

Web Development Report

Abdulmalik Elghedafi - 2420753

July 2025

Contents

1	Introduction	2
2	client side scripting	2
3	Client and Server-side scripting - compare and contrast	2
4	Critically evaluating the benefits and drawbacks of the Node.JS framework	4
4.1	What is Node.js	4
4.2	Benefits and drawbacks of Node.js	4
5	Documentation	5
5.1	Ajax documentation	5
5.2	User manual	6
5.3	Overview of the design	6
5.4	Discussion of the impressive parts of my application	7

1 Introduction

This is a report that discusses the differences between client and server side code, as well as investigates the node.js framework and critically analyses the advantages and disadvantages of node.js. The report also has a discussion documenting the REST Web Service frontend and backend with screenshots.

2 client side scripting

Client-side scripting is when a web browser uses JavaScript to execute code and offers interactivity between the client and webserver by running under an asynchronous model. Each component on the User Interface (UI) i.e. a button will use an event handler that can be executed immediately within the background while the user continues using the website. The client uses JavaScript to communicate with the Document Object Model (DOM) through API calls to allow components located inside the DOM object to reload the page without refreshing. An example would be clicking a button to submit a contact form; this would result in a response message from the server stating that it has been successfully processed without the need of reloading the page [1].

3 Client and Server-side scripting - compare and contrast

Client side and server-side scripting are both web components that deliver a dynamic and interactive experience to the user through the web browser, however both have their differences.

There are several similarities between client and server-side scripting. Server-side scripting refers to a HTTP request being sent to the web server to fetch data from the database, before receiving the data and displaying the information to the client's web browser using HTML. In contrast, client side allows full control in managing and rendering content through the web browser using HTML, CSS and JavaScript to render all the content without requiring the need to reload the page excluding the need of a web server. Both the client and server can process and format data, but in different ways. The client uses Asynchronous JavaScript and XML (AJAX) to request data from the web server, then rendering the JavaScript Object Notation (JSON) response in the background and generating the HTML directly in the web browser without requiring the page to be updated.

On the other hand, server handling offers data to be processed entirely before responding back to the client, which increases performance and handling, allowing the client to perform quick operations and load quicker. A downside of server side would entail the client side needing to reload the page to update the contents of the webpage. The client is superior as it be responsive in responding to

user actions, by dynamically using AJAX and locally generating content on the browser, meanwhile the server can entail in slower timing, due to requiring the page to be reloaded and updated to and from the web server. The complexity can entail the client becoming more difficult to develop due to creating AJAX functions for every component on the page, differing to the server which handles all the business logic through a RESTful API such as Django, Node.js [2].

Mesbah and Deursen state that when a web browser requires unprocessed data to update parts of the user interface, the client can make server calls to specific structured backend services that can respond with the information [3]. Meanwhile Kimak and Ellman highlights that when a client uses local storage to save data from a website, it uses IndexedDB. This is held on the users web browser. However, this does come with a major constraint, that being that different web browsers can impact how well the database performs for user's i.e. Google Chrome. Microsoft Edge, Firefox and Safari [4].

In contrast, Mesbah and Deursen describe server side as offering exhaustive database access directly. The term outlined by Mesbah and Deursen state that the server can handle operations such as the application being updated i.e. submitting a form, logging into the website and client actions such as fetching data through pressing buttons. The server has the ability to access to any resource such as a web service and database to allow the server to carry out the necessary actions to complete the request [3]. Having a server that can utilise SQL to process database operations, allows the server to handle the heavy lifting and performance.

The client side must use HTTP to be able to communicate with the web service, as the client is not permitted to access the database server or business logic. Delta connectors are used instead, which are essentially a light way of interacting with the web server by sending requests and response from and to the client. The only limitation of using Delta connectors, is that it must follow exactly what the web server states due to the constraints set in the Application Programming Interface (API) [3].

