www.phparch.com

October 2023
Volume 22 - Issue 10

php[architect]

The Magazine For PHP Professionals

Software Archeology

GraphQL APIs with PHP

Shared Thinking Patterns

ALSO INSIDE

Edited by Goblins and Ghouls

php[**architect**] is published twelve times a year by:

PHP Architect, LLC
9245 Twin Trails Dr #720503
San Diego, CA 92129, USA

Subscriptions
Print, digital, and corporate subscriptions are available. Visit https://www.phparch.com/magazine to subscribe or email contact@phparch.com for more information.

Advertising
To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Contact Information:
General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Scary PHP Content

Welcome, Ghouls and Goblins, to this month's spooky issue of PHP Architect. Of course, anytime you're working with code, things can get a little scary. Don't worry; we've got you covered with some great articles this month to help keep your code in line.

Sarah Aburu brings us a feature article this month titled, *GraphQL APIs with PHP: A Guide to Schema Design and Execution*. Sarah takes us through all things GraphQL in this extensive look at how an open-source querying language can help you to pull the exact data you need versus getting more/less data as in other API's.

Sydney Reinert brings us the next feature article this month, *The Shared Thinking Patterns Between Programmers and Artists*. Sydney shares the perspective of a GIMM student hoping to bridge the sometimes-formed gap between programmers and artists.

This month's Security Corner column has us looking within and at your own team/users. Eric Mann shares his insights about securing the system from the people who use it every day in his article, *The Meaning of "High Trust"*. It is a great read that offers tips for further protecting your system from its users.

The final installment of the Create Observability series is available this month in DDD Alley. Ed Barnard presents us with *Part 5: Offline Processes*. Ed will further explain the benefits of flow charts to capture the 'tribal knowledge' gained in the process of designing an email queue system. He points out that while flow charts are not necessary for writing PHP code, they are necessary for keeping your current and future team informed and to follow the 'why'.

Welcome to Radar, a new column focusing on trends in the PHP Community that every developer can benefit from. Matt Lantz will be taking the helm in this column and brings us the first article, *HTMX: The Simple Markup Extension We've Been Waiting For*. You'll definitely want to check out this thorough review of HTMX.

We're welcoming Steve McDougall to the column, Artisan Way, with the article, *Taking Laravel to the Orchestra: Building a Symfony Inspired Application*. Clean code is something we all strive for but don't necessarily achieve. Not anymore! Steve is going to show us the way by celebrating Laravel for what it is: a framework built for the developer experience, something to embrace versus fight against.

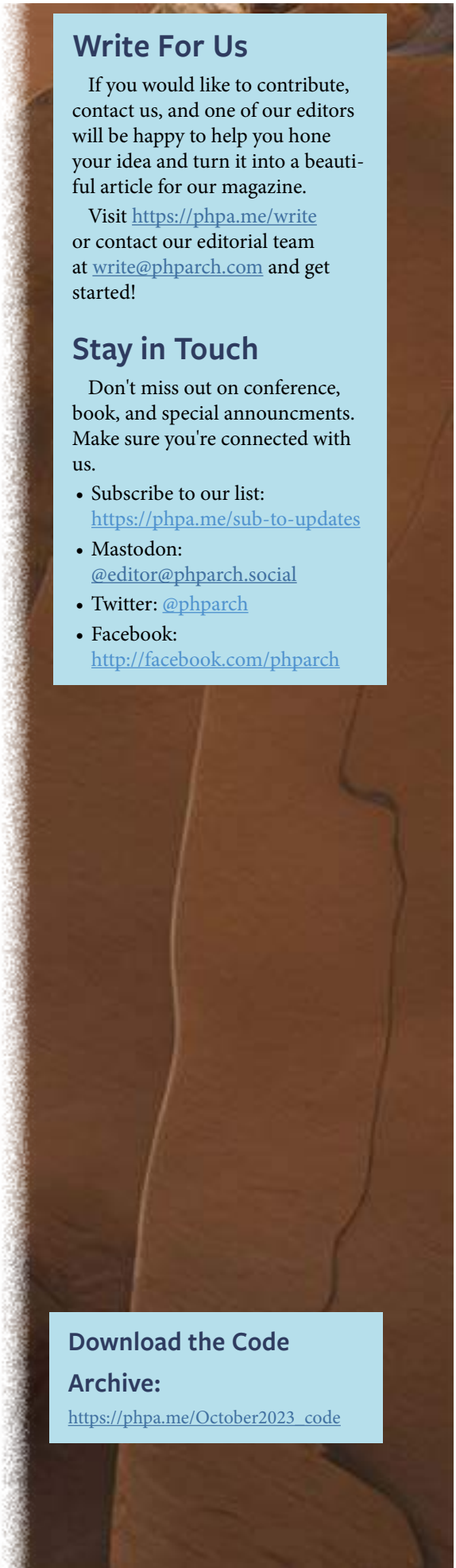
Puzzles is back with another way to sort this month. *Insertion Point* by Oscar Merida will continue exploring sorting algorithms. Check out how the insertion sort is set apart from the Comb Sort and Bubble Sort before it.

