

ACCA7051 Web development

Parjal Rai 2420643

by Pj Rai

Submission date: 15-May-2025 07:37PM (UTC+0100)

Submission ID: 258438317

File name: 58810_Pj_Rai_ACCA7051_Web_development_Parjal_Rai_2420643_661685_1889407287.docx
(4.13M)

Word count: 2579

Character count: 15654

ACCA7051 Web development
Parjal Rai
2420643

Table of Contents

1. Introduction	3
2. Client-side vs server-side scripting	3
3. Node JS.....	5
3.1 Node.js in the project.....	6
4. AJAX in the project.....	7
5. User Manual.....	8
6. Overview of the design of the code.....	9
7. Discussion of the key components	12
7.1 Main aspect	12
7.2 Interesting aspect	13
8. Evidence of Testing.....	15
9. Conclusion.....	20

1. Introduction

Web development began with the creation of the first webpage by Tim Berners-Lee in 1990, using technologies such as HTML (HyperText Markup Language), HTTP (HyperText Transfer Protocol), and the WorldWideWeb browser [1]. At this stage, HTML defined the static structure of content, and HTTP enabled communication between servers and browsers—there was no client-side or server-side scripting.

In the mid-1990s, the introduction of JavaScript enabled client-side scripting, allowing code to run directly in the user's browser. This made web pages more interactive without needing constant communication with the server. At the same time, the limitations of static HTML led to the adoption of server-side scripting. The Common Gateway Interface (CGI) allowed web servers to execute external programs and generate dynamic content. Building on CGI, Rasmus Lerdorf developed PHP (Personal Home Page), which later evolved into a server-embedded language to address CGI performance issues [2].

Initially, JavaScript was confined to client-side development, while back-end tasks were handled by languages like PHP and Java. However, the introduction of Node.js—built on the V8 JavaScript engine—enabled developers to write server-side code in JavaScript as well. This innovation unified front-end and back-end development under a single language, simplifying full-stack workflows [7].

This report explores topics such as the differences between client-side and server-side scripting, an investigation into Node.js, and documentation of a web application developed for vehicle plate registration and purchase. It also includes an overview of the code's design, discussion of key implementation features, and evidence of testing.

2. Client-side vs server-side scripting

Client-side scripting involves executing scripts directly in the user's browser (such as with Chrome's V8 engine) rather than on the web server. This allows the code to run on the user's device, dynamically updating the webpage without needing to reload the page or make frequent requests to the server. By executing locally, client-side scripting offers a faster, more responsive user experience with less reliance on server interactions. However, this approach also means there is no direct database connectivity from the client side. Instead, the client interacts with the backend server through APIs to retrieve or manipulate data stored in the database (Figure 1) [2].

Other key functionalities of client-side scripting include DOM manipulation, dynamic UI updates (such as form validation and dynamic menus), and asynchronous requests using AJAX (Figure 1). JavaScript is the most commonly used language for client-side scripting, though HTML and CSS handle the structure and styling of web pages [3].

```
import axiosInstance from './axiosInstance';

export const getVehiclePlatesByInput = async(input, page = 0, size = 5) =>
  await axiosInstance.get(`/vehiclePlates/reg/${input}`, {
    params: {
      page,
      size
    }
  });
```

Figure 1: API request in a React app to communicate with the backend server and retrieve vehicle plates that match the input.

Server-side scripting involves running code on the server, not in the browser. When a user sends a request from the browser — such as submitting a form or calling an API (Figure 1) — the server processes that request. The server runs scripts within a runtime environment (such as Apache Tomcat, Nginx as a reverse proxy, or an application server) to handle backend logic, data processing (for example, querying a database like PostgreSQL), and enforcing business rules. After processing the request, the server sends a response back to the browser. For instance, in Figure 2, a JSON response is served to the browser, although the response could also be in the form of HTML or XML. The browser can then use this response to render dynamic web content [4].

