

Web Development: Client-side vs Server-side Scripting and Node.js Framework

Justin Hendy

Student Number: 0250941

University of Wales Trinity Saint David

July 2025

Contents

1	Introduction	3
2	Compare and Contrast: Client-side vs Server-side Scripting	3
2.1	Client-Server Computing Model	3
2.2	Client-side Scripting	3
2.3	Server-side Scripting	4
3	Node.js Framework: Benefits & Drawbacks	4
3.1	Overview of Node.js	4
3.2	Benefits of Node.js	4
3.3	Drawbacks of Node.js	5
4	Documentation of Provided Node.js & AJAX Code for the Number Plate Purchasing App	5
4.1	Overview of Design	6
4.2	Discussion of Main/Interesting Aspects	6
4.3	Testing	13
5	Conclusion	14
6	Appendix One: User Manual	16
6.1	Running from Docker	16
6.2	Running without Docker	17
6.3	Using the Application	18
6.3.1	Searching for Number Plates	18
6.3.2	Purchasing Number Plates	18
6.3.3	Searching Previous Purchases	19
6.3.4	Adding Number Plates	20
7	Appendix Two: Test Code	20
7.1	Frontend Tests	20
7.1.1	Date Range Helper Tests	20
7.1.2	Price Range Helper Tests	22
7.1.3	Validate Methods Helper Tests	23
7.1.4	Add Number Plates Tests	25
8	Appendix Three: Test Results	27
8.1	Date Range Helper Test Coverage	27
8.2	Price Dropdown and Price Range Helper Test Coverage	28
8.3	Validate Methods Helper Test Coverage	29
8.4	Add Number Plates Test Coverage	30

1 Introduction

The purpose of this essay is to compare and contrast client-side and server-side scripting languages, explore the benefits and drawbacks of the Node.js framework, and provide an overview of a small web application developed using the Node.js framework and AJAX HTTP requests.

Before examining client-side and server-side scripting languages, it is essential to understand the role of scripting within web development. When a user or system interacts with a web page or website, these interactions can elicit a range of behaviours or responses, such as reacting to a button click, updating a page display, changing interactive menus, processing data, or communicating with a database. The term scripting refers to the process of writing the code that defines these actions, and the programming languages used to write this code are known as ‘scripting languages’ [1]. Scripting languages are programming languages that differ in several ways from ‘general-purpose languages’, with one difference being the lack of a compilation step required to execute the code. When executing a scripting language, an interpreter executes each line of code one at a time at runtime, and no translation into machine code takes place [1]. However, some scripting languages use what is known as Just-In-Time compilation, a hybrid of the two [2]. Other differences include their purposes/use cases, learning curves, typing and development complexity. Generally, scripting languages are concise [3] and require less boilerplate code, making them suitable for web development, where the ability to build and deploy features rapidly is crucial [4]. Scripting languages have uses beyond just web development; however, the purpose of the following section is to examine the application of scripting languages within web development, specifically the differences when applied to client-side and server-side development.

2 Compare and Contrast: Client-side vs Server-side Scripting

2.1 Client-Server Computing Model

When two systems need to communicate, they employ the client-server computing model, in which the client system requests a resource or service, and the server system responds to that request. Within web development, the client is the web browser, and the server or web/application server is a remote computer or system that hosts the website or web application [5]. The client, as previously mentioned, sends requests to the server, displays content to the user, providing interactivity, makes API calls and runs the client-side scripts [6]. The server receives requests, hosts files such as HTML, CSS, and images, returns API responses, and executes server-side scripts [7].

2.2 Client-side Scripting

Client-side scripting executes within the browser’s runtime environment, managing what the user can see and interact with [8]. Typical functionality associated with client-side scripting includes providing interactivity such as form validation, event handling, dynamic content updates; user interface enhancements such as animations and transitions, modal windows, tabs, carousels, dropdown menus and real-time UI changes; DOM manipulation such as the adding, removing or editing of HTML elements dynamically, real-time changes to text, styles or structure of a page; asynchronous communication with the server using AJAX or Fetch API, load new data without refreshing the whole page; browser control to access and manipulate browser features such as cookies, local and session storage, manage navigation history and control pop-ups; improve performance by reducing unnecessary server requests and providing immediate feedback and quicker UI response [9] [6] [10] [11].

In modern web development, JavaScript and TypeScript (a superset of JavaScript) are the primary client-side scripting languages; others, such as VBScript, have existed, but JavaScript now dominates

the client-side scripting landscape [11] [12]. When required, CSS is applied and used to style parsed and rendered HTML on the client-side, providing control over the structure and styling of a web page; however, these presentational languages are not scripting languages and do not technically execute on the client.

2.3 Server-side Scripting

Server-side scripting executes on the server, specifically, on the server's runtime environment that supports the scripting language used [8]. Web servers come in many forms and can be physical, in the shape of a single computer or a collection of computers running web server applications, such as Apache or Nginx [13]. Alternatively, they can be cloud-based servers, such as those provided by AWS or Azure. Typical functionality associated with server-side scripting includes database interaction, dynamic content generation, security, processing user inputs, applying business logic, routing requests, and managing files and resources [14]. Examples of server-side scripting languages include JavaScript (with Node.js), Java (with Spring), PHP, Python and Ruby. Utilising server-side scripting for data handling and database interactions improves the security of an application. Client-side runtime environments are beyond the developer's control, and scripts running directly in the user's browser could potentially expose sensitive data or provide opportunities to manipulate key application logic [8]. Therefore, it is crucial that data handling and database interactions occur on the server with the server-side scripting hidden from the client.

3 Node.js Framework: Benefits & Drawbacks

3.1 Overview of Node.js

Node.js is a runtime environment that enables JavaScript to execute outside the browser, allowing it to function as a server-side scripting language [15]. When developing applications using Node.js, developers have the option to utilise it natively or through a Node framework such as Express.js or NestJS. Both native Node.js and its associated frameworks have benefits and drawbacks. There are several key benefits associated with native Node.js, including its non-blocking and event-driven architecture, which can help produce scalable, high-performance services, particularly those classed as I/O-Intensive [16]. Node.js also equips developers with the opportunity to utilise JavaScript as both a client-side and a server-side scripting language, as well as providing access to the sizable NPM (Node Package Manager) ecosystem [17].

3.2 Benefits of Node.js

I/O-Intensive applications spend a significant amount of time processing inputs and outputs, such as querying a database, serving files over HTTP, calling external APIs, etc.; tasks that generally do not require heavy CPU usage. Node.js employs an event-driven, non-blocking I/O model through its event loop and the use of callbacks or promises, allowing it to resolve input and output processes without blocking other elements of the code from executing. In other words, it doesn't wait for tasks to complete before running additional tasks. As a result, the application can continue to handle and resolve other requests, send loading placeholders for improved UI experience, or render partial responses. Node.js runs the event loop on a single thread, providing low memory usage and high concurrency through its ability to handle thousands of concurrent I/O tasks. Node.js services utilise fewer resources due to their low memory and CPU usage, and therefore can serve more clients, allowing for significant horizontal scaling [15] [16] [17].

The benefits of using a single scripting language for full-stack development are relatively obvious, whether it's through improved or easier communication and collaboration across frontend and backend developers or the ability for a frontend developer to complete backend work without the need to learn a new language, or the opportunity to utilise consistent tooling, such as testing environments (Jest), linting (ESLint/Prettier), or build tools (Vite, Webpack, ESBuild) while also promoting code reuse and reduced context switching [18].

Node packages are reusable building blocks of code that developers can install and use within their Node.js projects, reducing the need to write boilerplate code and encouraging code reuse. The ubiquitous nature of popular packages also delivers developer confidence through continued testing and maintenance. There are over two million packages available via the NPM ecosystem, covering a range of tools, from testing libraries like Jest, Sinon, or Mocha, to database clients, UI frameworks, HTTP servers and utility libraries. NPM also provides rapid development tools, such as the ability to install packages via the CLI using a single 'npm i' followed by the package name. Small, single-purpose packages encourage developers to create modular applications, increasing maintainability and testability. Package versions are easily updated via a file called 'package.json', where developers can also add script commands for running specific tasks, such as running tests, building the app, or starting the app, to streamline their workflow [19].

