RESTful Web Services: A Tutorial

As REST has become the default for most Web and mobile apps, it's imperative to have the basics at your fingertips.

More than a decade after its introduction, REST has become one of the most important technologies for Web applications. Its importance is likely to continue growing quickly as all technologies move towards an API orientation. Every major development language now includes frameworks for building RESTful Web services. As such, it is important for Web developers and architects to have a clear understanding of REST and RESTful services. This tutorial explains REST architecturally, then dives into the details of using it for common API-based tasks.

While REST stands for Representational State Transfer, which is an architectural style for networked hypermedia applications, it 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. In this article, I examine the creation of RESTful services with HTTP.

Features of a RESTful Services

Every system uses resources. These resources can be pictures, video files, Web pages, business information, or anything that can be represented in a computer-based system. The purpose of a service is to provide a window to its clients so that they can access these resources. Service architects and developers want this service to be easy to implement, maintainable, extensible, and scalable. A RESTful design promises that and more. In general, RESTful services should have following properties and features, which I'll describe in detail:

- Representations
- Messages
- URIs
- Uniform interface
- Stateless
- Links between resources
- Caching

Representations

The focus of a RESTful service is on resources and how to provide access to these resources. A resource can easily be thought of as an object as in OOP. A resource can consist of other resources. While designing a system, the first thing to do is identify the resources and determine how they are related to each other. This is similar to the first step of designing a database: Identify entities and relations.

Once we have identified our resources, the next thing we need is to find a way to represent these resources in our system. You can use any format for representing the resources, as REST does not put a restriction on the format of a representation.

For example, depending on your requirement, you can decide to use JSON or XML. If you are building Web services that will be used by Web pages for AJAX calls, then

JSON is a good choice. XML can be used to represent more complex resources. For example a resource called "Person" can be represented as:

Listing One: JSON representation of a resource.

```
{
    "ID": "1",
    "Name": "M Vaqqas",
    "Email": "m.vaqqas@gmail.com",
    "Country": "India"
}
```

Listing Two: XML representation of a resource.

```
<Person>

<ID>1</ID>

<Name>M Vaqqas</Name>

<Email>m.vaqqas@gmail.com</Email>

<Country>India</Country>
</Person>
```

In fact, you can use more than one format and decide which one to use for a response depending on the type of client or some request parameters. Whichever format you use, a good representation should have some obvious qualities:

- Both client and server should be able to comprehend this format of representation.
- A representation should be able to completely represent a resource. If there is a need to partially represent a resource, then you should think about breaking this resource into child resources. Dividing big resources into smaller ones also allows you to transfer a smaller representation. Smaller representations mean less time required to create and transfer them, which means faster services.
- The representation should be capable of linking resources to each other. This can be done by placing the URI or unique ID of the related resource in a representation (more on this in the coming sections).

Messages

The client and service talk to each other via messages. Clients send a request to the server, and the server replies with a response. Apart from the actual data, these messages also contain some metadata about the message. It is important to have some background about the HTTP 1.1 request and response formats for designing RESTful Web services.

HTTP Request

An HTTP request has the format shown in Figure 1:

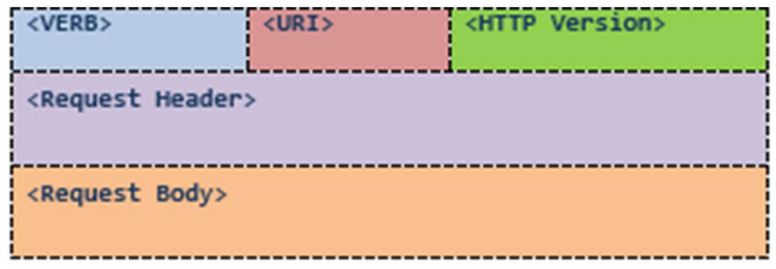

Figure 1: HTTP request format.

`<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.

Listing Three is a sample `POST` request message, which is supposed to insert a new resource `Person`.