Compared to client-side scripting, server-side scripting generally offers greater security because the code and sensitive operations (such as authentication, authorisation, and database interactions) remain hidden from the user. However, because server-side processing requires communication between the browser and the server, it can be slower than client-side scripting, which runs instantly within the user's browser. Nevertheless, server-side scripting is essential for operations involving CRUD (Create, Read, Update, Delete), database connectivity, authentication, and enforcing complex business rules that the client alone cannot securely or reliably handle [1].

```
@GetMapping("/{v1/vehiclePlates/{id}")  ⚠ Parjal Rai
public ResponseEntity<VehiclePlateDTO> getVehiclePlateById(@PathVariable(name="id") int id) {
    return ResponseEntity.ok(vehiclePlateDTOService.findVehiclePlateById(id));
}
```

Figure 2: REST API sending a JSON response to the client.

3. Node JS

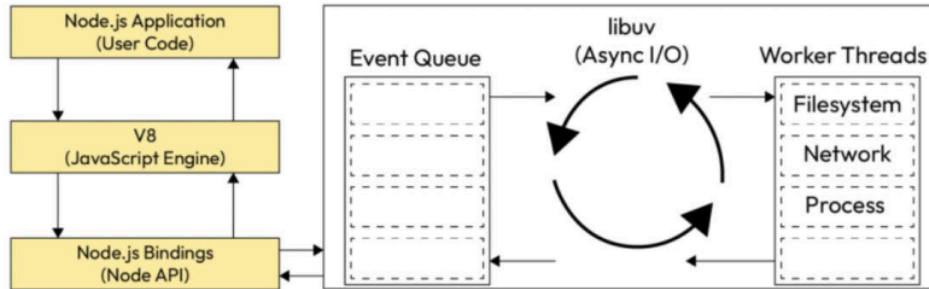


Figure 3: Diagram illustrating the relationship between user code, the Node.js API and libuv [9].

The architecture of Node.js is such that the code is written in JavaScript and executed by the V8 engine, which is embedded within Node.js. However, the underlying Node.js bindings and the libuv library are implemented in C/C++, which contributes to its high performance. Libuv manages I/O operations and utilizes a thread pool to handle blocking tasks efficiently [9].

A crucial component of the Node.js architecture is the event loop, which enables asynchronous, non-blocking I/O operations. This is a core feature of Node.js, allowing it to handle tasks like file reading, database queries, and HTTP requests without blocking the main thread—even though it runs on a single-threaded architecture [9, 10]. This unique event loop mechanism leverages callback functions to efficiently manage and process tasks. As a result, on a server with 8GB memory, Node JS can handle more than 40,000 user connections. In contrast, in traditional server-side languages like PHP Java or .NET, the same server can only handle about 4000 user connections [10].

The non-blocking I/O model, powered by the asynchronous event loop, allows the server to process multiple requests concurrently using very few threads. Rather than focusing on thread management, Node.js prioritises utilising the system's computational capabilities to efficiently handle tasks, reducing overhead and increasing performance. This event-driven, non-blocking architecture contributes to faster execution, quicker development, and greater scalability [11].

While Node.js excels at handling multiple I/O operations through its asynchronous, single-threaded model, this approach also has some drawbacks. Because a single thread is responsible for managing many concurrent operations, the application must be carefully structured. If too many requests are processed at once, it can become resource-intensive and lead to slowdowns, as Node.js may consume significant CPU resources without the benefit of true parallel execution.

Another common issue arises from poorly written asynchronous code—especially when deeply nested callbacks are involved. This so-called "callback hell" can make the code difficult to read, debug, and maintain, as each step in the chain often depends on the output of the previous one.

To address limitations like these, modern versions of Node.js now support multithreading through the worker threads module (Fig 3). This feature allows CPU-intensive JavaScript tasks to run in separate threads, keeping the main event loop responsive and improving performance for computationally heavy operations [11].