3.3 Drawbacks of Node.js

There are several drawbacks associated with the use of Node.js; however, in many cases, these relate to the suitability of using Node.js in specific situations. The single-threaded event loop, while excellent for I/O-Intensive applications, does not handle CPU-intensive tasks such as video encoding, image processing or machine learning, where sharing the load via multiple threads can provide performance benefits [17].

The use of callbacks can introduce complexity in flow control and error handling. When numerous asynchronous operations depend on each other and/or errors require handling at every level, the flow becomes hard to read and therefore becomes harder to maintain or extend [15]. Utilising the `async/await` keywords can partially mitigate the issues created due to callback complexity. Using the `await` keyword when carrying out a task pauses a function until that task is complete, and `await` tasks are only available when enclosed within an `async` function. `Async` functions always return a promise, and if rejected, the nearest `try/catch` block handles this rejection. The function itself remains asynchronous and behaves as expected within the event loop; however, the internal execution of each call occurs sequentially. `Await/async` provides clean logic and a single point of error handling; however, poor structure and logic can still result in callback complexity, producing code that is hard to maintain.

The NPM ecosystem is vast, open-source, and public; therefore, it contains many low-quality packages that are poorly maintained and carry potential security risks [19]. Thus, developers must exercise caution when sourcing packages for their applications. Within large-scale security-critical industries or companies, such as the DVLA, it is best to utilise a privately maintained mirrored version of NPM to implement tighter controls regarding package availability and improve download performance.

4 Documentation of Provided Node.js & AJAX Code for the Number Plate Purchasing App

A user manual covering how to run and use the app is available in Appendix One. The web application explored in this section is a full-stack number plate purchasing app. This section provides an analysis of

the code through an overview of the code's design, a discussion of its key aspects, and a demonstration of frontend and backend testing. Code snippets and screenshots support this section.

4.1 Overview of Design

This section provides an overview of the code design and structure. The backend follows a standard MVC (Model-View-Controller) pattern, where there are two models: NumberPlate and NumberPlatePurchase, which represent the data or business entities and map to database tables. The view takes the form of JSON responses returned to the frontend; there is no server-side HTML view rendering as the frontend provides the presentation layer by serving static files via Node.js/Express. Finally, the Java classes NumberPlateController and NumberPlatePurchaseController handle HTTP requests, coordinate with the service layers and return the responses. The full-stack application utilises a multi-tier architecture, comprising a frontend presentation layer, a backend application layer, and a database data layer. The frontend is composed of HTML, CSS, and JavaScript served via Node.js/Express framework. The frontend handles user interactions, form submissions and dynamically displays data using client-side rendering. The client-side JavaScript uses AJAX/Axios to post and fetch data via HTTP to and from the backend REST endpoints, builds new HTML tables and updates the DOM with this data. Both the frontend and backend provide data validation and error handling. The server-side application layer comprises a Java Spring Boot application that exposes RESTful API endpoints for searching number plates, purchasing, searching previous purchases, and creating new number plates. Business logic, validation and database communication all occur on the backend. Finally, the database data layer utilises PostgreSQL and stores data related to number plates and their purchases. Through the use of individual Dockerfiles, the root folder's docker-compose.yml file, and the 'docker compose' command, all three services run from their own separate Docker containers, where Docker ensures that all services communicate with each other as needed.

4.2 Discussion of Main/Interesting Aspects

The most interesting aspect of the application involves the searching and purchasing of number plates. Starting with the search number plates HTML, there are several key elements to this page (see HTML code below).

```
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5   <meta charset="UTF-8">
6   <title>Search for number plate</title>
7   <link rel="stylesheet" href="/css/table.css">
8   <link rel="stylesheet" href="/css/page.css">
9   <link rel="stylesheet" href="/css/plate.css">
10 </head>
11
12 <!-- Form to search for number plates -->
13 <body>
14   <form id="readdata-form">
15     <h2 class="font-reg">Search for a number plate</h2>
16     <label class="font-reg" for="getregnumber-text">Registration:</label>
17     <input class="plate" id="getregnumber-text" type="text" required maxlength="8"/>
18     <!-- The elements of this div will only appear if advanced search has been selected -->
19     <div id="advancedsearch" style="display:none">
20       <label class="font-reg" for="getbyavailable-checkbox">Available:</label>
21       <input id="getbyavailable-checkbox" type="checkbox" />
22       <label class="font-reg" for="getbyminprice">Min Price:</label>
23       <select id="getbyminprice">
24         <option value="0">£0</option>
25       </select>
```

```

26     <label class="font-reg" for="getbymaxprice">Max Price:</label>
27     <select id="getbymaxprice" >
28         <option value="0" selected="selected">--</option>
29     </select>
30 </div>
31 <br />
32 <input id="ajaxgetbyregnumber-button" type="button" value="Search" />
33 <input id="ajaxadvancedsearch-button" type="button" value="Advanced search" />
34 <input id="reset-button" type="reset" value="Reset" />
35 <!-- Will be replaced with search results -->
36 <div id="getbyregnumberresults">Waiting...</div>
37 </form>
38 <!-- Return to home page -->
39 <a class="font-reg" href="index.html">Home Page</a>
40 <script type="module" src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.1/jquery.min.js"></script>
41 <script type="module" src="/js/searchnumberplates.js"></script>
42 </body>
43
44 </html>
45

```

Listing 1: Search Number Plates HTML

The HTML body consists of a form element with a single label and an input element, three buttons and two div tags with the IDs ‘advancedsearch’ and ‘getbyregnumberresults’. The remainder of the body contains an anchor tag for returning to the home page and two imported scripts, one for jQuery and one for the JavaScript module ‘searchnumberplates.js’. Returning to the ‘advancedsearch’ element, this is the parent element to a series of label, input and select elements that the user can hide or display by targeting the style attribute of the parent element. The style ‘display: none’ hides the child elements, and this attribute updates to ‘display: inline-block’ when the user clicks the advanced search button, as highlighted by the associated ‘searchnumberplates.js’ JavaScript below.

```

1 // When called, advanced search criteria is displayed or hidden depending on whether the advanced
  // fields are currently visible
2 // When visible new css is set to display the fields and set them in an inline-block
3 export function showAdvanced() {
4     const $advanced = $('#advancedsearch');
5     if ($advanced.is(':visible')) {
6         $advanced.hide();
7     } else {
8         $advanced.css('display', 'inline-block');
9     }
10 }

```

Listing 2: Advanced Search Button Functionality

Contained within this ‘advancedsearch’ div tag are two select elements used to select the minimum and maximum price range. To improve code readability and maintainability, a JavaScript helper file creates the option elements that set the prices available from the select element dropdown lists. This same JavaScript file contains the code that prevents the maximum price from being set lower than the minimum price. The fully annotated code snippet below is the code for the helper, followed by a code snippet that demonstrates how the main searchnumberplates.js module imports and utilises this code.

```

1 // Price drop down is used on purchases search and advanced number plate search
2
3 // Creates the list of price values to appear in the drop down
4 export function populatePriceOptions(priceSelect, min, max, step) {
5   // for loop starts at minimum value and loops through to maximum value increasing by set increment
   // step value on each iteration
6   for (let price = min; price <= max; price += step) {
7     // Select elements use option elements to display values
8     const option = document.createElement('option');
9     // sets the value attribute of each option element to the current price of this iteration
10    option.value = price;
11    // sets the textContent attribute of each option element to a string containing \textsterling and
    the current price
12    option.textContent = `£${price}`;
13    // appends the option as a child element to the select element called minPriceSelect
14    priceSelect.appendChild(option);
15  }
16 }
17
18 // This function stops the max price selected being less than the min price selected
19 export function updateMaxPriceRange(
20   minPriceSelect,
21   maxPriceSelect,
22   min,
23   max,
24   step
25 ) {
26   minPriceSelect.addEventListener('change', () => {
27     // sets selectedMin variable to a float of the current minPriceSelect or just the min (0) value
28     const selectedMin = parseFloat(minPriceSelect.value) || min;
29
30     // sets maxPriceSelect to the following html
31     maxPriceSelect.innerHTML =
32       '<option value="0" selected="selected">-</option>';
33     // for loop sets the starting value to whatever the selectedMin value is. Remaining option elements
    are set by
34     // looping from selectedMin value to max value incrementing by the step value on each iteration
35     for (let price = selectedMin; price <= max; price += step) {
36       const option = document.createElement('option');
37       option.value = price;
38       option.textContent = `£${price}`;
39       maxPriceSelect.appendChild(option);
40     }
41   });
42 }

