API-Lab Architecture: A Visual Simulation of Client-Server Communication

"Archimede" High School - Treviglio

Author: Zanotti Giovanni

Abstract

In modern software development, the communication between Client and Server through APIs (Application Programming Interfaces) is the backbone of the web. However, for students and junior developers, visualizing this invisible exchange of data can be abstract and difficult to grasp. In our solution, we develop **API-Lab v2.1**, a web-based simulation environment built with React and Vite. This tool allows users to visualize the request-response cycle in real-time without the need for a complex backend infrastructure. By simulating network traffic directly within the browser using the Broadcast Channel API, the project demonstrates the mechanics of RESTful architecture, HTTP methods, and status codes in an interactive way.

Summary

1.0 Objective

Data exchange on the web relies heavily on strings and JSON objects sent over HTTP protocols. Understanding how a Client packages a request and how a Server processes it is fundamental for any Full Stack developer. The objective of this project is to develop a **Single Page Application (SPA)** that simulates a complete Client-Server architecture. Instead of relying on external tools like Postman or opaque backend logs, this project places the user in control of both sides. The application aims to:

1. Allow the user to compose HTTP requests (GET, POST, DELETE) as a Client.
2. Visualize these requests arriving at a "Server" dashboard in real-time.
3. Enable the user to act as the Server logic, manually selecting response codes (200, 404, 500) to see how the Client reacts.
4. Bridge the gap between frontend simulation and backend reality by generating dynamic Python (FastAPI) code snippets corresponding to the actions taken.

2.0 Tools and Technologies

To develop this project, a modern frontend stack has been selected to ensure performance, maintainability, and a smooth user experience.

2.1 Visual Studio Code

For development, we used Visual Studio Code. Its ecosystem of extensions (ESLint, Prettier, Tailwind CSS IntelliSense) makes it the industry standard for JavaScript and React development.

2.2 React & Vite

React is the JavaScript library used for building the user interface. It allows us to create reusable components (like the `Sidebar`, `FastAPIBox`, etc.) and manage the application state efficiently. We paired React with **Vite**, a build tool that provides a significantly faster development experience compared to legacy tools like Webpack, utilizing native ES modules in the browser.

2.2.1 Tailwind CSS

Styling is handled via Tailwind CSS, a utility-first CSS framework. This allowed for rapid UI development using predefined classes (e.g., `flex`, `bg-charcoal-blue-900`, `text-old-gold-500`) directly in the JSX code, ensuring a consistent design system (colors, spacing, typography) defined in the `index.css` file.

2.2.2 Broadcast Channel API

A critical technological choice for this project was the use of the **Broadcast Channel API**. Since the application runs entirely in the browser (client-side) but needs to simulate two distinct entities (Client and Server) often running in different tabs or windows, we needed a communication bus. The Broadcast Channel API allows different browsing contexts (tabs, windows, iframes) of the same origin to send and receive messages. This replaces the need for actual TCP/IP Sockets or HTTP requests for the scope of this simulation, providing instant, zero-latency communication.

2.3 JSON Formatting

JavaScript Object Notation is the universal standard for data exchange in RESTful APIs. It is lightweight, text-based, and language-independent. In this project, JSON plays a crucial role in data visualization. Since the data exchanged over the network is often complex, we utilized formatting techniques to make it readable for the user. In the `Client.jsx` component, we implemented a "Pretty Print" view for the server response using the native JavaScript method: `JSON.stringify(response.body, null, 2)` This function takes the raw data object and converts it into a formatted string with 2-space indentation, allowing the student to clearly see the hierarchy of keys and values, simulating exactly how a developer would inspect a real API response.

3.0 Client - Server Architecture

In a real-world scenario, the Client-Server architecture separates the user interface (Client) from the data management (Server).

- **The Client** initiates communication by sending a **Request**. This includes a Method (GET, POST, etc.), an Endpoint (URL), and optional Body data.

- **The Server** listens for requests, processes the logic, and returns a **Response**. This includes a Status Code (e.g., 200 OK, 404 Not Found) and data (usually JSON).