Welcome to Readable Code, a new column focused on just that, helping developers write clear, legible, and consistent code. Christopher Miller brings us our first article in this new column, titled, *Is Your Plan Extensive Enough To Help Someone Else?* In this article, you'll get Christopher's personal account of his own journey to create plans for development projects that get it right. He expands on this further as the article goes on to share how creating plans as an individual is different than as a team and some of the growing pains that he experienced in that discovery.

Over in Education Station, Chris Tankersley follows up on his article last month, bringing us *Software Archaeology - Part Two*. Chris encourages us to look at legacy code as more of an opportunity than a burden. He'll provide various helpful ways to make your way through old code, including real-world examples that employ modern testing frameworks.

Maxwell Ivey is sharing his experiences on using image-based content in *Emoticons, GIFs, and Stickers* in our Barrier Bytes column this month. Any developer hoping to make their code readable to all will want to check this out. Max discusses the process of using a screen reader to uncover what images are embedded in text or websites he uses. Your take away will be how with a few simple changes can make a difference.

And finally{} we have *The Heart of the Code*.



Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit https://phpa.me/write or contact our editorial team at write@phparch.com and get started!

Stay in Touch

Don't miss out on conference, book, and special announcments. Make sure you're connected with us.

- Subscribe to our list: https://phpa.me/sub-to-updates
- Mastodon: @editor@phparch.social
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download the Code Archive:

https://phpa.me/October2023_code

GraphQL APIs with PHP: A Guide to Schema Design and Execution

Sarah Aburu

This article is your passport to the world of GraphQL in PHP. We'll embark on a journey through each aspect of GraphQL, from its basics to advanced techniques. You'll discover how to set up your development environment, define schemas, write resolvers, and handle queries and mutations with PHP. We'll dive deep into security with authentication and authorization, and you'll learn to manage errors and optimize queries. Real-world examples and case studies will showcase practical applications. We'll wrap up with best practices, testing, and a plethora of resources. Get ready to master GraphQL in PHP and build efficient, flexible APIs.

Introduction

If you've been navigating the ever-evolving landscape of web development, you've probably heard whispers about GraphQL and how it's changing the game. In this article, we'll embark on a journey to demystify GraphQL and explore why it's shaking up the world of APIs. So, pull up a chair because there's a lot to uncover.

Brief Explanation of Graphql and Its Advantages Over Rest

Let's start with the fundamentals. Simply put, GraphQL is a querying language for your API. GraphQL gives you the ability to ask for exactly what you need, no more, no less, in contrast to conventional RESTful APIs, where you frequently get more or less data than you actually need. There are no longer any unnecessary ingredients on your plate, just like when you go to a restaurant and order exactly what you want.

```
Query {
  User(id: 123) {
    Name
    Email
    Posts {
      Title
      Content
    }
  }
}
```

Importance of Using PHP for Graphql APIs

Now, you might be wondering, "Why PHP for GraphQL?" Well, PHP isn't just for web development; it's your trusty sidekick for crafting powerful GraphQL APIs. With PHP, you get the flexibility and scalability needed to build APIs that can handle complex data queries and mutations. It's like having a toolbox full of versatile tools, ready to build the GraphQL server of your dreams.

Purpose of the Article

So, why are we here? Our mission is to guide you through the exciting journey of combining GraphQL and PHP. We'll start from square one, teaching you how to set up the environment, define schemas, write resolvers, and handle authentication. Along the way, we'll sprinkle in some best practices, real-world examples, and tips to help you avoid common pitfalls.

By the time you reach the end of this article, you'll understand the ins and outs of GraphQL in PHP and be armed with the knowledge to create efficient, flexible, and secure APIs that can take your projects to the next level. Let's dive in!

