REST and APIs

REST

The motivation behind the development of REST was to create a design pattern for how the Web should work, so that it could act as a guiding framework for the Web standards and when designing Web services.

REST-style architectures consist of clients and servers. Clients initiate requests to servers who process these requests and return responses based on these requests. These requests and responses are built around the transfer of representations of these resources. A resource can be any coherent and meaningful concept that can be addressed, while a representation of a resource is a document that captures the intended state of a resource. Fundamentally in REST each resource is first identified using a URL and a new resource for every service required is created. The data returned by the service must be linked to the other data, hence making it in to a network of information unlike the Object Oriented design which encourages the encapsulation of information.

REST architectural style describes six constraints applied to architecture:

1. Uniform Interface
Individual resources are identified using URLS. The resources (database) are themselves different from the representation (XML, JSON, HTML) sent to the client. The client can manipulate the resource through the representations provided they have the permissions. Each message sent between the client and the server is self-descriptive and includes enough information to describe how it is to be processed. The hypermedia that is hyperlinks and hypertext act as the engine for state transfer.

2. Stateless Interactions
None of the clients context is to be stored on the server side between the request. All of the information necessary to service the request is contained in the URL, query parameters, body or headers.

3. Cacheable
Clients can cache the responses. The responses must define themselves as cacheable or not to prevent the client from sending the inappropriate data in response to further requests.

4. Client-Server
The clients and the server are separated from each other thus the client is not concerned with the data storage thus the portability of the server. Client code is improved while on the server side the server is not concerned with the client interference thus the server is simpler and easy to scale.

5. Layered System
At any time client cannot tell if it is connected to the end server or to an intermediate. The intermediate layer helps to enforce the security policies and improve the system scalability by enabling load-balancing

6. Code on Demand
An optional constraint where the server temporarily extends the functionality of a client by the transfer of executable code. E.g. JavaScript

What makes a good REST API?

Building a fully functional API consists of multiple pieces that are eventually connected together, such as authentication, integration with external services and the core application logic. Here we will see what it takes to build a full API, and what you should consider when the time comes to build one.

REST resources:

Resource names

One of the first questions that comes to your mind when designing an API is probably this: What should a URI look like? A URI is the address to a resource, such as a user account. A URI might look like this: “/users/1/products/2”. Should these resources be singular or plural?

This has been a controversial topic for a long time. Both forms are seen commonly, but it is generally more acceptable to use plural names for all resources.

HTTP methods

It might seem tempting to delete a user using the following GET request: “/users/1/delete”. However, you should always use HTTP methods meant for that type of request; This way you don’t have to set up too many redundant URIs. You can simply call the same resource name with a different method, and the action will be different for each type of method.

GET requests should be used for ready-only queries, in other words, for getting data. POST requests should be used to create a new resource. PUT is the right method to use for updating an existing resource. If you only want to update a small portion of a resource, such as an email address, use PATCH instead. To delete a resource, use DELETE.

There are more HTTP methods available, but there are the most common ones you will likely implement in your API.

API keys

An API key should never be stored in the URI: URIs are public and can be shared. API keys are like passwords and thus should never be shared. It is good practice to send an API using HTTP headers, more specifically the authorisation header. Keep in mind that all APIs implementing this type of authorisation should use HTTPS to prevent the API key from being sent in plaintext.

Versioning an API

Versioning is something you should pay special attention to from the very beginning of development. If you believe your API will have multiple versions running at the same time, your API should be versioned. There are multiple ways to do this for APIs. Two popular ways are to send the Accept header from the front end to the API or to send the version in the URI of the API call, such as “GET /v1/users/1”.

The second approach is slightly controversial because it is not a fully RESTful approach. However, it is a very simple way and used by many popular APIs.

When you version your API, you can divide the core application logic into different namespaces and load code from a namespace based on what version the client requests.

Other API elements:

Authentication

If you application has user accounts, you will need to handle authentication with your API. The first thing you should do is decide how users authenticate. Is it a traditional username and password combination, an API key, or perhaps something more advanced like OAuth 2? Either way, if your application handles login details, you should also set up an SSL certificate for your API.

Permission Management

Whenever a user tries to do something, for example to modify the profile of another user, a piece of code is executed first to check whether the user has the correct role to perform that action. If yes, the execution continues normally, and if not, a permission denied error is returned.

CORS

If your front and back ends are located in a different address determined by the same-origin policy, you have to set up cross-origin resource sharing (CORS) on the API server. If you are on a shared web hosting plan, you can do this using the PHP header() function. If you have access to the configuration file of your HTTP server, you can also set it there.

HTTP methods

GET and POST requests are the most common HTTP methods, but these are not the only ones you can use in your applications. You may have heard about PUT, PATCH and HEAD, and if you are working on an RESTful API, you should be using them. Let’s see what these less common HTTP methods are and when you should use them.

PUT Request

When speaking from a RESTful viewpoint, a POST request is used to create a new resource, PUT updates it.

PUT replaces the entire resource, so you you are essentially recreating it when using a PUT request. This means you have to send a full representation of the resources, or in other words, all fields of the resource, when using PUT. If the target resource does not exist, it will be created. As an example, PUT http://example.com/customers/1/orders/1 would update all fields of the order and return the status code 200 (OK) if the update was successful.

PATCH Request

Similar to PUT, but instead of updating the whole resource at once, you can only update some specific fields with a PATCH request. You can specify these fields by using the name attribute in the form you are using to make the PATCH request.

