spring boot

Prepared By - Nirmala Kumar Sahu

INDEX

Spring Boot Rest

Introduction

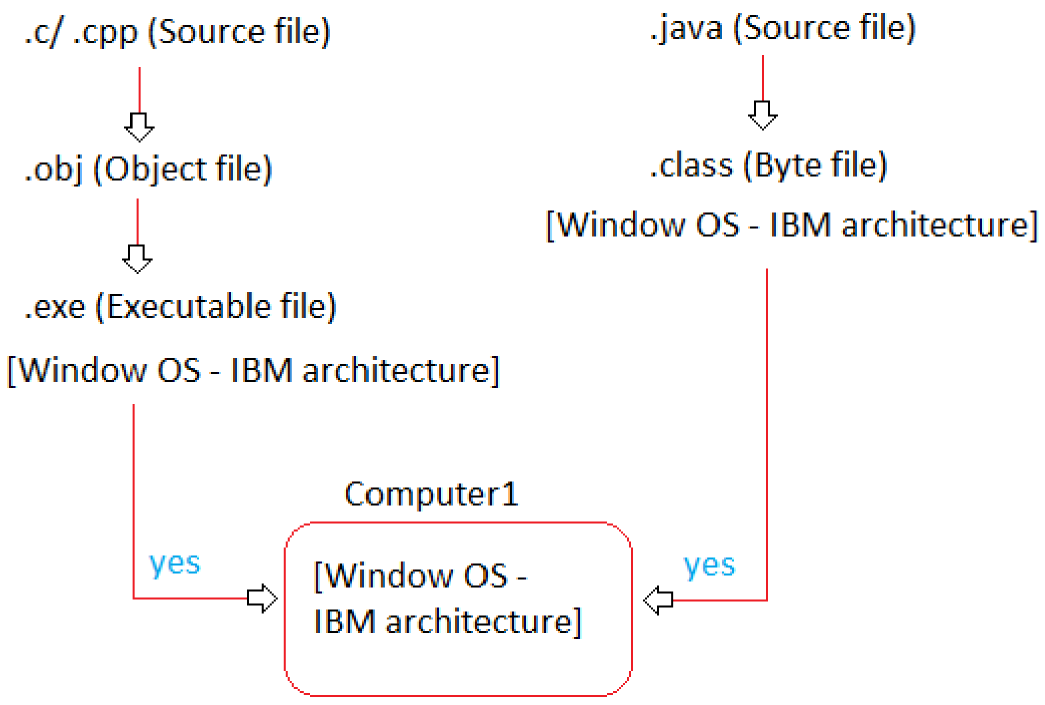
- Java is platform independent and Architecture neutral.
- The plan of manufacturing vehicle is called vehicle architecture like the four-wheeler architectures are,
  - Car architecture
  - Jeep architecture
  - Van architecture
  - Mini Bus architecture
  - Mini Lorry architecture

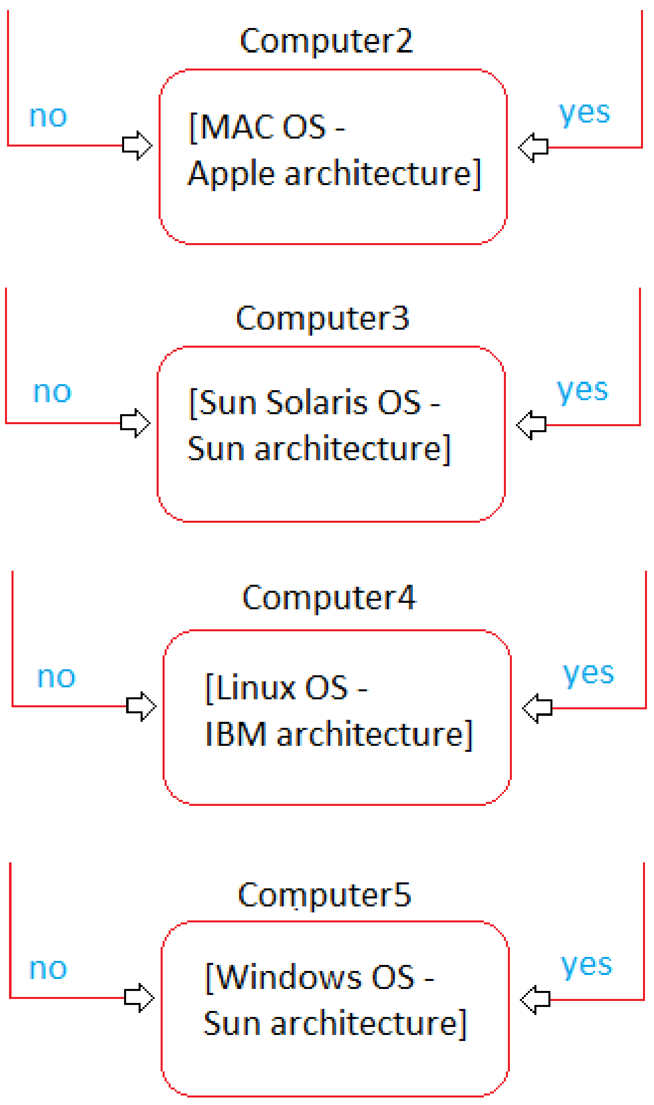
  and etc.

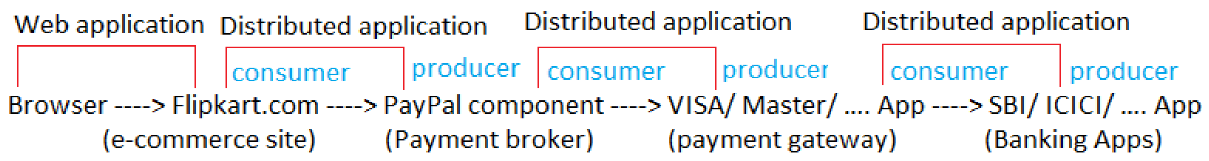
- The plan/ process of manufacturing computer is called computer architecture and the popular architectures are
  - IBM architecture (all our regular laptops and desktops fall under this architecture)
  - Apple/ MAC architecture
  - Sun Architecture

  and etc.

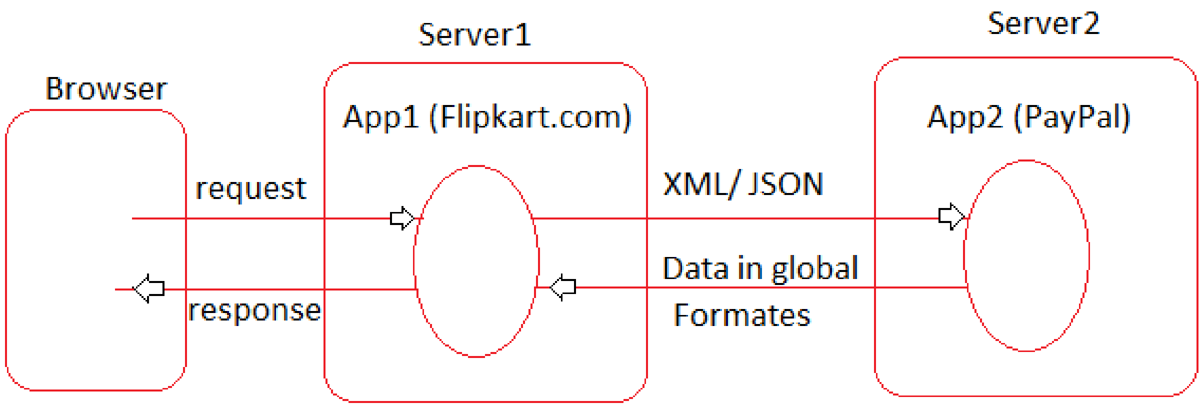
- C, C++ languages are not only platform dependent (OS) and they are also architecture dependent whereas Java is platform (OS) and architecture independent.




Computer2
no
[MAC OS - Apple architecture]
yes
Computer3
no
[Sun Solaris OS - Sun architecture]
yes
Computer4
no
[Linux OS - IBM architecture]
yes
Computer5
no
[Windows OS - Sun architecture]
yes

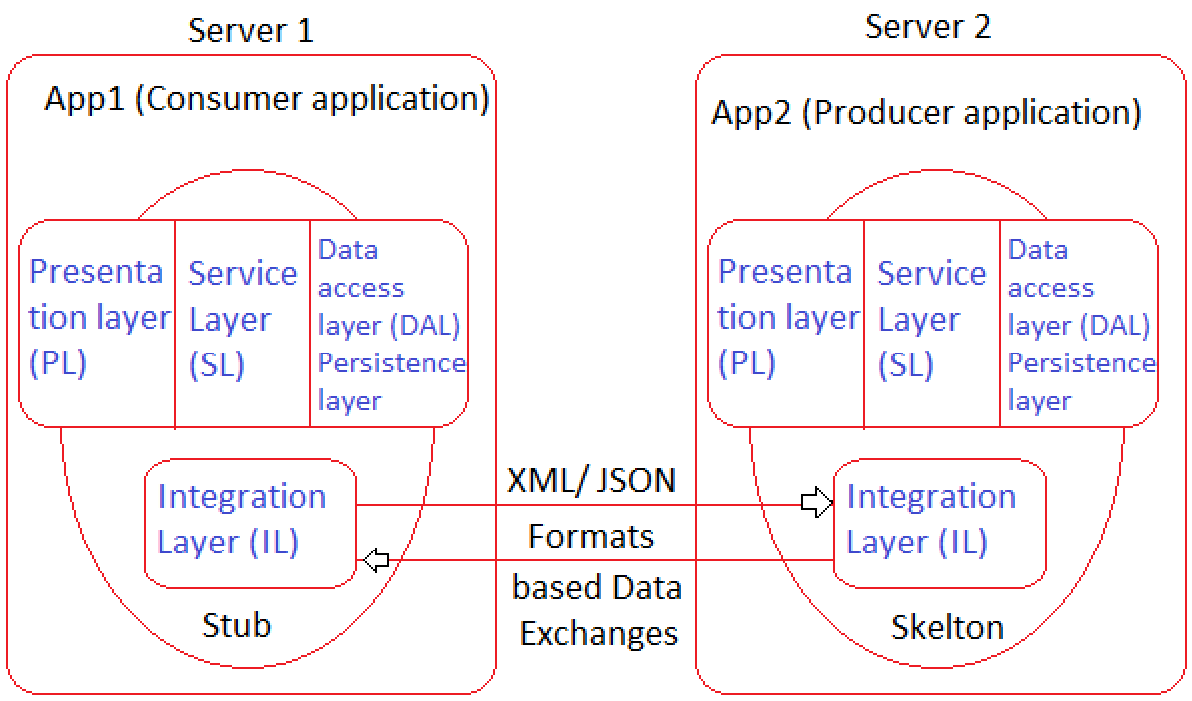


- VISA/ Master/…. and etc. Payment gateway apps providing worldwide infrastructure to perform cards based (debit card/ credit card) based transactions.
- PayPal/ Pay u money/ Rupee/ Razor pay and etc. are payment broker acting as bridge b/w e-commerce apps and payment gateways.
- The cards issued by payment gateway will come to customer through banks by linking with bank accounts.
- Distributed computing/ application development makes two apps of two different servers belonging to same machine or different machines talking each other and also exchanging the data.



- Generally, every application we develop contains 4 layers
    a. Presentation layer/ UI layer & Controller layer (View and controller), contains UI logics and controller logic [using Struts, JSF, Spring MVC, Servlet, JSP, Angular, React JS, HTML, JS, CSS].
    b. Service layer, contains business logics (calculations, analyzations, filtering, sorting and etc.) [using java class or Spring beans (Spring Core + Spring TX)].
    c. Persistence layer/ Data Access layer contains persistence logic like performing CURD Operations [using JDBC, Hibernate, Spring JDBC, Spring Data JPA, Spring Data MongoDB, Spring ORM and etc.].
    d. Data layer, contains the data of the application (The real persistence store) [DB s/w (SQL or No SQL), Files (Text files, properties files, XML files, JSON Files, ...)].

- If want to link one application with another application using Distributed computing or application development environment, we need to add an additional layer in both consumer and producer application that is called as "Integration layer". We can that logics of this Integration layer using Distributed technologies or frameworks like RMI, EJB, Webservices and etc.
- Consumer app/ Client app that consumes the services of server app/ producer app.
- Producer app/ Server app that develops the services of server app/ producer App and keeps them ready for consumption.
- The integration layer logic of consumer app is technically called as Stub logic.
- The integration layer logic of producer app is technically called as Skelton logics


Server 1
App1 (Consumer application)
Presentation layer (PL)
Service Layer (SL)
Data access layer (DAL) Persistence layer
Integration Layer (IL)
Stub
XML/ JSON Formats based Data Exchanges
Server 2
App2 (Producer application)
Presentation layer (PL)
Service Layer (SL)
Data access layer (DAL) Persistence layer
Integration Layer (IL)
Skelton

Other domain distributed technologies:

a. RPC (using C/ C++)
b. DCOM (using Microsoft technologies)
c. .Net Remoting (using .net technologies)
d. Corba (can be implement in multiple languages)
e. Webservices (can be implemented in multiple languages)

- RPC: Remote Procedure Calls
- CORBA: Common Object Request Broker Architecture
- DCOM: Distributed Computing

Different ways of Integration logics in Java environment/ Different Distributed technologies of Java

a. RMI (Remote Method Invocation) (Java based)
b. EJB (Enterprise Java Beans) (Java based)
c. CORBA (can be implemented in multiple languages)
d. Http Invoker (from Spring JEE module) (Java based)
e. Webservices (can be implemented in multiple languages)

RMI:

- Given by Sun MS.
- Part of JDK s/w (JSE module)
- It is language dependent i.e., both Consumer and Server app must be there in Java.

- It is platform independent and Architecture neutral.
- Can't use Internet network as the communication channel between consumer and producer apps.
- Uses JRMP (Java Remote Method Protocol) as the protocol for communication.
- Outdated because of EJB.
- Does not give any built-in middleware services.
  (The readymade secondary logics like security, transaction management and etc.)
- Here data will be exchange in binary format between consumer and producer.

EJB:

- Gives by Sun MS.
- Enhancement of RMI.
- It is language dependent.
- It is platform and Architecture independent
- Can use both LAN (Local Arean network) and Internet (WAN) as communication channel.
- Needs the heavy weight EJB Container to execute EJB components
  [EJB container is part of another heavy weight application server like WebLogic, Glassfish and etc.).
- Gives lots of built-in middleware (It was popular for banking apps earlier for this reason)
- Outdated because of Webservices.
- It is technology of JEE module and EJB Containers of application server softwares will come as the implementations.
- We must deploy the developed EJB components in the EJB containers of application server for execution.
- Very complex to lean and execute.
- Here also data will be exchange in binary format between consumer and producer.

Note: Though Tomcat is called as application server from version 7. It is still not providing EJB container.

CORBA:

- Platform, architecture and language independent.
- We can develop/ produce and consume services in multiple languages like Java, C and etc.

- CORBA is specification and it will be implemented as IDL (Interface definition language).
- It is complex to learn and apply.
- It is heavy weight to implement.
- CORBA looks great concept wise, but gives problems towards the implementations.

Webservices:

- Webservices is a mechanism of linking two applications as consumer and producer applications using the protocol HTTP.
- Webservices in platform independent and architecture independent and language independent.
- A producer developed in "Java" can be used/ consumed in .Net, Java, PHP, Phyton, JavaScript and etc. (even reverse is possible).
- Webservices says develop/ produce anywhere and consume anywhere.
- Webservices makes the consumer and producer apps exchanging data either in XML or in JSON (global formats).
- Java and other languages have provided multiple APIs/ technologies and frameworks to implement webservice logics as Skelton logics (Integration Logics of Producer) and as Stub logics (Integration logics of consumer).
- Webservices support HTTP request and HTTP response-based method invocation i.e., as HTTP request the consumer app invokes method of producer app and the results method execution goes back to consumer app as HTTP response.

Two ways of implementing Webservices

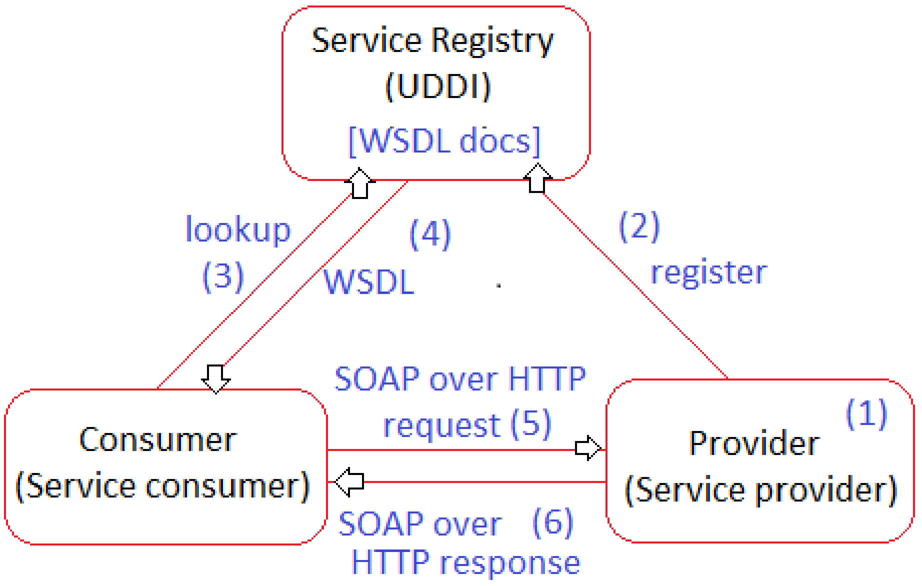
a. SOAP Webservices
b. RESTful Webservices

SOAP: Simple Object Access Protocol (Protocol)
REST: REpresentational STateful web services
(It is not a protocol it is kind of architecture/ mechanism)

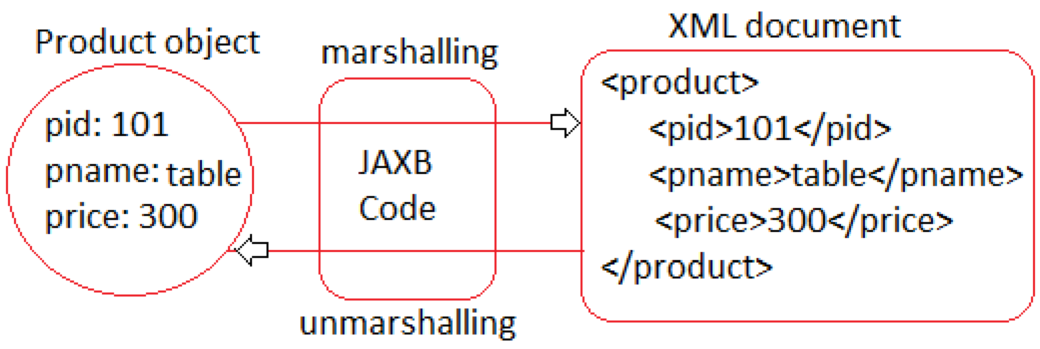
SOAP Webservices

- These runs based 3 component principles,
    a. Service Provider (producer/ server)
    b. Service Consumer (Consumer/Client)
    c. Service Registry (where services will be will be registered to expose)

- The Registries in SOAP based Webservices environment is UDDI (Universal Description Discovery and Integration).

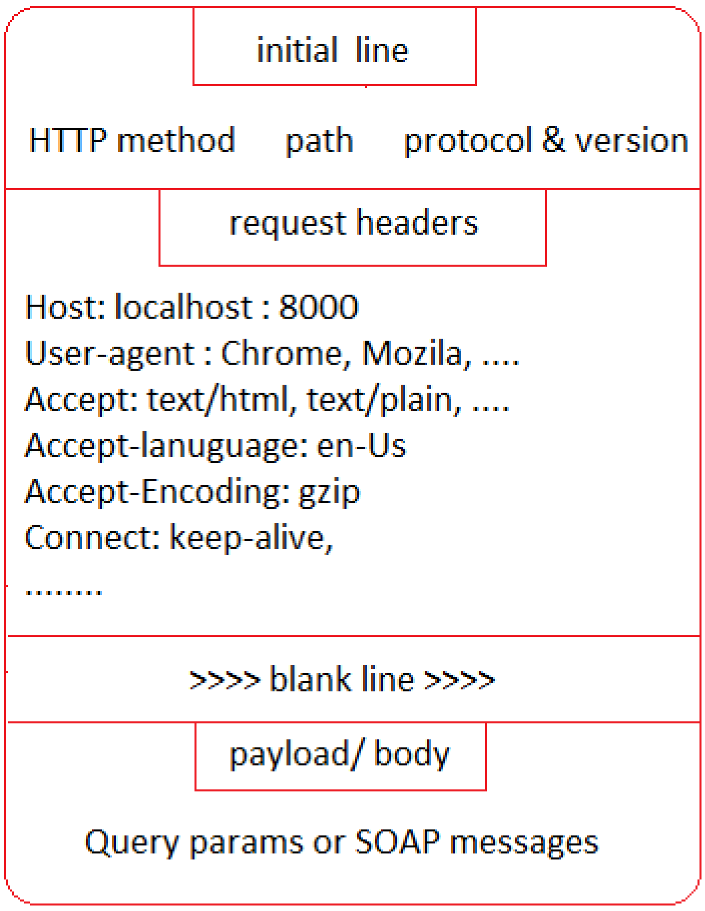


- For every SOAP webservice that is registered in Service Registry there will be one WSDL doc (xml document) having multiple details about web service like how to consume, the services names and etc. (endpoint details).
- WSDL: Web Service Description Language.
- The HTTP request carries XML based SOAP message as request body nothing but inputs.
- The HTTP response carries XML based SOAP message as response body nothing but outputs.
- SOAP messages are strictly typed or complex XML messages which will be converted to Java objects and reverse using JAXB API.
- JAXB: Java Architecture for XML Binding (nothing but converting Java object data to XML content and vice-versa).
- Java object data to XML conversion is called marshalling and reverse is called unmarshalling.



- The consumer sends inputs to producer as SOAP message (XML) in HTTP request.
- The Producer sends outputs to consumer as SOAP message (XML) in HTTP response.
- The http request and http response contain two parts
    a. HEAD part
       contains two sections
        - Initial Line
        - headers
    b. Body part

Structure of HTTP request

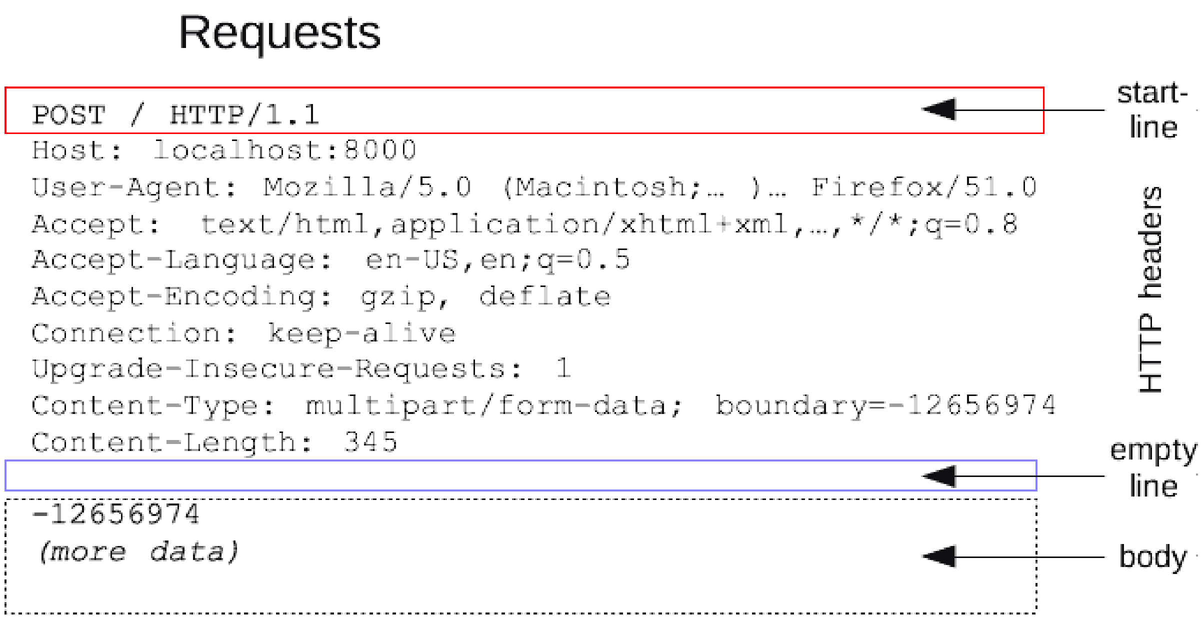


- In Normal HTTP request, the request body will be query String having request param names and values like sno=101&sname=raja&sadd=hyd. In SOAP over HTTP request, the request body will be XML based SOAP message (XML tags).

HTTP methods or modes are (9)

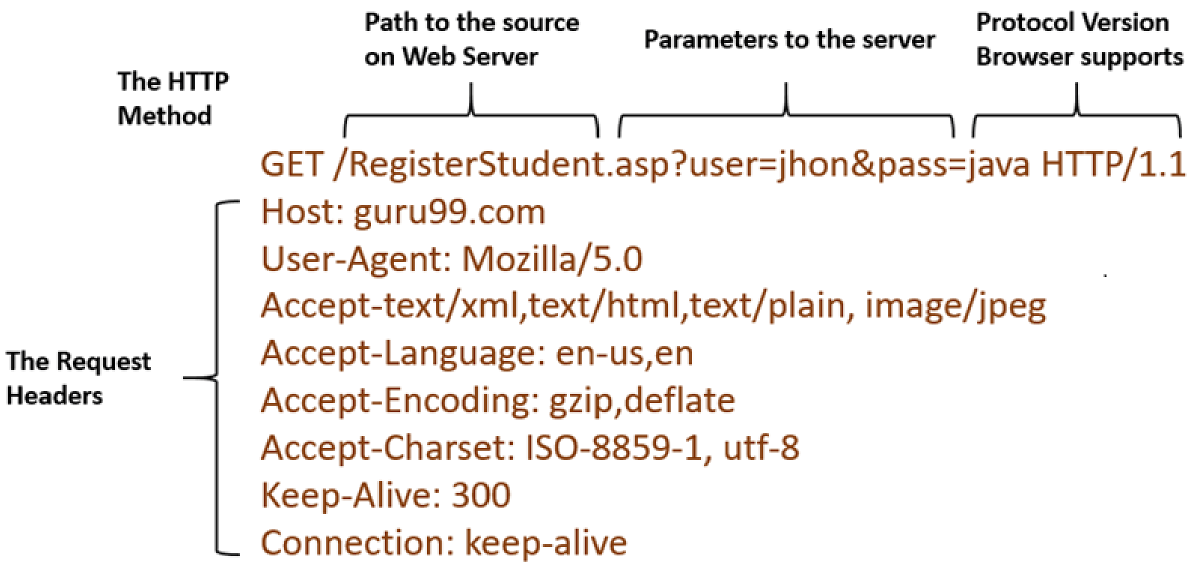
- GET

- POST
- HEAD
- TRACE
- PUT
- DELETE
- OPTIONS
- CONNECT (reserved for future)
- PATCH (for partial updates)



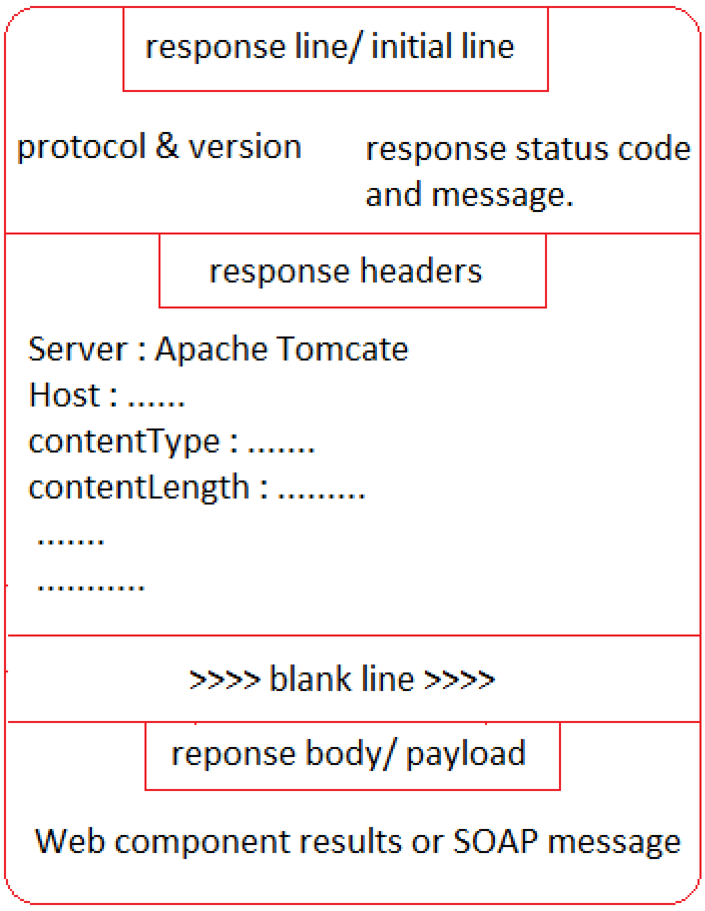
```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh;… )… Firefox/51.0
Accept: text/html,application/xhtml+xml,…,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345

-12656974
(more data)
```

- POST mode request contains request body representing query String whereas GET Mode request does not request body because its carries inputs as query String appended to the request URL.

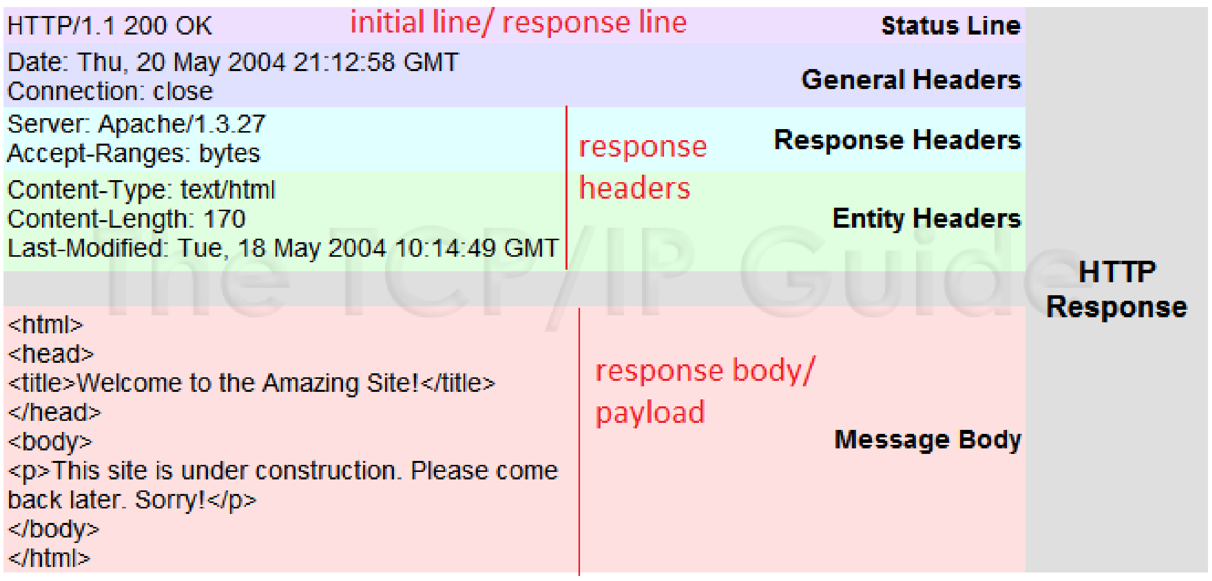

Path to the source on Web Server
Parameters to the server
Protocol Version Browser supports
The HTTP Method
The Request Headers

```
GET /RegisterStudent.asp?user=jhon&pass=java HTTP/1.1
Host: guru99.com
User-Agent: Mozilla/5.0
Accept-text/xml,text/html,text/plain, image/jpeg
Accept-Language: en-us,en
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1, utf-8
Keep-Alive: 300
Connection: keep-alive
```

Structure of HTTP response



- Normal HTTP response contains the web comp generated output (generally HTML code or plain text) as response body whereas SOAP over HTTP response contains the SOAP message (XML) as the response body.

