Building Applications with Python

Module 2

Summary:

Version: 1.0

Contents

Chapter I

	Exercise 01
Hyper Text Transfer Protocol (HTTP)	
Turn-in directory: *ex*01/	
Files to turn in: `do_HTTP.py`	
Allowed functions: `Standard Library, httpx`	

In the previous module we worked with `curl`, but now we will use Python to make HTTP requests.

Create a file called `do_HTTP.py` and do the following:

1. Create two functions: `do_GET` and `do_POST`.

2. The `do_GET` function should make an HTTP GET request (to this address) [https://jsonplaceholde
   and return the status code of the return and the result in json() format.

3. The `do_POST` function should make an HTTP POST request (to this address) [https://jsonplaceholder.typicode.com/posts], sending a JSON with the data below, also returning the status code and result in json() format:

```
{
   "title": "foo",
   "body": "bar",
   "userId": 1
}
```

4. Your `main()` function should print the results of `do_GET()` and `do_POST()`, as below.

Example of expected behavior:

```
$ python3 do_HTTP.py
200 {'userId': 1, 'id': 1, 'title': 'sunt ...', 'body': 'quia...'}
201 {'title': 'foo', 'body': 'bar', 'userId': 1, 'id': 101}
```

It is important to understand in this exercise what the difference is between GET and POST methods. Don't worry about other methods for now.

Chapter II

	Exercise 02
Handling JSON responses	
Turn-in directory: *ex*02/	
Files to turn in: `format_json.py`	
Allowed functions: `Standard Library, httpx`	

This following endpoint is great for retrieving student projects: (ENDPOINT) [https://assets.breathe

Make a request to the endpoint using the `httpx` library in Python, and after that, return the project name and the URL of the first image.

Expected output:

```
$ python format_json.py
$ Project name: {}, URL of the first image: {}
```

Chapter III

	Exercise 03
Responses	
Turn-in directory: *ex*03/	
Files to turn in: `response.py`	
Allowed functions: `No restrictions, httpx, json`	

Not every endpoint will always return what you need. You need to ensure that what is being returned is handled correctly, and deal with it appropriately, or at least give the user feedback on what happened.

There are several HTTP response codes that represent what happened. You can see more (by clicking here) [https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status].

1. Create a **program** that receives a URL as the only parameter and makes a request to it.

2. Your program will print the code and the message that is linked to the http status.

3. In case of an unexpected error (such as a connection error), print `Error:`, followed by the name of the error.

Example of expected output:

```
$ python3 response.py https://www.google.com
200 OK
$ python3 response.py https://invalidaddress
Error: ConnectError
```

Check the documentation of httpx to understand how to handle responses.

When we talk here about "unexpected errors", we are not referring to HTTP errors (such as 404 or 500), which are still valid responses from the server, but rather execution failures that prevent the request from being completed, such as DNS problems, timeouts, invalid address, etc.

Chapter IV

Information

Now that you know how to make requests and retrieve data, let's focus on making your programs more reliable. Two practices are important from now on:

1. Error handling: Your requests and data handling logic should always anticipate and handle potential failures. This means checking HTTP status codes, handling unexpected or empty responses, and, where applicable, providing clear feedback to the user.

2. Unit tests: To ensure your code works as expected — especially when dealing with failure scenarios, whether due to execution errors (e.g., exceptions in Python) or HTTP error responses (e.g., 4xx or 5xx codes).

From now on, each function you create must be accompanied by at least one unit test.

Remember, the focus is on meaningful tests: check the specific behavior of your code (how it reacts to a 404 error, invalid data, etc.), and not the basic functionalities of the Python language that you already trust.

-

Chapter V

	Exercise 04
Emoji Searcher	
Turn-in directory: *ex*04/	
Files to turn in: `*.py, static/form.html`	
Allowed functions: `Standard library, FastAPI, pytest, uvicorn`	

In this exercise you will modify a simple application written in FastAPI.

1. Before starting, make sure that the `charindex.py` and `mojifinder.py` modules are passing tests and have correct type annotations.

