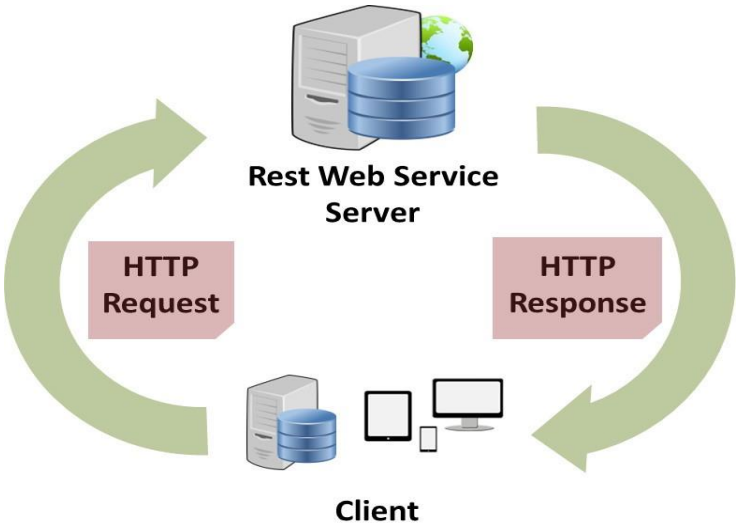
Overview

A REST API (also cited as RESTful API) is a application programming interface. (API or web API) that conforms to the constraints of REST variety style and allows for interaction with RESTful web services. REST stands for representational state transfer and was created by computer user Roy Fielding.

What's an API?

An API could also be grouped as of definitions and protocols for building and integrating application software. It's sometimes mentioned to as a contract between an information provider and an information user—establishing the content required from the patron (the call) and therefore the content required by the producer (the response). For instance, the API design for a weather service could specify that the user supply a zipper code which the producer reply with a 2-part answer, the primary being the warm temperature, and the second being the low.



In other words, if you wish to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you would like thereto system so it can understand and fulfil the request.

You can think about API as a mediator between the users or clients and therefore the resources or web services they require to induce. It's also a way for a corporation to share resources and information while maintaining security, control, and authentication—determining who gets access to what.

Another advantage of an API is that you simply don't have to know the specifics of caching—how your resource is retrieved or where it comes from.

What is the REST API style?

REST is commonly described as an architecture style.

It will be described as Set of formal and informal guides for making architectures — "constraints"

- Client-server
- Stateless
- Cacheable
- Uniform interface
- Layered system
- Code on demand (optional)

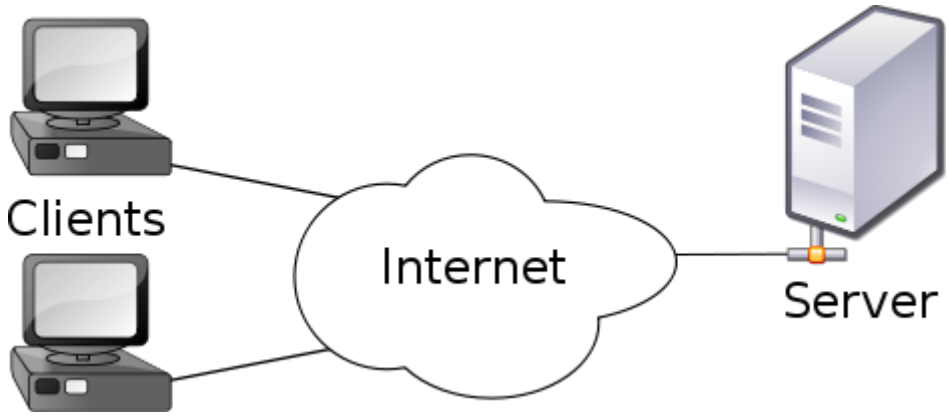
When we use REST style API an Application can be interact with resource by knowing only two things :

- Identifier of the resource
- Action to be performed on the resource

Understanding Constraints of REST style Architecture

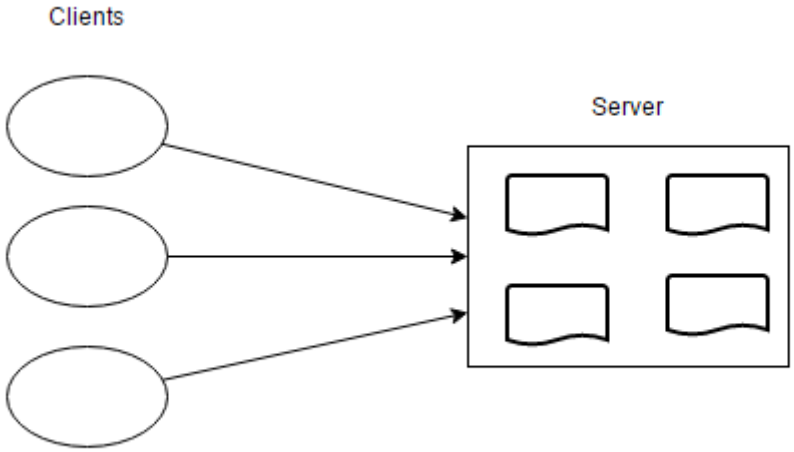
1. Client Server

Separation of concerns is that the principle behind the client-server constraints. By separating the interface that concerns from the information storage concerns, we improve the portability of the user interaction program across multiple platforms and improve scalability by simplifying the server components.