3.1 Node.js in the project

```
JS server.js > app.get('*') callback
1 // server.js
2 import express from 'express';
3 import path from 'path';
4 import { fileURLToPath } from 'url';
5
6 const __filename = fileURLToPath(import.meta.url);
7 const __dirname = path.dirname(__filename);
8
9 const app = express();
10 const PORT = 3000;
11
12 // Serve static files from 'dist'
13 app.use(express.static(path.join(__dirname, 'dist')));
14
15 app.get('/', (req, res) => {
16   res.sendFile(path.join(__dirname, 'dist', 'index.html'));
17 });
18 // handle client routing
19 app.get('*', (req, res) => {
20   res.sendFile(path.join(__dirname, 'dist', 'index.html'));
21 });
22
23 app.listen(PORT, () => {
24   console.log(`Server running on http://localhost:${PORT}`);
25 });
```

Figure 4: Express Server Request Flow for Static Assets and Client-Side Routing

Node.js was used to set up a basic Express server that serves static files from the “dist” directory (Figure 4). This directory is generated by Vite, which bundles the React application. The server is configured to serve “index.html” for both the root route (/) and all other routes (*), making it suitable for single-page applications (SPAs) that rely on client-side routing. Once the static site is delivered to the browser, client-side rendering enables dynamic content updates without requiring full page reloads [12]

4. AJAX in the project

```
const axiosInstance = axios.create({
  baseURL: 'http://localhost:8080/v1',
  headers: {
    'Content-Type': 'application/json'
  }
});

axiosInstance.interceptors.request.use(async (config) => {
  let token = authStorage.getAccessToken();

  // If no token but we have refresh token, try to refresh
  if (!token && localStorage.getItem("refreshToken")) {
    try {
      const newTokens = await refreshAuthToken();
      authStorage.setAccessToken(newTokens.accessToken);
      token = newTokens.accessToken;
    } catch (refreshError) {
      authStorage.clearTokens();
      window.location.href = '/login';
      throw new axios.Cancel('Failed to refresh token - redirecting to login');
    }
  }

  // If still no token (and no refresh token), cancel and redirect
  else if (!token) {
    authStorage.clearTokens();
    window.location.href = '/login';
    throw new axios.Cancel('No auth token available - redirecting to login');
  }

  config.headers.Authorization = `Bearer ${token}`;
  return config;
});
```

Figure 5: Use of Axios Interceptors to Enhance AJAX Request Handling

This code defines a custom Axios instance to facilitate AJAX-based communication between the frontend and a backend API hosted at “http://localhost:8080/v1”. Axios, which internally uses AJAX techniques, enables asynchronous HTTP requests without reloading the page. The configured instance includes request interceptors to manage authentication tokens. Before sending a request, it checks for a valid access token; if absent, it attempts to refresh the token using a stored refresh token. If both are unavailable or the refresh fails, the user is redirected to the login page and the request is cancelled.

5. User Manual

Client-side

BuyMyPlate-Client is a client-side application built with React.js and served using a Node.js Express web server.

To run the application, ensure that Docker is running in the background. Then, open your terminal and execute the following command:

```
docker compose up
```

Once the server is running, open a web browser and visit the URL displayed in the terminal

Server-side

BuyMyPlate-Server is a server-side application built with Java and Spring Boot, managed using Maven.

To run the application with keycloak and database already setup, go to the terminal and run the command:

```
docker compose up
```

Or

```
docker compose up -d (to run the app on the background)
```

To run tests:

1. Open a terminal at the root of the project.
2. Run: “mvn install”

6. Overview of the design of the code

BuyMyPlate is an application that follows a Model-View-Controller (MVC) architecture with React handling the View layer and communicating with a Spring Boot backend, which handles the Model and Controller layer, through an intermediary Express.js web server. The App is separated into different directory which serves different functions (fig 6). There are directories for reusable React components and custom hooks which promote separation of concerns and ease of maintenance. Pages in the app act as containers that invoke these components and manage state and logic. The api directory manages data fetching via Axios and the context file manages AuthContext which is used to manage and share user authentication state across the entire application without prop drilling. Express.js servers as a lightweight web server, routing client requests and managing communication between the React frontend and Spring Boot Backend.

