



Web Development Report



Jeffery Aideyan-Ohonba

Table of Contents

1. Comparing Client-Side and Server-Side Scripting.....	2
1.1 Examples of Server-Side and Client-Side Scripting Languages.....	3
1.1.1 Example Implementation of Client-Side Scripting.	4
1.1.2 Example Implementation of Server-Side Scripting	4
2. Understanding Node.js and How it Works	5
2.1 Benefits and Drawbacks of Using Node.js	6
3. Client-Side Application Design and Implementation	7
3.1 Overview of the Design of the Code	7
3.2 Main or Most Interesting Aspects of the Code	8
3.3 Evidence of Testing.....	9
4. Running the Application (User Manual)	10
4.1 Prerequisites.....	10
4.2 Running the Full Stack	10
4.3 Logging In.....	10
4.4 Keycloak & Realm Setup	11
4.5 PostgreSQL Database	11
5. Application Demo Video Link.....	11
6. References.....	12

1. Comparing Client-Side and Server-Side Scripting

When building a dynamic website, the scripts that make pages interactive or responsive can run in two main places: either on the client-side, in the user's web browser, or on the server-side, on the web server before the page even reaches the browser. Each approach has its strengths and weaknesses, and often, the best results come from combining both methods in a smart way [1].

Server-side scripting comes with several benefits. Since the processing happens on the server before the page is sent to the user, the code remains hidden, which helps protect logic and sensitive operations. It also allows access to local files or directories on the server, which is crucial for things like database operations or reading server-side data. Another benefit is that server-side scripts work independently of the user's browser, so there's no need to worry about browser compatibility.

However, one drawback of this approach is that it puts more strain on the server. Each time a user requests a page, the server has to parse and generate the content before sending it back, which can slow down performance, especially if many users are visiting the site at the same time [1].

Client-side scripting, on the other hand, shifts the workload to the user's browser. This lightens the load on the server since each user's device processes the code independently. It's particularly effective for things like form validation or updating content dynamically without refreshing the page.

But this method has its limitations. Client-side code is visible to users through their browser's developer tools, making it less secure. Also, it can't access files, directories, or databases on the server. So, while it's useful for improving responsiveness and enhancing user interaction, it can't handle sensitive or data-driven tasks on its own [1].

To get the best of both worlds, many modern websites combine server-side and client-side scripting, allowing them to handle secure tasks on the backend while delivering fast, interactive experiences on the frontend [1].

1.1 Examples of Server-Side and Client-Side Scripting Languages

A widely used example of a server-side scripting language is PHP. Originally developed in 1994, PHP started out as a set of simple scripts but quickly evolved into a powerful and widely adopted tool for building dynamic web pages. By version 3, it had grown into a full-fledged programming language and continued to develop through contributions from a growing community.

PHP runs entirely on the server, meaning any PHP code is processed before the resulting HTML is sent to the user's browser. This keeps the underlying code hidden and offloads computational tasks from the user's device. PHP also acts as a parsing engine, which means it examines the PHP file, executes the required tasks, and returns the result to the web server for delivery. As an interpreted language, it runs on demand without needing to be compiled in advance [2].

On the client-side, JavaScript is the most prominent scripting language. It runs within the browser itself, which means it doesn't need a web server to function. This allows developers to build responsive interfaces that can react instantly to user actions, such as checking input in a form before it's submitted or changing visual elements without reloading the page.

Because JavaScript runs locally on the user's machine, it can help reduce communication with the server and improve performance. However, it depends on the capabilities of the user's device, and if JavaScript is disabled (though rare today), it won't work at all [2].

JavaScript and PHP share a lot of similarities in their syntax and structure, even though they operate in different environments. JavaScript is object-oriented, loosely typed, and often used in combination with other web technologies to enhance functionality and interactivity [2].

1.1.1 Example Implementation of Client-Side Scripting.

The code snippet below shows an example implementation of Client-Side Scripting. The form uses JavaScript to validate user input before it is submitted. The plate number is checked against a regex pattern for format, banned words are filtered, and price is validated to ensure it's a positive number. These checks prevent invalid data from being sent to the server, demonstrating a key benefit of client-side scripting.

```
const handleSubmit = async e => {
  e.preventDefault();
  setError(''); // Clear previous errors

  const containsBannedWord = bannedWords.some(word =>
    | formData.plateNumber.toUpperCase().includes(word)
  );

  const isValidPlatePattern = platePatterns.some(pattern =>
    | pattern.test(formData.plateNumber)
  );

  if (!isValidPlatePattern) {
    setFieldErrors({
      | plateNumber:
      | "Plate number must match a valid UK registration format: 'AB12CDE', 'A123BCD', 'ABC123D', or 'ABC123'. Only letters and numbers are allowed, no spaces.",
    });
    return;
  }

  if (formData.price <= 0.0) {
    setFieldErrors({ price: 'Price must be greater than 0.' });
    return;
  }

  if (containsBannedWord) {
    setFieldErrors({
      | plateNumber: 'Plate number contains inappropriate content.',
    });
    return;
  }
}
```