Getting Started with Graphql and PHP

Now that we're all pumped up to explore the world of GraphQL in PHP, let's roll up our sleeves and get started.

Installing Necessary Tools (e.g., PHP, Graphql Library)

The appropriate equipment must be at our disposal before we set out on our GraphQL journey. Make sure to install

PHP on your computer if it isn't already configured. PHP is easily downloaded from the official website or by using a package manager for macOS like Homebrew.

Next, we require a GraphQL library to use. There is an excellent library for PHP called "webonyx/graphql-php." Open your terminal and enter the following commands to obtain it:

```
composer require webonyx/graphql-php
```

Composer, the PHP package manager, will fetch and install the library for you. Now, you're armed with the GraphQL powers of PHP.

Setting Up a Basic PHP Project Structure

Organizing your project is crucial for a smooth GraphQL journey. Let's create a simple project structure to keep things tidy. Here's a minimal setup:

```
Your-graphql-project/
│
├── index.php          # Your GraphQL entry point
│
├── schema/
│   ├── types/         # Define your GraphQL types here
│   ├── queries/       # Create your queries
│   ├── mutations/     # Implement mutations
│   └── schema.php     # Combine your types, queries,
│                      #      and mutations
└── resolvers/
    ├── QueryResolver.php      # Resolver for queries
    │
    └── MutationResolver.php   # Resolver for mutations
```

This structure separates your GraphQL schema definitions from your resolver logic. It helps maintain a clean and organized codebase, making it easier to understand and maintain your GraphQL API.

With your tools installed and project structure in place, we're now ready to dive into the nitty-gritty of GraphQL and PHP. Stay tuned for the next section, where we'll explore GraphQL schemas.

Understanding Graphql Schemas

In our journey to master GraphQL with PHP, we've reached a pivotal point: understanding GraphQL schemas. This is where we define the structure and rules of our API. So, let's roll up our sleeves and get into it.

Defining Types (scalar, Object, Input, Etc.)

In GraphQL, types are the building blocks of your schema. They define what kind of data you can request and what the response will look like. There are several types to know:

- **Scalar Types**: These represent atomic values like strings, integers, or booleans. They're the simplest types and include things like `String`, `Int`, `Boolean`, and more.

```
Type User {
  Id: ID!
  Name: String!
  Age: Int!
  isActive: Boolean!
}
```

- **Object Types**: These define the shape of your data. For example, `User` is an object type with fields like `id`, `name`, and `age`.
- **Input Types**: Used for passing arguments to queries and mutations, they're like blueprints for data you want to send.

```
Input UserInput {
  Name: String!
  Age: Int!
}
```

Establishing Relationships Between Types

One of GraphQL's superpowers is handling relationships between types seamlessly. Let's say a `User` can have multiple `Posts`. You can establish this relationship in your schema:

```
Type User {
  Id: ID!
  Name: String!
  Posts: [Post]  # A user can have multiple posts
}

Type Post {
  Id: ID!
  Title: String!
  Content: String!
  Author: User # Establish connection to author (User)
}
```

Handling Complex Data Structures

GraphQL can handle complex data structures effortlessly. You can nest types within each other to represent intricate data relationships. For example, a blog with authors, posts, and comments:

```
Type Blog {
  Id: ID!
  Title: String!
  Authors: [User]  # List of authors
  Posts: [Post]    # List of posts
}

Type Comment {
  Id: ID!
  Text: String!
  Author: User     # Comment's author
  Post: Post       # The post the comment belongs to
}
```

With these powerful schema-building capabilities, GraphQL becomes a flexible and expressive way to define your API.

Writing Resolvers

Now that we've built our GraphQL schema, it's time to bring it to life by writing resolvers. Think of resolvers as the brain of your GraphQL API. They're responsible for fetching and manipulating the data based on the queries and mutations you define.

