

Analysis_of_Web_Scripting_and_ Node.js _Ryan_Evans 242520

by Ryan Evans

Submission date: 14-May-2025 11:29PM (UTC+0100)

Submission ID: 259293436

File name:

58749_Ryan_Evans_Analysis_of_Web_Scripting_and_Node.js__Ryan_Evans_242520_661685_784757898.docx
(117.17K)

Word count: 3258

Character count: 22616

Analysis of Web Scripting and Node.js

Ryan Evans
242520

Contents

Client-Side vs Server-Side Scripting	2
Client-Side Scripting	2
Server-Side Scripting	3
Evaluation of the Node.js Framework.....	4
Project Documentation: Node.js and AJAX Application	5
User Manual	5
Design Overview.....	6
Interesting Code Features	6
Middleware-Based Authentication Context.....	6
Modular route handling	7
Clear API Design and Error Logging	8
Testing and Test Evidence	9
Password Validation.....	9
Number Plate Format Validation	10
Offensive Number Plate Filter	11
Price Validation	12
VIN Validation.....	13
Registration Form Logic	14

Client-Side vs Server-Side Scripting

Client-side and server-side scripting are two fundamental approaches in web development, used to create dynamic and interactive web applications. Each has distinct advantages and disadvantages that can impact the application's appearance and performance. Each operates in its own runtime environment and serves different purposes in the application lifecycle.

Client-Side Scripting

Client-side scripting executes code within the user's browser, enhancing responsiveness and interactivity for the user. A typical example of client-side scripting is the use of JavaScript, combined with HTML to define the page structure and CSS to handle visual styling[1].

Client-side scripts are responsible for a wide range of functionality, including:

- *Input validation*: Ensuring user input (e.g. email, phone number, or number plate format) adheres to specified rules before sending it to the server [2].
- *Form handling*: Dynamically updating the interface based on user actions, such as displaying confirmation messages or highlighting required fields[3].
- *DOM manipulation*: Changing the structure, content, or styling of HTML elements in response to events like mouse clicks or key presses[4].
- *Dynamic content updates*: Using AJAX (Asynchronous JavaScript and XML) or Fetch API to retrieve and display data from the server without reloading the page[5].

Despite its many advantages, client-side scripting also has limitations. Since the code runs in the user's browser, the user's system determines the performance capacity, and complex functionality can be impacted[6]. Additionally, it is exposed and can be inspected or modified. As a result, critical operations should be handled securely on the server side[7].

Example of a client-side number plate search script:

```
$.ajax({
  url: '/api/numberplates/reserve',
  method: 'POST',
  headers: {
    'x-customer-id': customerId
  },
  contentType: 'application/json',
  data: JSON.stringify({
    numberPlateId: numberPlateId,
  }),
  success: function () {
    alert("Purchase successful!");
    $('#searchForm').trigger('submit'); // refresh list

    if(typeof loadCustomerNumberPlates === 'function') {
      loadCustomerNumberPlates();
    }
  },
  error: function (err) {
    console.error("Purchase failed", err);
    alert("Could not complete purchase.");
  }
});
```

Server-Side Scripting

Server-side scripting executes code on a web server, handling the core logic and data management responsibilities of a web application. It enables dynamic content, database communication, and ensures secure processing of sensitive operations, such as authentication and user management. A typical example of server-side scripting is the use of Node.js, along with frameworks like Express.js, to build scalable, backend APIs.

Server-side scripts are responsible for a wide range of functionality, including:

- *Dynamic page generation*: Rendering HTML content on the server based on data, user roles, or session status before sending it to the client[8].
- *Database connectivity*: Interfacing with databases (e.g. PostgreSQL, MySQL) to store, retrieve, and update records based on client requests[9].
- *Authentication and authorisation*: Validating user credentials and determining access rights to secure parts of an application[10].
- *Processing requests*: Handling form submissions, applying business logic, and returning JSON or HTML responses via RESTful APIs[11].

Despite its benefits, server-side scripting introduces increased latency compared to local scripts, as every request must travel to and from the server[12]. Additionally, server load can become a bottleneck when handling a large number of simultaneous users or transactions.

Example of server-side numberplate search script:

```
router.get('/numberplates/search', async (req, res) => {
  try {
    const response = await axios.get(`${webServiceUrl}/numberplates/search`, {
      params: req.query
    });
    res.json(response.data);
  } catch (error) {
    logAxiosError('GET /Numberplates/search', error);
    res.status(500).json({ error: 'Failed to fetch data from webservice' });
  }
});
```

Evaluation of the Node.js Framework

Node.js is a runtime environment that allows developers to execute JavaScript code outside of the browser. It is built on the V8 engine from Google Chrome and has become a popular choice for developing scalable and high-performance applications[13]. It features an extensive package library promoting rapid development and prototyping while reducing manual development of core features[14].

