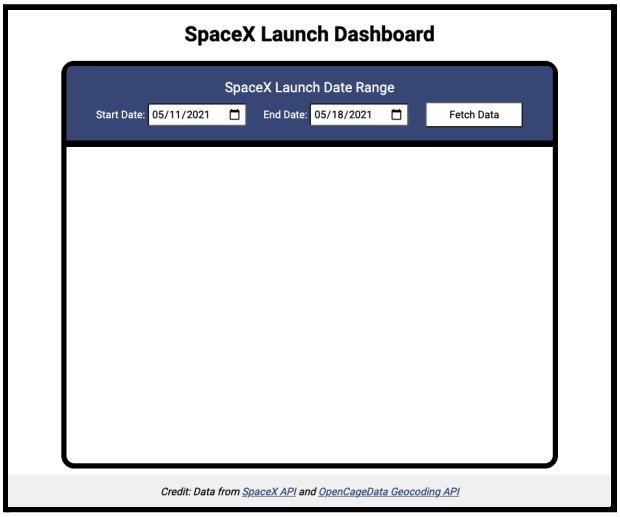
Homework Set 3 - AJAX with `fetch` (SpaceX Dashboard)

Due: Wednesday, May 26th 2AM PST (morning)

Overview

In this assignment, you will continue to get practice working with JavaScript and the DOM, but will learn how to use `fetch` to get data from the web (using APIs) and update the page as a result. Specifically, you will finish implementing a simple launch dashboard displaying results about recent SpaceX launches.

This assignment is about using **AJAX** to fetch data in text and JSON format and process it using DOM manipulation. You will only be writing JavaScript in this assignment - HTML and CSS are provided!



Initial page view

After fetching/populating data

Overview of Starter HTML/JS

In this assignment, you will continue to get practice working with JavaScript and the DOM, but will practice using `fetch` to get data from the web (using 2 APIs) and update the page as a result to populate a dashboard of recent SpaceX launches and their locations.

We have provided `h3-starter.zip` with starting HTML and JS for you to use and build upon. We have also included some CSS, although you're free to add styling on your own!

- `index.html` - provided HTML for the page
- `styles.css` - styles for the web page (these do not need to be changed, you though you may change the styles in your final submission if you would like)
- `spacex.js` - starter JS with function stubs for you to complete. This file also provides a few helper functions at the bottom which you may use if you would like.
- `imgs/spacex_default.png` - default image placeholder for launches without images
- `example-data` – A directory holding example JSON data as JS variables you can use for local testing before implementing fetch calls (or if you reach a request limit).

The parts you need to write in `spacex.js` are indicated with `// TODO` comments, which you should remove in the final submission. Instructions for these functions are provided below. You can also find a demo video of a finished solution on Canvas. At the bottom of `spacex.js`, we have also provided some helper functions you may find useful:

Provided functions:

- **`setDateDefaults()`, `dateToYMD(date)`, `formatUTCDate(date)`:** Helper functions to set and convert date values for the form so that a user can optionally specify a date range to search for launches.
- **`handleError(errMsg)`:** Outputs a string error message to the page, hiding any previous results.If given an error object instead of a string, instead outputs a generic error message to the page (useful for customizing error messages at different cases).
- **`checkStatus(response)`** : checks the status of an AJAX response
- **DOM object and query selectors (`id`, `qs`, `qsa`, and `gen`)** : same as lectures/previous assignments

Overview of APIs Used

API 1: SpaceX API

The SpaceX API offers a variety of endpoints to get data about SpaceX launches, launchpads, etc. In Parts 1-3 of this assignment, you will use it to retrieve data about launches within a specified date range. You will need to use two endpoints for this API to get a collection of launches, as well as data for each launch in that collection. Example data for the two requests can be found in `example-data/localAPI1aData.js` and `example-data/localAPI1bData.js`.

API 2: OpenCage Geocoding API

The second API is a geocoding API called OpenCageData. In Part 4, this API will be used to take the latitude and longitude values retrieved from the second SpaceX API response in Part 3 and convert them to addresses for each launch card on the dashboard. Unlike the SpaceX API, you will need to get an API key for this API, which will allow for up to 2500 requests/day.