Figure 1: JavaScript client-side form validation for a license plate submission form.

1.1.2 Example Implementation of Server-Side Scripting

The following example illustrates server-side scripting by validating incoming data and returning appropriate HTTP responses based on the backend logic.

```
@PostMapping
public ResponseEntity<> createLicensePlate(@Valid @RequestBody LicensePlate plate) {
  try{
    LicensePlate createdLicensePlate = licensePlateService.saveLicensePlate(plate);
    return ResponseEntity.status(HttpStatus.CREATED).body(createdLicensePlate);
  }catch (IllegalArgumentException e){
    //Custom validation exception for banned words, handled directly in the controller
    return ResponseEntity.status(HttpStatus.BAD_REQUEST).body(Map.of("plateNumber", e.getMessage()));
  }
}
```

Figure 2: A Spring Boot controller method that handles a POST request to create a license plate.

2. Understanding Node.js and How it Works

Node.js is a JavaScript runtime environment that lets developers run JavaScript outside of the browser, which is a big shift from how JavaScript was traditionally used just for client-side scripting. It was first released in 2009 by Ryan Dahl and is built on Google's Chrome V8 engine, the same engine that powers JavaScript in the Chrome browser. What makes Node.js different is that it brings JavaScript into the server-side world, putting it alongside languages like PHP or Python in terms of what it can do [3].

One of the things that sets Node.js apart is its event-driven, non-blocking I/O model. In simpler terms, this means it doesn't pause or wait around when handling tasks like reading files or querying databases. Instead, it handles multiple things at once using events and callbacks, which makes it really good for building fast and responsive applications [3]. This is especially useful for servers that need to handle lots of users at the same time.

Even though Node.js runs on a single thread, it still handles many concurrent connections efficiently thanks to its event loop and asynchronous approach. This makes it lightweight and scalable without needing to create a new thread for every user or request. As stated in [3], this design is part of what helps Node.js perform so well under heavy loads.

Another plus is that Node.js works across different operating systems like Windows, Linux, and macOS. This cross-platform support gives developers a lot of flexibility in terms of where and how they want to run their applications.

Node.js also supports modular programming through the CommonJS module system. This makes it easier to organise and reuse code by splitting things into smaller, manageable pieces, a big help when building large projects or working in teams [3].

2.1 Benefits and Drawbacks of Using Node.js

One of the biggest strengths of Node.js is its speed, especially when it comes to handling server connections. Thanks to its event-driven, non-blocking nature, Node.js can handle multiple requests at the same time without waiting for one to finish before starting another. This means it doesn't rely on creating multiple threads like other languages do, which helps reduce overhead and makes it more scalable [4]. Compared to PHP, for example, Node.js is faster for applications with many users because PHP uses a synchronous approach, while Node.js is asynchronous and event-driven [4].

Another good thing about Node.js is that it allows developers to use just one language: JavaScript, for both the frontend and backend. This is especially helpful for full-stack development, since you don't need to learn and juggle several different languages or technologies. Everything from browser interactions to server-side logic can be handled with JavaScript, and even data is stored using JSON, which fits naturally into the workflow [4].

Node.js also benefits from JavaScript being one of the most widely used languages in web development. For new developers, this means there's often less of a learning curve since they're likely already familiar with the basics of JavaScript on the frontend [4].

That said, Node.js isn't without its challenges. Because it uses asynchronous programming, developers need to be careful when structuring their code. Too many nested callbacks, often referred to as "callback hell", can make the code hard to read, debug, and maintain. If not handled well, the event loop can slow down when CPU-heavy tasks are involved, affecting the overall performance [4].

It's also not the easiest platform for beginners. JavaScript itself can be a bit messy and difficult to master, especially when it comes to asynchronous functions and advanced features like promises or `async/await`. On top of that, the Node Package Manager (NPM), while powerful, often lacks proper documentation and reusability across projects, which can make life harder for less experienced developers [4].