Advantages of Node.js

- *Efficient Asynchronous Processing*: Node.js handles I/O operations using a non-blocking and event-driven architecture, which enables handling many simultaneous requests, making it highly suited for real-time services[13].
- *Unified Development Language*: By using JavaScript for both the client and server, it simplifies the development process for developers. It promotes code reuse and simplifies data handling across the application[15].
- *Extensive Package Ecosystem*: Utilising Node Package Manager (NPM) allows developers to implement an extensive range of libraries and modules, providing standard functionality such as routing, authentication, testing and data handling without requiring manual development[14].

Disadvantages of Node.js

- *Poor Performance for CPU-Intensive Tasks*: Node.js runs on a single thread, which reduces its capabilities with heavy computation. Operations such as complex data transformation or image processing can significantly impact the performance of the application[16].
- *Complexity in Asynchronous Logic*: Although features such as async/wait have improved readability. Asynchronous logic can lead to poor code maintainability when increased complexity is required[17].
- *Variable Package Quality*: due to NPM being open-source, this can result in less than standard packages being developed with inconsistent maintenance

or poor documentation. This can lead to technical debt and security risks if implemented[14].

Project Documentation: Node.js and AJAX Application

User Manual

Set-up

Extract files

Open the project in IntelliJ

Run Docker compose up -d --build

Go to localhost:3000 on your browser of choice

Homepage

Search Bar left empty will list all number plates present in the database.

Anything present in the search bar will query for a matching element, e.g. 08 will return any numberplate containing 08)

Filter by status, min price, and max price

If logged in, **Purchase** button will appear in the Action Column

Purchase will change the numberplate status to Reserved, assign it to the customer, and remove the price

If logged in on an admin account, the add numberplate button will be displayed

Here you can add a new numberplate that is automatically listed as Available, and it will detect if the numberplate could be considered offensive

Login

You can enter login credentials, Username: User, Password: User1234, to see a user account

Create Account will allow you to create an account

You can enter login credentials, Username: Admin, Password: Admin123, to see an admin account

Once logged in, you will be redirected to the dashboard page, where you can see customer details, registered numberplates and reserved numberplates

(numberplates purchased but not registered, simulating a V778 Retention Document)

The login button will also be replaced with a logout button.

Dashboard

On the dashboard, any reserved plate can be registered to a vehicle. It will open a pop-up window, it will automatically populate the number plate, and the user can input the VIN number, type, make, model and colour.

Design Overview

The design of the application follows a modular and maintainable architecture, separating the presentation, logic, and data handling. The application uses Node.js with Express.js as the web framework and connects to a Spring Boot web service using RESTful API calls. The application follows key web development principles such as component reusability, middleware abstraction and client-server decoupling.

Architecture:

- The system follows a Model-View-Controller (MVC) pattern:
- Model: Data is retrieved from a separate backend web service (e.g., /numberplates/search)
- View: HTML files served via Express (e.g., index.html)
- Controller: Express routes handle incoming API requests and forward them using axios to a Spring Boot service

Technologies Used:

- Node.js with Express.js for routing
- Axios for server-to-server communication
- AJAX (jQuery) on the frontend for asynchronous calls
- HTML/CSS/JS for UI

Interesting Code Features

Middleware-Based Authentication Context

A custom middleware inspects the incoming request headers for a customer ID. This is cross-referenced against an in-memory cache to determine the user's authentication state. The result is injected into the res.locals object, making it accessible throughout the project.

```

server.use((req, res, next) => {
  const customerId = req.headers['x-customer-id'];

  if (customerId && loginCache.has(customerId)) {
    res.locals.isLoggedIn = true;
    res.locals.customerId = customerId;
  } else {
    res.locals.isLoggedIn = false;
    res.locals.customerId = null;
  }

  next();
});

```

Modular route handling

The project splits its routes using distinct routers, API and Index, to handle page rendering and communication between the frontend and backend using Axios.

