API 101: MAKING SOME CALLS

Lab 06 - CS 411 @ Boston University

API Terminology

API (Application Programming Interface)

- A set of definitions and protocols for building and integrating application software

REST API (or RESTful API)

- An API that conforms to the REST architectural style.

REST

- Stands for “Representational State Transfer”
- A client-server architecture made up of clients, servers, and resources, with requests managed through **HTTP**.

Request Terminology

GET

- Used to read/retrieve data from a web server. HTTP status code of 200 (OK) on success.

POST

- Used to send data (file, form data, etc.) to the server. HTTP status code of 201 on success.

PUT

- Used to modify the data on the server. Replaces entire content at a particular location with data that is passed in the body payload.

PATCH

- Similar to PUT request. It modifies part of the data, does NOT replace entire content.

DELETE

- Used to delete the data on the server at a specified location.

BOSTON
UNIVERSITY

Calling RESTful APIs

CS411 Spring 23
ChatGPT & Peter B. Golbus

Calling RESTful APIs in Python

```
import requests


def request(env):
    headers = {
        'X-RapidAPI-Key': env["api_key"],
        'X-RapidAPI-Host': env["api_host"]
    }
    with requests.get(env["api_url"], headers=headers) as r:
        word = r.json()[0]["word"]
        print(word)
        return word
```

Calling RESTful APIs in Python

```
import requests
```



```
def get_coins():
    """Curls random.org to get the coin flips

    Returns:
        An array of coin flips
    """
    url = 'https://www.random.org/integers/?format=plain&num=18&min=2&max=3&col=18&base=10'
    r = requests.get(url)
    text = r.text
    return [int(x) for x in text.strip().split('\t')]
```

Calling RESTful APIs in Python

```
import requests
Boston University Slideshow Title Goes Here


def chat(message):
    url = "https://chatgpt-api.shn.hk/v1/"
    headers = {
        "Content-Type": "application/json"
    }
    data = {
        "model": "gpt-3.5-turbo",
        "messages": [{"role": "user", "content": message}]
    }
    r = requests.post(url, headers=headers, data=data)
    print(r.json())


if __name__ == "__main__":
    chat("I hope you have a wonderful day")
```

sync vs async

- In Python, we get to make synchronous blocking calls. The script will wait until it gets a result. If it takes 4 seconds, it takes 4 seconds
- In JavaScript, we have to make asynchronous non-blocking calls. The webpage has to render and can't wait for a response. If it takes 4 seconds, you have to be displaying something in the meantime

Promises

- **Promises** in JavaScript are a way to **handle asynchronous operations** and provide a clean and organized way to write asynchronous code.
- A **Promise represents the eventual completion (or failure)** of an asynchronous operation and allows you to **attach callbacks to be executed when the operation completes**.
- The Promise object has **three states: pending, fulfilled, and rejected.**
  - **Pending**: the asynchronous operation is **still running**
  - **Fulfilled**: the asynchronous operation has **completed successfully**. The Promise has a **resolved value** that **can be accessed by the callbacks attached to it**.
  - Rejected: the asynchronous operation has **failed**. The Promise has a **reason for the rejection** that can also be **accessed by the attached callbacks**.

Callbacks

- A **callback** is a **function** that is **passed as an argument to another function** and is **executed by that function once a specific event or condition occurs**
- The primary purpose of a callback is to allow a function to call another function and then continue execution, rather than waiting for the called function to complete its task
- This makes callbacks **an essential part of asynchronous programming, where they are used to handle events and responses that may not be available immediately**
- Callbacks can be defined inline or as separate functions and are widely used in web development, especially in JavaScript.

When it’s done, what do you do then()

- The **Promise.then()** method is used to **attach success and error callbacks to a Promise.**
- The then() method takes **two arguments: a success callback and an error callback.**
- The success callback is executed when the Promise is fulfilled with a resolved value. The resolved value is passed as an argument to the success callback.
- The error callback is executed when the Promise is rejected with a reason. The reason for rejection is passed as an argument to the error callback.

