

Scripting in Web Development An Examination of Client-Side, Server-Side, and Node.js

by Bradley Watkins

Submission date: 14-May-2025 07:00PM (UTC+0100)

Submission ID: 259266703

File name: 58782_Bradley_Watkins_Scripting_in_Web_Development_An_Examination_of_Client-Side_Server-Side_and_Node.js_661685_195079717.pdf (113.89K)

Word count: 4496

Character count: 28000

Scripting in Web Development: An Examination of Client-Side, Server-Side, and Node.js

Bradley Watkins 2406413

May 2025

Contents

1	Introduction	3
2	Scripting in Web Development	3
2.1	Static Site Generation	3
2.2	Client-Side Scripting	4
2.3	Server-Side Rendering	6
2.4	Comparison Summary	7
3	Node.js Runtime	8
3.1	Benefits	8
3.2	Drawbacks	9
4	Conclusion	9
5	Code Documentation	10
5.1	Introduction	10
5.2	Design Overview	10
5.3	Backend Communucations	12
5.4	Authentication Flow	14
5.5	Testing	17
5.6	Conclusion	19
A	Appendix: User Guide	20

1 Introduction

Web development is a multifaceted discipline that relies heavily on the use of scripting languages to create dynamic and interactive content on the web. The following sections will delve into key aspects of web development, primarily focusing on client-side and server-side scripting solutions. The Node.js runtime environment will be investigated, presenting the potential advantages and drawbacks of its usage in modern web applications. Finally, a discussion around the code solution provided as part of this module will be presented.

2 Scripting in Web Development

Scripting languages are fundamental to modern web development, enabling the creation of responsive and dynamic applications. Both client-side and server-side scripting play crucial, yet distinct, roles in this process. This section will compare and contrast these two scripting paradigms, focusing on their functionality, notably database connectivity, and the requirements of their respective runtime environments.

2.1 Static Site Generation

Static site generation is a technique in which HTML pages are pre-built during the build process, often using templates and data sources. These pre-rendered HTML files are then served directly to the user without the need for real-time processing or injection of data. While static sites inherently lack dynamic content generation, they can still incorporate client-side JavaScript to provide interactive elements. The primary advantage of static site generation is exceptional performance and security due to the lack of database interaction and server-side code execution during runtime. However, updates to content typically require a rebuild and redeploy of the entire site [1].

The following snippet demonstrates a static web page that utilises client-side scripting for user interactivity, toggling the header text on button click:

Figure 1: Static Site Generation providing basic client-side interactivity

```
<body>
  <h1 id="interactive-content">Hello, world!</h1>
  <button id="toggle-button">Click Me!</button>
  <script>
    $(document).ready(function() {
      $("#toggle-button").click(function() {
        let currentText = $("#interactive-content").text();
        if (currentText === "Hello, world!") {
          $("#interactive-content").text("Goodbye, world!");
        } else {
          $("#interactive-content").text("Hello, world!");
        }
      });
    });
  </script>
</body>
```

2.2 Client-Side Scripting

Client-side scripting executes within the user's web browser, enabling dynamic content manipulation directly on the client. Typically, the browser receives an initial bundle comprising static HTML and associated JavaScript [2]. Upon loading, the JavaScript code interacts with the Document Object Model (DOM) to inject dynamic data into the HTML. This dynamic data is often retrieved from external services via asynchronous AJAX calls [3].

The asynchronous nature of browser input and output prevents blocking of user interactions during data retrieval, contributing to a more fluid user experience. However, network latency or slow service responses can still negatively impact the user if content loading is delayed. Performance studies indicate that client-side rendering can lead to poorer initial page load times compared to server-side rendering, though it can offer superior runtime performance for subsequent interactions [1]. This increase in overall performance is due to

direct manipulation of the DOM, reducing server round trips. The initial load can be slower, as the browser must download and parse the JavaScript before finally making necessary AJAX calls to populate the page. Delay is exacerbated in low-powered devices or with large bundles, potentially leading to user frustration.

The perceived performance is also impacted by factors like network latency and the efficiency of the JavaScript code itself. While offloading rendering to the browser can be beneficial, it introduces a dependency on the client's resources, which are highly variable and outside of the control of the developer.

