in.jyoti.Repository;



import org.springframework.data.jpa.repository.JpaRepository;

@Repository
public interface CrudRepo extends JpaRepository<Book, Integer> {

}

```

```java
package in.jyoti.rest;

import java.util.List;

@RestController
public class BookController {
    @Autowired
    private BookService service;
    @PostMapping("/book")
    public ResponseEntity<String> Addbook(@RequestBody Book book){
            String msg=

        return new ResponseEntity<>(msg,HttpStatus.CREATED);
    }
    @GetMapping("/books")
    public ResponseEntity<List<Book>> getAllbook(){
        List<Book> allBooks=service.getAllBook();
        return new ResponseEntity<>(allBooks,HttpStatus.OK);
    }
    @PutMapping("/book")
    public ResponseEntity<String> Updatebook(@RequestBody Book book){
        String msg=service.upsertBook(book);
        return new ResponseEntity<>(msg,HttpStatus.OK);

    }

    @DeleteMapping("/book/{BookId}")
    public ResponseEntity<String> Deletebook(@PathVariable Integer BookId ){
                        String msg=service.DeleteBook(BookId);
                        return new ResponseEntity<>(msg,HttpStatus.OK);
    }
```

Return type of 'Addbook()'

Press 'F2' for focus

```java
package in.jyoti.service;

import java.util.List;

public interface BookService {
    // for insert and update upsert can use
    public String upsertBook( Book book);

    public List<Book> getAllBook();
    public String DeleteBook(Integer BookId);
    // we need



}
```

```java
@Service
public class BookServiceIpl implements BookService {
    @Autowired
    private CrudRepo crudrepo;

    @Override
    public String upsertBook(Book book) {

        Integer bookid=book.getBookId();
        Book save=crudrepo.save(book);
        System.out.println(book);
        if(bookid==null)
            return "Record Inserted";
        else
            return "record updated";
    }

    @Override
    public List<Book> getAllBook() {
        // TODO Auto-generated method stub
        return crudrepo.findAll();
    }

    @Override
    public String DeleteBook(Integer bookId) {
        // TODO Auto-generated method stub
         crudrepo.deleteById(bookId);
         return "record deleted";
    }
```

```
spring.datasource.url=jdbc:mysql://localhost:3306/crud_db?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=placement@7
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
logging.level.org.springframework.web=DEBUG
spring.mvc.pathmatch.matching-strategy=ANT_PATH_MATCHER
```

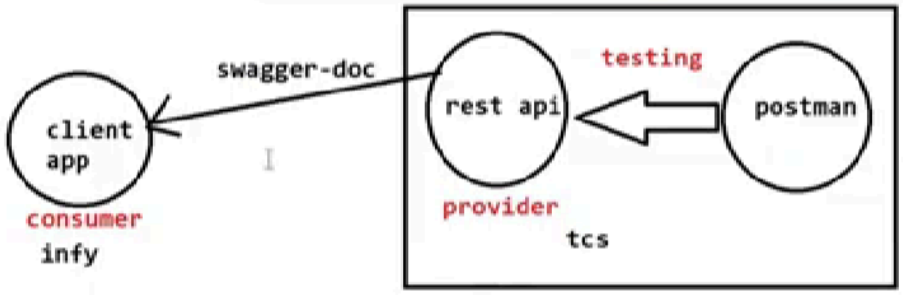
```java
@Configuration
public class Swaggerconfig {
    public Docket restApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.basePackage("in.jyoti.rest"))
                .paths(PathSelectors.any())
                .build();
    }

}
```

Servers

http://localhost:8080 - Generated server url

book-controller

PUT	/book
POST	/book
GET	/books
DELETE	/book/{BookId}



-> Client side development team will understand REST API details using swagger documentation

-> Swagger Doc provides operations(methods), url-patterns, request data & response data

########################
Embedded Database (H2)
#######################

-> Embedded Databases are temporary databases

-> Embedded Databases are used for POC development

-> We no need to download and install embedded databases

-> Embedded Databases will come along with application

-> When we start application then embedded db will start and when we stop the application then embedded db will be stopped

Note: Data is not permenent in the embedded db.