Defining Resolver Functions in PHP

In PHP, resolvers are plain functions that correspond to each field in your GraphQL schema. For example, if you have a `User` type with a `name` field, you'll need a resolver for `name`. Here's a simple PHP resolver:

```
// Resolver for the 'name' field of the 'User' type
function resolveUserName($user) {
    return $user->getName();
}
```

Linking Resolvers to Schema Types

To make your resolvers work their magic, you need to link them to your schema. This is where the magic happens. You'll connect each field in your schema to its respective resolver. (See Listing 1)

Listing 1.

```
use GraphQL\Type\Definition\ObjectType;
use GraphQL\Type\Definition\Type;

// Define the 'User' type
$userType = new ObjectType([
    'name' => 'User',
    'fields' => [
    'name' => [
        'type' => Type::string(),
        // Link to the resolver function
        'resolve' => 'resolveUserName',
    ],
        // ... other fields ...
    ],
]);
```

Error Handling and Data Validation in Resolvers

Resolvers not only fetch data but also handle errors gracefully. You can throw exceptions in your resolvers to convey issues to the client. GraphQL's structured errors make it easy to communicate problems in a friendly way.

```
function resolveUserName($user) {
    if ($user->isBlocked()) {
        throw new \Exception("This user is blocked.");
    }
    return $user->getName();
}
```

Data validation is crucial, too. Ensure the data you're returning matches the expected type and shape defined in your schema. This helps prevent unexpected surprises in your API responses.

With resolvers in place, your GraphQL API is ready to respond to queries and mutations.

Handling Queries and Mutations

Now that we've got our resolvers set up, it's time to unleash the power of GraphQL by understanding how to handle queries and mutations.

Defining and Executing Queries

GraphQL is all about asking for the exact data you need. To do this, you'll define your queries in your schema. For example, if you want to get a user's name and email:

```
Type Query {
  getUser(id: ID!): User
}
```

Now, let's execute this query in PHP:

```
$query = '
    {
      getUser(id: 123) {
        name
        email
      }
    }
    ';

$result = GraphQL::executeQuery($schema, $query)
    ->toArray();
```

Implementing Mutations for Data Manipulation

Mutations in GraphQL are like the "edit" or "delete" operations in REST. You can define them in your schema to manipulate data. For instance, creating a new user:

```
Type Mutation {
  createUser(input: UserInput!): User
}
```

And executing this mutation in PHP: (See Listing 2)

Listing 2.

```
$mutation = '
    Mutation {
      createUser(input: {
        name: "John Doe"
        age: 30
      }) {
        Name
        Age
      }
    }
';

$result = GraphQL::executeQuery($schema, $mutation)
                ->toArray();
```

Best Practices for Structuring Queries and Mutations

When structuring queries and mutations, it's essential to keep your schema organized and follow best practices. Group related queries and mutations logically, use clear and descriptive names and document them thoroughly.

Here's a GraphQL convention you'll often see:

- **Queries**: Used for fetching data. Keep them read-only and idempotent, meaning they don't change data.
- **Mutations**: Used for data modifications. They're your write operations and should be explicit about what they do.

```
Type Mutation {
  # Good
  createUser(input: UserInput!): User
  # Good
  updateUserName(id: ID!, name: String!): User
  # Avoid vague names
  addUser(name: String!): User
  # Avoid generic names
  modifyData(data: DataInput!): Data
}
```

By following these practices, you'll create a GraphQL API that's clean, maintainable, and a joy to work with.

Authentication and Authorization

In the world of GraphQL, security is paramount. We need to ensure that the right people have access to the right data. Here's how you can implement authentication and authorization:

Implementing Authentication Mechanisms

Authentication is the process of confirming the identity of users or systems trying to access your GraphQL API. There are various methods you can employ:

- **JWT (JSON Web Tokens)**: A popular choice, JWTs allow you to securely transmit user information between the client and server. You can use libraries like `firebase/php-jwt` to work with JWTs in PHP.

```
// Example JWT generation
use \Firebase\JWT\JWT;
$token = JWT::encode($payload, $secretKey, 'HS256');
```

- **OAuth**: If your app relies on third-party authentication (e.g., Facebook or Google), OAuth is a robust option. Libraries like `thephpleague/oauth2-client` can help you integrate OAuth into your PHP application.

```
// Example OAuth2 client setup
use \League\OAuth2\Client\Provider\GenericProvider;
$provider = new GenericProvider([
    'clientId' => 'YOUR_CLIENT_ID',
    'clientSecret' => 'YOUR_CLIENT_SECRET',
    'redirectUri' => 'YOUR_REDIRECT_URI',
]);
```

Enforcing Authorization Rules Within Graphql

Once you know who your users are, you need to ensure they only access what they're allowed to. This is where authorization comes in. GraphQL's flexibility allows you to implement fine-grained authorization rules at the resolver level.

```
function resolveSensitiveData($user, $context) {
  if ($user->canAccessSensitiveData($context['user']))
  {
    return $user->getSensitiveData();
  } else {
    $msg = "No permission to access this data.";
    throw new \Exception($msg);
  }
}
```

In this example, the resolver checks if the requesting user has permission to access sensitive data.