External Requirements

For full credit, your JS must behave as specified in this spec, the provided starter code, and provided video. You are free to update the UI (HTML/CSS) and *add* UI features to the dashboard (e.g. implementing pagination, sorting/filtering results, etc. with different endpoint options) as long as you meet the core requirements.

Part 0: Provided Code

The provided `spacex.js` includes starter code to set up the page which you will finish implementing in the following parts. We summarize the starter code here.

As discussed in Lecture 14, we can use the HTML form tag to group form elements like `<input>` tags with optional attributes for client-side validation. In this case, we have two `<input type="date">` elements to provide a user the ability to select a date range for searched SpaceX launches. These HTML5 elements are very useful as they provide a built-in date picker tool for users (which may look a bit different for different browsers).

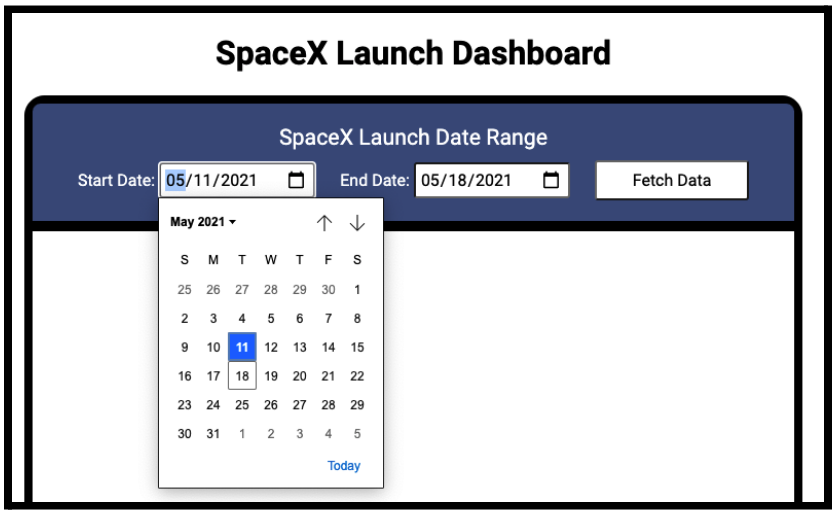
```
<form>
  <label>Start Date: <input type="date" id="start-date"></label>
  <label>End Date: <input type="date" id="end-date"></label>
   <button id="fetch-btn">Fetch Data</button>
</form>
```

On this page, we would like to provide the user the option to specify a date range to search for SpaceX launch data. But it would also be nice to include a default date range up to the current date. Since JS Dates can be a bit nuanced (but are still very useful to know!), we have included two functions for you that are used to improve the search options (and additional `formatUTCDate` function is provided for use in Part 2).

When the page loads, the `setDateDefaults` function is called to calculate the current date (which will be the default end date) and the default start date, which is 28 days earlier for a default range of 4 weeks. 28 days is just a standard range we've decided, so we've also specified this as a `DEFAULT_DATE_RANGE` program constant for easy program modification.

`setDateDefaults` is used to set the value of both date inputs before the user interacts with them. This function also uses a provided `dateToYMD` function which takes a Date object and returns a "YYYY-MM-DD" string representation of the date which is a required format for a date input's value attribute (unfortunately, with all the Date object methods we have access to, this is not one of them built-in).

When you open the page, you should see this behavior in action!



Finally, we have implemented the `init` function (with a helper `handleSubmit` function) for you so that when the form is submitted, the default submit behavior (a page refresh) is prevented (as discussed in Lecture 14) and the selected date range is validated before starting a fetch request to populate the dashboard.

```
function init() {
  setDateDefaults();
  qs("form").addEventListener("submit", handleSubmit);
}

function handleSubmit(evt) {
  evt.preventDefault();
  id("launches").innerHTML = ""; // clear any previous data
  if (id("start-date").value > id("end-date").value) {
    handleError("Error: Start date should not be later than end date.");
  } else {
    id("message-area").classList.add("hidden"); // hide any previous error message
    fetchLaunches();
  }
}
```

The rest of the implementation starts with a call to `fetchLaunches`, which you are to implement with the other functions discussed below.

Note: You may use either the `async`/`await` or fetch Promise pipeline syntax in this assignment, as long as you are consistent, though we encourage you to use `async`/`await`. You may need to modify the function stubs to properly identify the `async` functions (remember that you must label a function as `async` in order to use `await` within!).