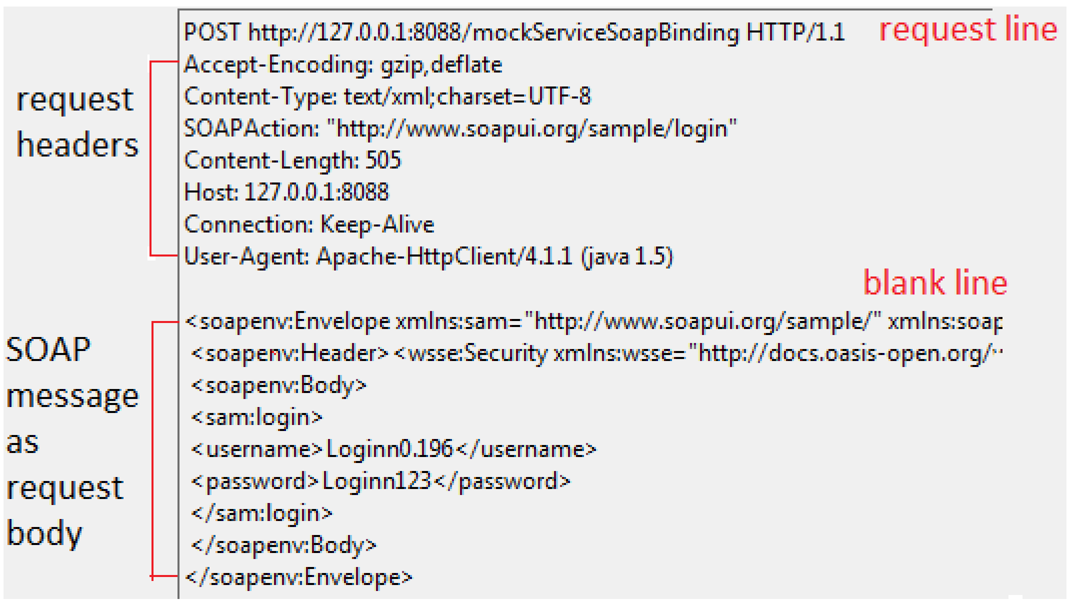


```
HTTP/1.1 200 OK
Date: Thu, 20 May 2004 21:12:58 GMT
Connection: close
Server: Apache/1.3.27
Accept-Ranges: bytes
Content-Type: text/html
Content-Length: 170
Last-Modified: Tue, 18 May 2004 10:14:49 GMT

<html>
<head>
<title>Welcome to the Amazing Site!</title>
</head>
<body>
<p>This site is under construction. Please come
back later. Sorry!</p>
</body>
</html>
```

HTTP response status code:

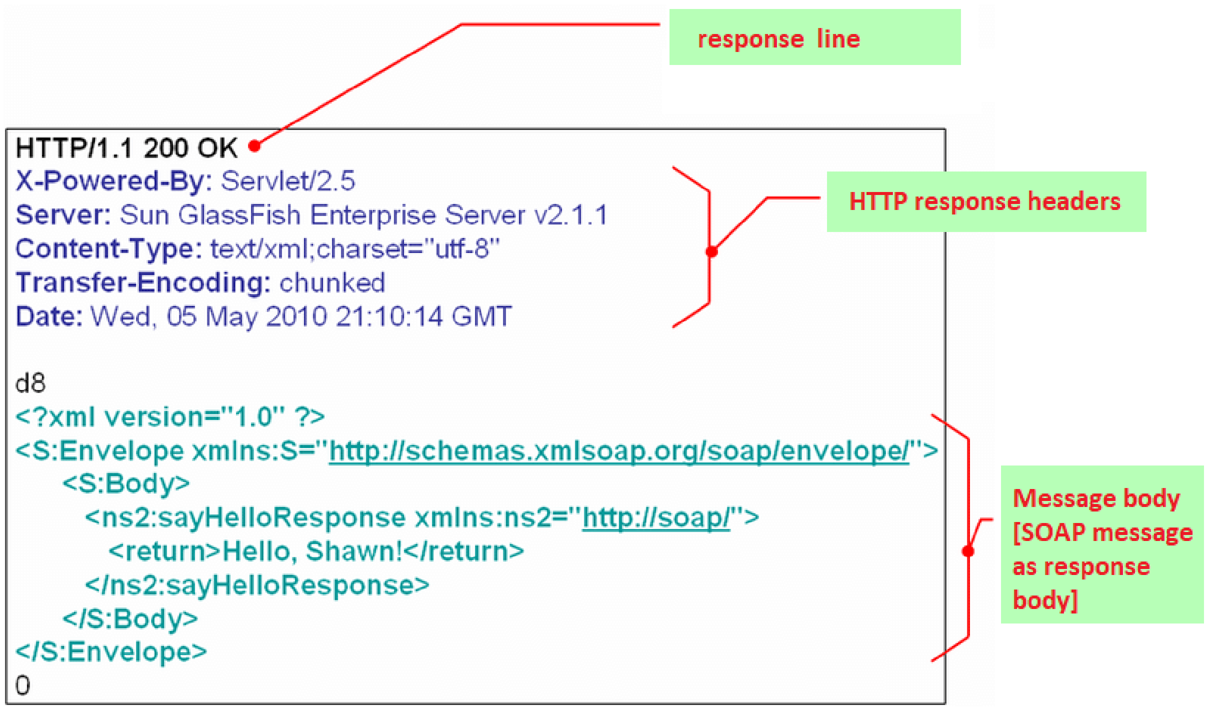
- 1xx (100-199): Informational
- 2xx (200-299): Success
- 3xx (300-399): Redirection
- 4xx (400-499): Incomplete (Client-side errors)
- 5xx (500-599): Server-side errors

SOAP over HTTP request:



```
POST http://127.0.0.1:8088/mockServiceSoapBinding HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: "http://www.soapui.org/sample/login"
Content-Length: 505
Host: 127.0.0.1:8088
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)

<soapenv:Envelope xmlns:sam="http://www.soapui.org/sample/" xmlns:soap
 <soapenv:Header><wsse:Security xmlns:wsse="http://docs.oasis-open.org/
 <soapenv:Body>
 <sam:login>
 <username>Loginn0.196</username>
 <password>Loginn123</password>
 </sam:login>
 </soapenv:Body>
</soapenv:Envelope>
```

SOAP over HTTP response:



```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: Sun GlassFish Enterprise Server v2.1.1
Content-Type: text/xml;charset="utf-8"
Transfer-Encoding: chunked
Date: Wed, 05 May 2010 21:10:14 GMT

d8
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
      <ns2:sayHelloResponse xmlns:ns2="http://soap/">
        <return>Hello, Shawn!</return>
      </ns2:sayHelloResponse>
    </S:Body>
</S:Envelope>
0
```

- SOAP based webservices fall under SOA architecture (Service Oriented Architecture).
- SOA is the mechanism of linking two applications.
- SOA contains service provider, service consumer and service registry as shown in the diagram.
- In Java environment JAX-WS is the API or technology (part of JSE module) that is given to developing SOAP based webservices and the frameworks implementing SOAP based webservices are,
    - Apache Axis
    - Apache CXF

    and etc.

Limitations of SOAP based Webservices:

a. These are based on SOA where separate registry is required to publish/ find webservices.
b. The service registry in SOA is very heavy to maintain.
c. The SOAP messages in HTTP request and HTTP response are complex to build and heavy to send over the network
d. Understanding about webservice by reading WSDL document is not easy process.
e. SOAP message use complex Schema, so building SOAP message is costly.
f. Does not give flexibility to send data in different global formats, should always use the heavy SOAP message as XML messages.
   and etc.

- To overcome these problems, use RESTful Webservices.

RESTful Webservices

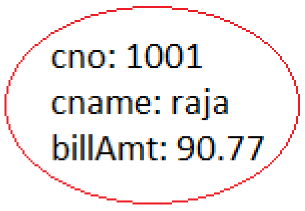
- REST is not a protocol; REST is the mechanism or architecture REpresenting the STate of web component in HTTP request and in HTTP response. So, it is called RESTful webservices.
- Allows to send data in both global formats XML, JSON.

Note:

- ✓ XML: eXtensible Markup language
  To convert XML tags data to Java object (unmarshalling) and reverse (marshalling) we use JAXB API (Java Architecture for XML Binding)
- ✓ JSON: Java Script Object Notation (nothing but object in Java script. The process of converting Java object to JSON content (key: value pairs) is called Serialization and reverse called Deserialization.

Java data representation

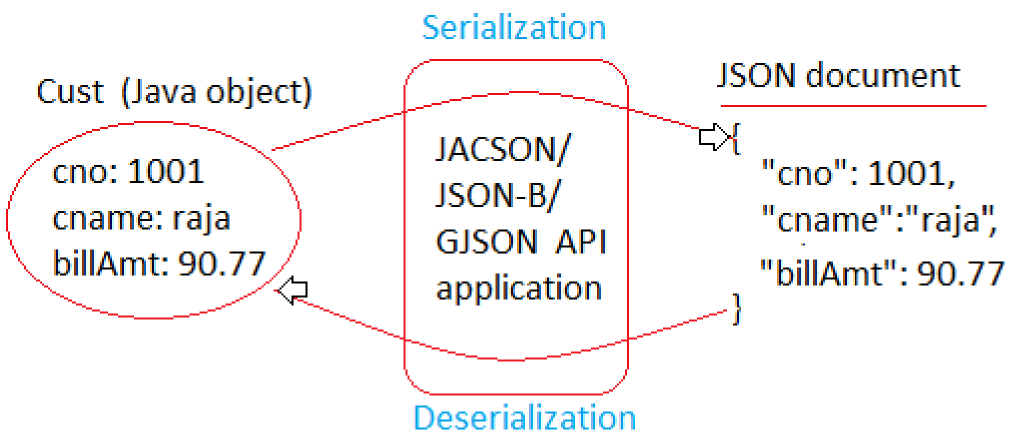
```
String name="raja"
int age=30;

Customer cust=new Customer();
cust.setCno(1001);
cust.setCname("raja");
cust.setBillAmt(90.77);
```



JavaScript data representation

```
var/let name="raja";
var/let age=30;
var/let cust= { "cno": 1001,
                "cname":"raja',
                "billAmt": 90.77};
```

- To convert JSON content into Java object and for reverse operation we use JSON APIs
  E.g., JACSON, JSON-B, GJSON and etc.



- JSON is light weight compare to XML to represent the data.

XML doc

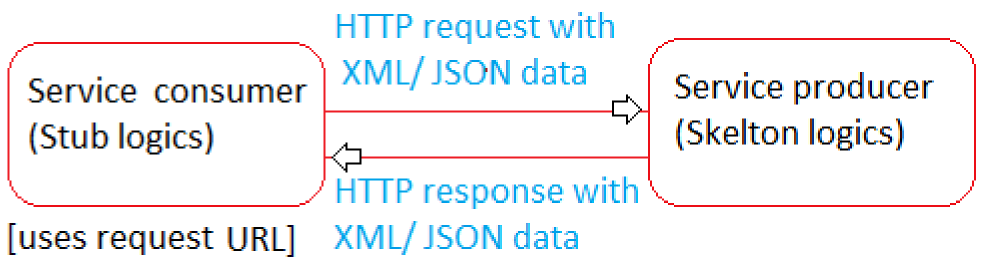
```
<customers>
     <customer>
          <cno>101</cno>
          <cname>raja </cname>
          <billAmt>90.77 </billAmt>
     </customer>
</customers>
```

JSON document

```
{
   "cno": 1001,
   "cname":"raja",
   "billAmt": 90.77
}
```

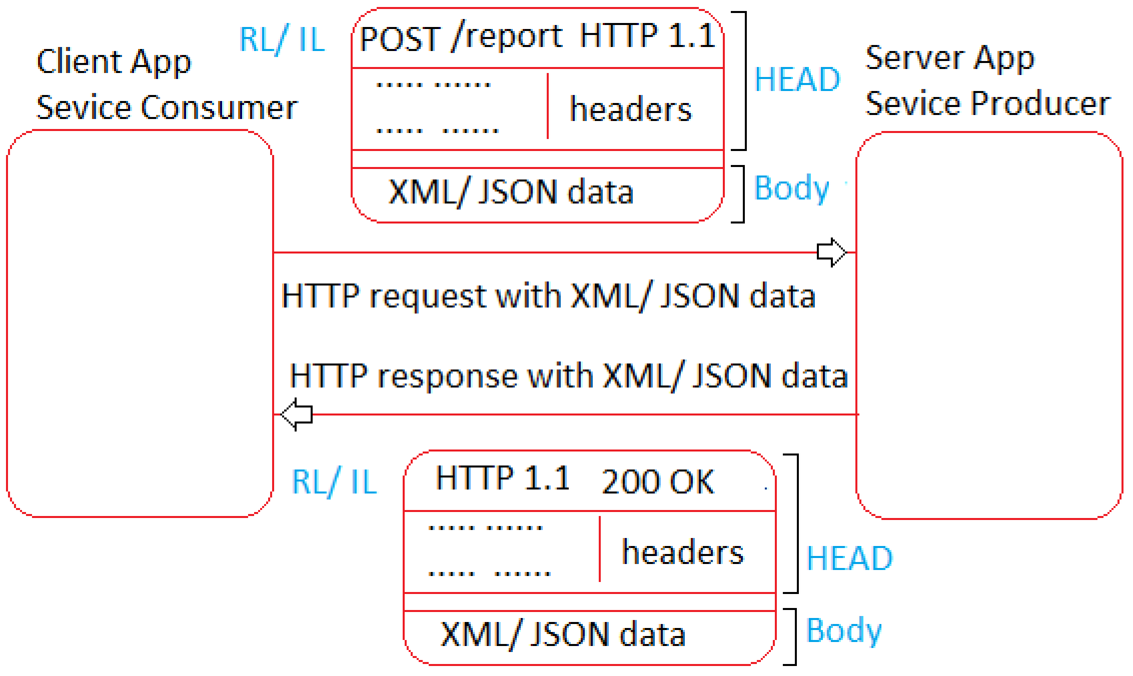
XML document	JSON document
a. Here compare to data the structure is more like closing tag for every opening tag is bit extra.	a. Here data and structure both are in good combination
b. Gives structured data having relationship among the data members in bit heavy environment.	b. Gives structured data having relationship among the data members in light environment

- RESTful is not based on SOA (Service Oriented Architecture), it is based on Consume- Producer Architecture with FrontController support i.e., no service registry is required here.
- Every Restful service producer is a web component and it is identified with its request URL



HTTP request with XML/ JSON data
Service consumer (Stub logics)
Service producer (Skelton logics)
HTTP response with XML/ JSON data
[uses request URL]

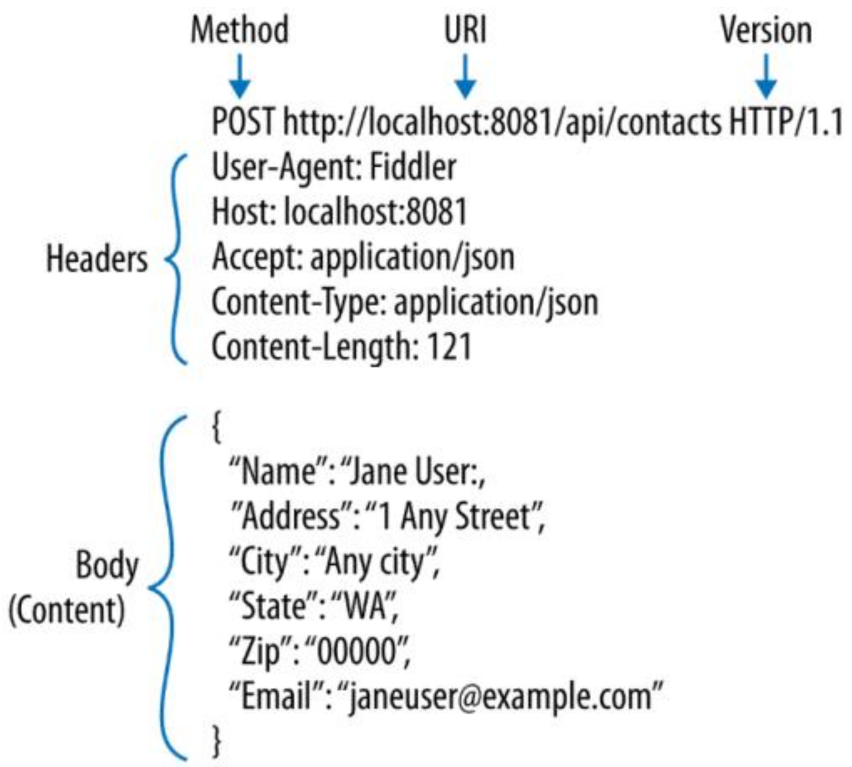
- In RESTful webservices we prefer JSON data a lot compare to XML data.



- The service producer (Restful component) can be developed in any language like Java, .Net, PHP, Phyton and etc.

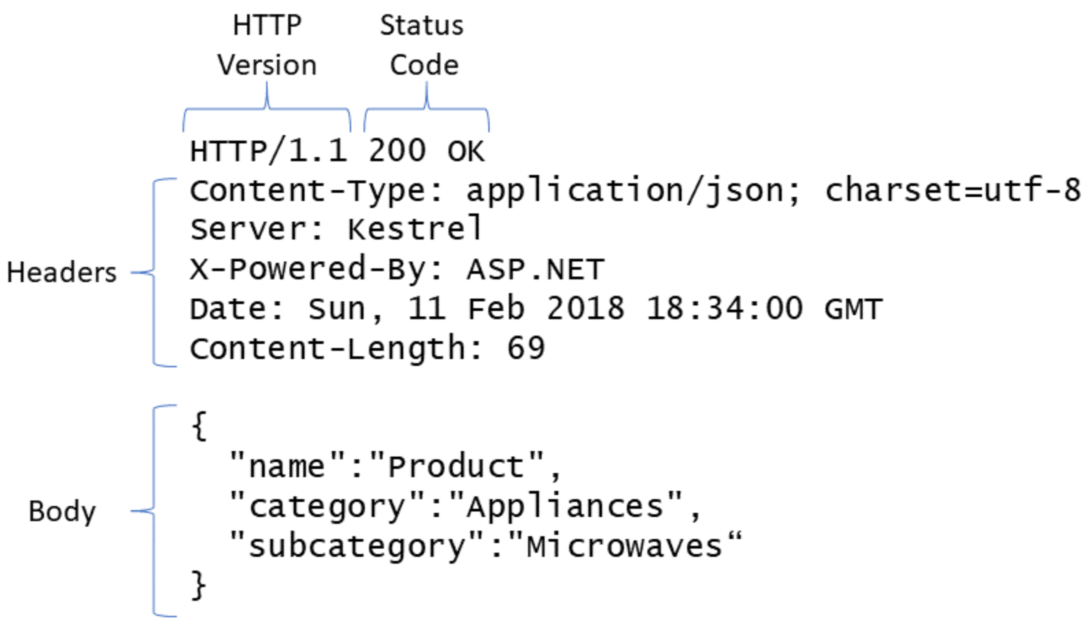
- The service consumer (client) can be developed in any language like Java, .Net, PHP, Phyton and etc. and also can be as any app like Java app, Android app, Desktop app, PHP app, Angular app, React app, IOS app and etc.

Http request with JSON Body:



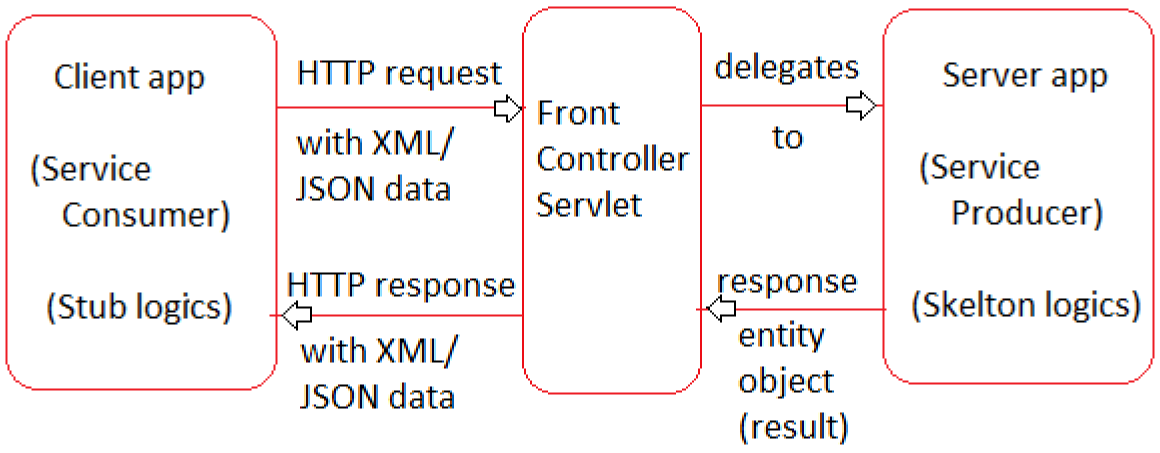
```
POST http://localhost:8081/api/contacts HTTP/1.1
User-Agent: Fiddler
Host: localhost:8081
Accept: application/json
Content-Type: application/json
Content-Length: 121

{
  “Name”: “Jane User:,
  ”Address”: “1 Any Street”,
  “City”: “Any city”,
  “State”: “WA”,
  “Zip”: “00000”,
  “Email”: “janeuser@example.com”
}
```

Http response with JSON Body:



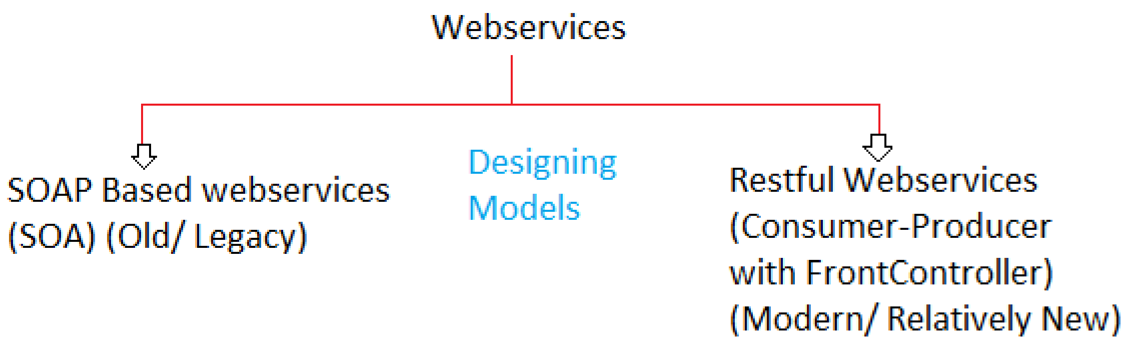
```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Kestrel
X-Powered-By: ASP.NET
Date: Sun, 11 Feb 2018 18:34:00 GMT
Content-Length: 69

{
  "name":"Product",
  "category":"Appliances",
  "subcategory":"Microwaves“
}
```

- The service provider in Restful app is Java class and it cannot take HTTP requests directly. So, we put one FrontController servlet like DispatcherServlet (Spring), ActionServlet (Struts) and etc.
- The special servlet that acts as entry and exit point for all requests and responses is called Frontcontroller servlet.



- In Java Jax-RS is technology/ API to develop Restful webservices whereas Spring Rest, Rest easy, Restlet, Jersy and etc. are the frameworks to Restful webservices implementation.
- Spring Rest is given on top of Spring MVC (Spring Rest = Spring MVC ++).

Summary on Webservices:


Webservices
SOAP Based webservices (SOA) (Old/ Legacy)
Designing Models
Restful Webservices (Consumer-Producer with FrontController) (Modern/ Relatively New)

Jax-WS ← Programming APIs → Jax-RS
Apache Axis, Apache CXF ← Programming Frameworks →Spring Rest, Rest easy, Restlet, Jersy
SOAP over HTTP ← Protocol → HTTP with JSON/ XML data

Note:
- ✓ Developing service providers in webservices is also called as developing APIs.

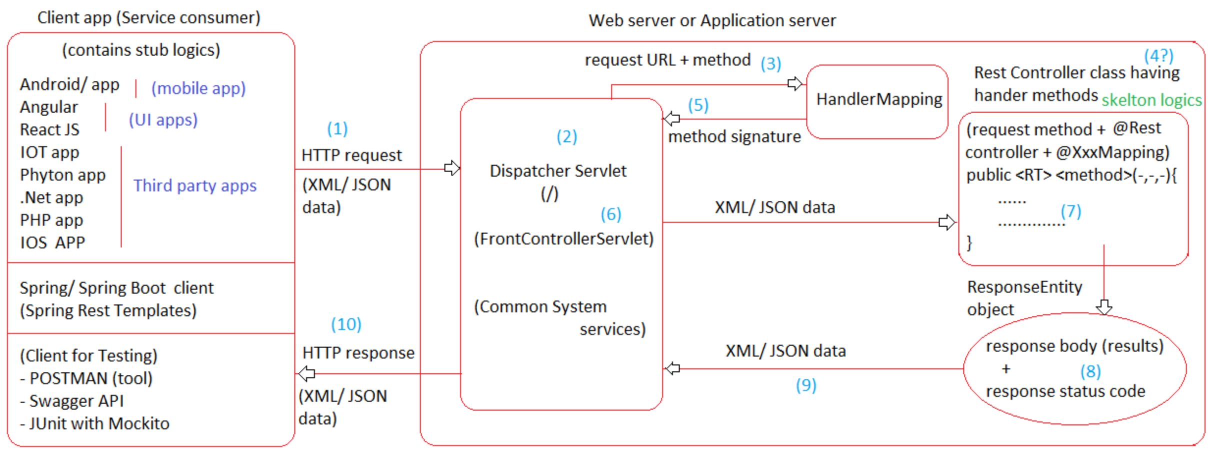
✓ Gathering and providing information that are required to develop service consumer for invoking the services of specific service provider is nothing but providing endpoints.

- Spring Rest = Spring MVC ++
- That means Spring MVC can be used to develop web application and also RESTful webservice components.
- Spring Rest is given based on Consumer - Producer + FrontController pattern.
- RESTful webservices does not fall under SOA (Service Oriented Architecture) because there is not service registry in RESTful webservices more importantly the service provider component of Restful webservice is identified with its request URL or path.
- FrontController is special web component of web application that traps and takes either all or multiple HTTP requests applies common system services and delegates requests to appropriate components (controller classes) and gathers the results to send to clients.
- In fact, it acts as entry and exit point for all the requests and responses of the web application.

Spring Rest Flow

Spring MVC Flow to towards developing RESTful webservices

- In Restful web applications, there will not be any kind of view components, the methods of controller or Rest Controller class directly send response to client app through Dispatcher Servlet having XML/ JSON data which initially generated as ResponseEntity object and later



Note:

✓ In web application the client is browser window (not programmable) giving request and getting response.

- ✓ In Distributed apps the client/ service consumer is programmable app having logics to invoke methods of service producer (webservice components). (Here browser cannot be taken as client).
- ✓ The methods of @RestController class does not need ViewResolver and View components because they have natural behaviour of sending response to Client app directly through DispatcherServlet.

With respect diagram:

1. The service consumer app writes stub logic by invoking service producer methods HTTP request will be generated having XML/ JSON data.
2. The FrontController, DispatcherServlet (DS) traps and takes requests to apply common system services like logging, auditing and etc.
3. The DS handovers the request to handler mapping component giving request URI/ path and request mode.
4. (4?) The handler mapping component searches for the @Controller or @RestController classes for matching handler method and gets its signature using Reflection API.
5. HandlerMapping components gives method signature to DS.
6. DS prepares necessary objects as required for the signature, gets controller or RestController class object and calls method.
7. Handler method executes either to process request directly or by taking the support of Service, DAO classes.
8. Handler method prepares HTTP ResponseEntity<T> object having response body (results) and HTTP status code.
9. Handler method returns HTTP ResponseEntity<T> object to DS having XML/ JSON data.
10. DS sends HTTP response to client app having XML/ JSON data.

- Developing Spring MVC handler method of controller class without returning logical view name and making it returning ResponseEntity<T> object makes the handler method ready for Spring Rest environment. (In fact, it becomes indirectly business method of Restful webservices).
- To send response to browser/ client without involving View, we need to place @ResponseBody on the top of handler method in @Controller class. Instead of writing two annotations we can just use @RestController.
  @RestController = @Controller + @ResponseBody

E.g.,1:
@Controller

```
@RequestMapping("/customer")
public class CustomerController {

        @ResponseBody
        @GetMaping("/display")
        public ResponseEntity<String> dispplayMessage(){
                …………
                ………
                return obj of ResponseEntity;
        }
}
```

E.g.,2:

```
@RestController
@RequestMapping("/customer")
public class CustomerController {

        @GetMaping("/display")
        public ResponseEntity<String> dispplayMessage(){
                …….
                ……….
                return obj of ResponseEntity;
        }
}
```

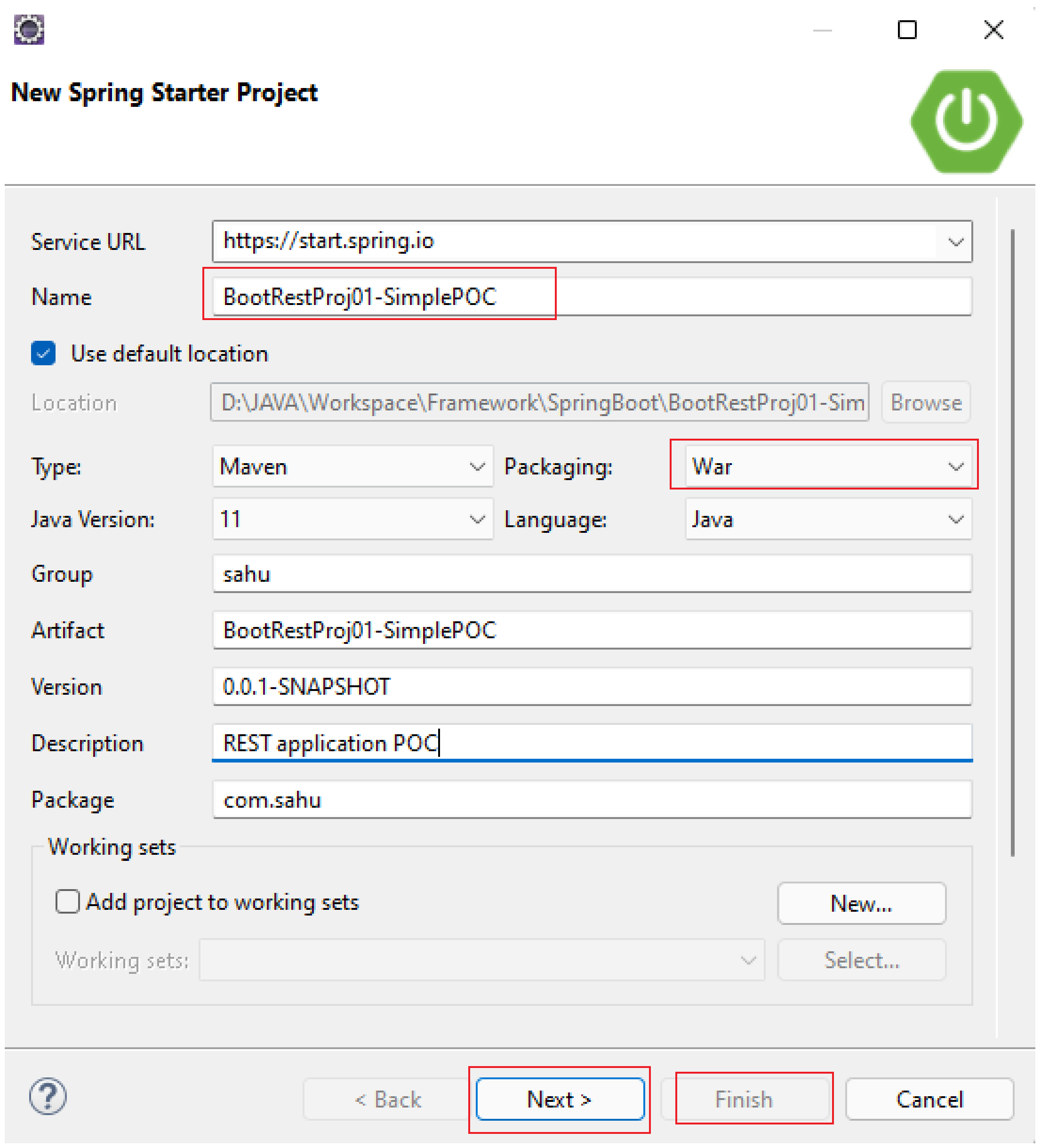
- In Spring MVC/ Spring Rest the following configuration takes place automatically
    - DispatcherServlet configuration as FrontController with "/" URL pattern.
    - HandlerMapping (Default is RequestMappingHandlerMapping).
    - Error Filters displaying white label error pages.
    - ViewResolver (default InternalResourceViewResolver).
      and etc.

- Above last two sub point, we do not them mostly in Spring Rest programming because it does not use browser as client.
- ResponseEntity<T> object contains two parts
    a. Response body (String/ object/ collection)
    b. Response Status code (we use HttpStatus enum constants for like HttpStatus.OK and etc.)

- ResponseStatus/ HttpStatus indicates whether the generated response/ output/ result should be given to client as success output (200-299) or as client-side error (400-499) or as server-side error (500-599) and etc.
- ResponseEntity<T> indicates (as the return type of handler method) that the generated output/ result should go to client/ browser directly through DispatcherServlet without involving any ViewResolver and View components.

Procedure to develop and test First Spring Rest application

Step 1: Create Spring Boot Starter project adding Spring web starter and taking packaging as War.


New Spring Starter Project

Service URL: https://start.spring.io
Name: BootRestProj01-SimplePOC
[✓] Use default location
Location: D:\JAVA\Workspace\Framework\SpringBoot\BootRestProj01-Sim
Type: Maven
Packaging: War
Java Version: 11
Language: Java
Group: sahu
Artifact: BootRestProj01-SimplePOC
Version: 0.0.1-SNAPSHOT
Description: REST application POC
Package: com.sahu
Working sets
[ ] Add project to working sets
Working sets:

Fill the required things then click on Next, then choose Spring Web starter then click on Next & Finish.

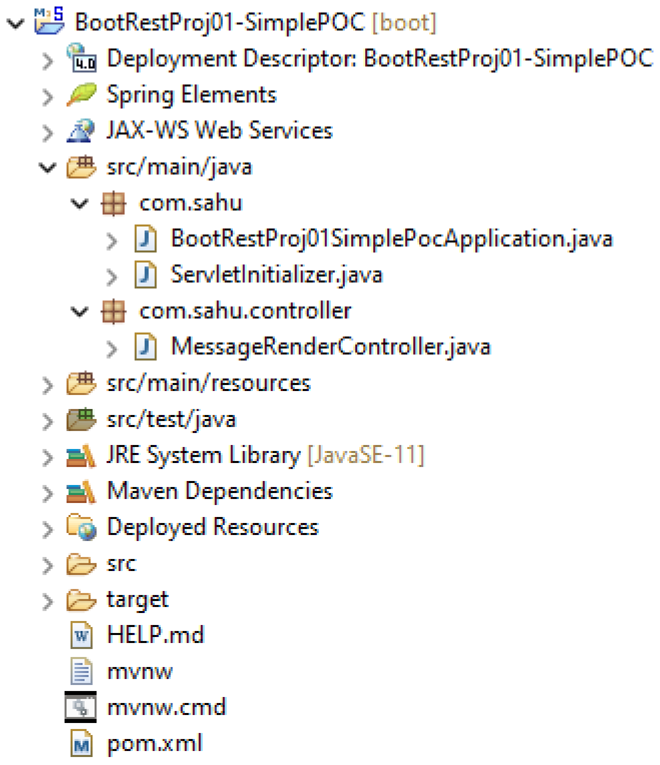
Step 2: Develop the @RestController class.