To enhance security further, consider using libraries like `spatie/laravel-permission` or `PHP-Gatekeeper` for role-based access control (RBAC) and permissions management.

By thoughtfully implementing authentication and authorization mechanisms, you'll fortify your GraphQL API against unauthorized access and ensure data privacy.

Error Handling and Logging

As you forge ahead with GraphQL in PHP, you'll encounter the need to manage errors effectively and keep a close eye on what's happening under the hood.

Dealing with Errors in Graphql

GraphQL provides a structured and user-friendly way to handle errors. When something goes wrong in your API, you can throw exceptions with clear messages to communicate the issue to the client.

```
function resolveUserName($user) {
    if (!$user) {
        throw new \Exception("User not found.");
    }
    return $user->getName();
}
```

GraphQL's response will include detailed error messages, making it easier for clients to understand and react to issues gracefully.

Logging for Debugging and Monitoring Purposes

Logging is your trusty sidekick in the world of debugging and monitoring. It's crucial to keep tabs on what's happening

in your GraphQL application. You can use libraries like Monolog for PHP logging. (See Listing 3)

Listing 3.

```
use Monolog\Logger;
use Monolog\Handler\StreamHandler;

// Create a logger
$logger = new Logger('graphql');
$handler = new StreamHandler('file.log', Logger::DEBUG));
$logger->pushHandler($handler);

// Log something
$logger->info('Query executed successfully.');
```

Logging can help you:

- Debug issues by providing insights into your application's flow.
- Monitor performance and spot bottlenecks.
- Trace requests to pinpoint problematic queries or mutations.

Remember to log not only successful executions but also errors and exceptional cases. This will provide a comprehensive view of your GraphQL API's behavior.

By mastering error handling and logging, you'll have the tools needed to maintain a robust GraphQL API that's reliable and easy to troubleshoot.

Optimizing Graphql Queries

Optimizing your GraphQL queries is like fine-tuning a race car engine. It's about getting the most out of your API while keeping things efficient and responsive.

Techniques for Efficient Data Retrieval

1. **Batching**: Whenever possible, fetch related data in a single request. GraphQL allows you to request multiple resources in a single query, reducing the number of round trips to the server. (See Listing 4)

Listing 4.

```
# Fetch user data along with their posts in one go
{
  User(id: 123) {
    Name
    Email
    Posts {
      Title
      Content
    }
  }
}
```

2. **Pagination**: When dealing with large datasets, implement pagination to limit the amount of data returned in a single query. Use arguments like `first` and `after` for more control.

```
# Fetch the first 10 posts
{
  Posts(first: 10) {
    Title
    Content
  }
}
```

Avoiding Over-fetching and Under-fetching

1. **Over-fetching**: With GraphQL, you request only the fields you need, preventing over-fetching. This means no more fetching unnecessary data from the server.

```
# Fetch only the user's name, not the entire profile
{
  User(id: 123) {
    Name
  }
}
```

2. **Under-fetching**: Conversely, GraphQL ensures you don't under-fetch by allowing you to request related data explicitly. No more back-and-forth to fetch missing details. (See Listing 5)

Listing 5.

```
# Fetch user details along with their posts
{
  User(id: 123) {
    Name
    Email
    Posts {
      Title
      Content
    }
  }
}
```

By mastering these techniques, you can optimize your GraphQL queries for efficient data retrieval, delivering a snappy user experience without wasting resources.

Testing Graphql APIs in PHP

Testing is the guardian of your GraphQL API's quality. It ensures that your code works as expected, from individual components to the entire API.

Unit Testing for Resolvers and Schema

Unit testing focuses on verifying the correctness of individual parts of your code, such as resolvers and types in your

GraphQL schema. PHP testing frameworks like PHPUnit are handy for this task.

Here's an example of unit testing a resolver: (See Listing 6)

Listing 6.

```
1. Use PHPUnit\Framework\TestCase;
2.
3. class UserResolverTest extends TestCase
4. {
5.     public function testUserNameResolver()
6.     {
7.         // Create a mock user object with a name
8.         $user = (object) ['name' => 'John Doe'];
9.
10.        // Invoke the resolveUserName function
11.        $result = resolveUserName($user);
12.
13.        // Use PHPUnit assertions to check the result
14.        $this->assertEquals('John Doe', $result);
15.    }
16. }
```

You can similarly write tests for your types, ensuring they produce the expected data.

