JavaScript Challenge 3 - Integrating OMDB and Star Wars APIs

Introduction

In this exercise, you will extend your previous experience with the OMDB API by integrating it with the Star Wars API. You'll create a dynamic application to display movie details and character information from the Star Wars universe.

This exercise is both demanding and invigorating, designed to provide an advanced challenge for those ready to tackle complex coding scenarios. It will put your skills to the test as you work with intricate code, integrating two different APIs using asynchronous JavaScript. Beyond merely executing API calls, you'll be delving deep into the DOM, manipulating elements and data to create a cohesive and dynamic user experience.

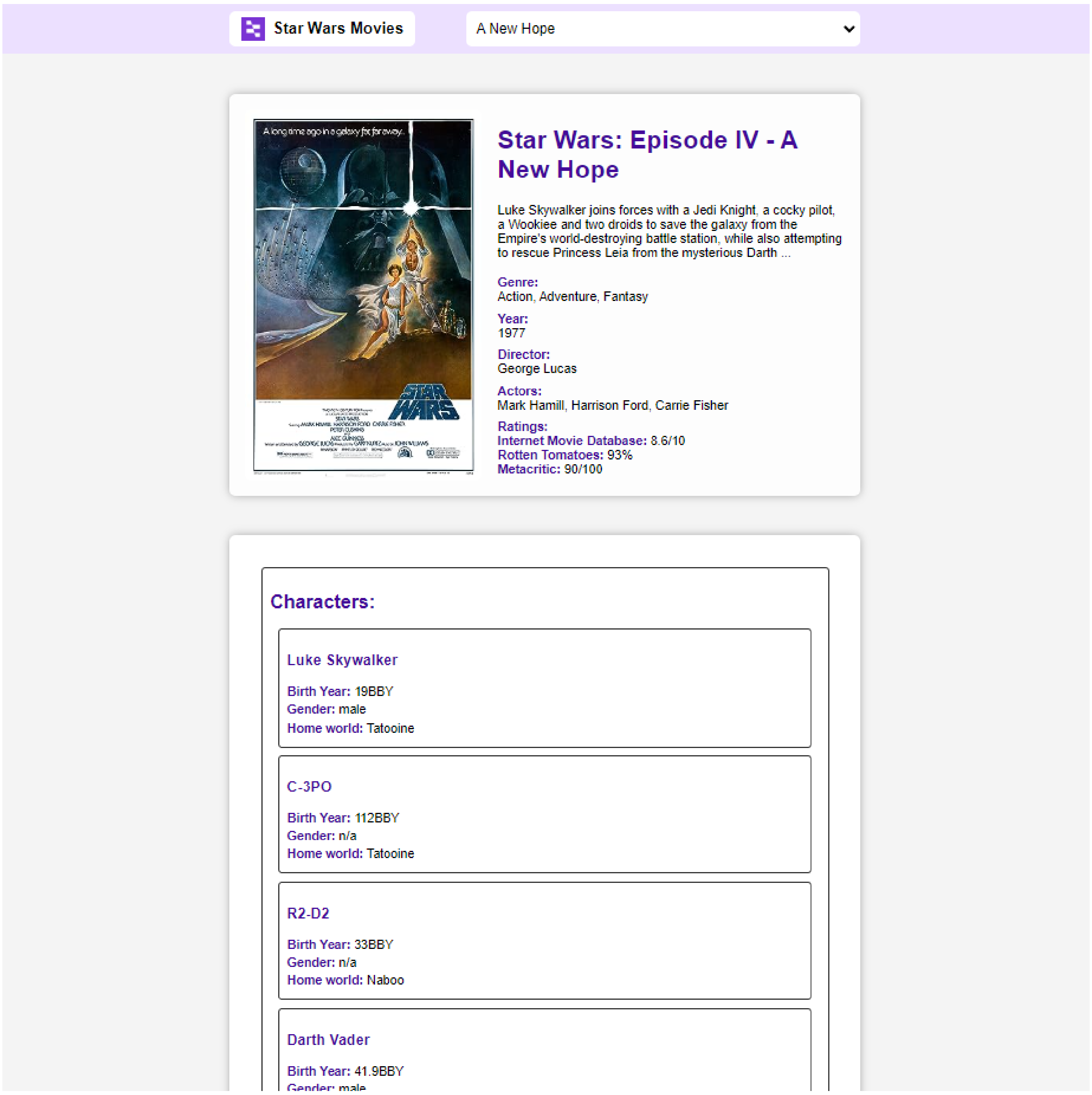
This is a rigorous journey that will solidify your understanding of key web development concepts.