Listing Three: A sample POST request.

```
POST http://MyService/Person/
Host: MyService
Content-Type: text/xml; charset=utf-8
Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<Person>
  <ID>1</ID>
  <Name>M Vaqqas</Name>
  <Email>m.vaqqas@gmail.com</Email>
  <Country>India</Country>
</Person>
```

You can see the `POST` command, which is followed by the URI and the HTTP version. This request also contains some request headers. `Host` is the address of the server. `Content-Type` tells about the type of contents in the message body. `Content-Length` is the length of the data in message body. `Content-Length` can be used to verify that the entire message body has been received. Notice there are no start or end tags in this message.

Listing Four is an actual `GET` request that was created by my browser when I tried to visit the HTTP 1.1 specifications on w3.org:

Listing Four: A GET request.

```
GET http://www.w3.org/Protocols/rfc2616/rfc2616.html HTTP/1.1
Host: www.w3.org
Accept: text/html,application/xhtml+xml,application/xml; …
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
```

There is no message body in this request. The `Accept` header tells the server about the various presentation formats this client supports. A server, if it supports more than one representation format, it can decide the representation format for a response at runtime depending on the value of the `Accept` header. `User-Agent` contains information about the type of client who made this request. `Accept-Encoding/Language` tells about the encoding and language this client supports.

HTTP Response

Figure 2 shows the format of an HTTP response:

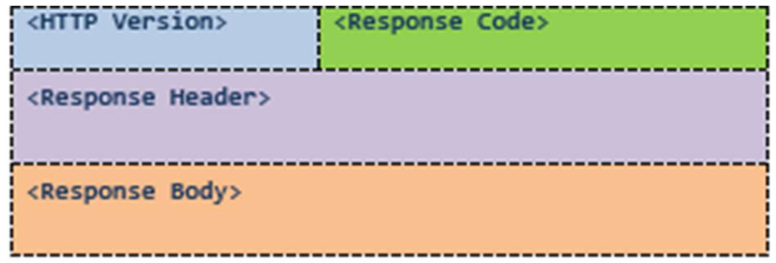

Figure 2: HTTP response format.

The server returns `<response code>`, which contains the status of the request. This response code is generally the 3-digit HTTP status code.

`<Response Header>` contains the metadata and settings about the response message.

`<Response Body>` contains the representation if the request was successful.

Listing Five is the actual response I received for the request cited in Listing Three:

Listing 5: An actual response to a GET request..

```
HTTP/1.1 200 OK
Date: Sat, 23 Aug 2014 18:31:04 GMT
Server: Apache/2
Last-Modified: Wed, 01 Sep 2004 13:24:52 GMT
Accept-Ranges: bytes
Content-Length: 32859
Cache-Control: max-age=21600, must-revalidate
Expires: Sun, 24 Aug 2014 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
strict.dtd">
<html xmlns='http://www.w3.org/1999/xhtml'>
<head><title>Hypertext Transfer Protocol -- HTTP/1.1</title></head>
<body>
...
```

The response code `200 OK` means that everything went well and the response message body contains a valid representation of the resource I requested. In this case, the representation is an HTML document that is declared by `Content-Type` header in the response header. The headers in the message are self-explanatory, but I will discuss some of them later in this article. There are many other attributes. You can catch and inspect such HTTP requests and responses using a free tool called Fiddler.

Addressing Resources

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.

Suppose we have a database of persons and we wish to expose it to the outer world through a service. A resource `person` can be addressed like this:

```
http://MyService/Persons/1
```

This URL has following format: `Protocol://ServiceName/ResourceType/ResourceID`

Here are some important recommendations for well-structured URIs:

- Use plural nouns for naming your resources.
- Avoid using spaces as they create confusion. Use an _ (underscore) or – (hyphen) instead.
- A URI is case insensitive. I use camel case in my URIs for better clarity. You can use all lower-case URIs.
- You can have your own conventions, but stay consistent throughout the service. Make sure your clients are aware of this convention. It becomes easier for your clients to construct the URIs programmatically if they are aware of the resource hierarchy and the URI convention you follow.
- A cool URI never changes; so give some thought before deciding on the URIs for your service. If you need to change the location of a resource, do not discard the old URI. If a request comes for the old URI, use status code `300` and redirect the client to the new location.
- Avoid verbs for your resource names until your resource is actually an operation or a process. Verbs are more suitable for the names of operations. For example, a RESTful service should not have the URIs `http://MyService/FetcthPerson/1` or `http://MyService/DeletePerson?id=1`.