Step 3: Test the app using browser or POSTMAN tool.

Note:

- ✓ If ResponseEntity object is having other than <String> type as generic type (object/ collection), then it internally converts and holds output as JSON data (key-value) pairs, whereas String type contains normal text content.
- ✓ Since ResponseEntity<String> is taken and handler method type is "GET" (in this controller) we can send that request even from browser (otherwise not possible from browser).
- ✓ Not recommended to use browser because it can send only GET or POST mode requests whereas @RestController can have GET, POST, DELETE, PUT, PATCH and etc. modes handler methods).

Directory Structure of BootRestProj01-SimplePOC:



```
BootRestProj01-SimplePOC [boot]
  Deployment Descriptor: BootRestProj01-SimplePOC
  Spring Elements
  JAX-WS Web Services
  src/main/java
    com.sahu
      BootRestProj01SimplePocApplication.java
      ServletInitializer.java
    com.sahu.controller
      MessageRenderController.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  Deployed Resources
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use Spring Web starter during project creation.

- Then place the following code with in their respective files.

MessageRenderController.java

```
package com.sahu.controller;

import java.time.LocalDateTime;

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

@RestController //@Controller + (@ResponseBody on all the handler
methods)
@RequestMapping("/message") //Global path (purely optional)
public class MessageRenderController {

	@GetMapping("/generate")
	public ResponseEntity<String> generateMessage(){
		//get system date  & time
		LocalDateTime ldt = LocalDateTime.now();
		//get current hour of the date
		int hour = ldt.getHour();
		String body = null;
		if (hour>12)
			body = "Good Morning";
		else if (hour<16)
			body = "Good Afternoon";
		else if (hour<20)
			body = "Good Evening";
		else
			body = "Good Night";
		HttpStatus status = HttpStatus.OK;
		ResponseEntity<String> entity = new
ResponseEntity<String>(body, status);
		return entity;
	}

}
```

To run using browser (not recommended):

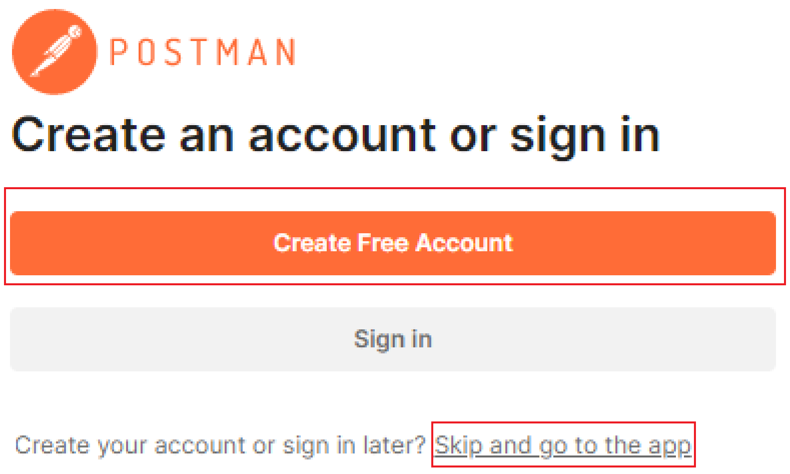
a. Run the application using Run as Server option
b. Use the following URL in the browser http://localhost:3030/BootRestProj01-SimplePOC/message/generate

Using POSTMAN tool (Recommended to use):

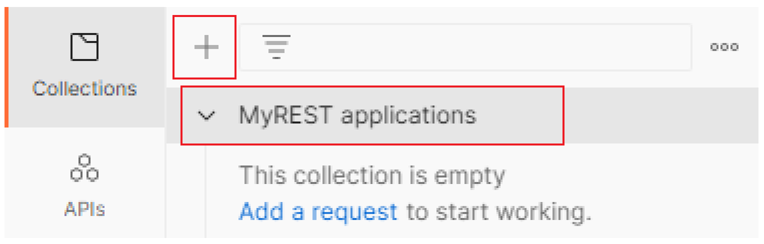
- A good tool to Test the Restful webservices/ service providers developed in any language/ technology/ framework having capability generate different modes of request and ability to receive text/ JSON/ XML content-based response.
- Follow the below step for download, installation and how to use.

Step1: Download the postman software [Click here].

Step 2: For installation just double click on that it will install automatically if you want you can skip the sign in process, but better to do the signup using your google account it will take only some minutes.

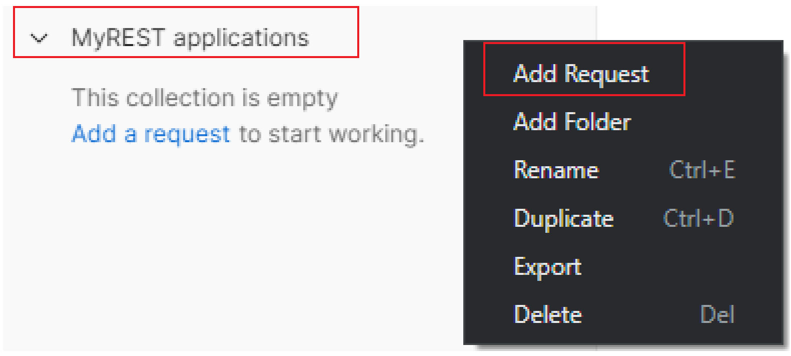

POSTMAN
Create an account or sign in
Create Free Account
Sign in
Create your account or sign in later? Skip and go to the app

Step 3: Create a new Collection, by click on the (+) plus symbol and give a proper name.

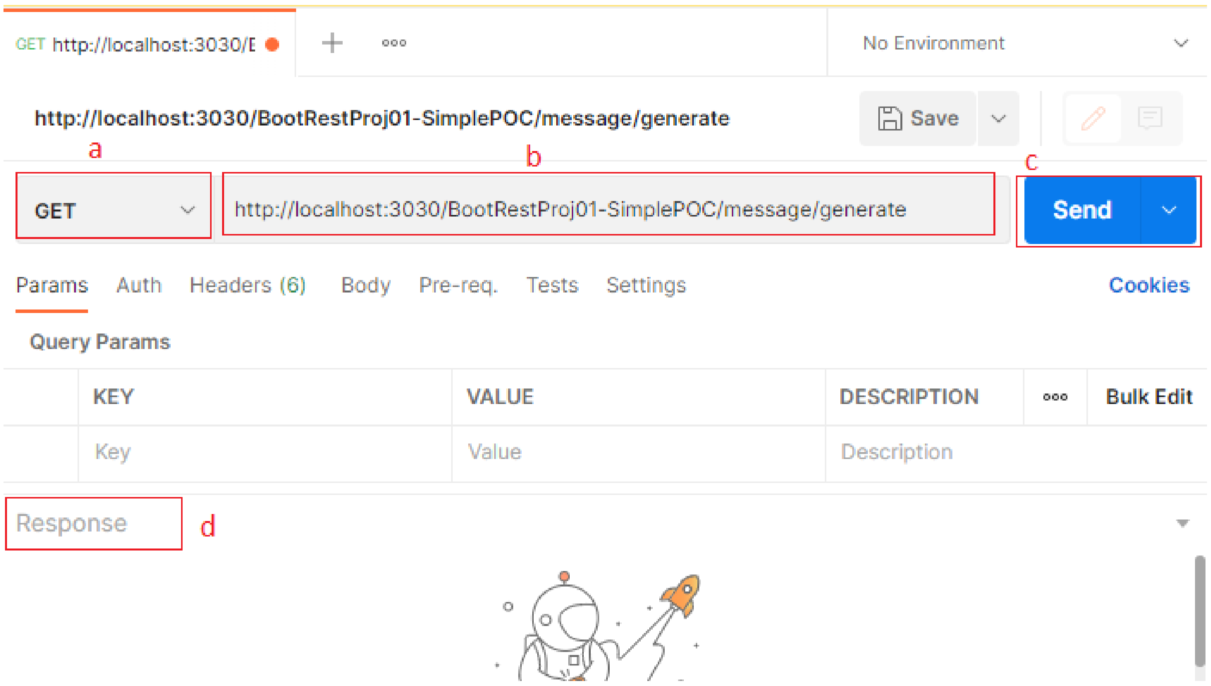


Step 4: Right click on your Collection then click on Add Request to create a request or in Open Scratch Pad Overview there is a (+) plus symbol you can

click on that also.



Step 5: Send request by using the request URL and click on the send button.


GET http://localhost:3030/E
No Environment
http://localhost:3030/BootRestProj01-SimplePOC/message/generate
Save
a
b
c
GET
http://localhost:3030/BootRestProj01-SimplePOC/message/generate
Send
Params
Auth
Headers (6)
Body
Pre-req.
Tests
Settings
Cookies
Query Params
KEY
VALUE
DESCRIPTION
Bulk Edit
Key
Value
Description
Response
d

a. This is request mode.
b. Here you have to pass the request URL.
c. Then you have to click on the send button for request.
d. Then in response section you will get the result.

Note: In this scenario also, the server must be on running mode.

Q. What is the difference between @Controller and @ResstController?
Ans.

@Controller	@RestController
a. Given in the Spring 2.5 version (Bit old annotation).	a. Given in the version Spring 4.0 (relatively new annotation).

b. Specialization of @Component.	b. Specialization of @Controller.
c. Makes the Java class Web controller class having capability to handle HTTP requests by taking them through DispatcherServlet.	c. Makes the Java class Rest Controller or Restful service provider class having capability to handle HTTP requests by taking them through DispatcherServlet.
d. Can be used in both Spring MVC and Spring Rest apps.	d. Recommended to use only in Spring Rest apps (Restful apps).
e. Every Handler method does not get @ResponseBody automatically. So, we need to add it explicitly.	e. Every handle method automatically gets @ReponseBody. So, there is no need of adding it separately.
f. Based on the return type of the handler method it decides whether view component should be involved or not (if the return type is other than ResponseEntity<T> then it involves ViewResolver and view component).	f. Makes the handler method sending its output/ results as response to client directly through DispatcherServlet without involving ViewResolver and view component.

Note: Instead of comparing @RestController with @Controller, please use it as convenient annotation given over @Controller providing easiness to developer Restful Service providers (@Contrller + @ResponseBody).

Http request methods/ modes:

- GET
- POST
- PUT
- DELETE
- OPTIONS
- TRACE
- PATCH
- HEAD
- CONNECT (Reserved for future)

Generally, we use the following HTTP request methods/ modes in Restful applications while performing CURD Operations.

- GET for Read operations (R) selecting records
- POST for Create operations (C) inserting records
- DELETE for DELETE operations (D) deleting records

- PUT for Update operations (U) modifying records
- PATCH for Update operations (U) modifying records

Note: PUT for complete modification of the record. PATCH for partial modification of the record.

```
public <RT> updateEmployee (Employee emp) {
            ……
            ……. use PUT mode request
}
public <RT> updateEmployeeEmail(String newEmail) {
            …….
            ……. Use PATCH mode request
}
```

- Since HEAD request mode HttpResponse does not contain body/ output/ result. So, it cannot be used for read/ select operations.
- Since TRACE is given to trace/ debug the components involved for the SUCCESS or FAILURE of request processing. So, it cannot be used for CURD operations.
- Since OPTIONS mode request gives the possible HTTP request methods/ modes that can be used to generate request to web component, its HttpResponse contains purely list modes/ methods that are possible to give request to web component.

```
@WebServlet("/testurl")
public class TestServlet {

            public void doGet(req, res){
                        …….
                        ……..
            }
}
```

If we give OPTIONS mode request to this web components we get response as,
Allow: GET, HEAD, OPTIONS, TRACE.

```
@WebServlet("/testurl")
public class TestServlet {
            public void doPost(req, res){
                        …….
                        …….
            }
}
```

If we give OPTIONS mode request to this web components we get response as,
Allow: POST, OPTIONS, TRACE.

- If the request UTL is not valid then we get 404 error (requested resource not found).
- If the request URL based web component is not ready to process given mode request, then we get 405 error (method/ mode not allowed).
- If the request fails in Authentication, then we get 401 error.
- If the request fails in Authorization, then we get 403 error (resource is forbidden).
- If the web component (Servlet/ JSP/ Producer) fails to instantiate for the given request then we get 500 error.
- If the return type of producer method is other than <String> generic in ResponseEntity object then the producer methods send JSON data along with the HttpResponse body.
- If the return type of producer method is <String> in ResponseEntity object then the producer methods send text data along with the HttpResponse body.

Every RESTful application/ project contains:

a. Server app/ producer app/ service provider app
   - Spring MVC application with @RestController with methods, also called as Rest API.
   - The request URL of @RestController and other required information for sending request are called end points.

b. Client app/ service consumer/ consumer app
   - Angular, ReactJS, PHP, IOS, IOT Devices, .Net, Python, Android app are programable clients.
   - POSTMAN tool for testing.
   - Swagger for testing and RESTful documentation.
   - Spring Rest Template is programmable Spring based client app.

Request Header and Request Parameters:

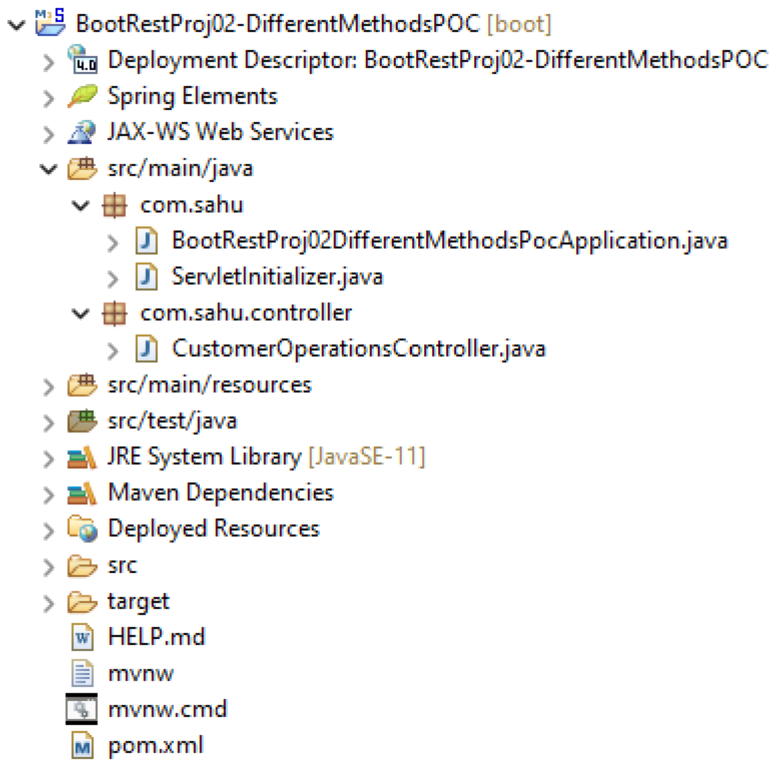
✓ Request headers are client generated inputs that go along with request automatically having fixed names (header names).
  E.g., accept, accept-language, user-agent, referrer, contentType and etc.

✓ Request parameters/ params are end user supplied values as query String/ form data. Request param names are not fixed and they are user-defined.
  E.g., sno=101&sname=raja&sadd=hyd
  sno, sname, sadd are request params or query params.

Note:

- ✓ API development means Developing Spring Rest Server app/ Service provider app.
- ✓ Giving API end points means providing request URL and other related information to developers for developing client apps/ service consumer app for consuming the services offered by service provider.
- ✓ The API/ services developed in SOAP based webservices cannot consumed using Rest client and vice-versa.
- ✓ The Restful webservices developed using one kind of Rest API/ framework (like Jersey, Spring Rest, JAX-RS, Restlet and etc.) can be consumed using same Rest API or different Rest APIs because both are in Restful webservice environment.
- ✓ There are multiple open/ public/ free APIs/ service providers developed in different technologies/ frameworks of Restful webservices and they consumed in our applications using our choice Rest APIs.
  E.g., Weather report API, Google Maps API, GPay API, ICC API, Covid API.

Directory Structure of BootRestProj02-DifferentMethodsPOC:



```
BootRestProj02-DifferentMethodsPOC [boot]
  > Deployment Descriptor: BootRestProj02-DifferentMethodsPOC
  > Spring Elements
  > JAX-WS Web Services
  v src/main/java
    v com.sahu
      > BootRestProj02DifferentMethodsPocApplication.java
      > ServletInitializer.java
    v com.sahu.controller
      > CustomerOperationsController.java
  > src/main/resources
  > src/test/java
  > JRE System Library [JavaSE-11]
  > Maven Dependencies
  > Deployed Resources
  > src
  > target
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use Spring Web starter during project creation.
- Then place the following code with in their respective files.

CustomerOperationsController.java

```
package com.sahu.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/customer")
public class CustomerOperationsController {


	@GetMapping("/report")
	public ResponseEntity<String> showCustomersReport() {
		return new ResponseEntity<String>("From GET-ShowReport 
method", HttpStatus.OK);
	}

	@PostMapping("/register")
	public ResponseEntity<String> registerCustomer() {
		return new ResponseEntity<String>("From POST-
registerCustomer method", HttpStatus.OK);
	}

	@PutMapping("/modify")
	public ResponseEntity<String> updateCustomer() {
		return new ResponseEntity<String>("From PUT-
updateCustomer method", HttpStatus.OK);
	}

	@PatchMapping("/pmodify")
	public ResponseEntity<String> updateCustomerByNo() {
		return new ResponseEntity<String>("From PATCH-
updateCustomerByNo method", HttpStatus.OK);
```

```
	}

	@DeleteMapping("/delete")
	public ResponseEntity<String> deleteCustomer() {
		return new ResponseEntity<String>("From DELETE-
deleteCustomer method", HttpStatus.OK);
	}

}
```

Give the following requests in POSTMAN:

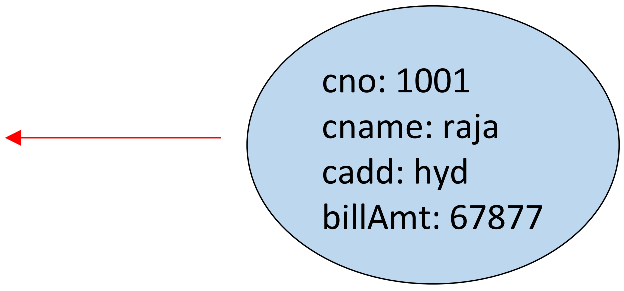
- GET http://localhost:3030/BootRestProj02-DifferentMethodsPOC/customer/report
- POST http://localhost:3030/BootRestProj02-DifferentMethodsPOC/customer/register
- PUT http://localhost:3030/BootRestProj02-DifferentMethodsPOC/customer/modify
- PATCH http://localhost:3030/BootRestProj02-DifferentMethodsPOC/customer/pmodify
- DELETE http://localhost:3030/BootRestProj02-DifferentMethodsPOC/customer/delete

Working with JSON data as HTTP Response body

- Nothing but sending JSON data as HTTP Response body to client from service provider app.
- If the method of Service Provider is returns ResponseEntity object having other than <String> generic (can be any other object) then given object will be converted to JSON data and will be placed in HttpResponse as body automatically.
- JSON: Java Script Object Notation.
- It is a way of representing object data using key: value format.
- Key must be in " " and value can be anything. If the value is string content, then it also should be in " ".
- One {} (flower bracket) represents one object or sub object represents array/ list/ set element values.
- In JSON array/ list/ set collection will be treated as array only, so their elements will be represented using [-, -, -].
- In JSON array/ list/ set collection are called 1D arrays.
- In JSON map collection is called 2D array.

- In JSON map collection elements and HAS-A property elements will be represented using sub object/ node {"key": value, "key": value, ...}.
  E.g.,

```
  Customer cust = new Customer (1001, "raja", "hyd", 67877);
  In JSON:
  {
          "cno": 1001,
          "cname": "raja",
          "cadd": "hyd",
          "billAmt": 67877
  }
```

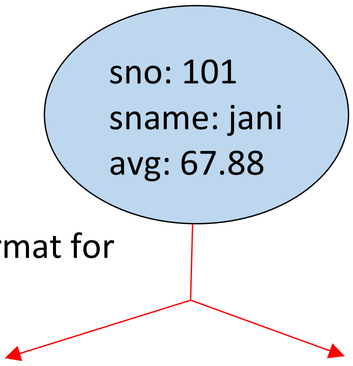


- HTML (Hyper Text Markup Language) is given to display data on browser by applying styles.
- JSON/ XML are given to describe data (to construct data having structure).

Q. What is the difference between JSON and XML?
Ans.

```
Student st = new Student (101, "jani", 67.88);
```

(Student object)



Both are global format for
Representing data

JSON way of representing data:

```
{
        "sno": 101,
        "sname": "jani"
        "avg": 67.88
}
```

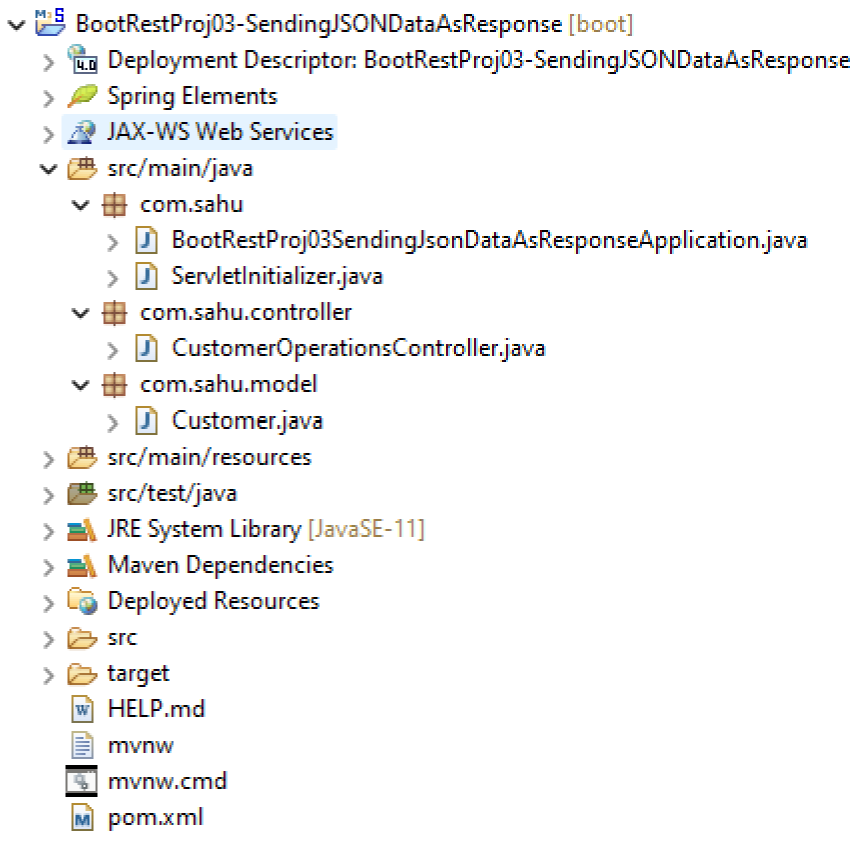
XML way of representing data:

```
.... //Schema or DTD import (optional)
<Student>
        <sno>101</sno>
        <sname>jani</sname>
        <avg>67.88</avg>
</Student>
```

Note: YML, MongoDB docs are inspired from JSON.

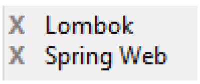
JSON	XML
a. It is Java Script Object Notation.	a. It is eXtensible Markup Language.
b. It is the way of representing object data.	b. It is way constructing/ describing data using tags.
c. Maintains the data as "key": "value" pairs {"key": "value", ….}.	c. Maintains the data using tags as the tags body and attributes {<> </>}.
d. Does not provide namespaces to validate the data.	d. Provides namespaces to validate data (namespace is a library that contains set tags, attributes and rules to construct XML data).
e. This format is given by Java Script language (Sun MS + Netscape).	e. This is given according to w3c recommendations and inspired from SGML (Standard Generalized Markup Language).
f. JSON files are easy to read and process.	f. XML files are bit complex to read and process.
g. To handle JSON data we use the support of JACSON API.	g. To handler XML data we take the support of XML APIs like JAXP (Java API for XML Processing). JAXP deals with SAX, DOM, JDOM APIs and etc.
h. To covert JSON data to object and vice-versa we can use GSON, JSON-B and etc. API.	h. To convert XML doc to objects and vice-versa we use JAXB API (Java API for XML Binding).
i. The process of converting object to JSON data is called serialization and reverse is called Deserialization.	i. The process of converting object to XML data is called marshalling and reverse is called unmarshalling.
j. JSON format/ files are light weight compare to XML format/ files.	j. XML format/ files are bit heavy weight compare to JSON format/ files.
k. Very much used is Restful webservices as alternate to XML to send and receive data.	k. Very much used in SOAP based webservices to send and receive data (SOAP message are XML messages).
l. JSON is less structured compare to XML	l. XML is more structured.
m. Does not support commenting	m. Supports commenting.
n. Less secured because it just contains key and values.	n. More secured because data is having hierarchy structure and namespaces protected.
o. Allows only UTF-8 Characters.	o. Allows lots of Charsets including UTF-8.

Directory Structure of BootRestProj03-SendingJSONDataAsResponse:



```
BootRestProj03-SendingJSONDataAsResponse [boot]
  Deployment Descriptor: BootRestProj03-SendingJSONDataAsResponse
  Spring Elements
  JAX-WS Web Services
  src/main/java
    com.sahu
      BootRestProj03SendingJsonDataAsResponseApplication.java
      ServletInitializer.java
    com.sahu.controller
      CustomerOperationsController.java
    com.sahu.model
      Customer.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  Deployed Resources
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starter during project creation.



X Lombok
X Spring Web

- Then place the following code with in their respective files.

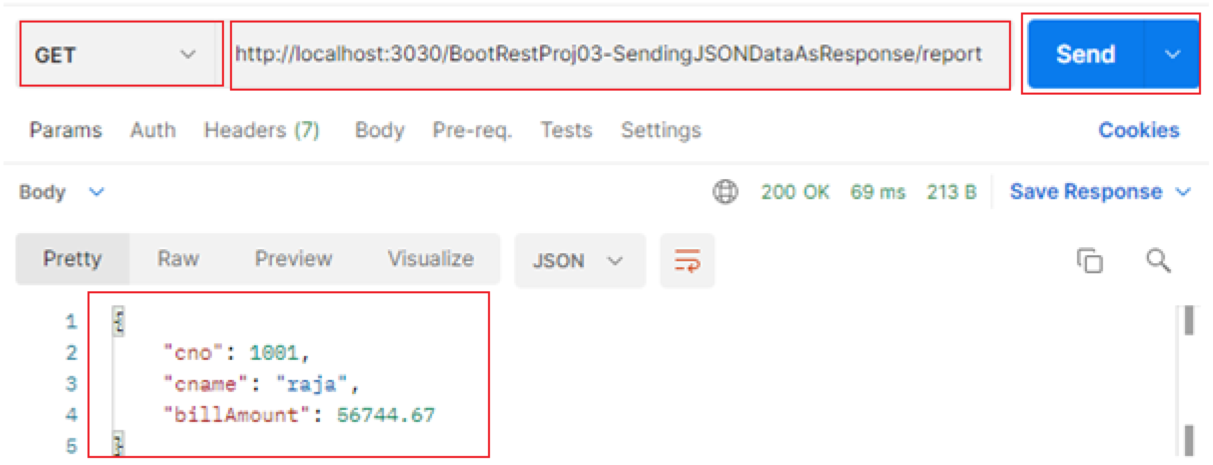
Customer.java

```java
package com.sahu.model;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Customer {
	private Integer cno;
	private String cname;
	private Float billAmount;
}
```

CustomerOperationsController.java

```java
package com.sahu.controller;

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

import com.sahu.model.Customer;

@RestController
public class CustomerOperationsController {

	@GetMapping("/report")
	public ResponseEntity<Customer> showData() {
		Customer cust = new Customer(1001, "raja", 56744.67f);
		return new ResponseEntity<Customer>(cust, HttpStatus.OK);
	}

}
```

Run the application and give the request in URL then you will get JSON response.
http://localhost:3030/BootRestProj03-SendingJSONDataAsResponse/report



```
GET http://localhost:3030/BootRestProj03-SendingJSONDataAsResponse/report
200 OK   69 ms   213 B
{
    "cno": 1001,
    "cname": "raja",
    "billAmount": 56744.67
}
```

To pass complex data:

- Create a Company.java class under com.sahu.model package.
- And place the following code with their respective files.

Company.java

```
package com.sahu.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Company {
	private String name;
	private String address;
	private String type;
	private Integer size;
}
```

Customer.java

```
package com.sahu.model;

import java.util.List;
import java.util.Map;
import java.util.Set;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Customer {
	private Integer cno;
	private String cname;
	private Float billAmount;
	private String[] favColors;
	private List<String> studies;
	private Set<Long> phoneNumbers;
	private Map<String, Object> idDetails;
	private Company company;
}
```

Company.java

```
@RestController
public class CustomerOperationsController {

	@GetMapping("/report")
	public ResponseEntity<Customer> showData() {
		Customer cust = new Customer(1001, "raja", 56744.67f,
						new String[] {"blue", "red", "yellow"},
						List.of("10th", "10+2", "BCA"),
						Set.of(56445353433l, 354534343343l, 
3434355333l),
						Map.of("aadhar",5454252333l, 
"panNo", 5452454262l),
						new Company("SAMSUNG","hyd", 
"Electronics", 4000));
		return new ResponseEntity<Customer>(cust, HttpStatus.OK);
	}

}
```

from POSTMAN:
GET http://localhost:3030/BootRestProj03-SendingJSONDataAsResponse/report

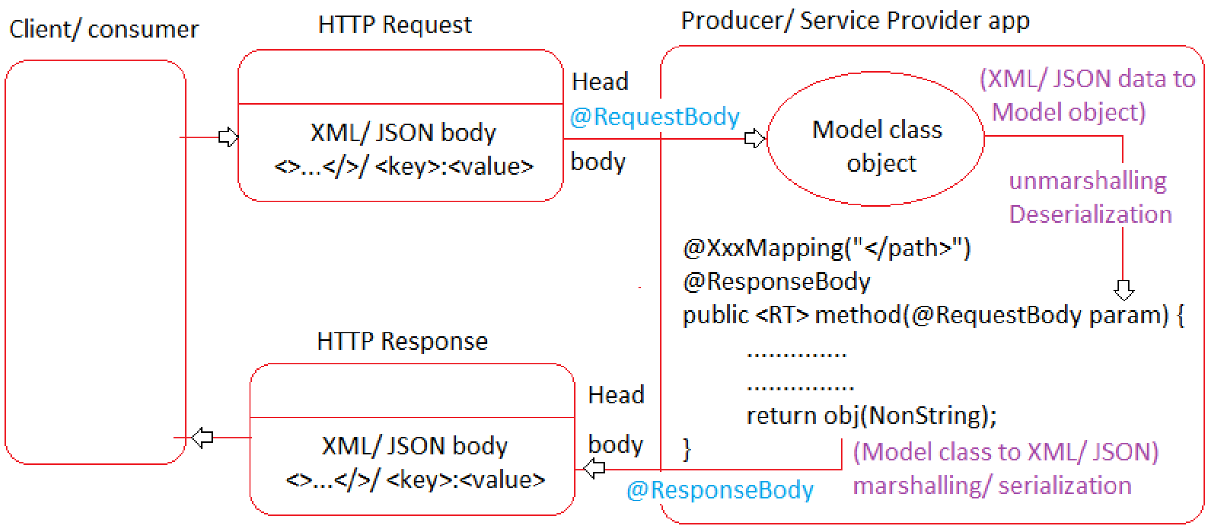
Response Body

```
{
   "cno": 1001,
   "cname": "raja",
   "billAmount": 56744.67,
   "favColors": [
      "blue",
      "red",
      "yellow"
   ],
   "studies": [
        "10th",
        "10+2",
        "BCA"
    ],
    "phoneNumbers": [
        56445353433,
        354534343343,
        3434355333
    ],
    "idDetails": {
        "panNo": 5452454262,
        "aadhar": 5454252333
    },
    "company": {
        "name": "SAMSUNG",
        "address": "hyd",
        "type": "Electronics",
        "size": 4000
    }
}
```

Working with Media Type Annotations

- Working with Media Type Annotations is nothing but working with @RequestBody and @ResponseBody.
- Using @RequestBody for the service provider/ producer method param we can convert given XML or JSON data into given Model class (Java bean class object).
- Using @ResponseBody on top of the service provider/ producer method we can convert non-String content into XML or JSON data and can send to client as response body.

- @ResponseBody will be applied automatically on every method of @RestController class.



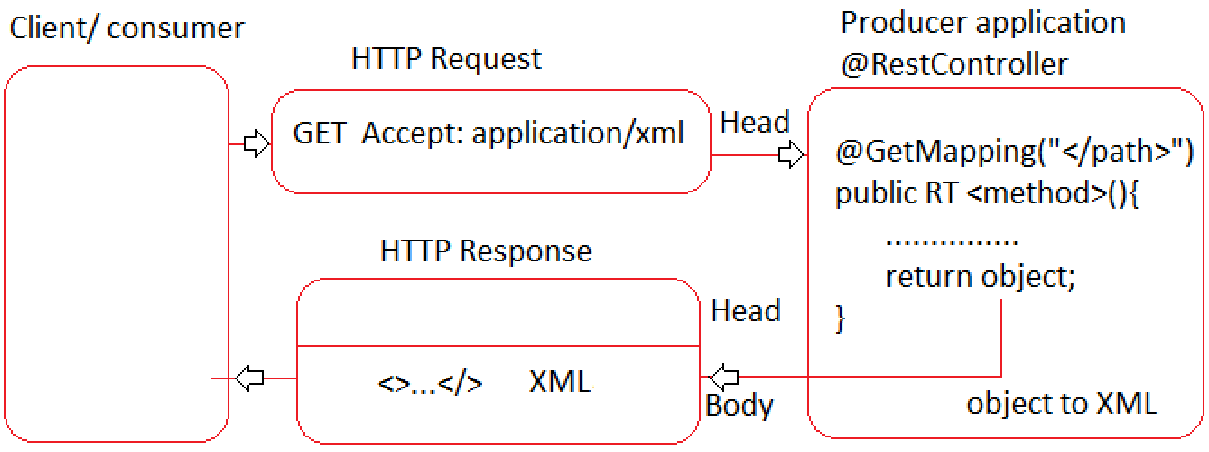
Making Spring Rest Producer app sending XML Response body to Client app

- Based on given request type, making producer app sending different type of content with content type is technically called Content negotiation.
- For this we need to use "Accept" request header specifying content type like "JSON", "XML" and etc. This makes producer application given content in response in the requested format.
- Though "Accept" is request header, it makes the producer/ server/ service provider app sending its content in required format like JSON, XML and etc.
- By default, every method in Producer app sends plain text response if the return type is ResponseEntity<String> otherwise for ResponseEntity<non-String> it sends JSON response.

Making Producer application sending XML response:

- For this the client app/ consume app must set "Accept" request header to "application/xml" (default is */*).
- Accept: application/xml tells to producer app give me response content as XML content
- Accept: */* tells to producer apps given me response in any format (what you want, that you can give).
- We need to add the following additional dependency (jar file) to the project for converting object data to XML format.

- [Jackson Data format XML]



Procedure:

Step 1: Create Spring Boot starter project having following starters, dependencies

- Spring Web
- Lombok
- [Jackson Data format XML]

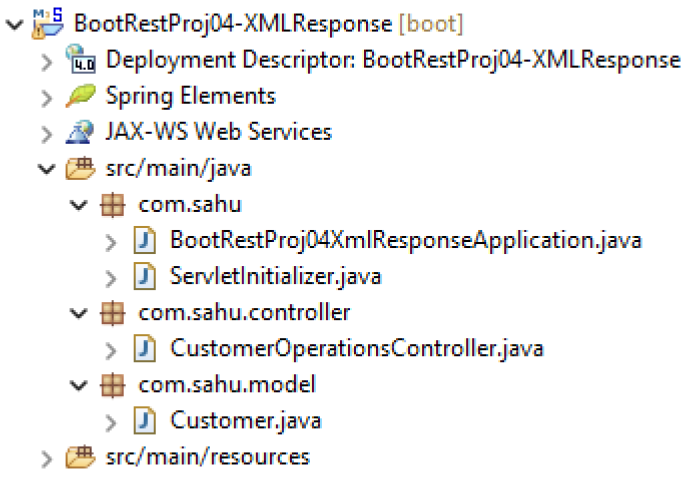
Step 2: Develop the Model class.

Step 3: Develop producer as @RestController having method with other then String return type.

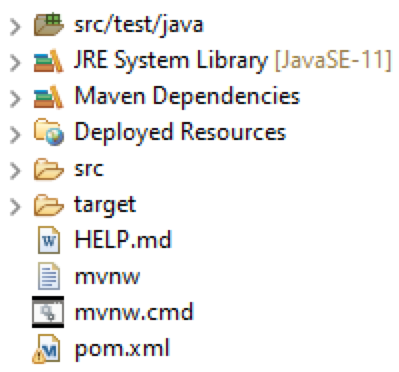
Step 4: Run the application, as Run-on Server.

Step 5: Give Request from POSTMAN Tool by setting "Accept" header with the value "application/xml".

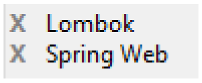
Directory Structure of BootRestProj04-XMLResponse:



```
BootRestProj04-XMLResponse [boot]
  Deployment Descriptor: BootRestProj04-XMLResponse
  Spring Elements
  JAX-WS Web Services
  src/main/java
    com.sahu
      BootRestProj04XmlResponseApplication.java
      ServletInitializer.java
    com.sahu.controller
      CustomerOperationsController.java
    com.sahu.model
      Customer.java
  src/main/resources
```


```
> src/test/java
> JRE System Library [JavaSE-11]
> Maven Dependencies
> Deployed Resources
> src
> target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starter and dependency during project creation.

