Overall the project took me 6 hours to complete(only the coding parts), I took sometime to read the documentation of the API and also cypress before implementing them.
Below I have described the steps I took:

1. Fetching the Data:

- The `fetchSpaceXLaunches` function is defined. It takes two parameters: `page` (the current page number) and `pageSize` (the number of launches to load per page).
- Inside the function, a `try...catch` block is used to handle errors that may occur during the fetch operation.
- A `Headers` object is created to specify the request headers. In this case, it sets the `Content-Type` header to `application/json`.
- The `body` object is created to define the query and options for the SpaceX API request. It specifies the `limit` (number of launches per page) and `offset` (the offset based on the current page) in the options.
- The `fetch` function is used to send a POST request to the SpaceX API endpoint (`https://api.spacexdata.com/v4/launches/query`). The request includes the `Content-Type` header and the JSON-encoded `body`.
- The response from the API is awaited, and the JSON data is extracted from it using `response.json()`. The response contains an array of launch documents, and `data.docs` is returned.
- If any error occurs during the fetch operation, it is caught in the `catch` block, and an error message is logged to the console.

Problems:

- **Duplicate Keys in Response Data**: Duplicate keys in response data are more common with REST APIs that return JSON objects. In GraphQL, the response structure is typically defined in the query itself, so duplicate keys should not be an issue unless they are explicitly requested in the query.
- **JSON Header Issues**: The code sets the **Content-Type** header to **application/json** when sending the POST request to the SpaceX API. This header is typically used to specify the format of the data being sent in the request body. In this case, it indicates that the request body contains JSON data. It should not cause issues with the API request itself.
- **Error Handling**: The code includes error handling using a **try...catch** block to catch and log errors that may occur during the fetch operation. This is a good practice to ensure that errors are properly handled and don't cause the application to crash.
- **Pagination**: When fetching large datasets, pagination is important to avoid loading all data at once. The code includes **limit** and **offset** options in the request body to implement pagination. Ensure that the pagination logic matches the API's pagination requirements.
- **Testing**: Consider testing the API requests and responses thoroughly to ensure they work as expected. Used tools like Postman.
- **API Documentation**: Always refer to the API's official documentation to understand its requirements, endpoints, and any specific quirks or limitations.

Displaying Data:

- **Displaying Launch Details**:
    1. Launch details are fetched from the SpaceX API and stored in the **details** field of the **SpaceXLaunch** object.
    2. To display launch details, the **truncateText** function is used to limit the length of the text to 100 characters. This is done to keep the displayed information concise and prevent long text from overflowing the card.
    3. The truncated launch details are shown under the "Details" section for each launch card.
- **Displaying Launch Date**:
    1. The launch date is obtained from the **date_utc** field of the **SpaceXLaunch** object.
    2. JavaScript's **toLocaleDateString** method is used to format the date in a human-readable format.
    3. The formatted launch date is displayed under the "Launch Date" section for each launch card.
- **Displaying Launch Status**:
    1. The launch status is determined by the **success** field of the **SpaceXLaunch** object.
    2. If **success** is **true**, the launch is considered "Successful," and if **false**, it's considered "Failed."
    3. The launch status is displayed under the "Status" section for each launch card.
- **Displaying Images**:
    1. Images of the launches are fetched from the SpaceX API's **links.patch.small** field.
    2. The **CardMedia** component from the Material-UI library is used to display the images in a card.
    3. The **image** prop of **CardMedia** is set to the URL of the launch image, and CSS is applied to ensure that the image fits the card and is displayed correctly.
- **Filtering and Searching**:
    1. The web application provides filtering and searching capabilities.
    2. Users can filter launches by selecting one of the filter options: "All Launches," "Successful Launches," or "Failed Launches." The filter criteria are applied to the displayed launches based on their success status.
    3. Users can also search for launches by entering a search query in the search input field. The search query is used to filter launches based on their names and details.
- **Pagination and Loading Indicator**:
    1. To improve user experience, the application fetches launches in pages using pagination.
    2. A loading indicator (a circular progress spinner) is displayed when new launches are being loaded from the API.

