Creative Project 3 Written

The written component of CP3 (cp3-written.pdf) guides your review of the material on asynchronous programming and APIs and has you connect the concepts to your own experiences in the real world.

This is a required part of CP3, and you supplement it with any reflection/diagrams similar to CP1/CP2 if you'd like for extra credit. Make a copy of this document and submit your answers to CodePost as cp3-written.pdf.

Students are allowed to collaborate on this written component (via Discord or in-person OH), as long as the answers are written individually.

As written answers, these are more open-ended and will be graded on demonstrated understanding of the material, going further into the concepts introduced in class, and connecting to your own experiences as users, developers, and members of society. You should aim to spend 45-60 minutes if you've been caught up with the material and are welcome to ask on Discord if you have questions.

Asynchronous Code, Promises, and Fetch

Over the term, we've seen various applications of asynchronous programming, starting with the event listener. We introduced "Promises" in Module 3 with fetch; fetch was added to browser specifications in 2015 (you can read a brief overview here if interested).

a. Provide the code for one of your fetch calls you might make to an API you chose for CP3 (whether you use then/catch or async/await syntax is up to you, but should start from the fetch statement and end with an error-handling function).

```
const API_URL =
"https://restcountries.com/v3.1/all/?fields=name,region,subregion,cap
ital,currencies,languages,flags";

        fetch(API_URL)

            .then(checkStatus)

            .then(response => response.json())

            .then(data => {

                allCountries = data;

                displayGroupedCountries("region");
```

```
        })
        .catch(handleError);
```

b. In a bulleted list or an annotated diagram, summarize the process of a fetch call, using your example. We are looking for you to demonstrate an understanding of fetch and the HTTP Request/Response protocol with a focus on **datatypes** at each step of execution (including Promises, their resolved values or errors, and callback functions) and the **role of the client vs. server** in the process

- Fetch request -> returns a Promise that resolves to a Response object.
    - Client: Makes a GET request to the API server
    - Server: gets request, processes it and sends back an HTTP response
- .then(checkStatus) ->checks if the response has a successful status (200–299).
    - If yes: Passes the response to the next step.
    - If no: Throws an error to be caught in .catch().
    - Datatype: Still a Response object.
- .then(response => response.json()) -> Parses the body of the HTTP response as JSON.
    - Returns: A Promise that resolves to a JavaScript object or array.
    - Datatype: Parsed JSON (in this case, an array of country objects).
- .then(data => { allCountries = data; displayGroupedCountries("region"); }) -> Stores the parsed data and uses it to update the UI.
    - Client: Updates internal state (allCountries) and calls a rendering function.
    - Datatype: data is a JavaScript array.
- .catch(handleError) -> Handles any error that occurred during fetch, status check, or JSON parsing.
    - Datatype: Catches an Error object.
    - Client: Logs error or shows an error message to the user.

c. In 1-2 sentences, what is the purpose of checkStatus?

i. The purpose of checkStatus is to verify that the HTTP response has a successful status code (typically in the 200–299 range). If the status indicates an error (like 404 or 500), it throws an error to be handled in the .catch() block.

API Documentation

Web APIs are an extremely powerful way to collect and share data on the web, with use cases in business, social networking, government, healthcare, education, research, etc. And unfortunately, in order of decreasing quality (consider the need for sharing data efficiently for research at institutions like Caltech!). It's not just design and implementation that makes an API usable.

API documentation is one of the most important things when it comes to designing and using APIs. You've been getting practice learning APIs through their documentation, and El talked about how some APIs are better than others. In the Final Project, you'll also be expected to write quality API documentation (at the start! You should have a draft going...). Others should clearly know how and when to use your APIs without knowing the implementation details.

In a recent lecture, students were to identify 3 APIs and reflect on their documentation. Briefly identify:

3 or more things that are particularly important for clients of API documentation

1. Clear and complete endpoint descriptions, what does each endpoint do, what is the required URL, etc.
2. Example requests and responses help user understand how to use the API for their needs
3. Error handling and descriptive status codes to allow for easier debugging
4. If applicable, then authentication requirement procedures like API keys or tokens should be clearly documented.

2 or more things that make an API difficult or unreliable to use

1. Lack of consistency with naming, structure, or parameter usage
2. Incomplete or outdated documentation that doesn't detail all of the information about the current iteration of the API.