  
  X Lombok
  X Spring Web

  [Jackson Data format XML]
- Then place the following code with in their respective files.

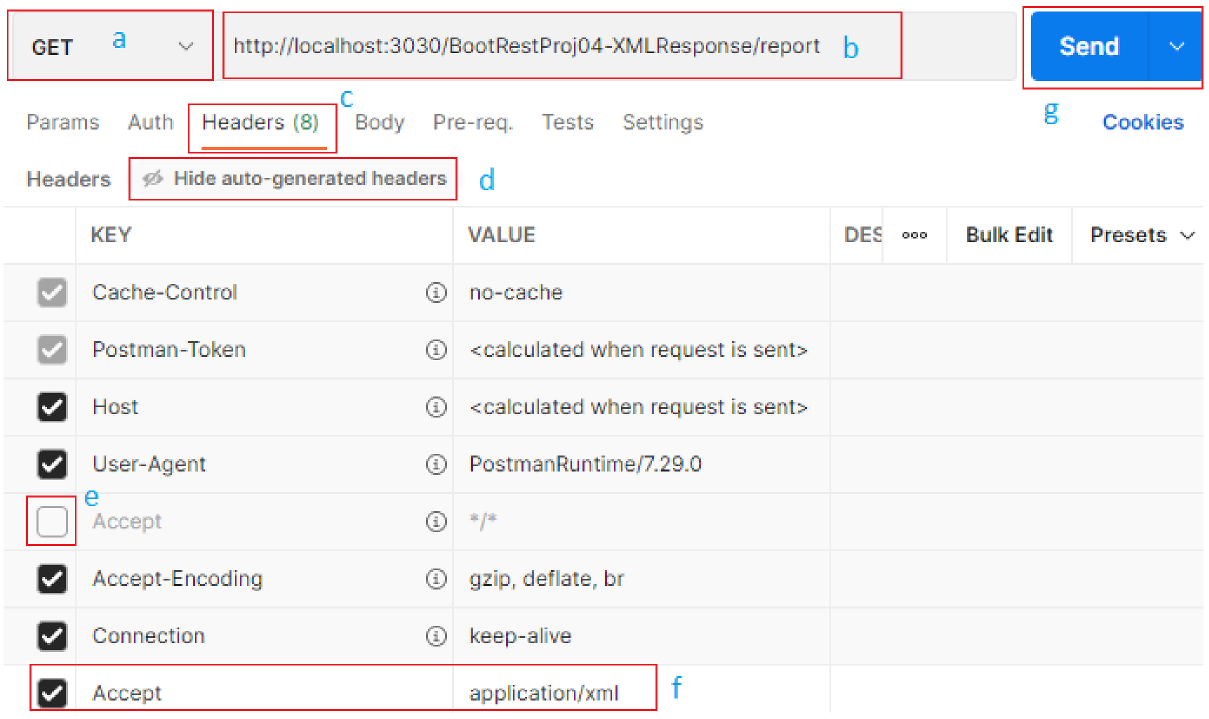
Customer.java

```
package com.sahu.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class Customer {
	private Integer cno;
	private String cname;
	private String cadd;
	private Float billAmount;
}
```

CustomerOperationsController.java

```
package com.sahu.controller;

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

```java
import com.sahu.model.Customer;

@RestController
public class CustomerOperationsController {

	@GetMapping("/report")
	public Customer showDate( ) {
		Customer cust = new Customer(1001, "raja", "hyd", 34353.5f);
		return cust;
	}

}
```

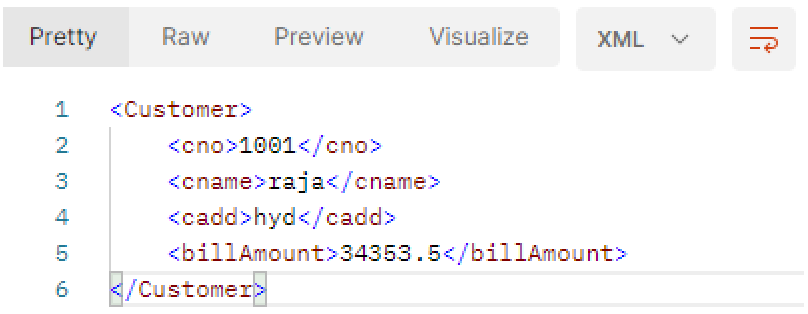


GET http://localhost:3030/BootRestProj04-XMLResponse/report

Params | Auth | Headers (8) | Body | Pre-req. | Tests | Settings

	KEY	VALUE
✓	Cache-Control	no-cache
✓	Postman-Token	<calculated when request is sent>
✓	Host	<calculated when request is sent>
✓	User-Agent	PostmanRuntime/7.29.0
☐	Accept	*/*
✓	Accept-Encoding	gzip, deflate, br
✓	Connection	keep-alive
✓	Accept	application/xml

Follow the above screen to give the request then you will get the following output.

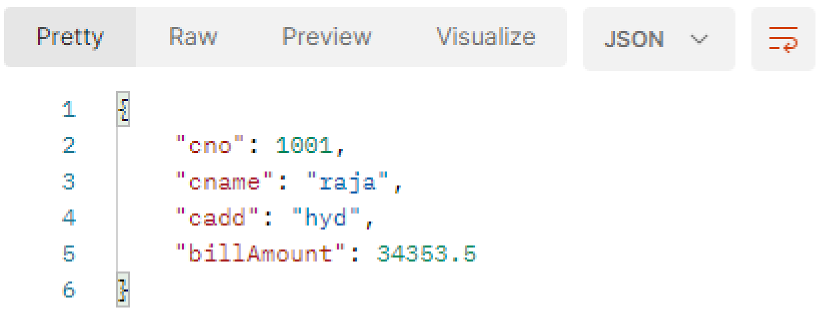


Pretty | Raw | Preview | Visualize | XML

```xml
<Customer>
    <cno>1001</cno>
    <cname>raja</cname>
    <cadd>hyd</cadd>
    <billAmount>34353.5</billAmount>
</Customer>
```

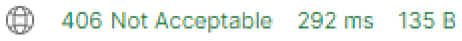
Note:

- ✓ Taking ResponseEntity<T> as return is recommended because it gives control on response status code.
- ✓ If we send the above request without placing Accept: application/xml request header then Accept: */* will be taken as default value and we get JSON response as show below.



```
{
    "cno": 1001,
    "cname": "raja",
    "cadd": "hyd",
    "billAmount": 34353.5
}
```

- ✓ In the above setup where we asking for XML response using "Accept: application/xml" request header, if we are not placing "Jackson-dataformat-xml" then we get 406 (Media Type Not Acceptable) error.

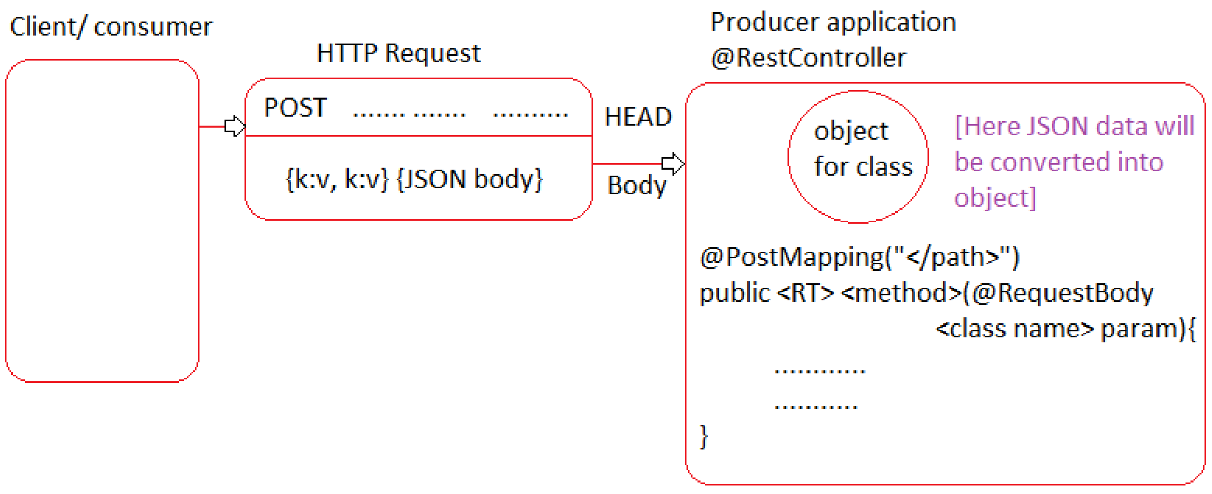


406 Not Acceptable 292 ms 135 B

- ✓ We run the same application with the above setup (not adding Jackson-dataformat-xml jar file) and without changing "Accept" header value to "application/xml" then we get JSON response from producer app because the method return type is non-string.

Working with @RequestBody

- @RequestBody is used to convert given XML/ JSON content into Model class object.
- If client generated HTTP request is having XML/ JSON content as the body and if you want to construct Java class object having that data then we need to specify @RequestBody for the parameter of Producer method whose type is Model class name/ Java class name.
- GET, HEAD mode request does not contain body part but they send little amount of data as request param names and values in query string.
- The response of GET mode request contains both HEAD, BODY part where the response of HEAD mode request does not contain BODY part, so we use GET mode request to get data from server and HEAD mode request to check whether certain web component is available or not (for debugging purpose).
- We can not to use GET mode here because GET mode request does contain body.



Syntax to use @RequestBody in method param:
@RtuestBody <Class Name> <param Name>
becomes object name internally

- If JSON body is coming as {} (empty curly braces) then the object for class given in @RequestBody will be created using 0-param constructor with default values given by JVM (0/0.0/ null/false).
- The keys in JSON body and the property names in the give class of @RequestBody must match in order to see Data binding to object. If few keys and property names are matching and others are not matching then binding will happen only for few/ matching properties and the remaining properties will hold default values.

Example Application:
Step 1: Create Spring Boot Project adding the following starters, dependency.
- Spring Web
- Lombok
- [Jackson Data format XML]

Step 2: Develop the Model class.

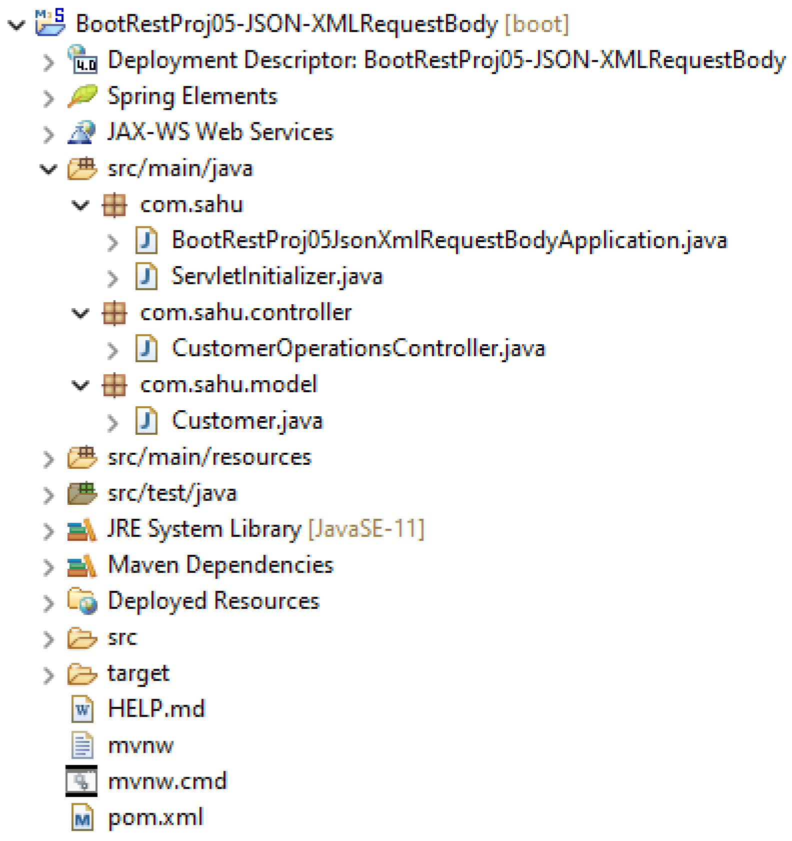
Note: @RequestBody of Spring Rest is very much similar to @ModelAttribute of Spring MVC.

Step 3: Develop producer application as @RestcController.

Step 4: Run the Application

Step 5: Send the following request from POSTMAN

Directory Structure of BootRestProj05-JSON-XMLRequestBody:



```
BootRestProj05-JSON-XMLRequestBody [boot]
  Deployment Descriptor: BootRestProj05-JSON-XMLRequestBody
  Spring Elements
  JAX-WS Web Services
  src/main/java
    com.sahu
      BootRestProj05JsonXmlRequestBodyApplication.java
      ServletInitializer.java
    com.sahu.controller
      CustomerOperationsController.java
    com.sahu.model
      Customer.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  Deployed Resources
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starter and dependency during project creation.

  X Lombok
  X Spring Web

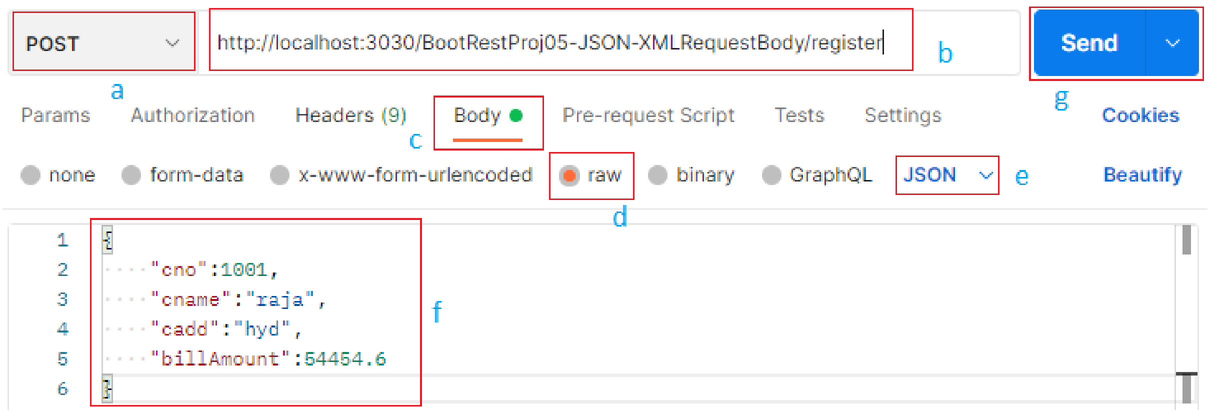
  [Jackson Data format XML]
- Then place the following code with in their respective files.

Customer.java

```
package com.sahu.model;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
@Data
public class Customer {
	private Integer cno;
	private String cname;
	private String cadd;
	private Float billAmount;
}
```

CustomerOperationsController.java

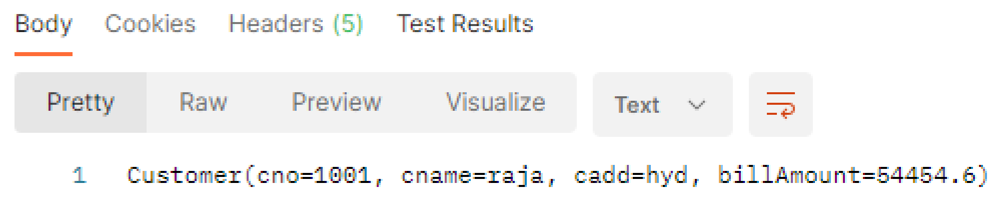
```java
package com.sahu.controller;

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

import com.sahu.model.Customer;

@RestController
public class CustomerOperationsController {

	@PostMapping("/register")
	public String registerCustomer(@RequestBody Customer customer) {
		return customer.toString();
	}

}
```



http://localhost:3030/BootRestProj05-JSON-XMLRequestBody/register

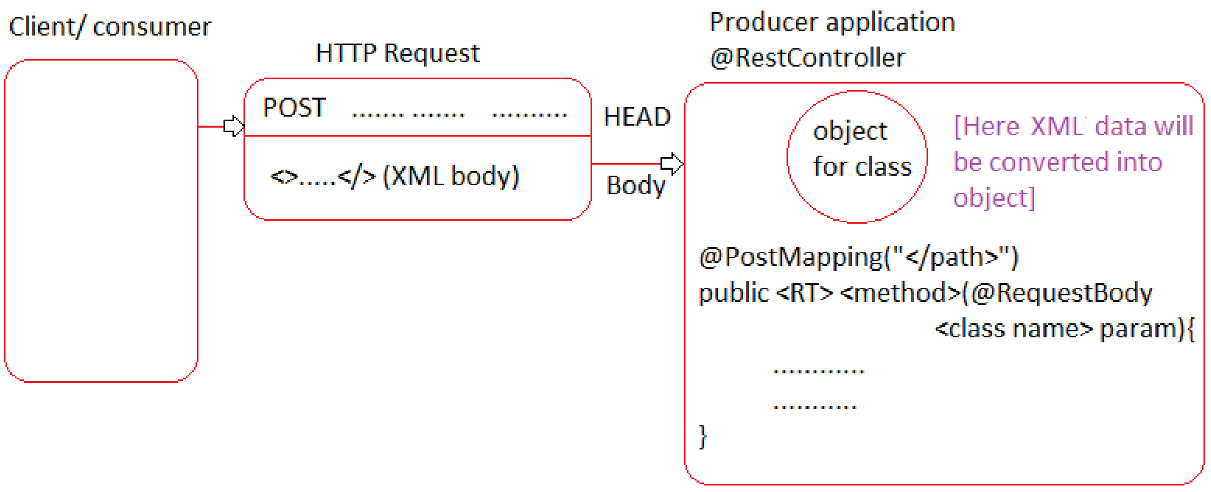
```
{
    "cno":1001,
    "cname":"raja",
    "cadd":"hyd",
    "billAmount":54454.6
}
```

Give the request through PSOTMAN by following the above screen then we will get below output.

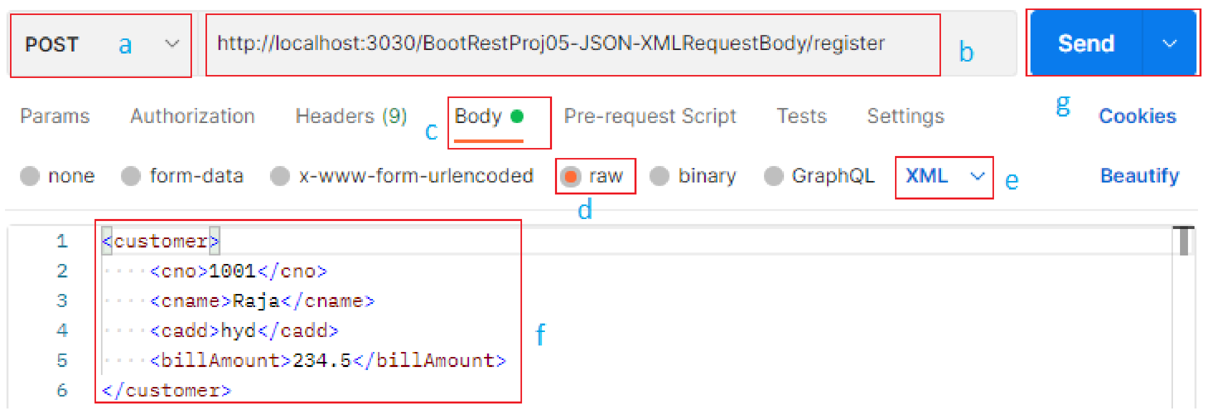


```
Customer(cno=1001, cname=raja, cadd=hyd, billAmount=54454.6)
```

Note: After taking request body as JSON/ XML and if we do not correct format body content along with the HttpRequest then we get 400 error response (Bad Request).

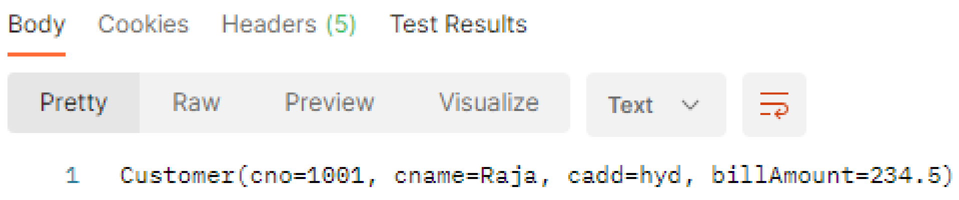


For test give the request through POSTMAN by following the below screen.



```
<customer>
    <cno>1001</cno>
    <cname>Raja</cname>
    <cadd>hyd</cadd>
    <billAmount>234.5</billAmount>
</customer>
```

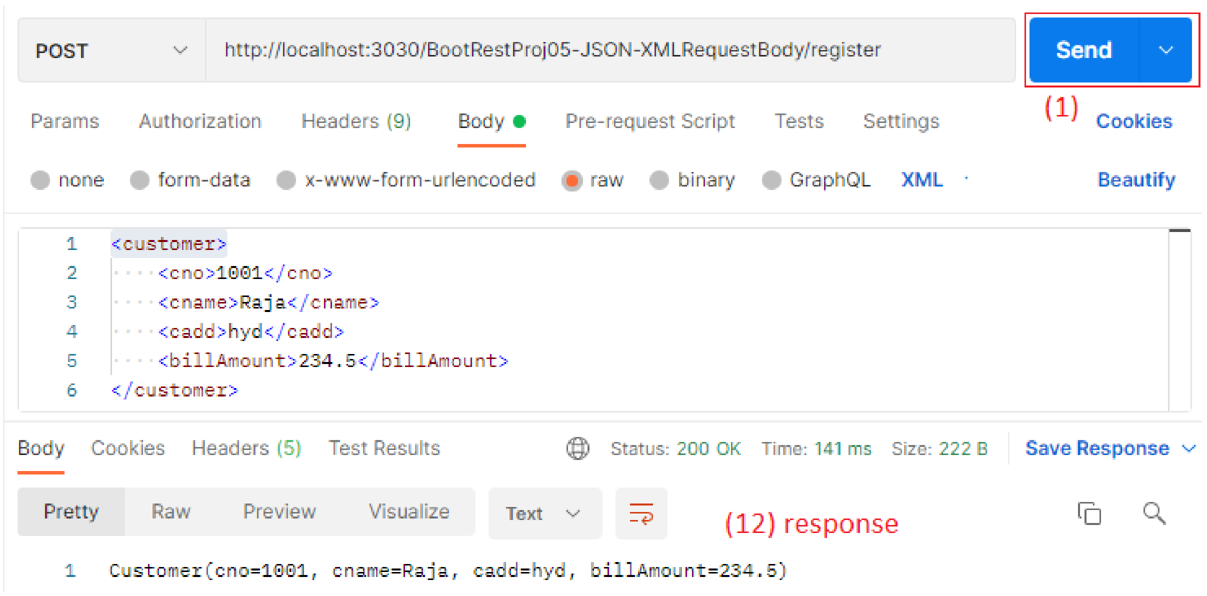
Output:



```
Customer(cno=1001, cname=Raja, cadd=hyd, billAmount=234.5)
```

Flow of Execution:

```
@RestController
public class CustomerOperationsController {          (4?)

        @PostMapping("/register") (9)
        public String registerCustomer(@RequestBody Customer customer) {
                return customer.toString(); (10)
        }
}
```

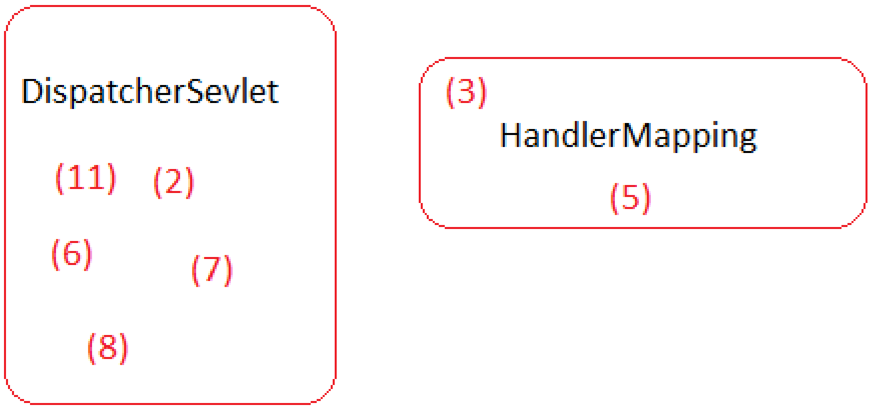


http://localhost:3030/BootRestProj05-JSON-XMLRequestBody/register

```
<customer>
    <cno>1001</cno>
    <cname>Raja</cname>
    <cadd>hyd</cadd>
    <billAmount>234.5</billAmount>
</customer>
```

Status: 200 OK Time: 141 ms Size: 222 B

```
Customer(cno=1001, cname=Raja, cadd=hyd, billAmount=234.5)
```



1. When we click on Send button the HttpRequest with body is generated.
2. Dispatcher Servlet traps the request.
3. DS handover the request to HandlerMapping.
4. Search for Post mapping method with/ register as request path.
5. Handler Mapping gets registerCustomer(-) signature + @RestController bean id.
6. DS gets details from HandlerMapping.
7. Based on @RequestBody param it users different API to convert XML content of request body to Customer class object data
8. Invokes the method having Customer object as the argument.
9. Handler method is executed.
10. It will return string content.
11. The result will come to DS.
12. Then we can see the content on the response body section in POSTMAN.

Note: If the Request body content type is XML and the producer is not having Jackson-data formatter-xml jar file in the build path the we get 415 error.

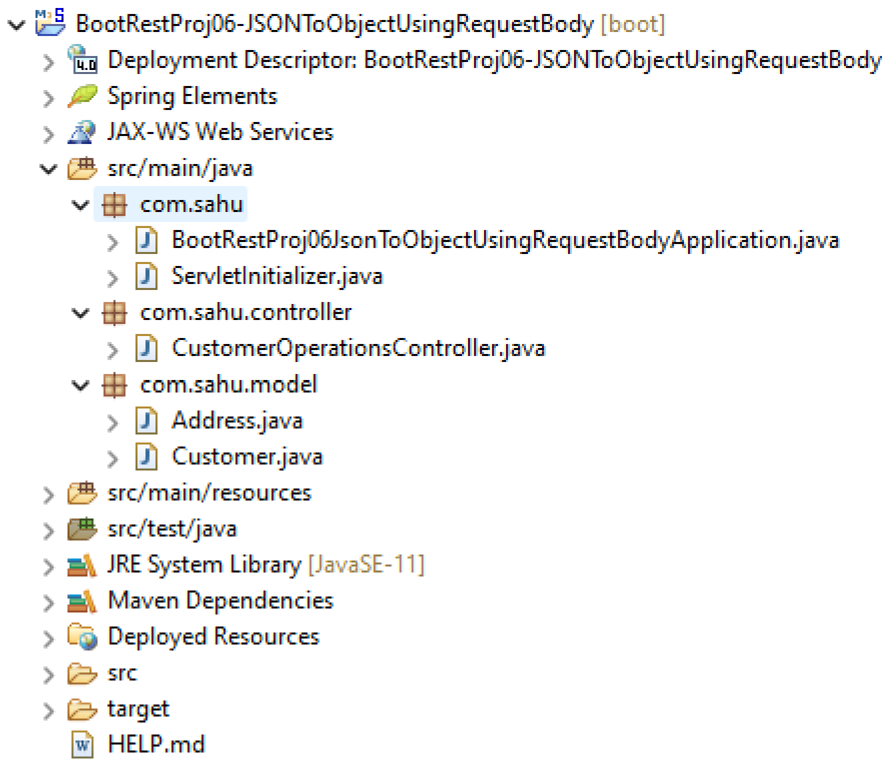
Conclusion:

- 406: We want to get XML response through "Accept: application/xml" but Jackson-data formatter-xml jar is not added in the producer application (related @ResponseBody).
- 415: We want to send XML body along with request to store into object using @RequestBody but Jackson-data formatter-xml jar is not added in the producer application.

