REST Easy with Infrared360

A discussion on HTTP-based RESTful Web Services and how to use them in Infrared360

What is REST ?

REST stands for Representational State Transfer, which is an architectural style for networked hypermedia applications, Today REST is primarily used to build Web services that are lightweight, maintainable, and scalable. A service based on REST is called a RESTful service. REST is not dependent on any protocol, but almost every RESTful service uses HTTP as its underlying protocol. Other kinds of web services, such as SOAP web services, expose their own arbitrary sets of operations.

History of REST

REST was first coined by computer scientist Roy Fielding in his year-**2000** Ph.D. dissertation at the University of California, titled *Architectural Styles and the Design of Network-based Software Architectures*. **SOAP Web Services started back in 1998** at Microsoft

"Representational State Transfer (REST)," described Fielding's beliefs about how best to architect distributed hypermedia systems. Fielding noted a number of boundary conditions that describe how REST-based systems should behave. These conditions are referred to as REST constraints, with **four of the key constraints** described below:

- **Use of a uniform interface (UI).** Resources in REST-based systems should be uniquely identifiable through a single URL, and only by using the underlying methods of the **network protocol, such as DELETE, PUT and GET with HTTP, should it be possible to manipulate a resource.**

History of REST (Continued)

- **Client-server-based**. In a REST-based system, there should be a clear delineation between the client and the server. UI and request-generating concerns are the domain of the client. Meanwhile, data access, workload management and security are the domain of the server. This separation allows loose coupling between the client and the server, and each can be developed and enhanced independent of the other.
- **Stateless operations**. All client-server operations should be stateless, and any state management that is required should happen on the client, not the server.
- **RESTful resource caching**. The ability to cache resources between client invocations is a priority in order to reduce latency and improve performance. As a result, all resources should allow caching unless an explicit indication is made that it is not possible.

REST URL formatting

Protocol://ServiceName/ResourceType/ResourceID

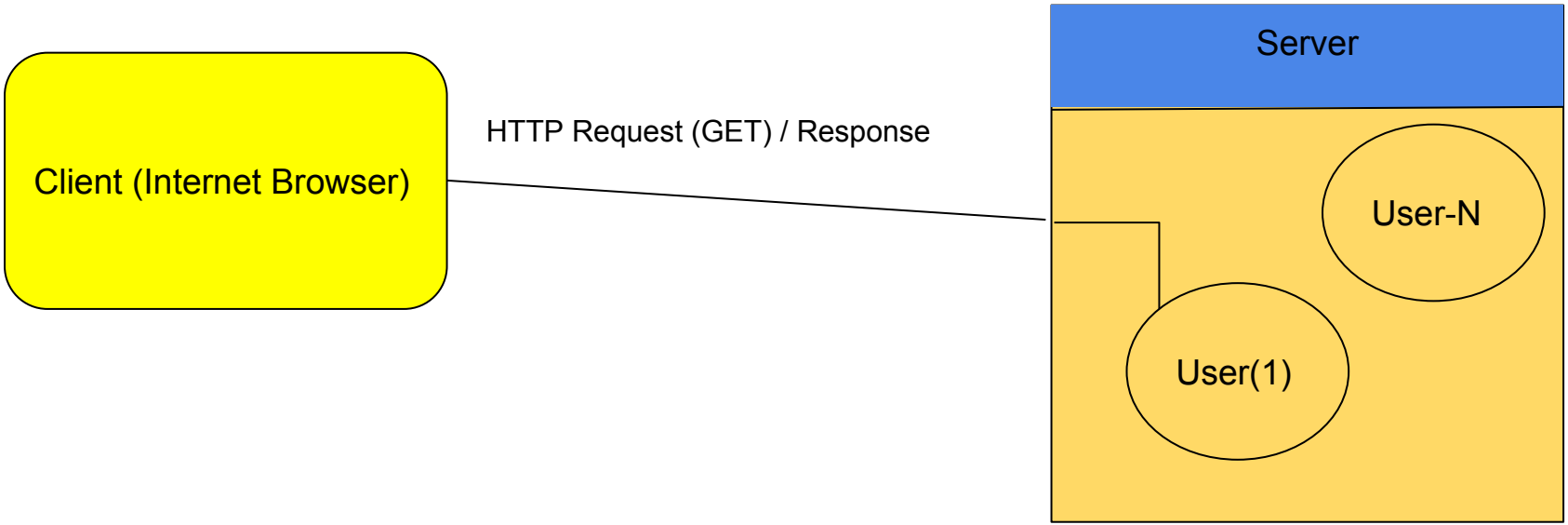
http://MyService/Users/1

REST is popular due to its simplicity and the fact that it builds upon existing systems and features *of the internet's HTTP in order to achieve its objectives, as opposed to creating new standards, frameworks and technologies.*

REST-based interactions all communicate their status using standard HTTP status codes. So, a 404 means a requested resource wasn't found; a 401 code means the request wasn't authorized; a 200 code means everything is OK; and a 500 means there was an unrecoverable application error on the server.