```

Listing 3: Price Range Helper JavaScript

```

1 import {
2   populatePriceOptions,
3   updateMaxPriceRange,
4 } from './helpers/pricedropdown.js';
5 import { validateAlphaNumeric } from './helpers/validatemethods.js';
6
7 // Assigns select elements on html page to variables in order to append option elements
8 const minPriceSelect = document.getElementById('getbyminprice');
9 const maxPriceSelect = document.getElementById('getbymaxprice');
10
11 // Variables for defining the min and max price of price select dropdowns plus the step/increment value
12 const min = 100;
13 const max = 5000;
14 const step = 100;
15
16 // Calls the populatePriceOptions method from the pricedropdown.js module to set the option elements
   // for min and max price select elements
17 populatePriceOptions(minPriceSelect, min, max, step);
18 populatePriceOptions(maxPriceSelect, min, max, step);
19
20 // Updates maxPriceSelect minimum option value when minPriceSelect option is selected
21 updateMaxPriceRange(minPriceSelect, maxPriceSelect, min, max, step);

```

Listing 4: Main Search Number Plates JavaScript

The remaining child element of the 'advancedsearch' div is a checkbox for selecting whether to search for available or unavailable number plates. The other div tag within the form element displays the text 'Waiting...' when the page first loads, and is replaced by a table containing the results of the search query once the search request and response loop is complete. The JavaScript below demonstrates how form data is collected, posted to the REST API endpoint, and returned to the frontend for display via the generated HTML table element. A key aspect of the table generation is the use of a ternary operator to set whether the user can purchase a number plate. When set to true, an anchor tag provides a link to the page where a user can purchase the selected number plate, with the ID of the number plate object added to the URL.

```

1 // Handler for setting whether to carry out an advanced search or a standard reg only search
2 export function searchHandler() {
3   const data = getFormData();
4   if (data.regNumber.length > 0) {
5     const validation = validateAlphaNumeric(data.regNumber);
6     if (!validation.valid) {
7       window.alert(validation.message);
8       return;
9     }
10  }
11
12  if ($('#advancedsearch').is(':visible')) {
13    advancedSearch(data);
14  } else {
15    getPlates(data.regNumber);
16  }
17 }
18
19 // Called to collect user inputs from the html form and return key value pairs
20 export function getFormData() {
21   // create JSON body
22   return {
23     regNumber: $('#getregnumber-text').val().trim(),
24     isAvailable: $('#getbyavailable-checkbox').is(':checked'),
25     minPrice: parseFloat($('#getbyminprice').val()),
26     maxPrice: parseFloat($('#getbymaxprice').val()),
27   };
28 }
29
30
31
32
33
34 // Continued...
35 // function to handle the advanced search criteria and send to numberplate end point
36 export function advancedSearch(formData) {
37   // post form data in order to retrieve number plate search results
38   $.ajax({
39     type: 'POST',
40     url: 'http://localhost:8080/numberplate/advancedsearch',
41     data: JSON.stringify(formData),
42     contentType: 'application/json; charset=utf-8',
43     dataType: 'json',
44     success: getResults,
45     error: (req, status, error) => {
46       window.alert('Failed to find number plate: ' + error);
47     },
48   });
49 }
50
51 // Creates a table for displaying the returned results
52 export function getResults(json, status, req) {
53   let html = '';
54
55   if (json.length === 0) {
56     html = '<p>No results found.</p>';
57   } else {
58     // Create a table to display the results
59     html =

```

```

60 " <table class='table table-striped'><thead><tr><th>Number Plate</th><th>Available</th><th>Price (
    f):</th><th>Purchase</th></thead><tbody>";
61 // most import td element contains the item.available check along with an anchor tag.
62 // If item is available the anchor tag will take you to the purchasenumberplates.html
63 // and place the id of the number plate into the url. If it's not available then N/A will be
    displayed
64 json.forEach(function (item) {
65     html += '<tr>
66             <td class='plate'>${item.regNumber}</td>
67             <td>${item.available ? 'Yes' : 'No'}</td>
68             <td>${item.price}</td>
69             <td>${
70                 item.available
71                 ? '<a href='purchasenumberplates.html?id=${item.id}'>Purchase</a>'
72                 : 'N/A'
73             }</td>
74             </tr>';
75     });
76     html += '</tbody></table>';
77 }
78 // replaces the div with the generated html table
79 $('#getbyregnumberresults').html(html);
80 }

```

Listing 5: Form Data Collection and AJAX POST

The last part of the HTML to examine is the ‘head’ element, where UTF-8 character encoding provides access to all Unicode characters, including symbols, emojis, and characters from many languages. The ‘title’ element sets the name displayed in the browser tab, and the ‘link’ elements provide access to three separate CSS stylesheets, where ‘table.css’ sets the styling for the table used to display any returned data, ‘page.css’ sets the general layout and style of the page and ‘plate.css’ sets the style of the registration number text box to resemble a UK number plate. The CSS for the number plate and registration font are below, along with an image of the search field.

```

1 .plate {
2     min-width: 120px;
3     max-width: 100%;
4     border-radius: 5px;
5     box-shadow: inset 0px 0px 0px 1px #000,0px 2px 4px 0px rgba(0,0,0,0.5);
6     background: #FBDA06;
7     color: black;
8     text-align: center;
9     padding: 4px 8px;
10    line-height: 1.1;
11 }
12
13 .font-reg, .plate {
14     font-family: sans-serif;
15     font-weight: bold;
16     text-transform: uppercase;
17 }

```

Listing 6: CSS for Number Plate and Registration Font

SEARCH FOR A NUMBER PLATE

REGISTRATION:

Figure 1: Search Number Plate Input Field

The code snippet below highlights the remaining functionality of the ‘searchnumberplates.js’ file. The comments explain the purpose of each block of code.

```
1 // Scripts for each button on the page
2 $(function () {
3     $.ajaxSetup({ cache: false });
4     // Script for search button
5     $('#ajaxgetbyregnumber-button').on('click', function (e) {
6         // if advanced search fields are visible then run the AdvancedSearch() function otherwise run a the
7         // standard GetPlates()
8         searchHandler();
9         e.preventDefault();
10        return false;
11    });
12    // Script for displaying advanced search criteria when button clicked
13    $('#ajaxadvancedsearch-button').on('click', function (e) {
14        showAdvanced();
15        e.preventDefault();
16        return false;
17    });
18
19    // Script for resetting search results and search criteria fields
20    $('#reset-button').on('click', function (e) {
21        resetSearch();
22        e.preventDefault();
23        return false;
24    });
25 });
26
27 // When called, using the form id tag readdata-form, this function uses reset() to reset the form
28 // fields, creates a new 'change' event to reset maximum price select option
29 // resets the results html to 'Waiting...'
30 export function resetSearch() {
31     document.getElementById('readdata-form').reset();
32     minPriceSelect.dispatchEvent(new Event('change'));
33     // returns search results to initial state
34     $('#getbyregnumberresults').html('Waiting...');
35 }
36
37 // Returns all plates or plates based on a simple reg number check
38 export function getPlates(regNumber) {
39     // Checks if reg number is blank, if so then uses the numberplate api get method that retrieves all
40     // number plates
41     if (regNumber === '') {
42         console.log('GetAllPlates called');
43         $.ajax({
44             type: 'GET',
45             url: 'http://localhost:8080/numberplate',
46             data: '',
47             contentType: 'application/json; charset=utf-8',
48             dataType: 'json',
49             success: getResults,
50             error: (req, status, error) => window.alert('AJAX error'),
51         });
52     } else {
53         // if reg number has a value then pass value to search end point and retrieve all number plates
54         // with a partial or full reg match
55         console.log('GetPlates called with regNumber: ' + regNumber);
56         $.ajax({
57             type: 'GET',
58             url: 'http://localhost:8080/numberplate/search/${regNumber}',
59             data: '',
60             contentType: 'application/json; charset=utf-8',
61             dataType: 'json',
62             success: getResults,
63             error: (req, status, error) => window.alert('AJAX error'),
64         });
65     }
66 }
```