Integration Testing with Graphql Clients

Integration testing takes your testing efforts up a notch by validating the behavior of your entire GraphQL API. Tools like PHPUnit or Postman can help you send requests to your API and check the responses.

Here's an example of an integration test using PHPUnit and Guzzle to make requests to your GraphQL endpoint: (See Listing 7)

Integration testing ensures that your API behaves correctly when real requests are made.

By combining unit tests for individual components and integration tests for the entire API, you can have confidence in the reliability of your GraphQL API in PHP.

Best Practices and Tips

Mastering GraphQL in PHP isn't just about code; it's also about following best practices and adopting smart strategies.

Naming Conventions for Types and Fields

1. **Be Descriptive**: Choose clear and descriptive names for your types and fields. This makes your schema more self-documenting.

```
Type BlogPost {
  postTitle: String
  postContent: String
}
```

2. **Consistency**: Stick to a naming convention across your schema. Whether you use CamelCase, snake_case, or another style, be consistent. (See Listing 8)

Handling Versioning and Backward Compatibility

1. **Avoid Breaking Changes**: Be cautious when making changes to your schema. Try to avoid breaking changes that could disrupt existing clients.
2. **Versioning**: Consider versioning your GraphQL API. It allows you to introduce changes without affecting existing clients. You can use URL versions (e.g., `/v1/graphql`) or custom headers to handle versions.

Listing 7.

```
1. use PHPUnit\Framework\TestCase;
2. use GuzzleHttp\Client;
3.
4. class GraphQLApiTest extends TestCase
5. {
6.     public function testGetUser()
7.     {
8.         // Replace this with the actual API endpoint URL
9.         $apiEndpoint = 'https://your-api-endpoint/graphql';
10.
11.        // Create a Guzzle HTTP client
12.        $client = new Client();
13.
14.        // Prepare the GraphQL query
15.        $query = '{
16.            getUser(id: 123) {
17.                name
18.                email
19.            }
20.        }';
21.
22.        // Send a POST request to the GraphQL API
23.        $response = $client->post($apiEndpoint, [
24.                    'json' => [
25.                        'query' => $query,
26.                    ],
27.        ]);
28.
29.        // Parse the response JSON
30.        $contents = $response->getBody()->getContents();
31.        $data = json_decode($contents, true);
32.
33.        // Assert that the user's name
34.        //   matches the expected value
35.        $actual = $data['data']['getUser']['name'];
36.        $this->assertEquals('John Doe', $actual);
37.    }
38. }
```

Listing 8.

```
1. Type BlogPost {
2.   postTitle: String
3.   postContent: String
4. }
5.
6. Type User {
7.   userName: String
8.   userEmail: String
9. }
```

```
// Handling versions in your PHP GraphQL server
$versionHeader = $_SERVER['HTTP_X_API_VERSION'];
if ($versionHeader === 'v1') {
 // Handle requests for version 1 of your API
} else {
 // Handle other versions / provide a default behavior
}
```

Performance Optimization Strategies

1. **Caching**: Implement caching mechanisms to store frequently requested data. Tools like Redis or Memcached can help reduce the load on your server.
2. **Batching**: As mentioned earlier, batch-related data requests to minimize round trips between the client and server. This can significantly improve performance.
3. **Database Optimization**: Optimize database queries to reduce query times. Use indexing, query optimization, and efficient database design.

```
// Example of using Laravel's Eloquent ORM
// for database optimization
$users = User::with('posts')->get();
```

By following these best practices and employing smart strategies, you can build and maintain efficient, user-friendly GraphQL APIs with PHP.

Real-world Examples and Case Studies

To truly appreciate the power of GraphQL in PHP, let's dive into some real-world scenarios where it has made a significant impact.