The backend is built with SpringBoot, where the Model layer represents entities, and the Service layer contains the business logic. The Controller layer handles HTTP requests and sends back response as JSON, and Data Transfer Objects (DTOs) are used to structure and manage data flow between frontend and backend. The Service layer interacts with repositories to fetch and manipulate data, which is stored in a PostgreSQL database, ensuring efficient and reliable data persistence. The entity and database are linked via object-relational mapping (ORM) with Hibernate. In terms of security, a central SecurityConfig class integrates Keycloak to manage authentication and authorisation across the application, protecting endpoints. The backend supports the dynamic features of the application, including data handling and user authentication, while seamlessly integrating with the frontend through RESTful APIs.

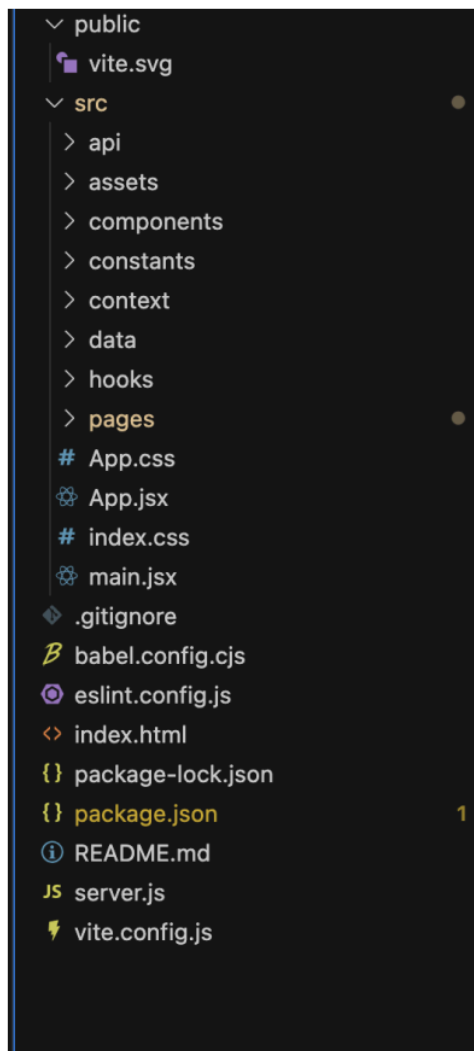


Figure 6: Project structure of the client-side app.

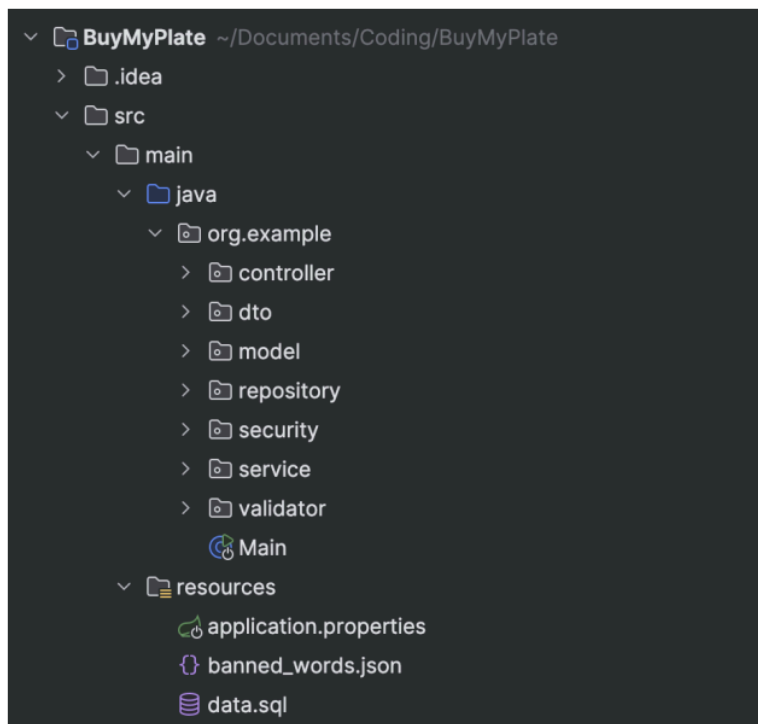


Figure 7: Project structure of the server-side app.

