Lab 09 - API

Description

In this lab, you'll learn how to interact with a real-world API (the New York Times Article Search API) and use the fetched data to build a dynamic web application using React. You'll gain experience with fetching data, parsing JSON responses, and displaying data in a user-friendly format.
You can see more about available APIs and request access at https://developer.nytimes.com.

Objective

Create a React web application that displays a grid of clickable New York Times article images, similar to the example provided. Each image should link to the original article on the New York Times website.

Getting Started

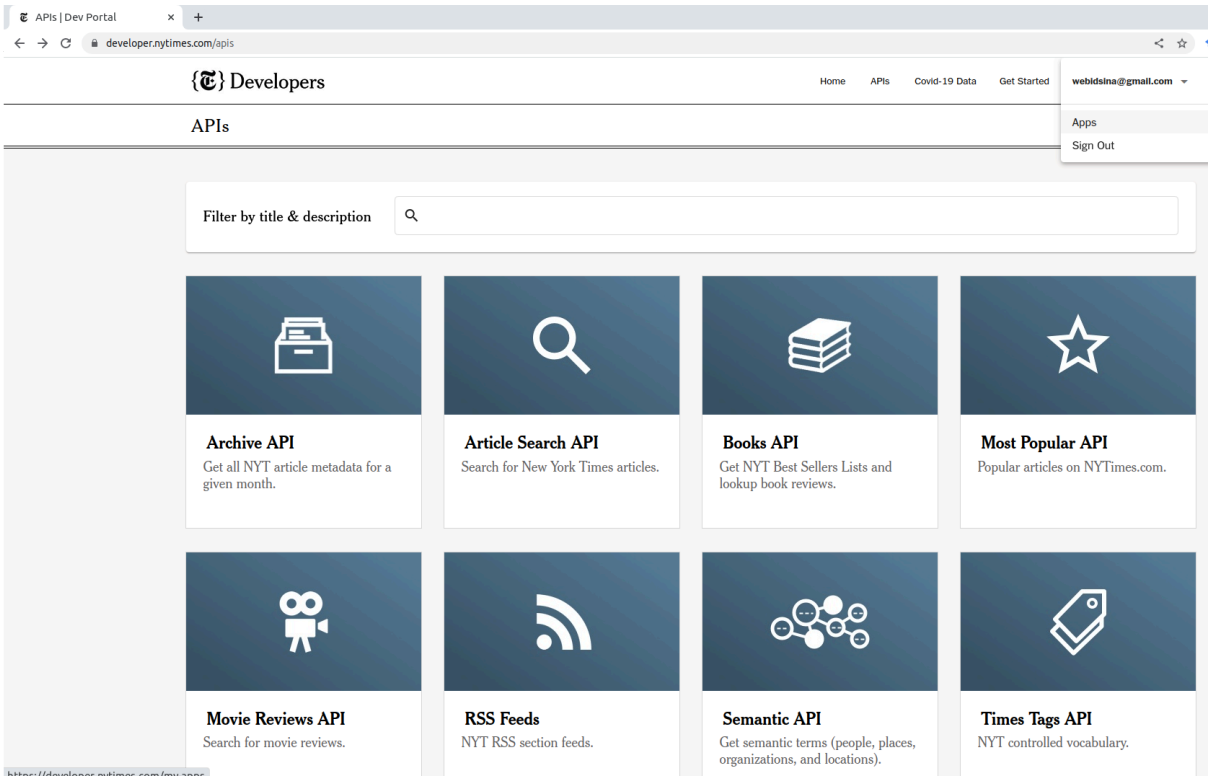
Obtain a New York Times API Key:

- Visit the New York Times Developer Network: https://developer.nytimes.com
- Create an account and request a developer key for the "Article Search API."
- Familiarize yourself with the API documentation and the structure of the JSON response. You can test the API with your key here: `https://api.nytimes.com/svc/search/v2/articlesearch.json?q=QUERY&api-key=YOUR_KEY` (replace `QUERY` and `YOUR_KEY`).

Task 1: API Key Acquisition (5 points)

Request a developer key from https://developer.nytimes.com.
Click on Apps, and request a developer key.

- Successfully obtain a developer key for the New York Times Article Search API.
- Submit a screenshot of your developer key request or confirmation.



Task 2 - Getting Data (10 points)

Now, we want to use this API to fetch content from NYTimes and show it on our web app.
First, let's create a new React component named ArticlesGrid to store the data in the page.