Demonstrating Practical Applications of Graphql in PHP

1. **E-commerce Platforms**: GraphQL shines in e-commerce, where product data can be highly structured. PHP-based platforms like WooCommerce have leveraged GraphQL to allow merchants to efficiently retrieve product details, manage inventory, and streamline the shopping experience. (See Listing 9)

Listing 9.

```
# Fetch product details
{
  Product(id: 123) {
    Name
    Price
    Description
    Reviews {
      Rating
      Comment
    }
  }
}
```

2. **Content Management Systems (CMS)**: PHP-powered CMS like WordPress have adopted GraphQL to give content creators fine-grained control over retrieving data. This flexibility enables the creation of dynamic and personalized content-rich websites. (See Listing 10)

Listing 10.

```
# Get blog post content
{
  Post(id: 456) {
    Title
    Content
    Author {
      Name
    }
  }
}
```

Showcasing Successful Implementation Stories

1. **GitHub**: GitHub's GraphQL API has been a game-changer for developers worldwide. It allows them to request exactly the data they need, reducing API load and improving performance. This has set a remarkable example of GraphQL's effectiveness in large-scale systems.
2. **Shopify**: As a leading e-commerce platform, Shopify's GraphQL API empowers merchants to customize their online stores efficiently. It has become a pivotal part of Shopify's success story, serving millions of businesses.

These examples and success stories underscore how GraphQL in PHP has transformed various industries, from e-commerce to content management. It offers developers and businesses the flexibility, efficiency, and performance needed to thrive in today's digital landscape.

As we wrap up our journey through GraphQL in PHP, let's summarize the key takeaways and encourage further exploration and experimentation.

Conclusion

We've embarked on a thrilling journey through the realm of GraphQL in PHP, unraveling its power and versatility. As we wrap up, let's recap the key lessons learned:

Recap of Key Takeaways

- GraphQL offers a revolutionary approach to API development, allowing clients to request precisely the data they need, reducing over-fetching and under-fetching.
- PHP is a robust choice for building GraphQL APIs, providing flexibility and scalability to handle complex data queries and mutations.

- Proper schema design, resolvers, and error handling are fundamental to creating a well-functioning GraphQL API.
- Security measures like authentication and authorization are critical for protecting your API and users' data.
- Effective testing, from unit tests for resolvers to integration tests for the entire API, ensures the reliability of your GraphQL PHP application.
- Best practices such as naming conventions, versioning, and performance optimization contribute to a successful GraphQL API.

Encouragement for Further Exploration and Experimentation

As you continue your journey with GraphQL in PHP, remember that the possibilities are limitless. Use the knowledge gained here as a springboard to explore new horizons:

- Dive deeper into advanced GraphQL features like subscriptions for real-time updates and persisted queries for optimization.
- Explore GraphQL libraries and frameworks like Apollo Client and Relay for the front end and Laravel GraphQL or Symfony GraphQL for the back end.
- Contribute to the GraphQL and PHP communities by sharing your experiences and knowledge with others.
- Keep experimenting and building exciting projects, whether crafting blazing-fast e-commerce platforms or dynamic content-rich websites.

Remember, the world of GraphQL is ever-evolving, and your journey is just beginning. Embrace the challenges, celebrate the victories, and never stop exploring the endless possibilities that GraphQL in PHP has to offer.

Additional Resources

As you continue your journey with GraphQL in PHP, these resources will serve as valuable companions, providing guidance, tools, and knowledge:

References to Relevant Documentation and Tutorials

1. **GraphQL Official Website**: Start with the official GraphQL documentation[1] to gain a deep understanding of the GraphQL concepts and best practices.
2. **PHP GraphQL Libraries**: webonyx/graphql-php[2]: The official GraphQL library for PHP.

- Lighthouse PHP[3]: A powerful Laravel package for building GraphQL APIs.

3. **Laravel GraphQL Tutorial**: If you're using Laravel, the Laravel GraphQL tutorial[4] is an excellent resource to get you started.

Links to Helpful Tools and Libraries

1. **Apollo Client**: Explore Apollo Client[5] for JavaScript, a popular GraphQL client for front-end development.
2. **Insomnia**: Use Insomnia[6] as a versatile GraphQL client tool for testing and debugging your API.
3. **Postman**: For more comprehensive API testing, Postman[7] offers a GraphQL-friendly environment.
4. **GraphQL Playground**: The GraphQL Playground[8] is a helpful in-browser IDE for interacting with GraphQL APIs during development.

These resources will be your trusty companions on your journey to master GraphQL in PHP. Whether exploring new concepts, finding solutions to challenges, or discovering the latest tools, they'll support your growth.

As you venture forward, remember that learning is a continuous process. Embrace curiosity, experiment, and never hesitate to seek help from the thriving GraphQL and PHP communities.