(Example of two API routes)

```

router.get('/numberplates/search', async (req, res) => {
  try {
    const response = await axios.get(`${webServiceUrl}/numberplates/search`, {
      params: req.query
    });
    res.json(response.data);
  } catch (error) {
    logAxiosError('GET /Numberplates/search', error);
    res.status(500).json({ error: 'Failed to fetch data from webservice' });
  }
});

router.post('/login', async (req, res) => {
  try {
    const response = await axios.post(`${webServiceUrl}/customer/login`,
req.body);
    const customerId = response.data.customerId;
    const isAdmin = response.data.isAdmin

    loginCache.set(customerId, {
      lastLogin: Date.now()
    })
    loginCache.set('isAdmin', isAdmin);

    res.json(response.data);
    console.log('Login successful', response.data, response.data.customerId);
  } catch (error) {
    logAxiosError('POST /login', error);

    res.status(500).json({ error: 'Failed to fetch data from webservice' });
  }
});

```

(Example of 2 Index Routes)

```
router.get('/login', (req, res) => {
  res.render('login');
})

router.get('/customerRegistration', (req, res) => {
  res.render('customerRegistration');
})

router.get('/logout', (req, res) => {
  const customerId = req.query.customerId;
  if (customerId) {
    loginCache.del(customerId);
    console.log(`Removed customerId ${customerId} from cache`);
  } else {
    console.warn('No customerId provided in query for logout');
  }

  res.redirect('/login');
});
```

Clear API Design and Error Logging

Within the apiRouter, all requests to the backend are wrapped in try/catch blocks with detailed logging (via logAxiosError). The responses are handled and errors are surfaced with informative messages.

```
function logAxiosError(context, error) {
  console.error(`Error calling ${context}:`, {
    message: error?.message || 'Unknown error',
    code: error?.code || 'N/A',
    config: error?.config ? {
      method: error.config.method,
      url: error.config.url,
      headers: error.config.headers,
      data: error.config.data
    } : null,
    response: error?.response ? {
      status: error.response.status,
      data: error.response.data
    } : null
  });
}

module.exports = logAxiosError;
```

```
router.get('/numberplates/search', async (req, res) => {
  try {
    const response = await axios.get(`${webServiceUrl}/numberplates/search`, {
      params: req.query
    });
    res.json(response.data);
  } catch (error) {
    logAxiosError('GET /Numberplates/search', error);
    res.status(500).json({ error: 'Failed to fetch data from webservice' });
  }
});
```

Testing and Test Evidence

Jest was used to implement and run automated unit tests for client-side JavaScript functions. The following sections detail the test cases implemented for various user input validation features, including number plate validation, password confirmation, offensive word detection, price checks, VIN structure validation, and registration form logic.

Password Validation

This validates the logic for users entering matching passwords when registering. It includes checks for matching values, trimming of whitespace, and detecting mismatched entries.

Test Case	Description	Expected Result	Status
Passwords match	Returns true when both passwords are identical	True	Passed
Passwords do not match	Returns false if passwords differ	False	Passed
Whitespace handling	Trims inputs before comparing	True	Passed

```
function passwordsMatch(password, confirmPassword) {
  return password.trim() === confirmPassword.trim();
}

describe('Password Validation', () => {
  test('should return true when passwords match', () => {
    expect(passwordsMatch('secret123', 'secret123')).toBe(true);
  });

  test('should return false when passwords do not match', () => {
    expect(passwordsMatch('secret123', 'Secret321')).toBe(false);
  });

  test('should trim whitespace before comparing', () => {
    expect(passwordsMatch(' secret ', 'secret')).toBe(true);
  });
});
```

Number Plate Format Validation

This validates that UK-style number plates entered follow the expected format (e.g. AB12 XYZ). It handles both spaced and unspaced input, case conversion, and rejects incorrect patterns.