Similarly, details such as encryption and data transport integrity are solved not by adding new frameworks or technologies, but instead by relying on well-known Secure Sockets Layer (SSL) encryption and Transport Layer Security (TLS). ***The entire REST architecture is built upon concepts with which most developers are already familiar.***

HTTP Client / Server (http://MyService/Users/1)



JSON vs XML data payload format comparison

JSON format example

```
{
   "ID": "1",
   "Name": "R Sordillo",
   "Email": "rs@avada.com",
   "Country": "US"
}
```

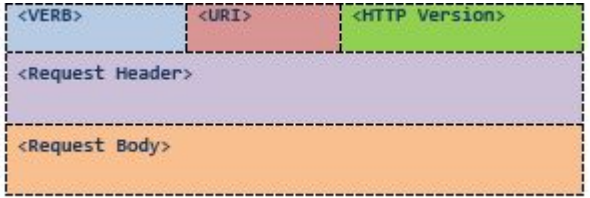
XML format example

```
<User>
   <ID>1</ID>
   <Name>R Sordillo</Name>
   <Email>rs@avada.com</Email>
   <Country>US</Country>
</User>
```

HTTP Request format

<VERB> is one of the HTTP methods like:

GET, PUT, POST, DELETE, OPTIONS, etc.



<URI> is the **URI of the resource** on which the operation is going to be performed

<HTTP Version> is the version of HTTP, generally **"HTTP v1.1" .**

<Request Header> contains the **metadata** as a collection of **key-value pairs of headers and their values**. These settings contain information about the message and its sender like client type, the formats client supports, format type of the message body, cache settings for the response, and a lot more information.

<Request Body> is the **actual message content**. In a RESTful service, that's where the representations of resources sit in a message.

There are no tags or markups to denote the beginning or end of a section in an HTML message.

HTTP POST and GET Request examples

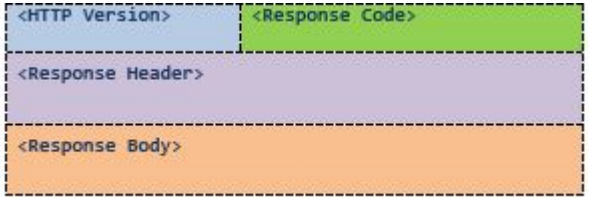
```
POST http://MyService/User/1

Host: MyService
Content-Type: application/json; charset=utf-8
Content-Length: 123

{
   "ID": "1",
   "Name": "R Sordillo",
   "Email": "rs@avada.com",
   "Country": "US"
}
```

```
GET
http://www.w3.org/Protocols/rfc2616/rfc26
16.html HTTP/1.1
Host: MyService
Accept: application/json; charset=utf-8
…
User-Agent: Mozilla/5.0 (Windows NT 6.3;
WOW64) AppleWebKit/537.36 …
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,hi;q=0.6
```

HTTP Response




```
HTTP/1.1 200 OK
Date: Sat, 23 Sep 2018 18:31:04 GMT
Server: Apache/2
Last-Modified: Wed, 01 Sep 2018 13:24:52 GMT
Accept-Ranges: bytes
Content-Length: 32859
Cache-Control: max-age=21600, must-revalidate
Expires: Sun, 24 Sep 2018 00:31:04 GMT
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns='http://www.w3.org/1999/xhtml'>

<head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head>

<body>

...
```

Difference between PUT and POST

The key difference between PUT and POST is that **PUT is idempotent while POST is not**. No matter how many times you send a PUT request, the results will be same. POST is not an idempotent method. Making a POST multiple times may result in multiple resources getting created on the server.

Another difference is that, with PUT, you must always specify the complete URI of the resource. This implies that the client should be able to construct the URI of a resource even if it does not yet exist on the server. This is possible when it is the client's job to choose a unique name or ID for the resource, just like creating a user on the server requires the client to choose a user ID. If a client is not able to guess the complete URI of the resource, then you have no option but to use POST.