The core distinction between client side and server side is where the data is kept to be processed, locally or via a server. Traditionally websites have relied on using databases via server-side scripting, as they take the load off the client and servers use enterprise database platforms which support SQL in managing complex queries. Despite this, client side is limited to using the database stored within the web browser and operates differently based on the web browser chosen i.e. Google Chrome and Firefox. The data is stored in key and value pairs and therefore does not use SQL to query complex operations due to the way it is storing data [4].

4 Critically evaluating the benefits and drawbacks of the Node.JS framework

4.1 What is Node.js

Node.js is a run time environment that runs JavaScript outside the browser, specifically for building backend applications as oppose for the front end. It was created to build scalable and fast web applications. Built on Chrome's V8 engine, Node.js offers translating JavaScript to machine code to let the computer understand it and is a cross-platform language, where it can be used on Windows, Linux, MacOS etc. Thanks to its non-blocking input/output and event-driven architecture, software developers can help build applications that handle extensive quantities of data across various devices through live updates. Node.js enhances productivity as it is created in a way to handle many users simultaneously without slowing down. This makes it a tailored approach for modern web applications [5].

4.2 Benefits and drawbacks of Node.js

An advantage of Node.js is that as a developer, you only require to learn and understand one language: JavaScript, to develop the front and backend of an application. This significantly reduces the requirement of learning multiple languages and switching between tools to create full stack applications. The author suggests that this creates an advantage to inexperienced developers such as juniors. While this does optimise the software cycle, it can introduce lack of flexibility in using other language's for the backend [6].

A drawback is that novice users can find Node.js difficult due understanding topics in JavaScript such as asynchronous and call back functions. These topics can be complex and complicated to new developers. Ultimately this can reduce time spent on development and increase software bugs if the developer is not using industry concepts in projects [6].

An additional advantage of using Node.js over other languages is that Node.js can handle multiple tasks at the same time without waiting for the server to finish the previous task using asynchronous. Thus, making it non-blocking and effective through single threading. However, with PHP the server handles one request per a thread. This can quickly introduce overhead in memory and computational power, resulting in less efficiency. This indicates that Node.js is the preferred choice [6].

A further drawback is that, when a program overloads the event loop with asynchronous calls that are running simultaneously which rely heavily on a computers Central Processing Unit (CPU), can hinder optimal performance. Therefore, it is crucial that thorough planning by developers becomes imperative. This mitigates problems such as "callback hell" where asynchronous code is

deeply nested together, which can impair readability and introduce error prone debugging. As a result of these limitations, Node.js is not ideal for applications that demand high performance computing power or handling complex logic [6].

5 Documentation

5.1 Ajax documentation

The client side uses Ajax for updating content when the user clicks a button for retrieving data and sending post requests without reloading the page.

The code in figure 1 is a function that works by getting the id of the button and points it to a function call to get all the available plates as well as purchasing plate button request. The figure 2 is the function which is a GET request to the available plate REST endpoint and calls the `GotAllAvailablePlatesResult` function on success, otherwise displays an error alert if it fails. This function is a HTTP request that presents the table, which uses Bootstrap and appends each field such as id, plate, availability and price with a purchase button on each row as shown in figure 3.

This code in figure 4 is when the user clicks the button to purchase plates, sending a GET request and returning the data in JSON format. The figure 5 presents the table using Bootstrap for the purchased plates and adds functionality for adding a new plate for the admin, which uses the `postNewLicensePlateData()` function call.

This figure 6 is a post request to handle the form submission from the admin, it gets the input field (price and plate), validates the plate using Regex, customised to match DVLA format with a twist of using hyphens instead of spaces, validates the price between 50 to 1000 inclusive, displays an error if it fails and sends a post request on success. This is the second part of the admin form validation, figure 7, it creates the data object and submits the new license plate to the server to the API call after validating all the information.