Test Case	Description	Expected Result	Status
Valid with space	Standard format like AB12 XYZ	True	Passed
Valid without space	Compressed format like AB12XYZ	True	Passed
Lowercase accepted	Converted to uppercase and validated	True	Passed
Malformed patterns	Too few/many characters or wrong structure	False	Passed
Trims input	Handles extra spaces	True	Passed
Special characters	Rejects symbols or non-alphanumeric characters	False	Passed

```
function isValidNumberPlate(plate) {
  return /^[A-Z]{2}[0-9]{2}\s?[A-Z]{3}$/.test(plate.trim().toUpperCase());
}

describe('UK number plate format validation', () => {
  test('accepts valid number plate with space', () => {
    expect(isValidNumberPlate('AB12 XYZ')).toBe(true);
  });

  test('accepts valid number plate without space', () => {
    expect(isValidNumberPlate('AB12XYZ')).toBe(true);
  });

  test('rejects lowercase input', () => {
    expect(isValidNumberPlate('ab12 xyz')).toBe(true); // valid after uppercase
  });

  test('rejects plate with incorrect format', () => {
    expect(isValidNumberPlate('A1 XYZ')).toBe(false); // not 2 digits
    expect(isValidNumberPlate('ABC12 XYZ')).toBe(false); // too many letters
  });

  test('rejects plate with too many digits', () => {
    expect(isValidNumberPlate('AB123 XYZ')).toBe(false); // too many digits
  });

  test('rejects plate with too few letters', () => {
    expect(isValidNumberPlate('AB12 XY')).toBe(false); // too few letters
  });

  test('trims and validates correctly', () => {
    expect(isValidNumberPlate(' AB12 XYZ ')).toBe(true);
  });

  test('rejects special characters or symbols', () => {
    expect(isValidNumberPlate('AB12@XYZ')).toBe(false);
  });
});
```

Offensive Number Plate Filter

This detects and blocks offensive content when adding number plates, when encoded using leetspeak. It uses normalisation and a banned word list to scan input securely.

Test Case	Description	Expected Result	Status
Direct match	Blocks clearly offensive words	True	Passed
Leetspeak variations	Detects encoded offensive words	True	Passed
Clean plates	Ignores harmless inputs	False	Passed
Case insensitivity	Matches regardless of letter case	True	Passed
False positives avoided	Does not block innocent patterns	False	Passed

```
const bannedWords = ['fuck', 'shit', 'bitch', 'cunt', 'twat', 'prick', 'wank', 'cock'];

function normalisePlate(plate) {
  return plate
    .toLowerCase()
    .replace(/4/g, 'a')
    .replace(/3/g, 'e')
    .replace(/1/g, 'i')
    .replace(/0/g, 'o')
    .replace(/5/g, 's')
    .replace(/7/g, 't')
    .replace(/[^a-z]/g, ''); // remove non-letters
}

function isOffensiveNumberPlate(plate) {
  const normalised = normalisePlate(plate);
  return bannedWords.some(word => normalised.includes(word));
}

describe('Offensive number plate filter', () => {
  test('blocks exact match to offensive word', () => {
    expect(isOffensiveNumberPlate('WANK')).toBe(true);
  });

  test('blocks leetspeak variations', () => {
    expect(isOffensiveNumberPlate('5H1T')).toBe(true); // shit
    expect(isOffensiveNumberPlate('F4CK')).toBe(true); // fuck
    expect(isOffensiveNumberPlate('C0CK')).toBe(true); // cock (if added)
  });

  test('ignores clean plates', () => {
    expect(isOffensiveNumberPlate('AB12 XYZ')).toBe(false);
    expect(isOffensiveNumberPlate('C00L')).toBe(false);
  });

  test('is case insensitive', () => {
    expect(isOffensiveNumberPlate('TwAt')).toBe(true);
  });

  test('ignores numbers/letters if not offensive', () => {
    expect(isOffensiveNumberPlate('T3ST PL4T3')).toBe(false);
  });
});
```

Price Validation

This Validates that prices are valid numbers within a specified range (£150–£1000 by default). It supports numeric strings and trims whitespace where applicable.

Test Case	Description	Expected Result	Status
Valid price	Value falls within £150–£1000	True	Passed
Below minimum	Rejects value under £150	False	Passed
Above maximum	Rejects value over £1000	False	Passed
Non-numeric input	Rejects invalid strings	False	Passed
Numeric string	Parses and accepts string values	True	Passed
Trims input	Handles spaces around number	True	Passed

```
function isValidPrice(price, min = 150, max = 1000) {
  const parsed = parseFloat(price);
  if (isNaN(parsed)) return false;
  return parsed >= min && parsed <= max;
}

describe('Price Validation', () => {
  test('accepts a valid price within range', () => {
    expect(isValidPrice(250)).toBe(true);
  });

  test('rejects a price below the minimum', () => {
    expect(isValidPrice(100)).toBe(false);
  });

  test('rejects a price above the maximum', () => {
    expect(isValidPrice(15000)).toBe(false);
  });

  test('rejects non-numeric input', () => {
    expect(isValidPrice("abc")).toBe(false);
  });

  test('accepts numeric string', () => {
    expect(isValidPrice("199.99")).toBe(true);
  });

  test('trims and parses numeric strings', () => {
    expect(isValidPrice(" 300.50 ")).toBe(true);
  });
});
```

VIN Validation

This Validates that the VIN input consists of 17 alphanumeric characters, excluding the letters I, O, and Q. It rejects lowercase and special characters and trims the input before validation.