Expect to be challenged, anticipate learning, and prepare to grow.

Good luck!

UI Images

Main Screen



Star Wars: Episode IV - A New Hope

Luke Skywalker joins forces with a Jedi Knight, a cocky pilot, a Wookiee and two droids to save the galaxy from the Empire's world-destroying battle station, while also attempting to rescue Princess Leia from the mysterious Darth ...

Genre:
Action, Adventure, Fantasy

Year:
1977

Director:
George Lucas

Actors:
Mark Hamill, Harrison Ford, Carrie Fisher

Ratings:
Internet Movie Database: 8.6/10
Rotten Tomatoes: 93%
Metacritic: 90/100

Characters:

Luke Skywalker

Birth Year: 19BBY
Gender: male
Home world: Tatooine

C-3PO

Birth Year: 112BBY
Gender: n/a
Home world: Tatooine

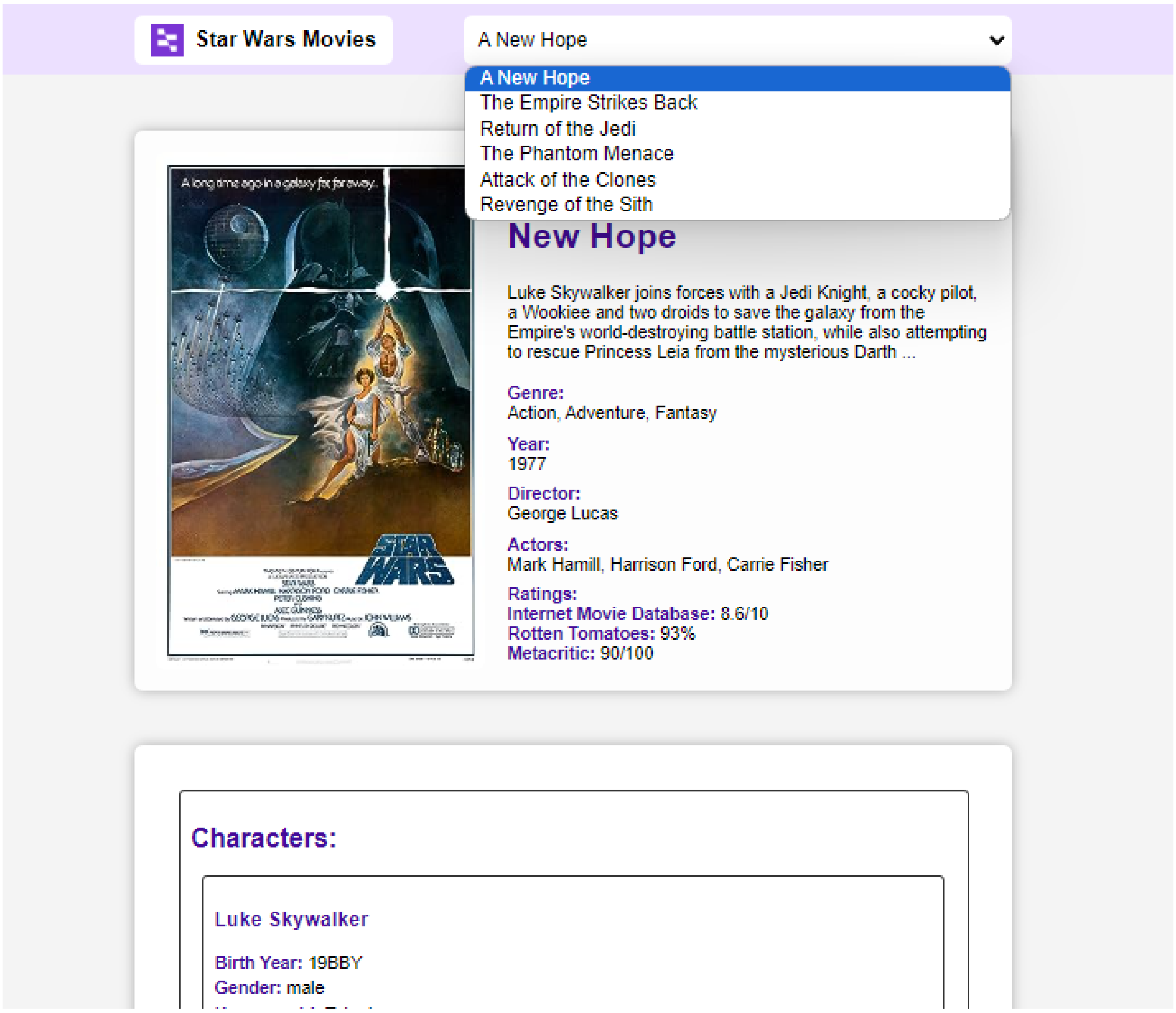
R2-D2

Birth Year: 33BBY
Gender: n/a
Home world: Naboo

Darth Vader

Birth Year: 41.9BBY
Gender: male

Select Dropdown



Star Wars Movies

A New Hope

A New Hope
The Empire Strikes Back
Return of the Jedi
The Phantom Menace
Attack of the Clones
Revenge of the Sith

New Hope

Luke Skywalker joins forces with a Jedi Knight, a cocky pilot, a Wookiee and two droids to save the galaxy from the Empire's world-destroying battle station, while also attempting to rescue Princess Leia from the mysterious Darth ...

