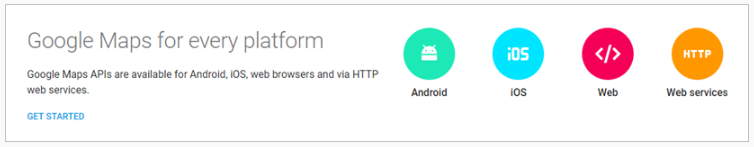
Many a times, you would want to use an already existing functionality of an application into other applications running on a different platforms and devices. For example, applications running on different platform can interact with Google Maps to provide location services.



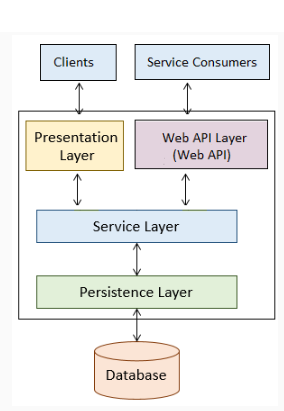
Similarly, users of Instagram, a photo sharing application, can share photographs not only with Instagram friends but also with friends on other social networking applications such as Twitter and Facebook.



Have you ever wondered how all this is possible? How application running on different platforms using different technologies can communicate with each other? All these are achieved using **Web Services**. So, let us learn what is a web service.

Note : All trademarks belong to their respective owners.

Web service is a standard way of communication between web applications running on diverse platforms and frameworks. In our scenario, Google Map is exposed as web service and consumed by different applications running on different devices. Web services are usually enclosed in the Web API layer or API layer, which communicates directly with the service layer. The Web Service layer provides the logic to convert software objects into different representations that can be sent to destination applications.



When an application is developed as a web service following are the benefits:

* Integrates with other systems easily
* Creates reusable components
* Cost savings
* Application built using different technologies can interact with each other, for example a Java based application can send data to a.NET application.

As web services provides way for communication between web application let us understand some concepts of web applications.

**Web Client**

A web client is a software using which you can interact with the server where the web application is hosted. The most commonly used web clients are web browsers such as Google Chrome, Mozilla Firefox, Internet Explorer, etc.

**Web Server**

A web server is a software where the web application is hosted. It processes the request received from the client and sends the response back to the client. It runs on some physical system and listens to client request on a specific port. Apache Tomcat is one of the most widely used web server.

**Uniform Resource Locator (URL)**

URL stands for Universal Resource Locator. It used by the web client to locate the web server and resource. Every resource on the web has its own unique URL. The following is an example of URL:

http://infy.com:8080/InfyBank/jsps/welcome.jsp

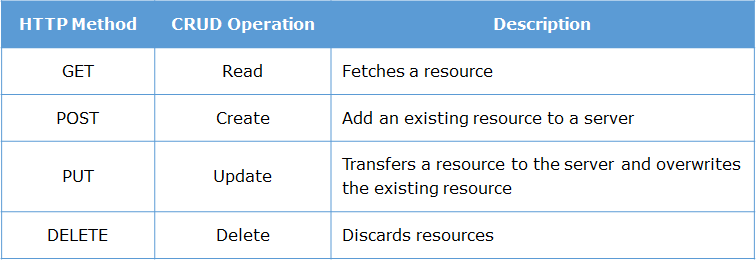
http:// – It is the communication protocol used for server-client communication.

infy.com  – It is the hostname of the server that maps to a unique IP address.

8080 – This is the port on which server is listening. It is optional and if not provided request will go to the default port of the protocol.

**Hypertext Transfer Protocol (HTTP)**

It is a protocol of communication between client and server. A client sends an HTTP request for a resource and the server returns an HTTP response with the desired resource. The HTTP request contains information about the action that the client is requesting the server to perform. This information is mentioned in HTTP request as the following methods:



**GET method**

The HTTP GET method is used to retrieve a resource. It means, “get the resource identified by this URL.”  If parameters have to be passed along with GET request then they are passed by appending a query string to the URL. For example, the following is a URL for passing tom as a username:

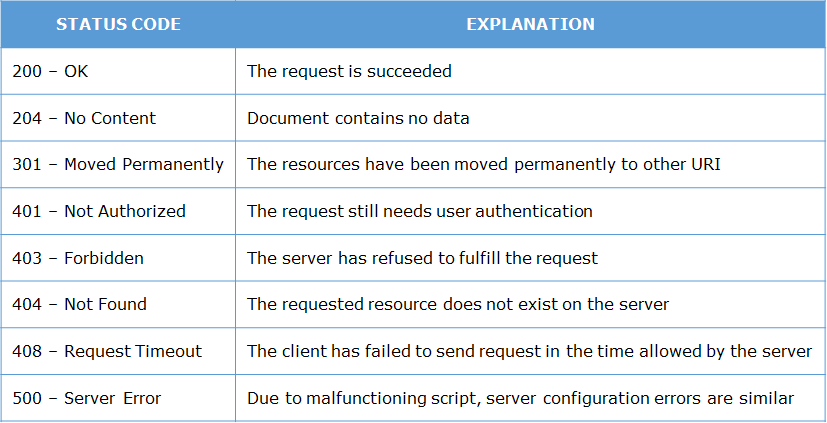
http://infy.com:8080/InfyBank?username=Tom

In above URL, the part after the question mark is called a query string. It consists of parameter name and value pairs separated by an ampersand (&). In HTTP GET request the form data gets appended to the URL which will be visible to the user

**POST method**

A POST request is used to send data to the server in order to be processed. It means,“post the data to the resource identified by this URL”. The data is sent as a body of the message.

An HTTP message sent by a server to a client is called an HTTP response. The initial line of an HTTP response is called the status line. It has three parts, separated by spaces: the HTTP version, a response status code that tells the result of the request, and an English phrase describing the status code. Some important response codes are as follows:



**Web Container**

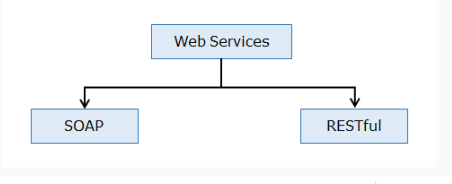
Web applications are composed of web components such as servlets, Java Server Page (JSP). These web components gets executed to generate content based on client request. So they need an environment for their execution. This execution environment is called as a web container. Web server takes the help of web container for generating dynamic content. Some commonly used web containers are

* Apache Tomcat
* GlassFish
* Jetty

**Application Server**

A web application resides in an application server. The application server provides the web application with easy and managed access to the resources of the system. It also provides low-level services, such as the HTTP protocol implementation and database connection management. A web container is just a part of an application server.

Web Services are of two types:



**RESTful Web Services**

RESTful web services are based on REST principles. As per REST principle everything is a resource.

**SOAP based Web Services**

SOAP based web services are services that are described, discovered and accessed using standards recommended by W3C (not covered as part of this course).

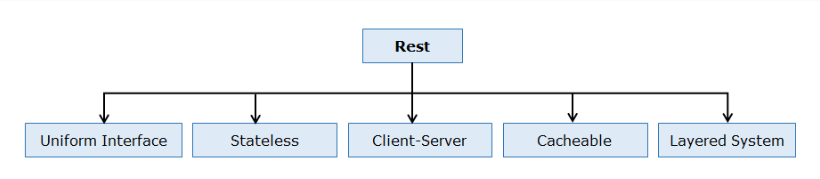
In this course, you will be learning about RESTful Web Services.

**What is REST?**

REST stands for REpresentational State Transfer. This term was coined by Roy Fielding in his doctoral dissertation in 2000. It specifies a set of architectural principles for web which uses HTTP protocol for communication.

**REST Principles**

The following figure shows five important REST principles:



**Uniform Interface**

In REST anything that can be accessed or manipulated is a resource. For example, "customer", "order", "blog“ all are resources. Every resource has a representation (XML/JSON) and are uniquely identified by a Uniform Resource Identifier (URI). Resources are manipulated using GET, POST, PUT and DELETE methods of HTTP to perform read, create, update and delete of the resource. You can perform different operation on resources using different HTTP methods but with same URI.

Example : If a customer is a resource, then customer details can be added, deleted, updated or retrieved. If a blog is another resource, then the operations we can perform on the blog can be create a blog, read a blog, update a blog or delete a blog.

**Stateless**

The communication between service consumer (client) and service provider(server) must be stateless between requests. Stateless requires each request from a service consumer to contain all the necessary information for the service to understand the meaning of the request and all session state data should then be returned to the service consumer at the end of each request.

Advantage: Statelessness ensures that each service consumer request can be treated independently by the service.

**Client Server**

Client-Server enforces the separation of concerns in the form of a client-server architecture, this helps establish a distributed architecture. This is the most foundational constraint.

**Cacheable**

Service consumers can cache and reuse response message data. HTTP GET, PUT and DELETE operations are cacheable. RESTful Web Services use HTTP protocol as a carrier for the data, they can use the metadata available in HTTP headers to enable caching. For example: Cache-Control headers are used to optimize caching for enhancing performance.