Another note: It can sometimes be useful to test your data processing and DOM manipulation first before implementing your fetch calls. This way, you can isolate any potential bugs as syntax/UI-specific issues vs. fetch issues (e.g. an invalid fetch request, a slow/failed network connection, using the wrong endpoint, etc.). We have provided example JSON responses for each of the three endpoints you'll use in the `example-data` directory. These are saved as local variables (`localAPI1LaunchesData`, `localAPI1LaunchData`, and `localAPI2Data`) for the three requests you'll make (in order). This data comes from successful requests made to the APIs for 8 launches prior to May 16th, 2021. You can access these global variables by uncommenting the three `<script>` tags in `index.html` and using them as temporary response data in functions that make fetch requests. Make sure to remove these `<script>` tags in your final version though; if you rely on these local JSON values in your submitted version instead of correctly implementing the fetch responses described below, you will lose a non-trivial number of points on the assignment.

Summary of Requests per Search

There are a few different endpoints we'll be working with, but in total you should make:

- 1 request to API 1's `POST /launches/query`
  - Example response in `example-data/localAPI1aData.js`
- X requests to API 1's `GET /launches/:launchpadid` where X is the number of elements in the `.docs` value of the previous response
  - Example response in `example-data/localAPI1bData.js`
- X requests to API 2's `GET ?key=<OPENCAGE_KEY>&q=<lat>+<long>` (one for each launchpad id)
  - Example response in `example-data/localAPI2Data.js`

Part 1: Fetch Launches

The `fetchLaunches` function starts the series of requests that is used to get data needed to populate the launch dashboard.

Both endpoints using the SpaceX API (Part 1 and Part 3) will use the base URL of https://api.spacexdata.com/v4/ which is also defined as a `SPACEX_API_URL` program constant in the provided `spacex.js`.

This function will initiate the first request to the `/launches/query` endpoint of the SpaceX API. This endpoint is a bit different than some of the simpler GET requests we've seen, and supports various query options **sent through a POST request**. We will only specify a few options for date

ranges, max launch limit, and sorting, but you are welcome to add support for more query filters on your own, as long as you meet the assignment requirements. We'll use this endpoint to search for launches between the two dates selected in the UI menu bar. The API supports a range of dates with the date_utc option in the request's query object. You will need to finish the `// TODOs` in this function to finish specifying the date parameters in the `requestBody`, and pass the `fetchOptions` defined below as a second parameter to a fetch request to the `/launches/query` endpoint.

`fetch` with POST vs. GET

Using fetch with a POST request works a bit differently than GET requests (which are default for fetch). First, you need to add a second parameter to the fetch request to pass in options, including the HTTP method (POST), any header information (in this case, just `{"Content-Type": "application/json"}`), and finally, the request body (the alternative to sending options as URL parameters in a GET request). The body will be a stringified representation of the options you finished specifying above.

```
let fetchOptions = {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify(requestBody)
};
```

For example, if we wanted to search between May 11th, 2021 and May 18th, 2021, we would pass in the following query object to the request:

```
{
  "query": {
    "date_utc": {
      $gte: "2021-05-11T00:00:00.000Z",
      $lte: "2021-05-18T00:00:00.000Z"
    }
  }
}
```

We can get the selected dates using the `.value` attribute for the two date inputs, but this just gives us the YYYY-MM-DD strings. To convert them to the expected ISO format above, you'll need to construct a new Date object with the string value, and then use the `.toISOString()` Date method.

The `$gte` ("greater-than or equal to") value should correspond to the ISO String for the `#start-date` input and the `$lte` ("lesser-than or equal to") value should correspond to the ISO string for the `#end-date` input.

Processing the Response JSON

Once the fetch request is correctly configured and sent, the API will return a JSON response with a `docs` key that maps to an array of all of the launches within the specified date range (an example response is found in `localAPI1aData.js`). There are some other pagination keys in the response that are not relevant to this assignment, but you can learn more about the response format here.

Finish `// TODO 1c` to check the status of the response (using the provided `checkStatus` function), parse the response JSON, extract the array of launch data objects within the `.docs` attribute, and pass that array to the `displayLaunches` function which you'll implement in Part 2. If an error occurs during the fetch request (e.g. an error thrown by `checkStatus`), use the provided `handleError` function appropriately to display an error message on the page. If you want to test with the local data first, you can just use the `.docs` array from `localAPI1LaunchesData`.

