ACTIVITY 1 CASE STUDY

Submitted By:

Group 4

Casampol, Claynne

Diaz, Joefhanie Marj J.

Reyes, Maria Claire M.

Sales, Ethaniel Klymore

Viñalon, Robert Glenn N.

Submitted To:

Mr. Severino Bedis Jr.

Date

October 13, 2024

Course & Subject Name

COMP 016

Web Development

Introduction

Project Overview: The project team is developing a dynamic blog platform using technology stacks for frontend HTML, CSS and JavaScript and for backend is PHP Laravel.

Main Issue: The frontend team encounters issues with retrieving the blog posts from the Laravel API, resulting in unexpected responses and difficulties displaying the content.

Objective: To identify the issue faced by both the frontend and backend team by proposing debugging strategies, outlining how to improve API, explaining how to integrate error handling, and describe testing methodology to test API integration for troubleshooting.

Frontend Team

- **Identify the specific issues you might face with the API responses.**

  Here are the issues we might face with the API responses:

    - **Data Parsing Errors**

      The frontend might receive data in the format where the front end is not expecting. This could lead to JavaScript errors when trying to parse or display the content. This can happen if the data structure returned by the API changes or if there are unexpected fields or data types.

    - **Empty or Incomplete Data**

      If the API returns empty arrays or objects, or the key fields such as titles, dates, or content are missing, the frontend will struggle to render the blog posts properly, affecting the display of blog posts. This can lead to broken layouts or missing information on the page that could mess up the interface .

    - **Network Issues and Slow Responses**

      A slow network response or connection issues can result in timeouts or incomplete data being sent to the frontend that is needed in a timely manner. This would result in a delayed content loading or a complete failure to retrieve the blog posts that could negatively affect the user's experience.

- **Propose two debugging strategies you would use to resolve these issues.**

  Here are the two debugging strategies:

  - **Backtracking**

    With this method, the error is tracked back through the code from where it started to determine its root cause. This works well if you have a good idea of where the problem might be coming from and can work backwards methodically to identify the source of the issue.

  - **Divide and conquer**

    Dividing large systems into smaller components may allow more effective problem identification. Using binary search, where you can add a log statement or breakpoint midway through a series of calls, is a great instance of this. If the problem still doesn't arise by then, you go through the procedure again in the second half, gradually reducing the size of the problem area.

- **Describe how you would communicate these issues to the Backend Team, including the information you would provide.**

The frontend team would provide **user reports** and **error messages** being visible within the page. We would **explain the impact** this may have on the site, for example in the network issues this will result in frustration among the users hence a decreased application performance. In data parsing errors this will **affect user experience**, leading to failed transactions or incorrect display of data. The frontend team has full focus on the site, and what happens within the site because what the team sees is the finished work which includes the work of the backend team. The team should **also provide frequencies** on how often these errors occur for a more detailed case, so that the backend team can strategize with a more efficient code than the last one. Of course, if the frontend team could do anything to also fix the issue, both teams could collaborate on the issues, but first the backend team must investigate the root cause of these issues. The **frontend team must provide accurate and precise metrics** of all these issues being raised, since the team is heavily focused on the *appearance* and *interactivity* of the application, **ensuring an intuitive and appealing user experience**, while the backend team focuses on *functionality* and *data management*, so both teams should communicate with each other regularly **to ensure the best possible user experience.**

Backend Team

- **Outline three best practices you should follow when designing the API endpoints in Laravel**

    - **Resource-Based Routing**

        When designing API endpoints in Laravel, following the RESTful (Representational State Transfer) architecture is best practice. This approach adheres to several key principles, including:

        - **Stateless:** Each request from a client to the server should include all the necessary information. This ensures that the server does not need to maintain any session state for the client.
        - **Uniform Interface:** The API should use standard HTTP methods (GET, POST, PUT, DELETE) for interacting with resources. This provides a consistent and predictable way for the clients to interact with API.
        - **Client-Server Separation:** The client (user interface) and server (data storage) are distinct entities that communicate through requests and responses. This separation promotes modularity and scalability.
        - **Cacheability:** Client can cache responses from the server to improve performance. This reduces the number of the requests to the server and speeds up data retrieval.
        - **Layered System:** The system is composed of multiple layers, each responsible for specific functionalities. This layered approach enhances maintainability and allows for easier modifications.

