⇒ Spring REST course

Web services are reusable software components which are available on the network for consumption by other applications, irrespective of their platform and technology. Web Services can either be SOAP or REST based. And, REST has become more popular in recent days.

REST stands for REpresentation State Transfer
(architectural design that define rules to create web services)
↳ lightweight, maintainable and scalable

↳ advantages

- Interoperability
- Resource sharing
- Reusability
- Independent client-server architecture
- Stateless transactions

Note:- There are plenty of communications that happens b/w different platform in different languages, and these communications happens with the help of web services.

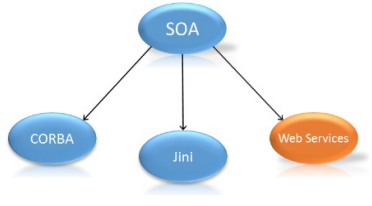
These communications are happening only because the functionalities that one application is trying to consume from other is exposed as web service.

→ SOA (Service Oriented Architecture)

It is a software model designed for achieving communication among distributed application components, irrespective of the differences in terms of technology, platform etc.

SOA takes care of:-

Principle	Explanation
Loose coupling	SOA aims at structuring procedures or software components as services. And, these services are designed in such a way that they are loosely coupled with the applications. So, these loosely-coupled services will come into picture only when needed.
Publishing the services	There should be a mechanism for publishing the services that include the functionality and input/output specifications of those services. In short, services are published in such a way that the developers can easily incorporate them into their applications.
Management	Finally, software components should be managed so that there will be a centralized control on the usage of the published services.



CORBA: Common Object Request Broker Architecture is a standard that achieves interoperability among the distributed objects. Using CORBA, it is much possible to make the distributed application components communicate with each other irrespective of the location (the place where the components are available), language and platform.

Jini: Jini or Apache River is a way of designing distributed applications with the help of services that interact and cooperate.

Web Services: Web services are a way of making applications interact with each other on the web. Web services leverage the existing protocols and technologies but, uses certain standards.

Web services

"A web service is a software system designed to support interoperable machine-to-machine interaction over a network".

- Web services are client-server applications that communicate (mostly) over Hyper Text Transfer Protocol (HTTP)
- Web services are a standard means of achieving interoperability between software applications running on diverse platforms and frameworks.
- Web Services are highly extensible. Services can be loosely coupled to solve complex operations.
- Web Service is a way of realizing Service Oriented Architecture (SOA). SOA is a style of software design where services are provided by the application components through a communication protocol. The basic principle of SOA makes it possible to be loosely-coupled with Vendor, product, and technology. As per SOA, a service is an individual unit of functionality that can be accessed remotely and updated independently, for example, retrieving a banking card statement online.

↳ difference b/w web applications & web services

Web Application	Web Services
A web application is meant for humans	Web Services are used to exchange data between two incompatible applications
The Web application is a complete application with GUI(Graphical User Interface)	Web Services do not necessarily have a user interface since it is used as a component in an end to end application
A typical web application, for example, a Java web application can be accessed by Java Clients (Web/Desktop) alone. And, a web application is generally accessed through browsers	Web Services can be accessed by applications of different languages and platforms
Reusing the functionalities of a web application is difficult	Reusing the functionalities that are exposed as web services is highly possible

↳ common benifits of web services

↳ Interoperability : Integrating with other system easily irrespective of platform or technology

↳ Reusability : Creates reusable components

↳ Ease of use : Save costs, effort and time

↳ Types of web services

↳ SOAP (Simple Object Access Protocol)

↳ RESTful

SOAP-based Web Services are **described, discovered and accessed** using the standards that are recommended by W3C.

RESTful Web Services are REST architecture based ones. As per REST architecture, everything is a resource. A resource (data/functionality/web page) is an object that has a specific type and associated data. Also, it has relationships to other resources and a well-defined set of methods that operate on it. RESTful web services are light-weight, scalable, maintainable and leverage the HTTP protocol to the maximum.

SOAP :- it defines a very strongly typed messaging framework that relies heavily on XML and schemas.