**Layered System**

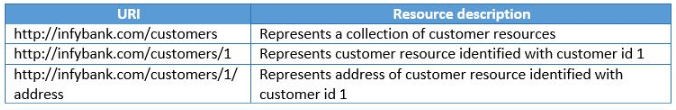
REST based solution can comprise of multiple architectural layers. This constraint builds on client-server where in we add middleware component(s). Consumer may be communicating with a service residing in a middleware layer or the actual provider. A service does not need to be aware of whether its consumer is further communicating with other services or acting as a provider for another consumer.

**What is RESTful Web Service?**

Any web service which satisfies REST principles is called RESTful web service. These web services are usually exposed to third‐party applications as application programming interface (API). Popular sites such as Facebook, Twitter, and LinkedIn offer REST web services for accessing stored data.

**REST Resources**

In REST everything is a Resource (data/functionality/web page) and is uniquely identified by a Uniform Resource Identifier (URI). The following table shows examples of URIs and the different resources they represent.



**REST URI Templates**

When creating REST API's, you need to represent the structure of a URI rather than the URI itself. For example, in InfyBank application, the URI http://infybank.com/customers/2001 would fetch details of the customer whose id is 2001. Similarly, the URI http://infybank.com/customers/2002 would fetch details of the customer whose id is 2002. In this scenario structure of URI is important. URI templates provides a standard way for describing structure of URI. The URI template for this scenario could be:

1. http:*//infybank.com/customers/{customerid}*

The curly braces {} indicate that the customerId part of the template is a variable, often called as a path variable. The clients consuming the web service can take thsi URI template as input, retrieve the value of path variable, and use it to retrieve the corresponding customer details.

**Representation**

Resources can be represented in many ways i.e: JSON/ XML/ HTML or any other type.

In the above example of http://www.infy.com/employees/john, suppose you make a request with the URL using HTTP GET, the server may send back a JSON data having employee John's information as below:

1. {
2. "name": "John",
3. "city":"California",
4. "phoneno":998765432
5. }

Or it can also be represented as XML as below:

1. <employee>
2. <name>John</name>
3. <city>California</city>
4. <phoneno>998765432</phoneno>
5. </employee>

This means the server can represent the resource in many ways.

What does **State Transfer** mean?

When you make a request to a resource like http://www.infy.com/employees/john using HTTP, we may be asking for the current "state" of this resource or its desired "state". You can use the same URI to indicate that,  we want to  :

Retrieve the current value of an Employee John (current state)

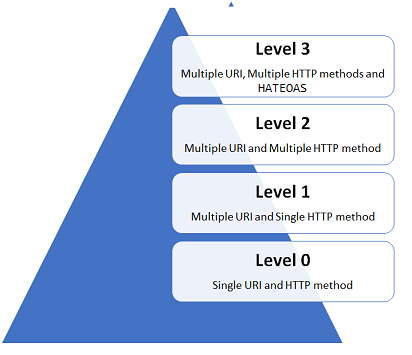
Perform Update operation on employee John's data (desired state)  etc.

The server will understand which operation has to be invoked based on the HTTP method that was used to make a call. In short it means that a client and server exchange, representation of a resource, which reflects its current state(GET) or its desired state(PUT).

In next module, you will learn how to create RESTful web services using Spring

**Richardson Maturity Model (RMM)**

The Richardson’s Maturity Model is used to divide REST based web services in different categories based on how much they follow REST principles. It was developed by Leonard Richardson. It has following levels:



**Level 0**

This is the basic level for a REST based web service. In this level single URI is used to expose the entire API. It uses a single HTTP method (typically POST or GET) to make remote procedure call on a single URI to perform operation on resources. SOAP based and XML RPC based web services comes in this level.

**Level 1**

The services in this level have multiple URI's for every resource but uses only one HTTP method – generally POST – to perform all the operations. Since specific resource is identified by a unique URI services in this level are closer to REST principles than services in level zero.

**Level 2**

The services in this level uses HTTP protocol and its methods and status codes along with URI’s to perform all the operations.  For example, to get the employees, we send a GET request with the URI /employees, and the server sends proper response 200 OK and to add a customer, we send a POST request with the URI /employees, and the server sends proper response 201 CREATED. Services which implements CRUD operations are Level 2 services.

