Day 5 - Testing and Backend Refinement - [avion furniture]

Key Learning Outcomes

1. Perform comprehensive testing, including functional, non-functional, user acceptance, and security testing.
2. Implement robust error handling mechanisms with clear, user-friendly fallback messages.
3. Optimize the marketplace for speed, responsiveness, and performance metrics.
4. Ensure cross-browser compatibility and device responsiveness for a seamless user experience.
5. Develop and submit professional testing documentation that meets industry standards, including a CSV-based test report.
6. Handle API errors gracefully with fallback UI elements and logs.

Key Areas of Focus

1. Functional Testing

- **Objective**: Validate that all marketplace features work as intended.
- **Core Functionalities to Test**:
  - Product listing: Ensure products are displayed correctly.
  - Filters and search: Validate accurate results based on user inputs.
  - Cart operations: Add, update, and remove items from the cart.
  - Dynamic routing: Verify individual product detail pages load correctly.
- **Testing Tools**:
  - Postman: For API response testing.
  - React Testing Library: For component behavior testing.
  - Cypress: For end-to-end testing.

2. Error Handling

- **Overview**: Error handling was a critical area during testing. Several issues were encountered related to query parameters, fetch API failures, and data handling. Below is a detailed breakdown of the errors faced and their resolutions.

Errors Encountered:

1. Query Parameter Errors:

- **Description**: Incorrect query parameters resulted in invalid API responses or no data being fetched.
- **Root Cause**:
  - Missing or improperly formatted parameters in API requests.
  - Backend API was case-sensitive to certain query parameters.
- **Resolution**:
  - Validated query parameters before making API requests.
  - Implemented helper functions to sanitize and format query strings properly.

Example Code Fix:

```
function formatQueryParams(params) {
  return Object.entries(params)
    .map(([key, value]) =>
`${encodeURIComponent(key)}=${encodeURIComponent(value)}`)
    .join('&');
}

const queryParams = formatQueryParams({ search: 'laptops', sort: 'price' });
const url = `/api/products?${queryParams}`;
fetch(url)
  .then(response => response.json())
  .then(data => setProducts(data))
  .catch(error => console.error('Failed to fetch products:', error));
```

2. Fetch API Errors:

- **Description**: API requests using `fetch` failed due to network issues, incorrect URLs, or CORS policies.
- **Root Cause**:
  - Unreachable endpoints or network disruptions.
  - Missing headers required by the API.
  - API not configured to handle CORS requests from the client.
- **Resolution**:
  - Added error handling using `try-catch` blocks to catch fetch errors and provide fallback mechanisms.
  - Verified API endpoints and added necessary headers (e.g., `Authorization` or `Content-Type`).
  - Updated server-side API configuration to allow CORS from the client.

Example Code Fix:

```
async function fetchProducts() {
  try {
    const response = await fetch('/api/products', {
      headers: {
        'Content-Type': 'application/json',
        Authorization: 'Bearer your-token-here',
      },
    });
```

```
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    const data = await response.json();
    setProducts(data);
  } catch (error) {
    console.error('Fetch API error:', error);
    setError('Unable to fetch products. Please try again later.');
  }
}
```

3. Fallback UI for Empty States:

- **Description**: When the API returned no data, the UI displayed a blank screen without any feedback.
- **Resolution**:
    - Added fallback UI elements to inform users when data is unavailable.

Example Code for Fallback UI:

```
{products.length === 0 ? (
  <div>No products available. Please try again later.</div>
) : (
  <ProductList products={products} />
)}
```

3. Performance Testing

- **Optimization Steps**:
    - Compress images using tools like TinyPNG or ImageOptim.
    - Use lazy loading for large images or assets.
    - Minimize and bundle JavaScript and CSS.
- **Testing Tools**:
    - Lighthouse
    - GTmetrix
    - WebPageTest

Performance Metrics to Aim For:

- Initial load time: Under 2 seconds.
- Smooth interactions with minimal lag.

4. Cross-Browser and Device Testing

- **Browser Testing**:
    - Test on Chrome, Firefox, Safari, and Edge.
    - Verify consistent rendering and functionality.

- **Device Testing**:
    - Use responsive design testing tools like BrowserStack or LambdaTest.
    - Test manually on at least one physical mobile device.

5. Security Testing

- **Best Practices**:
    - Validate input fields to prevent injection attacks.
    - Sanitize inputs using regular expressions.
    - Ensure API communication is secure with HTTPS.
- **Tools**:
    - OWASP ZAP: For automated vulnerability scanning.
    - Burp Suite: For advanced penetration testing.