2. Stateless

Statelessness means communication must be stateless in nature as within the client stateless server style, i.e. Each request from client to server must contain all of the knowledge necessary to go through the request, and can't benefit of any stored context on the server. The Session state is therefore combine entirely on the client.



3. Cacheable

In order to enhance network efficiency, cache constraints are added to the REST style.

Cache constraints require that the info within a response to an invitation be implicitly or explicitly noted as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the correct command to reuse that response data for later, equivalent requests.

The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and user-perceived performance by reducing the common latency of a series of interactions.

4. Uniform interface

The central feature that distinguishes the REST API style from other network-based styles is its emphasis on the same interface between components.

Resources are simply notions that can be found using URIs. URIs inform clients that a concept exists elsewhere. The client then selects a specific representation of the notion from the representations provided by the server. Web content, for example, could be a representation of a resource.

5. A multi-layered system We apply layered system limitations to further optimise behaviour for Internet-scale requirements.

The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior specified each component cannot “see” beyond the immediate layer with which they're interacting.

6. Code on demand

The code-on-demand style is the final addition to our REST constraint set.

REST allows client functionality to be extended by downloading and executing code with the sort of applets or scripts.

This simplifies clients by reducing the quantity of features required to be pre-implemented. Adding the ability to download features after they've been deployed improves the system's extensibility.

Creating a RESTful service

Steps

1. Define the domain and data.
2. Organize the information into groups.
3. Create URI to resource mapping.
4. Define the representations to the client (XML, HTML, CSS, …).
5. Link data across resources (connectedness or hypermedia).
6. Create use cases to map events/usage.
7. Plan for things going wrong.

Creating a REST API

which is on the way of organizing our code so we are able to access our data from multiple applications. Your REST API is server code whose job it's to supply access to your data and to enforce rules like who can see what. Then other programs use your REST API to interact along with your data.

To put it another way: a REST API is simply an internet app, very like to all of the net apps we’ve already built. The sole difference is that rather than of showing a web site for a GET request, it provides data. And rather than of using HTML forms to create a content of POST request, it takes POST requests from other applications! (Of course, one amongst those applications can be another web app that gets user input using HTML forms!)

REST

REST stands for representational state transfer, which is simply a unique name for a collection of rules that you can simply follow to make a online app that opens access to data in during a way that's reusable by multiple applications, or many users of the identical application. REST doesn't actually involve any new technical concepts. It's more some way of using the concepts we've already learned.

There are some basic ideas behind REST:

- You access data via URLs. as an example, /users/Ada allows you to access data about a couple of person named Ada.
- You use HTTP methods to access or change data For example, a GET call to /people/Ada will display Ada's data, whereas a POST request to /people/Ada will alter Ada's data. You are able to use the opposite side HTTP methods (like PUT or DELETE) to interact with the information in addition.
- You can represent data however you want. For instance that GET request might return a JSON string that represents the user data. The JSON string might then be included in the POST request.. Or it could take a binary string, or XML, or an inventory of properties. It's up to you.
- Each request should be standalone. In other words, you ought to not store session information on the server! Everything needed to full fill a call of request for participation must be included within the request itself!

All of those "rules" exist for a reason, but it's important to stay fed in mind that within the end, everything is up to you. You're the programmer. The REST police aren't visiting to come kick your door down if your code "violates" one among these rules. You must strictly treat REST as a tool, not as a strict set of rules that you just must follow in any respect of costs. Do what makes sense to you and is appropriate in your situation..

API stands for application programmer interface, which may be a fancy name for whatever a programmer uses to interact with a language or library. For example, Processing's reference is an API: it's the classes and functions we are supposed to write the Processing code. Similarly, the Java API is that the list of classes and functions we use to write Java code. You can view JavaScript's API on MDN. The major purpose is that an API could be a collection of things we are able do when writing code. So once we say we're creating a REST API, we just mean that we're using REST ideas to form impact something that programmers can use to interact with our data.

Simple Example REST API

```
import java.util.HashMap;

import java.util.Map;


/**

 * Example DataStore class that gives access to user data.

 * Pretend this class accesses a database.

 */

public class DataStore {


        //Map of names to Person instances.

        private Map<String, Person> personMap = new HashMap<>();
```

```
        //this class is a singleton that should not be instantiated directly!
        private static DataStore instance = new DataStore();
        public static DataStore getInstance(){
                return instance;
        }

        //private constructor so people know to use the getInstance() function instead
        private DataStore(){
                //dummy data
                personMap.put("Ada", new Person("Ada", "Ada Lovelace was the first programmer.", 1815));
                personMap.put("Kevin", new Person("Kevin", "Kevin is that  the author of HappyCoding.io.",
1986));
                personMap.put("Stanley", new Person("Stanley", "Stanley is Kevin's cat.", 2007));
        }

        public Person getPerson(String name) {
                return personMap.get(name);
        }

        public void putPerson(Person person) {
                personMap.put(person.getName(), person);
        }
}
```

References :

https://happycoding.io/tutorials/java-server/rest-api

https://medium.com/@sagar.mane006/understanding-rest-representational-state-transfer-85256b9424aa

https://www.ibm.com/cloud/learn/rest-apis

https://www.altexsoft.com/blog/rest-api-design/

https://karen-su-1010.medium.com/a-summary-note-for-rest-api-design-best-practice-f86baff4453b