Converting JSON body of Http request to Java class object using @RequestBody

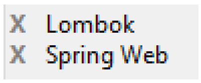
- @ReqeustBody and @ResponseBody annotations are called Media type annotations because they are useful to decide media of the incoming and outgoing content.
- @RequestBody is useful to convert JSON/ XML data to Java class object and @ResponseBody is useful to convert non-String data to JSON/ XML data.
- @RequestBody dealing with 1D and 2D arrays of JSON
  1D array: Array/ List/ Set collections - "<variable>": [<val1>, <val2>, ….]
  2D array: Map - "<variable>": {"<key1>":<<val1>, "<key1>":<val2>, ….}
  2D Array: HAS-A property - "<variable>": {"<subProp1>":<<val1>, "<subProp2>":<val2>, ….}

Directory Structure of BootRestProj06-JSONToObjectUsingRequestBody:



```
BootRestProj06-JSONToObjectUsingRequestBody [boot]
  Deployment Descriptor: BootRestProj06-JSONToObjectUsingRequestBody
  Spring Elements
  JAX-WS Web Services
  src/main/java
    com.sahu
      BootRestProj06JsonToObjectUsingRequestBodyApplication.java
      ServletInitializer.java
    com.sahu.controller
      CustomerOperationsController.java
    com.sahu.model
      Address.java
      Customer.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  Deployed Resources
  src
  target
  HELP.md
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starters during project creation.



X Lombok
X Spring Web

- Then place the following code with in their respective files.

Customer.java

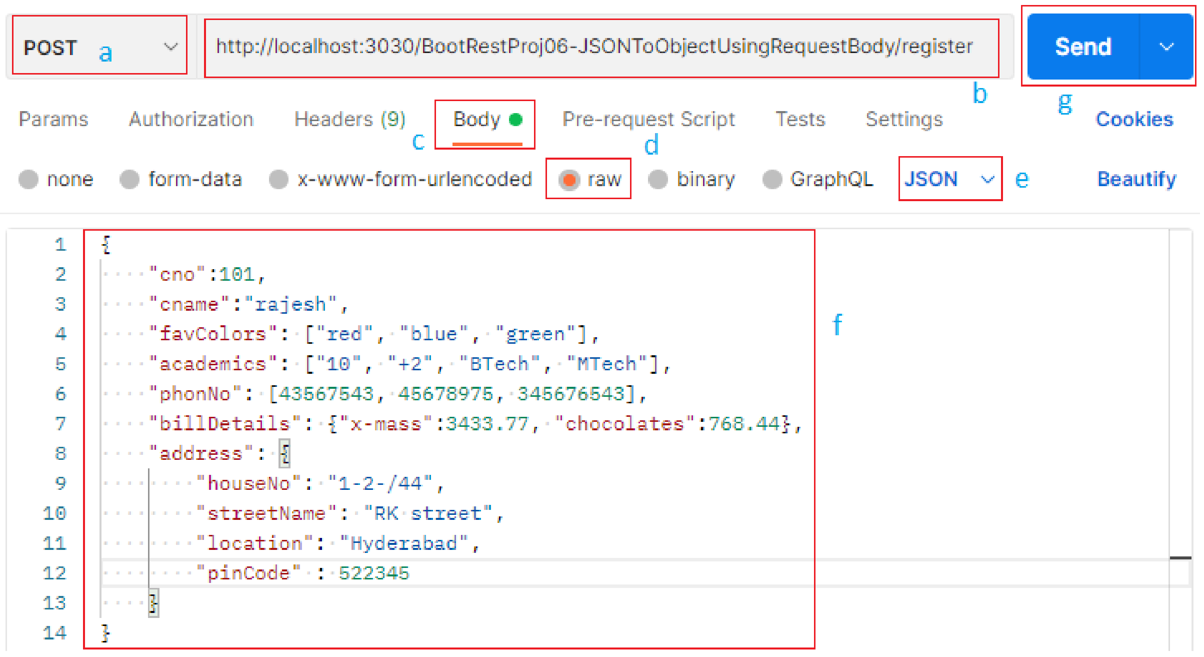
```
package com.sahu.model;

import java.util.List;
import java.util.Map;
import java.util.Set;

import lombok.Data;

@Data
public class Customer {
	private Integer cno;
	private String cname;
	private String[] favColors;
	private List<String> academics;
	private Set<Long> phonNo;
	private Map<String, Double> billDetails;
	private Address address;
}
```

Address.java

```
package com.sahu.model;

import lombok.Data;

@Data
public class Address {
	private String houseNo;
	private String streetName;
	private String location;
	private Integer pinCode;
}
```

CustomerOperationsController.java

```java
package com.sahu.controller;

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

import com.sahu.model.Customer;

@RestController
public class CustomerOperationsController {

	@PostMapping("/register")
	public ResponseEntity<String> saveCustomer(@RequestBody 
Customer customer) {
		return new ResponseEntity<String>(customer.toString(), 
HttpStatus.OK);
	}

}
```



```json
{
    "cno":101,
    "cname":"rajesh",
    "favColors": ["red", "blue", "green"],
    "academics": ["10", "+2", "BTech", "MTech"],
    "phonNo": [43567543, 45678975, 345676543],
    "billDetails": {"x-mass":3433.77, "chocolates":768.44},
    "address": {
        "houseNo": "1-2-/44",
        "streetName": "RK street",
        "location": "Hyderabad",
        "pinCode" : 522345
    }
}
```

Follow the above screen to give the request through the POSTMAN and you will get the output in response body.

Request Body

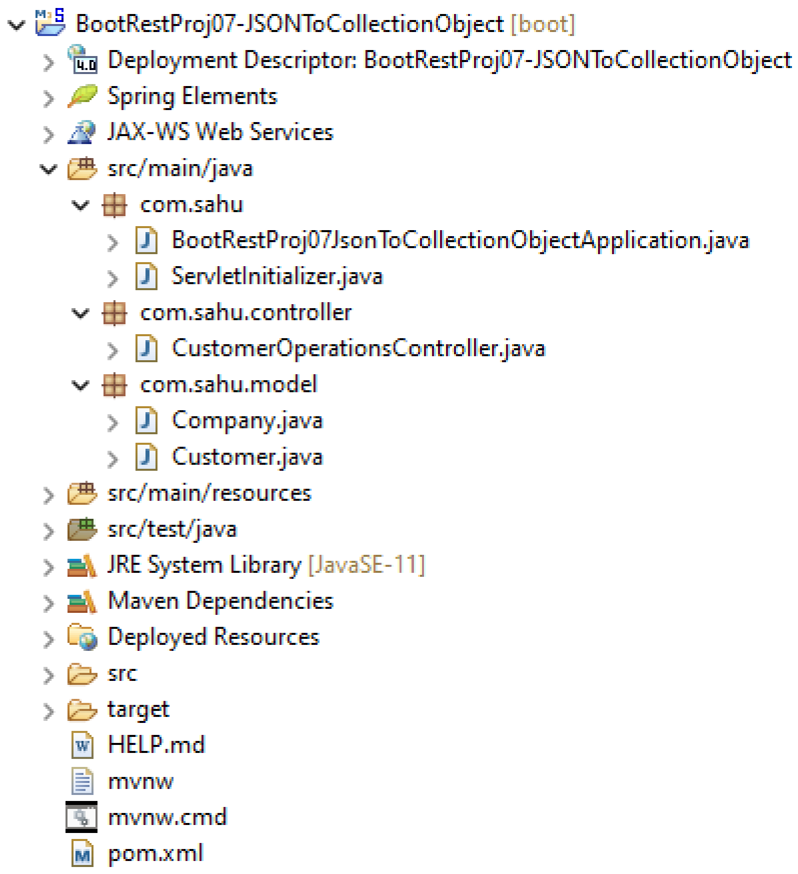
```
{
   "cno": 101,
   "cname": "rajesh",
   "favColors": [
      "red",
      "blue",
      "green"
   ],
   "academics": [
      "10",
      "+2",
      "BTech",
      "MTech"
   ],
   "phonNo": [
      43567543,
      45678975,
      345676543
   ],
   "billDetails": {
      "x-mass": 3433.77,
      "chocolates": 768.44
   },
   "address": {
      "houseNo": "1-2-/44",
      "streetName": "RK street",
      "location": "Hyderabad",
      "pinCode": 522345
   }
}
```

Converting JSON Data to Java class object using @RequestBody dealing with List<Object>, date, time values

- For List<Object> or Collection<Object> we need to take

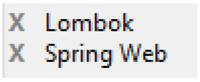
```
"<variable>": [
            {"key": value, "key": value, ….},
            {"key": value, "key": value, ….},
            …………
}
```

Directory Structure of BootRestProj07-JSONToCollectionoObject:



```
BootRestProj07-JSONToCollectionObject [boot]
  > Deployment Descriptor: BootRestProj07-JSONToCollectionObject
  > Spring Elements
  > JAX-WS Web Services
  v src/main/java
    v com.sahu
      > BootRestProj07JsonToCollectionObjectApplication.java
      > ServletInitializer.java
    v com.sahu.controller
      > CustomerOperationsController.java
    v com.sahu.model
      > Company.java
      > Customer.java
  > src/main/resources
  > src/test/java
  > JRE System Library [JavaSE-11]
  > Maven Dependencies
  > Deployed Resources
  > src
  > target
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starters during project creation.



```
X Lombok
X Spring Web
```

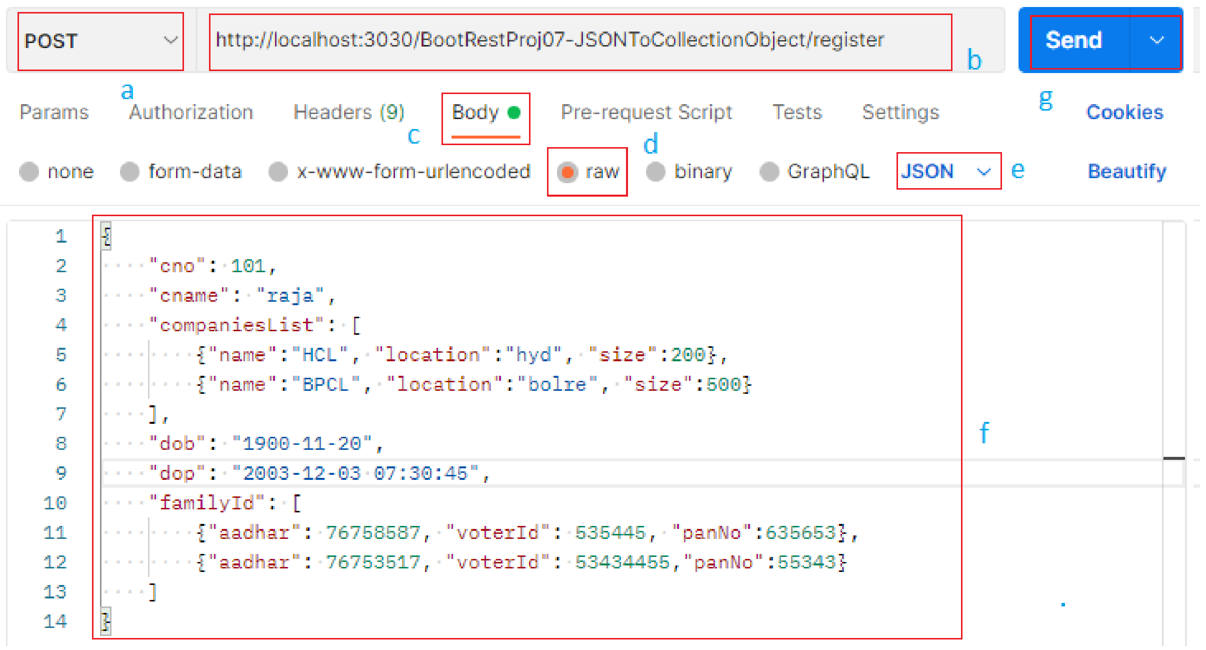
- Copy the CustomerOperationsController.java class from previous project.
- Then place the following code with in their respective files.

Company.java

```java
package com.sahu.model;

import lombok.Data;

@Data
public class Company {
	private String comapny;
	private String location;
	private Integer size;
}
```

Customer.java

```
package com.sahu.model;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.List;
import java.util.Map;

import com.fasterxml.jackson.annotation.JsonFormat;

import lombok.Data;

@Data
public class Customer {
	private Integer cno;
	private String cname;
	private List<Company> companiesList;
	@JsonFormat(pattern = "yyyy-MM-dd" )
	private LocalDate dob;
	@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
	private LocalDateTime dop;
	private List<Map<String, Long>> familyId;
}
```



```
{
    "cno": 101,
    "cname": "raja",
    "companiesList": [
        {"name":"HCL", "location":"hyd", "size":200},
        {"name":"BPCL", "location":"bolre", "size":500}
    ],
    "dob": "1900-11-20",
    "dop": "2003-12-03 07:30:45",
    "familyId": [
        {"aadhar": 76758587, "voterId": 535445, "panNo":635653},
        {"aadhar": 76753517, "voterId": 53434455,"panNo":55343}
    ]
}
```

Follow the above screen to give the request through the POSTMAN.

Request Body

```
{
   "cno": 101,
   "cname": "raja",
   "companiesList": [
      {
         "name": "HCL",
         "location": "hyd",
         "size": 200
      },
      {
         "name": "BPCL",
         "location": "bolre",
         "size": 500
      }
   ],
   "dob": "1900-11-20",
   "dop": "2003-12-03 07:30:45",
   "familyId": [
      {
         "aadhar": 76758587,
         "voterId": 535445,
         "panNo": 635653
      },
      {
         "aadhar": 76753517,
         "voterId": 53434455,
         "panNo": 55343
      }
   ]
}
```

Note: If the request body contains invalid JSON pattern content which can not be parsed by JSON parser then the RestController sends 400 (Bad Request) error response to browser.

Passing request Params to Spring Rest application

- The GET mode request does not contain body, so the data we want to send in GET mode request should always be in the form of Query String.

- The GET mode request can carry limited amount of data (max of 2KB) that to as query string params.
- In other mode requests (like POST, PUT, DELETE and etc.) the content in the request goes to server in the form of request body.
- For Spring RestController we can pass data in GET Mode request in two ways,
    a. As request params/ query param in the Query string of request URL (Supported by Spring MVC and Spring Rest).
       E.g., url?key1=va11&key2=va12&key3=va13

    b. As path variable values in the request URL (Supported by Spring Rest and not in Spring MVC)
       E.g., url/value/value/value
       {key} will be given in request path of business methods placed @RestController.

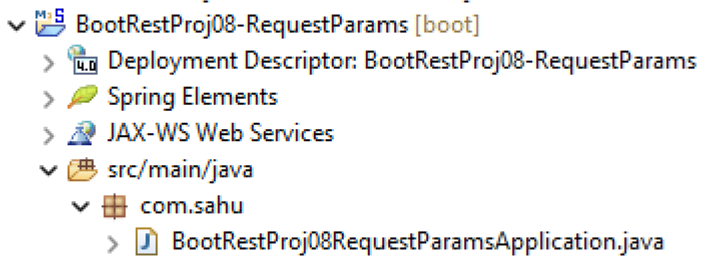
Note:

- ✓ All web technologies/ frameworks (both java and non-Java) support request params as the basic concept of web programming.
  E.g., Servlet, JSP, PHP, Asp.net, Node JS, Express JS and etc.
- ✓ Passing values in the request URL as PATH variable values in introduced in RESTful programming, so the all technologies and frameworks supporting the Restful programming (both java and non-java) gives provision to work with this path variable concept.

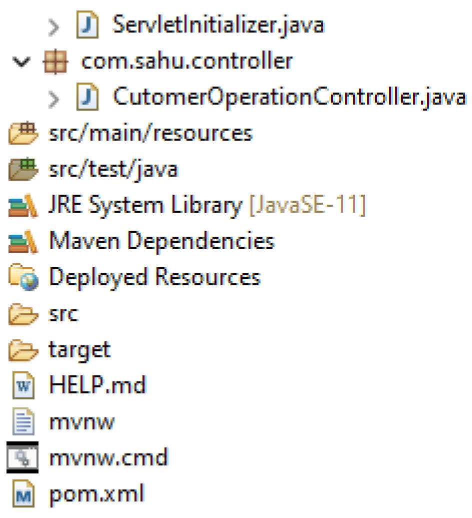
Request params/ Query params in Query String

- To place then in request URL, we need URL?key1=value1.
- To read them in the method of @RestController use,
  @RequestParam("key") Param Type paramName
  (or)
  @RequestParam Param Type paramName
  Here it must match with key of the request parameter

Directory Structure of BootRestProj08-RequestParams:

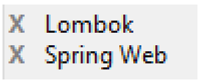


```
BootRestProj08-RequestParams [boot]
  > Deployment Descriptor: BootRestProj08-RequestParams
  > Spring Elements
  > JAX-WS Web Services
  v src/main/java
    v com.sahu
      > BootRestProj08RequestParamsApplication.java
```



```
      > ServletInitializer.java
    v com.sahu.controller
      > CutomerOperationController.java
  > src/main/resources
  > src/test/java
  > JRE System Library [JavaSE-11]
  > Maven Dependencies
  > Deployed Resources
  > src
  > target
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starters during project creation.



```
X Lombok
X Spring Web
```

- Then place the following code with in their respective files.

CustomerOperationController.java

```
package com.sahu.controller;

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

@RestController
public class CutomerOperationController {

	@GetMapping("/report")
	public String reportData(@RequestParam("cno") Integer no,
								@RequestParam String cname)
{
		return no+" "+cname;
	}

}
```

Give the following request through POSTMAN
http://localhost:3035/BootRestProj08-RequestParams/report?cno=101&cname=raja

- While request params in query String we can change the order of passing request param values.

```
        report?cname=raja&cno=101
                (or)                            Both are correct
        report?cno=101&cname=raja
```

- While using request params in query string if we pass additional params that expected that will not generate any error.

```
        report?cname=raja&cno=101&cadd=hyd
                        Extra but does not generate error
```

- When we pass query string to the URL in POST mode request, they internally become Request body content because the POST carries data as request body.

CustomerOperationController.java

```java
@RestController
public class CutomerOperationController {

	@GetMapping("/report")
	public String reportData(@RequestParam("cno") Integer no,
						@RequestParam(required =
false) String cname) {
		return no+" "+cname;
	}

}
```

URL: http://localhost:3030/SpringBootRestProj08-RequestParams/report?cno=101

gives 101 NULL, as response

Note: While required is true (default value), if we give request then we will get 400 bad requests.

Send data as Path variable values

- Supports to minimize characters in request URL while sending data.
- No need of passing separate query string in the request URL to send

data, we can pass data directly in the request URL itself.

- Passing "&, =" as values in request param values is not possible but can be done easily using path variable.
- Request URL with query string is not clean URL because it needs more characters (because both keys and values are required) and query string needs separate syntax to fallow.
- Request URL with path variable values are part of request URL itself and no need of separate syntax for it.
  Syntax: request URL (or) Path/<value1>/<value2>/<value3>/

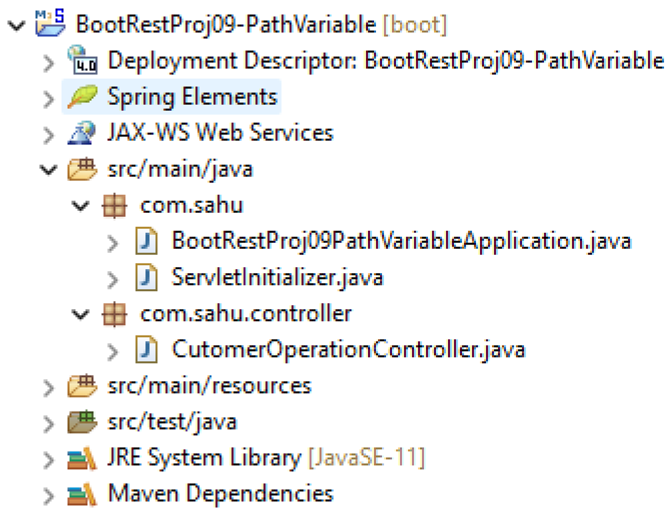
- {key} for those values will give at @RestController in XxxMapping(...) methods while defining the request path.
- To read path variable values in @RestController methods
  @PathVariable Datatype paramName
  param name must match {key} of request path
  (or)
  @PathVariable("key") Datatype paramName

- The request path contains two parts while working path variables
  a. static path (fixed path) (/<path>)
  b. Dynamic path (key name whose value comes from request URL) (/{key})

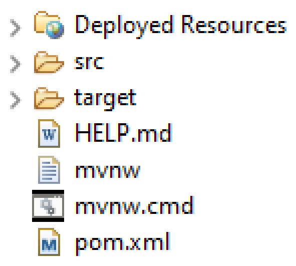
  E.g., GetMapping("{/report/{no}/{name}")
  static path Dynamic path

  Request URL
  http://localhost:3035/SpringRestProj9-PathVariables/report/101/raja

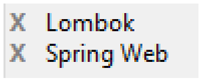
Directory Structure of BootRestProj09-PathVariable:



```
BootRestProj09-PathVariable [boot]
  Deployment Descriptor: BootRestProj09-PathVariable
  Spring Elements
  JAX-WS Web Services
  src/main/java
    com.sahu
      BootRestProj09PathVariableApplication.java
      ServletInitializer.java
    com.sahu.controller
      CutomerOperationController.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
```


```
> Deployed Resources
> src
> target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starters during project creation.



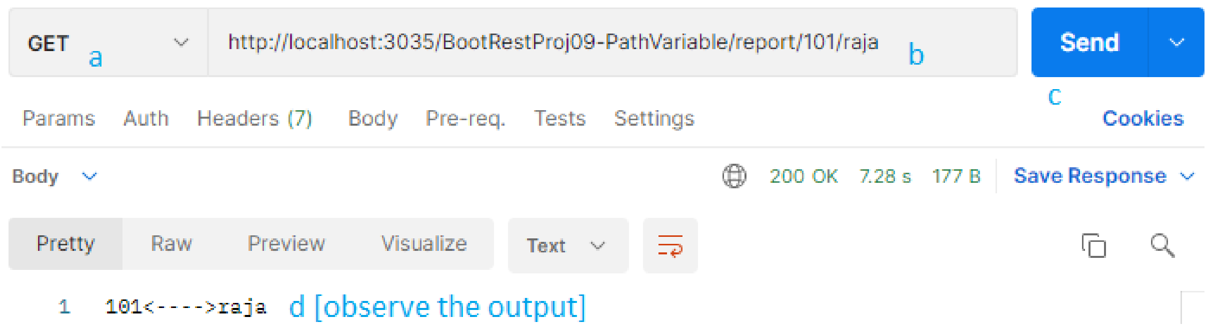
X Lombok
X Spring Web

- Then place the following code with in their respective files.

CustomerOperationController.java

```java
package com.sahu.controller;

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

@RestController
public class CutomerOperationController {

	@GetMapping("/report/{no}/{name}")
	public String reportData(@PathVariable("name") String cname,
							@PathVariable Integer no) {
		return no+"<---->"+cname;
	}

}
```

POSTMAN request and response



GET a http://localhost:3035/BootRestProj09-PathVariable/report/101/raja b Send

c

```
1  101<---->raja
```
d [observe the output]

- While working with request param we can change the order of passing their values.
  E.g.,
  /report?cno=101&cname=raja is same as
  /report?cname=raja&cno=101.

- While working path variable values we cannot change the order of passing values.
  E.g.,
  request path in @RestController method is
  GetMapping("/report/{no}/{name}")
  /report/101/raja valid
  /report/raja/101 invalid - Gives 400 Bad request if @PathVariable Integer no is taken
  /report/raja/101 valid - if both params are taken as String values but the wrong will be stored in method params so the business logic will be disturbed

- Giving extra path variable values (nothing but extra words path) in request URL results 404 error.
  E.g.,
  request path for @RestController method: /report/{no}/{name}
  request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/101/raja/hyd/india
  Extra dynamic values which are not required, so 404 error will come.

Working with Path Variables

- If we give more or less values as path variable values than expected then we get 404 error (requested resource is not found).
  E.g.,
  - Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/101/raja
    Gives: 101 <----> raja
  - Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/101
    404 error
  - Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report
    404 error

CustomerOperationController.java

```
@RestController
public class CutomerOperationController {

        @GetMapping("/report/{no}/{name}")
        public String reportData(@PathVariable(name="name", required =
false) String cname,
                                                    @PathVariable(required =
false) Integer no) {
                return no+"<---->"+cname;
        }

}
```

Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/101

- We expect 101 <----> null should come as the output but we get 404 error because number of levels in request path are 3 and we are giving only two so, it says requested resource is not found

Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report

- We expect null <----> null but we get 404 error (requested resource is not found).

- If multiple methods are having similar request paths having same number of levels, then the request that is having more static level matchings will get priority

CustomerOperationController.java

```
@RestController
public class CutomerOperationController {

        @GetMapping("/report/{no}/{name}")
        public String reportData(@PathVariable(name="name", required =
false) String cname,
                                                    @PathVariable(required =
false) Integer no) {
                return "from Report Data";
        }
        @GetMapping("/report/no/name")
        public String reportData1(@PathVariable(name="name", required =
false) String cname,
                        @PathVariable(required = false) Integer no) {
                return "from Report Data1";
        }

        @GetMapping("/report/no/{name}")
        public String reportData2(@PathVariable(name="name", required =
false) String cname,
                        @PathVariable(required = false) Integer no) {
                return "from Report Data2";
        }

        @GetMapping("/report/{no}/name")
        public String reportData3(@PathVariable(name="name", required =
false) String cname,
                        @PathVariable(required = false) Integer no) {
                return "from Report Data3";
        }

}
```

- Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/101/raja
  Output: from Report Data
- Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/no/name
  Output: from Report Data1
- Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/no/raja
  Output: from Report Data2
- Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/101/name
  Output: from Report Data

CustomerOperationController.java

```
@RestController
public class CutomerOperationController {
        @GetMapping("/report/{no}/{name}")
        public String reportData1(@PathVariable(name="name", required =
false) String cname,
                                                  @PathVariable(required =
false) Integer no) {
                return "from Report Data1";
        }

        @GetMapping("/report/101/raja")
        public String reportData2(@PathVariable(name="name", required =
false) String cname,
                        @PathVariable(required = false) Integer no) {
                return "from Report Data2";
        }

}
```

- Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/101/raja
  Output: from Report Data2
- Request URL: http://localhost:3030/SpringBootRestProj09-PathVariable/report/101/rajesh
  Output: from Report Data1

- If two methods of @RestController is having same request path with same number of levels then there is possibility of getting 500 IllegalStateException during the application startup

CustomerOperationController.java

```
@RestController
public class CutomerOperationController {

        @GetMapping("/report/{no}/{name}")
        public String reportData1(@PathVariable(name="name", required =
false) String cname,
                                                  @PathVariable(required =
false) Integer no) {
                return "from Report Data1";
        }
        @GetMapping("/report/101/raja")
        public String reportData2(@PathVariable(name="name", required =
false) String cname,
                        @PathVariable(required = false) Integer no) {
                return "from Report Data2";
        }

        @GetMapping("/report/101/raja")
        public String reportData3(@PathVariable(name="name", required =
false) String cname,
                        @PathVariable(required = false) Integer no) {
                return "from Report Data3";
        }

}
```

Caused by: java.lang.IllegalStateException: Ambiguous mapping. Cannot map 'cutomerOperationController' method

Q. What is the difference b/w Request params and Path variables way of passing data in Spring Rest app?

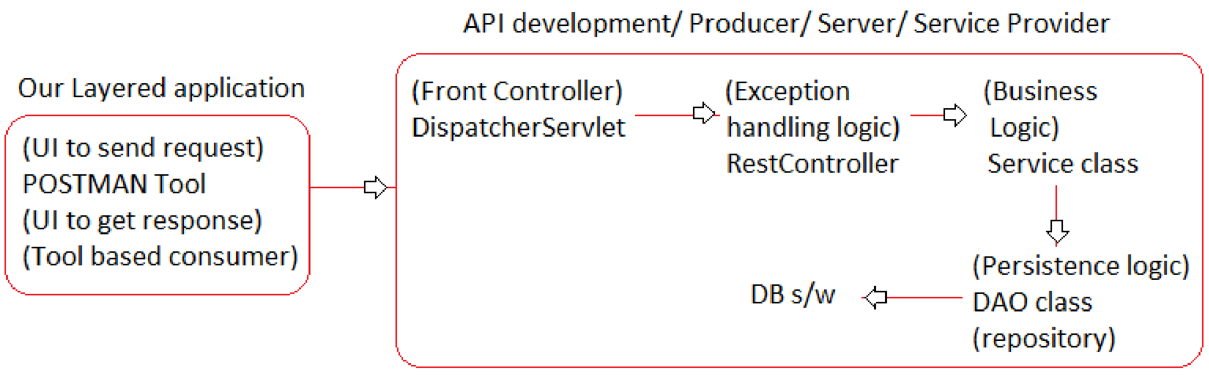
Ans.

Request params	Path variable
a. Syntax to pass data along with request URL is url?key1=val1&key2=val2&....	a. Syntax to pass data along with request URL is url/<static path>/va11/va12/.... {key}/{key}/ will be defined in @RestController class.
b. Syntax to read request param values is @RequestParam datatype param (or) @RequestParam("key") datatype param	b. Syntax to read path variable values is @PathVariable datatype param (or) @PathVariable("key") datatype param
c. While passing request param values in the query string of request URL the order need not to match.	c. While passing path variables values in the request URL the order must be matched.
d. If we pass more than required request params in query string the	d. If we pass more than required path variable values in the request URL

we will not error.	then we get 404 error.
e. If we pass less than required request params in query strung then we get error only when required-false is not taken @RequestParam.	e. If we pass less than required path variable values in the request URL then we get 404 error.
f. Does not allow to pass '&' as direct value, we must use %26 for that.	f. Does not allow "/" as value.
g) Supported by both Spring MVC and Spring Rest.	g. Supported only in Spring Rest.
h. In all web technologies that are there to develop web applications in different domains supports this feature basic concept to pass data.	h. supported only in RESTful programming of all domains.
i. URL is not clean URL (more characters are required in the URL to pass data).	i. URL is clean URL (less characters required in the URL to pass data).
j. Easy to read and interpret.	j. Complex to read to interpret.
k. Bit slow while sending data	k. Bit faster while sending the data.
l. Recomaned to use in non-Restful apps.	l. Very useful in Restful apps.
m. Generally used while working GET mode requests because data goes as query string.	m. Works with all modes of request because data goes directly from URL itself

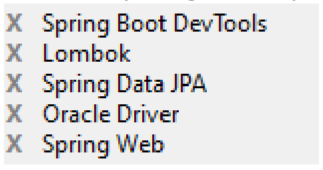
Mini Project using Spring Rest + Spring data JPA + Oracle + POSTMAN tool

- The methods of @RestController are called business methods or operations (best) or web operations.
- The operations of @RestController can have flexible signatures.
- We can place business logic directly in the operations of @RestController but it recommended to place in service class methods and invoke them from @RestController operations.
- Developing @RestController with multiple operations linked with Service, DAO class methods performing CURD Operations is nothing but developing Server/ Service Provider/ Rest API/ API/ Producer.
- Getting API and its operation details nothing but basic URL + request path + path variables info together is called gathering API and its endpoints.



Procedure:

Step 1: Create Spring Boot project adding the following staters,


- X Spring Boot DevTools
- X Lombok
- X Spring Data JPA
- X Oracle Driver
- X Spring Web

Step 2: Develop Entity class/ model class that is required in O-R mapping operations.

Step 3: Add the required properties in application.properties file for Data source configuration and JPA configuration.

Step 4: Develop the Repository Interface extending from JpaRepository.

Note:

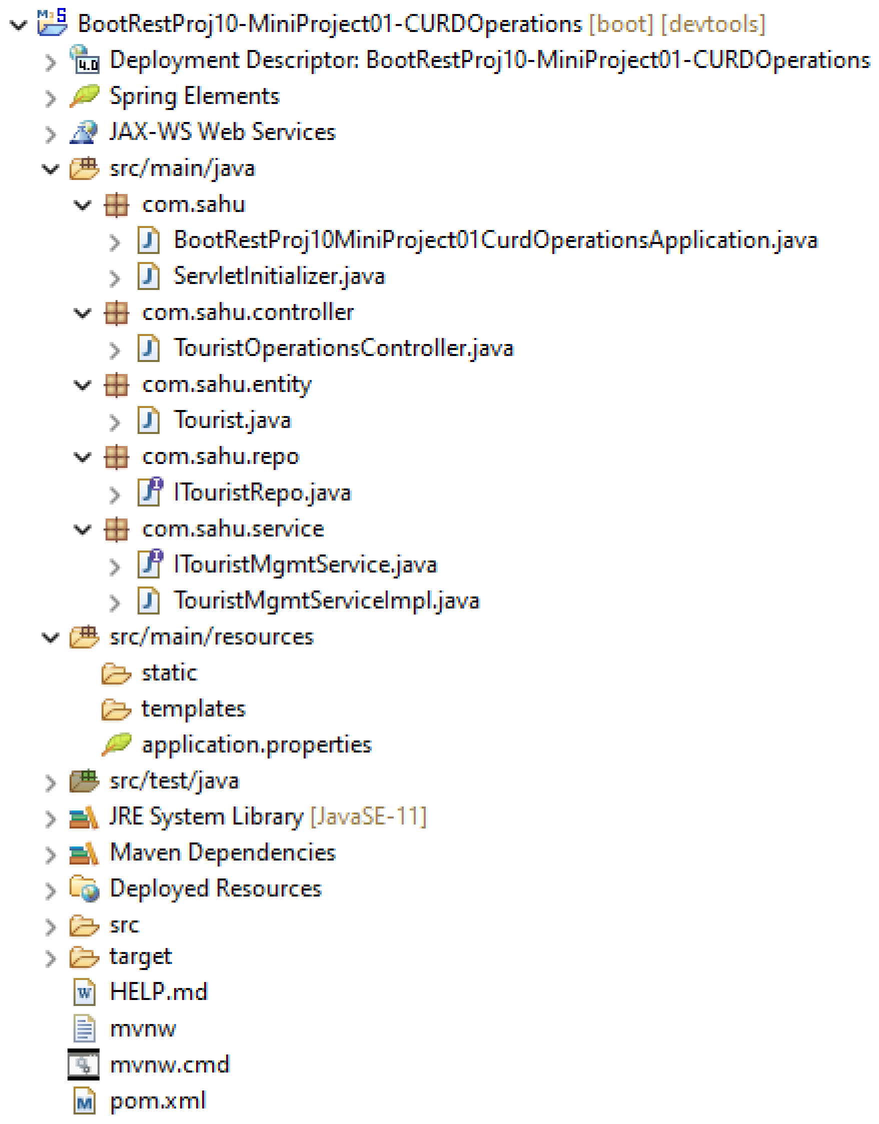
- ✓ The spring boot generates InMemory implementation class for our ITouristReo.java interface having Tourist Entity class-based O-R mapping persistence logic for all methods available or inherited for JpaRepository
- ✓ This kind of dynamic InMemory classes will be generated in the memory where apps run nothing but JVM memory of RAM.
- ✓ In JVM memory
  - heap for objects
  - stack for methods/local variables
  - Premgen/ Meta space for InMemory code (Java8) generation/ native code generation.

