Requests

Fishing in the River with Pine Tree, mid-20th Century
산정 서세옥 Suh Se-ok (1929-2020)

Learning Objectives

- Understand what a server is and how APIs allow us to access resources on one
- Be able to describe the CRUD paradigm in our own words
- Be able to describe what a header is
- Make a request to a server and process the result

A **request** is a process wherein one computer (the **requestor**) asks another computer (the **host**) for something, usually over the internet. If you've ever used a website of any kind, then you have (unknowingly) done this. You can think of a request as akin to going fishing, in that when we fish, we send something into the water (the server), "requesting" a fish back (the resource). If we've done everything right, we'll get a fish. But sometimes we don't get any fish—maybe there were not any in the water to begin with. Or, sometimes our line gets caught on a log underwater. If we're fishing in a river, the current may actually re-direct the line to an area with more fish. All of these analogies are representative of how a web server works.

What is a server?

A server is a computer. More precisely, a server is a computer that has features in place allowing it to interact with computers outside of itself and return a **response** when asked for a **resource**. A resource can be a variety of things, but is usually thought of as some file, script, or other operation on or performed by the server. One of these features is an internal mechanism to classify and report back to the requestor the status of the request so given. For example, if we were to send a request to a web server asking for a protein sequence, but the protein name we gave was not in the protein database the server has, then the server would send back a **status code 404—Resource not found** (or just "404" for short). This is a standardized code[1] *which describes the result of the server attempting to process our request.* A well-built server will properly inform you of the result of your request, but ultimately status codes are the perogative of the developer who has set up the scripts on the server to handle incoming requests. In other words, the quality of the response is only as good as the programmer who implemented it.

Application Programming Interfaces (APIs)

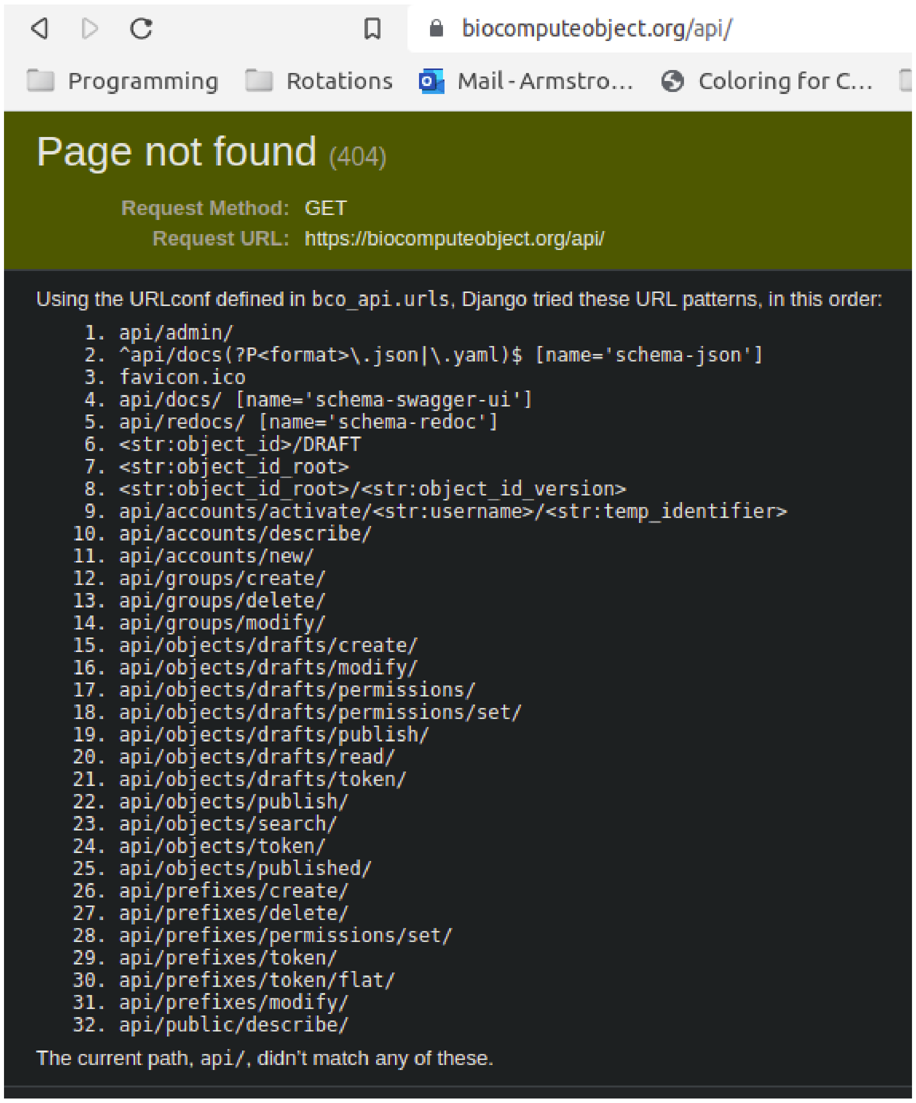
In order to standardize the way requests are handled and to allow requestors to understand when and what type of request to make, most professionally built servers will have an **application programming interface (API).** The name describes exactly what it is—an *interface* to allow *programs (programming)* to access an *application* on the server. As an example API, consider the one available at the website biocomputeobject.org, a portal which allows users to store information describing bioinformatics workflows in a standardized way.[2]

