W5 PRACTICE

React + Axios Integration

At the end of this practice, you can

- ✓ **Apply** APIs integrations between your existing APIs with ReactJS
- ✓ **Understanding** APIs integration with using different request methods

Get ready before this practice!

- ✓ **Read** the following documents to understand the nature of Express.js: https://expressjs.com/

- ✓ **Read** the following documents to know more about Axios: https://axios-http.com/docs/intro

- ✓ **Read** the following documents to know more about Axios with React: https://www.digitalocean.com/community/tutorials/react-axios-react

How to submit this practice?

- ✓ Once finished, push your **code to GITHUB**
- ✓ Join the **URL of your GITHUB** repository on LMS

*Student's name: **Someth Phay***

IDTB100019

SE G2 Gen10 Year2 Term3

Lect. Kim Ang Kheang

S2 – React + Axios Integration

Reflection Questions

GitHub Repo:

https://github.com/PhaySometh/Y2_Term3_W5_S2-React_plus_Axios_Integration.git

EXERCISE 1 *– APIs integration for Articles*

- *For this exercise you will start with a* ***START CODE (EX-1)***

Goals

Q1 – Display All Articles

Task: Build a component that fetches and displays all articles.

API Used: `GET /articles`

Axios REQUEST in ReactJS

```useEffect(() => {```
``` axios.get('http://localhost:5000/articles')```
```   .then(res => setArticles(res.data))```
```   .catch(err => console.error(err));```
```}, []);```

Q2 – View Article Details

Task: Show full content of an article using dynamic route `/articles/:id`.

API Used: `GET /articles/:id`
Use `useParams()` from React Router.

Axios REQUEST in ReactJS
``axios.get(`http://localhost:5000/articles/${id}`)``

Q3 – Add New Article

Task: Create a form to add a new article.

API Used: `POST /articles`
Fields: `title`, `content`, `journalistId`, `categoryId`

Axios POST REQUEST to create an article
`axios.post('http://localhost:5000/articles', articleData)`

Q4 – Update Existing Article

Task: Prefill a form and update an existing article.

API Used: `PUT /articles/:id`

Axios POST REQUEST to update an article
``axios.put(`http://localhost:5000/articles/${id}`, updatedData)``

Reflective Questions

1. How did using `useEffect()` and `axios` help separate logic from UI?

Using useEffect() along with Axios helped separate concerns between fetching data and rendering UI. By placing the API calls inside useEffect, the logic for data fetching was kept outside of the JSX, which made the component easier to understand and maintain. This way, the API integration became declarative — data is fetched on mount without interfering with the UI code. It also made testing and debugging more manageable because I could isolate fetch logic from rendering logic.

2. What state challenges did you face while managing form input and API response?

The main challenge I faced was synchronizing the form inputs with state changes. For instance, when adding or updating articles, I had to ensure all input fields were correctly controlled using useState. Another challenge was managing form resets after a successful POST/PUT request and making sure the article list updated immediately. Sometimes, the state didn't reflect changes instantly, so I had to manually trigger a re-fetch or update the local state after submitting.

3. How does REST structure help React developers write cleaner frontend code?

The RESTful API structure provides clear and consistent endpoints, such as /articles, /articles/:id, which makes data access predictable. This helped me organize frontend code to match each API interaction — fetching all articles, fetching one article, adding, or updating one. It also allowed reusable components like ArticleForm or ArticleList to work independently by just passing the right props or calling specific API endpoints. It encourages a modular and scalable frontend structure.

EXERCISE 2 – API Integration: Filter Articles by Journalist & Category

- *For this exercise you will start with a **START CODE (EX-2)***

Goals

- ✓ Use dropdown selections (<select>) to trigger filtered API requests
- ✓ Understand sub-resource routes in REST (/journalists/:id/articles, etc.)
- ✓ Dynamically render filtered content
- ✓ Manage dependent state and conditional rendering in React
- ✓ Practice clean UI/UX with form inputs

Context

Your frontend application should now allow users to filter articles based on:

- **A selected journalist**
- **A selected category**

Each filter option will call a different API route and update the displayed list of articles accordingly.
You will not filter client-side, but instead make API requests based on selection.

Q1 – Fetch All Journalists & Categories for Select Inputs

Task: On component mount, fetch journalists and categories to populate two dropdown menus.

APIs Used:

- `GET /journalists`
- `GET /categories`

Axios GET REQUEST to fetch an journalists and categories