Genre:
Action, Adventure, Fantasy

Year:
1977

Director:
George Lucas

Actors:
Mark Hamill, Harrison Ford, Carrie Fisher

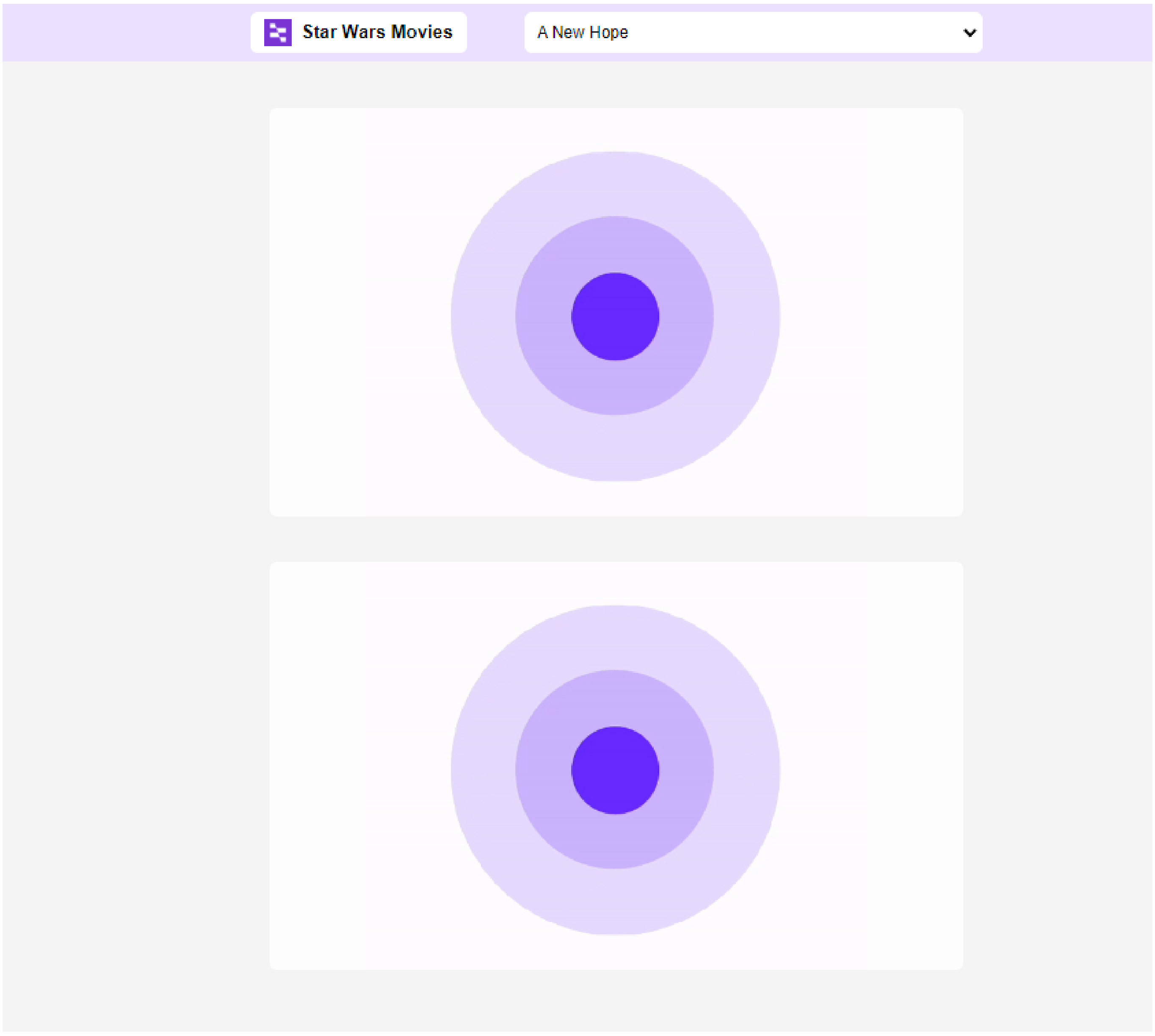
Ratings:
Internet Movie Database: 8.6/10
Rotten Tomatoes: 93%
Metacritic: 90/100

Characters:

Luke Skywalker

Birth Year: 19BBY
Gender: male

Loading Gifs



Objectives

1. **Fetch Data from Multiple APIs**: Learn to combine data from two APIs (OMDB and SWAPI) and display the information seamlessly in the user interface.
2. **Handle Asynchronous Operations**: Enhance your understanding of asynchronous programming with `fetch`, `async`, and `await`.
3. **Create Dynamic Templates**: Develop reusable functions to create HTML templates for movies and characters.
4. **Implement Event Listeners**: Listen for user actions to trigger updates in the UI.

HTML & CSS

- HTML and CSS code is provided.

Understanding the Data Structure from the swapi.dev API

Before starting the exercise, it is crucial to familiarize yourself with the data structure retrieved from the swapi.dev API. You will encounter information related to Films and Characters, each containing unique properties and associated URLs. Carefully explore the JSON response, identify the relevant attributes, and analyze how they relate to one another. This understanding will be essential for mapping the data to the HTML templates in this exercise. Your ability to interpret the structure will play a vital role in successfully completing this exercise.

Exercise Walkthrough

Step 1: Define Constants for API URLs

Before fetching data from APIs, we need to have their endpoints. Define two constant variables for the URLs of the APIs to be used.

```
const SWAPI_URL = 'https://swapi.dev/api/films';
const OMDB_URL = 'http://www.omdbapi.com/?apikey={your API key here}=';
```

- `SWAPI_URL` : This constant stores the URL for the Star Wars API, which provides data about films.
- `OMDB_URL` : This constant holds the URL for the OMDB API, which offers detailed information about movies. **Don't forget to include your API key in the URL**.