**Level 3**

This is the most mature level for a service. It is the combination of Level 2 and HATEOAS (Hypermedia as the Engine of Application State). Services in this level provides responses with links to related resources and controls which tells service client what to do next.

Spring Boot provides the spring-boot-starter-web starter for developing REST API. This starter can be added using following dependency in pom.xml:

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-web</artifactId>
4. </dependency>

This starter adds Tomcat as the embedded servlet container which runs on port 8080. You can set a different port number using property server.port in application.properties file. For example, the following code sets port number as 3456:

1. server.port = 3456

Spring Boot by default reads the static resources such as HTML pages, CSS, Java script, images, etc. present in following locations in classpath:

* /static
* /public
* /resources
* /META-INF/resources

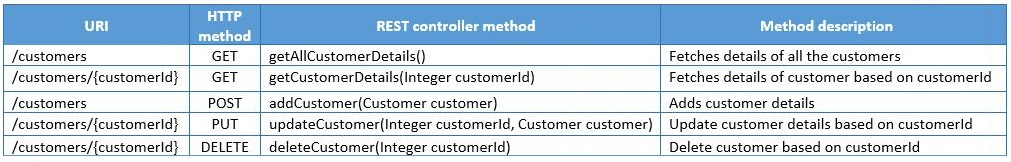
During application development, you might need to change code many times and for these changes to reflect in application, the application needs to be restarted. Spring Boot provides developer tools using which you can automate this process. To use developer tools, add Spring Boot Dev Tools dependency while creating Spring Boot project. It adds following dependency in pom.xml file:

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-devtools</artifactId>
4. <optional>true</optional>
5. </dependency>

Suppose for InfyBank application you want to develop REST API to expose its following functionalities:

1. Add a customer
2. Get all customers
3. Update customer details
4. Fetch details of particular customer
5. Delete a customer

To do this you have to create Spring controller class and implement the REST API endpoints in it. The methods in controller class which implement REST endpoints are called as REST controller methods. Now to implement above requirements you will create CustomerAPI Spring controller class. The following table shows the REST end points, HTTP methods and REST controller methods that you are going to implement in this class:



The following is implementation of CustomerAPI class:

1. @RestController
2. @RequestMapping(value = "/infybank")
3. public class CustomerAPI {
4. @Autowired
5. private CustomerService customerService;
6. @Autowired
7. private Environment environment;
8. @GetMapping(value = "/customers")
9. public ResponseEntity<List<CustomerDTO>> getAllCustomers() throws InfyBankException {
10. List<CustomerDTO> customerList = customerService.getAllCustomers();
11. return new ResponseEntity<>(customerList, HttpStatus.OK);
12. }
13. @GetMapping(value = "/customers/{customerId}")
14. public ResponseEntity<CustomerDTO> getCustomer(@PathVariable Integer customerId) throws InfyBankException {
15. CustomerDTO customer = customerService.getCustomer(customerId);
16. return new ResponseEntity<>(customer, HttpStatus.OK);
17. }
18. @PostMapping(value = "/customers")
19. public ResponseEntity<String> addCustomer(@RequestBody CustomerDTO customer) throws InfyBankException {
20. Integer customerId = customerService.addCustomer(customer);
21. String successMessage = environment.getProperty("API.INSERT\_SUCCESS") + customerId;
22. return new ResponseEntity<>(successMessage, HttpStatus.CREATED);
23. }
24. @PutMapping(value = "/customers/{customerId}")
25. public ResponseEntity<String> updateCustomer(@PathVariable Integer customerId, @RequestBody CustomerDTO customer)
26. throws InfyBankException {
27. customerService.updateCustomer(customerId, customer.getEmailId());
28. String successMessage = environment.getProperty("API.UPDATE\_SUCCESS");
29. return new ResponseEntity<>(successMessage, HttpStatus.OK);
30. }
31. @DeleteMapping(value = "/customers/{customerId}")
32. public ResponseEntity<String> deleteCustomer(@PathVariable Integer customerId) throws InfyBankException {
33. customerService.deleteCustomer(customerId);
34. String successMessage = environment.getProperty("API.DELETE\_SUCCESS");
35. return new ResponseEntity<>(successMessage, HttpStatus.OK);
36. }
37. }

Now let us understand above class in detail.

**@RestController**

This annotation is used to define REST controllers. It is a combination of @Controller and @ResponseBody annotation. @Controller annotation marks a class for discovery by component scanning. @ResponseBody tells Spring that all handler methods in the controller should have their return value written directly to the body of the response, rather than being carried in the model to a view for rendering.