WSDL :- Services are described using WSDL (web services description language)

UDDI :- Universal description, discovery, and Integration is a registery based on XML, where services can be discovered/registered.

↳ RESTful web services (fast, scalable and modifiable)
(it is an architectural style for developing web services which uses HTTP for communication)
In REST everything is Resource, that is uniquely identified by URI (uniform Resource Identifier)

Resource	URI
Data of an employee	http://www.infy.com/employees/john
Web Page	http://www.infy.com/about/infosys.html
Functionality that validates a credit card	http://www.infy.com/creditcards/1000001

↳ REST can return data in many formats like JSON, XML etc.

What does state transfer means?

We can use the same URI to indicate that, we want to :

- Retrieve the current value of an Employee John (current state) (GET)
- Perform Update operation on employee John's data (desired state) etc. (PUT)

Note:- It means that a client and server exchange, representation of a resource, which reflects its current state (GET) or its desired state (PUT).

→ both SOAP and REST provide support for building applications based on SOA

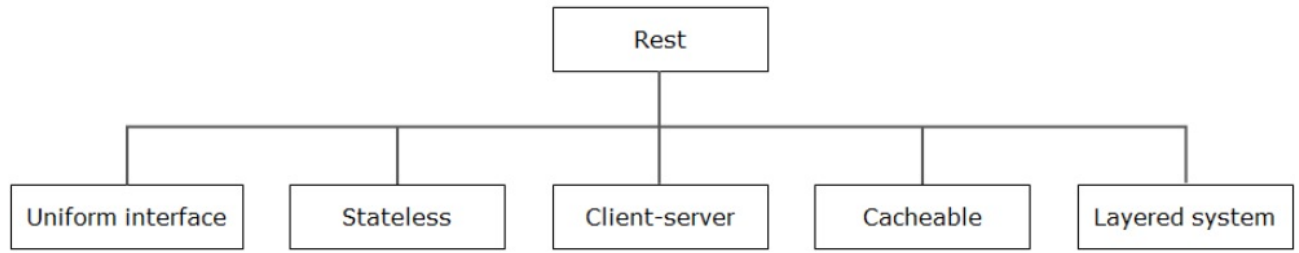
SOAP-based Web Service	RESTful Web Service
SOAP is a protocol (defines the way how messages have to be sent) which is transported over a transport layer Protocol, mostly HTTP.	REST is an architectural style.
SOAP messages can be transported using other transport layer protocols (SMTP, FTP) as well.	REST leverages HTTP Protocol.
Data (XML/JSON) wrapped in a SOAP message, comes with an additional overhead of XML processing at the client and server side as well.	REST is straight forward in handling requests and responses. Since REST supports multiple data formats other than XML, there is no overhead of wrapping the request and response data in XML. Also, REST uses the existing HTTP protocol for communication. All these factors together make REST, lightweight.
SOAP-based reads cannot be cached because SOAP messages are sent using the HTTP POST method.	REST-based reads can be cached.
A predefined formal contract is required between the consumer and the provider.	Formal contract is not mandatory. Service description can be done if needed, using a less verbose way with the help of WADL.

So eventually, if we want to develop an application which

- is light-weight
- can be accessed by any other application irrespective of the differences in terms of language and platform.
- leverages HTTP protocol
- is stateless
- does not require a specific message format to carry the request and response (like SOAP)
- represents data using various MIME types which all applications can consume

then, **RESTful** Web Service is the choice.

(Rest is fast, modifiable and scalable bcz of)



→ Set of architecture principles of REST

↳ Uniform Interface

1) In REST, data and functionality are considered as resource. Every resource has a representation (XML/JSON) and URI to identify the same.
2) Resources are manipulated using HTTP's POST, GET, DELETE, PUT

adv.:- Single URI can perform multiple operations on resource.