Step 2: Grab References to HTML Elements

Next, you'll want to reference specific HTML elements in your JavaScript code to dynamically update the content.

```
const movieContainer = document.getElementById('movieContainer');
```

- `movieContainer` : This reference points to the HTML element with the id 'movieContainer'. You'll use this reference to insert movie details into the DOM.

```
const movieSpinner = document.getElementById('movie-spinner');
```

- `movieSpinner` : This reference targets the HTML element with the id 'movie-spinner'. This is a gif image of a loading spinner that shows while data is being fetched.

```
const charactersCardContainer = document.querySelector('.characters-card-
container');
```

- `charactersCardContainer`: This reference targets the first element with the class 'characters-card-container'. You'll use this to insert character information into the DOM.

```
const charactersSpinner = document.getElementById('characters-spinner');
```

- `charactersSpinner`: Similar to the movie spinner, this reference targets the HTML element with the id 'characters-spinner'. It can be used to show a loading gif when fetching character information.

Step 3: Declare an Asynchronous Function `fetchData`

Use the `async` keyword to declare an asynchronous function named `fetchData`.

This function recievs a `url` parameter representing the URL that you want to fetch data from.

Fetch Data from the URL

Inside the function, utilize the `fetch` method to send a request to the provided URL. Store the response in a `response` constant.

Return the JSON Data

Return the `response.json()`.

```
  async function fetchData(url) {
    const response = await fetch(url);
    return response.json();
  }
```

Step 4: Create a `createMovieTemplate` function

Write a `createMovieTemplate` that receives a `data` parameter and returns the movie html as a template literal with the data details. You will find the movie HTML in the folder of this exercise.

Step 5: Create a `createCharacterCard` function

Write a similar function as the `createMovieTemplate` for the character card named `createMovieTemplate`. It accepts `characterDetails` parameter and returns a template literal of a character card with the relevant data from the `characterDetails` argument. The HTML of the character card is provided in the exercise folder.

Step 6: Create an Async Function `appendCharacterCard`

1: Declare Asynchronous Function

Declare an asynchronous function named `appendCharacterCard` with a single parameter, `movieUrl`, representing the URL to fetch the movie data.

2: Begin Try Block

Wrap the code inside a try-catch block to handle any potential errors during the asynchronous operations.

3: Hide Character Card Container and Show Spinner

Initially set the display property of the character card container to `none` and display the display property of the loading spinner to `flex`, so it will provide a visual indication that the character data is being loaded.

4: Clear Previous Character Cards

Clear any previous character cards from the container by setting it's `innerHTML` property to an empty string.

5: Fetch Movie Data

Call the `fetchData` function (previously defined) with the `movieUrl` parameter to retrieve the movie's data, including character URLs, and assign it to a `movieData` constant.

6: Create Characters Container

Create a new `<div>` element to hold the character cards, store it to a constant `charactersContainer` and set its class name to `characters-container`.

Create a `h2` element and assign it to a `title` constant. Set the text content of the `title` constant to `'Characters:'` and append it to newly created characters container div.

7: Concurrently Fetch Character Details

Extract the character URLs from `movieData.characters` to a `characterUrls` constant. Map to an array named `characterDetailsPromises` all of the character details Promises. To achieve this, pass to the `map` method the `getCharacterDetails` function (we will create it in the next step) and pass a `characterUrl` as an argument in each iteration.

Use `Promise.all` with the created array to fetch all character details concurrently, and store the results array in a `charactersDetails` variable.

8: Loop Through Characters and Append Cards

Iterate through the fetched character details and create character cards using the `createCharacterCard` function. Store the result in a `characterCard` constant.

Add each card to the `innerHTML` of the characters container like this:

```
charactersContainer.innerHTML += characterCard;
```

9: Append the characters container

Append the characters container to the characters card container.

7: Hide Spinner and show the characters card container

Set the display property of the loading spinner to `none` and set the display property of the characters card container to `block`.

9: Error Handling

In the catch block, call the `handleError` function (see step 12) with the relevant arguments.

```
    handleError(charactersSpinner, charactersCardContainer, error);
```

Step 7: Create an Async Function `getCharacterDetails`

1: Declare Asynchronous Function

Declare an asynchronous function named `getCharacterDetails` with a single parameter, `characterUrl`, representing the URL to fetch the character data.

