



Polytechnic University of the Philippines
College of Computer and Information Sciences
Sta. Mesa, Manila



Case Study: Front and Backend Development

by

Group 7

Bonavente, Willie M.

Capin, Marzella Nicole N.

Go, Wilmae Euhanna Isachar G.

Macatunao, Joshua C.

Mortel, Meg Edelvieste B.

COMP 016 - Web Development

Bedis Jr., Severino

Instructor



Activity #1 – Case Study about Front and Back Development

You are part of a project team developing a dynamic blog platform using HTML, CSS, JavaScript for the frontend, and PHP Laravel for the backend. The frontend team encounters issues retrieving blog posts from the Laravel API, resulting in unexpected responses and difficulties displaying the content.

Instructions:

Frontend Team:

I. Identify three specific issues you might face with the API responses.

The frontend team depends on API responses from a Laravel backend to retrieve blog posts. However, we have encountered three issues that interrupt the communication between the frontend and backend, resulting in unexpected responses and difficulties displaying the content.

(1) Delayed API responses

Factors such as processing time, network latency, payload size, and third-party dependency significantly affect the API response time. The delay in responses often leads to slow page load times and timeouts. As a result, the blog platform seems unresponsive.

(2) Inconsistent and Missing Data:

The inconsistent API response might be missing essential data resulting in errors when retrieving blog posts. Several responses are missing blog elements or returning unexpected data types. The blog posts are either incompletely or incorrectly displayed.

(3) Authentication and Authorization Issues:

Invalid authorization or expired API credentials result in authentication errors to the API. These affect the accessibility of some features or private content, even to authorized users.



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

To resolve these issues, we propose these two debugging strategies: *checking logs and metrics, and checking API Endpoints with tools.*

(1) Check Logs and Metrics

Logs and metrics contain detailed information that can help us understand the behavior and performance of the API. We can utilize this to identify the errors and to help the team in locating the specific areas that need optimization.

(2) Check API Endpoints with Tools

Validate the API endpoints by using tools like *Postman* to ensure that the correct HTTP methods are used. This will verify that the API is returning the expected data format and identify any issues causing inconsistent data structures.

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

The frontend team will communicate with the backend team by creating a detailed report containing the three identified issues and the proposed solutions or strategies.

To Backend Team,

We, the Frontend Team, have been experiencing issues with retrieving blog posts from the Laravel API. For more details, please refer from the data below:

Issue:

Unexpected Responses and Difficulties Displaying the Content in Retrieving Blog Posts from the Laravel API

Issue Description:

The frontend team has encountered issues when attempting to retrieve blog posts from the Laravel API, leading to unexpected responses and difficulties displaying the content. Specifically, we are experiencing delayed API responses, inconsistent data structures, and authorization failures. These issues are causing delays in loading content, rendering malformed or missing blog posts, and preventing users from accessing certain features and content.



Suggested Solutions/Strategies:

- Check the log and metrics
- Check the API Endpoints

We are looking forward to your response and assistance in resolving the issue. The Frontend Team is also willing to schedule a meeting for more detailed and live troubleshooting, collaborations, follow-ups, and answer questions regarding the issue.

Thank you!

Best regards,

Frontend Team

Backend Team:

I. Outline three best practices you should follow when designing the API endpoints in Laravel.

Three Best Practices: Designing API endpoints in Laravel

According to Jamie Juviler in his blog post, an API endpoint is a digital location where an API receives requests about a specific resource on its server. Essentially, an endpoint is a URL that provides the location of a resource, serving as the destination for API requests. This concept can be compared to a restaurant where the endpoint is the kitchen, and the API is the waiter relaying customer orders. When designing API endpoints in Laravel, it is essential to adopt certain best practices that leverage the framework's capabilities to ensure efficiency, scalability, and maintainability.

Why design API endpoints with best practices?

In Laravel, designing API endpoints with best practices not only keeps the codebase clean and organized but also enhances performance and scalability. Laravel provides powerful tools like resource routing, middleware, and request validation that help streamline how API interactions are handled. These tools ensure that the architecture remains manageable even as the complexity of the application grows.



Additionally, by using best practices, you create a smoother interaction between the frontend and backend. This makes the system easier to maintain over time, allowing Laravel to handle complex processes like nested resource relationships or asynchronous tasks more efficiently. Properly designed APIs lead to fewer errors, better code readability, and long-term stability.

(1) Implementing RESTful Practices

Laravel's built-in support for RESTful routing simplifies the design of clean, intuitive APIs. RESTful design principles ensure that each API endpoint corresponds to a single resource, with HTTP methods like GET, POST, PUT, and DELETE used to manipulate these resources. Following these principles leads to predictable and uniform interactions between the frontend and backend.

Laravel goes further by offering resource controllers, which minimize boilerplate code. By defining something as simple as `Route::resource('posts', 'PostController')`, developers can automatically generate routes for common operations like creating, updating, and deleting blog posts. This built-in support speeds up development and ensures consistency across different endpoints, allowing developers to focus on building logic rather than routing mechanics.

(2) API Versioning

As APIs evolve, maintaining backward compatibility becomes essential. Laravel's API versioning capabilities make it easy to manage this. Using the `Route::prefix()` method, developers can define versioned routes like `/api/v1/posts` and `/api/v2/posts`. This enables multiple versions of the API to coexist, allowing developers to introduce new features without breaking existing integrations.