7. Discussion of the key components

7.1 Main aspect

```
1 import React from 'react'
2 import usePlateForm from '../hooks/usePlateForm'
3 import './assets/button.css'
4 import './assets/vehicleplate.css'
5 import PlateFormCard from './PlateFormCard'
6 import { useNavigate } from 'react-router-dom'
7
8 const PlateForm = ({
9   mode = "create",
10   initialData = null,
11   onSubmit
12 }) => {
13   const form = usePlateForm({
14     mode,
15     initialData,
16     onSubmit
17   })
18
19   const navigate = useNavigate()
20   const handleGoBack = () => {
21     navigate(-1)
22   }
23
24   return (
25     <div className="vehicle-plate">
26       <div className="card-header position-relative">
27         <h3 className="text-center mb-0">{form.formTitle}</h3>
28         <button className="btn btn-warning btn-warning position-absolute start-0 ms-3" onClick={handleGoBack} style={{ margin: '1em', top: '-20px' }}>
29           Go back
30         </button>
31       </div>
32       <div className="reg-errors">
33         {form.renderErrors()}
34       </div>
35       <div>
36         <PlateFormCard {...form} />
37       </div>
38     </div>
39   )
40 }
41
42 export default PlateForm
```

Figure 8: Reusable react component form used for creating, editing and viewing vehicle plates

The PlateForm component is designed to render a vehicle plate form that can operate in “create”, “edit”, or “view” mode, depending on the mode prop. It is reused across all pages related to creating, editing, and viewing VehiclePlates, providing a consistent and modular interface. The component leverages a custom hook, usePlateForm, to manage form state, handle submission, and supply utilities like dynamic form titles and error rendering. The PlateFormCard component receives all relevant form props and handles the actual user interface. This structure promotes reusability and clean separation of concerns by encapsulating logic within the custom hook and delegating UI responsibilities to a dedicated component.

7.2 Interesting aspect

```
src > components > ProtectedRoute.jsx > [⌕] default
1  import React from 'react'
2  import { useAuth } from '../context/AuthContext'
3  import { Navigate, useLocation, Outlet } from 'react-router-dom';
4
5  const ProtectedRoute = ({roles}) => {
6    const {isAuthenticated, isLoading, role} = useAuth();
7    const location = useLocation();
8
9    if(isLoading){
10     return <div>Loading...</div>
11   }
12
13   if(!isAuthenticated || (roles && !roles.includes(role))){
14     return <Navigate to="/login" state={{from: location}} replace/>
15   }
16
17   return <Outlet />;
18 }
19
20 export default ProtectedRoute
```

Figure 8: ProtectedRoute component used for authentication on the client-side

This React component is used to protect routes in an application based on authentication and user roles. It utilises the `useAuth` hook to access authentication status, loading state and the user's role. The `useLocation` hook is used to track the current URL. If the app is in a loading state, it displays a loading message. If the user is not authenticated or the role is not included in the `roles` prop, they are redirected to the login page. If the user passes the checks, the component renders its child components via the `Outlet` component, which is a placeholder for nested routes. This ensures that only authorised users can access protected routes based on their roles as seen on figure 9.

```

src > App.jsx > App
18  function App() {
19
20
21      return (
22          <>
23              <Navbar/>
24              <Routes>
25                  <Route path="/" element={<Home/>}/>
26
27                  <Route path="/login" element={<LoginPage/>}/>
28                  <Route path="/plate/view/:id" element={<ViewPlatePage/>}/>
29
30                  <Route path="/dashboard" element={<Home/>}/>
31
32                  <Route element={<ProtectedRoute roles={['CUSTOMER']}/>}>
33                      <Route path="/my-plates/:id" element={<ViewCustomerPlatesPage/>}/>
34                      <Route path="/my-balance/:id" element={<BalancePage/>}/>
35                      <Route path="/buy-plate/:id" element={<BuyPlatePage/>}/>
36                  </Route>
37                  <Route element={<ProtectedRoute roles={['ADMIN']}/>}>
38                      <Route path="/register-plate" element={<RegisterPlatePage/>}/>
39                      <Route path="/plate/edit/:id" element = {<EditPlatePage/>}/>
40                      <Route path="/transactions" element={<TransactionPage/>}/>
41                  </Route>
42              </Routes>
43              <ToastContainer />
44          </>
45      )
46  }

