RESTful Web Services are basically REST Architecture based Web Services. In REST Architecture everything is a resource. RESTful web services are light weight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications.

## What is REST architecture?

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol.

It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

In REST architecture, a REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML. JSON is the most popular one.

### HTTP methods

Following four HTTP methods are commonly used in REST based architecture.

**GET** − Provides a read only access to a resource.

**POST** − Used to create a new resource.

**DELETE** − Used to remove a resource.

**PUT** − Used to update a existing resource or create a new resource.

## Introduction to RESTFul web services

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, provides resource representation such as JSON and set of HTTP Methods.

## Creating RESTFul Webservice

we'll create a webservice say user management with following functionalities −

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr.No.** | **URI** | **HTTP Method** | **POST body** | **Result** |
| 1 | /UserService/users | GET | empty | Show list of all the users. |
| 2 | /UserService/addUser | POST | JSON String | Add details of new user. |
| 3 | /UserService/getUser/:id | GET | empty | Show details of a user. |

# **RESTful Web Services - Environment Setup**

you on how to prepare a development environment to start your work with **Jersey Framework** to create RESTful Web Services. Jersey framework implements **JAX-RS 2.0** API, which is a standard specification to create RESTful Web Services. This tutorial will also teach you how to setup **JDK, Tomcat** and **Eclipse** on your machine before you the Jersey Framework is setup.

### Adding the Required Libraries

Add Jersey Framework and its dependencies (libraries) in our project. Copy all jars from following directories of download jersey zip folder in WEB-INF/lib directory of the project.

* \jaxrs-ri-2.17\jaxrs-ri\api
* \jaxrs-ri-2.17\jaxrs-ri\ext
* \jaxrs-ri-2.17\jaxrs-ri\lib

import java.util.List;

import javax.ws.rs.GET;

import javax.ws.rs.Path;

import javax.ws.rs.Produces;

import javax.ws.rs.core.MediaType;

@Path("/UserService")

public class UserService {

UserDao userDao = new UserDao();

@GET

@Path("/users")

@Produces(MediaType.APPLICATION\_XML)

public List<User> getUsers(){

return userDao.getAllUsers();

} }

The first step is to specify a path for the web service using @Path annotation to the UserService.

The second step is to specify a path for the particular web service method using @Path annotation to method of UserService.

## What is a Resource?

REST architecture treats every content as a resource. These resources can be Text Files, Html Pages, Images, Videos or Dynamic Business Data. REST Server simply provides access to resources and REST client accesses and modifies the resources. Here each resource is identified by URIs/ Global IDs. REST uses various representations to represent a resource where Text, JSON, XML. The most popular representations of resources are XML and JSON.

### Representation of Resources

A resource in REST is a similar Object in Object Oriented Programming or is like an Entity in a Database. Once a resource is identified then its representation is to be decided using a standard format so that the server can send the resource in the above said format and client can understand the same format.

a user is a resource which is represented using the following XML format −

<user>

<id>1</id>

<name>Mahesh</name>

<profession>Teacher</profession> </user>

The same resource can be represented in JSON format as follows −

{

"id":1,

"name":"Mahesh",

"profession":"Teacher" }

### Good Resources Representation

REST does not impose any restriction on the format of a resource representation. A client can ask for JSON representation whereas another client may ask for XML representation of the same resource to the server and so on. It is the responsibility of the REST server to pass the client the resource in the format that the client understands.

Following are some important points to be considered while designing a representation format of a resource in RESTful Web Services.

**Understandability** − Both the Server and the Client should be able to understand and utilize the representation format of the resource.

**Completeness** − Format should be able to represent a resource completely. For example, a resource can contain another resource. Format should be able to represent simple as well as complex structures of resources.

**Linkablity** − A resource can have a linkage to another resource, a format should be able to handle such situations.

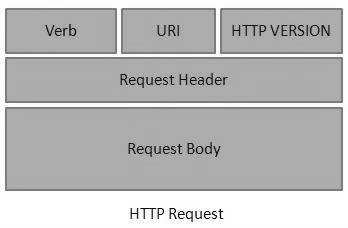
However, at present most of the web services are representing resources using either XML or JSON format. There are plenty of libraries and tools available to understand, parse, and modify XML and JSON data.