6. User Acceptance Testing (UAT)

- **Simulate Real-World Usage**:
    - Perform tasks like browsing products, adding items to the cart, and checking out.
- **Feedback Collection**:
    - Ask peers or mentors to interact with the application and provide feedback.

7. Documentation Updates

- **Details to Include**:
    - Test cases executed and their results.
    - Performance optimization steps.
    - Security measures implemented.
    - Challenges faced and resolutions applied.
- **Submission Format**:
    - PDF or Markdown.
    - Include a Table of Contents for easy navigation.

CSV-Based Testing Report

Columns to Include:

- Test Case ID: Unique identifier for each test case.
- Test Case Description: Explanation of what is being tested.
- Test Steps: Step-by-step procedure for execution.
- Expected Result: Anticipated outcome.
- Actual Result: Observed outcome.
- Status: Mark as "Passed," "Failed," or "Skipped."
- Severity Level: Categorize issues as High, Medium, or Low.

- Assigned To: Name of the person responsible for fixing issues.
- Remarks: Additional notes or comments.

Example CSV Structure:

Test Case ID	Description	Steps	Expected Result	Actual Result	Status	Severity	Assigned To	Remarks
TC001	Validate Product Listing	Visit homepage	Products are displayed	Products are displayed	Passed	Low	Developer	N/A
TC002	API Error Handling	Disconnect API	Show fallback UI	Fallback UI displayed	Passed	Medium	Developer	N/A

Expected Outputs by End of Day 5

1. Fully tested and functional marketplace components.
2. Clear and user-friendly error handling mechanisms.
3. Optimized performance with faster load times.
4. Responsive design tested thoroughly across multiple browsers and devices.
5. Comprehensive CSV-based testing report.
6. Professional documentation summarizing all testing and optimization efforts.

FAQs

1. What tools can I use for functional testing?

- Cypress, Postman, React Testing Library.

2. How do I handle API failures gracefully?

- Use `try-catch` blocks and display fallback UI elements or error messages.

3. What should my CSV-based testing report include?

- Include columns like Test Case ID, Description, Steps, Expected and Actual Results, Status, Severity, Assigned To, and Remarks.

4. How can I test for responsiveness across devices?

- Use tools like BrowserStack or LambdaTest and test on physical devices.

5. What are the best practices for performance optimization?

- Compress images, minimize JavaScript and CSS, and implement caching strategies.

6. How do I secure sensitive API keys?

- Store them in environment variables and use HTTPS for secure communication.

7. Is documentation mandatory for submission?

- Yes, documentation is mandatory and must summarize testing efforts, challenges, and resolutions.

8. How can I optimize load times?

- Reduce image sizes, use lazy loading, and run performance audits using Lighthouse.

9. What is User Acceptance Testing (UAT)?

- Simulating real-world user interactions to ensure workflows are intuitive and error-free.

Thunder Client Errors Report

Overview

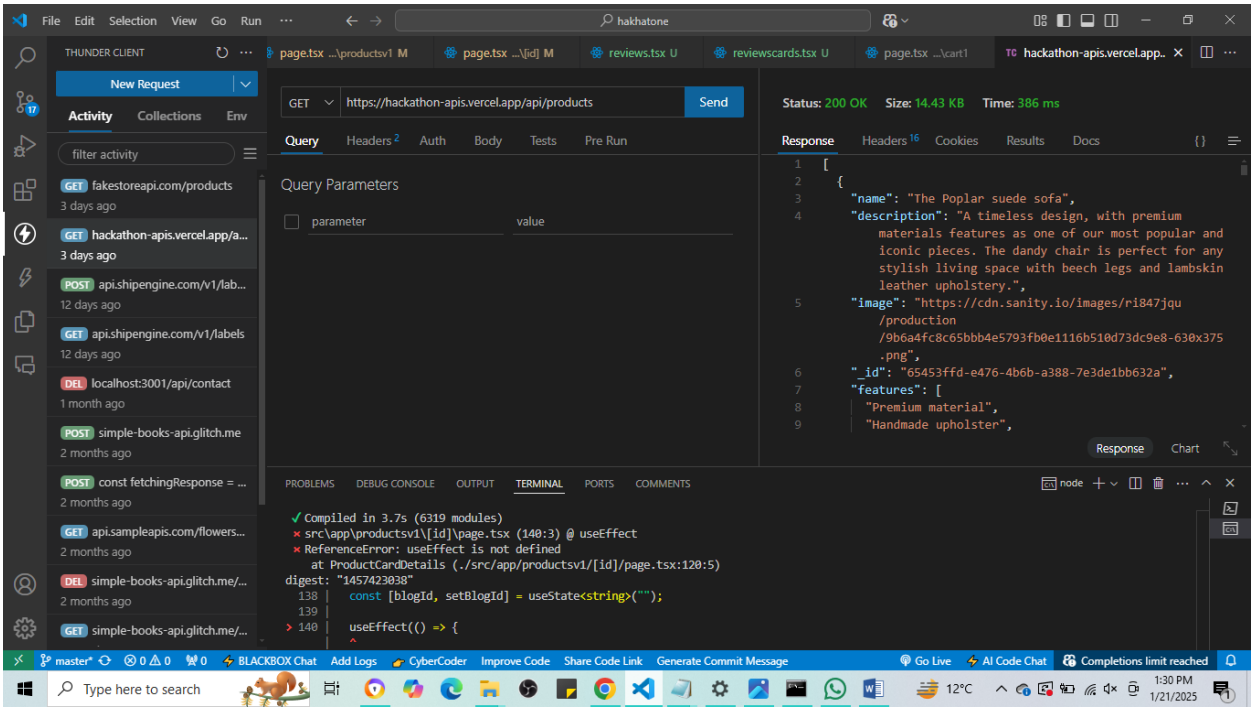
This report highlights the errors encountered while testing APIs using Thunder Client and the solutions implemented to resolve them. Thunder Client was used to validate API functionality during the development of the marketplace project.