**@RequestMapping**

This annotation is used to map the HTTP requests to handler methods of REST controllers.  You can use it at class-level or method-level in a controller. The class-level @RequestMapping specifies that any handler methods in this controller will handle requests with specific request path or pattern while method-level makes mappings more specific to handler methods. By default, this annotation matches to all HTTP methods. But usually controller methods should be mapped to a specific HTTP method. For this, you can use method attribute in this annotation. For example, to map an HTTP POST request you can use this annotation as follows:

1. @RequestMapping(method=RequestMethod.POST)

Spring 4.3 introduced new annotations for mapping handler methods with specific HTTP requests. These methods are as follows:

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping

**@GetMapping**

This annotation handles HTTP GET request. In CustomerAPI when an HTTP GET request is received for infybank/customers, getAllCustomerDetails() will be called so @GetMapping is used as follows:

1. @GetMapping(value="/customers")

The return type of getAllCustomerDetails() method is ResponseEntity. ResponseEntity represents the entire HTTP response which includes HTTP response code, response body and response headers. In method implementation, details of all the customers are fetched using CustomerService and then an instance of ResponseEntity is created with customer data and the HttpStatus.OK status code as follows:

1. ResponseEntity<List<Customer>> response = new ResponseEntity<List<Customer>>(customerList, HttpStatus.OK);

**@PathVariable**

This annotation is used to extract values from URI template into a method parameter. It indicates that a method parameter should be bound to a URI template variable [the one in '{}']. In getCustomerDetails() method, the value attribute of GetMapping annotation takes URI template /customers/{customerId} as parameter where the {customerId} portion of the path is a placeholder. The actual value in the request is given to the customerId parameter, which is mapped to the {customerId} placeholder by @PathVariable. So when HTTP GET request is received for infybank/customers/1 then details of only that customer whose customerId is 1 is fetched.

Note that the path variable name and the parameter name should match.  If you want to use a different parameter name, then you could specify the path variable name in the annotation as follows:

1. @GetMapping(value = "/customers/{customerId}")
2. public ResponseEntity<Customer> getCustomerDetails(@PathVariable(value="customerId") Integer custId) throws InfyBankException{
3. *//rest of the code*
4. }

You can also use @PathVariable("customerId") or @PathVariable(name="customerId"). This annotation can be used in any type of request method (GET, POST, DELETE, etc).

**@PostMapping**

This annotation handles HTTP POST request.  In CustomerAPI when an HTTP POST request is received for infybank/customers, then addCustomer() method will be called so @PostMapping is used as follows:

1. @PostMapping(value = "/customers")
2. public ResponseEntity<String> addCustomer(@RequestBody Customer customer) throws InfyBankException{
3. *//rest of the code*
4. }

You know that along with the POST request we have to send customer data. For this customer parameter of addCustomer() method is annotated with @RequestBody  annotation. It ensures that JSON data about customer in the request body is converted to a Customer object.

**@PutMapping**

This annotation handles HTTP PUT request. In CustomerAPI when an HTTP PUT request is received for infybank/customers/{customerId}, then updateCustomer() method will be called so @PutMapping is used as follows:

1. @PutMapping(value = "/customers/{customerId}")
2. public ResponseEntity<String> updateCustomer(@PathVariable Integer customerId, @RequestBody Customer customer) throws InfyBankException{
3. *//rest of the code*
4. }

**@DeleteMapping**

This annotation handles HTTP DELETE request.  In CustomerAPI when an HTTP DELETE request is received for infybank/customers/{customerId}, then deleteCustomer() method will be called so @DeleteMapping is used as follows:

1. @DeleteMapping(value = "/customers/{customerId}")
2. public ResponseEntity<String> deleteCustomer(@PathVariable Integer customerId) throws InfyBankException{
3. *//rest of the code*
4. }

**@RequestParam**

So far you have learned that using path variable you can extract values from URI template. But sometimes the request contains information as query string. A query string in part of URL and contains key-value pair. For example, consider the following URL:

1. http:*//localhost:8080/infybank/customers?customerId=1234&name=Smith*

In above URL customerId=1234&name=Smith is query string with key customerId and name. The value of cutomerId is 1234 and value of name is Smith.

Using @RequestParam annotation you can bind values of query string to the controller method parameters. For example following handler method is mapped with the request /customers?customerId = 12345 :

1. @GetMapping(value = "/customers")
2. public ResponseEntity<String> getCustomer(@RequestParam Integer customerId) throws InfyBankException{
3. *//rest of the code*
4. }