↳ Stateless

1) The interaction b/w service provider and consumer is stateless
2) Stateless means server will never store the info. of its client and every request should carry all the necessary info

adv:- stateless ensures every request is independent and this makes the applicat^{n} scalable, less complex and efficient

↳ Client Server

Enforces the seperation of concern helps to establise distributed architecture means change in one state will not affect the other state.
means it is loosely coupled.

adv:- supports the independent evolution of the client and server side logic

↳ Cachable

1) Service consumer can cache the response data and can be reused (e.g., HTTP, HEAD, GET)
2) RESTful webservices uses HTTP to carry the response data

adv:- Enhance performance

↳ Layered System

1) REST based solution comprises of multiple architecture layer
2) The consumer may interact directly with actual provider or through the service that is residing in the middleware layer.

adv:- Simplified information hiding and independent layer evolution

↳ Code On demand

1) This is an optional constraint which is primarily intended to allow client side logic to be modified without effecting server side logic
2) a service can be designed in such a way that it dynamically pushes part of the logic to the consumer.

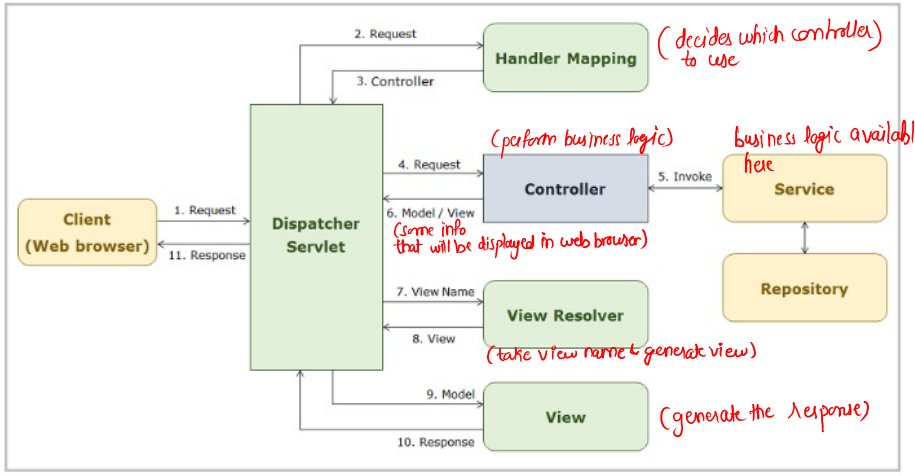
adv:- logic in client end can be modified/maintained independently without affecting server side logic

⇒ Spring REST - introduction

Spring is an end to end framework which has a lot of modules in an highly organised way.

Spring web module carries the support for REST as well.

↳ Spring MVC architecture



Note: By default, DispatcherServlet supports GET, HEAD, POST, PUT, PATCH and DELETE HTTP methods only.

For this, Spring 3.0 introduced @ResponseBody annotation. So, the methods of REST controllers that are annotated with @ResponseBody tells the DispatcherServlet that the result of execution need not to be mapped with a view.

Note:- @ResponseBody annotation automatically converts the response to JSON string literal by applying serialization on the return value of the method.

Note:- In Spring 4.0, @RestController annotation was introduced and

@RestController = @Controller + @ResponseBody

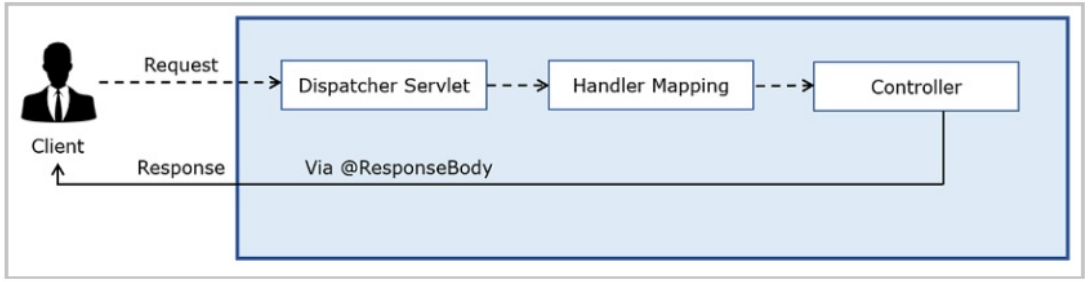
This annotation when used on a REST controller class bounds all the values returned by controller methods to the response body.