2. Run the example with this command line:

```
uvicorn mojifinder:app --reload --reload-include '*.html'
```

```
The '-reload' option should only be used during development.
With it, the 'uvicorn' server automatically reloads the application
when one of its files is modified.
```

```
It's important to start by making sure all tests pass, because if
something breaks later, it's easier to locate the problem.
```

1. Open your browser at the URL http://127.0.0.1:8000/ and search for some emojis.

Interesting searches to try: cat, cat face, forty two, egyptian, hexagram.

1. Now begins the real work. Your mission is to add another route to the application, to perform reverse searches: given one or more characters, display a list with the names of the characters, using the output format of the "/search" route. The new route will use the path "/names". For example, if the query is the word "action", the HTML will display:

```
4 characters found
U+0061 a   LATIN SMALL LETTER A
U+00E7 c   LATIN SMALL LETTER C
U+00E3 t   LATIN SMALL LETTER T
U+00E3 i   LATIN SMALL LETTER I
U+00E3 o   LATIN SMALL LETTER O
U+00E3 n   LATIN SMALL LETTER N
```

`Start with the test. Write a very simple test for the "/names" route in the test file and then implement the route.`

1. With both routes working in the backend, now adjust the frontend. You will need to make an adjustment to the JavaScript of the search form with the following logic:

   - if the query starts with the character '?', the "/names" route should be used. Otherwise, use the "/search" route.

   -

Chapter VI

	Exercise 05
Fast API	
Turn-in directory: *ex*05/	
Files to turn in: `api.py`	
Allowed functions: `No restrictions, fastapi, uvicorn`	

Here you will create your first API using FastAPI. You will install FastAPI and create some routes (or, also known as _endpoints_). The response codes should be:

- 200
- 201
- 404
- 500

You will accept both POST and GET.

In your route, you will have logic implemented to return one of these codes.

- If the request is a GET at the root without data in the body, return code 200.
- If the request is a POST with data, return code 201.
- If the request is a POST without data, return code 400.
- If the request is a GET with a path beyond the root, return code 404.
- If the request is a GET with the path `/broken`, it should access a function with this name that raises an exception 50% of the time, and the API should return code 500 or 200, according to the behavior of the function.

`Access the FastAPI documentation and go to the "Quickstart" to get started.`

You can run your api with:

```
uvicorn api:app
```

Chapter VII

	Exercise 06
Banking API	
Turn-in directory: *ex*06/	
Files to turn in: `api.py, test_api.py`	
Allowed functions: `No restrictions, fastapi, pydantic, uvicorn`	

Now that you know how to create an API with FastAPI, let's explore some other important components in creating an API. Normally, you don't want the data sent via POST to be anything, you need to validate that information first.

You will create a new API and, this time, this API will have a POST route that will receive a JSON. This JSON is from a user trying to create a checking account.

Some information is mandatory, and you can only allow creation if all are filled (fields: `name`, `age`, `email`, and `balance`), otherwise, return the correct corresponding HTTP code.

`See about code 422.`

Also, for the validation of the input JSON, you will use pydantic

For pydantic to work, you must create a model for it. This model is a class where you determine the input type of the data.

Return the appropriate code in case of errors.

Chapter VIII

Bonus

	Exercise 07
Docker	
Turn-in directory: *ex*07/	
Files to turn in: `api.py`, `Dockerfile`, `requirements.txt`	
Allowed functions: `No restrictions`, `fastapi`, `pydantic`, `uvicorn`	

Containerize your application. It will be executed with the commands `docker build` and `docker run`.

Here you will need to create a Dockerfile. Since your environment has requirements that must be installed (such as ‘fastapi‘, for example), also create a ‘requirements.txt‘ file and add a command to your Dockerfile to install these prerequisites automatically.
You can create a requirements file using the command ‘pip freeze > requirements.txt‘.

You are already familiar with the Docker references. Research further the ‘RUN‘, ‘COPY‘, and ‘WORKDIR‘ commands.