PATCH is identical to PUT with the exception that only the fields you send will be updated. All other fields will keep their old values. An example of this could be: PATCH http://example.com/users/1/products/1. In this case, you should have the fields you want to modify in your form.

DELETE Request

DELETE is the easiest method: it simply deletes the entire resource. For example to delete a user with the ID of 1, you can send a DELETE request to http://example.com/users/1. A successful request returns the status code 200. Notice that DELETE always removes the whole

resource, not a single field.

HEAD Request

Like PATCH is related to PUT, HEAD is related to GET. The difference between HEAD and GET is that HEAD only returns the headers of the response. The body, which contains the actual content, is not sent.

OPTIONS Request

This is a method you don't see very often, but is an important one for sure. OPTIONS can be used to query a specific URI to check what methods it supports. If you are unsure what methods have been implemented for some URI, send an OPTIONS request first and then a supported type of request that best suits your needs.

It's sometimes confusing when to use PUT and when to use POST

Let's compare them for better understanding.

PUT:

RFC-2616 clearly mention that PUT method requests for the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource – an update operation will happen, otherwise create operation should happen if Request-URI is a valid resource URI (assuming client is allowed to determine resource identifier).

PUT /questions/{question-id}

POST:

The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. It essentially means that POST request-URI should be of a collection URI.

POST /questions

PUT:

PUT method is idempotent. So if you send retry a request multiple times, that should be equivalent to single request modification.

POST:

POST is NOT idempotent. So if you retry the request N times, you will end up having N resources with N different URIs created on server.

PUT:

Use PUT when you want to modify a singular resource which is already a part of resources collection. PUT replaces the resource in its entirety. Use PATCH if request updates part of the resource.

POST:

Use POST when you want to add a child resource under resources collection.

PUT:

PUT is idempotent, so you can cache the response.

POST:

Responses to this method are not cacheable, unless the response includes appropriate Cache-Control or Expires header fields. However, the 303 (See Other) response can be used to direct the user agent to retrieve a cacheable resource.

PUT:

Generally, in practice, always use PUT for UPDATE operations.

POST:

Always use POST for CREATE operations.

What is the Richardson Maturity Model?

The Richardson Maturity Model is a way to grade your API according to the constraints of REST. The better your API adheres to these constraints, the higher its score is. The Richardson Maturity Model knows 4 levels (0-3), where level 3 designates a truly RESTful API.

Level 0: Swamp of POX

Level 0 uses its implementing protocol (normally HTTP, but it doesn't have to be) like a transport protocol. That is, it tunnels requests and responses through its protocol without using the protocol to indicate application state. It will use only one entry point (URI) and one kind of method (in HTTP, this normally is the POST method). Examples of these are SOAP and XML-RPC.

Level 1: Resources

When your API can distinguish between different resources, it might be level 1. This level uses multiple URIs, where every URI is the entry point to a specific resource. Instead of going through http://example.org/articles, you actually distinguish between http://example.org/article/1 and http://example.org/article/2. Still, this level uses only one single method like POST.

Level 2: HTTP verbs

To be honest, I don't like this level. This is because this level suggests that in order to be truly RESTful, your API MUST use HTTP verbs. It doesn't. REST is completely protocol agnostic, so if you want to use a different protocol, your API can still be RESTful. That said, almost all REST APIs use HTTP.

This level indicates that your API should use the protocol properties in order to deal with scalability and failures. Don't use a single POST method for all, but make use of GET when you are requesting resources, and use the DELETE method when you want to delete a resources. Also, use the response codes of your application protocol. Don't use 200 (OK) code when something went wrong for instance. By doing this for the HTTP application protocol, or any other application protocol you like to use, you have reached level 2.
Level 3: Hypermedia controls

Level 3, the highest level, uses HATEOAS to deal with discovering the possibilities of your API towards the clients.

HATEOAS

Example:

GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK
<?xml version="1.0"?>
<account>
<account_number>12345</account_number>
<balance currency="gbp">100.00</balance>
<link rel="deposit" href="/account/12345/deposit" />
<link rel="withdraw" href="/account/12345/withdraw" />
<link rel="transfer" href="/account/12345/transfer" />
<link rel="close" href="/account/12345/close" />
</account>

Apart from the fact that we have £100 in our account, we can see 4 options: deposit more money, withdraw money, transfer money to another account, or close our account. The "link"-tags allows us to find out the URLs that are needed for the specified actions. Now, let's suppose we didn't have £100 in the bank, but we actually are in the red:

GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK
<?xml version="1.0"?>
<account>
<account_number>12345</account_number>
<balance currency="gbp">-25.00</balance>
<link rel="deposit" href="/account/12345/deposit" />
</account>

Now we are £25 in the red. Do you see that right now we have lost many of our options, and only depositing money is valid? As long as we are in the red, we cannot close our account, nor transfer or withdraw any money from the account. The hypertext is actually telling us what is allowed and what not: HATEOAS

There's no such thing as a REST API

Not everyone agrees with the Richardson Maturity Model - A position often expounded by overzealous REST purists, who miss the point that just because REST doesn't dictate HTTP, doesn't mean that therefore any HTTP API cannot be RESTful... But to be clear - you can have a REST API that doesn't follow the Maturity Model. But a well-crafted API *will* follow it. What you choose to name it comes down to semantics

Also - using cookies doesn't break HATEOAS - to believe so indicates that you don't understand how cookies work.