↳ difference b/w

↳ Spring REST :- data is retured back directly to the client

↳ Spring MVC :- response is View/Page by default

⇒ Spring REST using Spring Boot



Here, view resolver have no role to play

A class which needs to be exposed as RESTful resource has annotation @RestController

@RestController

- This annotation is used to create REST controllers.
- It is applied on a class in order to mark it as a request handler/REST resource.
- This annotation is a combination of @Controller and @ResponseBody annotations.
- @ResponseBody is responsible for the automatic conversion of the response to a JSON string literal. If @Restcontroller is in place, there is no need to use @ResponseBody annotation to denote that the Service method simply returns data, not a view.
- @RestController is an annotation that takes care of instantiating the bean and marking the same as REST controller.
- It belongs to the package, org.springframework.web.bind.annotation.RestController

@RequestMapping

- This annotation is used for mapping web requests onto methods that are available in the resource classes. It is capable of getting applied at both class and method levels. At method level, we use this annotation mostly to specify the HTTP method.

URI	HTTP Method	CustomerController method	Method description	Annotation to be applied at the method level	New Annotation that can be applied instead of @RequestMapping at the method level
/customers	GET	fetchCustomer()	Will fetch all the customers of Infytel App and return the same.	@RequestMapping(method = RequestMethod.GET)	@GetMapping
/customers	POST	createCustomer()	Will create a new customer	@RequestMapping(method = RequestMethod.POST)	@PostMapping
/customers	DELETE	deleteCustomer()	Will delete an existing customer	@RequestMapping(method = RequestMethod.DELETE)	@DeletMapping
/customers	UPDATE	updateCustomer()	Will update the details of an existing customer	@RequestMapping(method = RequestMethod.PUT)	@PutMapping

@SpringBootApplication is a convenient annotation that adds the following:

- @Configuration marks the class as a source where bean definitions can be found for the application context.
- @EnableAutoConfiguration tells Spring Boot to start adding beans based on the classpath and property settings.
- @ComponentScan tells Spring to look for other components, configurations, and, services in the packages being specified.

Example for @RestController, and all other HTTP operations

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

    //Fetching the customer details
    @GetMapping
    public String fetchCustomer()
    {
        //This method will fetch the customers of Infytel and return the same.
        return "customers fetched successfully";
    }

    //Adding a new customer
    @PostMapping
    public String createCustomer()
    {
        //This method will persist the details of a customer
        return "Customer added successfully";
    }


    //Updating an existing customer
    @PutMapping
    public String updateCustomer()
    {
        //This method will update the details of an existing customer
        return "customer details updated successfully";
    }


    //Deleting a customer
    @DeleteMapping
    public String deleteCustomer()
    {
        //This method will delete a customer
        return "customer details deleted successfully";
    }
}
```

⇒ Request Handling in Spring REST

↳ @RequestBody

@RequestBody is the annotation that helps map the HTTP request body to a Java Data Transfer Object(DTO). And, this annotation has to be applied on the local parameter (Java DTO) of the request method.

Ex1

```
@PostMapping
public String  createCustomer( @RequestBody CustomerDTO customerDTO)
{
        // logic goes here
}
```

Ex2

```
@PostMapping(consumes="application/json")
public ResponseEntity<String> createCustomer( @RequestBody CustomerDTO customerDTO)
{
    // logic goes here
}
```

How to send Java object in Response

Eg1

```
@GetMapping
public  List<CustomerDTO> fetchCustomer()
{
    //business logic goes here
    return customerService.fetchCustomer();
}
```

Eg2

```
@GetMapping(produces="application/json")
public  List<CustomerDTO> fetchCustomer()
{
    //This method will return the customers of Infytel
    return customerService.fetchCustomer();
}
```

↳ ResponseEntity

While sending a response, set the HTTP status code and headers .To help achieving this, we can use ResponseEntity class.

ResponseEntity<T> Will help us add a HttpStatus status code and headers to the response.

Ex1

