REST API

Table of Contents

Contents

[Introduction 4](#_Toc483963298)

[What is an API? 4](#_Toc483963299)

[What is REST? 4](#_Toc483963300)

[Examples:- 5](#_Toc483963301)

[Anatomy of a REST URL 5](#_Toc483963302)

[Nested Resources 6](#_Toc483963303)

[HTTP Status Codes 6](#_Toc483963304)

[Success codes: 7](#_Toc483963305)

[User error codes: 7](#_Toc483963306)

[API response formats 7](#_Toc483963307)

[RestFul Web Services 8](#_Toc483963308)

[Rest Server Implementation 9](#_Toc483963309)

[Jersey Rest 9](#_Toc483963310)

[Grizzly Web Server 9](#_Toc483963311)

[Rest Client Implementation 10](#_Toc483963312)

[Unirest for Java 10](#_Toc483963313)

[Features 10](#_Toc483963314)

[Request 10](#_Toc483963315)

[Creating Request 11](#_Toc483963316)

[Route Parameters 11](#_Toc483963317)

[Custom Entity Body 11](#_Toc483963318)

[Response 11](#_Toc483963319)

[Basic Authentication 12](#_Toc483963320)

[Concurrency 12](#_Toc483963321)

[Timeouts 12](#_Toc483963322)

[Exiting an application 12](#_Toc483963323)

[TOOLS 14](#_Toc483963324)

[POSTMAN 14](#_Toc483963325)

[IMPORTANT LINKS 16](#_Toc483963326)

[Unirest tutorials 16](#_Toc483963327)

[POSTMAN 16](#_Toc483963328)

[Example 16](#_Toc483963329)

# Introduction

## What is an API?

For example, let's say we have an application that allows you to create, read, edit, and delete widgets (CRUD operations). We could create an HTTP API that allows us to perform these functions:

http://example.com/view\_widgets

http://example.com/create\_new\_widget?name=Widgetizer

http://example.com/update\_widget?id=123&name=Foo

http://example.com/delete\_widget?id=123

A problem has started to arise when everyone starts implementing their own APIs. Without a standard way of naming URLs, we always have to refer to the documentation to understand how the API works. One API might have a URL like /view\_widgets whereas another API might use /widgets/all.

## What is REST?

REST (**RE**presentational **S**tate **T**ransfer) is a web standards based architecture which uses HTTP Protocol for data communication. This term is invented by Roy Fielding to describe a standard way of creating HTTP APIs. He noticed that the four common actions (view, create, edit, and delete) map directly to HTTP verbs that are already implemented GET,  POST,  PUT,  DELETE.

In REST architecture, a **REST Server** simply provides access to resources and the **REST client** accesses and presents the resources. Here each resource is identified by **URIs/Global IDs**. REST uses various representations to represent a resource like Text, JSON and XML. JSON is now the most popular format being used in Web Services

5 basic fundamentals of Rest are

1. Everything is **resource** – Image, file, data, video etc.
2. Every resource is identified by **Unique Identifier**. – Every URI [Rest URLs]
3. Use simple and **uniform interface.**– Keep your interfaces simple
   1. GET Get resource
   2. PUT Create & Update resource
   3. DELETE Delete
   4. POST Submit data to resource
   5. OPTIONS
4. All communication done via **representation (request & response between client and server)**.
   1. Can be sent using **XML or JSON format**
5. Stateless – Every request will be independent request.

### Examples:-

If we wanted to view all widgets, the URL would look like this:

**GET** http://example.com/widgets

Create a new widget by posting the data:

**POST** http://example.com/widgets

Data:

name = Foobar

To view a single widget we "get" it by specifying that widget's id:

GET http://example.com/widgets/123

Update that widget by "putting" the new data:

PUT http://example.com/widgets/123

Data:

name = New name

color = blue

Delete that widget:

DELETE http://example.com/widgets/123

## Anatomy of a REST URL

We might have noticed from the previous examples that REST URLs use a consistent naming scheme. When we are interacting with an API, we are almost always manipulating some sort of object. In our examples, this is a Widget. In REST terminology, this is called a **Resource**. The first part of the URL is always the plural form of the resource:

/widgets

/Instrument

This is always used when referring to this collection of resources ("list all" and "add one" actions). When you are working with a specific resource, you add the ID to the URL.