Step 5: Develop Service Interface and Service implementation class for save object operation.

Step 6: Develop the @RestController class having method for save operation.

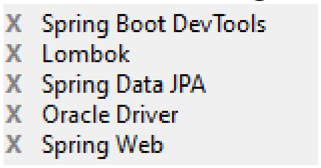
Step 7: Run the application and give request through POSTMAN.

Directory Structure of BootRestProj10-MiniProject01-CURDOperations:



```
BootRestProj10-MiniProject01-CURDOperations [boot] [devtools]
  Deployment Descriptor: BootRestProj10-MiniProject01-CURDOperations
  Spring Elements
  JAX-WS Web Services
  src/main/java
    com.sahu
      BootRestProj10MiniProject01CurdOperationsApplication.java
      ServletInitializer.java
    com.sahu.controller
      TouristOperationsController.java
    com.sahu.entity
      Tourist.java
    com.sahu.repo
      ITouristRepo.java
    com.sahu.service
      ITouristMgmtService.java
      TouristMgmtServiceImpl.java
  src/main/resources
    static
    templates
    application.properties
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  Deployed Resources
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starters during project creation.



X Spring Boot DevTools
X Lombok
X Spring Data JPA
X Oracle Driver
X Spring Web

- Then place the following code with in their respective files.

Tourist.java

```
package com.sahu.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
```

```java
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@RequiredArgsConstructor
@Table(name = "REST_TOURIST")
public class Tourist {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Integer tid;
	@Column(length = 20)
	@NonNull
	private String name;
	@Column(length = 20)
	@NonNull
	private String city;
	@Column(length = 20)
	@NonNull
	private String packageType;
	@NonNull
	private Double budget;
}
```

application.properties

```properties
#Datasource
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager

#JPA configurations
spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

ITouristRepo.java

```java
package com.sahu.repo;

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

import com.sahu.entity.Tourist;

public interface ITouristRepo extends JpaRepository<Tourist, Integer> {

}
```

ITouristMgmtService.java

```java
package com.sahu.service;

import com.sahu.entity.Tourist;

public interface ITouristMgmtService {
	public String registerTourist(Tourist tourist);
}
```

TouristMgmtServiceImpl.java

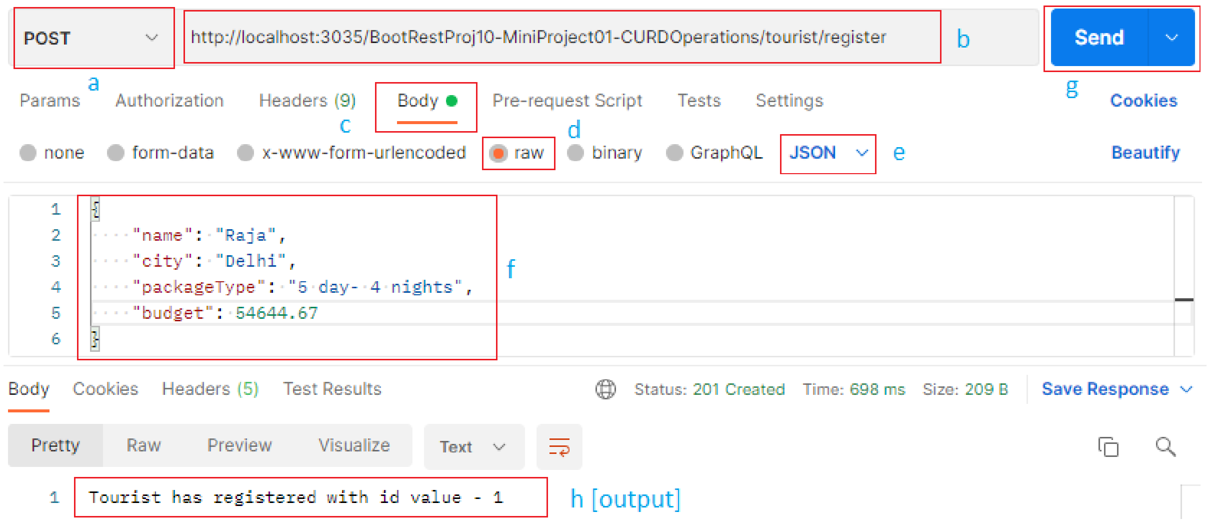
```java
package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.entity.Tourist;
import com.sahu.repo.ITouristRepo;

@Service("touristService")
public class TouristMgmtServiceImpl implements ITouristMgmtService {
        @Autowired
        private ITouristRepo touristRepo;

        @Override
        public String registerTourist(Tourist tourist) {
                int idVal = touristRepo.save(tourist).getTid();
                return "Tourist has registered with id value - "+idVal;
        }

}
```

TouristOperationsController.java

```
package com.sahu.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.sahu.entity.Tourist;
import com.sahu.service.ITouristMgmtService;

@RestController
@RequestMapping("/tourist")
public class TouristOperationsController {

        @Autowired
        private ITouristMgmtService touristMgmtService;

        @PostMapping("/register")
        public ResponseEntity<String> enrollTourist(@RequestBody Tourist
tourist){
                try {
                        //use service
                        String resultMsg =
touristMgmtService.registerTourist(tourist);
                        //HttpStatus.CREATED - 201 content created successfully
```

```
                    return new ResponseEntity<String>(resultMsg,
HttpStatus.CREATED);
                }
                catch (Exception e) {
                    return new ResponseEntity<String>("Problem in tourist
enrollment", HttpStatus.INTERNAL_SERVER_ERROR);
                }
        }

}
```

Give the request through POSTMAN and see the repose



POST http://localhost:3035/BootRestProj10-MiniProject01-CURDOperations/tourist/register

Body: none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
    "name": "Raja",
    "city": "Delhi",
    "packageType": "5 day- 4 nights",
    "budget": 54644.67
}
```

Status: 201 Created Time: 698 ms Size: 209 B

```
Tourist has registered with id value - 1
```

Performing findAll () operation in Mini project

ITouristMgmtService.java

```
        public String registerTourist(Tourist tourist);
```

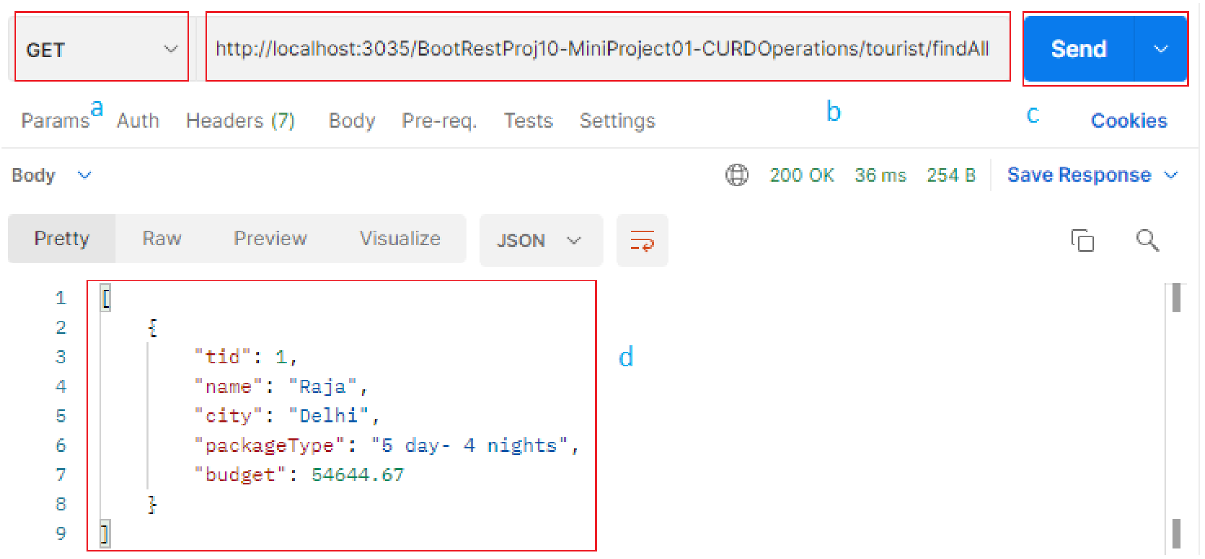
TouristMgmtServiceImpl.java

```
        @Override
        public List<Tourist> fetchAllTourist() {
                List<Tourist> listTourist = touristRepo.findAll();
                listTourist.sort((t1,t2)->t1.getTid().compareTo(t2.getTid()));
                return listTourist;
        }
```

TouristOperationsController.java

```
        @GetMapping("/findAll")
        public ResponseEntity<?> displayAllTourist(){
                try {
                        List<Tourist> listTourist = 
touristMgmtService.fetchAllTourist();
                        return new ResponseEntity<List<Tourist>>(listTourist, 
HttpStatus.OK);
                }
                catch (Exception e) {
                        return new ResponseEntity<String>("Problem in 
fetchinhg tourists", HttpStatus.INTERNAL_SERVER_ERROR);
                }
        }
```

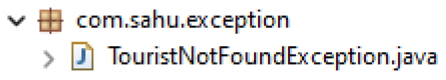
Give the following request through POSTMAN and get the response



```
[
    {
        "tid": 1,
        "name": "Raja",
        "city": "Delhi",
        "packageType": "5 day- 4 nights",
        "budget": 54644.67
    }
]
```

Performing findTouristByTid () operation in Mini project

- Create a custom exception class under exception package as shown like below



ITouristMgmtService.java

```
        public Tourist fetchTouristById(Integer tid) throws 
TouristNotFoundException;
```

TouristMgmtServiceImpl.java

```
	@Override
	public Tourist fetchTouristById(Integer tid) throws 
TouristNotFoundException {
		return touristRepo.findById(tid).orElseThrow(()-> new 
TouristNotFoundException(tid+" tourist not found"));
	}
```

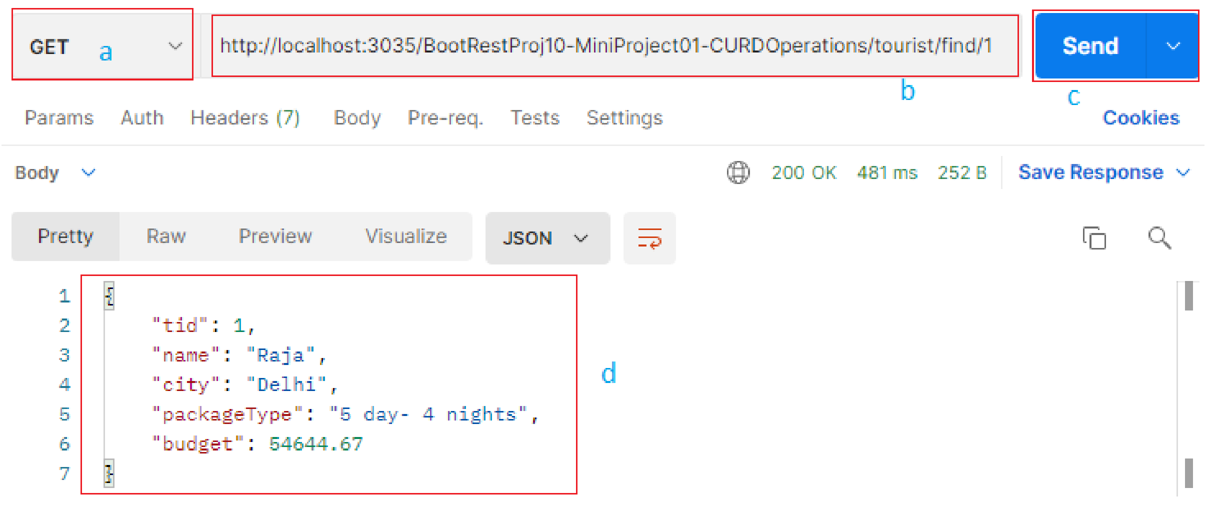
TouristNotFoundException.java

```
package com.sahu.exception;

public class TouristNotFoundException extends Exception {

	private static final long serialVersionUID = 1L;

	public TouristNotFoundException(String msg) {
		super(msg);
	}

}
```

TouristOperationsController.java

```
	@GetMapping("/find/{id}")
	public ResponseEntity<?> displayTouristById(@PathVariable("id") 
Integer id){
		try {
			Tourist tourist = touristMgmtService.fetchTouristById(id);
			return new ResponseEntity<Tourist>(tourist, 
HttpStatus.OK);
		}
		catch (Exception e) {
			return new ResponseEntity<String>(e.getMessage(), 
HttpStatus.INTERNAL_SERVER_ERROR);
		}
	}
```

Give the following request through POSTMAN and get the response, and if “id” is not there then you will get a message id tourist not found



http://localhost:3035/BootRestProj10-MiniProject01-CURDOperations/tourist/find/1

```
{
    "tid": 1,
    "name": "Raja",
    "city": "Delhi",
    "packageType": "5 day- 4 nights",
    "budget": 54644.67
}
```

Performing update operation in Mini project

ITouristMgmtService.java

```
        public String updateTouristDetails(Tourist tourist)  throws
TouristNotFoundException;
```

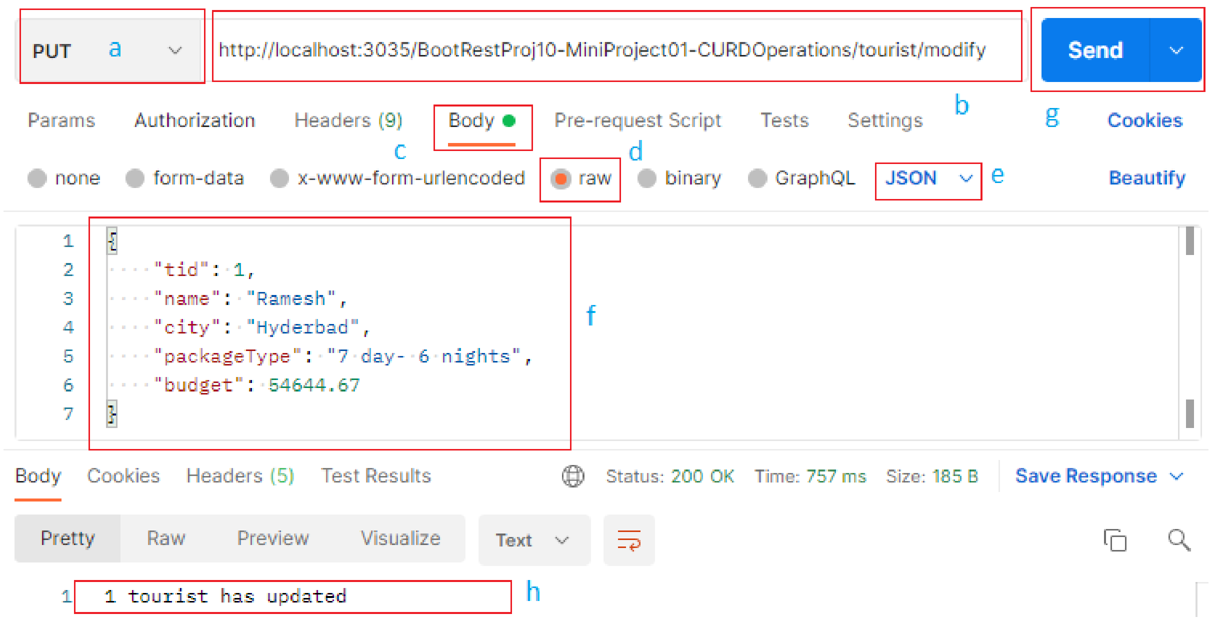
TouristMgmtServiceImpl.java

```
        @Override
        public String updateTouristDetails(Tourist tourist) throws
TouristNotFoundException {
                Optional<Tourist> optTourist =
touristRepo.findById(tourist.getTid());
                if (optTourist.isPresent()) {
                        //save(-) performs either save object or update object
operation
                        touristRepo.save(tourist);
                        return tourist.getTid()+" tourist has updated";
                }
                throw new TouristNotFoundException(tourist.getTid()+" tourist
not found");
        }
```

TouristOperationsController.java

```
        @PutMapping("/modify")
        public ResponseEntity<String> modifyTourist(@RequestBody Tourist
tourist){
                try {
                        String msg = 
touristMgmtService.updateTouristDetails(tourist);
                        return new ResponseEntity<String>(msg, HttpStatus.OK);
                }
                catch (Exception e) {
                        e.printStackTrace();
                        return new ResponseEntity<String>(e.getMessage(), 
HttpStatus.INTERNAL_SERVER_ERROR);

                }
        }
```

Give the following request through POSTMAN and get the response



http://localhost:3035/BootRestProj10-MiniProject01-CURDOperations/tourist/modify

```
{
    "tid": 1,
    "name": "Ramesh",
    "city": "Hyderbad",
    "packageType": "7 day- 6 nights",
    "budget": 54644.67
}
```

Status: 200 OK   Time: 757 ms   Size: 185 B

```
1 tourist has updated
```

Delete operation in Mini project

ITouristMgmtService.java

```
        public String deleteTourist(Integer tid) throws 
TouristNotFoundException;
```

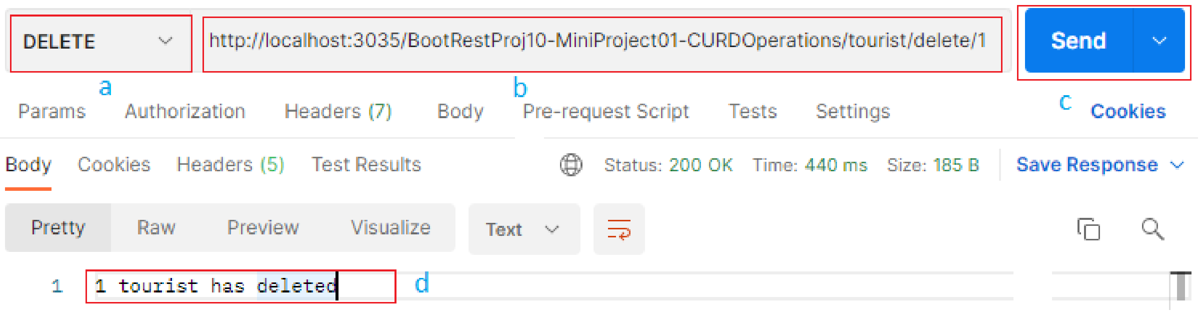
TouristMgmtServiceImpl.java

```
        @Override
        public String deleteTourist(Integer tid) throws 
TouristNotFoundException {
                Optional<Tourist> optTourist = touristRepo.findById(tid);
                if (optTourist.isPresent()) {
                        touristRepo.delete(optTourist.get());
                        return tid + " tourist has deleted";
                } else {
                        throw new TouristNotFoundException(tid + " tourist not
found");
                }
        }
```

TouristOperationsController.java

```
        @DeleteMapping("/delete/{id}")
        public ResponseEntity<String> removeTourist(@PathVariable("id")
Integer tid) {
                try {
                        String msg = touristMgmtService.deleteTourist(tid);
                        return new ResponseEntity<String>(msg, HttpStatus.OK);
                }
                catch (Exception e) {
                        e.printStackTrace();
                        return new ResponseEntity<String>(e.getMessage(),
HttpStatus.NOT_FOUND);
                }
        }
```

Give the following request through POSTMAN and get the response


DELETE http://localhost:3035/BootRestProj10-MiniProject01-CURDOperations/tourist/delete/1

Status: 200 OK Time: 440 ms Size: 185 B

```
1 tourist has deleted
```

Performing partial update using @PatchMapping

ITouristMgmtService.java

```
        public String updateTouristBudgetById(Integer tid, Float  hikePercent)
throws TouristNotFoundException;
```

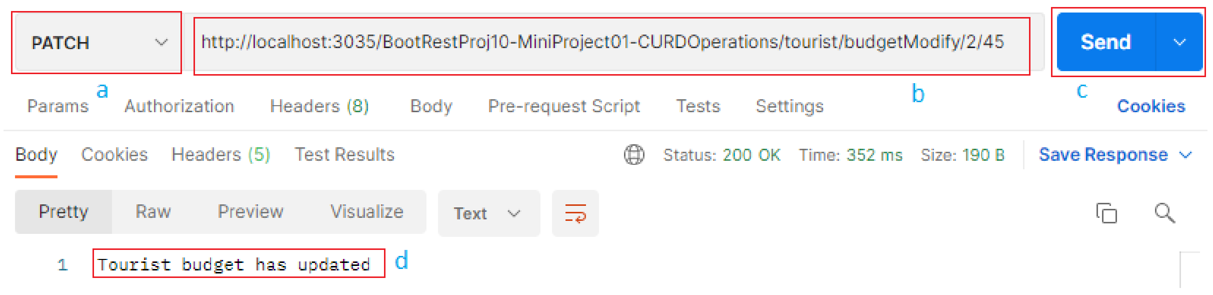
TouristMgmtServiceImpl.java

```
	@Override
	public String updateTouristBudgetById(Integer tid, Float hikePercent)
throws TouristNotFoundException {
		Optional<Tourist> optTourist = touristRepo.findById(tid);
		if (optTourist.isPresent()) {
			Tourist tourist = optTourist.get();

	tourist.setBudget(tourist.getBudget()+(tourist.getBudget()*(hikePerce
nt/100)));
			//use repo
			touristRepo.save(tourist);
			return "Tourist budget has updated";
		} else {
			throw new TouristNotFoundException(tid + " tourist not
found");
		}
	}
```

TouristOperationsController.java

```
	@PatchMapping("/budgetModify/{id}/{hike}")
	public ResponseEntity<String>
ModifyTouristBudget(@PathVariable("id") Integer id, @PathVariable("hike")
Float hikePercent) {
		try {
			//use service
			String msg =
touristMgmtService.updateTouristBudgetById(id, hikePercent);
			return new ResponseEntity<String>(msg, HttpStatus.OK);
		}
		catch (Exception e) {
			e.printStackTrace();
			return new ResponseEntity<String>(e.getMessage(),
HttpStatus.NOT_FOUND);
		}
	}
```

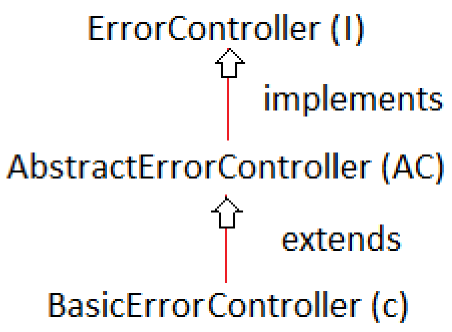
Give the following request through POSTMAN and get the response



```
Tourist budget has updated
```

Error Controller in Spring Boot Rest

- Spring Rest/ Spring Boot MVC is having pre-defined Controller to handle this HTTP Errors/ exception and to give default White Label error pages.



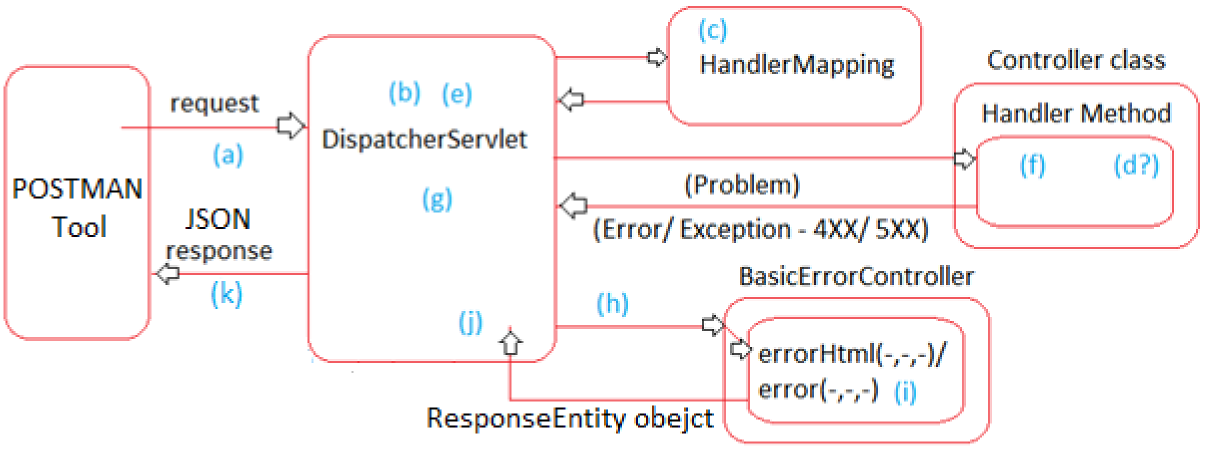
BasicErrorController

- Request path is (<contextpath>/error)
- This ErrorController can handle both Spring Boot MVC & Spring Rest application HTTP Errors/ Exceptions and generates White Label Error pages.
- This class have two important methods
  - errorHtml (-, -, -) (return type is ModelAndView) [Browser generated requests for Spring MVC/ Spring Rest application)
  - error (-, -, -) (return type is ResponseEntity) [Non-browser generated requests for Spring MVC/ Spring Rest application Desktop app/ Mobile app/ Tools (Postman), etc.]

Flow of Error/ Exception handling in Spring Boot Rest

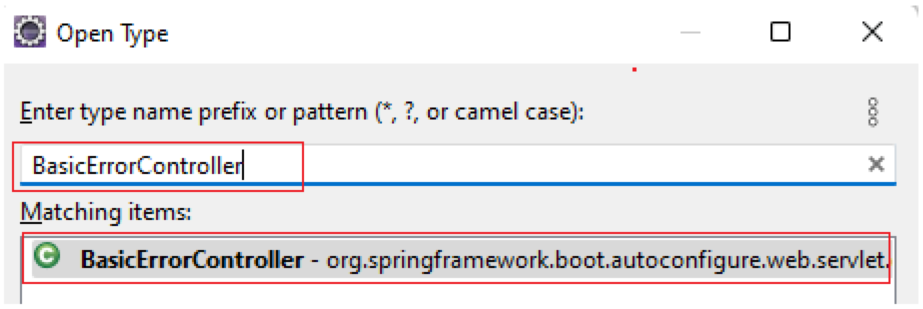
- If we do not catch and handle exception in the business methods of RestController (API) class then the DS gets the Propagated Exception and gives to error (-, -, -) of pre-defined controller class BasicErrorController which is mapped with <request path>/error (like tourist/find/error).
- The error (-, -, -) of BasicErrorController class returns ResponseEntity<Map<String, Object>> object having default error messages to DS and DS converts those messages to JSON Details to send

to Client as error JSON Response.



To feel this practically

a. Open BasicErrorController class using CTRL + SHIFT + T option.



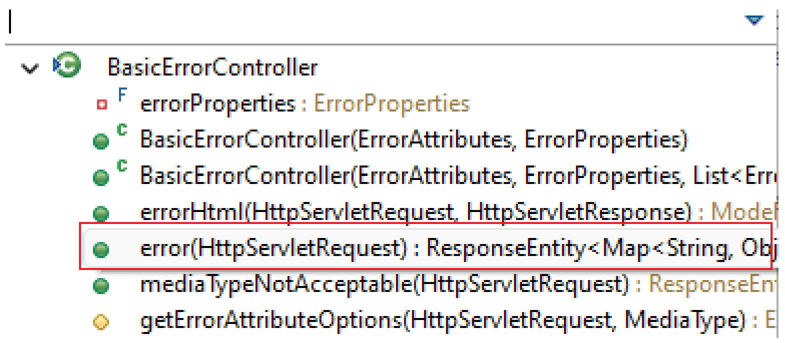
Enter type name prefix or pattern (*, ?, or camel case):

BasicErrorController

Matching items:

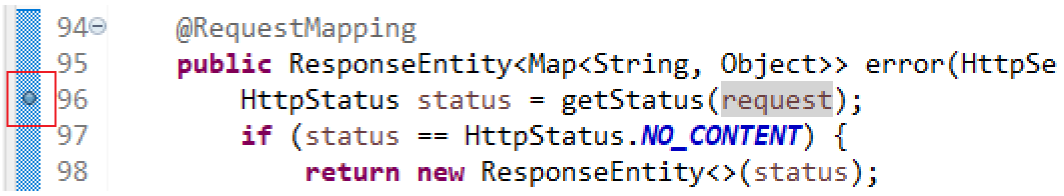
BasicErrorController - org.springframework.boot.autoconfigure.web.servlet.

b. Get of list of methods using CTRL + O, open error (-,-,-) source code.



```
BasicErrorController
    errorProperties : ErrorProperties
    BasicErrorController(ErrorAttributes, ErrorProperties)
    BasicErrorController(ErrorAttributes, ErrorProperties, List<Err
    errorHtml(HttpServletRequest, HttpServletResponse) : Model
    error(HttpServletRequest) : ResponseEntity<Map<String, Obj
    mediaTypeNotAcceptable(HttpServletRequest) : ResponseEnt
    getErrorAttributeOptions(HttpServletRequest, MediaType) : E
```

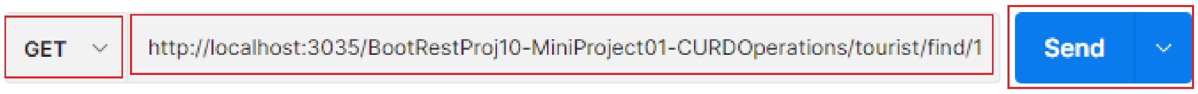
c. Keep one breakpoint inside the error (-, -, -) method using CTRL + B or using double click in left margin.



```
@RequestMapping
public ResponseEntity<Map<String, Object>> error(HttpSe
    HttpStatus status = getStatus(request);
    if (status == HttpStatus.NO_CONTENT) {
        return new ResponseEntity<>(status);
```

d. Run the app using Debug As server option.

e. Given give request from POSTMAN causing exception

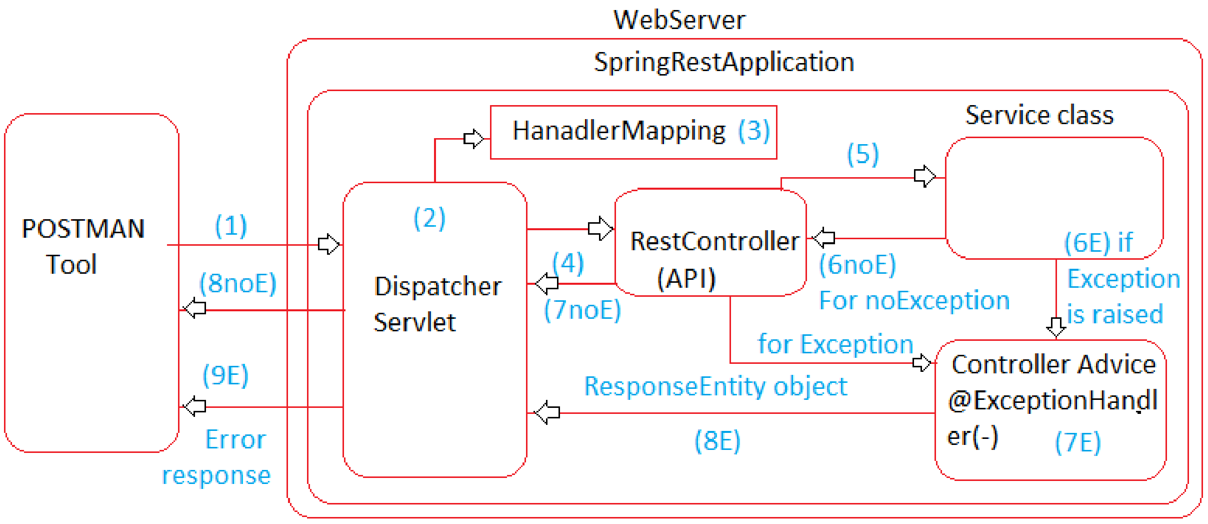


http://localhost:3035/BootRestProj10-MiniProject01-CURDOperations/tourist/find/1

f. Press F8 button to move control from default errorHtml (-, -, -) method to error (-, -, -) method then continuously press f7 button to go further.

g. Observe ResponseEntity<Map<String, Object>> object-based JSON Error response in POSTMAN tool.

- Instead of using BasicErrorController to handle the exceptions raised or propagated to RestController methods we can use custom throws advice class.
- That is developed using @ControllerAdvice (or) @RestControllerAdvice + @ExceptionHandler which can return ResponseEntity object to DispatcherServlet directly by catch the exception from Service class method as shown the below diagram.



<span style="color:red">Note:</span> The @ControllerAdvice or @RestControllerAdvice class @ExceptionHandler methods respond to execute for exceptions raised in Repository, Service, RestController classes.

Example Application:

Step 1: Keep Mini project ready

Step 2: Develop ErrorDetail.java class as the model class to hold more details.

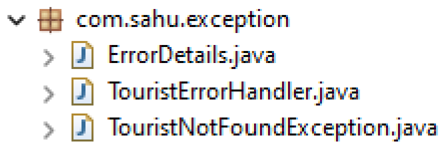
Step 3: Develop @ControllerAdvice or @RestControllerAdvice class

Step 4: Remove try catch blocks in the methods of RestController class.

Step 5: Run the class that will cause the exception.

Directory Structure of BootRestProj11-MiniProject-ExceptionHandler:

- Copy and paste the BootRestProj10-MiniProject01-CURDOperations and rename to BootRestProj11-MiniProject-ExceptionHandler.
- After that change the Web Project Setting context root to BootRestProj11-MiniProject-ExceptionHandler.
- Then change in <artifactId> & <name> tag of pom.xml.
- Add the following class under the following package.