Consider the following HTML snippet, illustrating client-side data fetching and injection:

Figure 2: Client-Side rendering of user greeting

```
<body>
  <h1 id="greeting">Hello World</h1>
  <script>
    $(document).ready(function () {
      $.ajax({
        url: "https://stub/users/1",
        method: "GET",
        success: function (data) {
          $("#greeting").text(`Hello, ${data.name}`);
        },
        error: function () {
          $("#greeting").text("Something went wrong!");
        }
      });
    });
  </script>
</body>
```

This example highlights the efficiency of rapidly delivering static content and the limitation of relying on client-side code execution for dynamic content. Client-side JavaScript does not have a means of direct database access, requiring intermediary API calls. This can introduce performance bottlenecks due to network

overhead. There is also a concern for browser runtime environment compatibility, where variations in browser implementations can lead to unexpected and inconsistent application behaviour.

The above snippet is rudimentary and has been simplified for brevity but provides insight into potential pitfalls in client-side scripting techniques. The snippet exposes the API endpoint `https://stub/users/1` directly in the client-side code, posing a security risk. This JavaScript bundle will be directly available to users within the browser, introducing an attack vector for malicious users to access sensitive data or launch attacks.

2.3 Server-Side Rendering

Server-side rendering also involves serving HTML and JavaScript to the client. However, it differs fundamentally from client-side scripting in that the server pre-populates the HTML with dynamic data before sending it to the browser. This pre-rendering minimises the need for client-side AJAX calls, shifting the content injection burden to the server [1].

One significant advantage of server-side rendering lies in its ability to operate outside the constraints of the browser's runtime environment. This is particularly important for ensuring cross-browser compatibility, as the server can generate HTML tailored to work in any browser and adhere to web standards [4]. Server-side scripting often allows for direct database access, enabling optimised data access and reducing network latency when compared to client-side AJAX calls.

Server-side rendering generally offers faster load times, as the browser receives fully rendered HTML. This is particularly beneficial for Search Engine Optimisation (SEO), as web crawlers can readily index the complete content [1, 2]. Real-time data updates are also more easily managed server-side, as the server can efficiently push updates to the client or serve updated pages on reload. Client-side solutions often require more complex mechanisms such as WebSockets to provide real-time updates.

Consider this server-side rendering example, mirroring the previous client-side example:

Figure 3: Server-Side rendering of user greeting

```
const express = require("express")
const database = require("./db")

const app = express()

app.get("/:id", async (req, res) => {
  const user = await database.getUserById(req.params.id)
  res.send(`

# Hello, ${user.name}</h1>`) }) app.listen(3000, () => { console.log(`Server running at http://localhost:${3000}`); })


```

This example clearly demonstrates direct database interaction within the server, reducing the number of network requests and improving efficiency. However, scalability can become a concern when many concurrent requests are received. In real-time applications many thousands of requests may be received at once, leading to massive server load and potential for slowed response times [5, 6]. There would typically be more than one database call in a service request, and there may be external API calls. This introduction of concurrency can lead to optimisation issues and massive increases in complexity. Caching, well-provisioned services, and optimisation strategies can alleviate these concerns, but they are important to consider [7].

2.4 Comparison Summary

Web developers have several architectural choices in regard to the rendering technique they employ. Static site generation offers peak performance for content that doesn't require frequent dynamic updates. Client-side scripting offers rich interactivity and offloads rendering to the browser. This can lead to increased performance for subsequent interactions but slower initial loads. Server-side rendering emphasises faster initial loads, improved SEO, optimised database queries, and looser coupling to the browser's runtime environment [1, 2]. It

can, however, lead to increased server load and reduced overall response times [1].

A hybrid approach is likely to offer the greatest success when there is not an obvious use case for one paradigm over the others. Static content generation and server-side rendering of dynamic data should be maximised to increase SEO gains. Client-side scripting can then be utilised for retrieval of data that the server does not directly control and to provide rich interactive experiences to users.