/widgets/123

This URL is used when you want to "view", "edit", or "delete" the particular resource.

### Nested Resources

Let's say our widgets have many users associated with them. What would this URL structure look like?

List all:

GET /widgets/123/users

Add one:

POST /widgets/123/users

Data:

name = Andrew

Nested resources are perfectly acceptable in URLs. However, it's not a best practice to go more than two levels deep. It's not necessary because you can simply refer to those nested resources by ID rather than nesting them within their parents. For example:

/widgets/123/users/456/sports/789

...can be referenced as:

/users/456/sports/789

...or even:

/sports/789

## HTTP Status Codes

Another important part of REST is responding with the correct status code for the type of request that was made. If you're new to HTTP status codes, here is a quick summary. When you make an HTTP request, the server will respond with a code which corresponds to whether or not the request was successful and how the client should proceed. There are four different levels of codes:

* 2xx = Success
* 3xx = Redirect
* 4xx = User error
* 5xx = Server error

Here's a list of the most important status codes:

### Success codes:

* 200 - OK (the default)
* 201 - Created
* 202 - Accepted (often used for delete requests)

### User error codes:

* 400 - Bad Request (generic user error/bad data)
* 401 - Unauthorized (this area requires you to log in)
* 404 - Not Found (bad URL)
* 405 - Method Not Allowed (wrong HTTP method)
* 409 - Conflict (i.e. trying to create the same resource with a PUT request)

## API response formats

When you make an HTTP request, you can request the format that you want to receive. For example, making a request for a webpage, you want the format to be in HTML, or if you are downloading an image, the format returned should be an image. However, it's the server's responsibility to respond in the format that was requested.

JSON has quickly become the format of choice for REST APIs. It has a lightweight, readable syntax that can be easily manipulated. So when a user of our API makes a request and specifies JSON as the format they would prefer:

GET /widgets

Accept: application/json

...our API will return an array of widgets formatted as JSON:

[

{

id: 123,

name: 'Simple Widget'

},

{

id: 456,

name: 'My other widget'

}

]

If the user requests a format that we haven't implemented, what do we do? You can throw some type of error, but I would recommend enforcing JSON as your standard response format. It's the format that your developers will want to use. No reason to support other formats unless you already have an API which needs to be supported.

# 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.

Web services **based on REST Architecture** are known as **RESTful Web Services.** These web services use HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI (Uniform Resource Identifier), which is a service that provides resource representation such as JSON and a set of HTTP Methods.

***RESTful***  (REpresentational State Transfer) ***API*** (Application Programming Interface) ***DLs*** (Description Languages) are formal languages designed to provide a structured description of a RESTful web API that is useful both to a human and for automated machine processing. API Description Languages are sometimes called interface description languages (IDLs). The structured description might be used to generate documentation for human programmers; such documentation may be easier to read than free-form documentation, since all documentation generated by the same tool follows the same formatting conventions. Additionally, the description language is usually precise enough to allow automated generation of various software artifacts, like libraries, to access the API from various programming languages, which takes the burden of manually creating them off the programmers.

There are two previous major description languages: **WSDL2.0** (Web Services Description Language) and **WADL** (Web Application Description Language). Neither is widely adopted in the industry for describing RESTful APIs, citing poor human readability of both and WADL being actually unable to fully describe a RESTful API

# Rest Server Implementation

## Jersey Rest

Jersey RESTful Web Services is an open source framework for developing RESTful Web Services in Java. Jersey provides support for **JAX-RS** APIs and serves as a JAX-RS Reference Implementation. Jersey simplifies the development of RESTful Web services and their clients in Java in a standard and portable way.