In **API-Lab**, we emulate this behavior. The `Client.jsx` page acts as the requester, and the `Server.jsx` page acts as the listener. Instead of traveling through the internet, data travels through the browser's internal memory via the BroadcastChannel.

4.0 Discussion

4.1 Simulation Logic

The core of the project relies on the synchronization between the Client and Server components. This is achieved through a custom hook named `useApiChannel.js`.

The Hook Implementation:

```
export const useApiChannel = (role) => {
  const [lastMessage, setLastMessage] = useState(null);
  const channelRef = useRef(null);

  useEffect(() => {
    channelRef.current = new BroadcastChannel('api_lab_channel');
    channelRef.current.onmessage = (event) => {
      // Logic to filter messages based on role (CLIENT vs SERVER)
      if (role === 'SERVER' && event.data.type === 'REQUEST') {
        setLastMessage(event.data);
      } else if (role === 'CLIENT' && event.data.type === 'RESPONSE') {
        setLastMessage(event.data);
      }
    };
    // Cleanup
    return () => channelRef.current.close();
  }, [role]);
  // ...
};
```

This code creates a "virtual wire".

1. **Sending a Request:** In `Client.jsx`, when the user clicks "Send Request", the app bundles the `method`, `endpoint`, and `timestamp` into an object and uses `sendMessage()` to broadcast it.

2. **Receiving the Request:** The `Server.jsx` component, using the hook with the `'SERVER'` role, detects the message. It updates its `logs` state and displays the "Pending Request" card with an animation.
3. **Sending a Response:** The user (acting as the Server) clicks a status button (e.g., 200 OK). `Server.jsx` broadcasts a response object back on the same channel.
4. **Closing the Loop:** `Client.jsx` receives the response and updates the UI to show the result (Green for success, Red for errors).

4.2 The "FastAPI Box"

One of the unique educational features discussed in this solution is the `FastAPIBox.jsx` component located in the Server dashboard. While the simulation runs on React, the goal is to teach backend concepts. This component takes the *current active request* and dynamically generates the Python code that would be required to handle it using the **FastAPI** library.

Dynamic Code Generation: The component extracts the method and endpoint from the request object:

```
const method = request?.method || 'GET';
const cleanEndpoint = endpoint.split('?')[0];

const codeSnippet = `
@app.${method.toLowerCase()}("${cleanEndpoint}")
async def handler():
    return {"status": "success"}
`;
```

This string is then rendered with basic syntax highlighting (using Tailwind colors to simulate an IDE). This feature bridges the gap between the abstract simulation and the concrete implementation code a developer would write in a real-world scenario (specifically utilizing the `pydantic` and `fastapi` libraries mentioned in the standard curriculum).

4.3 Internationalization

To make the tool accessible, we implemented a `LanguageContext`. This uses a Context Provider to wrap the application, supplying a translation function `t()` to all components. The translations are stored in a JSON-like structure (`utils/translations.js`), allowing instant switching between English and Italian without reloading the page.

5.0 Conclusions

The **API-Lab v2.1** project successfully demonstrates how modern frontend technologies can be used to simulate complex architectural concepts. By using **React** for the UI and the **Broadcast Channel API** for communication, we created a robust environment where students can experiment with HTTP requests safely.

The project fulfills its objectives by:

- Providing a visual interface for "invisible" network operations.
- Offering an immediate feedback loop between Client and Server actions.
- Integrating educational snippets (FastAPI code) directly into the workflow.

Future developments could include replacing the Broadcast Channel with a real connection to a Python backend, transforming the simulation into a fully functional Full Stack application.

6.0 Project Links

- GitHub Repository: https://github.com/GiZano/api-presentation
- Web Presentation (for author personal portfolio use): https://giovanni-zanotti.is-a.dev/Pages/Works/api-presentation/

7.0 Bibliography

- **React Documentation:** https://react.dev/
- **Vite Build Tool:** https://vitejs.dev/
- **MDN Web Docs - Broadcast Channel API:** https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API
- **Tailwind CSS Documentation:** https://tailwindcss.com/
- **FastAPI Documentation:** https://fastapi.tiangolo.com/