```

Figure 9: ProtectedRoute ensuring that only authorised users can access specific routes.

8. Evidence of Testing

Client-side

```
pages > Login Page > LoginPage.test.jsx > describe('Login Page') > beforeEach > beforeEach() > beforeEach()
describe('Login Page', () => {
  beforeEach(() => {
    // Mock return values
    require('react-router-dom').useNavigate.mockReturnValue(mockNavigate);
    require('../../context/AuthContext').useAuth.mockReturnValue({
      login: mockLogin,
    });
  });

  afterEach(() => {
    jest.clearAllMocks();
  });

  it('renders the login form', () => {
    render(<LoginPage />);
    expect(screen.getByText('Enter login credentials:')).toBeInTheDocument();
    expect(screen.getByLabelText(/Username/i)).toBeInTheDocument();
    expect(screen.getByLabelText(/Password/i)).toBeInTheDocument();
  });

  it('navigates to dashboard on successful login', async () => {
    mockLogin.mockResolvedValue({ success: true });
    render(<LoginPage />);

    fireEvent.change(screen.getByLabelText('Username:'), { target: { value: 'testuser' } });
    fireEvent.change(screen.getByLabelText('Password:'), { target: { value: 'password123' } });
    fireEvent.submit(screen.getByTestId('login-form'));

    await waitFor(() => {
      expect(mockNavigate).toHaveBeenCalledWith('/dashboard');
    });
  });

  it('shows error toast on failed login', async () => {
    mockLogin.mockResolvedValue({ success: false, error: 'Invalid credentials' });
    render(<LoginPage />);

    fireEvent.submit(screen.getByTestId('login-form'));

    await waitFor(() => {
      expect(toast.error).toHaveBeenCalledWith('Login failed. Invalid credentials');
    });
  });
});
```

Figure 10: Login Page testing handling successful and unsuccessful login

```

import axiosInstance from "../axiosInstance.js";
import { getBalanceByCustomerId } from "../balanceApi.js";

jest.mock('../axiosInstance');

describe("getBalanceByCustomerId", () => {
  it('should make a GET request to the correct endpoint with customerId', async() => {
    // arrange
    const customerId = 1;
    const mockResponse = {data: {amount: 200}};
    // mock the resolved value
    axiosInstance.get.mockResolvedValue(mockResponse);

    // act
    const response = await getBalanceByCustomerId(customerId);

    // assert
    expect(axiosInstance.get).toHaveBeenCalledTimes(1);
    expect(axiosInstance.get).toHaveBeenCalledWith(`/balance/customerId/${customerId}`)
    expect(response).toEqual(mockResponse);
  })

  it("should handle error properly", async() => {
    // arrange
    const customerId = 2;
    const error = new Error('Request failed. Customer not found')

    // act
    axiosInstance.get.mockRejectedValue(error)

    // assert
    await expect(getBalanceByCustomerId(customerId)).rejects.toThrow(error)
  })
})