Representational State Transfer (REST) is a software architecture style for creating scalable web services. REST is a simpler alternative to SOAP and WSDL-based Web services and has achieved a great deal of popularity. RESTful systems communicate using the Hypertext Transfer Protocol (HTTP) using the same verbs (GET, POST, PUT, DELETE, etc.) that web browsers use to retrieve web pages and send data to remote servers.

The Java API for RESTful Web Services (JAX-RS) provides portable APIs for developing, exposing and accessing Web applications designed and implemented in compliance with principles of REST architectural style. The latest release of JAX-RS is version 2.0. **The Jersey framework is the reference implementation of JAX-RS**.

## Grizzly Web Server

Project Grizzly is a pure Java web service built using the NIO API. Grizzly's main use case is the web server component for the GlassFish application server. With Grizzly we can build scalable and **robust servers** using NIO as well as offering extended framework components including Web Framework (HTTP/S), WebSocket, Comet and more.

# Rest Client Implementation

Java REST client can be implemented using various available libraries such as:

* Netflix Feign
* Jersey
* Spring RestTemplate
* unirest

## Unirest for Java

Unirest is a set of lightweight HTTP Request client libraries which are available in multiple languages, built and maintained by Mashape.

### Features

* Make GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS requests
* Both **synchronous and asynchronous** (non-blocking) requests
* It supports form parameters, file uploads and custom body entities
* Easily add route parameters without ugly string concatenations
* Supports gzip
* Supports **Basic Authentication** natively
* Customizable timeout, concurrency levels and proxy settings
* Customizable default headers for every request (DRY)
* Customizable HttpClient and HttpAsyncClient implementation
* Automatic JSON parsing into a native object for JSON responses
* Customizable binding, with mapping from response body to java Object

### Request

We can start building our request by creating a HttpRequest object using one of the following:

GetRequest request = Unirest.get(String url);

GetRequest request = Unirest.head(String url);

HttpRequestWithBody request = Unirest.post(String url);

HttpRequestWithBody request = Unirest.put(String url);

HttpRequestWithBody request = Unirest.patch(String url);

HttpRequestWithBody request = Unirest.options(String url);

HttpRequestWithBody request = Unirest.delete(String url);

### Creating Request

Using Unirest it is very easy to make requests as shown below.

HttpResponse<JsonNode> jsonResponse = Unirest.post("http://httpbin.org/post")

.header("accept", "application/json")

.queryString("apiKey", "123")

.field("parameter", "value")

.field("foo", "bar")

.asJson();

### Route Parameters

Sometimes you want to add dynamic parameters in the URL, you can easily do that by adding a placeholder in the URL, and then by setting the route parameters with the routeParam function, like:

Unirest.get("http://httpbin.org/{method}")

.routeParam("method", "get")

.queryString("name", "Mark")

.asJson();

In the example above the final URL will be http://httpbin.org/get - Basically the placeholder {method} will be replaced with get.

The placeholder's format is as easy as: {custom\_name}

### Custom Entity Body

HttpResponse<JsonNode> jsonResponse = Unirest.post("http://httpbin.org/post")

.header("accept", "application/json")

.body("{\"parameter\":\"value\", \"foo\":\"bar\"}")

.asJson();

### Response

Upon recieving a response Unirest returns the result in the form of an Object, this object should always have the same keys for each language regarding to the response details.

* .getStatus() - HTTP Response Status Code (Example: 200)
* .getStatusText() - HTTP Response Status Text (Example: "OK")
* .getHeaders() - HTTP Response Headers
* .getBody() - Parsed response body where applicable, for example JSON responses are parsed to Objects / Associative Arrays.
* .getRawBody() - Un-parsed response body

### Basic Authentication

Authenticating the request with basic authentication can be done by calling the basicAuth(username, password) function:

HttpResponse<JsonNode> response = Unirest.get("http://httpbin.org/headers").basicAuth("username", "password").asJson();

### Concurrency

You can set custom concurrency levels if you need to tune the performance of the syncronous or asyncronous client:

Unirest.setConcurrency(int maxTotal, int maxPerRoute);

By default the maxTotal (overall connection limit in the pool) is 200, and the maxPerRoute (connection limit per target host) is 20.