Listing 7: Remaining Functionality of Search Number Plates JavaScript

The code snippets below highlight a section of the ‘validatemethods.js’ helper, which contains the method for checking that a text field only contains alphanumeric characters and the associated regex pattern imported from the ‘regexpatterns.js’ helper.

```

1 // Validates that the input data contains only alphanumeric characters and a single space after the
  first batch of letters or numbers
2 export function validateAlphaNumeric(data) {
3   // Checks if data matches the regex pattern
4   if (!alphaNumericRegexPattern.test(data)) {
5     // If not, returns an object with valid set to false and a message
6     return { valid: false, message: 'Please enter only numbers or letters.' };
7   } else {
8     // If it does match, returns an object with valid set to true
9     return { valid: true };
10  }
11 }

```

Listing 8: Text Input Validation Method

```

1 // Alpha-numeric regex pattern including a single space after first batch of letters or numbers
2 export const alphaNumericRegexPattern = /^[a-zA-Z0-9]+(?: [a-zA-Z0-9]+)?$/;

```

Listing 9: Regex Pattern for Alphanumeric Characters

As detailed in the application instructions, once the user selects a number plate to purchase, a guest checkout form is generated. The HTML is not too dissimilar to the search page, but the JavaScript ‘purchasenumplates.js’ contains several functions that provide different functionality. The code below demonstrates how the application extracts the number plate object ID from the URL and retrieves the corresponding number plate registration to display on the page for the user.

```

1 // Retrieves the id of the number plate from the url when the DOM has loaded
2 document.addEventListener('DOMContentLoaded', () => {
3   const params = new URLSearchParams(window.location.search);
4   id = params.get('id');
5   if (!id) {
6     alert('No ID provided in the URL.');
```

```

7   return;
8 }
9
10 // find reg number in db by using id in url
11 axios
12   .get('http://localhost:8080/numberplate/${id}')
13   .then((response) => {
14     // sets the reg number as the value for an input element that is set to read only
15     document.getElementById('getregnumber-text').value =
16       response.data.regNumber;
17   })
18   .catch((error) => {
19     console.error('Error fetching number plates:', error);
20     window.alert('AJAX error');
21   });
22 });

```

Listing 10: Extracting Number Plate ID from URL

The code below handles the purchase request and performs the various text input validations required to create a successful purchase request.

```

1 // Called from purchase button and handles all stages of the purchase including form data validation
2 export function addPurchaseHandler() {
3   // Collects form data from the DOM
4   const data = getFormdata();
5   // Validates the form data using a series of validation functions
6   const validations = [
7     () => validateForm(data),
8     () => validateFirstName(data.firstName),
9     () => validateLastName(data.lastName),
10    () => validateEmail(data.email),
11    () => validatePhone(data.phone),

```

```

12     () => validatePostcode(data.postcode),
13 ];
14 let allValid = true;
15 // Loops through each validation function and checks if the result is valid
16 for (const validate of validations) {
17     // Calls the validate function and checks if the result is valid
18     const result = validate();
19     // If the result is not valid, alerts the user and sets allValid to false
20     if (!result.valid) {
21         // Displays the error message in an alert
22         window.alert(result.message);
23         allValid = false;
24         break;
25     }
26 }
27 // If all validations are valid, posts the data to the backend
28 if (allValid) {
29     postData(data);
30 }
31 }

```

Listing 11: Purchase Request and Input Validation

4.3 Testing

Several unit tests cover elements of both the frontend and backend. The frontend tests cover all three helper JavaScript modules as well as the ‘addnumberplates.js’ module. The tests utilise both Jest and Sinon JavaScript testing libraries, and in total, there are 25 tests across four test suites. The backend tests cover the service and controller layers and utilise the Junit testing framework. Appendix Two includes annotated code for all frontend tests, and Appendix Three displays the coverage reports for the same tests. The unit tests in Appendix Two verify the reliability of the frontend helper modules, ensuring the correct handling of user inputs and that the date range and price range validation work as expected. Additionally, the price range test covers the logic used to create the dropdown price list. The add number plate form test ensures the module correctly posts user form data and handles the response accordingly.

Date Range Helper tests verify that date input fields update correctly and prevent invalid ranges. The test path is as follows:

1. Create two date input elements (fromDateInput, toDateInput).
2. Attach them to the DOM and apply the setDateRange function.
3. Change the value of one input and dispatch a change event.
4. Assert that the other input’s min/max value updates accordingly.
5. Test that if the date range becomes invalid (e.g., from date after to date), the invalid field clears.

Price Range Helper tests ensure price dropdowns are populated and update correctly. The test path is as follows:

1. Create two select elements (minPriceSelect, maxPriceSelect).
2. Populate both with price options using populatePriceOptions.
3. Assert that the options are correct (values and text).
4. Change the min price and dispatch a change event.

5. Assert that the max price dropdown updates so its minimum matches the selected min price.

Validate Methods Helper tests check that all form validation functions (for names, email, phone, postcode, registration number, and alphanumeric input) correctly identify valid and invalid inputs, returning appropriate messages. Here are the steps taken by the test:

1. Call each validation function with invalid input and assert it returns `valid: false, message: ...`.
2. Call each validation function with valid input and assert it returns `valid: true`.
3. Test the overall form validation with missing and complete data.
4. Test specific field validations (first name, last name, email, phone, postcode, registration number, alphanumeric input) with both valid and invalid examples.

Add Number Plates test form data collection and AJAX submission logic using mocks and stubs and confirm responses are handled correctly. Here are the steps taken by the test:

1. Mock jQuery's AJAX method and the browser's alert function.
2. Test `getFormData` by setting up mock input elements and asserting the returned data matches expectations.
3. Test successful AJAX submission:
 - (a) Stub AJAX to call the success callback with a mock response.
 - (b) Assert the stub is called once
 - (c) Assert the correct URL, type and data are passed to AJAX.
 - (d) Assert the passed data matches the expected form data.
 - (e) Assert that alert is called with the correct success message.
4. Test failed AJAX submission:
 - (a) Stub AJAX to call the error callback with a mock error.
 - (b) Call `postData` and assert that alert is called with the correct error message.

5 Conclusion

To conclude, this essay has compared and contrasted client-side and server-side scripting languages, explored the benefits and drawbacks of the Node.js framework, and provided an overview of a small web application developed using the Node.js framework and AJAX HTTP requests. The essay has highlighted the key differences between client-side and server-side scripting, including their roles in web development, the languages used, and the functionality they provide. It has also discussed the benefits of using Node.js for full-stack development, such as its non-blocking architecture, single language for both client and server, and access to a vast package ecosystem. Finally, the essay has provided an overview of a number plate purchasing application, demonstrating how client-side and server-side scripting can work together to create a dynamic and interactive web application.