# **RESTful Web Services - Messages**

RESTful Web Services make use of HTTP protocols as a medium of communication between client and server. A client sends a message in form of a HTTP Request and the server responds in the form of an HTTP Response. This technique is termed as Messaging.

These messages contain message data and metadata i.e. information about message itself. Let us have a look on the HTTP Request and HTTP Response messages for HTTP 1.1.

## HTTP Request



An HTTP Request has five major parts −

**Verb** − Indicates the HTTP methods such as GET, POST, DELETE, PUT, etc.

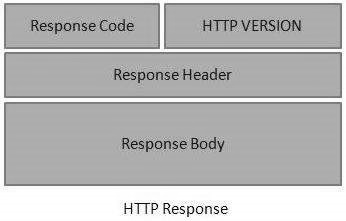
**URI** − Uniform Resource Identifier (URI) to identify the resource on the server.

**HTTP Version** − Indicates the HTTP version. For example, HTTP v1.1.

**Request Header** − Contains metadata for the HTTP Request message as key-value pairs. For example, client (or browser) type, format supported by the client, format of the message body, cache settings, etc.

**Request Body** − Message content or Resource representation.

## HTTP Response



An HTTP Response has four major parts −

**Status/Response Code** − Indicates the Server status for the requested resource. For example, 404 means resource not found and 200 means response is ok.

**HTTP Version** − Indicates the HTTP version. For example HTTP v1.1.

**Response Header** − Contains metadata for the HTTP Response message as keyvalue pairs. For example, content length, content type, response date, server type, etc.

**Response Body** − Response message content or Resource representation.

### Example

let us put http://localhost:8080/UserManagement/rest/UserService/users in the POSTMAN with a GET request. If you click on the Preview button which is near the send button of Postman and then click on the Send button, you may see the following output.



# **RESTful Web Services - Addressing**

Addressing refers to locating a resource or multiple resources lying on the server. It is analogous to locate a postal address of a person.

Each resource in REST architecture is identified by its URI (Uniform Resource Identifier). A URI is of the following format −

<protocol>://<service-name>/<ResourceType>/<ResourceID>

Purpose of an URI is to locate a resource(s) on the server hosting the web service. Another important attribute of a request is VERB which identifies the operation to be performed on the resource. For example,

URI is

**http://localhost:8080/UserManagement/rest/UserService/users**

&& the VERB is GET.

## Constructing a Standard URI

The following are important points to be considered while designing a URI −

**Use Plural Noun** − Use plural noun to define resources. For example, we've used users to identify users as a resource.

**Avoid using spaces** − Use underscore (\_) or hyphen (-) when using a long resource name. For example, use authorized\_users instead of authorized%20users.

**Use lowercase letters** − Although URI is case-insensitive, it is a good practice to keep the url in lower case letters only.

**Maintain Backward Compatibility** − As Web Service is a public service, a URI once made public should always be available. In case, URI gets updated, redirect the older URI to a new URI using the HTTP Status code, 300.

**Use HTTP Verb** − Always use HTTP Verb like GET, PUT and DELETE to do the operations on the resource. It is not good to use operations name in the URI.

### Example

Following is an example of a poor URI to fetch a user.

http://localhost:8080/UserManagement/rest/UserService/getUser/1

Following is an example of a good URI to fetch a user.

http://localhost:8080/UserManagement/rest/UserService/users/1

# **RESTful Web Services - Methods**

|  |  |
| --- | --- |
| **HTTP Method** | GET |
| **URI** | http://localhost:8080/UserManagement/rest/UserService/users |
| **Operation** | Get list of users |
| **Operation Type** | Read Only |
| **HTTP Method** | GET |
| **URI** | http://localhost:8080/UserManagement/rest/UserService/users/1 |
| **Operation** | Get user of Id 1 |
| **Operation Type** | Read Only |

|  |  |
| --- | --- |
| **HTTP Method** | POST |
| **URI** | http://localhost:8080/UserManagement/rest/UserService/users/2 |
| **Operation** | Insert user with Id 2 |
| **Operation Type** | Non-Idempotent |

|  |  |
| --- | --- |
| **HTTP Method** | PUT |
| **URI** | http://localhost:8080/UserManagement/rest/UserService/users/2 |
| **Operation** | Update User with Id 2 |
| **Operation Type** | N/A |