Let's take a look at an attempt to access the url https://biocomputeobject.org/api in Figure 1.
We can glean several key pieces of information from this request. First, we see that the server returned status code 404 for our request, indicating that the URL we tried to access was not found. But how can the server return anything if the URL was not found? The answer is that we must not confuse the server's behavior as a processor of requests with the server's behavior as a repository of resources—the server is perfectly fine telling us that what we asked for was not there. The second piece of information which is relevant to our request is the "Request Method" line near the top. We will discuss request methods more in depth below, but for now we'll notice that the method used was GET, as is typical for a request made in the URL bar in a web browser. Lastly, we can see that the server (API) has several URLs available to us, grouped by related functionality. For example, we can see that requests of the form "api/accounts/..." form a macro grouping relating to API account operations, as do requests of the form "api/objects/...", which relate to API object operations. This server is therefore well-structured to explain to the requestor the available request URLs and their general purposes. When this is the case, we say that the server is employing **semantic URLs.**[3] If you are ever in a position to design an API of any sort, then you should most certainly use semantic URLs.

[1] `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`

[2] See `https://docs.biocomputeobject.org/` for a more complete description of BioCompute Objects (BCOs).

[3] For a concise explanation of this concept, see `https://unihost.com/blog/semantic-urls-and-why-you-should-use-them/`



```
Page not found (404)
        Request Method:  GET
           Request URL:  https://biocomputeobject.org/api/

Using the URLconf defined in bco_api.urls, Django tried these URL patterns, in this order:
     1. api/admin/
     2. ^api/docs(?P<format>\.json|\.yaml)$ [name='schema-json']
     3. favicon.ico
     4. api/docs/ [name='schema-swagger-ui']
     5. api/redocs/ [name='schema-redoc']
     6. <str:object_id>/DRAFT
     7. <str:object_id_root>
     8. <str:object_id_root>/<str:object_id_version>
     9. api/accounts/activate/<str:username>/<str:temp_identifier>
    10. api/accounts/describe/
    11. api/accounts/new/
    12. api/groups/create/
    13. api/groups/delete/
    14. api/groups/modify/
    15. api/objects/drafts/create/
    16. api/objects/drafts/modify/
    17. api/objects/drafts/permissions/
    18. api/objects/drafts/permissions/set/
    19. api/objects/drafts/publish/
    20. api/objects/drafts/read/
    21. api/objects/drafts/token/
    22. api/objects/publish/
    23. api/objects/search/
    24. api/objects/token/
    25. api/objects/published/
    26. api/prefixes/create/
    27. api/prefixes/delete/
    28. api/prefixes/permissions/set/
    29. api/prefixes/token/
    30. api/prefixes/token/flat/
    31. api/prefixes/modify/
    32. api/public/describe/
The current path, api/, didn't match any of these.
```

Figure 1: The response from attempting to access biocomputeobject.org.

Request Methods, Headers, and Bodies

As mentioned above, our request method was "GET". As of this writing, there are 9 request methods available to you[4] (common ones bolded): CONNECT, **DELETE**, **GET**, HEAD, **OPTIONS**, **PATCH**, **POST**, PUT, TRACE. A request and the method used to send it is actually a very complex topic and chain of events, so we won't focus on those here. The main thing to know is that each of the request methods is supposed to be used for different operations on the server. For our purposes, we will summarize the intended use of each method in Table 1.