References

- [1] J. K. Ousterhout, “Scripting: higher level programming for the 21st Century,” *Computer*, vol. 31, no. 3, pp. 23–30, Mar. 1998.
- [2] J. Aycock, “A brief history of just-in-time,” *ACM Computing Surveys*, vol. 35, no. 2, pp. 97–113, Jun. 2003.
- [3] S. Nanz and C. A. Furia, “A Comparative Study of Programming Languages in Rosetta Code,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, May 2015, pp. 778–788.
- [4] Coffee, Peter, “‘Scripted’ Development Answers; Find the right time and place for scripting languages in the enterprise,” *eWeek*, vol. 22, no. 31, pp. 1–D2, Aug 08 2005, copyright - Copyright (c) 2005 Ziff Davis Media Inc. All Rights Reserved. Originally appearing in eWeek; Document feature - photographs; Last updated - 2024-08-22. [Online]. Available: <https://www.proquest.com/trade-journals/scripted-development-answers-find-right-time/docview/198559498/se-2>
- [5] L. Musthaler, “Client/server: What does that really mean?” *Managing Office Technology*, vol. 40, no. 6, p. 20, 06 1995, copyright - Copyright Penton Publishing Jun 1995; Last updated - 2024-11-19; SubjectsTermNotLitGenreText - US. [Online]. Available: <https://www.proquest.com/trade-journals/client-server-what-does-that-really-mean/docview/233491086/se-2>
- [6] 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.
- [7] N. Gershenfeld, “Client-server: What is it and are we there yet?” *Online*, vol. 19, no. 2, p. 60, 03 1995, copyright - Copyright Online, Incorporated Mar 1995; Last updated - 2024-12-06; CODEN - ONLIDN; SubjectsTermNotLitGenreText - Word Processing; Electronic Mail; Workstations; Computer Storage Devices; Computer Peripherals; Computer Networks; Reaction Time; Client Server Architecture. [Online]. Available: <https://www.proquest.com/trade-journals/client-server-what-is-are-we-there-yet/docview/199922003/se-2>
- [8] T. Andrews, “Scripting: Server-side vs. client-side,” *Active Server Developer’s Journal*, vol. 2, no. 9, pp. 15–16, 09 1998, copyright - Copyright Cobb Group Sep 1998; Last updated - 2024-08-24; SubjectsTermNotLitGenreText - United States; US. [Online]. Available: <https://www.proquest.com/trade-journals/scripting-server-side-vs-client/docview/206870773/se-2>
- [9] L. D. Paulson, “Building rich web applications with Ajax,” *Computer*, vol. 38, no. 10, pp. 14–17, Oct. 2005.
- [10] A. Fedorchuk, O. Usata, and O. Nakonechna, “WEB DESIGN AND WEB PROGRAMMING IN THE MODERN INTERNET WORLD,” *Municipal economy of cities*, vol. 6, no. 180, pp. 12–20, Dec. 2023.
- [11] J. Ye, C. Zhang, L. Ma, H. Yu, and J. Zhao, “Efficient and Precise Dynamic Slicing for Client-Side JavaScript Programs,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Mar. 2016, pp. 449–459.

- [12] J. D. Scarsbrook, M. Utting, and R. K. L. Ko, “TypeScript’s Evolution: An Analysis of Feature Adoption Over Time,” in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2023, pp. 109–114.
- [13] G. Roberts, “Learning Server-Side Scripting,” *Computers in Libraries*, vol. 25, no. 8, pp. 37–39, 09 2005, copyright - Copyright Information Today, Inc. Sep 2005; Last updated - 2024-12-06; SubjectsTermNotLitGenreText - Computer Literacy; Librarians; Learning Experience; Educational Technology; Management Systems; Syntax; Computer Oriented Programs; Coding; Programming; Library Personnel. [Online]. Available: <https://www.proquest.com/trade-journals/learning-server-side-scripting/docview/231140399/se-2>
- [14] H. Yu and Z. Lu, “How to Choose Server-Side Scripting Languages & Database Servers in Web Mining,” in *2008 International Symposium on Knowledge Acquisition and Modeling*. IEEE, Dec. 2008, pp. 98–102.
- [15] 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.
- [16] T. Ogasawara, “Workload characterization of server-side JavaScript,” in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Oct. 2014, pp. 13–21.
- [17] O. C. Novac, D. Ghiurău, M. C. Novac, C. E. Gordan, M. Oproescu, G. Bujdoso, and V. O. Mihalca, “Comparison of Node.Js and Spring Boot in Web Development,” in *2023 15th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*. IEEE, Jun. 2023, pp. 1–7.
- [18] 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.
- [19] J. Latendresse, S. Mujahid, D. E. Costa, and E. Shihab, “Not All Dependencies are Equal: An Empirical Study on Production Dependencies in NPM,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22. ACM, Oct. 2022, pp. 1–12.

6 Appendix One: User Manual

6.1 Running from Docker

1. Install Docker Desktop from the official website.
2. Download the zip folder containing the application files.
3. Extract the zip folder to a suitable location on your computer.
4. Open your preferred IDE (e.g., Visual Studio Code, IntelliJ IDEA).
5. Open the root folder titled ‘numberplateapp’ in your IDE.
6. Open the terminal or command line interface (CLI) within your IDE.

7. Navigate to the root folder of the application using the command `'cd numberplateapp'`.
8. Run the command `'docker compose up --build'` to build and start the application services.
9. Wait for the three services (frontend, backend, and database) to start running. You should see logs indicating that the services are up and running.
10. Open a web browser and navigate to `http://localhost:3000` to access the application.
11. You can now use the application to search for and purchase number plates, search for previous purchases, and add new number plates.

6.2 Running without Docker

1. Ensure you have Java Development Kit (JDK) and Node.js installed on your machine.
2. Download the zip folder containing the application files.
3. Extract the zip folder to a suitable location on your computer.
4. Open your preferred IDE (e.g., Visual Studio Code, IntelliJ IDEA - recommended for backend).
5. Open the root folder titled `'numberplateapp'` in your IDE.
6. Navigate to the backend folder: `'numberplateapp/backend/NumberPlateRest'`.
7. Open the `'application.properties'` file located in `'src/main/resources'`.
8. Copy the database connection URL from the `'docker-compose.yml'` file located in the root folder and paste it into the `'application.properties'` file, ensuring it references your local database connection string (e.g. `'jdbc:postgresql://localhost:5432/numberplatedata'`)*.
9. Add the database username and password to the `'application.properties'` file.
10. Ensure your local database is running and accessible.
11. Open a terminal or command line interface (CLI) within your IDE.
12. Change directory to the backend folder using the command `'cd numberplateapp/backend/NumberPlateRest'`.
13. Run the command `'./mvnw clean package'` to build the backend application.
14. After the build completes, run the command `'java -jar target/app-name-version.jar'` (replace `'app-name-version'` with the actual name of the generated .jar file in the `'target'` folder).
15. Open a new terminal or command line interface (CLI) within your IDE.
16. Navigate to the frontend folder: `'numberplateapp/frontend/nodewebserver'`.
17. Delete the `'node_modules'` folder and the `'package-lock.json'` file if they exist.
18. Run the command `'npm i'` to install the required Node.js packages.
19. After the installation completes, run the command `'npm start'` to start the frontend application.

20. Open a web browser and navigate to `http://localhost:3000` to access the application.
21. You can now use the application to search for and purchase number plates, search for previous purchases, and add new number plates.

* To improve maintainability, the `application.properties` file currently references the Spring data-source URL, username and password values set in the `docker-compose.yml`, found in the root folder of the application.

6.3 Using the Application

The application offers four main features: searching and purchasing number plates, searching for previous purchases, and adding new number plates.