```
@PostMapping(consumes="application/json")
public ResponseEntity<String> createCustomer(@RequestBody CustomerDTO customerDTO)
{
    //This method will create a customer
    String response = customerService.createCustomer(customerDTO);
    return ResponseEntity.ok(response);
}
```

Ex2

```
@PostMapping(consumes="application/json")
public ResponseEntity<String> createCustomer(@RequestBody CustomerDTO customerDTO)
{
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.set("MyResponseHeaders", "Value1");
    String response = customerService.createCustomer(customerDTO);
    return new ResponseEntity<String>(response, responseHeaders, HttpStatus.CREATED);
}
```

Constructor	Description
ResponseEntity(HttpStatus status)	Creates a ResponseEntity with only status code and no body
ResponseEntity(MultiValueMap<String,String> headers, HttpStatus status)	Creates a ResponseEntity object with headers and statuscode but, no body
ResponseEntity(T body, HttpStatus status)	Creates a ResponseEntity with a body of type T and HTTP status
ResponseEntity(T body, MultiValueMap<String,String> headers, HttpStatus status)	Creates a ResponseEntity with a body of type T, header and HTTP status

methods available:-

↳ ok(T body) → create ResponseEntity with status ok & body T

↳ ResponseBuilder badRequest() → Return ResponseBuilder with status BAD_Request

↳ ResponseBuilder notFound() → Return ResponseBuilder with status NOT_FOUND

⇒ Handling URI Data

↳ Parameter Injection

Sometimes client doesn't prefer to send data as RequestBody. client can choose to send data as part of request URI.

1) Query Parameter / Request Parameter

http://localhost:8081/infytel-1/customers/calldetails?phoneNo=9123456789

Query Parameters

- Query parameters or request parameters usually travel with the URI and are delimited by question mark.
- The query parameters in turn are delimited by ampersand from one another.
- The annotation @RequestParam helps map query/request parameters to the method arguments.
- @RequestParam annotation expects the name that it holds to be similar to the one that is present in the request URI.This makes the code tightly coupled with the request URI.

Ex

```
@RestController
@RequestMapping("/calldetails")
public class CallDetailsController
{
    //Fetching call details based on the request parameters being passed along with the URI
    @GetMapping(produces = "application/json")
    public List<CallDetailsDTO> callDetails(
            @RequestParam("calledBy") long calledBy, @RequestParam("calledOn") String calledOn)
    {
        //code goes here
    }
}
```

2) Path Variable :-

http://localhost:8081/infytel-1/customers/9123456789/calldetails

Path Variable

- Path variables are usually available at the end of the request URIs delimited by slash (/).
- @Pathvariable annotation is applied on the argument of the controller method whose value needs to be extracted out of the request URI.
- A request URI can have any number of path variables.
- Multiple path variables require the usage of multiple @PathVariable annotations.
- @Pathvariable can be used with any type of request method. For example, GET, POST, DELETE, etc.,

Example of Path Variable

```
@RestController
@RequestMapping("/customers")
public class CustomerController
{
    //Updating an existing customer
    @PutMapping(value = "/{phoneNumber}", consumes = "application/json")
    public String updateCustomer(
            @PathVariable("phoneNumber") long phoneNumber,
                @RequestBody CustomerDTO customerDTO) {
        //code goes here
    }

    // Deleting a customer
    @DeleteMapping(value="/{phoneNumber}",produces="text/html")
    public String deleteCustomer(
            @PathVariable("phoneNumber") long phoneNumber)
                    throws NoSuchCustomerException {
        //code goes here
    }
}
```

3) Matrix Variable

http://localhost:8081/infytel-1/customers;phoneNo=9123456789/calldetails

Matrix Variables

- Matrix variables are a block/segment of values that travel along with the URI. For example, /localRate=1,2,3/
- These variables may appear in the middle of the path unlike query parameters which appear only towards the end of the URI.
- Matrix variables follow name=value format and use semicolon to get delimited from one other matrix variable.
- A matrix variable can carry any number of values, delimited by commas.
- @MatrixVariable is used to extract the matrix variables.

Ex