Aside: Why POST?
You may notice that our first call to the SpaceX API is a POST request (rather than a GET request). This may seem strange at first considering that all we're doing is retrieving data from the API. However, we had to include some extra data in `requestBody` to specify the query options supported by the API in our request (mainly that we got the launches within the specified date range window). If we hadn't used this `/launches/query` endpoint as a POST request, we wouldn't be able to specify these things and would simply be given a list of over a hundred launches. Processing only 8 launches at most in this example, you can observe that this is wildly inefficient.

To send a potentially-considerable amount of data (getting the correct number of launches from the correct time period, correctly sorting them by date, among other query options), the authors of the API decided to use a POST request as it allows clients to send a "request body" which can contain much more information than a simple GET URL parameter. This is starting to become a very common strategy among API developers for requests that can send over more data (remember that GET request parameters are sent through the URL, POST parameters are not). If you ever have a question about which type of request to use, consult the documentation! API documentation should give you information on the endpoint and the type of request(s) which can be served through that endpoint.

And one more note (particularly for upcoming API-developing assignments) - just because there are advantages to the POST request in this case, GET requests are still more appropriate for simple requests to an API that 1) just retrieve data from a resource instead of updating data on the

server, 2) aren't sending more parameters than would be reasonable in a URL string, and 3) it is not a security concern to have the request parameters visible in the URL.

Part 2: Processing Launches

Next, you will process the JSON response you get from Part 1 to populate the `#results-view` on the dashboard with "launch card" articles. The first response you got includes a lot of useful data about the collection of launches returned from the endpoint - almost everything we need to populate each launch card. In this part, you'll do most of the work for generating an `article` element for each card, and in Part 3 you will make a second request to the SpaceX API to get latitude and longitude coordinates for each launch.

```
<article>
  <img src="" alt="">
  <h2>Launch Name</h2>
  <p>Month D, YYYY</p>
  <p>Address</p>
</article>
```

Card Image
First, the image should be generated and added to the specific launch's `article`. Each launch may have an image path returned by the response, found at the launch's `links.path.small` attribute (refer to the example here, which has only 1 launch in the `docs` array though you may have more to process at a time). If this attribute is `null`, use the default `imgs/spacex_default.png` in the provided `imgs/` folder for the card's image. Make sure to provide appropriate alt text for any generated images.

Card Title
The name of the launch should be appended to the `article` as an `h2` element. In the linked example, the single launch's name was "KoreaSat 5A".

Date of Launch
Each launch object has information about the date of the launch stored in the `date_utc` key. This is in a UTC date/time format (e.g. "2017-10-30T19:34:00.000Z"), which isn't exactly what a user might want to see. We'll use the more useful format of "Month, Day YYYY" (e.g. "October 30 2017"). This will require a bit more date-conversion though, since this representation is unique to US English formats.

We have provided a `formatUTCDate` function to use in this part, but you will need to construct the date using the `date_utc` key of the corresponding launch data, and append the formatted string result to the `article` as a paragraph as shown above.

At this point, you should make sure to have your launch card articles correctly constructed and appended to the `#results-view`.

Example result of adding a single launchpad card:


```
<article>
  <img src="https://imgur.com/BrW201S.png"
       alt="Mission patch">
  <h2>Starlink-26 (v1.0) + Capella-6 + Tyvak-0130</h2>
  <p>May 15, 2021</p>
  <!-- Appended in Part 2, populated in Part 4 -->
  <p>
    Pad 39A, Crawlerway, Brevard County, FL,
    United States of America
  </p>
</article>
```

You are welcome to add other information/filters on these cards (e.g. a dropdown view for more details), as long as you include at minimum the ones above.

Make sure you have the expected number of articles with the correct structure before moving on to the next part to get the address for the final paragraph in each article. To avoid extra processing/looping over the DOM, you should fetch the address information (in Part 4) for each launchpad's latitude/longitude coordinates *while* generating the `article` element. You can still append the article to the page while the fetch request is pending (you should see a slight delay in the paragraph text appearing which is ok).

Part 3: Getting Launch Coordinates

Finally, we want to add information about the address of the launch on the card - unfortunately, the data in the `docs` array in the first API endpoint doesn't quite have what we need, so we'll need to make a request to another endpoint to get more launch-specific information using **each** launchpad id from the first response's `docs` array. The `launchpad` attribute for each `.docs` item will hold the id you should use here (not the `id` attribute).