6.3.1 Searching for Number Plates

1. Open the application in a web browser by navigating to `http://localhost:3000`.
2. On the home page, click the ‘Search Number Plates’ link to open the number plate search page.
3. In the search field, enter the registration number of the number plate you wish to find. You can enter a partial match, and the search will return any number plates that contain the entered text. The field accepts up to eight alphanumeric characters.
4. If you leave the search field blank and click the ‘Search’ button, the application will return all number plates currently stored in the database.
5. Use the ‘Advanced Search’ button to access additional search criteria, such as an ‘Available’ tick box to filter results by availability, and min/max price dropdown menus to filter results by price range.
6. Set the minimum price to adjust the range of available prices in the maximum price dropdown menu.
7. Click the ‘Reset’ button to clear all search criteria and remove any returned results.
8. The search results will display a list of number plates matching your criteria, including their registration numbers, availability status, and prices.
9. To purchase a found and available number plate, click the ‘Purchase’ link at the end of the row for that number plate. This will direct you to the number plate purchase page.

6.3.2 Purchasing Number Plates

1. On the number plate purchase page, you will see a read-only text field displaying the registration number of the number plate you are about to purchase.
2. Below the registration number, there is a guest checkout form with several text input fields, including first name, last name, email, phone number, and postcode.
3. Fill in all the required fields in the guest checkout form. Each field must have an input; any missing fields will result in a validation error, and the purchase will not complete.

4. Ensure that the input types for first name, last name, email, phone number, and postcode are valid.
5. Once all validation passes, click the 'Purchase' button to complete the purchase.
6. If the purchase is successful, a modal will display a message confirming that the purchase was successful, along with the registration number of the purchased number plate.
7. If an error occurs during the purchase, a modal will display an error message.
8. Upon a successful purchase, clicking 'OK' on the modal will redirect you to the search page, where you can see the updated list of available number plates.
9. If you encounter an error, clicking 'OK' on the error modal will return you to the purchase page, allowing you to correct any issues and attempt the purchase again.
10. To return to the search page without making a purchase, click the link provided on the purchase page.

6.3.3 Searching Previous Purchases

1. On the home page, click the 'Search Purchases' link to open the search purchases page.
2. On the search purchases page, you will see several text input fields, including registration number, first name, last name, email, and a date range selector with 'from date' and 'to date' fields.
3. Fill in any of the text input fields to filter your search. You can enter partial matches for the registration number, first name, last name, and email fields.
4. Use the date range selector to specify a date range for your search. Ensure that the 'from date' is earlier than the 'to date'.
5. Optionally, use the min/max price dropdown menus to filter results by price range.
6. Click the 'Search' button to submit your search criteria.
7. The application will return a list of previous purchases that match your search criteria, including the registration number, first name, last name, email, purchase date, and price.
8. If you leave all fields blank and click the 'Search' button, the application will return all previous purchases stored in the database.
9. If you wish to reset the search criteria, click the 'Reset' button, which will clear all fields and dropdown menus.
10. To return to the home page, click the link provided on the search purchases page.

6.3.4 Adding Number Plates

1. On the home page, click the ‘Add Number Plate’ link to open the add number plate page.
2. On the add number plate page, you will see a text input field for the registration number, a dropdown menu for availability (with options ‘true’ or ‘false’), a number input field for the price, and a button titled ‘Add Number Plate’.
3. Fill in the registration number field with a valid UK DVLA registration pattern (e.g., AB12 CDE) and enter a price greater than zero in the price field.
4. If you leave the registration number field blank or enter a price less than or equal to zero, a modal warning will appear, informing you that the registration field cannot be blank or that the price must be greater than zero.
5. If the registration number does not match a valid UK DVLA registration pattern or if the combination and sequence of numbers and letters match any number plate registration numbers currently held in the database, a validation error will occur, and a modal will inform you of the issue.
6. Once the details pass validation, click the ‘Add Number Plate’ button to submit the form.
7. If the addition is successful, a modal will display a success message confirming that the number plate has been added, along with the registration number and availability status.
8. If an error occurs during the addition, a modal will display an error message.
9. Upon a successful addition, the new number plate will appear in the number plate search and purchase page and if the availability is set to true, it will be available for purchase.
10. If you encounter an error, clicking ‘OK’ on the error modal will return you to the add number plate page, allowing you to correct any issues and attempt to add the number plate again.
11. To return to the home page without adding a number plate, click the link provided on the add number plate page.

7 Appendix Two: Test Code

7.1 Frontend Tests

7.1.1 Date Range Helper Tests

```
1 import { setDateRange } from '../daterange.js';
2
3 // Tests below check that changing date values updates earliest and latest available dates depending on
4 // which element has been changed
5 describe('setDateRange', () => {
6   let fromDateInput;
7   let toDateInput;
8
9   beforeEach(() => {
10     fromDateInput = document.createElement('input');
11     toDateInput = document.createElement('input');
12     fromDateInput.type = 'date';
13     toDateInput.type = 'date';
14
15     document.body.appendChild(fromDateInput);
16     document.body.appendChild(toDateInput);
```

```

17     setDateRange(fromDateInput, toDateInput);
18   });
19   test('sets toDateInput.min when fromDateInput changes', () => {
20     fromDateInput.value = '2025-07-05';
21     const event = new Event('change');
22     fromDateInput.dispatchEvent(event);
23
24     expect(toDateInput.min).toBe('2025-07-05');
25   });
26
27   test('sets fromDateInput.max when toDateInput changes', () => {
28     toDateInput.value = '2025-07-05';
29     const event = new Event('change');
30     toDateInput.dispatchEvent(event);
31
32     expect(fromDateInput.max).toBe('2025-07-05');
33   });
34
35   test('sets toDateInput.value to null when toDateInput less than fromDateInput', () => {
36     toDateInput.value = '2025-07-05';
37     fromDateInput.value = '2025-07-06';
38     const event = new Event('change');
39     fromDateInput.dispatchEvent(event);
40
41     expect(toDateInput.value).toBe('');
42   });
43
44   test('sets fromDateInput.value to null when fromDateInput greater than toDateInput', () => {
45     toDateInput.value = '2025-07-05';
46     fromDateInput.value = '2025-07-06';
47     const event = new Event('change');
48     toDateInput.dispatchEvent(event);
49
50     expect(fromDateInput.value).toBe('');
51   });
52 });
53

```

Listing 12: Frontend Date Range Helper Test Code

7.1.2 Price Range Helper Tests

```
1
2 import { populatePriceOptions, updateMaxPriceRange } from '../pricedropdown.js';
3
4 // Tests checks price options added correctly and lowest price available to max price drop down is set
5 // to the value of the minimum price selected
6
7 describe('setPriceRange', () => {
8   let minPriceSelect;
9   let maxPriceSelect;
10  let min;
11  let max;
12  let step;
13
14  beforeEach(() => {
15    minPriceSelect = document.createElement('select');
16    maxPriceSelect = document.createElement('select');
17    min = 100;
18    max = 500;
19    step = 100;
20
21    document.body.appendChild(minPriceSelect);
22    document.body.appendChild(maxPriceSelect);
23
24    populatePriceOptions(minPriceSelect, min, max, step);
25    populatePriceOptions(maxPriceSelect, min, max, step);
26  });
27  test('sets minPriceSelect and maxPriceSelect options for select elements', () => {
28    expect(minPriceSelect.options.length).toBeGreaterThan(0);
29    expect(minPriceSelect.options[0].value).toBe('100');
30    expect(minPriceSelect.options[0].text).toBe('£100');
31    expect(minPriceSelect.options.length).toBe(5);
32    expect(maxPriceSelect.options.length).toBeGreaterThan(0);
33    expect(maxPriceSelect.options[0].value).toBe('100');
34    expect(maxPriceSelect.options[0].text).toBe('£100');
35    expect(maxPriceSelect.options.length).toBe(5);
36  });
37
38  test('sets minimum maxPriceSelect option to minPriceSelect value', () => {
39    updateMaxPriceRange(minPriceSelect, maxPriceSelect, min, max, step);
40
41    minPriceSelect.value = '300';
42    const event = new Event('change');
43    minPriceSelect.dispatchEvent(event);
44
45    expect(maxPriceSelect.options[0].value).toBe('0');
46    expect(maxPriceSelect.options[0].text).toBe('-');
47    expect(maxPriceSelect.options[1].value).toBe('300');
48    expect(maxPriceSelect.options[1].text).toBe('£300');
49  });
50 });
```