The figure 8, clears the input for the textboxes, after submitting the data to the server and reloads the available plates table. The `postPurchasePlateButtonRequest()` function sends a POST request by the vehicle plate ID. If successful, an alert will appear. It will also check if the vehicle plate is already purchased. It will also handle server errors such as HTTP 400/500.

This function figure 9, sets up the search button handler, gets the search input and makes a call with it, but if the user does not enter any text to search, it will show all available plates. It will then make a request to the endpoint.

This figure 10 is for searching purchased plates functionality and button handler. It is very similar to the search functionality for license plate, except for purchasing. It is a GET request to the purchase endpoint and then returns the data and displays it.

5.2 User manual

To install my REST Web Service application. This is what you should see when it loads figure 11

Steps:

1. Git clone the repo: <https://github.com/2420753/RestWebServiceAssignment>
2. Run docker-compose up in the terminal
3. Use ideally a private / incognito tab to remove any cache by visiting <http://localhost:8080>
4. To stop the application, use docker-compose down

5.3 Overview of the design

- **Frontend:** It is written in HTML with Ajax to validate front end components and reload page without refreshing browser. I also use Bootstrap for tables and buttons and jQuery script.
- **Backend:** Spring boot using the REST API with ORM in Hibernate, utilising JPA queries and custom queries for the SQL database.
- **Database:** Uses PostgreSQL to store three tables, customer, license plate and purchase.
- **Docker:** This holds settings for running docker, database, spring boot application to deploy with one command.
- **Controller layer:** This handles the HTTP Get and Post mapping for the end points. I have two, one for license plate and the other is a purchase controller.
- **DTO:** Handles security issues; by not including any sensitive data so I can use it to display data to the front end, rather than using the entity class which includes all fields.
- **Entity:** Sets all the fields for each of the classes, along with a default constructor and one for the class that sets all the values to the field for the object. It also handles the entity relationships mappings for foreign keys in other tables. Also uses FetchType.LAZY to only get data when it is needed.

- **Repository:** This provides access to CRUD methods from the server to database.
- **Service layer:** Handles the business logic and methods such as getting all the available plates.

5.4 Discussion of the impressive parts of my application

One of the coolest parts of building my Spring Boot REST web service was seeing the data load on the front-end using AJAX without needing to refresh the page. Which felt modern. I also really enjoyed implementing front end validation with AJAX. Getting instant feedback when something was wrong with the input was super satisfying to see in action.

I used Regex to create a custom pattern for the DVLA vehicle plates, but with a twist. I replaced the usual spaces with hyphens. The format I went with was one to three letters, followed by one to two letters, and then one to four numbers, all separated by hyphens. I also set a max length and made sure users could only type in that specific format. When the user types a plate in lowercase, it will automatically convert to uppercase. Another part I liked was the transition of plates from the "available" table to the "purchased" table it made the whole process feel more real and complete. I added validation to the admin form too, so if someone tries to add a plate that already exists in either table, it throws an error.

I also added partial search functionality for both available and purchased plates, which means you can type part of a plate and still get results. And just to be extra safe, I included server side validation as well, so even if someone somehow bypasses the front end checks, the backend still catches it. That extra layer of protection felt like a solid win.

```

1  // Ajax function for GET request of available plates button, when user clicks button.
2  //Prevents default behaviour of the button and disables cache.
3
4  $(function() :void) { Abdulmalik Elghedafi *
5  {
6      $.ajaxSetup( target: { cache: false });
7
8      $("#showAllAvailablePlatesBtn").click(function (e) :boolean {
9          getAllAvailablePlates();
10         e.preventDefault();
11         return false;
12     });
13
14     // Button functionality for purchasing plates.
15     $(document).on( types: "click", selector: ".purchasePlates-button", data: function () :void {
16         // grabbing the id value that is being returned from the button.
17         var purchaseVehiclePlateID = $(this).attr( name: "id");
18         // Points to getting the ID to purchase plate.
19         postPurchasePlateButtonRequest(purchaseVehiclePlateID);
20     });
21 }
22