|  |  |
| --- | --- |
| **HTTP Method** | DELETE |
| **URI** | http://localhost:8080/UserManagement/rest/UserService/users/1 |
| **Operation** | Delete User with Id 1 |
| **Operation Type** | Idempotent |

|  |  |
| --- | --- |
| **HTTP Method** | OPTIONS |
| **URI** | http://localhost:8080/UserManagement/rest/UserService/users |
| **Operation** | List the supported operations in web service |
| **Operation Type** | Read Only |

|  |  |
| --- | --- |
| **HTTP Method** | HEAD |
| **URI** | http://localhost:8080/UserManagement/rest/UserService/users |
| **Operation** | Returns only HTTP Header, no Body |
| **Operation Type** | Read Only |

Here are important points to be considered:

**GET** operations are read only and are safe.

**PUT** and **DELETE** operations are **idempotent** means their result will always same no matter how many times these operations are invoked.

**PUT** and **POST** operation are nearly same with the difference lying only in the result where PUT operation is **idempotent** and POST operation can cause different result.

@PUT

@Path("/users")

@Produces(MediaType.APPLICATION\_XML)

@Consumes(MediaType.APPLICATION\_FORM\_URLENCODED)

public String updateUser(@FormParam("id") int id,

@FormParam("name") String name,

@FormParam("profession") String profession,

@Context HttpServletResponse servletResponse) throws IOException{

User user = new User(id, name, profession);

int result = userDao.updateUser(user);

if(result == 1){

return SUCCESS\_RESULT;

}

return FAILURE\_RESULT;

}

DELETE

@Path("/users/{userid}")

@Produces(MediaType.APPLICATION\_XML)

public String deleteUser(@PathParam("userid") int userid){

int result = userDao.deleteUser(userid);

if(result == 1){

return SUCCESS\_RESULT;

}

return FAILURE\_RESULT;

}

@POST

@Path("/users")

@Produces(MediaType.APPLICATION\_XML)

@Consumes(MediaType.APPLICATION\_FORM\_URLENCODED)

public String createUser(@FormParam("id") int id,

@FormParam("name") String name,

@FormParam("profession") String profession,

@Context HttpServletResponse servletResponse) throws IOException{

User user = new User(id, name, profession);

int result = userDao.addUser(user);

if(result == 1){

return SUCCESS\_RESULT;

}

return FAILURE\_RESULT;

}

@GET

@Path("/users")

@Produces(MediaType.APPLICATION\_XML)

public List<User> getUsers(){

return userDao.getAllUsers();

}

@GET

@Path("/users/{userid}")

@Produces(MediaType.APPLICATION\_XML)

public User getUser(@PathParam("userid") int userid){

return userDao.getUser(userid);

}

# **RESTful Web Services - Statelessness**

As per the REST architecture, a RESTful Web Service should not keep a client state on the server. This restriction is called Statelessness.

It is the responsibility of the client to pass its context to the server and then the server can store this context to process the client's further request.

For example, session maintained by server is identified by session identifier passed by the client.

RESTful Web Services should adhere to this restriction. We have seen this in the [RESTful Web Services - Methods](https://www.tutorialspoint.com/restful/restful_methods.htm) chapter, that the web service methods are not storing any information from the client they are invoked from.

**Consider the following URL −**

https://localhost:8080/UserManagement/rest/UserService/users/1

If you hit the above url using your browser or using a java based client or using Postman, result will always be the User XML whose Id is 1 because the server does not store any information about the client.

<user>

<id>1</id>

<name>mahesh</name>

<profession>1</profession> </user>

## Advantages of Statelessness

Following are the **benefits** of statelessness in RESTful Web Services −

> Web services can treat each method request independently.

> Web services need not maintain the client's previous interactions. It simplifies the application design.

> As HTTP is itself a statelessness protocol, RESTful Web Services work seamlessly with the HTTP protocols.

**Disadvantages of Statelessness**

Following are the **disadvantages** of statelessness in RESTful Web Services −

>Web services need to get extra information in each request and then interpret to get the client's state in case the client interactions are to be taken care of.

# **RESTful Web Services - Caching**

Caching refers to storing the server response in the client itself, so that a client need not make a server request for the same resource again and again. A server response should have information about how caching is to be done, so that a client caches the response for a time-period or never caches the server response.