```
com.sahu.exception
  ErrorDetails.java
  TouristErrorHandler.java
  TouristNotFoundException.java
```

- Then place the following code with in their respective files.

TouristErrorHandler.java

```
package com.sahu.exception;

import java.time.LocalDateTime;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

//@ControllerAdvice
@RestControllerAdvice
public class TouristErrorHandler {

	//@ExceptionHandler(Exception.class)
	@ExceptionHandler(TouristNotFoundException.class)
	public ResponseEntity<ErrorDetails>
handleTouristNotFound(TouristNotFoundException notFoundException){

	System.out.println("TouristErrorHandler.handleTouristNotFound()");
		ErrorDetails details = new ErrorDetails(LocalDateTime.now(),
notFoundException.getMessage(), "404-Tourist not found");
notFoundException.getMessage(), "404-Tourist not found");
                return new ResponseEntity<ErrorDetails>(details,
HttpStatus.NOT_FOUND);
        }

        @ExceptionHandler(Exception.class)
        public ResponseEntity<ErrorDetails> handleAllProblems(Exception e){
                System.out.println("TouristErrorHandler.handleAllProblems()");
                ErrorDetails details = new ErrorDetails(LocalDateTime.now(),
e.getMessage(), "Problem in execution");
                return new ResponseEntity<ErrorDetails>(details,
HttpStatus.INTERNAL_SERVER_ERROR);
        }

}
```

ErrorDetails.java

```
package com.sahu.exception;

import java.time.LocalDateTime;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class ErrorDetails {
        private LocalDateTime time;
        private String message;
        private String status;
}
```

TouristOperationsController.java

```
@RestController
@RequestMapping("/tourist")
public class TouristOperationsController {


        @Autowired
```

```
	@Autowired
	private ITouristMgmtService touristMgmtService;

	@PostMapping("/register")
	public ResponseEntity<String> enrollTourist(@RequestBody Tourist
tourist) throws Exception {
		//use service
		String resultMsg = touristMgmtService.registerTourist(tourist);
		//HttpStatus.CREATED - 201 content created sucessfully
		return new ResponseEntity<String>(resultMsg,
HttpStatus.CREATED);
	}

	@GetMapping("/findAll")
	public ResponseEntity<?> displayAllTourist() throws Exception {
		List<Tourist> listTourist = touristMgmtService.fetchAllTourist();
		return new ResponseEntity<List<Tourist>>(listTourist,
HttpStatus.OK);
	}

	@GetMapping("/find/{id}")
	public ResponseEntity<?> displayTouristById(@PathVariable("id")
Integer id) throws Exception{
		Tourist tourist = touristMgmtService.fetchTouristById(id);
		return new ResponseEntity<Tourist>(tourist, HttpStatus.OK);
	}

	@PutMapping("/modify")
	public ResponseEntity<String> modifyTourist(@RequestBody Tourist
tourist) throws Exception {
		String msg = touristMgmtService.updateTouristDetails(tourist);
		return new ResponseEntity<String>(msg, HttpStatus.OK);
	}

	@DeleteMapping("/delete/{id}")
	public ResponseEntity<String> removeTourist(@PathVariable("id")
Integer tid) throws Exception {
		String msg = touristMgmtService.deleteTourist(tid);
		return new ResponseEntity<String>(msg, HttpStatus.OK);
	}
```

```
                @PatchMapping("/budgetModify/{id}/{hike}")
        public ResponseEntity<String>
ModifyTouristBudget(@PathVariable("id") Integer id, @PathVariable("hike")
Float hikePercent) throws Exception {
                //use service
                String msg = touristMgmtService.updateTouristBudgetById(id,
hikePercent);
                return new ResponseEntity<String>(msg, HttpStatus.OK);
        }

}
```

Swagger - API Documentation

API Creation/ Development:

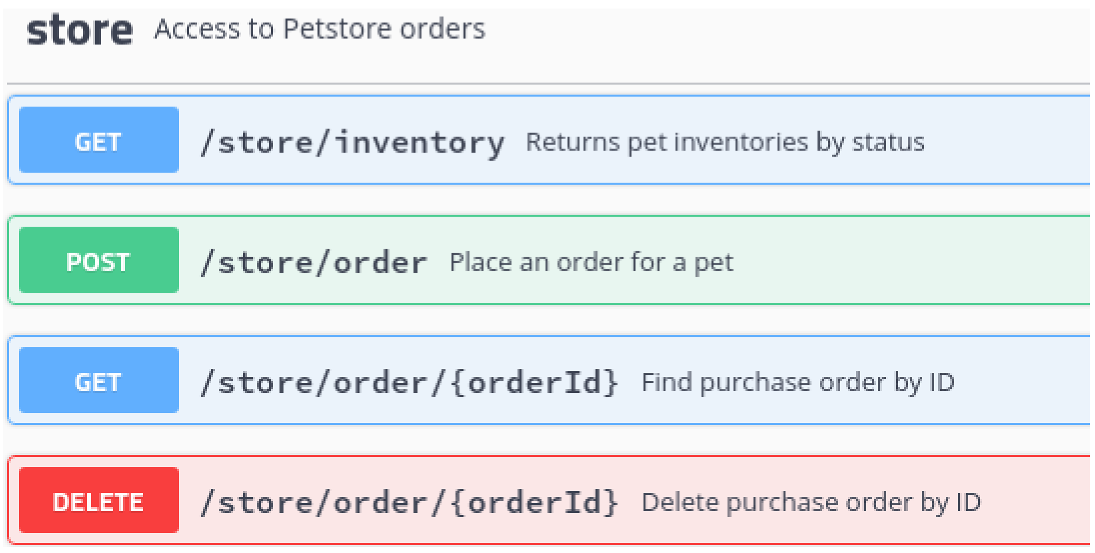
- Developing RestController having different methods/ operations for various HTTP method types like and etc. is called API Creation/ Development.
- We can generally take one @RestController per 1 module so, developing each RestController is called API creation.
- E.g.,
    - Accounts module --> AccountsController (@RestController) is called Accounts API creation.
    - Daily Tx module --> DailyTxController (@RestController) is called Daily Tx API creation.

      and etc.

End Points:

- Providing multiple details or collection of details that are required to call methods/ operations of @RestController from Client Apps or to send requests from different tools like POSTMAN is called Providing end points.
- E.g., For AccountController nothing but Accounts API the end points are
    - Base URL: http://localhost:3030/RestProj1/tourist
    - register (): /register ----> POSR
    - findById(-): /find/{id} ----> GET
    - deleteById(-): /delete/{id} ----> DELETE

      and etc.
- In End Points of any API, we need to provide multiple details like URL, method names, request paths, HTTP method types, content type & etc.


store Access to Petstore orders

GET /store/inventory Returns pet inventories by status

POST /store/order Place an order for a pet

GET /store/order/{orderId} Find purchase order by ID

DELETE /store/order/{orderId} Delete purchase order by ID

API Documentation:

- We can write documentation for java classes in multiple ways
  - Using separate Text docs.
  - Using API documentation comments and Javadoc tool.

Note: Both these approaches non-responsive documentations i.e., we can read about Java classes and methods but we cannot test them immediately.

- After developing Rest API, we can provide API documentation for Rest API in the following ways
  - Using separate Text docs Create Non-Responsive
  - Using API doc comments (/** …. */) API documentation
  - Using Swagger /Swagger API (Best)
    (or) Open API Creates Responsive API Documentation.

Note: Open API is alternate to Swagger API, the industry standard is still Swagger API.

Swagger API

- It is an open Source Third Party Library to provide Responsive API documentation for RestController and its methods.
- For All RestController of the project we can create API documentation from single place while working Swagger API.
- Responsive Documentation means not only we get docs about API and its methods (End points) we can test immediately by providing inputs and by getting outputs.

- If this is used there is no need of using POSTMAN tool separately and also, very useful to provide documentation-based testing environment from Clients.
- Spring Fox + Swagger together released libraries that are required to use Swagger API in Spring Boot applications.
- Swagger API documentation provides the following details in the GUI Responsive docs
    a) API Info (company, title, license URL, and etc.)
    b) End points info
    c) Model classes info
    and etc.

- While working swagger documentation for reading and testing we need not to remember and give URL, HTTP method types, content type and etc. we just need to give required inputs and get the outputs.

Procedure to work with Swagger API

Step 1: Keep RestController/ Rest API project ready.

Step 2: Add the following two jar file in pom.xml related to Swagger API
- SpringFox Swagger2
- SpringFox Swagger UI

Step 3: Develop separate Configuration class enabling Swagger API.
- In Configuration class create Docket object having
    - Documentation type (screen type)
    - Specify base package of RestController
    - Specify request paths info
    - Other details of API (ApiInfo object having company name, license URL and etc.).

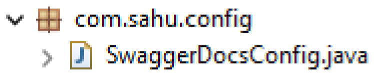
Step 4: Run the application

Step 5: Use the following URL to get Swagger API docs and to test the API
URL: http://localhost:3030/SpringBootRestProj12-MiniProject-Swagger-API/swagger-ui.html

Note: If you use Spring Boot 2.6, which is very recent and new thus Spring Doc and SpringFox might not support it already. So, I would also suggest you use Spring Boot 2.5.7.

Directory Structure of BootRestProj12-MiniProject-Swagger-API:

- Copy and paste the BootRestProj11-MiniProject01-ExceptionHandler and rename to BootRestProj12-MiniProject-Swagger-API.
- After that change the Web Project Setting context root to BootRestProj12-MiniProject-Swagger-API.
- Then change in <artifactId> & <name> tag of pom.xml.
- Add the following class and package.



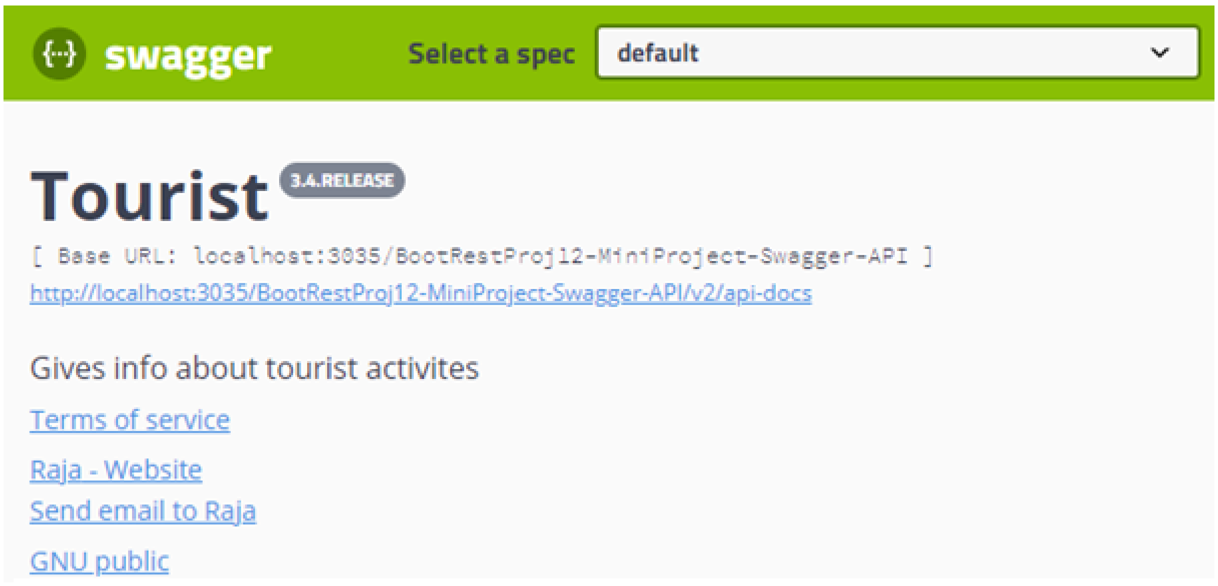
```
com.sahu.config
  SwaggerDocsConfig.java
```

- If you Spring boot version is 2.6.* change to 2.5.* or 2.5.7 (recommended).
- Then place the following code with in their respective files.

SwaggerDocsConfig.java

```
package com.sahu.config;

import java.util.Collections;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerDocsConfig {

	@Bean
	public Docket createDocket() {
		return new Docket(DocumentationType.SWAGGER_2) //UI
screen type
				.select() //To specify RestController

	.apis(RequestHandlerSelectors.basePackage("com.sahu.controller"))
//base package for RestController
                                        .paths(PathSelectors.regex("/tourist.*")) //To
specify Request paths
                                        .build()
                                        .useDefaultResponseMessages(true)
                                        .apiInfo(getApiInfo());
          }

          private ApiInfo getApiInfo() {
                    Contact contact = new Contact("Raja",
"http://www.HCL.com/tourist", "raja@gmail.com");
                    return new ApiInfo("Tourist",
                                                            "Gives info about tourist activites",
                                                            "3.4.RELEASE",
                                                            "http://www.hcl.com/license",
                                                            contact,
                                                            "GNU public",
                                                            "http://apache.org/license/gnu",
                                                            Collections.emptyList());
          }

}
```

Give the below request URL in browser
http://localhost:3030/SpringBootRestProj12-MiniProject-Swagger-API/swagger-ui.html



Tourist 3.4.RELEASE

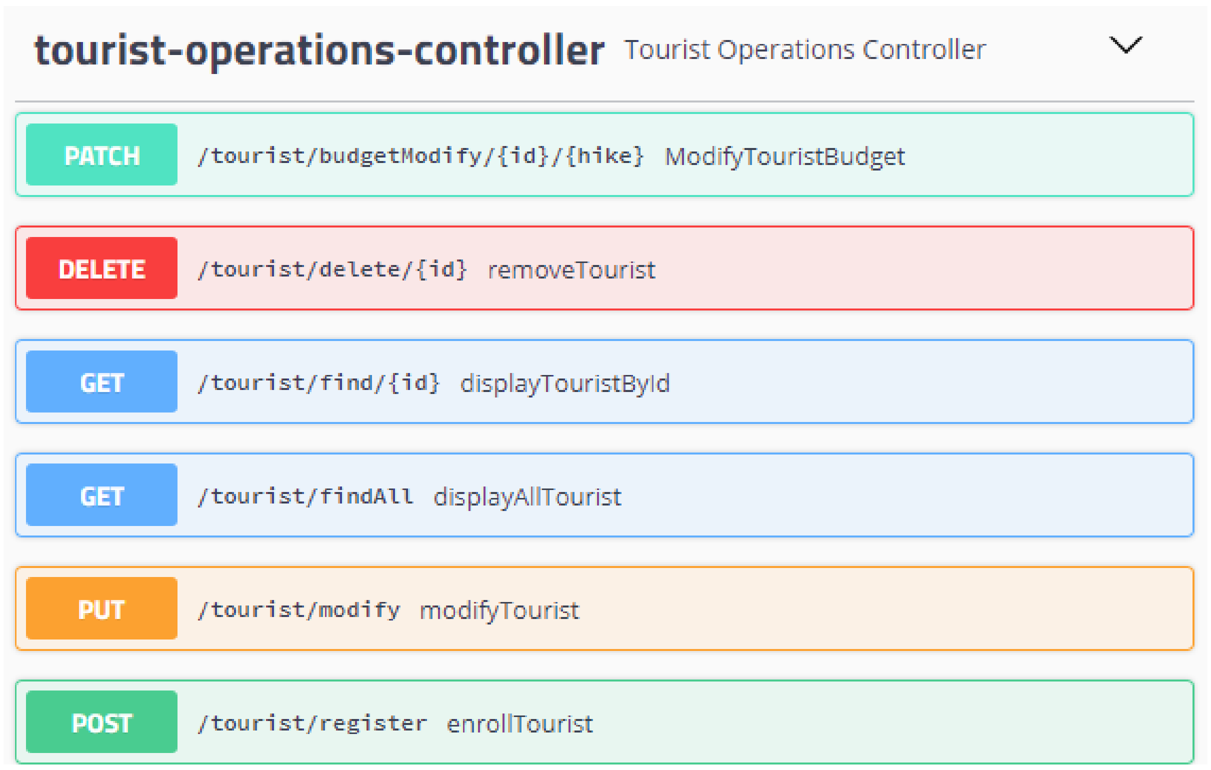
[Base URL: localhost:3035/BootRestProj12-MiniProject-Swagger-API]
http://localhost:3035/BootRestProj12-MiniProject-Swagger-API/v2/api-docs

Gives info about tourist activites

Terms of service

Raja - Website
Send email to Raja

GNU public



tourist-operations-controller Tourist Operations Controller

PATCH /tourist/budgetModify/{id}/{hike} ModifyTouristBudget

DELETE /tourist/delete/{id} removeTourist

GET /tourist/find/{id} displayTouristById

GET /tourist/findAll displayAllTourist

PUT /tourist/modify modifyTourist

POST /tourist/register enrollTourist

Note: @ApiOperation ("….") can be applied on the top of Rest API/ Controller methods to provide our choice description and that reflects in swagger docs.

```
@PostMapping("/register")
@ApiOperation("For Tourist registration")
public ResponseEntity<String> enrollTourist(@RequestBody Tourist tourist)
throws Exception {
        //use service
        String resultMsg = touristMgmtService.registerTourist(tourist);
        return new ResponseEntity<String>(resultMsg, HttpStatus.CREATED);
}
```

Developing Consumer application using Rest Template

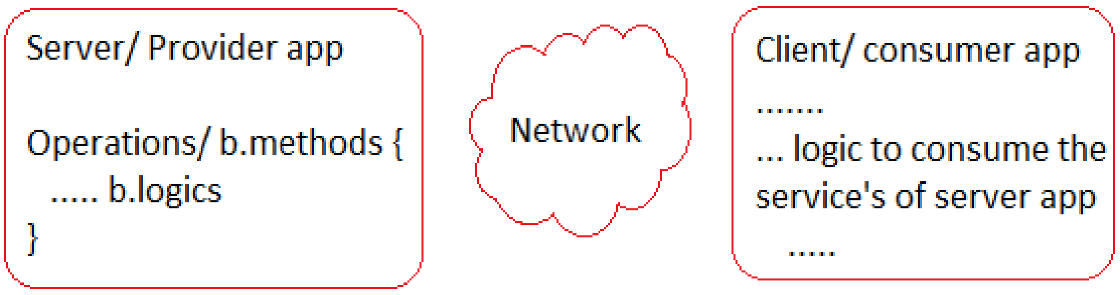
- It allows to develop the consumer/ client app Restful webservice as programmable client application in Java environment.
- We need to take separate Webservice/ MVC project for these having logics to consume webservice/ API by calling methods.
- This object (RestTemplate) does not come through AutoConfiguration process. It must be created either using "new" operator or using @Bean method.

```
RestTemplate template = new RestTemplate ();
                    (or)
```

In @Configuration class

```
@Bean("template")
public RestTemplate createTemplate () {
        return new RestTemplate ();
}
```

- This object provides methods to generate different modes requests like GET/ POST/ PUT/ DELETE and etc., to consume the Restful webservice/ API i.e., we can call methods/ operations of Restful webservice Server app/ provider app.
- While using this object to consume Restful Webservice/ API we need detailed inputs (nothing but end points) like base URL, HTTP method type, HTTP header info like content type and etc.
- It provides xxxForEntity (....) methods like getForEntity (....), postForEntity (....) and etc. taking URL, request object (body, header) to send different modes HTTP requests as method calls to consume the Restful webservice (server/ provider app).
- Don't forget, Webservice is given to link two different apps that are developed either in same language or in different languages and in running same server or different servers belonging to same machine or different machines.



Server/ Provider app		Client/ Consumer app
Payment Gateway app	<------------------->	Paytm
Paytm/ GPay/ PhonePe	<------------------>	Flipkart/ Amazon
Weather Report App	<---------------------->	Yahoo.com/ Tourist.com
ICC Score component	<--------------------->	CrickInfo.com, CrickBuzz.com and etc.

- The Restful webservice (server/ provider app) must be the web application.
- The client/ consumer app can be the standalone app or mobile App or IOT app or web application or etc.
- So far, we have developed only Restful webservice (server/ provider

app) and we tested that server app using tools like POSTMAN/ Swagger.
- ➢ Instead of using these tools we can develop programmable real client apps with the support RestTemplate in Spring environment.
- ➢ RestTemplate can be used only in Spring or Spring Boot environment.

Example application:

Step 1: Develop Restful webservice app (old style app) as web application (server/ provider app).

Step 2: Develop the API nothing but RestController class.

Step 3: Run The application on server (Run As, Run on Server).

Step 4: Develop the Consumer app as separate project.

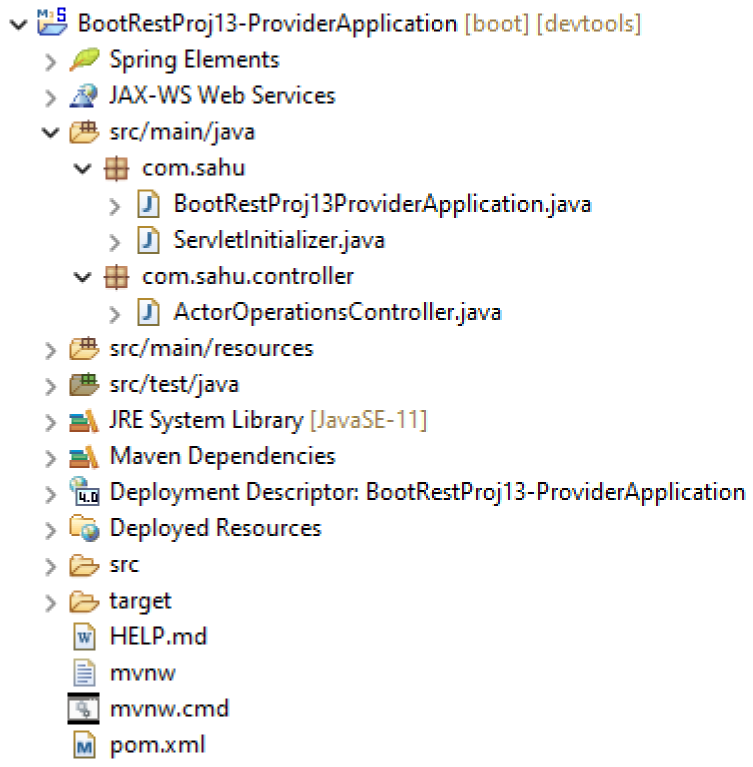
Step 5: Place the following entries in application. properties in consumer app.

```
server.port=4040
```

Step 6: Develop the Runner class in consumer app.

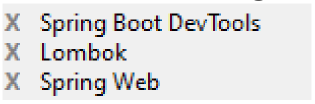
Step 7: Run Consumer app as Spring Boot app that uses Embedded Tomcat server.

Directory Structure of BootRestProj13-ProviderApplication:



```
BootRestProj13-ProviderApplication [boot] [devtools]
  Spring Elements
  JAX-WS Web Services
  src/main/java
    com.sahu
      BootRestProj13ProviderApplication.java
      ServletInitializer.java
    com.sahu.controller
      ActorOperationsController.java
  src/main/resources
  src/test/java
  JRE System Library [JavaSE-11]
  Maven Dependencies
  Deployment Descriptor: BootRestProj13-ProviderApplication
  Deployed Resources
  src
  target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starters during project creation.

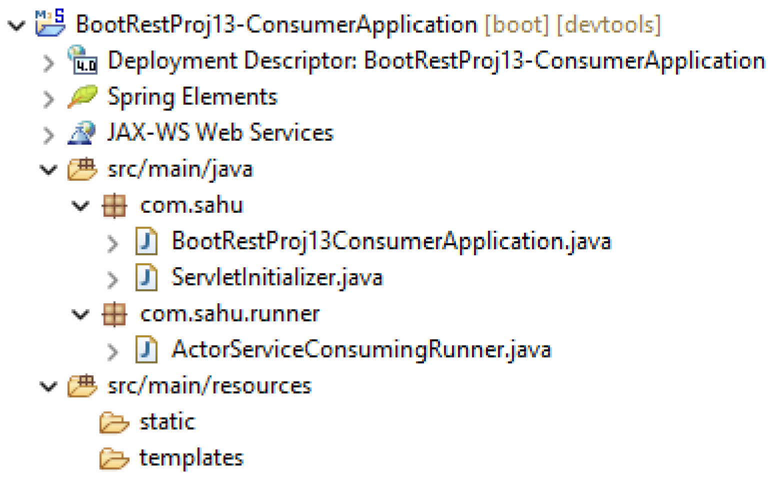

X Spring Boot DevTools
X Lombok
X Spring Web

- Then place the following code with in their respective files.

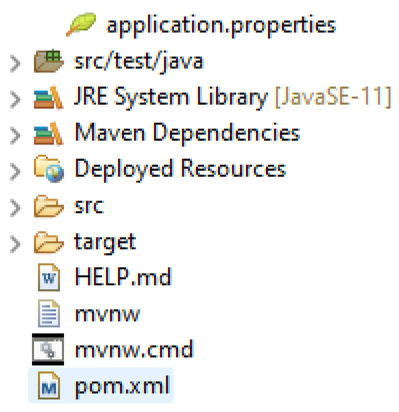
ActorOperationsController.java

```
package com.sahu.controller;

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

@RestController
@RequestMapping("/actor")
public class ActorOperationsController {

	@GetMapping("/wish")
	public ResponseEntity<String> displayWishMessage() {
		return new ResponseEntity<String>("Good morning",
HttpStatus.OK);
	}

}
```

Directory Structure of BootRestProj13-ConsumerApplication:



```
BootRestProj13-ConsumerApplication [boot] [devtools]
  Deployment Descriptor: BootRestProj13-ConsumerApplication
  Spring Elements
  JAX-WS Web Services
  src/main/java
    com.sahu
      BootRestProj13ConsumerApplication.java
      ServletInitializer.java
    com.sahu.runner
      ActorServiceConsumingRunner.java
  src/main/resources
    static
    templates
```



```
    application.properties
> src/test/java
> JRE System Library [JavaSE-11]
> Maven Dependencies
> Deployed Resources
> src
> target
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as war and create the package and class also.
- Use the following starters during project creation.

```
X Spring Boot DevTools
X Lombok
X Spring Web
```

- Then place the following code with in their respective files.

application.properties

```
#Embedded Tomcat port
server.port=4040
```

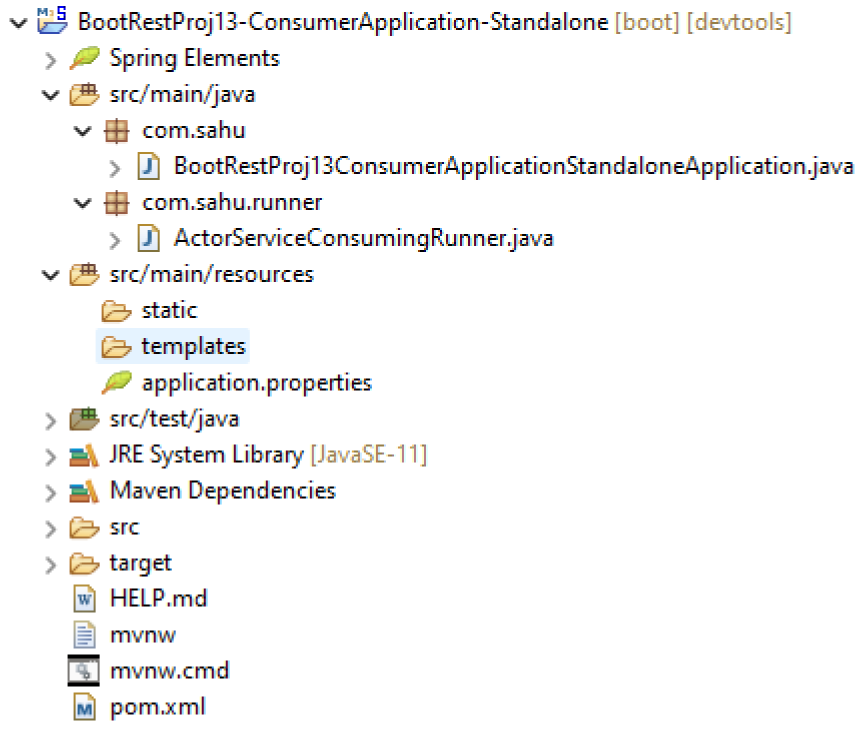
ActorServiceConsumingRunner.java

```java
package com.sahu.runner;

import org.springframework.boot.CommandLineRunner;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class ActorServiceConsumingRunner implements
CommandLineRunner {

	@Override
	public void run(String... args) throws Exception {
		//Create RestTemplate class object
		RestTemplate restTemplate = new RestTemplate();
		//Define URL
		String serviceURL = "http://localhost:3035/BootRestProj13-
ProviderApplication/actor/wish";
```

```
		//Generate HTTP request call with GET mode to consume the
web service  (API).
		ResponseEntity<String> response =
restTemplate.getForEntity(serviceURL, String.class);
		//Display the received details from the response
		System.out.println("Response body (output) -
"+response.getBody());
		System.out.println("Response status code value -
"+response.getStatusCodeValue());
		System.out.println("Response stauts code -
"+response.getStatusCode().name());

		//System.exit(0); //optional
	}

}
```

Note: The above consumer app can also be developed as standalone app (package type is jar) adding Spring web starters as shown below.

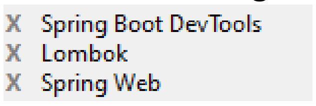
Directory Structure of BootRestProj13-ConsumerApplication-Standalone:



```
BootRestProj13-ConsumerApplication-Standalone [boot] [devtools]
  > Spring Elements
  v src/main/java
    v com.sahu
      > BootRestProj13ConsumerApplicationStandaloneApplication.java
    v com.sahu.runner
      > ActorServiceConsumingRunner.java
  v src/main/resources
      static
      templates
      application.properties
  > src/test/java
  > JRE System Library [JavaSE-11]
  > Maven Dependencies
  > src
  > target
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml
```

- Develop the above directory structure using Spring Starter project option & choose packaging as jar and create the package and class also.

- Use the following starters during project creation.



X Spring Boot DevTools
X Lombok
X Spring Web

- Then place the following code with in their respective files.

application.properties

```
#Embedded Tomcat port
server.port=4041
```

ActorServiceConsumingRunner.java

```
package com.sahu.runner;

import org.springframework.boot.CommandLineRunner;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class ActorServiceConsumingRunner implements
CommandLineRunner {

	@Override
	public void run(String... args) throws Exception {
		//Create RestTemplate class object
		RestTemplate restTemplate = new RestTemplate();
		//Define URL
		String serviceURL = "http://localhost:3035/BootRestProj13-
ProviderApplication/actor/wish";
		//Generate HTTP request call
		ResponseEntity<String> response =
restTemplate.getForEntity(serviceURL, String.class);
		//Display the received details from the response
		System.out.println("Response body (output) -
"+response.getBody());
		System.out.println("Response status code value -
"+response.getStatusCodeValue());
		System.out.println("Response stauts code -
"+response.getStatusCode().name());
	}
}
```

Q. What is the difference b/w getforEntity(..) and getForObject(..) methods?

Ans.

- getForEntity(..) return type is ResponseEntity<T> which contains response body (output), headers, status code and etc.
- getForObject(..) return type is Object which gives only response body (output) i.e., it does not give response headers, status code and etc.

Note:

- ✓ When we develop Spring Web MVC/ Spring Rest app as war file (web application) we can use both external server and embedded server (like Embedded Tomcat) for deployment.
- ✓ When we develop Spring Web MVC/ Spring Rest app as jar file (standalone web application) we can use only embedded server (like Embedded Tomcat) for deployment.

Passing Path variable values along with HTTP request calls using RestTemplate

- Both getForObject(...), getForEntity(...) are having multiple overloaded forms. The form that is taking more params is maintaining last param as var args param which is given to pass any number of path variable values along with http request call.

<T> ResponseEntity<T>	getForEntity(String url, Class<T> responseType, Map<String,?> uriVariables) Retrieve a representation by doing a GET on the URI template. To pass path variable names and values as Map collection and need not follow the order while passing values
<T> ResponseEntity<T>	getForEntity(String url, Class<T> responseType, Object... uriVariables) Retrieve an entity by doing a GET on the specified URL. To pass path variable values as var args and needs follow the order while passing values.

<T> T	getForObject(String url, Class<T> responseType, Map<String,?> uriVariables) Retrieve a representation by doing a GET on the URI template. To pass path variable names and values as Map collection and need not follow the order while passing values.
<T> T	getForObject(String url, Class<T> responseType, Object... uriVariables) Retrieve a representation by doing a GET on the specified URL. To pass path variable values as var args and needs follow the order while passing values.

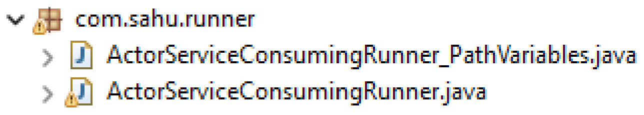
In Server/ Provider application

ActorOperationsController.java

```
@RestController
@RequestMapping("/actor")
public class ActorOperationsController {

	@GetMapping("/wish/{id}/{name}")
	public ResponseEntity<String> displayWishMessage(@PathVariable Integer id, @PathVariable String name) {
		return new ResponseEntity<String>("Good morning "+id+" "+name, HttpStatus.OK);
	}

}
```

In Client/ Consumer application

- In client or consumer application create another runner class as like below.