```

Figure 1: Button functionality for available plates button

```

23 // Function for available plates in Ajax.
24 function getAllAvailablePlates() :void Show usages & Abdulmalk Elghedafi
25 {
26     $.ajax({
27         cache: false,
28         type: "GET",
29         url: "/licenseplates/available?" + new Date().getTime(),
30         dataType: "json",
31         success: GotAllAvailablePlatesResult,
32         error: (req, status, error) :void => window.alert("Error getting available plates " + error)
33     });
34 }
35

```

Figure 2: Function for GET request of all available plates

```

36 // HTTP request for presenting the information.
37 function GotAllAvailablePlatesResult(json, status, req) :void Show usages & Abdulmalk Elghedafi
38 {
39     // Creating the table using bootstrap for headers for each column
40     var html = `<table class="table table-striped">`;
41     html += `<thead><tr><th>Id</th><th>Vehicle Plate</th><th>Availability</th><th>Listed Price</th></tr></thead><tbody>`;
42
43     // Adding the fields from the database to grab the data and display them for each row.
44     for (let i :number = 0; i < json.length; i++)
45     {
46         html += `<tr>`;
47         html += `<td>` + json[i].id + `</td>`;
48         html += `<td>` + json[i].vehiclePlate + `</td>`;
49         html += `<td>` + json[i].availability + `</td>`;
50         html += `<td>` + json[i].listedPrice + `</td>`;
51         html += `<td><button type="button" class="purchasePlates-button" id="${ json[i].id }" >Purchase Plate</button></td>`;
52         html += `</tr>`;
53     }
54     html += `</tbody></table>`;
55
56     // This is returning the data to display when the user has clicked the button to append the html data to display the table to the client side.
57     $("#dataReceivedFromBackend").html(html);
58 }
59

```

Figure 3: Displays the data in a table for the frontend using bootstrap

```

64 // Function for when the user clicks the button, it will load the function for getting all PURCHASED plates
65 $(function() :void & Abdulmalk Elghedafi
66 {
67     $.ajaxSetup({ cache: false });
68
69     $("#showAllPurchasedPlatesBtn").click(function (e) :boolean {
70         getAllPurchasedPlates();
71         e.preventDefault();
72         return false;
73     });
74 });
75
76 // Get request for all purchased plates via the Springboot backend
77 function getAllPurchasedPlates() :void Show usages & Abdulmalk Elghedafi
78 {
79     $.ajax({
80         cache: false,
81         type: "GET",
82         url: "/purchases?" + new Date().getTime(),
83         dataType: "json",
84         success: GotAllPurchasedPlatesResult,
85         error: (req, status, error) :void => window.alert("Error in getting purchased plates " + error)
86     });
87 }
88

```

Figure 4: Purchase plate functionality with button handler

```

119 // HTTP request for presenting the information
120 function GetAllPurchasePlatesResult(json, status, req) :void Show usages: @Abdulmalik Elghedafi
121 {
122     // Creating the table using bootstrap for headers for each column
123     var html :string = '<table class="table table-striped">';
124     html += '<thead><tr><th>ID</th><th>Date of Purchase</th><th>Price Paid</th><th>Vehicle Plate</th></tr></thead><tbody>';
125     // Adding the fields from the database to grab the data and display them for each row.
126     for (let i :number = 0; i < json.length; i++)
127     {
128         html += '<tr>';
129         html += '<td>' + json[i].id + '</td>';
130         html += '<td>' + json[i].dateOfPurchase + '</td>';
131         html += '<td>' + json[i].pricePaid + '</td>';
132         html += '<td>' + json[i].vehiclePlate + '</td>';
133         html += '</tr>';
134     }
135     html += '</tbody> </table>';
136
137     // This is returning the data to display when the user has clicked the button to append the html data to display the table to the client side
138     $("#dataReceivedFromBackend").html(html);
139 }
140
141 // Adding functionality to allow an ADMIN ONLY to post new plates to the system
142 $(function() :void @Abdulmalik Elghedafi
143 {
144     $.ajaxSetup({url: '', cache: false });
145
146     $("#addNewPlatesSubmitButton").click(function (e) :boolean {
147         postNewLicensePlateData();
148         e.preventDefault();
149         return false;
150     });
151 });
152