3. An "End of launches" message is displayed when the user reaches the end of the available launches.

- **Responsiveness**:
    1. The layout is responsive, with cards displaying launch information in a grid format.
    2. The number of cards displayed per row adjusts based on the screen size (e.g., 1 card per row on mobile, 2 on tablet, and 3 on larger screens).

Search and Filtering:

- **Search Functionality**:
    1. Search functionality allows users to enter a search query in the provided search input field.
    2. The search query is stored in the **searchQuery** state variable using the **setSearchQuery** function.
    3. As the user types in the search input, the **onChange** event handler is triggered, updating the **searchQuery** state in real-time.
    4. The **searchQuery** is converted to lowercase to perform case-insensitive searches.
    5. Launches are filtered based on whether their names or details contain the search query in lowercase.
    6. The filtered launches are stored in the **filteredLaunches** array, which is displayed on the page.
- **Filtering by Launch Success Status**:
    1. Users can filter launches based on their success status, including "All Launches," "Successful Launches," and "Failed Launches."
    2. A dropdown **Select** component from the Material-UI library is used for this purpose.
    3. The selected filter criteria are stored in the **filterCriteria** state variable using the **setFilterCriteria** function.
    4. When the user selects a filter option from the dropdown, the **onChange** event handler is triggered, updating the **filterCriteria** state.
    5. Launches are filtered based on the selected filter criteria:
        1. If "All Launches" is selected, no additional filtering is applied.
        2. If "Successful Launches" is selected, only launches with a **success** property set to **true** are displayed.
        3. If "Failed Launches" is selected, only launches with a **success** property set to **false** are displayed.
- **Displaying Filtered Launches**:
    1. The **filteredLaunches** array contains launches that match the user's search query and the selected filter criteria.
    2. These filtered launches are displayed on the web page as cards, each containing launch details, launch date, launch status, and an image.
    3. The number of cards displayed per row in the grid adjusts based on the screen size to provide a responsive layout.

- **Updating Filters and Search Criteria**:
    1. Whenever the user interacts with the search input field or selects a filter option, the relevant state variables (**searchQuery** and **filterCriteria**) are updated.
    2. In response to these updates, the **useEffect** hooks are triggered to reload the launches based on the new filter and search criteria.
    3. The **loadLaunches** function is called with the updated criteria, and the filtered launches are updated accordingly.
- **Pagination and Load More**:
    1. The application implements pagination to fetch launches in batches from the SpaceX API.
    2. As the user scrolls down the page and reaches the bottom, new launches are loaded and appended to the existing list of launches.
    3. This allows users to explore additional launches without needing to load all launches at once.

JWT Authentication:

Create an Authentication Service

- Develop an authentication service that includes functions for login, logout, and checking if the user is authenticated.
- When a user logs in, store a JWT (JSON Web Token) in local storage. This token represents the user's session.
- When the user logs out, remove the JWT from local storage.
- Implement a function to check if the user is authenticated by verifying the presence of the JWT in local storage.

Configure Routes

- Define the routes in React application, including routes for protected content that should only be accessible to authenticated users.
- Create a protected route component that checks if the user is authenticated.
- For example, one can use the **react-router-dom** library to set up and manage routes.

Create Authentication Logic

- In components, implement authentication logic. For instance:
    1. Display a "Login" button if the user is not authenticated.
    2. Display a "Logout" button if the user is authenticated.
    3. Redirect unauthenticated users to a login page or the home page when they try to access protected routes.

Implement Login and Logout Actions

- When the user clicks the "Login" button, call the login function from the authentication service.
- Inside the login function, store a JWT token in local storage. In a real application, this token would be obtained from an authentication server during the login process.
- When the user clicks the "Logout" button, call the logout function from the authentication service.
- Inside the logout function, remove the JWT token from local storage.

Infinite Scrolling (handleIntersection, bottomOfPageRef): Infinite scrolling is implemented using the Intersection Observer API. When the user scrolls to the bottom of the page (or a specified threshold), the **handleIntersection** function is triggered. It checks if conditions for loading more data are met (not loading, not reached the last page, and not already at the bottom). If conditions are met, it increments the **page** state to load more data.