REST in a Nutshell: A Mini Guide for Python Developers

REST is essentially a set of useful conventions for structuring a web API. By "web API", I mean an API that you interact with over HTTP - making requests to specific URLs, and often getting relevant data back in the response.

There are whole books written about this topic, but I can give you a quick start here. In HTTP, we have different "methods", as they are called. GET and POST are the most common; these are used by web browsers to load a page and submit a form, respectively. In REST, you use these to to indicate different actions. **GET** is generally used to get information about some object or record that already exists.

Crucially, the GET does not modify anything, or at least isn't supposed to. For example, imagine a kind of todo-list

web service. You might do an HTTP GET to the url "/
tasks/" to get a list of current tasks to be done.

So it may return something like this:

```
[
  { "id": 3643, "summary": "Wash car" },
  { "id": 3697, "summary": "Visit gym" }
]
```

This is a list of JSON objects. (A "JSON object" is a data type very similar to a Python dictionary.)

In contrast, **POST** is typically used when you want to create something. So to add a new item to the todo list, you might trigger an HTTP POST to "/tasks/".

That's right, it is the same URL: that is allowed in REST. The different methods GET and POST are like different verbs, and the URL is like a noun.

When you do a POST, normally you will include a body in the request. That means you send along some sequence of bytes - some data defining the object or record you are creating.

What kind of data? These days, it's very common to pass JSON objects. So the API may state that a POST to /tasks/ must include a single object with two fields, "summary" and "description", like this:

```
{
  "summary": "Get milk",
  "description": "A half gallon of organic 2% milk."
}
```

This is a string, encoding a JSON object. The API server then parses it and creates the equivalent Python dictionary.

What happens next? Well, that depends on the API, but generally speaking you will get a response back with some useful information, along two dimensions. First is the *status code*. This is a positive number, something like 200 or 404 or 302. The meaning of each status code is well defined by the HTTP protocol standard; search for "http status codes" and the first hit will probably be the official reference. Anything in the 200s indicates success.

The other thing you get back is the *response body*. When your web browser GETs a web page, the HTML sent back is the response body. For an API, this can response body can be empty, or not - it depends on the API and the end point.

For example, when we POST to /tasks/ to add something to our todo list, we may get back an automatically assigned task ID. This can again be in the form of a JSON object:

```
{ "id": 3792 }
```

Then if we GET /tasks/ again, our list of tasks will include this new one:

```
[
  { "id": 3643, "summary": "Wash car" },
  { "id": 3697, "summary": "Visit gym" },
  { "id": 3792, "summary": "Get milk" }
]
```

There are other methods besides GET and POST. In the HTTP standard, **PUT** is used to modify an existing resource (e.g., change a task's summary). Another method called **DELETE** will... well, delete it. You could use this when a task is done, to remove it from your list.

HTTP Methods for REST APIs

The HTTP verbs comprise a major portion of our “uniform interface” constraint and provide us the action counterpart to the noun-based resource. The primary or most-commonly-used HTTP verbs (or methods, as they are properly called) are POST, GET, PUT, PATCH, and DELETE.

These correspond to create, read, update, and delete (or CRUD) operations, respectively. There are a number of other verbs, too, but are utilized less frequently. Of those less-frequent methods, OPTIONS and HEAD are used more often than others.

POST → Create New Object

The POST verb is most-often utilized to **create** new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information.

Examples:

```
POST http://www.example.com/customers
POST http://www.example.com/customers/12345/orders
```

`GET` → Read Object

The HTTP GET method is used to **read** (or retrieve) a representation of a resource. In the "happy" (or non-error) path, GET returns a representation in XML or JSON and an HTTP response code of 200 (OK). In an error case, it most often returns a 404 (NOT FOUND) or 400 (BAD REQUEST).

According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption—calling it once has the same effect as calling it 10 times, or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request.

Do not expose unsafe operations via GET—it should never modify any resources on the server.