Query Parameters in URI

The preceding URI is constructed with the help of a query parameter:

```
http://MyService/Persons?id=1
```

The query parameter approach works just fine and REST does not stop you from using query parameters. However, this approach has a few disadvantages.

- Increased complexity and reduced readability, which will increase if you have more parameters
- Search-engine crawlers and indexers like Google ignore URIs with query parameters. If you are developing for the Web, this would be a great disadvantage as a portion of your Web service will be hidden from the search engines.

The basic purpose of query parameters is to provide parameters to an operation that needs the data items. For example, if you want the format of the presentation to be decided by the client. You can achieve that through a parameter like this:

```
http://MyService/Persons/1?format=xml&encoding=UTF8
```

or

```
http://MyService/Persons/1?format=json&encoding=UTF8
```

Including the parameters `format` and `encoding` here in the main URI in a parent-child hierarchy will not be logically correct as they have no such relation:

```
http://MyService/Persons/1/json/UTF8
```

Query parameters also allow optional parameters. This is not otherwise possible in a URI. You should use query parameters only for the use they are intended for: providing parameter values to a process.

Uniform Interface

RESTful systems should have a uniform interface. HTTP 1.1 provides a set of methods, called verbs, for this purpose. Among these the more important verbs are:

Method	**Operation performed on server**	**Quality**
GET	Read a resource.	Safe
PUT	Insert a new resource or update if the resource already exists.	Idempotent
POST	Insert a new resource. Also can be used to update an existing resource.	N/A
DELETE	Delete a resource .	Idempotent
OPTIONS	List the allowed operations on a resource.	Safe
HEAD	Return only the response headers and no response body.	Safe

A Safe operation is an operation that does not have any effect on the original value of the resource. For example, the mathematical operation "divide by 1" is a safe operation because no matter how many times you divide a number by 1, the original value will not change. An Idempotent operation is an operation that gives the same result no matter how many times you perform it. For example, the mathematical operation "multiply by zero" is idempotent because no matter how many times you multiply a number by zero, the result is always same. Similarly, a Safe HTTP method does not make any changes to the resource on the server. An Idempotent HTTP method has same effect no matter how many times it is performed. Classifying methods as Safe and Idempotent makes it easy to predict the results in the unreliable environment of the Web where the client may fire the same request again.

`GET` is probably the most popular method on the Web. It is used to fetch a resource.

HEAD returns only the response headers with an empty body. This method can be used in a scenario when you do not need the entire representation of the resource. For example, HEAD can be used to quickly check whether a resource exists on the server or not.

The method OPTIONS is used to get a list of allowed operations on the resource. For example consider the request:

```
OPTIONS http://MyService/Persons/1 HTTP/1.1

HOST: MyService
```

The service after authorizing and authenticating the request can return something like:

```
200 OK

Allow: HEAD, GET, PUT
```

The second line contains the list of operations that are allowed for this client.

You should use these methods only for the purpose for which they are intended. For instance, never use GET to create or delete a resource on the server. If you do, it will confuse your clients and they may end up performing unintended operations. To illustrate, this let's consider this request:

```
GET http://MyService/DeletePersons/1 HTTP/1.1

HOST: MyService
```

By HTTP 1.1 specification, a GET request is supposed to fetch resources from the server. But it is very easy to implement your service such that this request actually deletes a Person. This request may work perfectly, but this is not a RESTful design. Instead, use the DELETE method to delete a resource like this:

```
DELETE http://MyService/Persons/1 HTTP/1.1

HOST: MyService
```

REST recommends a uniform interface and HTTP provides you that uniform interface. However, it is up to service architects and developers to keep it uniform.

Difference between PUT and POST

The short descriptions of these two methods I provided above are almost the same. These two methods confuse a lot of developers. So let's discuss these separately.

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.

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.