```
@RestController
@RequestMapping("/plans")
public class PlanController
{

   //{query} here is a place holder for the matrix variables that travel in the URI,
   //it is not mandatory that the client URI should hold a string literal called query
   @GetMapping(value = "/{query}/plan", produces = {"application/json","application/xml"})
   public EntityList<PlanDTO> plansLocalRate(
           @MatrixVariable(pathVar="query") Map<String, List<Integer>> map ) {
   //code goes here
   }
}
```

⇒ Exception Handling & Data Validation

↳ Exception handling make our application robust mean it will prevent app.n to stop abrupty

@ExceptionHandler is used in method level (helps handle except.n specifically to that method)

@RestControllerAdvice is used in class level (help handle except.n globaly)

Ex:-

```
@RestControllerAdvice
public class ExceptionControllerAdvice {

    @ExceptionHandler(NoSuchCustomerException.class)
    public ResponseEntity<String> exceptionHandler2(NoSuchCustomerException ex) {

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

    }
}
```

we generally make a seperate class to handle exception

```
public class ErrorMessage {
    private int errorCode;
    private String message;
    //getters and setters go here
}
```

} error message type class

```
@RestControllerAdvice
public class ExceptionControllerAdvice {

    @ExceptionHandler(NoSuchCustomerException.class)
    public ResponseEntity<ErrorMessage> exceptionHandler2(NoSuchCustomerException ex) {
        ErrorMessage error = new ErrorMessage();
        error.setErrorCode(HttpStatus.BAD_GATEWAY.value());
        error.setMessage(ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.OK);

    }
}
```

} class to handle exception

↳ Data Validation

when data send is missing something or not in valid format then we need to have some validation.

Annotations that we have are :-

- ↳ Digits
- ↳ Email
- ↳ Min
- ↳ Max
- ↳ NotEmpty
- ↳ NotNull
- ↳ Null
- ↳ Pattern

```
public class CustomerDTO {
    long phoneNo;
    @NotNull
    String name;
    @Email(message = "Email id is not in format, please check")
    String email;
    int age;
    char gender;
    List<FriendFamilyDTO> friendAndFamily;
    String password;
    String address;
    PlanDTO currentPlan;
    // getter and setters go here
}
```

In addition, we have to mention that validation is required while JSON data is deserialized to CustomerDTO and get the validn error

Apply @Valid annotation to handler method

@Valid @RequestBody CustomerDTO customerDTO - indicates that CustomerDTO should be validated before being taken.

```
@PostMapping(consumes="application/json")
    public ResponseEntity createCustomer(@Valid @RequestBody CustomerDTO customerDTO, Errors errors)
    {
        String response = "";
        if (errors.hasErrors())
        {
            response  = errors.getAllErrors().stream()
                                             .map(ObjectError::getDefaultMessage)
                                             .collect(Collectors.joining(","));
            ErrorMessage error = new ErrorMessage();
            error.setErrorCode(HttpStatus.NOT_ACCEPTABLE.value());
            error.setMessage(response);
            return ResponseEntity.ok(error); → body
        }
        else
        {
            response = customerService.createCustomer(customerDTO);
            return ResponseEntity.ok(response);
        }

    }
```

if we faced some error

if there is no error

along with validating the incoming object/DTO, we need to validate URI parameters as well.

↳ @Validated

The controller where the URI parameter validation is carried out should be annotated with @Validated.