```

Figure 5: Displaying the purchased plates table

```

153 // Post request for submitting new licensePlate information via the Front end.
154 function postNewLicensePlateData() :void Show usages: @Abdulmalik Elghedafi
155 {
156     // Taking the value from the two input boxes for adding new plate, removing any white space.
157     var vehiclePlateInput = $("#vehiclePlate-text").val().trim().toUpperCase();
158     var listedPriceInput = $("#listedPrice-text").val().trim();
159
160     var regExp :RegExp = /^[A-Z]{1,3}[A-Z]{1,2}[0-9]{1,4}$/;
161     const lowestPrice :number = 50;
162     const maxPrice :number = 1000;
163     var priceInput :number = parseInt(listedPriceInput);
164
165     // Front end validation of the regex input against the vehicle plate input
166     if (regExp.test(vehiclePlateInput)) {
167         $("#postNewPlatesResult").html("<div>The value is good</div>");
168     } else {
169         $("#postNewPlatesResult").html("<div>The data entered for vehicle plate or listed price is not valid, please try again.</div>");
170         return;
171     }
172
173     if (isNaN(priceInput)) {
174         $("#postNewPlatesResult").html("<div>The price entered is not a number!</div>");
175         return;
176     } else if (priceInput < lowestPrice) {
177         $("#postNewPlatesResult").html("<div>Too low! Enter a price higher than 50</div>");
178         return;
179     } else if (priceInput > maxPrice) {
180         $("#postNewPlatesResult").html("<div>Too high! Enter a price lower than 1000</div>");
181         return;
182     }
183 }
184

```

Figure 6: Validation of admin form for vehicle plate and price

```

157     var data = {
158         vehiclePlate: vehiclePlateInput,
159         availability: true,
160         listedPrice: listedPriceInput,
161     };
162
163     $.ajax({
164         cache: false,
165         type: "POST",
166         url: "/licenseplate/add",
167         data: JSON.stringify(data),
168         contentType: "application/json",
169         dataType: "json",
170         // If successful run the below function
171         success: GetNewLicensePlateData,
172         // Otherwise an error will be executed
173         error: function(req, status, error) {
174             // Checks if the request sent to the server is empty then the plate already exists in the available table.
175             // Else an error is sent.
176             if (req.responseText === "") {
177                 alert("Sorry, the vehicle plate you are trying to add has already been added or has been purchased!");
178             } else {
179                 alert("Error! Status is: " + status + ". The error is: " + error);
180             }
181         }
182     });
183 }

```

Figure 7: Functionality for the validation of the admin form

```

184 // Changing the input box for adding new plate via frontend and a message appears when a plate has been added successfully.
185 function GetNewLicensePlateData(status, status, req) {
186     // Show message & Add/remove signpost
187     if ("vehiclePlate" === req) {
188         $("#vehiclePlate").val("");
189     }
190     if ("listedPrice" === req) {
191         $("#listedPrice").val("");
192     }
193     if ("listedPrice" === req) {
194         $("#listedPrice").val("");
195     }
196     // This updates the available plates table when an admin adds a new plate via the frontend
197     getAvailablePlates();
198 }
199
200 // Functionality for the purchase button when user is on the available plates page
201 // This will allow the user to click purchase and buy the plate.
202 function postPurchasePlateButtonRequest(purchaseVehiclePlateID) {
203     // Show message & Add/remove signpost
204     $.ajax({
205         cache: false,
206         type: "POST",
207         url: "/purchase/" + purchaseVehiclePlateID,
208         data: JSON.stringify({}),
209         contentType: "application/json",
210         success: function(value) {
211             // Here I am checking if the value is null (if it is in the purchased table) or returns an empty string, then alert user of the error, as it's a duplicated ID.
212             if (value === null || value === "") {
213                 alert("Error! Duplicated ID!");
214             } else {
215                 alert("Success! You have managed to purchase a vehicle plate!");
216                 getAvailablePlates();
217             }
218         },
219         // This error function is for server issues HTTP 400/500
220         error: function(req, status, error) {
221             alert("There has been an issue with the server: " + error + ". The status is: " + status);
222         }
223     });
224 }