Difference between PUT and POST (continued)

Request	Operation
PUT http://MyService/Persons/	Won't work. PUT requires a complete URI
PUT http://MyService/Persons/1	Insert a new person with PersonID=1 if it does not already exist, or else update the existing resource
POST http://MyService/Persons/	Insert a new person every time this request is made and generate a new PersonID.
POST http://MyService/Persons/1	Update the existing person where PersonID=1

It is clear from the above table that a **PUT request will not modify or create more than one resource no matter how many times it is fired** (if the URI is same). There is no difference between PUT and POST if the resource already exists, both update the existing resource. The third request (POST http://MyService/Persons/) will create a resource each time it is fired. ***A lot of developers think that REST does not allow POST to be used for update operation; however, REST imposes no such restrictions.***

Caching

Caching is the concept of storing the generated results and using the stored results instead of generating them repeatedly if the same request arrives in the near future. This can be done on the client, the server, or on any other component between them, such as a proxy server. Caching is a great way of enhancing the service performance, but if not managed properly, it can result in client being served stale results.

Date - Date and time when this representation was generated

Last Modified - Date and time when the server last modified this representation.

Cache-Control - The HTTP 1.1 header used to control caching

Expires - Expiration date and time for this representation. To support HTTP 1.0 clients.

Age - Duration passed in seconds since this was fetched from the server. Can be inserted by an intermediary component.

Cache-Control Directives

Directive	Application
Public	The default. Indicates any component can cache this representation.
Private	Intermediary components cannot cache this representation, only client or server can do so.
no-cache/no-store	Caching turned off.
max-age	Duration in seconds after the date-time marked in the Date header for which this representation is valid.
s-maxage	Similar to max-age but only meant for the intermediary caching.
must-revalidate	Indicates that the representation must be revalidated by the server if max-age has passed.
proxy-validate	Similar to max-validate but only meant for the intermediary caching.

REST requirements

REST requires each resource to have at least one URI. **A RESTful service uses a directory hierarchy like human readable URIs to address its resources**. The job of a URI is to identify a resource or a collection of resources. The actual operation is determined by an HTTP verb. **The URI should not say anything about the operation or action**. This enables us to call the same URI with different HTTP verbs to perform different operations.

Advantages of using REST

REST is **language-independent** architectural style. REST-based applications can be written using any language, be it Java, Kotlin, .NET, AngularJS or JavaScript. As long as a programming language can make web-based requests using HTTP, it is possible for that language to be used to invoke a RESTful API or web service. Similarly, **RESTful web services can be written using any language**, so developers tasked with implementing such services can choose technologies that work best for their situation.

The other benefit of using **REST is its pervasiveness**. On the server side, there are a variety of REST-based frameworks for helping developers create RESTful web services, including RESTlet and Apache CXF. From the client side, all of the new JavaScript frameworks, such as JQuery, Node.js, Angular and EmberJS, all have standard libraries built into their APIs that make invoking RESTful web services and consuming the XML- or JSON-based data they return a relatively straightforward endeavor.

Disadvantages of using REST

The benefit of REST using HTTP constructs also creates restrictions, however. Many of the limitations of HTTP likewise turn into shortcomings of the REST architectural style. For example, HTTP does not store state-based information between request-response cycles, which means REST-based applications must be *stateless* and any state management tasks must be performed by the client.

HTTP *doesn't have any mechanism to send push notifications from the server to the client*, it is difficult to implement any type of services where the server updates the client without the use of client-side polling of the server or some other type of web hook.

From an implementation standpoint, **a common problem with REST is the fact that developers disagree with exactly what it means to be REST-based**. Some software developers incorrectly consider anything that isn't SOAP-based to be RESTful. Driving this common misconception about REST is the fact that it is an architectural style, so there is *no reference implementation or definitive standard* that will confirm whether a given design is RESTful. As a result, there is discourse as to whether a given API conforms to REST-based principles.

REST Alternatives

Alternate technologies for creating SOA-based systems or creating APIs for invoking remote microservices include XML over HTTP (XML-RPC), CORBA, RMI over IIOP and the **Simple Object Access Protocol (SOAP)**.

Each technology has its own set of benefits and drawbacks, *but the compelling feature of REST that sets it apart is the fact that, rather than asking a developer to work with a set of custom protocols or to create a special data format for exchanging messages between a client and a server, REST insists the best way to implement a network-based web service is to simply use the basic construct of the network protocol itself, which in the case of the internet is HTTP.*

This is an important point, as REST is not intended to apply just to the internet; rather, its principles are intended to apply to all protocols, including WEBDAV, FTP and so on.

Summary

REST is another tool in your tool belt. Most vendors, integrators, and companies are providing REST interfaces to their products and solutions. This is due to the low cost of development and ownership. REST might not be the only Web Service technology that you will use. SOAP is still relevant in today's environments, but REST appears to be the future, Cloud and microservices are adopting the RESTFUL web services route..

Avada Infrared360 DEMO

Administration REST Interface

SWAGGER - REST documentation interface

URL Service using REST concepts to perform GET, PUT, DELETE HTTP verbs.

Custom Services that you can use to make your own custom REST code

Internet Browser Developer Tools (F12)