Following are the headers which a server response can have in order to configure a client's caching −

|  |  |
| --- | --- |
| **Sr.No.** | **Header & Description** |
| 1 | **Date**  Date and Time of the resource when it was created. |
| 2 | **Last Modified**  Date and Time of the resource when it was last modified. |
| 3 | **Cache-Control**  Primary header to control caching. |
| 4 | **Expires**  Expiration date and time of caching. |
| 5 | **Age**  Duration in seconds from when resource was fetched from the server. |

## Cache-Control Header

Following are the details of a Cache-Control header −

|  |  |
| --- | --- |
| **Sr.No.** | **Directive & Description** |
| 1 | **Public**  Indicates that resource is cacheable by any component. |
| 2 | **Private**  Indicates that resource is cacheable only by the client and the server, no intermediary can cache the resource. |
| 3 | **no-cache/no-store**  Indicates that a resource is not cacheable. |
| 4 | **max-age**  Indicates the caching is valid up to max-age in seconds. After this, client has to make another request. |
| 5 | **must-revalidate**  Indication to server to revalidate resource if max-age has passed. |

# **RESTful Web Services - Security**

As RESTful Web Services work with HTTP URL Paths, it is very important to safeguard a RESTful Web Service in the same manner as a website is secured.

Following are the best practices to be adhered to while designing a RESTful Web Service −

**Validation** − Validate all inputs on the server. Protect your server against SQL or NoSQL injection attacks.

**Session Based Authentication** − Use session based authentication to authenticate a user whenever a request is made to a Web Service method.

**No Sensitive Data in the URL** − Never use username, password or session token in a URL, these values should be passed to Web Service via the POST method.

**Restriction on Method Execution** − Allow restricted use of methods like GET, POST and DELETE methods. The GET method should not be able to delete data.

**Validate Malformed XML/JSON** − Check for well-formed input passed to a web service method.

**Throw generic Error Messages** − A web service method should use HTTP error messages like 403 to show access forbidden, etc.

**HTTP Code**

|  |  |
| --- | --- |
| **Sr.No.** | **HTTP Code & Description** |
| 1 | **200**  **OK** − shows success. |
| 2 | **201**  **CREATED** − when a resource is successfully created using POST or PUT request. Returns link to the newly created resource using the location header. |
| 3 | **204**  **NO CONTENT** − when response body is empty. For example, a DELETE request. |
| 4 | **304**  **NOT MODIFIED** − used to reduce network bandwidth usage in case of conditional GET requests. Response body should be empty. Headers should have date, location, etc. |
| 5 | **400**  **BAD REQUEST** − states that an invalid input is provided. For example, validation error, missing data. |
| 6 | **401**  **UNAUTHORIZED** − states that user is using invalid or wrong authentication token. |
| 7 | **403**  **FORBIDDEN** − states that the user is not having access to the method being used. For example, Delete access without admin rights. |
| 8 | **404**  **NOT FOUND** − states that the method is not available. |
| 9 | **409**  **CONFLICT** − states conflict situation while executing the method. For example, adding duplicate entry. |
| 10 | **500**  **INTERNAL SERVER ERROR** − states that the server has thrown some exception while executing the method. |

# **RESTful Web Services - Java (JAX-RS)**

**JAX-RS** stands for JAVA API for RESTful Web Services. JAX-RS is a JAVA based programming language API and specification to provide support for created RESTful Web Services. Its 2.0 version was released on the 24th May 2013. JAX-RS uses annotations available from Java SE 5 to simplify the development of JAVA based web services creation and deployment. It also provides supports for creating clients for RESTful Web Services.

**@Path**

Relative path of the resource class/method.

**@GET**

HTTP Get request, used to fetch resource.

**@PUT**

HTTP PUT request, used to update resource.

**@POST**

HTTP POST request, used to create a new resource.

**@DELETE**

HTTP DELETE request, used to delete resource.

**@HEAD**

HTTP HEAD request, used to get status of method availability.

**@Produces**

States the HTTP Response generated by web service. For example, APPLICATION/XML, TEXT/HTML, APPLICATION/JSON etc.

**@Consumes**

States the HTTP Request type. For example, application/x-www-formurlencoded to accept form data in HTTP body during POST request.