Test Case	Description	Expected Result	Status
Valid VIN	17-character alphanumeric code	True	Passed
Too short	Fewer than 17 characters	False	Passed
Too long	More than 17 characters	False	Passed
Disallowed characters	Rejects I, O, Q	False	Passed
Lowercase letters	Only uppercase is valid	False	Passed
Symbols	Rejects special characters	False	Passed
Trims input	Whitespace is ignored in check	True	Passed

```
function isValidVin(vin) {
  return /^[A-HJ-NPR-Z0-9]{17}$/.test(vin.trim());
}

describe('VIN validation', () => {
  test('valid VIN passes', () => {
    expect(isValidVin('1HGCM82633A004352')).toBe(true);
  });

  test('fails if too short', () => {
    expect(isValidVin('1HGCM82633A00435')).toBe(false); // 16 chars
  });

  test('fails if too long', () => {
    expect(isValidVin('1HGCM82633A004352EXTRA')).toBe(false); // >17
  });

  test('fails if contains disallowed letters (I, O, Q)', () => {
    expect(isValidVin('1HGCM82633A004352')).toBe(false); // contains O
    expect(isValidVin('1HGCM82633AI04352')).toBe(false); // contains I
    expect(isValidVin('1HGCM82633AQ04352')).toBe(false); // contains Q
  });

  test('fails if contains lowercase letters', () => {
    expect(isValidVin('1hgcm82633a004352')).toBe(false);
  });

  test('fails if contains symbols', () => {
    expect(isValidVin('1HGCM82633A00$352')).toBe(false);
  });

  test('trims input before validating', () => {
    expect(isValidVin(' 1HGCM82633A004352 ')).toBe(true);
  });
});
```

Registration Form Logic

This validates multiple form fields, ensuring all required fields are present and properly validated. It includes checks for username presence, email format, password length and password mismatch, and confirmation.

Test Case	Description	Expected Result	Status
Valid input	All fields completed correctly	[]	Passed
Missing username	Detects empty username	["Username is required."]	Passed
Password mismatch	Detects mismatch between password and confirmation	["Passwords do not match."]	Passed
Short password	Requires minimum length of 8 characters	["Password must be at least 8 characters."]	Passed
Invalid email	Detects malformed email address	["A valid email is required."]	Passed

```
const validateRegistrationForm =
require('../__tests__/js/validateRegistrationForm.js');

describe('Registration form logic', () => {
  test('passes with valid input', () => {
    const result = validateRegistrationForm({
      username: 'testUser',
      password: 'secure123',
      confirmPassword: 'secure123',
      email: 'test@example.com'
    });
    expect(result.length).toBe(0);
  });

  test('detects missing username', () => {
    const result = validateRegistrationForm({
      username: '',
      password: 'secure123',
      confirmPassword: 'secure123',
      email: 'test@example.com'
    });
    expect(result).toContain("Username is required.");
  });

  test('detects password mismatch', () => {
    const result = validateRegistrationForm({
      username: 'test',
      password: 'secure123',
      confirmPassword: 'different123',
      email: 'test@example.com'
    });
    expect(result).toContain("Passwords do not match.");
  });

  test('detects short passwords', () => {
    const result = validateRegistrationForm({
      username: 'test',
      password: '123',
      confirmPassword: '123',
      email: 'test@example.com'
    });
    expect(result).toContain("Password must be at least 8 characters.");
  });
});
```

```
test('detects invalid email', () => {  
  const result = validateRegistrationForm({  
    username: 'test',  
    password: 'secure123',  
    confirmPassword: 'secure123',  
    email: 'invalid-email'  });  
  expect(result).toContain("A valid email is required.");  
});
```