4 of the methods above, POST, GET, PATCH, and DELETE, are often described as the **Create, Read, Update, and Delete (CRUD)** paradigm. These are generally the most common request methods. Ideally, a server is set up to tie URLs as they are understood semantically with the request

[4]See `https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_methods` for a more complete description of these.

Request Method	**Purpose**
DELETE	Delete a resource on the server
GET	Get a resource from the server
OPTIONS	Describe what capabilities the server has with respect to a resource
PATCH	Update an existing resource on the server
POST	Create a new resource on the server

Table 1: Common request methods and their purposes.

method being used (i.e., a URL like /api/objects/create/ uses POST whereas a URL like /api/objects/update/ uses PATCH). However, other than a few major exceptions, this linkage between request method and url description is not enforced by most servers. Again, the request method and the "semanticness" of the URL requested are only of a quality as good as the programmer who created them. Put succinctly, **the request method matters and the types of requests available to you are dictated by the method you use to make them.** Another way of saying this is that **a URL will always have a list of accepted request methods associated with it.** If you are trying to access /api/objects/create/ using GET, you'll get an error, because you are not using the right request method for this URL.

In addition to the request method, we can also supply **headers**, which sit at the "head" of our request to the server.[5] Headers are analagous to metadata, which means that they provide information to the server which helps it understand what it is being asked to do. A lot of the time, your browser will attach the headers for you automatically. But when we're making requests in Python, *they must be provided manually.* Headers can be **request** or **response** headers. Some of the most commonly used request headers in Python are **Access-Control-Request-Method**, **Authorization**, **Content-Encoding**, and **Content-Type**. These generally have to do with providing security credentials or specifying the type of information we are trying to send to the server. Some of the most common response headers are **Access-Control-Allow-*** (several of this form), **Allow**, **Content-Encoding**, **Content-Type**, **Server**, and **Status**. These also have to do with the type of information the server is returning, security, and the status of the response.

Lastly, when we send a request, we can append a **request body**. One specific thing to know here is that appending a request body requires that we use the POST request method. This makes sense because POST is used to create a new resource, and we must, of course, provide the resource that we wish the server to create. Other request methods, such as GET, do not allow a request body to be sent. One major issue with poorly designed servers is mixing up the resource management of what should be GET- and POST-facing requests. For example, if a server had the URL /api/objects/create/ and we had to use a GET request to create a new object (perhaps /api/objects/create/objectName='some_name'&objectCreator='some_creator'...) then the server would be mixing up resource creation with resource retrieval, which should be conducted with POST and GET, respectively. POST is also a bit more secure at a browser level because the URL bar usually retains GET requests (with parameters) but POST requests have to be found by digging into browser history more thoroughly.

[5]See `https://en.wikipedia.org/wiki/List_of_HTTP_header_fields`

In-class coding challenge

The design file "requests.py" contains a few imports and code that we'll need to complete the goals below. We'll be making calls to the Gene Ontology (GO) and Panther Databases. See `http://api.geneontology.org/api` and `http://pantherdb.org/services/openAPISpec.jsp` to help us make the calls. We wont need any request headers or bodies to complete these goals.

- **Goal 1:** Add a required command line argument "-i"/"--id" (identity) which takes a GO id.
- **Goal 2:** Create a function "strip_terms" using the GO file "goslim_mouse.obo", stripping from each [Term] the id, name, and def and store all of these in a dictionary. **Hint:** we don't need "alt_id"s and we don't need double quotes from the "def" lines.
- **Goal 3:** Come up with some way to show the user that the command line argument "identity" was or was not found in the file "goslim_mouse.obo".
- **Goal 4:** If the command line argument "identity" WAS found in the file, then make an API call to GO /api/bioentity/function/{id}, THEN take a random Panther Family value from the response and make a second call to PantherDB /services/oai/pantherdb/familymsa. Write the response to a file. Do you notice something wrong with the way Panther has set up this API call?

NOTE: Your solution should have all functions accessible in a class.

References

Fishing in the River with Pine Tree. Kang Collection. (n.d.). Retrieved March 1, 2022, from http://www.kangcollection.com/paintings/suh-se-ok-sanjeong-seoseog-1929