**@PathParam**

Binds the parameter passed to the method to a value in path.

**@QueryParam**

Binds the parameter passed to method to a query parameter in the path.

**@MatrixParam**

Binds the parameter passed to the method to a HTTP matrix parameter in path.

**@HeaderParam**

Binds the parameter passed to the method to a HTTP header.

**@CookieParam**

Binds the parameter passed to the method to a Cookie.

**@FormParam**

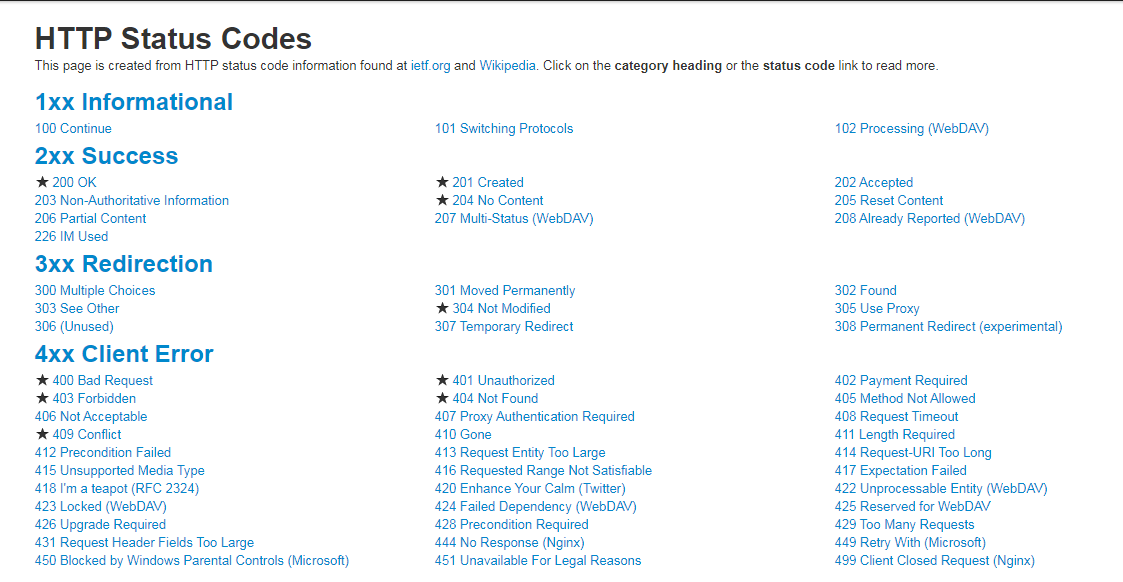
Binds the parameter passed to the method to a form value.

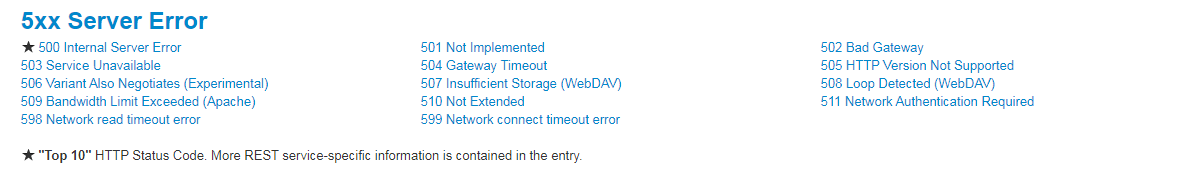
**@DefaultValue**

Assigns a default value to a parameter passed to the method.

**@Context**

Context of the resource. For example, HTTPRequest as a context.





## **Idempotence**

Idempotence is a funky word that often hooks people. Idempotence is sometimes a confusing concept, at least from the academic definition.

From a RESTful service standpoint, for an operation (or service call) to be idempotent, clients can make that same call repeatedly while producing the same result. In other words, making multiple identical requests has the same effect as making a single request. Note that while idempotent operations produce the same result on the server (no side effects), the response itself may not be the same (e.g. a resource's state may change between requests).

The PUT and DELETE methods are defined to be idempotent. However, there is a caveat on DELETE. The problem with DELETE, which if successful would normally return a 200 (OK) or 204 (No Content), will often return a 404 (Not Found) on subsequent calls, unless the service is configured to "mark" resources for deletion without actually deleting them. However, when the service actually deletes the resource, the next call will not find the resource to delete it and return a 404. However, the state on the server is the same after each DELETE call, but the response is different.