        By implementing RESTful architecture in the Laravel framework, we can build a robust and structured system. This approach simplifies the understanding of how ro implement necessary components without hindering the addition of more clients or services. It also promotes independence, making the system compatible with various platforms and devices.

- **Input Validation and Output Formatting**

One of the features of Laravel is having built-in validation features. Input validation is a good practice and a crucial aspect to consider, as it addresses issues with maintaining data consistency and security vulnerabilities. By using Laravel's built-in functions, we can manipulate input fields, set minimum and maximum lengths, and require unique entries and valid email addresses. This allows us to clearly define validation rules for each endpoint and avoid unnecessary errors. If validation fails, we can provide users with meaningful error responses, including clear error messages and HTTP status codes (e.g., 400 Bad Request, 422 Unprocessable Entity).

When it comes to output formatting, it's good practice to use a standard format like JSON (JavaScript Object Notation). This format is lightweight, flexible, and Laravel offers robust tools for managing JSON data. Sticking to JSON maintains a consistent structure across your API, making it easier for clients to parse and consume the data. Additionally, it allows for seamless integration with pagination for large datasets, preventing clients from being overwhelmed with too much data at once.

- **Rate Limiting and Throttling**

Implementing rate limiting and throttling in Laravel is a good practice because it enhances the security and stability of the API by preventing abuse and distributed denial-of-service (DDoS) attacks, ensures fair usage among users, optimizes performance, and provides a smoother, more consistent user experience. Laravel's built-in middleware support and flexible customization options make it straightforward to set up and manage these controls, seamlessly integrating with other aspects of the application for efficient and effective rate management. Implementing rate limiting and throttling in the API comes with a lot of benefits:

- **Protection against Abuse:** Rate limiting prevents a single user from overloading the server with too many requests in a short time. This helps avoid potential Denial of Service (DoS) attacks and ensures the API remains available to all users.
- **Fair Usage:** Ensure all users have fair access to the API resources by throttling requests. This prevents a few users from consuming all the available resources, providing a balanced and equitable usage experience.
- **Performance Optimization:** Controlling the number of requests hitting the server can improve its overall performance and stability. This leads to better response times and a smoother experience for all users.

- **Explain how you would implement error handling in your API to assist the Frontend Team in troubleshooting.**
    - **Detailed Error Message**
        - **Descriptive Messages:** Provide clear and descriptive error messages that explain the issue to the frontend team. Avoid generic messages like "Error occurred." Instead, include more context or details.

  - **Contextual Information:** For example, if a blog post is not found, include the requested ID in the error message. This allows the frontend to display a more specific error to the user.

- **Consistent Error Response Structure**

  - **Standardized Format:** Ensure all error responses adhere to a consistent format, making it easier for the frontend to handle errors efficiently.
  - **JSON Structure:** Use a JSON structure like this:

    ```
    {
     "success": false,
      "error": {
             "code": "404",
             "message": "Blog post not found.",
             "details": "The requested blog post with ID 123 does not
    exist."
              }
    }
    ```

- **Appropriate HTTP Status Codes**

  - **Signal Error Types:** Use appropriate HTTP status codes to signal the type of error to the frontend. This helps the frontend handle errors correctly.
  - **Common Status Codes:**
    - 400 Bad Request: Invalid input data.
    - 401 Unauthorized: Authentication required.
    - 403 Forbidden: Insufficient permissions.
    - 404 Not Found: Resource not found.
    - 500 Internal Server Error: Server-side error.

- **Logging**

  By leveraging detailed logging, the backend team can implement more effective error handling and provide the frontend team with the information they need to troubleshoot and resolve issues swiftly and accurately. Here’s how it would help:

  - **Detailed Error Context:** Logs capture comprehensive details about errors, including stack traces, request parameters, and user context. This level of detail allows the backend team to understand exactly what went wrong and under what circumstances. When the frontend team reports an issue, the backend team can quickly reference the logs to identify the root cause and provide precise fixes or guidance.