3 Node.js Runtime

Node.js is a JavaScript runtime based on Google's V8 browser runtime. It is event-driven, asynchronous, and non-blocking, making it well-suited for scalable, real-time applications [3]. The framework supports long-running server processes and uses callbacks and event handlers to manage asynchronous tasks efficiently. In their own words, HTTP is a first-class citizen in the Node ecosystem, providing low-latency network streaming with minimal effort [8].

Bundled with Node is the Node Package Manager (NPM). NPM serves as a repository for JavaScript libraries that can be used in any project. These libraries enable rapid development through code sharing, allowing developers to prioritise business requirements and spend less time on reinventing the wheel [9]. Currently in use by more than seventeen million developers globally, NPM has created an enormous ecosystem of support for JavaScript solutions [10].

The following sections will evaluate the benefits and drawbacks of the Node ecosystem.

3.1 Benefits

Node.js offers several advantages in modern web development. Its single-language environment allows developers to use JavaScript for both client-side and server-side scripting, streamlining workflows and reducing context-switching between languages. This also provides all of the available libraries and packages from NPM in both front-end and back-end environments.

Node.js is highly performant due to its non-blocking I/O model. It has been shown to handle up to 30,000 concurrent connections without failures, outperforming other runtimes and frameworks in memory usage and scalability [7]. When combined with Nginx, Node.js excels in serving both static and dynamic content [5].

The event-driven nature of Node.js allows developers to build loosely coupled services, putting separation of concerns first [6]. This is increasingly important in modern web architectures, especially in microservices. Serverless solutions often make use of Node.js as a means of creating compact and highly cohesive modules [11].

When deploying services to the cloud, it is critical to consider their integration with containerisation solutions. Node.js has been shown to be effective for developing containerised back-end solutions through services such as Docker. Fastify, a Node.js framework, excelled especially in this area, providing the greatest performance in regard to latency and network throughput. This was exemplified when Fastify was paired with Object Relational Mapping solutions, providing efficient database access [12].

3.2 Drawbacks

Despite its strengths and widespread usage in industry, Node.js has some limitations that must be taken into consideration. By design, Node.js is a single-threaded architecture that leverages smart task delegation to perform asynchronous actions. This can, however, lead to performance bottlenecks in CPU-intensive tasks. Studies have shown that requests begin to consistently fail when concurrent operations exceed thirty thousand [6]. This can be alleviated through horizontal scaling, offloading requests to other services. However, this requires scalable architecture to be implemented and is still prone to hotspots.

JavaScript also introduces a development antipattern known as "callback hell". Node.js utilises callback-based programming paradigms, through JavaScript, to provide seamless asynchronous functionality. Callback programming can introduce massive levels of complexity, making code difficult to read, understand, and maintain. Modern constructs like promises and `async/await` aim to mitigate this issue. Developers must remain vigilant in order to prevent complexity creep in their code [3, 11].

The NPM ecosystem introduces concerns and antipatterns that developers are known to ignore. Currently, more than 55% of NPM packages are considered trivial, requiring minimal effort from developers to implement manually. Trivial packages have been identified in core functions within many industry projects [9]. Development efforts are negatively impacted by breaking changes introduced in trivial packages. Trivial packages are often abandoned, leading to security concerns. These brittle dependencies can introduce long-term maintenance concerns, threatening to increase costs. Developers should be mindful of their package usage, avoiding them when the long-term tradeoff would be problematic [13].

4 Conclusion

Node.js presents a compelling runtime environment for modern web development, offering significant advantages in scalability, performance, and the unification of the development stack via JavaScript. The non-blocking I/O model and the vast ecosystem of the Node Package Manager (NPM) have empowered developers to build and

share efficient solutions rapidly. However, the single-threaded nature of Node.js necessitates careful consideration of CPU-intensive tasks, and the callback-heavy asynchronous patterns can introduce complexity. The sheer size and decentralised nature of the NPM ecosystem require developers to exercise diligence in managing dependencies. Despite these drawbacks, Node.js remains a powerful and widely adopted technology, playing a crucial role in the architecture of many web applications and services.

5 Code Documentation

The following section will discuss the front-end portion of code provided for the combined assessment for modules ACCA7051 and ACCA7052.