GET, HEAD, OPTIONS and TRACE methods are defined as safe, meaning they are only intended for retrieving data. This makes them idempotent as well since multiple, identical requests will behave the same.

The REST architectural style describes six constraints. These constraints, applied to the architecture, were originally communicated by Roy Fielding in his doctoral dissertation and defines the basis of RESTful-style.

The six constraints are: (click the constraint to read more)

Uniform Interface

Stateless

Cacheable

Client-Server

Layered System

Code on Demand (optional)

# **REST API Quick Tips**

## **Use HTTP Verbs to Make Your Requests Mean Something**

API consumers are capable of sending GET, POST, PUT, and DELETE verbs, which greatly enhance the clarity of a given request.

Generally, the four primary HTTP verbs are used as follows:

**GET**

Read a specific resource (by an identifier) or a collection of resources.

**PUT**

Update a specific resource (by an identifier) or a collection of resources. Can also be used to create a specific resource if the resource identifier is known before-hand.

**DELETE**

Remove/delete a specific resource by an identifier.

**POST**

Create a new resource. Also a catch-all verb for operations that don't fit into the other categories.

### **Note**

GET requests must not change any underlying resource data. Measurements and tracking which update data may still occur, but the resource data identified by the URI should not change.

**Provide Sensible Resource Names**

Producing a great API is 80% art and 20% science. Creating a URL hierarchy representing sensible resources is the art part. Having sensible resource names (which are just URL paths, such as /customers/12345/orders) improves the clarity of what a given request does.

Appropriate resource names provide context for a service request, increasing understandability of the API. Resources are viewed hierarchically via their URI names, offering consumers a friendly, easily-understood hierarchy of resources to leverage in their applications.

**Here are some quick-hit rules for URL path (resource name) design:**

1. Use identifiers in your URLs instead of in the query-string. Using URL query-string parameters is fantastic for filtering, but not for resource names.

Good: /users/12345

Poor: /api?type=user&id=23

1. Leverage the hierarchical nature of the URL to imply structure.
2. Design for your clients, not for your data.
3. Resource names should be nouns. Avoid verbs as resource names, to improve clarity. Use the HTTP methods to specify the verb portion of the request.
4. Use plurals in URL segments to keep your API URIs consistent across all HTTP methods, using the collection metaphor.

Recommended: /customers/33245/orders/8769/lineitems/1

Not: /customer/33245/order/8769/lineitem/1

1. Avoid using collection verbiage in URLs. For example 'customer\_list' as a resource. Use pluralization to indicate the collection metaphor (e.g. customers vs. customer\_list).
2. Use lower-case in URL segments, separating words with underscores ('\_') or hyphens ('-'). Some servers ignore case so it's best to be clear.
3. Keep URLs as short as possible, with as few segments as makes sense.

## **Use HTTP Response Codes to Indicate Status**

Response status codes are part of the HTTP specification. There are quite a number of them to address the most common situations. In the spirit of having our RESTful services embrace the HTTP specification, our Web APIs should return relevant HTTP status codes. For example, when a resource is successfully created (e.g. from a POST request), the API should return HTTP status code 201. A list of valid [HTTP status codes](https://www.restapitutorial.com/httpstatuscodes.html) is available [here](https://www.restapitutorial.com/httpstatuscodes.html) which lists detailed descriptions of each.

Suggested usages for the "Top 10" HTTP Response Status Codes are as follows:

**200 OK**

General success status code. This is the most common code. Used to indicate success.

**201 CREATED**

Successful creation occurred (via either POST or PUT). Set the Location header to contain a link to the newly-created resource (on POST). Response body content may or may not be present.

**204 NO CONTENT**

Indicates success but nothing is in the response body, often used for DELETE and PUT operations.

**400 BAD REQUEST**

General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples.

**401 UNAUTHORIZED**

Error code response for missing or invalid authentication token.

**403 FORBIDDEN**

Error code for when the user is not authorized to perform the operation or the resource is unavailable for some reason (e.g. time constraints, etc.).

**404 NOT FOUND**

Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask.