Listing 13: Frontend Price Range Helper Test Code

7.1.3 Validate Methods Helper Tests

```
1 import {
2   validateAlphaNumeric,
3   validateEmail,
4   validateFirstName,
5   validateForm,
6   validateLastName,
7   validateNumberPlate,
8   validatePhone,
9   validatePostcode,
10 } from '../validatemethods.js';
11
12 // Tests for checking all validation methods for when validation is met and not met
13
14 describe('validate methods', () => {
15   test('return object contains valid: false and a message when complete form field validation fails',
16     () => {
17     const formData = '';
18     expect(validateForm(formData)).toEqual({
19       valid: false,
20       message: 'Please complete all fields.',
21     });
22   });
23
24   test('return object contains valid: true when complete form field validation is successfull', () => {
25     const formData = {
26       firstName: 'X',
27       lastName: 'X',
28       email: 'X',
29       phone: 'X',
30       address: 'X',
31       city: 'X',
32       county: 'X',
33       postcode: 'X',
34     };
35     expect(validateForm(formData)).toEqual({ valid: true });
36   });
37
38   test('return object contains valid: false and a message when first name validation fails', () => {
39     const name = '(*&^(*&^(*&*)*(&';
40     expect(validateFirstName(name)).toEqual({
41       valid: false,
42       message: 'Please enter a valid first name.',
43     });
44   });
45
46   test('return object contains valid: true when first name validation is successfull', () => {
47     const name = 'Luke';
48     expect(validateFirstName(name)).toEqual({ valid: true });
49   });
50
51   test('return object contains valid: false and a message when last name validation fails', () => {
52     const name = '(*&^(*&^(*&*)*(&';
53     expect(validateLastName(name)).toEqual({
54       valid: false,
55       message: 'Please enter a valid last name.',
56     });
57   });
58
59   test('return object contains valid: true when last name validation is successfull', () => {
60     const name = 'Skywalker';
61     expect(validateLastName(name)).toEqual({ valid: true });
62   });
63
64   test('return object contains valid: false and a message when email validation fails', () => {
65     const email = 'emailemail.com';
66     expect(validateEmail(email)).toEqual({
67       valid: false,
68       message: 'Please enter a valid email address.',
69     });
70   });
71 });
```

```

70
71 test('return object contains valid: true when email validation is successfull', () => {
72   const email = 'email@email.com';
73   expect(validateEmail(email)).toEqual({ valid: true });
74 });
75
76 test('return object contains valid: false and a message when phone validation fails', () => {
77   const phone = 'X';
78   expect(validatePhone(phone)).toEqual({
79     valid: false,
80     message: 'Please enter a valid UK phone number.',
81   });
82 });
83
84 test('return object contains valid: true when phone validation is successfull', () => {
85   const phoneStartWithZero = '07777 777777';
86   const phoneStartWithPlus = '+447777 777777';
87   const phoneNineDigits = '0800 000 000';
88   expect(validatePhone(phoneStartWithZero)).toEqual({ valid: true });
89   expect(validatePhone(phoneStartWithPlus)).toEqual({ valid: true });
90   expect(validatePhone(phoneNineDigits)).toEqual({ valid: true });
91 });
92
93 test('return object contains valid: false and a message when postcode validation fails', () => {
94   const postcode = 'X';
95   expect(validatePostcode(postcode)).toEqual({
96     valid: false,
97     message: 'Please enter a valid UK postcode.',
98   });
99 });
100
101 test('return object contains valid: true when postcode validation is successfull', () => {
102   const postcode = 'SA1 1AA';
103   expect(validatePostcode(postcode)).toEqual({ valid: true });
104 });
105
106 test('return object contains valid: false and a message when reg number validation fails', () => {
107   const regNumber = 'X';
108   expect(validateNumberPlate(regNumber)).toEqual({
109     valid: false,
110     message: 'Not a valid UK number plate.',
111   });
112 });
113
114 test('return object contains valid: true when reg number validation is successfull', () => {
115   const regNumberStandardNoSpace = 'AA00AAA';
116   const regNumberStandardSpace = 'AA00 AAA';
117   const regNumberOldStyleLetterFirst = 'A000 AAA';
118   const regNumberOldStyleLetterFirstNoSpace = 'A000AAA';
119   const regNumberOldStyleLetterLast = 'AAA 000A';
120   const regNumberOldStyleLetterLastNoSpace = 'AAA000A';
121   expect(validateNumberPlate(regNumberStandardNoSpace)).toEqual({
122     valid: true,
123   });
124   expect(validateNumberPlate(regNumberStandardSpace)).toEqual({
125     valid: true,
126   });
127   expect(validateNumberPlate(regNumberOldStyleLetterFirst)).toEqual({
128     valid: true,
129   });
130   expect(validateNumberPlate(regNumberOldStyleLetterFirstNoSpace)).toEqual({
131     valid: true,
132   });
133   expect(validateNumberPlate(regNumberOldStyleLetterLast)).toEqual({
134     valid: true,
135   });
136   expect(validateNumberPlate(regNumberOldStyleLetterLastNoSpace)).toEqual({
137     valid: true,
138   });
139 });
140
141 test('return object contains valid: false and a message when alpha numeric validation fails', () => {

```

```

142   const specialCharacters = '*^(A96*^';
143   expect(validateAlphaNumeric(specialCharacters)).toEqual({
144     valid: false,
145     message: 'Please enter only numbers or letters.',
146   });
147 });
148
149 test('return object contains valid: true when alpha numeric validation is successfull', () => {
150   const lettersAndNumbers = 'SA1 1AA';
151   expect(validateAlphaNumeric(lettersAndNumbers)).toEqual({ valid: true });
152 });
153 });

```

Listing 14: Frontend Validate Methods Helper Test Code

7.1.4 Add Number Plates Tests

```

1 import sinon from 'sinon';
2 import jQuery from 'jquery';
3
4 let postData;
5 let getFormData;
6 let addnumberplateform;
7
8 // Use async to allow await to be used when importing the modules under test. Modules need to be
9 // imported this way to ensure the mocked jQuery object is set first otherwise the actual
10 // implementation of jQuery within the modules themselves will be used
11 beforeAll(async () => {
12   // Configure tests to work with jQuery Ajax calls
13   // Assign jQuery library to globally accessible variable $, tests won't run without this set
14   global.$ = jQuery;
15   // Import modules under test after mocking jQuery
16   addnumberplateform = await import('../addnumberplateform.js');
17   // assign methods required for testing to declared variables of same name
18   postData = addnumberplateform.postData;
19   getFormData = addnumberplateform.getFormData;
20 });
21
22 describe('create number plate using Sinon', () => {
23   // Declare ajaxStub in order to access within each test
24   let ajaxStub;
25   // Declare alertMock in order to access within each test
26   let alertMock;
27
28   beforeEach(() => {
29     // Use sinon.stub to create a stubbed function in order to mock its behaviour and test its
30     // interaction with the module
31     // and prevent actual http requests being sent
32     ajaxStub = sinon.stub($, 'ajax');
33     // Mock window.alert using jest in order to check returned messages upon success or failure
34     alertMock = jest.spyOn(window, 'alert').mockImplementation(() => {});
35   });
36
37   afterEach(() => {
38     // Restores the original implementation of the methods being mocked .restore() is from the sinon
39     // library and mockRestore() from jest
40     ajaxStub.restore();
41     alertMock.mockRestore();
42   });
43
44   test('getFormData returns data from form input', () => {
45     // Mock form input data using element id tags from actual code and passing testable data
46     document.body.innerHTML = `
47       <input id="reg-text" type="text" value="AA00 AAA"/>
48       <option id="booleanDropdown" value="true">
49       <input id="price-numberbox" type="number" value="1000"/>
50     `;
51     // call getFormData function and assign result to variable data
52     const data = getFormData();
53   });
54 }