```

Figure 8: Functionality for purchasing plate post request

```

220 // Adding the functionality to make the searching of a license plate on the front end.
221 $(function() {
222     $.ajaxSetup({ cache: false });
223
224     $('#searchAvailableLicensePlateButton').click(function (e) {
225         var captureSearchPlateInput = $('#LicensePlateInputBox').val();
226         if (captureSearchPlateInput) {
227             SearchGetRequestForAllAvailablePlates(captureSearchPlateInput);
228         } else {
229             getAllAvailablePlates();
230         }
231         e.preventDefault();
232         return false;
233     });
234 });
235
236 // Function for doing the search for available plates
237 function SearchGetRequestForAllAvailablePlates(captureSearchPlateInput) {
238     $.ajax({
239         url: "/licencesplates/available/search/vehiclePlates" + captureSearchPlateInput,
240         type: "GET",
241         dataType: "json",
242         // data: JSON.stringify({}),
243         contentType: "application/json",
244         success: GetAllAvailablePlatesResult,
245         error: (req, status, error) => window.alert("Error in getting search input for available plates: " + error + ". The status is: " + status)
246     });
247 }

```

Figure 9: Search button functionality and button handler

```

252 // Functionality for the search of purchased plates
253 $(function() {
254     $.ajaxSetup({ cache: false });
255
256     $('#searchPurchasedLicensePlateButton').click(function (e) {
257         var captureSearchPlateInput = $('#LicensePlateInputBox').val();
258         if (captureSearchPlateInput) {
259             SearchGetRequestForAllPurchasedPlates(captureSearchPlateInput);
260         } else {
261             getAllPurchasedPlates();
262         }
263         e.preventDefault();
264         return false;
265     });
266 });
267
268 // Function for getting the search for purchased plates
269 function SearchGetRequestForAllPurchasedPlates(captureSearchPlateInput) {
270     $.ajax({
271         url: "/licencesplates/purchased/search/vehiclePlates" + captureSearchPlateInput,
272         type: "GET",
273         dataType: "json",
274         // data: JSON.stringify({}),
275         contentType: "application/json",
276         success: GetAllPurchasedPlatesResult,
277         error: (req, status, error) => window.alert("Error in searching for purchased plates " + error + ". The status is: " + status)
278     });
279 }

```

Figure 10: Functionality for searching purchased plates

Figure 11: User interface for my REST Spring boot application

References

- [1] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, “A study of causes and consequences of client-side javascript bugs,” *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 128–144, 2017.
- [2] J. D. Walker and S. C. Chapra, “A client-side web application for interactive environmental simulation modeling,” *Environmental Modelling Software*, vol. 55, pp. 49–60, 2014. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1364815214000334>
- [3] A. Mesbah and A. Van Deursen, “An architectural style for ajax,” in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA’07)*. IEEE, 2007, pp. 9–9.
- [4] S. Kimak and J. Ellman, “Performance testing and comparison of client side databases versus server side,” *Northumbria University*, 2013.
- [5] S. S. N. Challapalli, P. Kaushik, S. Suman, B. D. Shivahare, V. Bibhu, and A. D. Gupta, “Web development and performance comparison of web development technologies in node. js and python,” in *2021 International Conference on Technological Advancements and Innovations (ICTAI)*. IEEE, 2021, pp. 303–307.
- [6] 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, 2022.