**405 METHOD NOT ALLOWED**

Used to indicate that the requested URL exists, but the requested HTTP method is not applicable. For example, POST */users/12345* where the API doesn't support creation of resources this way (with a provided ID). The Allow HTTP header must be set when returning a 405 to indicate the HTTP methods that are supported. In the previous case, the header would look like "Allow: GET, PUT, DELETE"

**409 CONFLICT**

Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, such as trying to create two customers with the same information, and deleting root objects when cascade-delete is not supported are a couple of examples.

**500 INTERNAL SERVER ERROR**

Never return this intentionally. The general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end.

## **Offer Both JSON and XML**

Favor JSON support unless you're in a highly-standardized and regulated industry that requires XML, schema validation and namespaces, and offer both JSON and XML unless the costs are staggering. Ideally, let consumers switch between formats using the HTTP Accept header, or by just changing an extension from .xml to .json on the URL.

Be aware that as soon as we start talking about XML support, we start talking about schemas for validation, namespaces, etc. Unless required by your industry, avoid supporting all that complexity initially, if ever. JSON is designed to be simple, terse and functional. Make your XML look like that if you can.

In other words, make the XML that is returned more JSON-like — simple and easy to read, without the schema and namespace details present, just data and links. If it ends up being more complex than this, the cost of XML will be staggering. In my experience no one has used XML responses anyway for the last several years, it's just too expensive to consume.

Note that [JSON-Schema](https://json-schema.org/) offers schema-style validation capabilities, if you need that sort of thing.

# **Using HTTP Methods for RESTful Services**

The HTTP verbs comprise a major portion of our “uniform interface” constraint and provide us the action counterpart to the noun-based resource. The primary or most-commonly-used HTTP verbs (or methods, as they are properly called) are POST, GET, PUT, PATCH, and DELETE. These correspond to create, read, update, and delete (or CRUD) operations, respectively. There are a number of other verbs, too, but are utilized less frequently. Of those less-frequent methods, OPTIONS and HEAD are used more often than others.

Below is a table summarizing recommended return values of the primary HTTP methods in combination with the resource URIs:

|  |  |  |  |
| --- | --- | --- | --- |
| **HTTP Verb** | **CRUD** | **Entire Collection (e.g. /customers)** | **Specific Item (e.g. /customers/{id})** |
| POST | Create | 201 (Created), 'Location' header with link to /customers/{id} containing new ID. | 404 (Not Found), 409 (Conflict) if resource already exists.. |
| GET | Read | 200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists. | 200 (OK), single customer. 404 (Not Found), if ID not found or invalid. |
| PUT | Update/Replace | 405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| PATCH | Update/Modify | 405 (Method Not Allowed), unless you want to modify the collection itself. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| DELETE | Delete | 405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable. | 200 (OK). 404 (Not Found), if ID not found or invalid. |

**POST**

The **POST** verb is most-often utilized to \*\*create\*\* new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information.

****Examples:****

* *POST http://www.example.com/customers*
* *POST http://www.example.com/customers/12345/orders*

**GET**

The HTTP **GET** method is used to \*\*read\*\* (or retrieve) a representation of a resource. In the “happy” (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption—calling it once has the same effect as calling it 10 times, or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request.

Do not expose unsafe operations via GET—it should never modify any resources on the server.

****Examples:****

* *GET http://www.example.com/customers/12345*
* *GET http://www.example.com/customers/12345/orders*
* *GET http://www.example.com/buckets/sample*

**PUT**

**PUT** is most-often utilized for \*\*update\*\* capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.

However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.

Alternatively, use POST to create new resources and provide the client-defined ID in the body representation—presumably to a URI that doesn't include the ID of the resource (see POST below).

On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to return a link via a Location header in the creation case since the client already set the resource ID.

PUT is not a safe operation, in that it modifies (or creates) state on the server, but it is idempotent. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call.

If, for instance, calling PUT on a resource increments a counter within the resource, the call is no longer idempotent. Sometimes that happens and it may be enough to document that the call is not idempotent. However, it's recommended to keep PUT requests idempotent. It is strongly recommended to use POST for non-idempotent requests.