```
@Validated
public class CustomerController
```

→ Now, to make the applicatn centralized. we apply exception handler for:-

- MethodArgumentNotValidException - Validation failures in DTOs
- ConstraintViolationException - Validation failures in URI parameters

```
//validation failures on DTOs
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ErrorMessage> handleValidationExceptions(
      MethodArgumentNotValidException ex) {

        ErrorMessage error = new ErrorMessage();
        error.setErrorCode(HttpStatus.BAD_REQUEST.value());
        error.setMessage(ex.getBindingResult().getAllErrors().stream()
                                              .map(ObjectError::getDefaultMessage)
                                              .collect(Collectors.joining(", ")));
        return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);
    }

//Validation failures on URI parameters
@ExceptionHandler(ConstraintViolationException.class)
public ResponseEntity<ErrorMessage> handleConstraintValidationExceptions(
        ConstraintViolationException ex) {

        ErrorMessage error = new ErrorMessage();
        error.setErrorCode(HttpStatus.BAD_REQUEST.value());
        error.setMessage(ex.getConstraintViolations().stream()
                                               .map(ConstraintViolation::getMessage)
                                              .collect(Collectors.joining(", ")));
        return new ResponseEntity<>(error, HttpStatus.BAD_REQUEST);

}
```

⇒ Spring REST Client

When REST services need to contact other RESTful services. then we use REST client and name of that REST client is REST Template.

1) Calling a RESTful service of HTTP request method Get:

```
RestTemplate restTemplate = new RestTemplate();
String url="http://localhost:8080/infytel/customers";
List<CustomerDTO> customers = (List<CustomerDTO>) restTemplate.getForObject(url, List.class);
for(CustomerDTO customer:customers)
{
    System.out.println("Customer Name: "+customer.getName());
    System.out.println("Customer Phone: "+customer.getPhoneNo());
    System.out.println("Email Id: "+customer.getEmail());

}
```

) Calling a RESTful service of HTTP request method Post:

```
CustomerDTO customerrequest= new CustomerDTO();
//populate CustomerDTO object
//....

String url="http://localhost:8080/infytel/customers"
RestTemplate restTemplate = new RestTemplate();
ResponseEntity<String> response =   restTemplate.postForEntity(url,customerrequest,String.class );
System.out.println("status code :" + response.getStatusCodeValue());
System.out.println("Info        :" + response.getBody());
```

⇒ Versioning Spring REST API

versioning is needed when we are creating a new service but we need to keep to previous version as well.

Different ways of API versioning are :-

- URI versioning
- ✱ Request Parameter versioning gmp
- Custom Header versioning
- Accept Header versioning

1) URI versioning

↳ It involves different URI for every newer version of API.

V-1

```
@GetMapping(value = "v1/{planId}")

public ResponseEntity<PlanDTO> getPlan(@PathVariable("planId")String planId)
{
   //Return the complete plan details
}
```

V-2

```
@GetMapping(value = "v2/{planId}")

public ResponseEntity<PlanDTO> getPlanv2(@PathVariable("planId")String planId)
{
   //Returns only the localRate and nationalRate
}
```

2) Custom Header

V-1

```
@GetMapping(value = "{planId}",headers = "X-API-VERSION=1")

public ResponseEntity<PlanDTO> getPlan(@PathVariable("planId")String planId)
{
   //Return the complete plan details
}
```

V-2

```
@GetMapping(value = "{planId}",headers = "X-API-VERSION=2")

public ResponseEntity<PlanDTO> getPlanv2(@PathVariable("planId")String planId)
{
   //Returns only the localRate and nationalRate
}
```

3) Accept Header

V-1

```
@GetMapping(value = "/{planId}", produces = "application/vnd.plans.app-v1+json")

public ResponseEntity<PlanDTO> getPlan(@PathVariable("planId")String planId)
{
   //Return the complete plan details
}
```

V-2

```
@GetMapping(value = "/{planId}", produces = "application/vnd.plans.app-v2+json")

public ResponseEntity<PlanDTO> getPlanv2(@PathVariable("planId")String planId)
{
   //Returns only the localRate and nationalRate
}
```

4) Request Parameter Versioning

V-1

```
@GetMapping(value = "/{planId}", params = "version=1")

public ResponseEntity<PlanDTO> getPlan(@PathVariable("planId")String planId)
{
   //Return the complete plan details
}
```

V-2