  - **Tracking and Monitoring:** With centralized logging services like Laravel Telescope or Sentry, you can monitor errors in real-time. This immediate visibility enables the backend team to detect and address issues as they occur, often before the frontend team even encounters them. Quick detection and response to errors minimize downtime and improve the overall user experience.

  - **Pattern Recognition:** Consistent logging allows the backend team to recognize patterns in errors over time. This helps in identifying recurring issues or common points of failure within the API. By addressing these patterns, the backend team can implement more robust error handling and preemptively fix potential bugs, reducing the frequency of similar errors reported by the frontend team.

- **Describe one testing methodology you would use to ensure that your API is functioning correctly before the Frontend Team integrates it.**
    - **Behavior-Driven Development (BDD)** is a solid testing methodology to ensure the API functions correctly before the Frontend Team integrates it. BDD goes beyond traditional testing methodologies by promoting a shared understanding of how the API should behave under various scenarios. In Laravel, integrating BDD can be accomplished with tools like Behat or PhpSpec, alongside PHPUnit. We start by writing scenarios that describe the expected behavior of the API in plain language and then implement tests based on those scenarios. This approach ensures that the API not only works as intended but also meets the user's requirements and expectations. It also facilitates communication between developers and non-technical stakeholders, ensuring everyone is on the same page.

      Benefits of Using BDD in Laravel:
        - **Improved Communication:** By using a common language to describe scenarios, BDD bridges the gap between technical and non-technical team members, ensuring everyone is aligned on the project goals.
        - **Higher Quality:** Automated tests derived from well-defined scenarios help catch bugs and inconsistencies early, leading to a more reliable API.
        - **Documentation:** The scenarios and tests serve as living documentation, providing a clear and up-to-date reference on how the API should behave.

References

Holcombe, J. (2024, April 30). *Data Validation in Laravel: Convenient and Powerful.* Kinsta. https://kinsta.com/blog/laravel-validation/

Ram, M. (2023, June 23). *Building a RESTful API with Laravel: Best practices and tools.* Medium. https://medium.com/@mukesh.ram/building-a-restful-api-with-laravel-best-practices-and-tools-907bdf4b5621

Redacción Aguayo (n.d.) *Frontend vs. Backend and Their Influence on UX* Aguayo. https://aguayo.co/en/blog-aguayp-user-experience/frontend-vs-backend-ux-influence/#:~:text=Frontend%3A%20Focuses%20on%20the%20appearance,can%20handle%20user%20requests%20efficiently.

LearNowx. (2023, November 29). Common challenges faced by Front-End developers and how to overcome them. *Medium*. https://medium.com/@rs4319026/common-challenges-faced-by-front-end-developers-and-how-to-overcome-them-b97c10dd445c

iTitans. (2023, November 24). Most common Front-End mistakes and how to avoid them. *iTitans*. https://ititans.com/blog/most-common-front-end-mistakes-and-how-to-avoid-them/

Laravel - The PHP Framework For Web Artisans. (2024). Retrieved October 12, 2024, from Laravel.com website: https://laravel.com/docs/8.x/logging?form=MG0AV3

Laravel - The PHP Framework For Web Artisans. (2024). Retrieved October 12, 2024, from Laravel.com website: https://laravel.com/docs/8.x/telescope?form=MG0AV3

Laravel - The PHP Framework For Web Artisans. (2024). Retrieved October 12, 2024, from Laravel.com website: https://laravel.com/docs/8.x/testing?form=MG0AV3

Behaviour-Driven Development - Cucumber Documentation. (2019). Retrieved October 12, 2024, from Cucumber.io website: https://cucumber.io/docs/bdd/?form=MG0AV3

Laravel - The PHP Framework For Web Artisans. (2024). Retrieved October 12, 2024, from Laravel.com website: https://laravel.com/docs/8.x/routing?form=MG0AV3#rate-limiting