```

```

50 // Check data returned from getFormData equals data set below
51 expect(data).toEqual({
52   regNumber: 'AA00 AAA',
53   available: 'true',
54   price: '1000',
55 });
56 });
57
58 test('should call $.ajax with form data and handle success', () => {
59   // Using ajaxStub and sinon method callsFake to return a success message and a reg number
60   ajaxStub.callsFake((options) => {
61     options.success({
62       message: 'Number plate created',
63       regNumber: 'AA00 AAA',
64     });
65   });
66
67   // Set JSON form data
68   const formData = {
69     regNumber: 'AA00 AAA',
70     available: 'true',
71     price: '1000',
72   };
73
74   // Call actual postData method passing formData set above
75   postData(formData);
76
77   // Assert that ajaxStub is called once
78   sinon.assert.calledOnce(ajaxStub);
79
80   // Extract the first argument from the first call of the stub. The argument contains the AJAX
81   // options which include type, url, data and contentType
82   const args = ajaxStub.getCall(0).args[0];
83   expect(args.type).toBe('POST');
84   expect(args.url).toBe('http://localhost:8080/numberplate');
85   expect(args.contentType).toBe('application/json; charset=utf-8');
86
87   // Extract the data option and parse to JSON assigning it to the parsedData variable
88   const parsedData = JSON.parse(args.data);
89   // Expect that the parsedData will equal the initial formData
90   expect(parsedData).toEqual(formData);
91
92   // Expect that the alertMock will be called and will return the message and regNumber data in this
93   // format with a colon between the two
94   expect(alertMock).toHaveBeenCalledWith('Number plate created: AA00 AAA');
95 });
96
97 test('should call $.ajax with form data and handle error', () => {
98   // Using ajaxStub and sinon method callsFake to return an error message and undefined error type
99   ajaxStub.callsFake((options) => {
100     options.error({
101       message: 'Failed to add number plate',
102       error: undefined,
103     });
104   });
105
106   // Set JSON form data
107   const formData = {
108     regNumber: 'AA00 ABC',
109     available: 'true',
110     price: '1000',
111   };
112
113   // Call actual postData method passing formData
114   postData(formData);
115   // Expect that the alertMock will be called and will return the message and regNumber data
116   expect(alertMock).toHaveBeenCalledWith(
117     'Failed to add number plate:' + ' ' + undefined
118   );
119 });

```

Listing 15: Frontend Add Number Plates Test Code

8 Appendix Three: Test Results

8.1 Date Range Helper Test Coverage

29/07/2025, 00:10

Code coverage report for daterange.js

All files daterange.js

100% Statements 10/10 100% Branches 8/8 100% Functions 3/3 100% Lines 10/10

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1      // Ensures data from date cannot be set after to date and vice versa
2
3      export function setDateRange(fromDateInput, toDateInput) {
4 4x    fromDateInput.addEventListener('change', () => {
5        // when fromDate changes the value set is assigned to fromDateValue
6 2x    const fromDateValue = fromDateInput.value;
7
8        // sets the minimum toDateInput to the date set by fromDateValue
9 2x    toDateInput.min = fromDateValue;
10
11 2x    if (toDateInput.value && toDateInput.value < fromDateValue) {
12 1x      toDateInput.value = '';
13    }
14  });
15
16 4x  toDateInput.addEventListener('change', () => {
17    // when toDate changes the value set is assigned to toDateValue
18 2x    const toDateValue = toDateInput.value;
19
20    // sets the maximum fromDateInput to the date set by toDateValue
21 2x    fromDateInput.max = toDateValue;
22
23 2x    if (fromDateInput.value && fromDateInput.value > toDateValue) {
24 1x      fromDateInput.value = '';
25    }
26  });
27  }
28
```

Figure 2: Date Range Helper Test Coverage

8.2 Price Dropdown and Price Range Helper Test Coverage

29/07/2025, 00:10

Code coverage report for pricedropdown.js

All files pricedropdown.js

100% Statements 15/15 50% Branches 1/2 100% Functions 3/3 100% Lines 13/13

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1      // Price drop down is used on purchases search and advanced number plate search
2
3      // sets the minimum and maximum values along with the increment between each value
4
5
6      // Creates the list of price values to appear in the drop down
7      export function populatePriceOptions(priceSelect, min, max, step) {
8          // for loop starts at minimum value and loops through to maximum value increasing by set increment step value on each iteration
9      4x      for (let price = min; price <= max; price += step) {
10         // Select elements use option elements to display values
11     20x         const option = document.createElement('option');
12         // sets the value attribute of each option element to the current price
13     20x         option.value = price;
14         // sets the textContent attribute of each option element to a string containing £ and the current price
15     20x         option.textContent = `£${price}`;
16         // appends the option as a child element to the select element called minPriceSelect
17     20x         priceSelect.appendChild(option);
18     }
19 }
20
21 // This function stops the min price selected being greater than the max price selected
22 export function updateMaxPriceRange(minPriceSelect, maxPriceSelect, max, step) {
23 1x minPriceSelect.addEventListener('change', () => {
24     // sets selectedMin variable to a float of the current minPriceSelect or just the min (0) value
25 1x     const selectedMin = parseFloat(minPriceSelect.value) || min;
26
27     // sets maxPriceSelect to the following html
28 1x     maxPriceSelect.innerHTML =
29         '<option value="0" selected="selected"></option>';
30     // for loop sets the starting value to whatever the selectedMin value is then uses same system for as used previously for
31     // adding further values
32 1x     for (let price = selectedMin; price <= max; price += step) {
33         3x         const option = document.createElement('option');
34         3x         option.value = price;
35         3x         option.textContent = `£${price}`;
36         3x         maxPriceSelect.appendChild(option);
37     }
38 }):
```

Figure 3: Price Range Helper Test Coverage

8.3 Validate Methods Helper Test Coverage

29/07/2025, 00:26

Code coverage report for validatemethods.js

All files **validatemethods.js**

100% Statements 26/26 **100%** Branches 16/16 **100%** Functions 7/7 **100%** Lines 26/26

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Figure 4: Validate Methods Helper Test Coverage

8.4 Add Number Plates Test Coverage

All files / js addnumberplateform.js

29.41% Statements 5/17 0% Branches 0/6 57.14% Functions 4/7 29.41% Lines 5/17

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1      import { validateNumberPlate } from './helpers/validatemethods.js';
2
3 1x $(function () {
4      $.ajaxSetup({ cache: false });
5      // Script for add button
6      $('#add-button').on('click', function (e) {
7          addNumberPlateHandler();
8          e.preventDefault();
9          return false;
10     });
11 });
12
13 export function getFormData() {
14     // Create JSON key value pairs
15 1x return {
16         regNumber: $('#reg-text').val(),
17         available: $('#booleanDropdown').val(),
18         price: $('#price-numberbox').val(),
19     };
20 }
21
22 // function for adding a number plate to the database
23 export function postData(data) {
24     // POST request to numberplate endpoint
25 2x $.ajax({
26         type: 'POST',
27         url: 'http://localhost:8080/numberplate',
28         data: JSON.stringify(data),
29         contentType: 'application/json; charset=utf-8',
30         dataType: 'json',
31         success: (json, status, res) =>
32 1x         window.alert(json.message + ': ' + json.regNumber),
33         error: (req, status, error) =>
34 1x         window.alert('Failed to add number plate: ' + error),
35     });
36 }
37
38 export function addNumberPlateHandler() {
39     const data = getFormData();
40     if (data.regNumber && data.regNumber.length > 0) {
```

Figure 5: Add Number Plates Test Coverage