- Create a new React component named `ArticlesGrid`.
- Use the `useEffect` hook to fetch data from the New York Times API.
    - Use the provided API endpoint and your API key.
    - Fetch data with a relevant search query (e.g., "technology," "politics").

```
function ArticleGrids() {
   const [articles, setArticles] = React.useState([]);
...
```

Then start by getting some data from the API with a search term. We will do this with jQuery ***useEffect*** hook.

```
React.useEffect(() => {
                                    const          url          =
'https://api.nytimes.com/svc/search/v2/articlesearch.json?q=QUERY&'
```

```
      + 'api-key=YOUR_KEY'; // replace query and your_key

   const fetchData = async () => {
       const response = await fetch(url);
       const data = await response.json();
       setArticles(parse(data));
   };
```

The useEffect hook lets you perform "side effects" in functional React components. Side effects are any actions that interact with the outside world or cause changes that React needs to respond to. Basically, when we need a way to tell React, 'After rendering, also perform this side effect...', we use the useEffect hook.

Here is a sample of useEffect example:

```
useEffect(() => {
// Code to fetch data from an API
fetch('https://api.example.com/data')
   .then(response => response.json())
   .then(data => setData(data));
}, []); // The empty array controls when the effect runs
```

Note: The empty array is called the *dependency array*. This array is the second argument to the useEffect hook. It controls when your effect function should re-run. The dependency array can contain props or state values. React compares the current values in the array to the values from the previous render. If any value has changed, the effect function runs again. In our example, It's empty. An empty array signals to React that your effect doesn't depend on any changing props or state values from your component.

Use this code to get data from NYTimes API (or any api).

```
React.useEffect(() => {
                                                const       url       =
'https://api.nytimes.com/svc/search/v2/articlesearch.json?q=QUERY&'
           + 'api-key=YOUR_KEY; // replace query and your_key

       const fetchData = async () => {
           const response = await fetch(url);
           const data = await response.json();
           setArticles(parse(data));
       };

       fetchData();
```

```
}, []);
```

Task: Get the data from API and print it in your console. Take a screenshot of the output.

Task 2 - Parsing Data (20 points)

Now, we need to parse the returned JSON data. Open the url and key in your browser to see the full response:
https://api.nytimes.com/svc/search/v2/articlesearch.json?q=QUERY&api-key=YOUR_KEY
Note: you need to have a api key from article search api.

Task: The NY Times API will return a lot of raw data. For each article, keep the image URL, title, and URL to original article if a large image exists.

```
const parse = (results) => {
   if (!results || !results.response) return [];
   const articles = results.response.docs;
   return articles.filter(article => article.multimedia.find(isXL))
       .map(article => ({
           id: article._id,
           title: article.headline.main || 'Untitled',
           imageURL: article.multimedia.find(isXL).url || '#',
           webURL: article.web_url || '#'
       }));
};
const isXL = (image) => image.subtype === 'xlarge';
```

Task 3 - Displaying Data (15 points)

Now that we have the data, let's make a component to display each article! This is shorthand for a component that only has a "render" function. Use this code to create an article component.

```
const Article = ({ article }) => {
   var imgURL = 'https://static01.nyt.com/' + article.imageURL;
   return (
       <div className='article'>
           <a className='article-link' href={article.webURL}>
                 <img className='article-image' title={article.title}
```

```
src={imgURL} />
            </a>
        </div>
    );
}
```

Finally, you need to render the Articles Grid to tie it all together.

```
return articles && ( // Conditional rendering
    <div className='articles'>
        {articles.map(article => (
            <Article article={article} key={article.id} />
        ))}
    </div>
);
```

Task 4: Fetching and Displaying Movies from TMDB (50 points)

- Create a new React component named `MovieGrid`.
- Use the `useEffect` hook to fetch movie data from The Movie Database (TMDB) API's "Discover" endpoint.
    - Use your TMDB API key.
    - Fetch data with relevant parameters (e.g., sort by popularity, specify genres).
- Parse the JSON response:
    - Extract the poster image URL, title, and movie overview for each movie.
    - Store the parsed data in the component's state.
- Create a separate `Movie` component to display individual movies.
- Pass the parsed movie data (poster URL, title, overview) as props to the `Movie` component.
- Display the movie poster image and title.
- Render the `MovieGrid` component and map over the parsed data to display the movies in a grid layout.
- Use appropriate CSS to style the grid and movie elements.