```

Figure 11: Testing of the GET API call to the backend, verifying handling of expected and unexpected responses.

```

5
6 describe("createVehiclePlate", () => {
7   const mockRequestBody = {plateNumber: "AB12 UJX", available: true, price: 30}
8   const mockResponseData = {id: 1, plateNumber: "AB12 UJX", available: true, price: 30}
9
10  beforeEach(() => {
11    axiosInstance.post.mockClear();
12  })
13  it("should make a POST request to /vehiclePlates with the correct body", async() => {
14    // arrange
15    axiosInstance.post.mockResolvedValueOnce({data: mockResponseData})
16
17    // act
18    const result = await createVehiclePlate(mockRequestBody)
19
20    // assert
21    expect(axiosInstance.post).toHaveBeenCalledTimes(1)
22    expect(axiosInstance.post).toHaveBeenCalledWith("/vehiclePlates", mockRequestBody)
23    expect(result).toEqual(mockResponseData)
24  })
25 })
26
27 describe("updateVehiclePlate", () => {
28   const mockId = 1;
29   const mockBody = {available: true}
30
31   beforeEach(() => {
32     axiosInstance.patch.mockClear();
33   })
34
35   it('should make a PATCH request to the correct URL with the body', async() => {
36     // arrange
37     axiosInstance.patch.mockResolvedValueOnce({data: mockBody})
38
39     // act
40     await updateVehiclePlate(mockId, mockBody)
41
42     // assert
43     expect(axiosInstance.patch).toHaveBeenCalledTimes(1)
44     expect(axiosInstance.patch).toHaveBeenCalledWith(
45       `/vehiclePlates/${mockId}`,
46       mockBody
47     )
48   })
49 })
50 })

```

Figure 12: Testing of the POST and PATCH API call to the backend.

Server-side

```
this is the role: CUSTOMER
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.060 s -- in service.UserDTOServiceTest
[INFO] Running service.TransactionDTOServiceTest
[TransactionDTO(transactionId=1, userId=0, vehiclePlateId=0, pricePaid=50.5, transactionDate=2025-04-30), TransactionDTO(transactionId=2, userId=0, vehiclePlateId=0, pricePaid=70.6, transactionDate=2025-04-30)]
[INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.064 s -- in service.TransactionDTOServiceTest
[INFO] Running service.VehiclePlateDTOServiceTest
[INFO] Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 s -- in service.VehiclePlateDTOServiceTest
[INFO] Running service.BalanceDTOServiceTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.020 s -- in service.BalanceDTOServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 59, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.456 s
[INFO] Finished at: 2025-04-30T18:20:06+01:00
[INFO] -----
```

Figure 13: Successful test runs on server-side.

Test Name	Description	Expected Outcome
test_getAllVehiclePlates	Retrieves all vehicle plates with pagination	HTTP 200 OK with a page of DTOs
test_getAllVehiclePlatesByReg	Retrieves plates matching a registration pattern	HTTP 200 OK with filtered list of DTOs
test_getVehiclePlateById	Retrieves a specific vehicle plate by ID	HTTP 200 OK with matching DTO
test_addVehiclePlate	Adds a new vehicle plate with valid data	HTTP 201 CREATED with created DTO
test_addVehiclePlate_ReturnsBadRequest	Attempts to add a plate with invalid/missing data	HTTP 400 BAD REQUEST
test_replaceVehiclePlate	Replaces an existing plate with valid data	HTTP 200 OK with updated DTO
test_replaceVehiclePlate_ReturnsBadRequest	Attempts to replace a plate with invalid data	HTTP 400 BAD REQUEST
test_updateVehiclePlate	Partially updates an existing plate	HTTP 200 OK with updated DTO
test_deleteVehiclePlate_returnsTrueAndFalse	Deletes an existing plate and handles non-existent case	HTTP 204 NO CONTENT and 404 NOT FOUND
test_getAllVehiclePlatesByUserId	Retrieves all plates associated with a specific user ID	HTTP 200 OK with user-specific plate list

Figure 14: Description and Expected Outcomes of Unit Tests for VehiclePlateController Methods

9. Conclusion

In conclusion, the progression from static web pages to dynamic, interactive applications has been shaped by the development of client-side and server-side scripting technologies. The emergence of JavaScript and the subsequent introduction of Node.js have played a pivotal role in enabling unified, full-stack development using a single programming language. This report builds upon this historical and technological context to present a comparative analysis of client-side and server-side scripting, a focused investigation into Node.js, and comprehensive documentation of a web-based application designed for vehicle plate registration and purchase.

