Project 1: Currency Converter

Create a *real-time* currency converter similar to the one you would see on Google. It should update the resulting currency in real time, do not rely on submitting a form to call the API.

Requirements:

- Make the application in HTML, CSS, and JavaScript. **Do not ChatGPT any code**, instead google errors or look at documentation.
    - This includes styling! You ain't gonna learn shit if you let it generate everything for you
    - Follow proper code etiquette
        - don't name variables x and y
        - descriptive comments but not too many
        - just imagine you're sending this to a company, imagine how you would want them to see your code (organized, follows industry standards, etc)
- Have a dropdown of currencies to choose from (both for the starting currency and ending currency)

Extra features (I really recommended to add these):

- Include the symbols of the currency
- Include the flags of the country where the currency is from.
    - Hint: the API I used was this (https://flagpedia.net/download/api)
    - I'll let you figure out how you can use your first free API, get certain info from it, and use that info to make a call using this API.

Process

- Use of a free API – think about what type of API you want to use, what information you need, etc.
    - Calling an API every time the input changes is really dumb, it is expensive and you'll run out of API calls extremely quickly. Find a way around this.
    - Go read through the documentation of the API you want to use, see the possible endpoints and what information they will give you.
- Get inspiration for what you want your project to look like.
    - Steal a design from Google or my favourite https://dribbble.com/
    - Use Figma if you want to quickly create a prototype, so you know what fonts, colours, and layouts you want to use.
- Create the HTML, CSS, JS files.
    - Start with fleshing out your HTML, CSS files first.
- Learn how to use APIs in Javascript
    - Learn what Promises are
        - .then() and .catch() methods are crucial here
        - Can also use async and await (I prefer .then() and .catch() tho)
    - Learn how to use fetch() in Javascript (this is the actual API call)

        - Using .then() with a callback or await to get the JSON response
            - Learn what callbacks are!

To be honest, you don't need to understand anything I mentioned above except for the fetch function and .then(). If you really want to understand Javascript, here are my favourite videos (plus my own notes/explanations on these topics as a starting point)

Functions

- Typical way – button.onclick function() { … }
- ES6 new concise way – button.onclick = () => { … }
    - This is useful for callbacks
- **Callbacks** – functions being passed as arguments for other functions
    - Main purpose is for asynchronous calls where the callback will be executed once the asynchronous call is done.
    - E.g., map(( ) => { … }) can define it concisely within the map function

Promises - https://www.youtube.com/watch?v=li7FzDHYZpc

- Manages a single asynchronous value (something that might not be known right now)
- Creating promises includes either **resolving** them if all went good or **rejecting** them for an error
- USING promises to get a result from an API for example, you can use the .then and .catch methods to deal with the data
    - .then() handles fulfilled/resolved promises
    - .catch() handles rejected promises
- We can use the keyword await inside an async function to wait for the promise to be fulfilled before executing any other code.

APIS - https://www.youtube.com/watch?v=cuEtnrL9-H0&t=155s

EXTRA – if you want to understand how code is executed in JavaScript and how asynchronous stuff works (API calls are asynchronous!)

this video is REALLY good (https://www.youtube.com/watch?v=EI7sN1dDwcY)

Event Loop

- JavaScript is primarily single-threaded, completes the work from one event/callback queue – a.k.a completes one task at a time
- The event/callback **queue** completes or executes the work on a first in first out basis
- The event loop continuously executes the tasks from the event queue until it is empty. It does this in a FIFO manner, which is why the queue data structure is used!

- However, when there are **callbacks**, we add those nested functions onto a call **stack**, as we need to wait until the most nested instruction is done executing and then finish.

- The most recent function calls are completed first (a.k.a the most inner nested)
- Now the event loop executes the item from the queue, and then follows the execution logic until the call stack is empty

- For asynchronous call happens, the callback is not immediately executed, it is actually put at the back of the event queue.
    - For example, a waiter waiting for the kitchen to finish cooking a table's meal. It doesn't wait in the kitchen. They continue working and when the kitchen is done, they don't immediately drop everything to finish delivering the food. They add a mental note to deliver the food (the callback) after they've completed the tasks they already had.
    - Another example, setTimeout(() => { … }, 5000). In this case, the callback function waits 5000ms and then adds the callback function onto the event queue (sometimes why its called the callback queue)