```
spring.datasource.url=jdbc:mem:h2:testdb
spring.datasource.username=sa
spring.datasource.password=sa
spring.datasource.driver-class-name=org.h2.Driver


spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.mvc.pathmatch.matching-strategy = ANT_PATH_MATCHER
```

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

Note: In Memory DBs are not used for realtime project development in the company

-> We can use H2 DB as an in-memory db for practise purpose

-> Add below dependency in pom.xml to

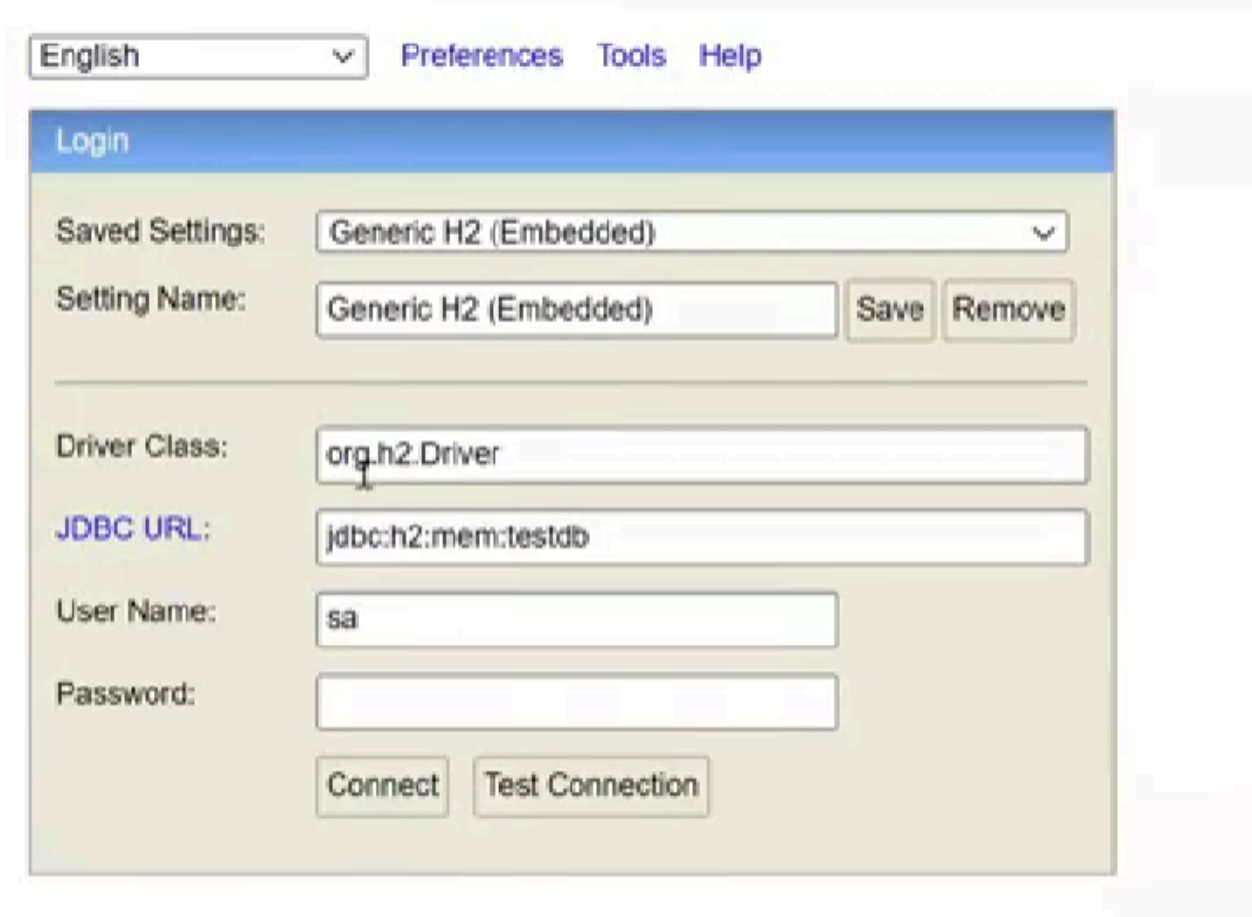
```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
```

- Configure H2 Datasource properties

```
pring.datasource.url=jdbc:h2:mem:testdb
pring.datasource.username=sa
pring.datasource.password=sa
pring.datasource.driver-class-name=org.h2.Driver
```

- Run the application and access h2-console using below URL

http://localhost:8080/h2-console



Lecture 21 are for deployment on heroku and aws

1) Create free tier account in AWS

2) Launch EC2 Linux Virtual Machine in AWS Cloud

3) Connect to EC2 Linux VM using MobaXterm / PuTTY

4) Upload our Spring Boot Jar file into EC2 VM

5) Install Java software in EC2 VM

6) Run the Java application in EC2 VM

==> Launch EC2 VM in AWS : https://youtu.be/ul2iDk8iTps

==> Deploy Spring Boot App in AWS Cloud : https://youtu.be/cRQPgbwOWq0

Application.java

```java
package in.ashokit;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class Application {

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

        String apiURL = "https://ashokit-sb-rest-api.herokuapp.com/";

        RestTemplate rt = new RestTemplate();

        ResponseEntity<String> forEntity = rt.getForEntity(apiURL, String.class);

        String body = forEntity.getBody();

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

Note: To develop REST Client we need to know REST API details

1) API URL
2) API Request Method
2) API request data structure
3) API reponse data structure

-> REST API team will send swagger documentation to REST Client side team. Using swagger documentation we need to understand the api and we need to develop REST Client logic to access REST API.

