**Perquisites to learn REST API**

Basic computer literacy, a basic understanding of HTML and CSS JavaScript basics (see first steps building blocks JavaScript objects).

REST is an architectural style, or design pattern, for APIs.

*Who invented REST?*

*Roy Fielding, a computer scientist, defined REST. He presented the REST principles in his PhD dissertation in 2000.*

Before we dive into what makes an API RESTful and what constraints and rules you should follow if you want to create RESTful APIs, let’s explain 2 key terms:

1. **Client** — the client is the person or software who uses the API. It can be a developer, for example you, as a developer, can use Twitter API to read and write data from Twitter, create a new tweet and do more actions in a program that you write. Your program will call Twitter’s API. The client can also be a web browser. When you go to Twitter website, your browser is the client who calls Twitter API and uses the returned data to render information on the screen.
2. **Resource** — a resource can be any object the API can provide information about. In Instagram’s API, for example, a resource can be a user, a photo, a hashtag. Each resource has a unique identifier. The identifier can be a name or a number.

Now let’s get back to REST.

A RESTful web application exposes information about itself in the form of information about its resources. It also enables the client to take actions on those resources, such as create new resources (i.e. create a new user) or change existing resources (i.e. edit a post).

In order for your APIs to be RESTful, you have to follow a set of constraints when you write them. The REST set of constraints will make your APIs easier to use and also easier to discover, meaning a developer who is just starting to use your APIs will have an easier time learning how to do so.

REST stands for **RE**presentational **S**tate **T**ransfer.

It means when a RESTful API is called, the server will *transfer* to the client a *representation* of the *state* of the requested resource.

For example, when a developer calls Instagram API to fetch a specific user (the resource), the API will return the state of that user, including their name, the number of posts that user posted on Instagram so far, how many followers they have, and more.

The representation of the state can be in a JSON format, and probably for most APIs this is indeed the case. It can also be in XML or HTML format.

What the server does when you, the client, call one of its APIs depends on 2 things that you need to provide to the server:

1. An identifier for the resource you are interested in. This is the URL for the resource, also known as the **endpoint**. In fact, URL stands for Uniform Resource Locator.
2. The operation you want the server to perform on that resource, in the form of an **HTTP method**, or **verb**. The common HTTP methods are GET, POST, PUT, and DELETE.

For example, fetching a specific Twitter user, using Twitter’s RESTful API, will require a URL that identify that user and the HTTP method GET.

Another example, this URL: [**www.twitter.com/jk\_rowling**](http://www.twitter.com/jk_rowling) has the unique identifier for J. K. Rowling’s Twitter user, which is her username, jk\_rowling. Twitter uses the username as the identifier, and indeed Twitter usernames are unique — there are no 2 Twitter users with the same username.

The HTTP method **GET** indicates that we want to **get** the state of that user.

In order for an API to be RESTful, it has to adhere to 6 constraints:

* Uniform interface
* Client — server separation
* Stateless
* Layered system
* Cacheable
* Code-on-demand

**Uniform interface**

This constraint has 4 parts:

1. The request to the server has to include a resource identifier
2. The response the server returns include enough information so the client can modify the resource
3. Each request to the API contains all the information the server needs to perform the request, and each response the server returns contain all the information the client needs in order to understand the response.
4. Hypermedia as the engine of application state — this may sound a bit cryptic, so let’s break it down: by application we mean the web application that the server is running. By hypermedia we refer to the hyperlinks, or simply links, that the server can include in the response. The whole sentence means that the server can inform the client , in a response, of the ways to change the state of the web application. If the client asked for a specific user, the server can provide not only the state of that user but also information about how to change the state of the user, for example how to update the user’s name or how to delete the user. It is easy to think about the way it is done by thinking about a server returning a response in HTML format to a browser (which is the client). The HTML will include tags with links (this is the hypermedia part) to another web page where the user can be updated (for example a link to a ‘profile settings’ page). To put all of this in perspective, most web **pages** do implement hypermedia as the engine of application state, but the most common web **APIs** do not adhere to this constraint. To further understand this concept, I highly recommend watching [this](https://www.youtube.com/watch?v=6UXc71O7htc) 30 minutes YouTube video.

The result of the uniform interface is that requests from different clients look the same, whether the client is a chrome browser, a linux server, a python script, an android app or anything else.

**Client — server separation**

The client and the server act independently, each on its own, and the interaction between them is only in the form of requests, initiated by the client only, and responses, which the server send to the client only as a reaction to a request. The server just sits there waiting for requests from the client to come. The server doesn’t start sending away information about the state of some resources on its own.

**Stateless**

Stateless means the server does not remember anything about the user who uses the API. It does not remember if the user of the API already sent a GET request for the same resource in the past, it does not remember which resources the user of the API requested before, and so on.

Each individual request contains all the information the server needs to perform the request and return a response, regardless of other requests made by the same API user.

**Layered system**

Between the client who requests a representation of a resource’s state, and the server who sends the response back, there might be a number of servers in the middle. These servers might provide a security layer, a caching layer, a load-balancing layer, or other functionality. Those layers should not affect the request or the response. The client is agnostic as to how many layers, if any, there are between the client and the actual server responding to the request.

**Cacheable**

This means that the data the server sends contain information about whether or not the data is cacheable. If the data is cacheable, it might contain some sort of a version number. The version number is what makes caching possible: since the client knows which version of the data it already has (from a previous response), the client can avoid requesting the same data again and again. The client should also know if the current version of the data is expired, in which case the client will know it should send another request to the server to get the most updated data about the state of a resource.

**Code-on-demand**

This constraint is optional — an API can be RESTful even without providing code on demand.

The client can request code from the server, and then the response from the server will contain some code, usually in the form of a script, when the response is in HTML format. The client then can execute that code.