References

- [1]
L. Martin, "Server-Side Scripting Languages," *Encyclopedia of Social Network Analysis and Mining*, pp. 2466–2474, 2018, doi: https://doi.org/10.1007/978-1-4939-7131-2_363.

- [2]
C. Charan, "Expanding Web Development Horizons: Integrating WebAssembly with React, Vue, and Angular," *Journal of Scientific and Engineering Research*, 2021, doi: <https://doi.org/10.5281/zenodo.15075098>.

- [3]
"View of WEB PROGRAMMING INFORMATION," *Interoncof.com*, 2025.
<https://interoncof.com/index.php/finland/article/view/4080/3792>

- [4]
] T. Crawford and T. Hussain, "A Comparison of Server Side Scripting Technologies," in *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, Athens, 2017. www.proquest.com.
<https://www.proquest.com/docview/2139472343?pq-origsite=gscholar&fromopenview=true&sourcetype=Conference%20Papers%20&%20Proceedings>

- [5]
A. Kashyap, B. Verma, T. Kumarsahu, and Priyanka Sahu, "EVOLUTION OF WEB DEVELOPMENT," Peer-Reviewed, Open Access. Available:
https://www.irjmet.com/uploadedfiles/paper/issue_6_june_2023/41542/final/fin_irjmet1686036998.pdf

- [6]
D. Flanagan, *JAVASCRIPT : the Definitive Guide*. O'Reilly Media, Inc., 2020.

- [7]
X. Huang, "Research and Application of Node.js Core Technology," *2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)*, Dec. 2020, doi: <https://doi.org/10.1109/ichci51889.2020.00008>.

- [8]
"Distributed Systems with Node.js," *Google Books*, 2020.
<https://books.google.co.uk/books?hl=en&lr=&id=aBYHEAAAQBAJ&oi=fnd&pg=PP1&dq=node+js+&ots=a5ZNJRLCJz&sig=bks->

[Lv9u5LtwjYvhTQd3rKARMk&redir_esc=y#v=onepage&q&f=false](#) (accessed Apr. 29, 2025).

[9]

“Node.js for Beginners,” *Google Books*, 2024.

https://books.google.co.uk/books?hl=en&lr=&id=vZoAEQAAQBAJ&oi=fnd&pg=PP1&dq=node+js+&ots=4jJiatZS7&sig=nR6hEJwlOIfoaxERvgMQlvl-ns&redir_esc=y#v=onepage&q&f=false (accessed Apr. 29, 2025).

[10]

B. Basumatary and N. Agnihotri, “Benefits and Challenges of Using NodeJS,” *International Journal of Innovative Research in Computer Science & Technology*, vol. 10, no. 3, pp. 67–70, May 2022, doi: <https://doi.org/10.55524/ijircst.2022.10.3.13>.

[11] G. Jadhav and F. Gonsalves, "Role of Node.js in Modern Web Application Development," *International Research Journal of Engineering and Technology (IRJET)*, vol. 7, no. 6, Jun. 2020. <https://www.irjet.net/archives/V7/i6/IRJET-V7I61149.pdf>

[12]

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, p. 012136, Jun. 2020, doi: <https://doi.org/10.1088/1757-899x/801/1/012136>.

GRADEMARK REPORT

FINAL GRADE

75/100

GENERAL COMMENTS

Server/Client Code (25): 19

Node.js (25):19

SW Doc (50): 37

FEEDBACK

=====

You have covered most of the key features of client and server scripting. I was hoping for a bit more detail on the server side environment in particular.

Very good discussion of the internals of node.js, I was expecting more discussion of the benefits of its use such as NPM.

Could have provided a bit more discussion of the code, but there is some code discussion and good evidence of testing.

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