Issues Faced

1. Invalid Response Formats

Description:

Some API responses returned invalid or unexpected formats despite correct request configurations. This led to issues when parsing the responses in the frontend application.



Challenges:

- Identifying whether the issue was with the API response or Thunder Client configuration.
- Debugging inconsistencies between the Thunder Client results and the application's Fetch API calls.

Solutions:

- Verified the structure and format of API responses using Thunder Client's raw response viewer.
- Cross-checked responses against backend documentation to ensure they adhered to the expected schema.
- Implemented additional parsing and validation logic in the frontend to handle edge cases.

---

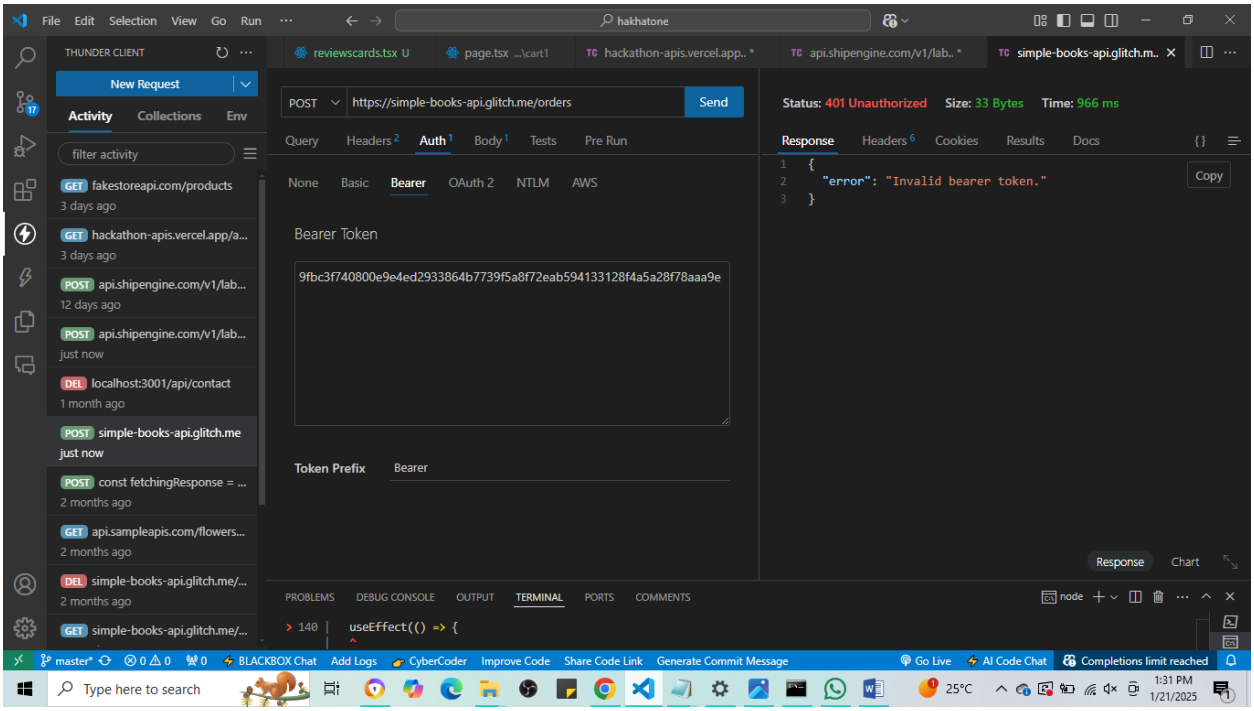
2. Missing or Incorrect Headers

Description:

Errors occurred due to missing headers (e.g., `Authorization` or `Content-Type`) or incorrect values in request headers.

Challenges:

- Debugging header-related errors when the API returned cryptic messages.



- Ensuring consistent header usage across Thunder Client and the application.

Solutions:

- Reviewed backend documentation to confirm required headers for each endpoint.
- Configured Thunder Client to include all necessary headers for the test requests.
- Example of resolved error:

```
Error: "Invalid API Key"
Resolution: Updated the `.env` file with the correct API key and verified the
header values in Thunder Client.
```

---

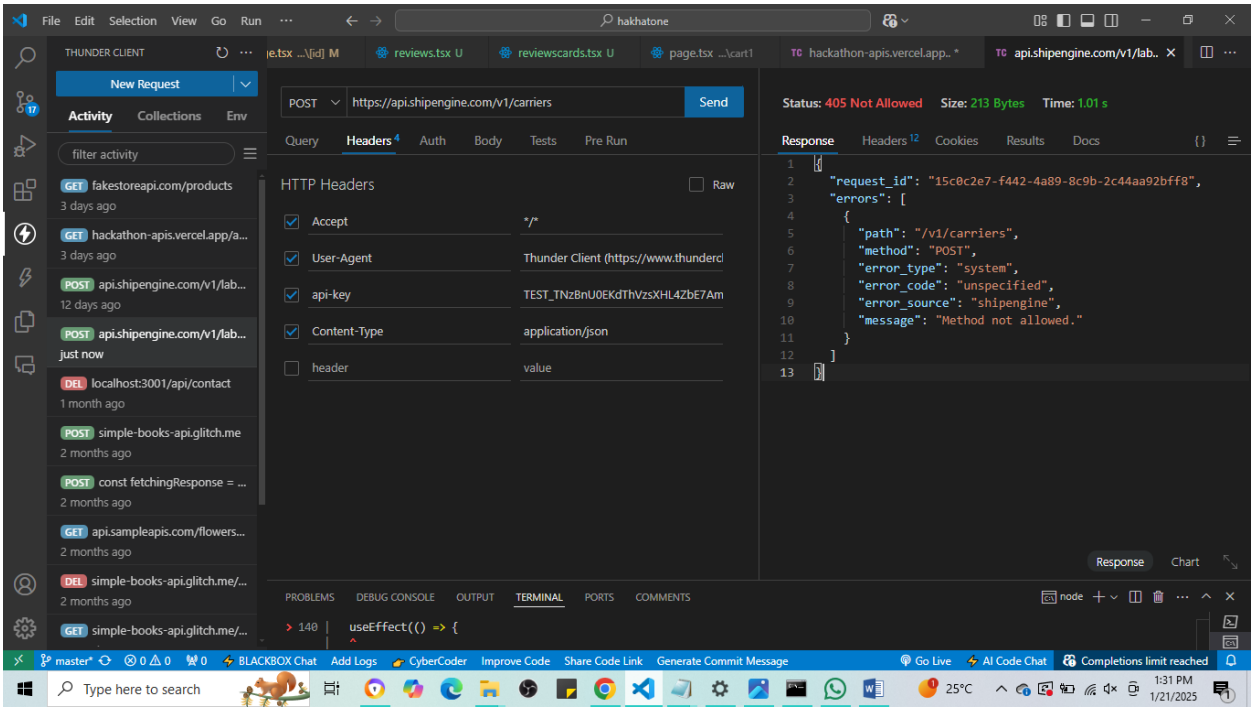
3. HTTP Method Mismatches

Description:

Some errors were due to using incorrect HTTP methods (e.g., `POST` instead of `GET`) for specific endpoints.

Challenges:

- Ensuring the correct method was used for each API call.
- Understanding server-side expectations for methods and payloads.



Solutions:

- Checked backend API documentation for correct HTTP methods.
- Updated Thunder Client configurations to match the expected methods and payload structures.
- Retested endpoints to ensure consistent behavior.