Examples:

```
GET http://www.example.com/customers/12345
GET http://www.example.com/customers/12345/orders
GET http://www.example.com/buckets/sample
```

PUT → Update/Replace Object

PUT is most-often utilized for **update** capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource.

However, PUT can also be used to create a resource in the case where the resource ID is chosen by the client instead of by the server. In other words, if the PUT is to a URI that contains the value of a non-existent resource ID. Again, the request body contains a resource representation. Many feel this is convoluted and confusing. Consequently, this method of creation should be used sparingly, if at all.

Alternatively, use POST to create new resources and provide the client-defined ID in the body representation—presumably to a URI that doesn't include the ID of the resource (see POST below).

On successful update, return 200 (or 204 if not returning any content in the body) from a PUT. If using PUT for create, return HTTP status 201 on successful creation. A body in the response is optional—providing one consumes more bandwidth. It is not necessary to return a link via a Location header in the creation case since the client already set the resource ID.

PUT is not a safe operation, in that it modifies (or creates) state on the server, but it is idempotent. In other words, if you create or update a resource using PUT and then make that same call again, the resource is still there and still has the same state as it did with the first call.

If, for instance, calling PUT on a resource increments a counter within the resource, the call is no longer idempotent. Sometimes that happens and it may be enough to document that the call is not idempotent. However, it's recommended to keep PUT requests idempotent. It is strongly recommended to use POST for non-idempotent requests.

Examples:

```
PUT http://www.example.com/customers/125
PUT http://www.example.com/customers/125/orders/98
PUT http://www.example.com/buckets/secret_stuff
```

`PATCH` → Updated/Modify Object

PATCH is used for **modify** capabilities. The PATCH request only needs to contain the changes to the resource, not the complete resource.

This resembles PUT, but the body contains a set of instructions describing how a resource currently residing on the server should be modified to produce a new version. This means that the PATCH body should not just be a modified part of the resource, but in some kind of patch language like JSON Patch or XML Patch.

PATCH is neither safe nor idempotent. However, a PATCH request can be issued in such a way as to be idempotent, which also helps prevent bad outcomes from collisions between two PATCH requests on the same resource in a similar time frame. Collisions from multiple PATCH requests may be more dangerous than PUT collisions because some patch formats need to operate from a known base-point or else they will corrupt the resource. Clients using this kind of patch application should use a conditional request such that the request will fail if the resource has been updated since the client last accessed the resource. For example, the client can use a strong ETag in an If-Match header on the PATCH request.

Examples:

```
PATCH http://www.example.com/customers/125
PATCH http://www.example.com/customers/125/orders/98
PATCH http://www.example.com/buckets/secret_stuff
```

DELETE → Delete Object

DELETE is pretty easy to understand. It is used to **delete** a resource identified by a URI.

On successful deletion, return HTTP status 200 (OK) along with a response body, perhaps the representation of the deleted item (often demands too much bandwidth), or a wrapped response (see Return Values below). Either that or return HTTP status 204 (NO CONTENT) with no response body. In other words, a 204 status with no body, or the JSEND-style response and HTTP status 200 are the recommended responses.

HTTP-spec-wise, DELETE operations are idempotent. If you DELETE a resource, it's removed. Repeatedly calling

DELETE on that resource ends up the same: the resource is gone. If calling DELETE say, decrements a counter (within the resource), the DELETE call is no longer idempotent. As mentioned previously, usage statistics and measurements may be updated while still considering the service idempotent as long as no resource data is changed. Using POST for non-idempotent resource requests is recommended.

There is a caveat about DELETE idempotence, however. Calling DELETE on a resource a second time will often return a 404 (NOT FOUND) since it was already removed and therefore is no longer findable. This, by some opinions, makes DELETE operations no longer idempotent, however, the end-state of the resource is the same. Returning a 404 is acceptable and communicates accurately the status of the call.

Examples:

```
DELETE http://www.example.com/customers/12345
DELETE http://www.example.com/customers/12345/orders
DELETE http://www.example.com/bucket/sample
```