Versioning also allows for smoother rollouts and better testing of new features. For example, you can deploy new functionalities to a small group of users by directing them to a new API version while still supporting users on the older version. This flexibility is critical for ensuring that updates don't disrupt users relying on earlier versions, making Laravel ideal for long-term API management.



(3) Clear Error Handling

No matter how well an API is designed, errors will inevitably occur. Laravel simplifies error handling by offering built-in tools like the *abort()* function and exception handling features. These tools allow developers to send clear, informative error messages along with appropriate HTTP status codes, such as a 404 when a resource isn't found or a 422 for validation errors.

Consistent error handling across the application improves the developer experience and reduces debugging time. By leveraging middleware, you can standardize how errors are caught and handled, providing a reliable framework for managing unexpected issues. Laravel's exception handling system not only improves API reliability but also ensures that errors are communicated effectively to the frontend, enhancing the overall user experience.

In conclusion, Laravel provides a rich set of features that make it ideal for developing scalable, maintainable APIs. By following best practices like implementing RESTful routing, utilizing API versioning, and providing clear error handling, developers can build robust APIs that are easy to manage and evolve over time. These practices harness Laravel's strengths to ensure efficient communication between the frontend and backend, delivering an enhanced user experience while maintaining long-term stability and adaptability.

II. *Explain how you would implement error handling in your API to assist the Frontend Team in troubleshooting.*

IMPLEMENTING ERROR HANDLING IN THE LARAVEL API

In Laravel, error handling begins with ensuring that the API returns appropriate *HTTP status codes* for different scenarios. For instance, when a request is successfully processed, a status code of *200* should be returned, whereas a *404* status code signals that the requested resource, such as a blog post, was not found. By returning the correct status codes, the API communicates the nature of the problem to the frontend team, allowing them to adjust their requests accordingly.

In addition to status codes, providing *custom error messages* within the JSON responses enhances the troubleshooting process. Instead of a generic error response, the API should return detailed messages that describe what went wrong. For example, if an invalid blog



post ID is provided, the API could respond with a message like, “Invalid blog post ID,” along with additional details explaining that the ID must be an integer. These informative messages help the frontend team diagnose issues more effectively.

To centralize error handling, Laravel’s *Exception Handler* can be leveraged. By customizing the `render()` method within the handler, all exceptions can be captured and returned as consistent JSON responses, complete with appropriate error messages. This allows for unified error management across the entire API, simplifying the process for developers when issues arise. Additionally, implementing *logging* ensures that all errors are recorded, providing a valuable resource for the development team to investigate issues in-depth. Laravel’s logging system can be configured to store logs in files or send them to external services like *Sentry* for real-time error monitoring.

Validation errors, which occur when the data sent to the API does not meet specified criteria, can also be handled through Laravel’s *Validator* class. By validating incoming requests and returning descriptive error messages, the API can guide the frontend team in correcting their input. For example, if a blog post title is missing, the API might return a 422 status code with a message stating, “The title field is required,” allowing the frontend to rectify the issue without further delays.

III. *Describe one testing methodology you would use to ensure that your API is functioning correctly before the Frontend Team integrates it.*

TESTING METHODOLOGY: INTEGRATION TESTING

Before integrating the API with the frontend, it is essential to ensure that it functions correctly and handles all edge cases. One of the most effective testing methodologies for this is *integration testing*, which focuses on verifying that different components of the system work together as expected. In Laravel, integration testing can be performed using *PHPUnit*, the framework’s built-in testing suite.

Integration testing allows the development team to simulate real-world interactions between the API’s controllers, models, and routes. For example, to test whether the API can successfully retrieve blog posts, a test case could be written to simulate the creation of a blog post in the database. The test would then send a *GET* request to the API and assert that the response contains the correct data, such as the blog post’s title and body. If the response matches the expected output, the test passes, confirming that the API behaves as intended.



Integration tests should cover various scenarios, including both successful data retrieval and error cases. For instance, tests could be written to check how the API responds when an invalid blog post ID is provided, ensuring that the system returns a 404 status code and a clear error message. Additionally, tests should verify that the API handles authorization correctly, returning a 401 status code when a user attempts to access a restricted resource without proper credentials.

By running these integration tests before the frontend team begins using the API, the development team can ensure that the API is reliable and robust. This approach minimizes the risk of unexpected errors during the frontend integration process, allowing the project to progress smoothly. Furthermore, integration testing ensures that the API can handle real-world usage scenarios, providing a solid foundation for the frontend team to build upon.



References

- <https://www.astera.com/type/blog/api-errors/>
- <https://technologyadvice.com/blog/information-technology/api-error/>
- <https://apitoolkit.io/blog/frontend-api-performance/>
- <https://www.hypertest.co/api-testing/top-reasons-for-api-failures>
- <https://www.linkedin.com/advice/1/what-best-api-debugging-techniques-skills-programming-yiucc>
- <https://prismic.io/blog/api-response-times>
- <https://www.baeldung.com/rest-api-error-handling-best-practices>
- <https://blog.hubspot.com/website/api-endpoint>