2: Beginning Try-Catch Block

Initiate a try-catch block to ensure proper error handling.

2: Initiate Data Fetch

First, make an asynchronous call to the `fetchData` function with the given character URL and store the parsed response to a `characterData` constant.

3: Fetching Homeworld Data

With the character data obtained, make a second asynchronous call to the `fetchData` function using the homeworld URL from the character data. This call retrieves detailed information about the character's homeworld.

4: Creating and Returning Character Object

Once both the character data and homeworld data are obtained, construct an object containing selected details about the character:

- **Name:** The character's name.
- **Birth Year:** The character's birth year.
- **Gender:** The character's gender.
- **HomeWorld:** The name of the character's homeworld, obtained from the previously fetched homeworld data.

Return this object, allowing other parts of the code to access these specific details about the character.

5: Error Handling

In the catch block log the error to the console with `console.error`.

Step 8: Create an Async Function `fetchMovieData`

1: Declare Asynchronous Function

Declare an asynchronous function named `fetchMovieData` with a single parameter, `selectedUrl`, representing the URL to fetch the movie data.

2: Beginning Try-Catch Block

Initiate a try-catch block to ensure proper error handling.

3: Fetch Movie Data

Make an asynchronous call to the `fetchData` function with the given selected URL and store the parsed response to a constant named `data`.

4: Search for the Movie

With the movie data obtained, make an asynchronous call to the `searchMovie` function using the title of the movie obtained from the data. This will search for the movie and handle its rendering within the UI. No need to store the result to a constant, since the `searchMovie` doesn't return anything.

5: Append Character Card

Next, make an asynchronous call to the `appendCharacterCard` function with the selected URL. This function is responsible for creating and appending character details to the associated movie card. No need to store the result to a constant, since the `appendCharacterCard` doesn't return anything.

6: Error Handling

In the catch block, log the error to the console with `console.error`.

Step 9: Create an Async Function `searchMovie`

1: Declare Asynchronous Function

Declare an asynchronous function named `searchMovie` with a single parameter, `title`, representing the title of the movie to search for.

2: Beginning Try-Catch Block

Initiate a try-catch block to ensure proper error handling.

3: Display Loading Spinner

Set the display style of `movieSpinner` to 'flex', making the loading spinner visible while the movie data is being fetched.

4: Fetch Movie Data from OMDB API

Concatenate the OMDB_URL with the title of the movie and make an asynchronous call to the `fetchData` function with this concatenated URL, exactly like this:

```
const data = await fetchData(OMDB_URL + "- " + title);
```

Store the parsed response to a constant named `data`.

5: Create Movie Template

Call the `createMovieTemplate` function with the fetched data to create the HTML template for the movie and store the returned value to a `template` constant.

6: Hide Loading Spinner and Display Movie Container

Set the display style of `movieSpinner` to `none` to hide the loading spinner, and set the display style of `movieContainer` to `block` to make the movie container visible.

7: Inject Movie Template into Container

Set the `innerHTML` of `movieContainer` to the previously created movie template. This will render the movie details within the container.

8: Handle Movie Ratings

Call the `handleRatings` function, that we will write in the next step, with the ratings from the fetched data and the element with the id 'movieRatings'. This function will handle the display of the movie's ratings.

```
handleRatings(data.Ratings, document.getElementById('movieRatings'));
```

9: Error Handling

In the catch block, call the `handleError` function (see step 12) with the relevant arguments.

```
handleError(movieSpinner, movieContainer, err);
```

Step 10: Define a Function `handleRatings`

Same as the function in the 'Movie Search Engine' exercise.

Step 11: Create an Async Function `initializeMovies`

1: Declare Asynchronous Function

Declare an asynchronous function named `initializeMovies` that receives no parameter, responsible for initializing the movie-related functionality in the application.

2: Beginning Try-Catch Block

Initiate a try-catch block to handle any potential errors during the asynchronous operations within the function.