```
useEffect(() => {
  axios.get('http://localhost:5000/journalists').then(res =>
setJournalists(res.data));
  axios.get('http://localhost:5000/categories').then(res =>
setCategories(res.data));
}, []);
```

Q2 – Filter Articles by Selected Journalist

Task: When a journalist is selected from the dropdown, fetch articles written by that journalist.

API Used:

- `GET /journalists/:id/articles`

Axios GET REQUEST to fetch articles

```
axios.get(`http://localhost:5000/journalists/${selectedJournalistId}/article
s`)
  .then(res => setArticles(res.data));
```

Q3 – Filter Articles by Selected Category

Task: When a category is selected from the dropdown, fetch articles belonging to that category.

API Used:

```
Axios GET REQUEST to fetch articles
axios.get(`http://localhost:5000/journalists/${selectedJournalistId}/article
s`)
  .then(res => setArticles(res.data));
```

Q4 – (Bonus) Combine Filters (Frontend Side)

Task: Apply **combined filtering** on the frontend (e.g., articles that match both selected journalist and category) after fetching results from one of the filters.

- Adjust the backend to support combined filtering through query parameters if needed:

```
// Example: GET /articles?journalistId=1&categoryId=2
```

REFLECTIVE QUESTIONS

- *For this part, submit it in separate PDF files*

1. **How do sub-resource routes like `/journalists/:id/articles` help in designing a clear and organized API?**
   Explain the benefits of using nested routes for resource relationships in REST APIs.

Sub-resource routes like /journalists/:id/articles clearly express the relationship between resources — for example, that articles belong to a specific journalist. This makes the API intuitive and logically organized. It reduces ambiguity and helps avoid redundant filtering logic on the client side. From a frontend developer's perspective, it improves readability and simplifies the way I make API calls for nested data (e.g., journalist's articles).

2. **What challenges did you face when managing multiple filter states (journalist and category) in React?**
   Discuss how you handled state updates and conditional rendering when multiple inputs affect the same output.

Managing both journalist and category filters meant I had to maintain multiple pieces of state that could both influence the article list. A key challenge was deciding when to trigger which API call and ensuring they didn't overwrite each other. I used useState for both selections and had to carefully structure the useEffect and handler functions to apply filters properly. Conditional rendering of articles based on combined states required extra care to avoid displaying incorrect or outdated results.

3. **What would be the advantages and disadvantages of handling the filtering entirely on the frontend versus using API-based filtering?**
   Compare client-side filtering (in-memory) vs. server-side filtering (via query or nested routes).

Client-side filtering (frontend) is fast and responsive since all data is already loaded, but it doesn't scale well for large datasets. It also exposes all the data to the client, which may not be secure or efficient.

Server-side filtering (backend) ensures that only relevant data is fetched, making it efficient and secure. However, it requires more API design considerations and introduces slight delays due to network latency. In this exercise, using API-based filtering gave me more accurate control and kept the UI snappy even with many articles.

4. **If you needed to allow filtering by both journalist and category at the same time on the backend, how would you modify the API structure?**
   Think about adding query parameters or designing a new combined route.

To support combined filtering, I would modify the backend to accept query parameters. For example, a route like /articles?journalistId=1&categoryId=2 would allow the server to apply both filters before returning the result. On the backend, I'd use conditionals to check for the existence of these query params and apply the appropriate WHERE clauses in the database query.

5. **How did this exercise help you understand the interaction between React state, form controls, and RESTful API data?**
   Reflect on how state changes triggered data fetching and influenced the UI rendering logic.

This exercise taught me that React state is the bridge between the user interface and backend data. When a form control changes, the state updates, which then

```
triggers an API call. That new data is then stored in state and used to re-render
the UI. Understanding this interaction gave me better control over user experience
and data consistency. I now see how important state management is when working with
dynamic content and external APIs.
```