Lastly, despite growing popularity, there's still a shortage of skilled Node.js developers in the job market. Many organisations and developers are more comfortable sticking

with familiar technologies like PHP or .NET, and there's still a lack of awareness around the asynchronous programming model Node.js is built on [4].

3. Client-Side Application Design and Implementation

3.1 Overview of the Design of the Code

This project is built using a React frontend and a Spring Boot backend that talk to each other through REST APIs. Authentication is handled using Keycloak, which supports login, logout, and role-based access control.

There's a clear separation between each part of the app:

Routing & Navigation: All routes are defined in one place using React Router. I wrapped restricted routes in a *ProtectedRoute* component, which checks if the user is logged in and has the right role.

Authentication: When the frontend app loads, it does a **silent check** using Keycloak's check-sso mode to see if the user is already logged in (based on an existing session). If they are, it sets the *isAuthenticated* state right away. If not, the user can log in manually by clicking the login button in the nav bar. Once authenticated, Keycloak also helps determine what role the user has. I've set things up so different pages are only accessible depending on the user's role, either *CUSTOMER* or *ADMIN*.

API Calls: Axios is used for all HTTP requests. I've split the logic into two main files, one for license plate-related endpoints and another for purchases. Each API call gets the user's token from Keycloak and attaches it to the request headers.

Folder Structure: I kept both the frontend and backend under the same root folder, which makes it easier to manage.

Overall, the goal was to keep the design modular and readable, with security and role-based access.

3.2 Main or Most Interesting Aspects of the Code

One of the more interesting parts of the project was adding logic to help admins create valid license plates, and stop them from creating anything dodgy or incorrectly formatted. I wanted to make sure the plates followed actual UK styles and didn't contain inappropriate words.

To handle that, I defined a list of banned words, and then a few regex patterns that match valid UK plate formats. When the admin tries to submit the form, the code checks whether the plate number fits one of the patterns and doesn't contain any banned content.

```
const bannedWords = ['BAD', 'OFFENSIVE', 'XXX'];
const platePatterns = [
  /^[A-Z]{2}[0-9]{2}[A-Z]{3}$/ ,
  /^[A-Z][0-9]{1,3}[A-Z]{3}$/ ,
  /^[A-Z]{3}[0-9]{1,3}[A-Z]$ / ,
  /^[A-Z]{3}[0-9]{3}$/ ,
];
```

If something's wrong, like if the plate is BA12XXX or doesn't match any known pattern, the form throws a validation error and doesn't let the request go through. This validation runs on both the client side and the backend, so users get instant feedback before anything even hits the backend, and the backend will act as the last layer of defence.

I also added a small info button “(?)” next to the plate input field that opens a modal explaining all the valid plate formats. This helps guide admins and cuts down on confusion.

Another thing I added was partial search functionality so users can type just a part of plate and still get relevant results. This makes the app a lot more user-friendly. It works for both plate browsing and purchase history.

Last thing worth mentioning – the dynamic navigation bar is fully role-aware. When a user logs in, the app checks what role they have and shows the correct links. Customers see “My Purchases,” and admins see “Create Plate” or “View Sold Plates”. It's a small touch, but makes the UI feel more tailored.

3.3 Evidence of Testing

I wrote unit tests for the frontend API utility functions that interact with the backend. The tests focus on making sure the correct HTTP requests are made and that authorization headers are included when needed, especially for operations that require the user to be logged in through Keycloak.

For the license plate API, I tested fetching all plates, getting a plate by ID, and creating a new plate. For the purchase API, I tested making a purchase, retrieving all purchases, getting purchases for a specific user, and performing a plate number search.

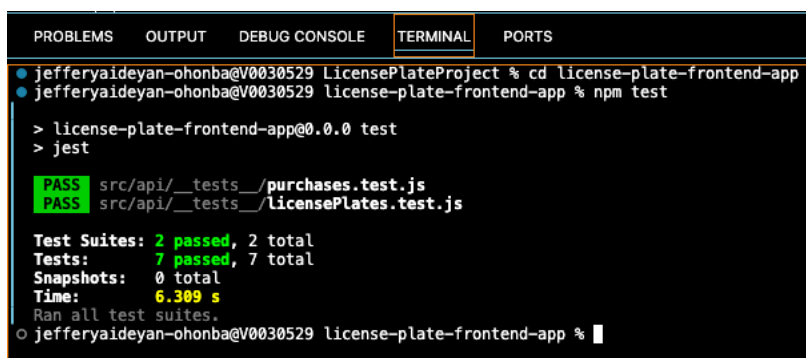
Each test uses mocked versions of axios and the *getAuthHeaders* function to simulate the expected behaviour without needing a real backend or token.