Statelessness

A RESTful service is stateless and does not maintain the application state for any client. A request cannot be dependent on a past request and a service treats each request independently. HTTP is a stateless protocol by design and you need to do something extra to implement a stateful service using HTTP. But it is really easy to implement stateful services with current technologies. We need a clear understanding of a stateless and stateful design so that we can avoid misinterpretation.

A stateless design looks like so:

Request1: `GET http://MyService/Persons/1 HTTP/1.1`

Request2: `GET http://MyService/Persons/2 HTTP/1.1`

Each of these requests can be treated separately.

A stateful design, on the other hand, looks like so:

Request1: `GET http://MyService/Persons/1 HTTP/1.1`

Request2: `GET http://MyService/NextPerson HTTP/1.1`

To process the second request, the server needs to remember the last `PersonID` that the client fetched. In other words, the server needs to remember the current state — otherwise Request2 cannot be processed. Design your service in a way that a request never refers to a previous request. Stateless services are easier to host, easy to maintain, and more scalable. Plus, such services can provide better response time to requests, as it is much easier to load balance them.

Links Between Resources

A resource representation can contain links to other resources like an HTML page contains links to other pages. The representations returned by the service should drive the process flow as in case of a website. When you visit any website, you are presented with an index page. You click one of the links and move to another page and so on. Here, the representation is in the HTML documents and the user is driven through the website by these HTML documents themselves. The user does not need a map before coming to a website. A service can be (and should be) designed in the same manner.

Let's consider the case in which a client requests one resource that contains multiple other resources. Instead of dumping all these resources, you can list the resources and provide links to them. Links help keep the representations small in size.

For an example, if multiple `Persons` can be part of a `Club`, then a `Club` can be represented in `MyService` as in Listing Six:

Listing Six: A Club with links to Persons.

```
<Club>
    <Name>Authors Club</Name>
    <Persons>
        <Person>
            <Name>M. Vaqqas</Name>
            <URI>http://MyService/Persons/1</URI>
        </Person>
        <Person>
            <Name>S. Allamaraju</Name>
            <URI>http://MyService/Persons/12</URI>
        </Person>
    </Persons>
</Club>
```

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.

Caching can be controlled using these HTTP headers:

Header	Application
`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.

Values of these headers can be used in combination with the directives in a `Cache-Control` header to check if the cached results are still valid or not. The most common directives for `Cache-Control` header are:

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.

You have seen some of these headers and directives above in Listing Five. Depending on the nature of the resources, a service can decide the values of these headers and directives. For example, a service providing stock market updates would keep the cache age limit to as low as possible or even turn off caching completely as this is a critical information and users should get the latest results all the time. On the other hand, a public picture repository whose contents do not change so frequently would use a longer caching age and slack caching rules. The server, the client, and any intermediate component between them should follow these directives to avoid outdated information getting served.

Documenting a RESTful Service

RESTful services do not necessarily require a document to help clients discover them. Due to URIs, links, and a uniform interface, it is extremely simple to discover RESTful services at runtime. A client can simply know the base address of the service and from there it can discover the service on its own by traversing through the resources using links. The method OPTION can be used effectively in the process of discovering a service.

This does not mean that RESTful services require no documentation at all. There is no excuse for not documenting your service. You should document every resource and URI for client developers. You can use any format for structuring your document, but it should contain enough information about resources, URIs, Available Methods, and any other information required for accessing your service. The Table below is a sample documentation of MyService. This is a simple and short document that contains all the aspects of MyService and should be sufficient for developing a client.

```
Service Name: MyService

Address: http://MyService/
```

Resource	Methods	URI	Description
Person	GET, POST, PUT, DELETE	http://MyService/Persons/{PersonID}	Contains information about a person {PersonID} is optional **Format:** text/xml
Club	GET, POST, PUT	http://MyService/Clubs/{ClubID}	Contains information about a club. A club can be joined my multiple people {ClubID} is optional **Format:** text/xml

Search	GET	http://MyService/Search?	Search a person or a club **Format:** text/xml **Query Parameters:** Name: String, Name of a person or a club Country: String, optional, Name of the country of a person or a club Type**:** String, optional, Person or Club. If not provided then search will result in both Person and Cubs

You may also like to document the representations of each resource and provide some sample representations.