5.1 Introduction

The provided application uses a modern web development stack centred around a React Router front-end architecture. This choice enables the rapid creation of fluid, single-page application experiences. The application's front end is served to the browser using the Vite development server, utilising the speed and efficiency of the Node.js runtime. To ensure type safety during development, the application front end is written in TypeScript, providing static typing and improved developer tooling. The user interface is styled using Tailwind CSS, a utility-first framework that provides rapid and consistent styling throughout the application. Unit tests have been developed using the Vitest testing framework, chosen for its speed and compatibility with the React ecosystem.

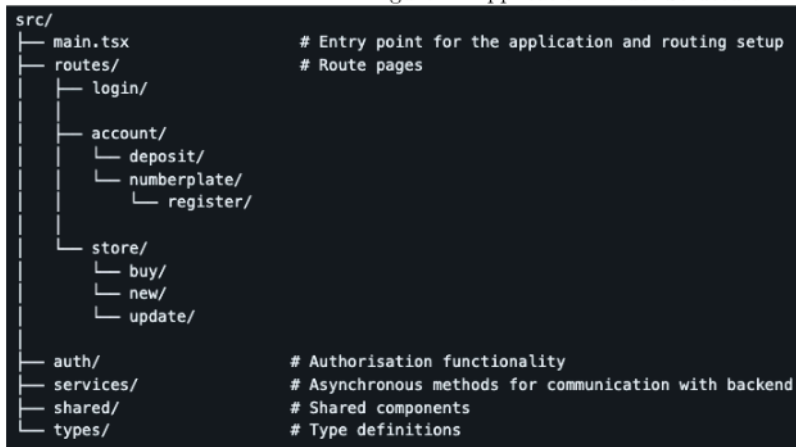
5.2 Design Overview

Route mapping within the application is handled through React Router and can be found within **main.tsx**. This is where components are assigned to routes, and restricted routes are assigned expected roles.

The application employs a folder-to-route mapping strategy, where the folder structure under `src/routes` mirrors the application's route hierarchy. For instance, the `/store`, `/store/buy/:id`, and `/store/new` routes have corresponding folders (store and subfolders within store) containing the respective React components. This design choice helps to improve code organisation and maintainability. Through creating a link between routes and component files, it becomes significantly easier to locate, understand, and modify route-related logic. When working on a specific feature such as the store, all relevant files are grouped, reducing the need to search the codebase.

While this convention is not enforced within the application, it may facilitate scalability and reduce development time when introducing new routes. There is also potential, however, that in larger applications this folder structure could become cumbersome and may lead to misunderstandings. This is largely due to the lack of enforcement, meaning that any mistakes are likely to be missed. With many routes and developers, mistakes may become likely as folder nesting becomes more prevalent. However, on a project of this scale it was not a concern, and the increased modularity and separation of concerns proved worth any trade-offs.

Figure 4: Application Folder Structure



To maximise code reuse and maintain a consistent user interface, key components are placed within the top-level layout file. This ensures that elements like the header, containing site navigation and a dynamic login/logout button, are mounted and rendered across all pages. Dynamic content within these shared components is managed using React hooks. This approach significantly reduces code duplication and centralises route navigation logic.

Component design within the application heavily prioritises reusability, adhering to the Don't Repeat Yourself (DRY) principle. Through effectively leveraging props and callback functions, a number of general-purpose components have been created. For example, a flexible **ErrorCode** component is used across multiple error routes, displaying a status code passed as a prop and reducing the need to create explicit pages. Similarly, a reusable form **Error** component accepts a dynamic error message and the id of the input element that it is connected to, eliminating redundant error message logic. The implementation of the **Error** component is illustrated in the following snippet.

Figure 5: Dynamic React Component For Error Messages

```
interface Props {
  error: string;
  inputId: string;
}

export default function Error({ error, inputId }: Props) {
  const focusInput = () => {
    document.getElementById(inputId)?.focus();
  }

  return (
    <div
      className="font-bold text-red-500 hover:text-red-300 hover:cursor-pointer"
      onClick={focusInput}
    >
      {error}
    </div>
  )
}
```

5.3 Backend Communications

Asynchronous communication within the application is managed by TypeScript services, which centralise data persistence, error handling, and authentication logic. A key aspect of this design is the utilisation of an OpenAPI contract shared between the front end and back end. Leveraging the **openapi-typescript** npm dependency, type definitions are automatically generated from this contract's YAML as **paths**. These paths provide comprehensive type information for all available endpoints and their potential responses, enabling robust and type-safe handling of various scenarios.