References

- [1] K. Sun and S. Ryu, "Analysis of JavaScript Programs: Challenges and Research Trends," *ACM Comput Surv*, vol. 50, no. 4, pp. 1–34, 2018, doi: 10.1145/3106741.
- [2] D. M. Groenewegen and E. Visser, "Integration of data validation and user interface concerns in a DSL for web applications," *Softw Syst Model*, vol. 12, no. 1, pp. 35–52, Feb. 2013, doi: <https://doi.org/10.1007/s10270-010-0173-9>.
- [3] T. Powell, "Pros and cons of client-side programming," *InternetWeek (Manhasset, N.Y.)*, no. 633, p. 61, 1996.
- [4] A. Hildyard, "Generalized event handling JavaScript," *Web Techniques*, vol. 4, no. 2, pp. 75–84, Feb. 1999, [Online]. Available: <https://www.proquest.com/trade-journals/generalized-event-handling-javascript/docview/274860315/se-2?accountid=130472>
- [5] J. Brodtkin, "AJAX faces development challenges," *Network World*, vol. 24, no. 6, p. 8, Feb. 2007, [Online]. Available: <https://www.proquest.com/trade-journals/ajax-faces-development-challenges/docview/215982001/se-2?accountid=130472>
- [6] J. de Leeuw, "Server-Side Statistics Scripting in PHP," *J Stat Softw*, vol. 2, no. 1, pp. 1–13, 1997, doi: 10.18637/jss.v002.i01.
- [7] K. Selvamani, A. Duraisamy, and A. Kannan, "Protection of Web Applications from Cross-Site Scripting Attacks in Browser Side," 2010, doi: 10.48550/arxiv.1004.1769.
- [8] A. Datta, K. Dutta, H. Thomas, D. Vandermeer, and K. Ramamritham, "Web Caching Accelerating Dynamic Web Content Generation As a middle-tier, server-side caching engine, the Dynamic Content Accelerator reduces dynamic page-generation." [Online]. Available: <http://computer.org/internet/>
- [9] F. Piessens, F. Massacci, W. De Groef, N. van Ginkel, and G. Tolomei, "A Server-Side JavaScript Security Architecture for Secure Integration of Third-Party Libraries," *Security and communication networks*, vol. 2019, no. 2019, pp. 1–21, 2019, doi: 10.1155/2019/9629034.
- [10] S. Wedman, T. Annette, and H. and Saiedian, "An Analytical Study of Web Application Session Management Mechanisms and HTTP Session Hijacking Attacks," *Information Security Journal: A Global Perspective*, vol. 22, no. 2, pp. 55–67, Mar. 2013, doi: 10.1080/19393555.2013.783952.
- [11] P. Reeser and R. Hariharan, "An Analytic Model of Web Servers in Distributed Computing Environments," *Telecommun Syst*, vol. 21, no. 2–4, p. 283, Dec. 2002, [Online]. Available: <https://www.proquest.com/scholarly-journals/analytic->

model-web-servers-distributed-computing/docview/212042146/se-2?accountid=130472

- [12] P. Reeser and R. Hariharan, "An Analytic Model of Web Servers in Distributed Computing Environments," *Telecommun Syst*, vol. 21, no. 2–4, p. 283, Dec. 2002, [Online]. Available: <https://www.proquest.com/scholarly-journals/analytic-model-web-servers-distributed-computing/docview/212042146/se-2?accountid=130472>
- [13] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to Build High-Performance Network Programs," 2010. [Online]. Available: www.computer.org/internet/
- [14] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem," 2017, *Cornell University Library, arXiv.org, Ithaca*.
- [15] K. I. D. Kyriakou and N. D. Tselikas, "Complementing JavaScript in High-Performance Node.js and Web Applications with Rust and WebAssembly," *Electronics (Switzerland)*, vol. 11, no. 19, Oct. 2022, doi: 10.3390/electronics11193217.
- [16] T. Ostretsova, S. Pereiaslavska, and D. Ostretsov, "COMPARATIVE ANALYSIS OF THE PERFORMANCE OF NODE AND BUN PLATFORMS IN EDUCATIONAL WEB APPLICATIONS," *Vidkrite osvîtnê e-seredoviše sučasnogo universitetu*, no. 18, pp. 119–131, 2025, doi: 10.28925/2414-0325.2025.1810.
- [17] J. Luo, B. Zhou, Y. Zheng, and W. Pan, "Research on high performance web service construction method based on JavaScript asynchronous programming technique," *Applied Mathematics and Nonlinear Sciences*, vol. 9, no. 1, Jan. 2024, doi: 10.2478/amns-2024-2811.

Analysis_of_Web_Scripting_and_Node.js _Ryan_Evans 242520

GRADEMARK REPORT

FINAL GRADE

74/100

GENERAL COMMENTS

Server/Client Code (25): 19

Node.js (25): 15

SW Doc (50): 40

FEEDBACK

=====

You have covered most of the key features of client and server scripting. I was hoping for a bit more detail on the execution of these scripts.

I was anticipating much more detail on the architecture of node.js and generally a bit more detail.

I was hoping for a little more discussion of the code but there is good evidence of testing.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17