##
RestTemplate Example to access REST API
##

```java
String apiURL = "https://ashokit-sb-rest-api.herokuapp.com/";

RestTemplate rt = new RestTemplate();

ResponseEntity<String> forEntity = rt.getForEntity(apiURL, String.class);

String body = forEntity.getBody();

System.out.println(body);
```

Now here how rest client we can create ie how clint app can acces our rest apis which we created previously book apis .

Book.java

```java
package in.ashokit.binding;

import lombok.Data;

@Data
public class Book {

    private Integer bookId;
    private String bookName;
    private Double bookPrice;
}
```

Now below is how to send the post request

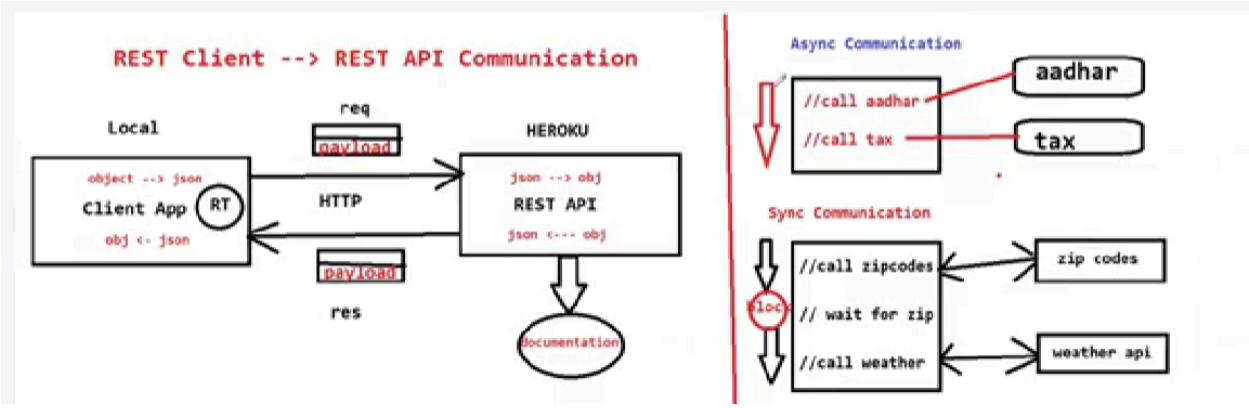
```java
import org.springframework.web.client.RestTemplate;

import in.ashokit.binding.Book;

@Service
public class BookClient {

    public void invokeBookTicket() {

        String apiUrl = "https://ashokit-book-app.herokuapp.com/book";

        Book book = new Book();
        book.setBookName("Java");
        book.setBookPrice(345.00);

        RestTemplate rt = new RestTemplate();

        ResponseEntity<String> postForEntity = rt.postForEntity(apiUrl, book, String.class);
        System.out.println(postForEntity.getBody());
    }
}
```

Start class will be like

```java
package in.ashokit;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class Application {

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

        BookClient bean = run.getBean(BookClient.class);

        bean.invokeBookTicket();
    }

}
```

```
******************Sending HTTP  GET Request using RestTemplate*****************

        String apiURL = "https://ashokit-sb-rest-api.herokuapp.com/";

        RestTemplate rt = new RestTemplate();

        ResponseEntity<String> forEntity = rt.getForEntity(apiURL, String.class);

        String body = forEntity.getBody();

        System.out.println(body);
```

```
*********************Sending GET Request and binding Response JSON to Binding Obj**************

public void invokeGetBooksNew() {
        String apiUrl = "https://ashokit-book-app.herokuapp.com/books";

        RestTemplate rt = new RestTemplate();
        ResponseEntity<Book[]> forEntity = rt.getForEntity(apiUrl, Book[].class);
        Book[] body = forEntity.getBody();
        for(Book book : body) {
            System.out.println(book);
        }
```