```
com.sahu.runner
  ActorServiceConsumingRunner_PathVariables.java
  ActorServiceConsumingRunner.java
```

ActorServiceConsumingRunner.java

```
@Component
public class ActorServiceConsumingRunner_PathVariables implements
CommandLineRunner {

	@Override
	public void run(String... args) throws Exception {
		//Create RestTemplate class object
		RestTemplate restTemplate = new RestTemplate();
		//Define URL
		String serviceURL = "http://localhost:3035/BootRestProj13-
ProviderApplication/actor/wish/{id}/{name}";
		//Generate HTTP request call with GET mode to consume the
web service  (API).
		//ResponseEntity<String> response =
restTemplate.getForEntity(serviceURL, String.class, 101, "Raja");
		ResponseEntity<String> response =
restTemplate.getForEntity(serviceURL, String.class, Map.of("name",
"Rajesh", "id", 101));
		//Display the received details from the response
		System.out.println("Response body (output) -
"+response.getBody());
		System.out.println("Response status code value -
"+response.getStatusCodeValue());
		System.out.println("Response stauts code -
"+response.getStatusCode().name());
		System.out.println("Response header -
"+response.getHeaders());
	}

}
```

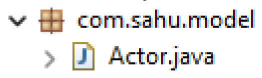
Sending JSON Data from Consumer app along with POST mode request using RestTemplate

- POST mode request contains request body, request headers and initial line whereas GET mode request contains only request headers and initial line.
- Every request structure contains 2 parts
    - HEAD part (initial line + request headers)

  - BODY/ payload part (request body)
- HEAD, BODY of request structure will be separate with blank line.
- Here we need to use postForEntity(..) or postForObject(...) methods of RestTemplate to send post mode request and response completely or partially.

In Server/ Provider application

- In server or provider application create model class as like below



```
com.sahu.model
  Actor.java
```

Actor.java

```
package com.sahu.model;

import lombok.Data;

@Data
public class Actor {
	private Integer aid;
	private String name;
	private Integer age;
	private String type;
}
```

ActorOperationsController.java

```
@RestController
@RequestMapping("/actor")
public class ActorOperationsController {

	@PostMapping("/register")
	public ResponseEntity<String> regiserActor(@RequestBody Actor
actor) {
		return new ResponseEntity<String>("Actor data
"+actor.toString(), HttpStatus.OK);
	}

}
```

In Client/ Consumer application

- In client or consumer application create another runner class.

ActorServiceConsumingRunner_PostingJSONData.java

```java
package com.sahu.runner;

import org.springframework.boot.CommandLineRunner;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class ActorServiceConsumingRunner_PostingJSONData implements
CommandLineRunner {

	@Override
	public void run(String... args) throws Exception {
		//Create RestTemplate class object
		RestTemplate restTemplate = new RestTemplate();
		//Define URL
		String serviceURL = "http://localhost:3035/BootRestProj13-
ProviderApplication/actor/register";
		//Prepare JSON Data (JSON body)
		String json_body = "{\"aid\":1001, \"name\":\"Suresh\",
\"age\":30, \"type\":\"hero\"}";
		//Prepare Headers
		HttpHeaders headers = new HttpHeaders();
		headers.setContentType(MediaType.APPLICATION_JSON);
		//Prepare HTTP request as HttpEntiry object having Head, body
		HttpEntity<String> request = new
HttpEntity<String>(json_body, headers);
		//Make HTTP request call in post mode
		ResponseEntity<String> response =
restTemplate.postForEntity(serviceURL, request, String.class);
		//Display the received details from the response
		System.out.println("Response body (output) -
"+response.getBody());
		System.out.println("Response status code value -
"+response.getStatusCodeValue());
	}
}
```

<T> ResponseEntity<T>	postForEntity(String url, Object request, Class<T> responseType, Map<String,?> uriVariables) Create a new resource by POSTing the given object to the URI template, and returns the response as **HttpEntity**.
<T> ResponseEntity<T>	postForEntity(String url, Object request, Class<T> responseType, Object... uriVariables) Create a new resource by POSTing the given object to the URI template, and returns the response as **ResponseEntity**.
<T> ResponseEntity<T>	postForEntity(URI url, Object request, Class<T> responseType) Create a new resource by POSTing the given object to the URL, and returns the response as **ResponseEntity**.
<T> T	postForObject(String url, Object request, Class<T> responseType, Map<String,?> uriVariables) Create a new resource by POSTing the given object to the URI template, and returns the representation found in the response.
<T> T	postForObject(String url, Object request, Class<T> responseType, Object... uriVariables) Create a new resource by POSTing the given object to the URI template, and returns the representation found in the response.
<T> T	postForObject(URI url, Object request, Class<T> responseType) Create a new resource by POSTing the given object to the URL, and returns the representation found in the response.

Working with exchange () method in RestTemplate

- Instead of calling getForXxx(), postForXxx(), putFoXxx(..), deleteForXxx() and etc. as separate methods to generate different modes of requests we can use single exchange (....) for all operations (One method to generate all modes of request).

```
public <T> ResponseEntity<T> exchange(String url,
                                HttpMethod method,
                                @Nullable
                                HttpEntity<?> requestEntity,
                                Class<T> responseType,
                                Object... uriVariables)
                         throws RestClientException
```

Execute the HTTP method to the given URI template, writing the given request entity to the request, and returns the response as ResponseEntity.

URI Template variables are expanded using the given URI variables, if any.

Specified by:

exchange in interface RestOperations

Parameters:

url - the URL
method - the HTTP method (GET, POST, etc)
requestEntity - the entity (headers and/or body) to write to the request may be null) (header + body)
responseType - the type to convert the response to, or Void.class for no body (required response type)
uriVariables - the variables to expand in the template (path variable values)

Returns:

the response as entity

Throws:

RestClientException

- exchange (....) is alternate for the following methods getForEntity (...), postForEntity (....), getForObject (....), postForObject (....), delete (....), patchForEntity (….), patchForObject (….), put (….) and etc.

Directory Structure of BootRestProj14-ProviderApplication:

- Copy and paste the BootRestProj13-ProviderApplication and rename to BootRestProj14-ProviderApplication.
- After that change the Web Project Setting context root to BootRestProj14-ProviderApplication.
- Then place the following code with in their respective files.

ActorOperationsController.java

```java
package com.sahu.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.sahu.model.Actor;

@RestController
@RequestMapping("/actor")
public class ActorOperationsController {

	@GetMapping("/wish")
	public ResponseEntity<String> displayWishMessage() {
		return new ResponseEntity<String>("Good morning", 
HttpStatus.OK);
	}

	@GetMapping("/wish/{id}/{name}")
	public ResponseEntity<String> displayWishMessage(@PathVariable 
Integer id, @PathVariable String name) {
		return new ResponseEntity<String>("Good morning "+id+" 
"+name, HttpStatus.OK);
	}

	@PostMapping("/register")
```

```
        public ResponseEntity<String> regiserActor(@RequestBody Actor
actor) {
                return new ResponseEntity<String>("Actor data
"+actor.toString(), HttpStatus.OK);
        }

}
```

Directory Structure of BootRestProj14-ConsumerApplication-exchange:

- Copy and paste the BootRestProj14-ConsumerApplication-exchangeand rename to BootRestProj14-ConsumerApplication-exchange.
- After that change the Web Project Setting context root to BootRestProj14-ConsumerApplication-exchange.
- Then place the following code with in their respective files.

ActorServiceConsumingRunner.java

```
package com.sahu.runner;

import org.springframework.boot.CommandLineRunner;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class ActorServiceConsumingRunner implements
CommandLineRunner {

        @Override
        public void run(String... args) throws Exception {
                //Create RestTemplate class object
                RestTemplate restTemplate = new RestTemplate();
                //Define URL
                String wishURL = "http://localhost:3035/BootRestProj14-
ProviderApplication/actor/wish";

                //Invoke service method/ operation using exchange(-,-,-)
method
                ResponseEntity<String> response =
restTemplate.exchange(wishURL, HttpMethod.GET, null, String.class);
		//Display the details
		System.out.println("Response body (output) - 
"+response.getBody());
		System.out.println("Response status code value - 
"+response.getStatusCodeValue());
		System.out.println("Response stauts code - 
"+response.getStatusCode().name());
	}

}
```

ActorServiceConsumingRunner_PathVariables.java

```
package com.sahu.runner;

import java.util.Map;

import org.springframework.boot.CommandLineRunner;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class ActorServiceConsumingRunner_PathVariables implements 
CommandLineRunner {

	@Override
	public void run(String... args) throws Exception {
		//Create RestTemplate class object
		RestTemplate restTemplate = new RestTemplate();
		//Define URL
		String serviceURL = "http://localhost:3035/BootRestProj14-
ProviderApplication/actor/wish/{id}/{name}";
		//Invoke service method/ operation using exchange(-,-,-) 
method
		ResponseEntity<String> response = 
restTemplate.exchange(serviceURL, HttpMethod.GET, null, String.class, 
Map.of("name", "Rajesh", "id", 101));
		//Display the received details from the response
```

```
                System.out.println("Response body (output) -
"+response.getBody());
                System.out.println("Response status code value -
"+response.getStatusCodeValue());
                System.out.println("Response stauts code -
"+response.getStatusCode().name());
                System.out.println("Response header -
"+response.getHeaders());
        }

}
```

ActorServiceConsumingRunner_PostingJSONData.java

```
package com.sahu.runner;

import org.springframework.boot.CommandLineRunner;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class ActorServiceConsumingRunner_PostingJSONData implements
CommandLineRunner {

        @Override
        public void run(String... args) throws Exception {
                //Create RestTemplate class object
                RestTemplate restTemplate = new RestTemplate();
                //Define URL
                String serviceURL = "http://localhost:3035/BootRestProj14-
ProviderApplication/actor/register";
                //Prepare JSON Data (JSON body)
                String json_body = "{\"aid\":1001, \"name\":\"Ranveer\",
\"age\":30, \"type\":\"hero\"}";
                //Prepare Headers
                HttpHeaders headers = new HttpHeaders();
```

```
                    headers.setContentType(MediaType.APPLICATION_JSON);
                    //Prepare HTTP request as HttpEntiry object having Head, body
                    HttpEntity<String> entity = new HttpEntity<String>(json_body,
headers);
                    //Make HTTP request call in post mode
                    ResponseEntity<String> response =
restTemplate.exchange(serviceURL, HttpMethod.POST, entity, String.class);
                    //Display the received details from the response
                    System.out.println("Response body (output) -
"+response.getBody());
                    System.out.println("Response status code value -
"+response.getStatusCodeValue());
                    System.out.println("Response stauts code -
"+response.getStatusCode().name());
          }

}
```

- First run the server/ provider application as run as server.
- Then run the client/ consumer application as run as Spring Boot application.

Note: Instead of creating Multiple objects for RestTemplate class in different runner classes of Consumer app, create RestTemplate class object in main class using @Bean method and inject the Spring bean object in runner classes with the support @Autowired annotation.

Receiving and processing different return type values to webservice methods/ operations using RestTemplate

- If webservice method/ operation return type is ResponseEntity<String>/ String then we get plain text as the response.
- If webservice method/ operation return type is other than String/ ResponseEntity<String> then we get JSON response as text content by default.
- This JSON text content can be used directly or can be converted to Java object using JACKSON API (built-in Spring Boot Rest application).
- Once we add spring MVC starter to the project we get JACSON API automatically we can be used to JSON data to object (Deserialization) and object to JSON data (Serialization).

- Irrespective of the return of webservice method/ operation (String or non-String) we generally take the return value of exchange (....) or getForXxx(...) or postForXxx(...) and etc. methods as String based return value like ResponseEntity<String> object. i.e., receive every thing from provider app as String content and convert it to other types if needed.
- The ObjectMapper class of Jackson API gives methods for Serialization and Desterilization.
    a) mapper.readValue(-): JSON text to Object (Deserialization)
    b) mapper.writeValue(-): Object to JSON text (Serialization)

In Server/ Provider application

- In server or provider application place the following code in their respective files.

Actor.java

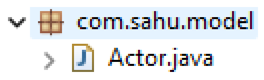
```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Actor {
        private Integer aid;
        private String name;
        private Integer age;
        private String type;
}
```

ActorOperationsController.java

```
        @GetMapping("/find")
        public ResponseEntity<Actor> searhActor() {
                return new ResponseEntity<Actor>(new Actor(200, "Salaman",
56, "Hero"), HttpStatus.OK);
        }
```

In Client/ Consumer application

- In client or consumer application create another runner class and copy and paste the Actor.java class from Provide application including package.
- And place the following code with their respective files.


com.sahu.model
Actor.java

ActorServiceConsuming_Getting_JSONData_Runner.java

```
package com.sahu.runner;

import org.springframework.boot.CommandLineRunner;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.sahu.model.Actor;

@Component
public class ActorServiceConsuming_Getting_JSONData_Runner
implements CommandLineRunner {

	@Override
	public void run(String... args) throws Exception {
		//Create RestTemplate class object
		RestTemplate restTemplate = new RestTemplate();
		//Define URL
		String wishURL = "http://localhost:3035/BootRestProj14-
ProviderApplication/actor/find";

		//Invoke service method/ operation using exchange(-,-,-)
method
		ResponseEntity<String> response =
restTemplate.exchange(wishURL, HttpMethod.GET, null, String.class);

		//Display the details
		System.out.println("Response body (output) -
"+response.getBody());
		System.out.println("Response status code value -
"+response.getStatusCodeValue());
		System.out.println("Response stauts code -
"+response.getStatusCode().name());
		System.out.println("-----------------------------------");

		//Convert JSON Text response (body) to Java class object/
model class object/ Entity class object using JACSON API
```

```
                String jsonBody = response.getBody();
                //Create ObjectMapper
                ObjectMapper mapper = new ObjectMapper();
                Actor actor = mapper.readValue(jsonBody, Actor.class);
                System.out.println(actor);
        }

}
```

Converting List of objects (1D Array) data of JSON given By Server/ Producer/ Provider app into List of objects in Consumer app using Jackson API

In Server/ Provider application

- In server or provider application place the following code in their respective files.

ActorOperationsController.java

```
        @GetMapping("/findAll")
        public ResponseEntity<List<Actor>> fetchAllAtors() {
                return new ResponseEntity<List<Actor>>(List.of(
                                new Actor(101, "Salman", 55, "Hero"),
                                new Actor(102, "Rajesh", 65, "ero"),
                                new  Actor(103, "Raveen", 35, "Hero")),
                                HttpStatus.OK);
        }
```

In Client/ Consumer application

- In client or consumer application place the following code with their respective files.

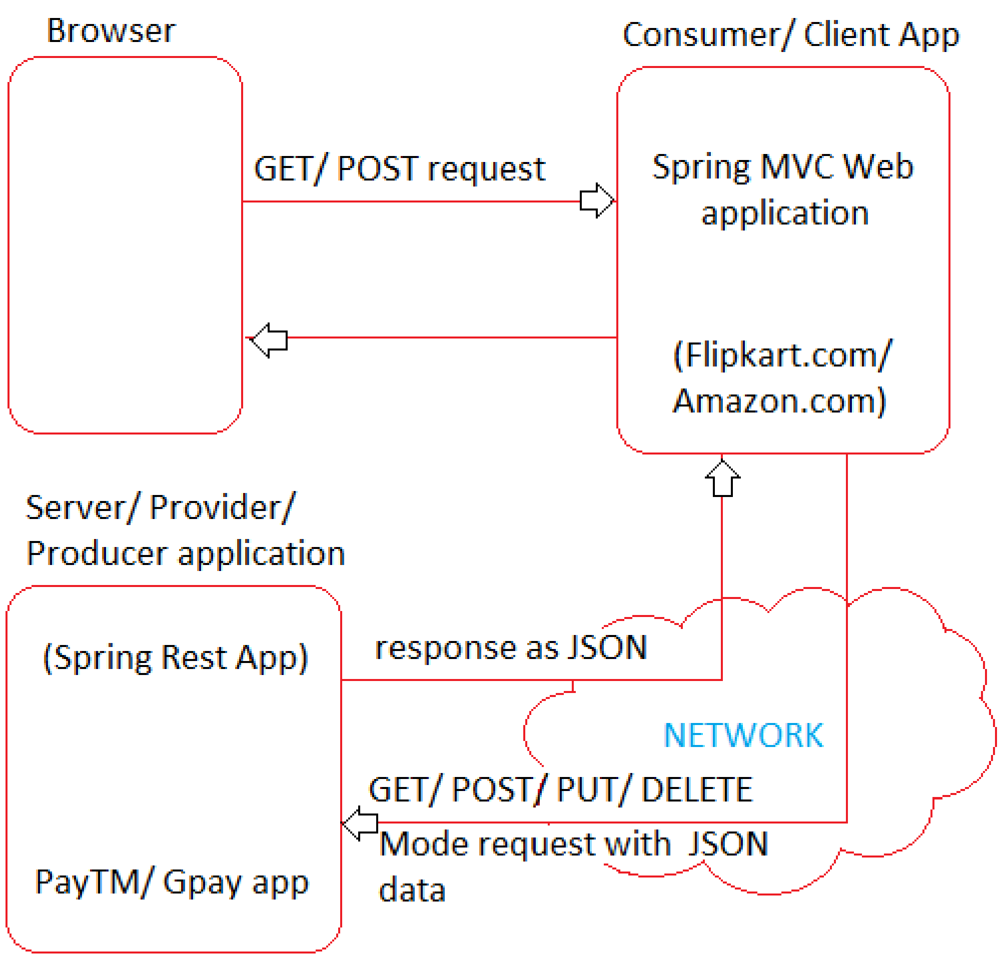
ActorServiceConsuming_Getting_JSONData_Runner.java

```
package com.sahu.runner;

import java.util.Arrays;
import java.util.List;

import org.springframework.boot.CommandLineRunner;
import org.springframework.http.HttpMethod;
import org.springframework.http.ResponseEntity;
```

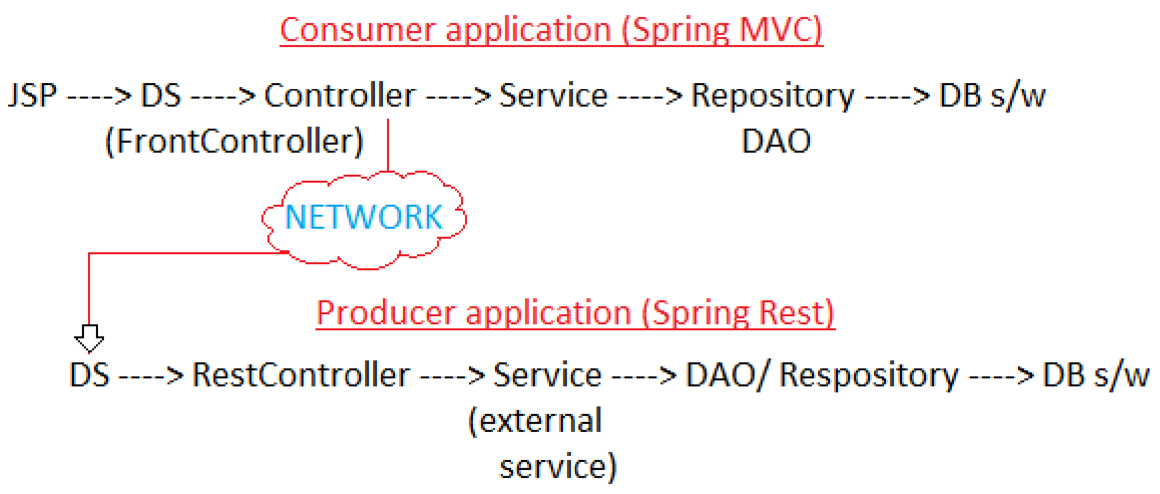
```java
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.sahu.model.Actor;

@Component
public class ActorServiceConsuming_Getting_JSONData_Runner
implements CommandLineRunner {

	@Override
	public void run(String... args) throws Exception {
		//Create RestTemplate class object
		RestTemplate restTemplate = new RestTemplate();
		//Define URL
		String serverURL = "http://localhost:3035/BootRestProj14-
ProviderApplication/actor/findAll";

		//Invoke service method/ operation using exchange(-,-,-)
method
		ResponseEntity<String> response =
restTemplate.exchange(serverURL, HttpMethod.GET, null, String.class);

		//Display the details
		System.out.println("Response body (output) -
"+response.getBody());
		System.out.println("Response status code value -
"+response.getStatusCodeValue());
		System.out.println("Response stauts code -
"+response.getStatusCode().name());
		System.out.println("-----------------------------------");

		//Convert JSON Text response (body) to Java class object/
model class object/ Entity class object using JACSON API
		String jsonBody = response.getBody();
		//Create ObjectMapper
		ObjectMapper mapper = new ObjectMapper();
		Actor[] actor = mapper.readValue(jsonBody, Actor[].class);
		List<Actor> listActor = Arrays.asList(actor);
		System.out.println(listActor);
		System.out.println("------------------------------------");
		List<Actor> listActors = mapper.readValue(jsonBody, new
TypeReference<List<Actor>>() {});
		listActors.forEach(System.out::println);
	}

}
```

Note: TypeReference is Jackson API supplied abstract class is used for obtaining full generics type information by sub-classing.

Consumer application for Mini Project


Browser
Consumer/ Client App
GET/ POST request
Spring MVC Web application
(Flipkart.com/ Amazon.com)
Server/ Provider/ Producer application
(Spring Rest App)
PayTM/ Gpay app
response as JSON
NETWORK
GET/ POST/ PUT/ DELETE Mode request with JSON data

- The distributed components are also called external component/ external services because they will not part of Consumer app/ project. They always reside outside the of consumer project/ app giving services to Consumer application.


Consumer application (Spring MVC)
JSP ----> DS ----> Controller ----> Service ----> Repository ----> DB s/w
(FrontController)
DAO
NETWORK
Producer application (Spring Rest)
DS ----> RestController ----> Service ----> DAO/ Respository ----> DB s/w
(external service)

Q. What is the difference b/w put (-) and exchange (-, -, -, -) of RestTemplate?

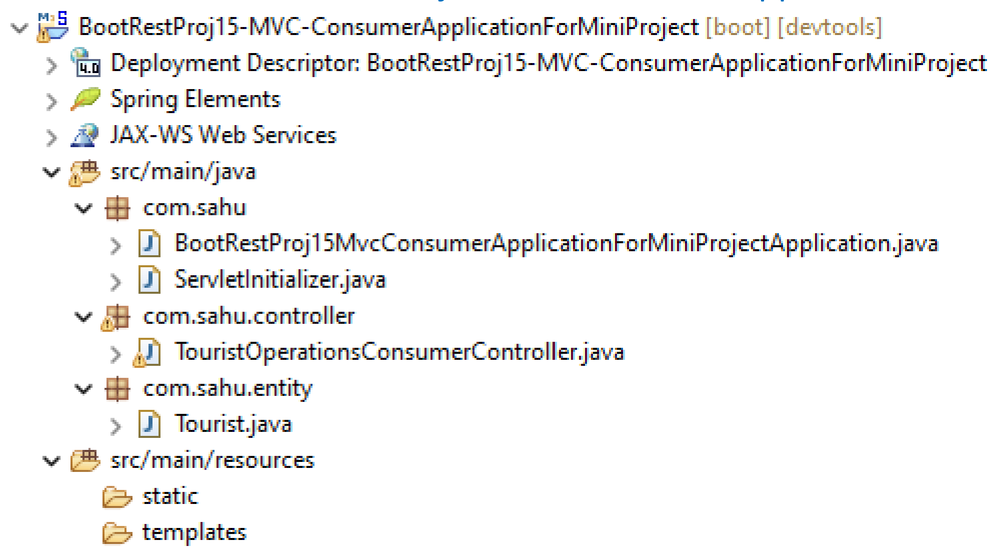
Ans.

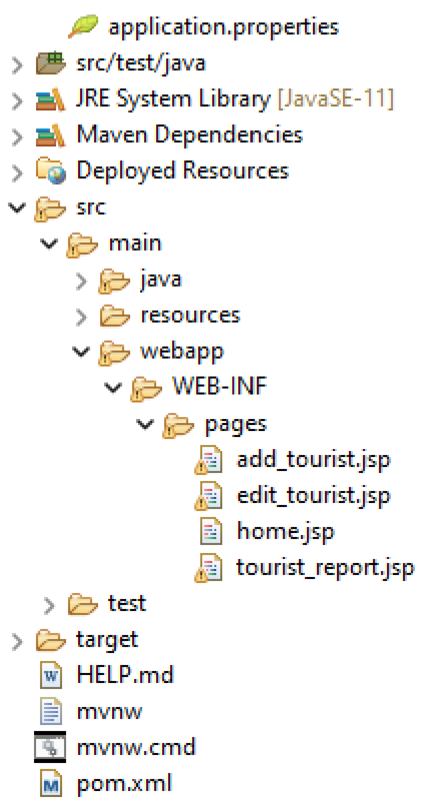
- put (-) method can send only PUT mode request to producer app whereas exchange(...) can send different modes of requests.
- put (-) method return type is void i.e., we cannot get response body given by producer app whereas exchange (....) return type is ResponseEntity<T> i.e., we can get result/ response given by provider app.

Q. What is the difference b/w delete (-) and exchange (-, -, -, -) of RestTemplate?

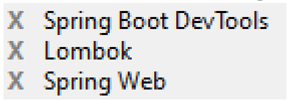
Ans. Same as above.

Directory Structure of BootRestProj15-MVC-ConsumerApplicationForMiniProject:


```
BootRestProj15-MVC-ConsumerApplicationForMiniProject [boot] [devtools]
  > Deployment Descriptor: BootRestProj15-MVC-ConsumerApplicationForMiniProject
  > Spring Elements
  > JAX-WS Web Services
  v src/main/java
    v com.sahu
      > BootRestProj15MvcConsumerApplicationForMiniProjectApplication.java
      > ServletInitializer.java
    v com.sahu.controller
      > TouristOperationsConsumerController.java
    v com.sahu.entity
      > Tourist.java
  v src/main/resources
      static
      templates
```


```
        application.properties
>   src/test/java
>   JRE System Library [JavaSE-11]
>   Maven Dependencies
>   Deployed Resources
v   src
    v   main
        >   java
        >   resources
        v   webapp
            v   WEB-INF
                v   pages
                        add_tourist.jsp
                        edit_tourist.jsp
                        home.jsp
                        tourist_report.jsp
    >   test
>   target
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml
```

- Develop the above directory structure using Spring Starter Project option & and create the package, class, folder and files.
- Use the following starter during project creation.



```
X  Spring Boot DevTools
X  Lombok
X  Spring Web
```

- Add JSTL jar dependency in pom.xml
- Then place the following code with in their respective files.

application.properties

```
#View Resolver Configuration
spring.mvc.view.prefix=/WEB-INF/pages/
spring.mvc.view.suffix=.jsp

#Embedded tomat port
server.port=4040
```

home.jsp

```
<h1 style="text-align: center;">
        <a href="list_tourist">List All Tourist</a>
</h1>
```

tourist_report.jsp

```
<%@ page isELIgnored="false" %>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<c:choose>
	<c:when test="${!empty touristList}">
		<h1 style="color: green; text-align: center;">The Tourists 
Info</h1>
		<table  border="1" align="center" bgcolor="cyan">
			<tr>
				<th>TID</th>
				<th>Name</th>
				<th>City</th>
				<th>Package Type</th>
				<th>Budget</th>
				<th>Operations</th>
			</tr>
			<c:forEach var="tst" items="${touristList}">
				<tr>
					<td>${tst.tid}</td>
					<td>${tst.name}</td>
					<td>${tst.city}</td>
					<td>${tst.packageType}</td>
					<td>${tst.budget}</td>
					<td><a 
href="edit?id=${tst.tid}">Edit</a></td>
					<td><a 
href="edit?id=${tst.tid}">Delete</a></td>
				</tr>
			</c:forEach>
		</table>
	</c:when>
	<c:otherwise>
		<h1 style="text-align: center; color: red;">Tourist not 
found</h1>
	</c:otherwise>
</c:choose>
<br>
<h2 style="text-align: center;"><a href="add">Add Tourist</a></h2>
<br><br>
```

add_tourist.jsp

```
<%@ page isELIgnored="false" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="frm"
%>

<h1 style="color: red; text-align: center;">Add Tourist</h1>
<frm:form modelAttribute="tst">
        <table border="0" align="center" bgcolor="cyan">
                <tr>
                        <td>Tourist Name: </td>
                        <td><frm:input path="name"/></td>
                </tr>
                <tr>
                        <td>City: </td>
                        <td><frm:input path="city"/></td>
                </tr>
                <tr>
                        <td>Package Type: </td>
                        <td><frm:input path="packageType"/></td>
                </tr>
                <tr>
                        <td>Budget: </td>
                        <td><frm:input path="budget"/></td>
                </tr>
                <tr>
                        <td><input type="submit" value="Register
Tourist"/></td>
                </tr>
        </table>
</frm:form>
```

edit_tourist.jsp

```
<%@ page isELIgnored="false" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="frm"
%>

<h1 style="color: red; text-align: center;">Edit Tourist</h1>
<frm:form modelAttribute="tst">
        <table border="0" align="center" bgcolor="cyan">
```

```
                <tr>
                        <td>Tourist Id: </td>
                        <td><frm:input path="tid" readonly="true"/></td>
                </tr>
                <tr>
                        <td>Tourist Name: </td>
                        <td><frm:input path="name"/></td>
                </tr>
                <tr>
                        <td>City: </td>
                        <td><frm:input path="city"/></td>
                </tr>
                <tr>
                        <td>Package Type: </td>
                        <td><frm:input path="packageType"/></td>
                </tr>
                <tr>
                        <td>Budget: </td>
                        <td><frm:input path="budget"/></td>
                </tr>
                <tr>
                        <td><input type="submit" value="Register
Tourist"/></td>
                </tr>
        </table>
</frm:form>
```

BootRestProj15MvcConsumerApplicationForMiniProjectApplication.java

```
@SpringBootApplication
public class
BootRestProj15MvcConsumerApplicationForMiniProjectApplication {

        @Bean
        public RestTemplate createRestTemplate() {
                return new RestTemplate();
        }

        public static void main(String[] args) {
```

Tourist.java

```java
package com.sahu.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Tourist {
	private Integer tid;
	private String name;
	private String city;
	private String packageType;
	private Double budget;
}
```

TouristOperationsConsumerController.java

```java
package com.sahu.controller;

import java.util.List;
import java.util.Map;

import org.springframework.beans.BeanUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import com.fasterxml.jackson.core.type.TypeReference
import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.sahu.entity.Tourist;

@Controller
public class TouristOperationsConsumerController {

	@Autowired
	private RestTemplate template;

	@GetMapping("/")
	public String showHome() {
		return "home";
	}

	@GetMapping("/list_tourist")
	public String displayTouristReport(Map<String, Object> map) throws
Exception {
		//Consume Spring Rest service method/ operation
		/*
		 * URL/URI : http://localhost:3035/BootRestProj10-
MiniProject01-CURDOperations/tourist/findAll
		 * Method : GET
		 * Response Content Type : application/json (default)
		 * No Path variables/ No Query String
		 */
		String serviceURL = "http://localhost:3035/BootRestProj10-
MiniProject01-CURDOperations/tourist/findAll";
		ResponseEntity<String> response
=template.exchange(serviceURL, HttpMethod.GET, null, String.class);
		String jsonBody = response.getBody();
		//Convert String jsonBody to List<Tourist>
		ObjectMapper mapper = new ObjectMapper();
		List<Tourist> touristList = mapper.readValue(jsonBody, new
TypeReference<List<Tourist>>() {});
		//Keep results in map collection
		map.put("touristList", touristList);
		return "tourist_report";
	}
	//To Launch Form page
	@GetMapping("/add")
	public String show_registerTouristForm(@ModelAttribute("tst")
Tourist tourist) {
		return "add_tourist";
	}

	@PostMapping("/add")
	public String registerTourist(RedirectAttributes attributes,
@ModelAttribute("tst") Tourist tourist) throws Exception {
		//Convert object to JSON data using Jackson API
		ObjectMapper mapper = new ObjectMapper();
		String jsonData = mapper.writeValueAsString(tourist);
		//invoke Spring Rest Service
		String serviceURL = "http://localhost:3035/BootRestProj10-
MiniProject01-CURDOperations/tourist/register";
		//Prepare HttpEntity object (header+body)
		HttpHeaders headers = new HttpHeaders();
		headers.setContentType(MediaType.APPLICATION_JSON);
		HttpEntity<String> entity = new HttpEntity<String>(jsonData,
headers);
		ResponseEntity<String> response
=template.exchange(serviceURL, HttpMethod.POST, entity, String.class);
		String result = response.getBody();
		attributes.addFlashAttribute("resultMsg", result);
		return "redirect:list_tourist";
	}

	@GetMapping("/edit")
	public String showEditFormPage(@RequestParam("id") Integer tid,
@ModelAttribute("tst") Tourist tourist) throws Exception {
		//Invoke rest service
		String serviceURL = "http://localhost:3035/BootRestProj10-
MiniProject01-CURDOperations/tourist/find/{id}";
		ResponseEntity<String> response
=template.exchange(serviceURL, HttpMethod.GET, null, String.class, tid);
		//get JSON body from response
		String jsonBody = response.getBody();
		//Convert jsonBody to tourist object using ObjectMapper
		ObjectMapper mapper = new ObjectMapper();
```

```
			Tourist tourist1 = mapper.readValue(jsonBody, Tourist.class);
			BeanUtils.copyProperties(tourist1, tourist);
			return "edit_tourist";
	}

	@PostMapping("/edit")
	public String editTourist(RedirectAttributes attributes,
@ModelAttribute("tst") Tourist tourist) throws Exception {
			//Convert object to JSON data using Jackson API
			ObjectMapper mapper = new ObjectMapper();
			String jsonData = mapper.writeValueAsString(tourist);
			//invoke Spring Rest Service
			String serviceURL = "http://localhost:3035/BootRestProj10-
MiniProject01-CURDOperations/tourist/modify";
			//Prepare HttpEntity object (header+body)
			HttpHeaders headers = new HttpHeaders();
			headers.setContentType(MediaType.APPLICATION_JSON);
			HttpEntity<String> entity = new HttpEntity<String>(jsonData,
headers);
			ResponseEntity<String> response
=template.exchange(serviceURL, HttpMethod.PUT, entity, String.class);
			String result = response.getBody();
			attributes.addFlashAttribute("resultMsg", result);
			return "redirect:list_tourist";
	}

}
```

- Run the Mini project then run this application.

--- The END ---