****Examples:****

* *PUT http://www.example.com/customers/12345*
* *PUT http://www.example.com/customers/12345/orders/98765*
* *PUT http://www.example.com/buckets/secret\_stuff*

**PATCH**

**PATCH** is used for \*\*modify\*\* capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource.

This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch.

PATCH is neither safe nor idempotent. However, a PATCH request can be issued in such a way as to be idempotent, which also helps prevent bad outcomes from collisions between two PATCH requests on the same resource in a similar time frame. Collisions from multiple PATCH requests may be more dangerous than PUT collisions because some patch formats need to operate from a known base-point or else they will corrupt the resource. Clients using this kind of patch application should use a conditional request such that the request will fail if the resource has been updated since the client last accessed the resource. For example, the client can use a strong ETag in an If-Match header on the PATCH request.

****Examples:****

* *PATCH http://www.example.com/customers/12345*
* *PATCH http://www.example.com/customers/12345/orders/98765*
* *PATCH http://www.example.com/buckets/secret\_stuff*

**DELETE**

**DELETE** is pretty easy to understand. It is used to \*\*delete\*\* a resource identified by a URI.

On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with no body, or the JSEND-style response and HTTP status 200 are the recommended responses.

HTTP-spec-wise, DELETE operations are idempotent. If you DELETE a resource, it's removed. Repeatedly calling DELETE on that resource ends up the same: the resource is gone. If calling DELETE say, decrements a counter (within the resource), the DELETE call is no longer idempotent. As mentioned previously, usage statistics and measurements may be updated while still considering the service idempotent as long as no resource data is changed. Using POST for non-idempotent resource requests is recommended.

There is a caveat about DELETE idempotence, however. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable. This, by some opinions, makes DELETE operations no longer idempotent, however, the end-state of the resource is the same. Returning a 404 is acceptable and communicates accurately the status of the call.

****Examples:****

* *DELETE http://www.example.com/customers/12345*
* *DELETE http://www.example.com/customers/12345/orders*
* *DELETE http://www.example.com/bucket/sample*

# **Resource Naming**

n addition to utilizing the HTTP verbs appropriately, resource naming is arguably the most debated and most important concept to grasp when creating an understandable, easily leveraged Web service API. When resources are named well, an API is intuitive and easy to use. Done poorly, that same API can feel klutzy and be difficult to use and understand. Below are a few tips to get you going when creating the resource URIs for your new API.

Essentially, a RESTful API ends up being simply a collection of URIs, HTTP calls to those URIs and some JSON and/or XML representations of resources, many of which will contain relational links. The RESTful principal of addressability is covered by the URIs. Each resource has its own address or URI—every interesting piece of information the server can provide is exposed as a resource. The constraint of uniform interface is partially addressed by the combination of URIs and HTTP verbs, and using them in line with the standards and conventions.

In deciding what resources are within your system, name them as nouns as opposed to verbs or actions. In other words, a RESTful URI should refer to a resource that is a thing instead of referring to an action. Nouns have properties as verbs do not, just another distinguishing factor.

Some example resources are:

* Users of the system.
* Courses in which a student is enrolled.
* A user's timeline of posts.
* The users that follow another user.
* An article about horseback riding.

Each resource in a service suite will have at least one URI identifying it. And it's best when that URI makes sense and adequately describes the resource. URIs should follow a predictable, hierarchical structure to enhance understandability and, therefore, usability: predictable in the sense that they're consistent, hierarchical in the sense that data has structure—relationships. This is not a REST rule or constraint, but it enhances the API.

RESTful APIs are written for consumers. The name and structure of URIs should convey meaning to those consumers. It's often difficult to know what the data boundaries should be, but with understanding of your data, you most-likely are equipped to take a stab and what makes sense to return as a representation to your clients. Design for your clients, not for your data.

Let's say we're describing an order system with customers, orders, line items, products, etc. Consider the URIs involved in describing the resources in this service suite:

To insert (create) a new customer in the system, we might use:  
*POST http://www.example.com/customers*

To read a customer with Customer ID# 33245:  
*GET http://www.example.com/customers/33245* The same URI would be used for PUT and DELETE, to update and delete, respectively.

Here are proposed URIs for products:  
*POST http://www.example.com/products* for creating a new product.

*GET|PUT|DELETE http://www.example.com/products/66432*  
for reading, updating, deleting product 66432, respectively.