What is Rest?

REST is acronym for REpresentational State Transfer. It is architectural style for distributed hypermedia systems and was first presented by Roy Fielding in 2000 in his famous dissertation.

Web services based on REST Architecture are known as RESTful web services. These web services 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.

Use HTTP Methods for Restful services:

## **HTTP GET**

Use GET requests to retrieve/read resource representation/information only – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be safe methods. Additionally, GET APIs should be idempotent, which means that making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

#### **Example request URIs**

* HTTP GET http://www.appdomain.com/users
* HTTP GET http://www.appdomain.com/users?size=20&page=5
* HTTP GET http://www.appdomain.com/users/123/address

## **HTTP POST**

Use POST APIs to create new subordinate resources, e.g. a file is subordinate to a directory containing it or a row is subordinate to a database table. Talking strictly in terms of REST, POST methods are used to create a new resource into the collection of resources.POST is neither safe nor idempotent and invoking two identical POST requests will result in two different resources containing the same information

#### **Example request URIs**

* HTTP POST http://www.appdomain.com/users
* HTTP POST http://www.appdomain.com/users/123/accounts

## **HTTP PUT**

Use PUT APIs primarily to update existing resource (if the resource does not exist then API may decide to create a new resource or not). If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and if an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request. *The difference between the POST and PUT APIs can be observed in request URIs. POST requests are made on resource collections whereas PUT requests are made on an individual resource.*

#### **Example request URIs**

* HTTP PUT http://www.appdomain.com/users/123
* HTTP PUT http://www.appdomain.com/users/123/accounts/456

## **HTTP DELETE**

## DELETE APIs are used to delete resources.DELETE operations are idempotent. If you DELETE a resource, it’s removed from the collection of resource. If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

#### **Example request URIs**

* HTTP DELETE http://www.appdomain.com/users/123
* HTTP DELETE http://www.appdomain.com/users/123/accounts/456

## **HTTP PATCH**

HTTP PATCH requests are to make partial update on a resource. If you see PUT requests also modify a resource entity so to make more clear – PATCH method is the correct choice for partially updating an existing resource and PUT should only be used if you’re replacing a resource in its entirety.

**Examples:**

* *PATCH http://www.example.com/customers/12345*
* *PATCH http://www.example.com/customers/12345/orders/98765*

**HTTP OPTIONS**

The HTTP OPTIONS method is used to describe the communication options for the target resource. The client can specify a URL for the OPTIONS method, or an asterisk (\*) to refer to the entire server.

**Examples:**

curl -X OPTIONS http://example.org -i

**HTTP CONNECT**

The HTTP CONNECT starts with two way communication with a requsted resource.The CONNECT method can be used to access websites that use HTTPS. The client asks an HTTP Proxy server to tunnel the TCP connection to the desired destination. The server then proceeds to make the connection on behalf of the client. Once the connection has been established by the server, the Proxy server continues to proxy the TCP stream to and from the client.

**Examples:**

CONNECT server.example.com:80 HTTP/1.1

**HTTP Status Codes:**

1xx: Informational:

It means the request has been received and the process is continuing.

2xx: Success:

It means the action was successfully received, understood, and accepted.

3xx: Redirection

It means further action must be taken in order to complete the request.

4xx: Client Error

It means the request contains incorrect syntax or cannot be fulfilled.

5xx: Server Error

It means the server failed to fulfill an apparently valid request.

**How REST API Works?**

REST is used to build Web services that are lightweight, maintainable, and scalable in nature. A service which is built on the REST architecture is called a RESTful service. The underlying protocol for REST is HTTP, which is the basic web protocol. REST stands for REpresentational State Transfer.

## **Restful Architecture**

An application or architecture considered RESTful or REST-style has the following characteristics

1. State and functionality are divided into distributed resources – This means that every resource should be accessible via the normal HTTP commands of GET, POST, PUT, or DELETE. So if someone wanted to get a file from a server, they should be able to issue the GET request and get the file. If they want to put a file on the server, they should be able to either issue the POST or PUT request. And finally, if they wanted to delete a file from the server, they an issue the DELETE request.
2. The architecture is client/server, stateless, layered, and supports caching –

* Client-server is the typical architecture where the server can be the web server hosting the application, and the client can be as simple as the web browser.
* Stateless means that the state of the application is not maintained in REST.
* For example, if you delete a resource from a server using the DELETE command, you cannot expect that delete information to be passed to the next request.
* In order to ensure that the resource is deleted, you would need to issue the GET request. The GET request would be used to first get all the resources on the server. After which one would need to see if the resource was actually deleted.

## **RESTFul Principles and Constraints**

#### **Uniform interface**

As the constraint name itself applies, you MUST decide APIs interface for resources inside the system which are exposed to API consumers and follow religiously. A resource in the system should have only one logical URI and that should provide a way to fetch related or additional data. It’s always better to synonymise a resource with a web page.

Any single resource should not be too large and contain each and everything in its representation. Whenever relevant, a resource should contain links (HATEOAS) pointing to relative URIs to fetch related information.

Also, the resource representations across system should follow certain guidelines such as naming conventions, link formats or data format (xml or/and json).

All resources should be accessible through a common approach such as HTTP GET and similarly modified using a consistent approach.

*Once a developer becomes familiar with one of your API, he should be able to follow the similar approach for other APIs.*

#### **Client–server**

This essentially means that client application and server application MUST be able to evolve separately without any dependency on each other. A client should know only resource URIs and that’s all. Today, this is normal practice in web development so nothing fancy is required from your side. Keep it simple.

*Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.*

#### **Stateless**

Roy fielding got inspiration from HTTP, so it reflects in this constraint. Make all client-server interaction stateless. Server will not store anything about latest HTTP request client made. It will treat each and every request as new. No session, no history.

If client application needs to be a stateful application for the end user, where user logs in once and do other authorized operations thereafter, then each request from the client should contain all the information necessary to service the request – including authentication and authorization details.

*No client context shall be stored on the server between requests. The client is responsible for managing the state of the application.*

#### **Cacheable**

In today’s world, caching of data and responses is of utmost important wherever they are applicable/possible. The webpage you are reading here is also a cached version of the HTML page. Caching brings performance improvement for client side, and better scope for scalability for a server because the load has reduced.

In REST, caching shall be applied to resources when applicable and then these resources MUST declare themselves cacheable. Caching can be implemented on the server or client side.

*Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance.*

#### **Layered system**

REST allows you to use a layered system architecture where you deploy the APIs on server A, and store data on server B and authenticate requests in Server C, for example. A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.

#### **Code on demand (optional)**

Well, this constraint is optional. Most of the time you will be sending the static representations of resources in form of XML or JSON. But when you need to, you are free to return executable code to support a part of your application e.g. clients may call your API to get a UI widget rendering code. It is permitted.

*All above constraints help you build a truly RESTful API and you should follow them. Still, at times you may find yourself violating one or two constraints. Do not worry, you are still making a RESTful API – but not “truly RESTful”.*

Notice that all the above constraints are most closely related to WWW (the web). Using RESTful APIs, you can do the same thing with your web services what you do to web pages.

@Path("/hello")

public class HelloService {

@POST

@Path("/Test/")

@Produces(MediaType.TEXT\_PLAIN)

public Response getMesages() {

sop("Test Service");

String message = "Data Recieved";

ResponseBuilder responseBuilderr = Response.status(Status.OK).entity(message);

Response response = responseBuilderr.build();

return response;

}

}

client:

http://localhost:7000/TestService/rest/hello/Test/