API 1 Endpoint 2: Getting launchpad data
`GET /launchpads/:launchpadid`

Example:
https://api.spacexdata.com/v4/launchpads/5e9e4502f509094188566f88

This endpoint gives us some details about each launchpad (specified by its launchpad id from Part 2) which you can read more about here. But we want to specifically get the latitude and longitude coordinates to find the corresponding address (we could instead use the region or locality keys, but that wouldn't be as specific).

In this part, finish `getLaunchPad` to fetch from this second endpoint (for **each** launchpad id processed in Part 2) to check the status of the response, parse the JSON, and finally call `setLaunchpadAddress` with the response data's latitude and longitude values as well as the empty `para` parameter passed originally from Part 2. Even though this paragraph was created and appended in `displayLaunches`, any updates in `setLaunchPadAddress` will update the text content of the paragraph even after it's been appended to the DOM. For local testing, you can find an example response for the second endpoint in `example-data/localAPI1bData.js` which corresponds to the launchpad with id of "5e9e4502f509094188566f88".

Part 4: Geocoding!

Now, you're ready to implement the final API request using the OpenCageData API. This API is used to take the latitude and longitude coordinates from the `/launchpads/:launchpadid` response and return a corresponding address in a JSON response. If you want to test with local data first, you can find an example response in `example-data/localAPI2Data.js`.

As mentioned in the API Overview, you will need to sign up for an API key for the OpenCageData API which you can find on the API documentation page.

The API uses the base UR of https://api.opencagedata.com/geocode/v1/json which is defined as `OPENCAGE_API_URL` in the provided `spacex.js`.

To get an address for a pair of latitude/longitude coordinates, you'll use the following endpoint appended to the geocoding URL:

```
?key=API_KEY&q=<latitude>+<longitude>
```

Use the passed latitude and longitude values from Part 3 to finish `setLaunchpadAddress`, updating the text content of the passed paragraph element to the first address in the OpenCage response's results array (using the `formatted` address value).

Internal Correctness Requirements

For full credit, your page must not only match the External Requirements listed above, you must also demonstrate that you understand what it means to write code following a set of programming standards. Your JS should also maintain good code quality by following the CS 101 Code Quality Guide. Make sure to review the section specific to JavaScript! We also expect you to implement relevant feedback from previous assignments. Some guidelines that are particularly important to remember for this assignment:

JavaScript:

- Your .js file must be in the module pattern and run in strict mode by putting `"use strict";` within the module pattern.
- Do not use any global variables, and minimize the use of module-global variables. Do not ever store DOM element objects, such as those returned by the `id`, `qs`, or `qsa` functions, as global variables.
    - If you use the local JSON data before implementing fetch requests, make sure you remove the `<script>` tags in index.html so that these variables are not globally-accessible/used.
- Do not save JSON results as module-global variables, unless it makes sense to in an extra feature you implement
- Format your JS to be as readable as possible, similarly to the examples from class: Properly use whitespace and indent your CSS and JS code as shown in class. You can find information about JS conventions in the Code Quality Guide.
- Use camelCase naming conventions for variables and functions.
- Express all stylistic information on the page using CSS defined in `styles.css` - do not set styles in JS (use `classList` instead).
- Remove any TODOs/helper comments in provided code. Your final submission should also not have any debugging code (e.g. `console.log` or `debugger)` or commented-out code.
- Don't call APIs more than you need to.
- Store common/complicated operations in variables (e.g. variables to access JSON data).
- You may not use any JS frameworks for this assignment.
- Your page must use valid JS and successfully pass both JSLint with no errors.

Documentation

The HTML and CSS are provided for you, but for full credit in documentation you must replace the file documentation for `spacex.js` with your student information as well as a descriptive file comment.

You will be expected to properly document the functions in spacex.js using JSDoc with `@param` and `@returns` where necessary/incomplete and for any helper functions you add. Refer to the Code Quality Guide for some examples. **Remove any TODOs/commented-out code in the HTML/JS for full credit in the final submission.**

Grading

This assignment will be out of 15 points and will be distributed as follows:

- External Correctness (9 pts)
- Internal Correctness (5 pts)
- Documentation (1 pts)