Rest client application using webclint now
Here is logic

```
##############################
HTTP Post Request with WebClient
##############################

public void invokeSaveBook() {

        Book book = new Book();
        book.setBookName("Angular");
        book.setBookPrice(450.00);

        String apiUrl = "https://ashokit-book-app.herokuapp.com/book";

        WebClient client = WebClient.create();

        String resp = client.post() // HTTP POST Request
                            .uri(apiUrl) // Endpoint URL
                            .bodyValue(book) // HTTP Request Body Data
                            .retrieve() // Retrieve HTTP Response Body
                            .bodyToMono(String.class) //Bind Response to string var
                            .block(); // Make it as Sync client

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

About Wistia
Report a problem
Copy link and thumbnail

```
##################################
HTTP GET Request using WebClient
##################################

public void invokeGetBooks() {
        String apiUrl = "https://ashokit-book-app.herokuapp.com/books";
        WebClient client = WebClient.create();

        /*
         String body = client.get() // GET Request
                .uri(apiUrl) // Endpoint URL
                .retrieve() // retrieve response body
                .bodyToMono(String.class) // bind response data to string var
                .block(); // make it sync
         */

        Book[] responseData = client.get()
                                        .uri(apiUrl)
                                        .retrieve()
                                        .bodyToMono(Book[].class)
                                        .block();
```

```
##################################
Asynchronus call using Webclient
##################################

public void invokeGetBooksAsync() {
        String apiUrl = "https://ashokit-book-app.herokuapp.com/books";
        WebClient client = WebClient.create();

                client.get()
                        .uri(apiUrl)
                        .retrieve()
                        .bodyToMono(Book[].class)
                        .subscribe(BookClient::respHandler); // Async Communication

        System.out.println("**************Request Sent************");
}

public static void respHandler(Book[] books) {
        for(Book b : books) {
            System.out.println(b);
        }
```

Webclint method use for sync and async both Rest template for sync

application.properties file vs application.yml file
##

-> When we create spring boot application by default application.properties will be created

-> We can avoid hard coded values by configuring app properties in this application.properties file

-> properties file will represent data only in key-value format

Ex:

```
server.port = 9090
spring.mvc.view.prefix = /views/
spring.mvc.view.suffix = .jsp
```

-> properties file will represent data in sequential format

-> .properties file will be supported by only java

-> For every profile we need to create seperate properties file

******** As an alternate to .properties file we can use .yml file in spring boot **********

-> YML stands YET ANOTHER MARKUP Language

-> YML represents data in hierarchial format

```
server:
  port: 9090
```

-> YML supports key-value, list and map values also

-> YML supported by other programming languages also (It is universal)

-> All the profiles can be configured in single YML file

application.properties

```
server.port = 9090

spring.application.name = sb-rest-api

spring.mvc.view.prefix = /views/
spring.mvc.view.suffix = .jsp
```

application.yml

indent spaces are important

```
server:
 port: 9090

spring:
 application:
    name: sb-rest-api

 mvc:
  view:
    prefix: /views/
    suffix: .jsp
```

```
+++++++++++++++++
application.properties
++++++++++++++++++++++
server.port=9090

welcome.msg = Welcome to Ashok IT..!!


++++++++++++++++++++++++++++++++++
@RestController
public class WelcomeRestController {

	@Value("${welcome.msg}")
	private String msg;

	@GetMapping("/")
	public String welcomeMsg() {

		return msg;
	}
}
++++++++++++++++++++++++++++++++++
```

-> application messages and REST ENdpoint URLS are not recommended to hardcode in java classes. Because if we change any message or any url then we have to compile and package entire application.

-> To avoid this problem we will confiure messages and URLs in application.properties file or application.yml file

File Edit View

```
#######################

-> Actuator is one of the powerful feature introduced in Spring Boot

-> Actuators are used to monitor and manage our application

-> Actuators are giving Production ready features for our boot application

++++++++++++++++++
Actuator Endpoints
++++++++++++++++++

/health : To get application health status

/info : To get application information

/beans : To get what beans loaded by our application

/mappings: To get what URL patterns available in our application

/configProps : To get configuration properties loaded by our application
```

```
/heapdump : To download heap data

/threaddump : To get threads information

/shutdown : To stop our application (This is special, it is binding to POST request)


=> To work with actuators we have to use 'spring-boot-starter-actuator' dependency
```

how

```
=> To work with actuators we have to use 'spring-boot-starter-actuator' dependency


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

-> We can see actuator exposed endpoints using below URL

        URL : http://localhost:8080/actuator


Note: /health is a default endpoint which we can access directly


-> We can expose other actuator endpoints using below proper
```

How we can expose endpoint of actuator below

```yaml
management:
  endpoints:
    web:
      exposure:
        include: '*'
        exclude: beans
```

here include all end point except beans so exclude