3: Fetch Movies from API

Make an asynchronous call using the `fetchData` function with the specified `SWAPI_URL` , and then store the parsed JSON response in a constant named `data` .

4: Obtain Movie Select Element

Find the HTML element with the ID 'movieSelect', which represents the select element for choosing a movie, and store it in a constant named `movieSelect`.

5: Retrieve Selected Movie URL

Access the `value` of the currently selected option within the `movieSelect` element, and store it in a constant named `selectedUrl`.

```
  const selectedUrl =
movieSelect.options[movieSelect.selectedIndex].value;
```

6: Fetch Movie Data

Call the previously defined `fetchMovieData` function, passing the `selectedUrl` as a parameter. This function will take care of fetching the movie data, searching for the movie title, appending character cards, and handling the UI components related to the selected movie. Since this function is not returning anything, no need to store nothing to a constant.

7: Error Handling

In the catch block, log the error to the console with `console.error`.

Step 12: Create Function `handleError`

1: Declare Function

Declare a function named `handleError`, with three parameters:

- **spinnerElement:** The HTML element representing the loading spinner.
- **containerElement:** The HTML element representing the container where the error message will be displayed.
- **error:** The error object to be logged.

2: Hide Loading Spinner

Set the display style of the `spinnerElement` to 'none', hiding the loading spinner that may have been visible during an ongoing operation.

3: Display Error Message

Set the `innerHTML` of the `containerElement` to an HTML structure containing a div with class "error" and an h1 element displaying a user-friendly error message. This will render the error message within the specified container.

```
  containerElement.innerHTML = `<div class="error"><h1>An Error Occurred. Please try again later.</h1></div>`;
```

4: Log Error to Console

Log the error to the console using `console.error`, providing detailed information about the error that occurred.

In summary, the `handleError` function is a generic error-handling mechanism that can be used across various parts of the application and helps us avoid writing the same logic numerous times.

By hiding the loading spinner and displaying a user-friendly error message within the specified container, it ensures a consistent user experience when an error occurs. Additionally, logging the error to the console provides valuable insights for developers to understand and fix the underlying issue.

Step 13: Attach Event Listener and Initialize Movies

1: Attach Change Event Listener to Movie Dropdown

Get the HTML element with the id 'movieSelect' and attach a 'change' event listener to it. This listener is set to execute an asynchronous function whenever the selected value in the dropdown changes.

2: Define the Event Handler Function

Within the event handler function:

- **Pass the `event` object** Pass the `event` object to the callback function.
- **Retrieve Selected URL:** Obtain the value of the selected option from the event's target value, and store it in a constant named `selectedUrl`.

```
const selectedUrl = event.target.value;
```

- **Fetch Movie Data:** Call the previously defined `fetchMovieData` function, passing in the `selectedUrl`. This initiates the process of fetching data for the selected movie and displaying it. No need to store the result to a constant, since the `fetchMovieData` doesn't return anything.

Step 14: Initialize Movies on Page Load

Call the previously defined `initializeMovies` function. This initiates the process of fetching the initial list of movies and populating the dropdown, setting up the initial state of the application.

Testing

Thoroughly test the application to ensure all parts are working as expected. During the work on this exercise, constantly use `console.log` to see the data received from the APIs, and to see if you get correct responses. Also pay attention to how the application behaves when switching between different movies.

Important Note: Characters Data Loading Time

Loading the characters' data in this exercise may take a significant amount of time due to the multiple HTTP requests made for each character's details. This cumulative time leads to a noticeable delay, especially with a large number of characters.

Extra Challenge in Appendix

For those interested in optimizing this aspect, the appendix of this exercise offers an extra challenge. It involves using placeholders, the `IntersectionObserver`, and the `observer.observe` method to enhance loading efficiency. By accepting this challenge, you can improve both user experience and application performance.

Conclusion

This exercise aims to challenge you by combining two APIs and handling more complex asynchronous operations. By completing this exercise, you should gain more confidence in working with external data sources, handling user interactions, and structuring your JavaScript code in a clean and maintainable way.