AJAX requests are dispatched using the native Fetch API. For secure communication, most requests include

an authorisation header containing a JSON Web Token (JWT). This token contains user-specific data, eliminating the need to send redundant user context data in the request body. API endpoint URLs are dynamically configured via environment variables accessed through the **import.meta.env** interface. This allows for seamless deployment across different environments, such as local development and production, without requiring code modification.

The following code snippet illustrates AJAX request usage and response handling, demonstrating the type safety provided by the **paths** generated from the OpenAPI contract.

Figure 6: AJAX Communications via Fetch

```
export async function fetchAccount(): Promise<Account | null> {
  const token = authService.getToken();

  if (!token) {
    throw new Error('Not authenticated');
  }

  const res = await fetch(`${import.meta.env.VITE_SERVICE_URL}/customer`, {
    method: 'GET',
    headers: {
      'Authorization': 'Bearer ' + token
    }
  });

  authService.handleNoAuthStatus(res.status);

  if (res.status === 404) {
    return null;
  }

  return await res.json()
    as paths['/customer']['get']['responses']['200']['content']['application/json'];
}
```

5.4 Authentication Flow

Authentication and authorisation within the application are implemented using the OAuth2 protocol and JSON Web Token (JWT) management. To minimise the complexities and security risks associated with managing sensitive user credentials, a dedicated third-party identity provider, KeyCloak, is employed for user authentica-

tion.

The authentication flow involves the user providing credentials to KeyCloak, which, upon successful validation, returns a JWT. This token securely encapsulates user identity information, assigned roles within the application, and an expiration time. A critical aspect of maintaining security is ensuring users must re-authenticate upon token expiry. This is handled on the front end by a custom React hook that listens for cookie change events. Before any potentially authorised action, the JWT's expiration is checked. If expired, the user is logged out and a cookie change event is triggered, preventing further unauthorised actions.

Authorisation on the backend is enforced by verifying the presence and validity of the JWT in request headers. On the front end, access control is determined by user roles, encoded within the JWT. This is achieved through a two-step process. The JWT is first decoded using the **jwt-decode** library, which parses the token and maps its claims to a defined type, making role information and user details accessible. Then, a **ProtectedRoute** higher-order component wraps protected routes and components, ensuring that the currently authenticated user possesses the necessary roles before rendering them. The following code snippet demonstrates the **ProtectedRoute** component's role-based access control logic.

Figure 7: ProtectedRoute Component Prevents Unauthorised Access to Routes

```
interface Props {  
  children: React.ReactNode;  
  role: string;  
}  
  
export default function ProtectedRoute({ children, role }: Props) {  
  const isAuthenticated = useAuth();  
  
  const location = useLocation();  
  return (  
    !isAuthenticated ? (  
      <Navigate to="/login" state={{ from: location.pathname }} />  
    ) : (  
      authService.hasRole(role) ? (  
        <>{children}</>  
      ) : (  
        <Navigate to="/NotAuthorised" state={{ from: location.pathname }} />  
      )  
    )  
  )  
}
```

The code snippet below demonstrates how the **ProtectedRoute** component is used to secure both administrator and user-specific pages.

Figure 8: Restricted Route Components are Wrapped in a ProtectedRoute to Prevent Unauthorised Access

```
<Route path='/store/update/:numberplate' element={  
  <ProtectedRoute role='numberplate-ui-admin'>  
    <EditNumberplate />  
  </ProtectedRoute>  
} />  
  
<Route path='/account' element={  
  <ProtectedRoute role='numberplate-ui-user'>  
    <UserAccount />  
  </ProtectedRoute>  
} />
```

5.5 Testing

The application's testing strategy is built upon the Vitest testing framework. Vitest's comprehensive mocking features are particularly valuable for isolating and testing asynchronous communication patterns, such as those involving AJAX requests. A key aspect of testing backend communications is to align test cases with the response structures defined in the OpenAPI contract. By explicitly testing against the contract, we can ensure that the application gracefully handles all expected responses from the backend, leading to more reliable and predictable behaviour.