```
@GetMapping(value = "/{planId}", params = "version=2")

public ResponseEntity<PlanDTO> getPlanv2(@PathVariable("planId")String planId)
{
   //Returns only the localRate and nationalRate
}
```

⇒ CORS - Cross Origin Resource Sharing

Typically, scripts running in a web app. of one origin cannot have access to the resource of a server available at a different origin. (In this condition CORS error will be thrown)

error :-

```
Cross-Origin Request Blocked: The Same Origin Policy disallows
reading the remote resource at https://domain-1.com
```

Cross-Origin Calls can be enabled using 3 ways

- Cors configuration on REST controller methods
- Cors configuration on REST controller itself
- Cors configuration globally

1) On method

(by default its on all methods)

ex.

```
@Controller
@RequestMapping(path="/customers")
public class CustomerController
{
    @CrossOrigin(origins = "*", allowedHeaders = "*")
    @GetMapping()
    public String homeInit(Model model) {
        return "home";
    }
}
```

all the fields that we can apply on @CrossOrigin

- origins - list of allowed origins to access the method
- methods - list of supported HTTP request methods
- allowedHeaders - list of request headers which can be used during the request
- exposedHeaders - list of response headers which the browser can allow the clients to access
- allowCredentials - determines whether the browser can include cookies that are associated with the request

2) On Controller class itself

```
@Controller
@CrossOrigin(origins = "*", allowedHeaders = "*")
@RequestMapping(path="/customers")
public class CustomerController
{
    @GetMapping()
    public String homeInit(Model model) {
        return "home";
    }
}
```

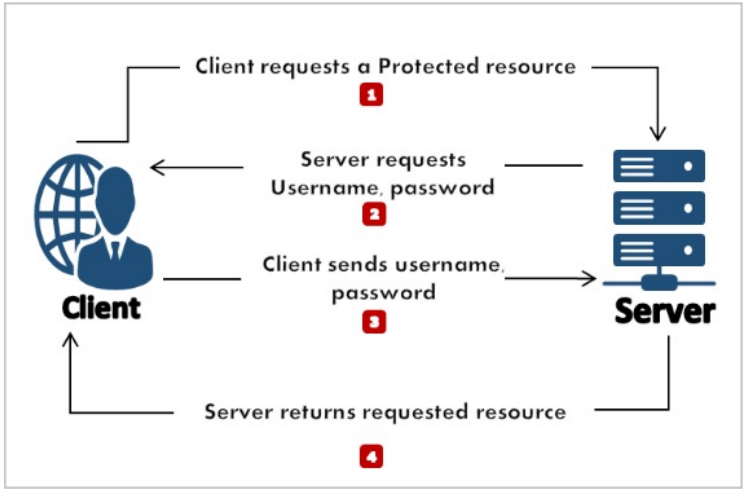
3) Globally

Look at the code below. The starter class has been modified to implement WebMvcConfigurer and override addCorsMappings() method that takes the CorsRegistry object as argument using which we can configure the allowed set of domains and HTTP methods as well.

```
@SpringBootApplication
public class InfytelApplication implements WebMvcConfigurer {

    public static void main(String[] args) {
        SpringApplication.run(InfytelDemo8BApplication.class, args);
    }

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**").allowedMethods("GET", "POST");
    }

}
```

⇒ Securing Spring REST API

using basic authentication



↳ dependency that we need for security

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

→ There are some predefined properties as well which need to be configured in application.properties file

↳ spring.security.user.name

↳ spring.security.user.password

→ Steps we need to follow when we are customizing security related aspects

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
}
```

need to override these methods as well:-

```
    //Configuring username and password and authorities
    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
          .withUser("admin").password(passwordEncoder().encode("infytel"))
          .authorities("ROLE_USER");
    }

//Configuring the HttpSecurity
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
          .antMatchers("/securityNone").permitAll()//URLS starting with securityNone need not be authenticated
          .anyRequest().authenticated() //rest of the URL's should be authenticated
          .and().
          httpBasic().and()//Every request should be authenticated it should not store in a session
          .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.NEVER).and().csrf().disable();

    }
//The passwordEncorder to be used.

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
```