When it’s done, what do you do then()

- **The then() method returns a new Promise** that can be used to chain additional then() methods or catch() methods.
- If the success or error callback returns a value, the new Promise returned by then() will be fulfilled with that value.
- If the success or error callback throws an error or returns a rejected Promise, the new Promise returned by then() will be rejected with that error or reason.
- The then() method can be **called multiple times on the same Promise, allowing you to chain multiple success and error callbacks together**.

But what is the return “value?”

- That’s the wrong question
- then() executes a callback when a promise is fulfilled
- It returns a new promise and if you want to, you can attach another callback to be executed when that new promise is fulfilled

Async

Run this function in the background

When that promise resolves create a new one

```
async function fetchThenLog() {
    fetch('https://jsonplaceholder.typicode.com/posts/1')
        .then(response => response.json())
        .then(data => console.log(data))
        .catch(error => console.error(error));
}
```

returns a promise

When this problem resolves, I am finished logging

PG write a minimal working example of a javascript async function that uses fetch and then

Async/Await

Run this function in the background

```
async function fetchExample() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts/1');
    const data = await response.json();
    console.log(data);
    return data;
  } catch (error) {
    console.error(error);
  }
}

fetchExample().then(data => {
  console.log(`Returned data from fetchExample: ${data}`);
}).catch(error => {
  console.error(`Error occurred in fetchExample: ${error}`);
});
```

execution continues but **THIS** function blocks

when that resolves, execution continues but **THIS** function blocks again

async functions return promises

PG

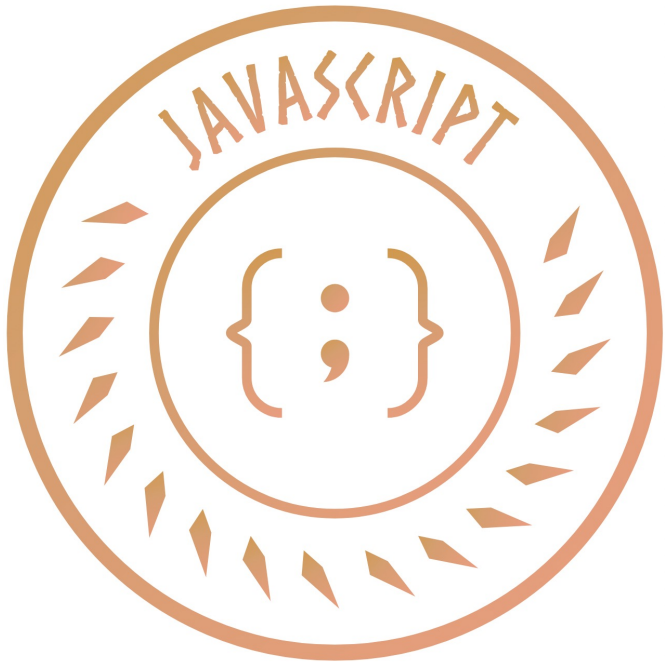
write a minimal working example of a javascript async function that uses fetch, then and await

LETS MAKE SOME…

API CALLS

Tasks for Today!

- Solidify your APIs
  - Basically, choose **at least two** you want to move forward with and make sure they work
  - You can do this with Postman at first
- Start writing some code!
  - Once you know your APIs work, with the language of your choice try to make a call to one (or both) of them
  - This should be a great starting point for your project
- As per usual I'll be coming around if you need help



MDN web docs
moz://a

DevDocs

DEVELOPMENT RESOURCES

https://www.theodinproject.com/
https://developer.mozilla.org/en-US/
https://devdocs.io/

```
git add lab06.txt
git commit -m “complete”
git push origin master
```

Lab 06 - CS 411 @ Boston University

(dcmag@bu.edu)