The snippets below showcase tests for the **fetchAccount** method, which is responsible for communicating with the backend service. These snippets demonstrate testing for successful responses, handling HTTP status codes, and error management. For clarity and brevity, the code snippets have been shortened.

Figure 9: Happy Path Testing of fetchAccount

```
it('should fetch account data successfully', async () => {  
  const mockAccount: Account = { balance: 100 };  
  mockFetch.mockResolvedValueOnce({  
    ok: true,  
    status: 200,  
    json: vi.fn().mockResolvedValue(mockAccount),  
  } as unknown as Response);  
  
  const account = await customerService.fetchAccount();  
  expect(authService.getToken).toHaveBeenCalledTimes(1);  
  expect(mockFetch).toHaveBeenCalledWith(  
    `${import.meta.env.VITE_SERVICE_URL}/customer`, {  
      method: 'GET',  
      headers: {  
        'Authorization': 'Bearer testToken',  
      },  
    },  
  );  
  expect(authService.handleNoAuthStatus).toHaveBeenCalledWith(200);  
  expect(account).toEqual(mockAccount);  
});
```

Figure 10: Testing Not Found Response Scenario

```
it('should return null if account is not found (404)', async () => {  
  mockFetch.mockResolvedValueOnce({  
    ok: false,  
    status: 404,  
  } as Response);  
  
  const account = await customerService.fetchAccount();  
  expect(account).toBeNull();  
  expect(authService.handleNoAuthStatus).toHaveBeenCalledWith(404);  
});
```

Figure 11: Testing Error Handling When User is not Authenticated

```
it('should throw an error if not authenticated', async () => {  
  (authService.getToken as ReturnType<typeof vi.fn>)  
    .mockReturnValue(undefined);  
  await expect(customerService.fetchAccount())  
    .rejects.toThrow('Not authenticated');  
  expect(authService.getToken).toHaveBeenCalledTimes(1);  
  expect(mockFetch).not.toHaveBeenCalled();  
  expect(authService.handleNoAuthStatus).not.toHaveBeenCalled();  
});
```

5.6 Conclusion

This documentation provides an overview of the front-end architecture and key design considerations of the provided web application. The application makes use of a modern Node.js stack, including React Router, TypeScript, and Tailwind CSS, to create a robust and maintainable user interface. The component-based

architecture, adherence to DRY principles, and comprehensive type definitions generated from the OpenAPI contract promote code reuse, modularity, and type safety. The implementation of OAuth2 for authentication and role-based access control ensures secure access to application features. Finally, the testing strategy, aligned with the OpenAPI contract, validates the application's reliability and resilience.

A Appendix: User Guide

The following README.md has been extracted from the application. This provides a comprehensive guide on running the applications through Docker and using the application once running.

```
# User Guide: Running the l7-personalised-numberplates Application
```

```
This guide outlines the necessary steps to run the l7-personalised-numberplates
application.
```

```
**Prerequisites**
```

```
* Docker
* Maven
```

```
**Application Setup**
```

```
* Open your terminal or command line.
* Navigate to the 'numberplate_service' directory:
```

```
'''bash
cd numberplate_service
'''
```

```
* Clean and install the numberplate service using Maven:
```

```
'''bash
```

```
mvn clean install
'''
```

* Navigate back to the project's root directory:

```
'''bash
cd ..
'''
```

* Start the numberplate service using Docker Compose (and build the image if necessary):

```
'''bash
docker compose up --build
'''
```

User Guide: l7-personalised-numberplates Application

This guide explains how to use the l7-personalised-numberplates application.

****Accessing the Application****

1. Open your web browser.
2. Navigate to the following URL: 'http://localhost:5173'
3. You will be presented with the application's home screen.

****Navigation****

* ****Store:**** To browse available number plates, click on the link or button that navigates to '/store'.

* **Login:** To log in to your user account, click on the link or button that navigates to `/login`.

User Accounts

The application provides two default user accounts for demonstration purposes:

* **Basic User:**

- * Username: `basicuser`
- * Password: `password`

* **Admin User:**

- * Username: `adminuser`
- * Password: `password`

User Functionality (Post-Login)

Upon successful login, your account balance will be displayed.