Error-Handling and HTTP Status Codes

a. What are two reasons the API would appropriately return a 400-level error to a fetch request? A 500-level error? You don't need to use the actual behavior of the API, but you can identify reasonable causes if the API was designed to handle different cases with the appropriate codes.

A 400-level error is a client side error so something like missing or invalid parameters in the URL would cause a 400-level error. A 500-level error is server side so an unexpected internal server bug or failure are some ways that a 500-level error would occur.

b. How would you modify your fetch-related code to display a different message to a user depending on the cause of the error? Hint: there are a few reasonable solutions here.

After checkStatus throws an error, we can make an error handler catch the error as so:

```
function handleError(error) {
    const messageElement = document.getElementById("error-message");
    messageElement.textContent = error.message;
    messageElement.style.display = "block";
}
```

APIs in the Real World and the Network Debugger

In this exercise, we want to make sure you 1) are comfortable using the debugger and Network tab for CP3 onward, 2) develop a growing awareness of network requests and the amount of data collected from and sent to a browser, and 3) build strategies for refining your own API endpoint and response schema design.

VIsit a webpage of your choosing. Identify 1 (or more) features that likely use an HTTP fetch request (e.g. a search button to search for all local coffee products based on checkboxes for options); predict the parameters/schema, and use the Network tab to compare the results with what you predicted. Webpage:

a. Feature(s) chosen:

Search feature of https://openweathermap.org. Input “pasadena, ca, us” verbatim.

b. Prediction of endpoint design/possible query/path parameters ("Request" in Network tab):

https://openweathermap.org/?fields=pasadena,ca,us

c. Prediction of information returned by fetch request ("Response" in Network tab)

```
{
    “city” {
        “coords”: {
            “latitude” : XX;,
            “longitude” : YY
        },
        “weather” : {
```

“description” : “temperate”,

“temp” : …

…

}

d. Actual endpoint:

https://api.openweathermap.org/data/2.5/find?q=pasadena,%20ca,%20us&appid=5796abbde9106b7da4febfae8c44c232&units=metric

e. Actual schema/response result:


```
{
   "message": "accurate",
   "cod": "200",
   "count": 1,
   "list": [
      {
         "id": 5381396,
         "name": "Pasadena",
         "coord": {
            "lat": 34.1478,
            "lon": -118.1445
         },
         "main": {
            "temp": 15.79,
            "feels_like": 15.51,
            "temp_min": 14.65,
```

```
        "temp_max": 16.69,
        "pressure": 1015,
        "humidity": 80,
        "sea_level": 1015,
        "grnd_level": 958
    },
    "dt": 1748412578,
    "wind": {
        "speed": 0.45,
        "deg": 247
    },
    "sys": {
        "country": "US"
    },
    "rain": null,
    "snow": null,
    "clouds": {
        "all": 1
    },
    "weather": [
        {
            "id": 800,
            "main": "Clear",
```

```
            "description": "clear sky",
            "icon": "01n"
        }
      ]
    }
  ]
}
```


Applying APIs to Privacy, Security, Ethics, and Accessibility

This final exercise gets students connecting the material to the real-world; part of the learning outcomes of CS 132 are to develop mindsets in these areas, using your own experience as users of the web to become better web developers. We are giving you some freedom to explore a topic of your choice; choose 1 (of course, you may choose more) of the following to answer. For full credit, we're looking for approximately 5-10 minutes spent on this exercise.

Option 1: Choose 2 of the following topics to write at least 2 sentences of an example where they could apply to APIs. This is open-ended, but we want you to think about these, drawing on your own experience as a user, member of society, developer, etc. Try to refer to a specific anecdote from your own experience, a relevant news article, or example related to one of your Caltech courses/events.

Accessibility

One type of API I've encountered through cursory browsing are APIs specifically geared towards ADA compliance and ensuring that sites are compliant with the accessibility laws in the US. A lot of them provide automated accessibility testing which I feel would be useful for developers who may not themselves require those accessibility options.

Ethics

These days a particular example of ethics and APIs is the ethics of using AI APIs for creative purposes. A common example is prompting AI art, which is commonly known to be created using reference material from existing artists without citation or credit being given to the original artists.