---

4. Partial Responses

Description:

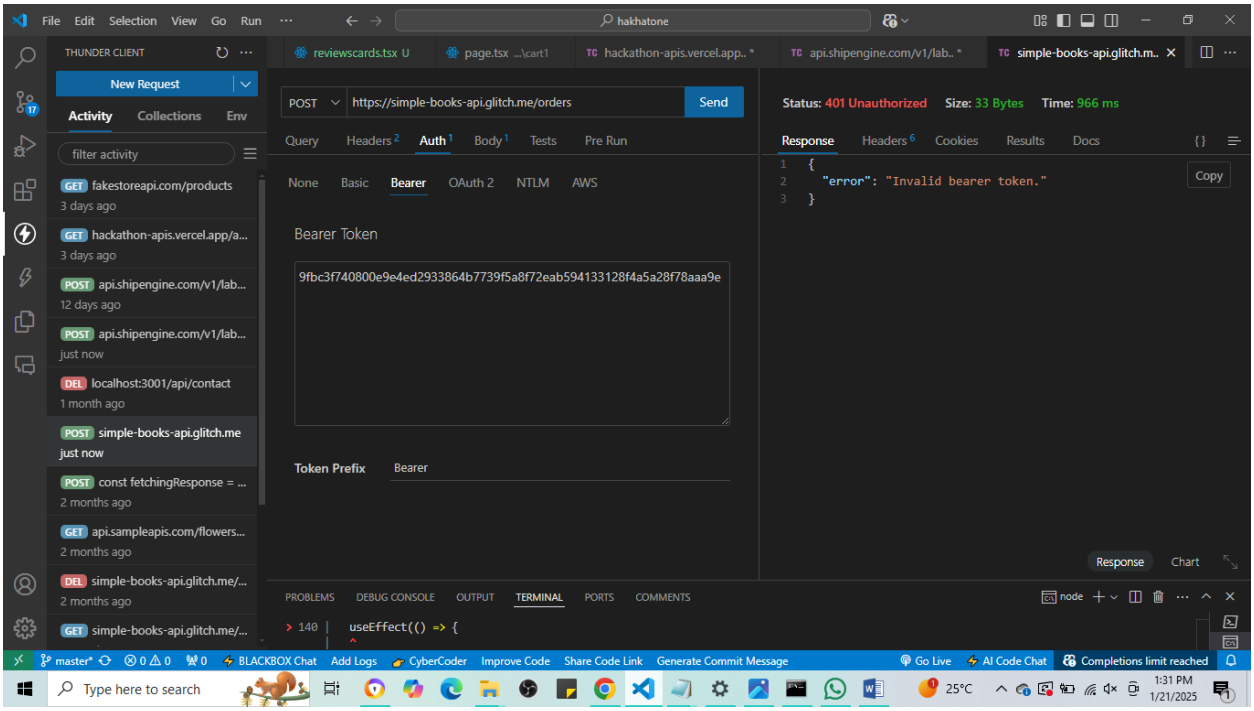
The server occasionally returned partial responses, which caused downstream issues when attempting to process incomplete data.

Challenges:

- Identifying the root cause of partial responses.
- Ensuring the application could handle such cases gracefully.

Solutions:

- Used Thunder Client's detailed request/response logs to analyze the issue.
- Implemented fallback UI elements in the application to handle partial or missing data.



- Communicated with the backend team to resolve inconsistencies causing partial responses.

Lessons Learned

- Always validate request configurations in tools like Thunder Client against backend documentation.
- Maintain consistent headers, methods, and parameter formats across testing tools and the application.
- Use detailed logs from Thunder Client to debug and identify root causes effectively.

Recommendations

1. **Improve Backend Documentation:** Ensure clear documentation for required headers, methods, and response structures.
2. **Automate API Tests:** Use tools like Postman or automated scripts to regularly validate API endpoints.
3. **Centralize Error Handling:** Implement a shared error-handling strategy across testing tools and the application.
4. **Use Environment Variables:** Store sensitive keys and configurations securely to avoid manual errors during testing.

This report aims to serve as a reference for resolving similar issues in future projects and improving the efficiency of API testing workflows.

- **Functional Testing**: ✓ / ✗
- **Error Handling**: ✓ / ✗
- **Performance Optimization**: ✓ / ✗
- **Cross-Browser and Device Testing**: ✓ / ✗
- **Security Testing**: ✓ / ✗
- **User Acceptance Testing**: ✓ / ✗
- **Documentation Updates**: ✓ / ✗