### Timeouts

You can set custom connection and socket timeout values (in **milliseconds**):

Unirest.setTimeouts(long connectionTimeout, long socketTimeout);

By default the connection timeout (the time it takes to connect to a server) is 10000, and the socket timeout (the time it takes to receive data) is 60000. You can set any of these timeouts to zero to disable the timeout.

### Exiting an application

Unirest starts a background event loop and your Java application won't be able to exit until you manually shutdown all the threads by invoking:

Unirest.shutdown();

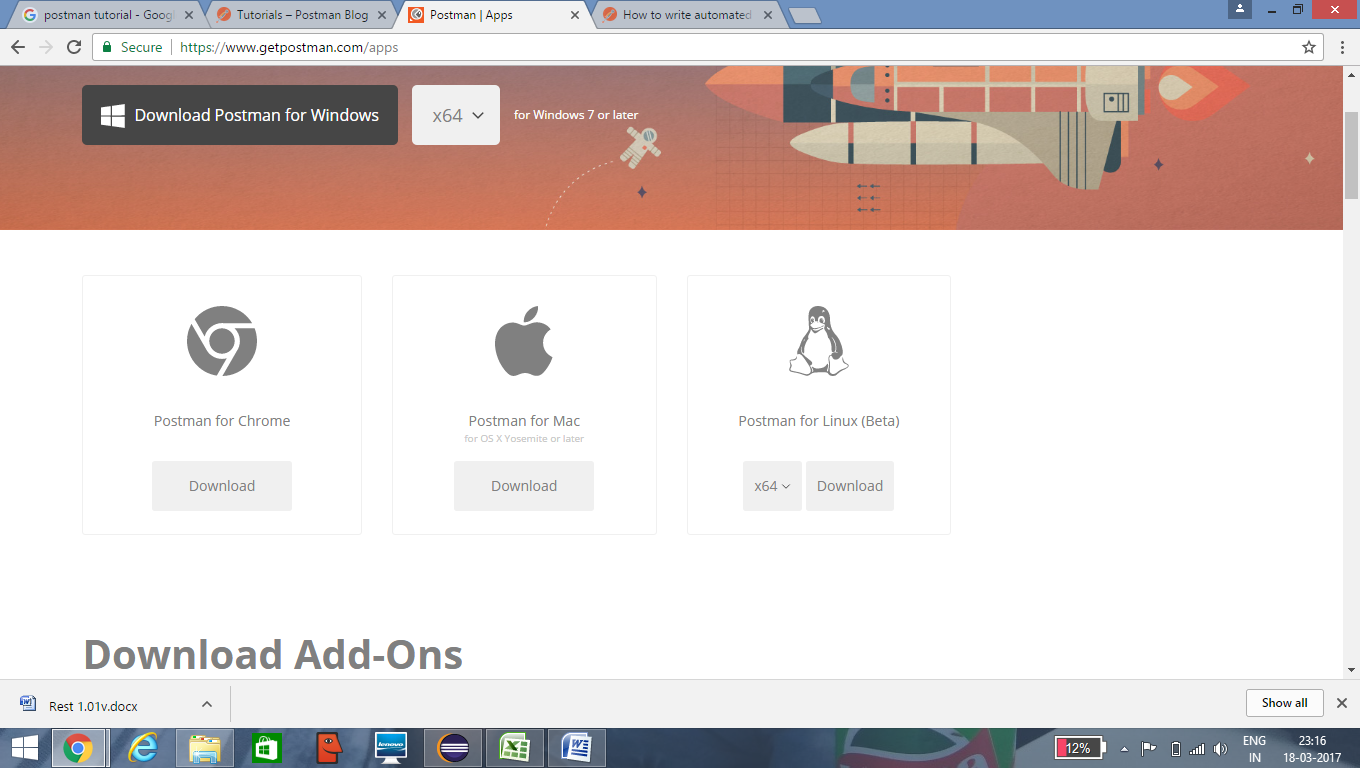