* **Account Details:** To manage your account, click on the link or button that navigates to `/account`. From here, you can:

- * **Deposit Funds:** Navigate to `/account/deposit` to add funds to your account (Bank Number can be any value).
- * **View Owned Number Plates:** Navigate to `/account/numberplate` to see the number plates you currently own.
- * **Register New Number Plate:** Navigate to `/account/numberplate/register` to register a new number plate.

* **Purchasing Number Plates:**

- * Navigate back to the store (`/store`) to browse and purchase number plates.
- * Ensure that your account balance is sufficient to cover the cost of the number plate before purchasing.

****Admin User Functionality (Accessed via Store)****

The admin user has additional privileges to manage the number plates in the store. These functions are accessed via buttons or links available on the store page ('/store').

* ****Add New Number Plate:**** An admin user can add new number plates to the store from '/store/new'.

* ****Edit Number Plate:**** An admin user can modify the details of existing number plates in the store from '/store/edit/:id'.

References

- [1] R. Hanafi, A. Haq, and N. Agustin, "Comparison of web page rendering methods based on next.js framework using page loading time test," *Teknika*, vol. 13, no. 1, pp. 102–108, Mar. 2024.
- [2] T. Fadhilah Iskandar, M. Lubis, T. Fabrianti Kusumasari, and A. Ridho Lubis, "Comparison between client-side and server-side rendering in the web development," *IOP Conference Series: Materials Science and Engineering*, vol. 801, no. 1, p. 012136, May 2020.
- [3] S. Tilkov and S. Vinoski, "Node.js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, Nov. 2010.
- [4] D. V. Kornienko, S. V. Mishina, S. V. Shcherbatykh, and M. O. Melnikov, "Principles of securing restful api web services developed with python frameworks," *Journal of Physics: Conference Series*, vol. 2094, no. 3, p. 032016, Nov. 2021.
- [5] I. K. Chaniotis, K.-I. D. Kyriakou, and N. D. Tselikas, "Is node.js a viable option for building modern web applications? a performance evaluation study," *Computing*, vol. 97, no. 10, pp. 1023–1044, Mar. 2014.
- [6] K.-I. D. Kyriakou and N. D. Tselikas, "Complementing javascript in high-performance node.js and web applications with rust and webassembly," *Electronics*, vol. 11, no. 19, p. 3217, Oct. 2022.

-
- [7] S. D. Meglio and L. Libero Lucio Starace, "Evaluating performance and resource consumption of rest frameworks and execution environments: Insights and guidelines for developers and companies," *IEEE Access*, vol. 12, pp. 161 649–161 669, 2024.
- [8] OpenJS Foundation. About Node.js. Accessed on: 2025-05-7. [Online]. Available: <https://nodejs.org/en/about>
- [9] M. A. R. Chowdhury, R. Abdalkareem, E. Shihab, and B. Adams, "On the untriviality of trivial packages: An empirical study of npm javascript packages," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2695–2708, Aug. 2022.
- [10] npmjs.com. About npm. Accessed on: 2025-05-7. [Online]. Available: <https://docs.npmjs.com/about-npm>
- [11] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. microservice architecture: A performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022.
- [12] I. P. A. E. Pratama and I. M. S. Raharja, "Node.js performance benchmarking and analysis at virtualbox, docker, and podman environment using node-bench method," *JOIV : International Journal on Informatics Visualization*, vol. 7, no. 4, Dec. 2023.
- [13] D. Kong, J. Liu, L. Bao, and D. Lo, "Toward better comprehension of breaking changes in the npm ecosystem," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–23, Apr. 2025.

Scripting in Web Development An Examination of Client-Side, Server-Side, and Node.js

GRADEMARK REPORT

FINAL GRADE

90/100

GENERAL COMMENTS

Server/Client Code (25): 22

Node.js (25): 23

SW Doc (50): 45

FEEDBACK

=====

Very good discussion of client and server side code but I would have liked a little more on the execution procoess.

Very good discussion of node.js.

Could have provided a little more discussion of the code, but this very good.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24