Here's an example expectation from the purchase tests:

```
expect(getAuthHeaders).toHaveBeenCalled();
expect(axios.post).toHaveBeenCalledWith(
  'http://localhost:8080/license-plates/${purchaseId}/purchases',
  {},
  { headers: mockHeaders }
);
expect(result.data).toEqual({ success: true });
});
```

This confirms the app is sending the right request structure and including the auth token.

Below is a screenshot showing all the tests passing:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
jefferyaideyan-ohonba@V0030529 LicensePlateProject % cd license-plate-frontend-app
jefferyaideyan-ohonba@V0030529 license-plate-frontend-app % npm test

> license-plate-frontend-app@0.0.0 test
> jest

PASS src/api/_tests_/purchases.test.js
PASS src/api/_tests_/LicensePlates.test.js

Test Suites: 2 passed, 2 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        6.309 s
Ran all test suites.
jefferyaideyan-ohonba@V0030529 license-plate-frontend-app %
```

4. Running the Application (User Manual)

This section provides setup instructions to run the full application stack using Docker Compose, including test credentials.

4.1 Prerequisites

- Docker and Docker Compose must be installed on your system.
- No other services should be occupying the following ports:
 - 5173 for the frontend (Vite dev server)
 - 8080 for the Spring Boot backend
 - 8090 for Keycloak
 - 5000 for PostgreSQL

4.2 Running the Full Stack

1. Open a terminal and navigate to the root directory of the project:

LICENSEPLATEPROJECT/

2. Run the following command from the root:

docker compose up --build

This command will:

- Build and start the frontend (Vite + React)
- Start the Spring Boot backend service
- Start the PostgreSQL database
- Start Keycloak and automatically import the preconfigured realm, users, and roles

3. Once the stack is running:

- Open your browser and go to: <http://localhost:5173>

4.3 Logging In

Two test user accounts have been preconfigured in Keycloak for testing purposes:

- Admin User
 - Username: adminuser
 - Password: password

- Customer User
 - Username: customeruser
 - Password: password

You can log in via the “Login” button in the navigation bar on the frontend. After logging in, the app will show different functionality depending on the user's role.

4.4 Keycloak & Realm Setup

Keycloak is included as a service in docker-compose.yml. It loads a custom realm from the `/keycloak/LicensePlateManagementAuthRealm-export.json` file on container startup using the `--import-realm` flag.

No additional Keycloak configuration is required; the realm contains:

- ADMIN and CUSTOMER roles
- Mapped user-role relationships
- Client configuration for the frontend
- Realm settings for token handling

4.5 PostgreSQL Database

The PostgreSQL database is also containerised and configured with:

- Database: VehicleLicensePlateMgmt
- Username: postgres
- Password: password

The Spring Boot backend automatically connects to the database using environment variables defined in the docker-compose.yml.

5. Application Demo Video Link

<https://youtu.be/GJvtgEwCO0Q>

6. References

- [1] Anonymous “Scripting: Server-side vs. client-side,” *Active Server Developer's Journal*, vol. 2, (9), pp. 15-16, 1998. Available: <https://www.proquest.com/trade-journals/scripting-server-side-vs-client/docview/206870773/se-2>.
- [2] M. Mendez, *The Missing Link: An Introduction to Web Development and Programming*, Open SUNY Textbooks, 2014, ch. 23, “Server-Side and Client-Side Scripting”. [Online]. Available: <https://courses.lumenlearning.com/suny-the-missing-link-an-introduction-to-web-development-and-programming/chapter/chapter-23-server-side-and-client-side-scripting/>. [Accessed: May 6, 2025].
- [3] B. Wu, Y. Cao and X. Liu, “Research and Analysis of Commonly Used Node.js Framework,” *2024 Prognostics and System Health Management Conference (PHM)*, Stockholm, Sweden, 2024, pp. 104-107, doi: 10.1109/PHM61473.2024.00027.
- [4] B. Basumatary and N. Agnihotri, “Benefits and Challenges of Using NodeJS,” *International Journal of Innovative Research in Computer Science and Technology*, vol. 10, no. 3, pp. 67–70, 2022, doi: 10.55524/ijircst.2022.10.3.13.