# TOOLS

## POSTMAN

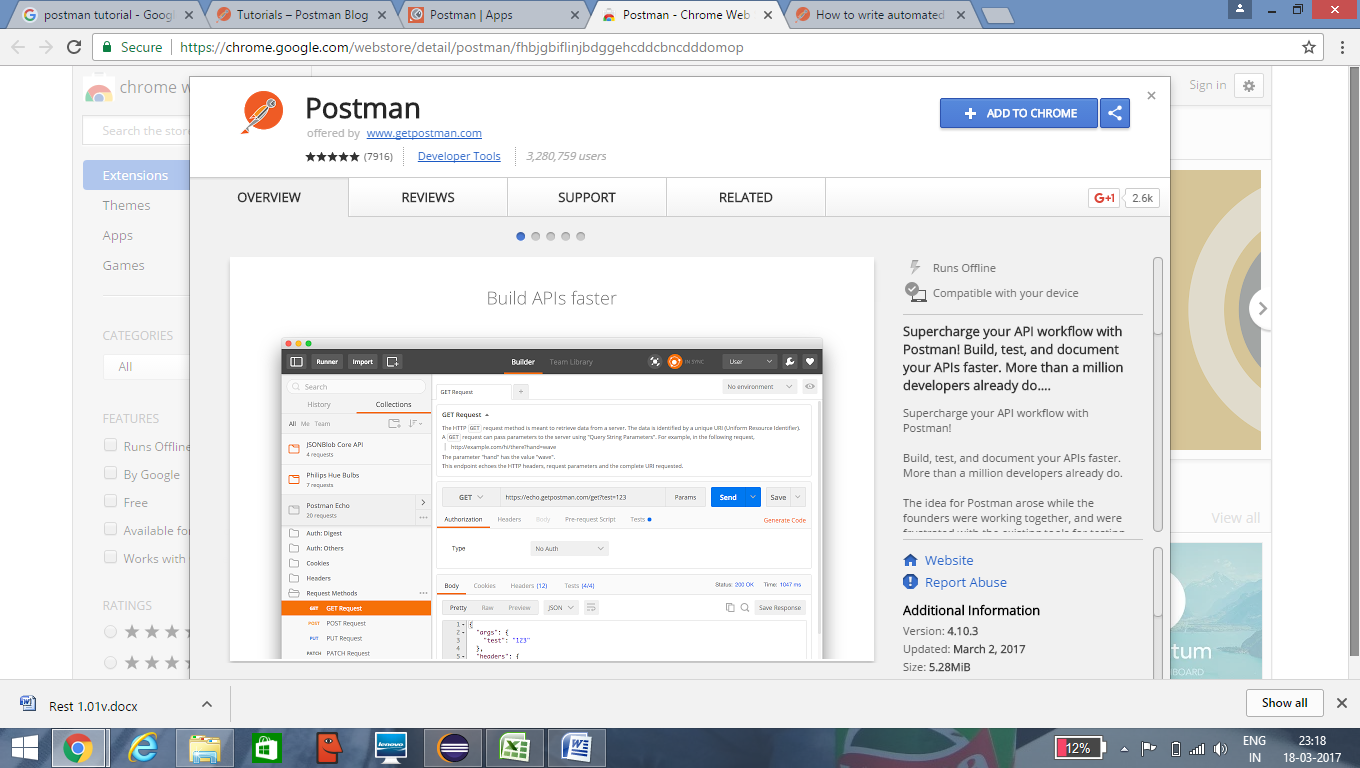
This tool can be easily integrated with browser and can be used to test HTTP APIs. This can be used to run automated tests on our requests also.

Download

Go to url https://www.getpostman.com/apps and click download as per your requirement as shown below.



Click **"ADD TO CHROME"**



# IMPORTANT LINKS

## Unirest tutorials

http://unirest.io/java.html

http://howtoprogram.xyz/2016/07/27/java-rest-client-using-unirest-java-api/

## POSTMAN

https://www.getpostman.com/docs/

http://blog.getpostman.com/2014/03/07/writing-automated-tests-for-apis-using-postman/

## Example

https://www.openfigi.com/api