May your GraphQL endeavors be fruitful and your coding adventures endlessly exciting!

Sarah Aburu is a seasoned technical writer with a passion for simplifying complex programming concepts. With over two years of experience, They've delved into the intricacies of PHP and a range of other programming languages, translating them into user-friendly documentation. Their mission is to empower developers with clear, concise, and well-structured guides, making their coding journey smoother and more enjoyable. When they're not writing about code, you can find them exploring the latest tech trends or experimenting with new programming challenges.

1 https://graphql.org/learn/
2 https://github.com/webonyx/graphql-php
3 https://lighthouse-php.com
4 https://phpa.me/lighthouse
5 https://www.apollographql.com/docs/react/
6 https://insomnia.rest/graphql/
7 https://phpa.me/gql-dev
8 https://phpa.me/playground

The Shared Thinking Patterns Between Programmers and Artists

Sydney Reinert

When You Get Home After a Workday, You May Often Want to Unwind to One of Your Favorite Video Games to Relax. As a Player, You Usually Pay Attention to Two Main Aspects of the Game. Those Being Its Mechanics and Art. You Marvel At the Technical Expertise of the Programmers Who Turned Fantastical Concepts Into a Playable Reality. Simultaneously, You Admire the Dedication to Details That Artists Incorporated As They Breathed Life Into a Virtual World.

Even Though We Normally Differentiate Between Programming and Art Skills, There is Often a Very Overlooked Collaboration That Needs to Happen to Create an Immersive Gaming Experience. Both Teams Need to Have a Mutual Understanding of What the Other Discipline Does. There is an Essential Flow Between Programmers and Artists. in This Article, I Want to Highlight the Flow Between Programmers and Artists As a Means to Discuss the Similarities Between the Two. by Discussing These Similarities, I Want Readers to Have a Deeper Understanding of Each Other's Respective Roles to Gain a Number of Advantages, Including Stronger Collaboration, Communication, and Mutual Respect. These Advantages Should Allow Any Team of Developers and Artists to Create Products Better Than If They Have No Idea What the Others Did.

Introduction

When I started my college education at Boise State University, I didn't expect it would do such a fantastic job of merging the arts and (computer) sciences. As a student in the Games, Interactive, Media, and Mobile Technology (GIMM) program, I am being pushed into worlds that focus on both programming and art. I am being asked to stretch my abilities in both areas. As a result of stretching my abilities, I have learned to be open to new learning opportunities regardless of the associated discomfort. As my peers and I learn about these topics together, a sense of partnership has formed. One is between individuals who are more programming-centric and those who are more art-centric in their abilities. We all have a collective understanding of what the other's roles are going to be in a project. Due to this shared understanding, group projects are much smoother due to fewer roadblocks commonly associated with miscommunication.

As I continue through the GIMM program, though, students start to select either programming or art as their strength. When students get more involved with their preferred strengths, I see a rift that begins to form between the two disciplines. This rift mirrors the physical separation of different disciplines in areas like universities and office buildings. As students enter the professional work field, this separation grows even further. As an individual gains more knowledge about the field they are studying, the more they get involved with it, which is not bad at all! However, as a result, the gap in understanding between the other fields of study widens.

This phenomenon of disconnect between roles is something we see happen in all walks of life. You can see it happen in many technology disciplines, such as web development, server administration, game development, and more! If this is a naturally occurring phenomenon, why should we be concerned? We should care because it hinders the final products we can design and implement as creators. I firmly believe that bridging this gap between programmers and artists allows for a deeper understanding of the intricacies of each other's roles, the tools they use, and the unique pipelines that guide their respective work. This knowledge can help further collaboration between the two. I want to help the two disciplines to appreciate the creative effort invested into every project from their combined efforts. To achieve this, I will be using game development as an example to highlight the similarities between programmers and artists. Specifically, I will review the tools and workflow pipelines each must use to make a game. During this, you will see commonalities between the two disciplines and how you can better work with team members different from yours.

Tools of the Trade

A Programmer's Essentials

In this section, I will be focusing on three essentials in a programmer's toolkit. Those are the game engine, code editors, and code management systems. After describing how each tool shapes the landscape of a game, we will explore how tools fall into a programmer's pipeline.

Game Engines

The decision of what game engine to use is usually the first question many

game developers ask themselves. The two most popular engines currently used by many are Unity[1] and Unreal Engine[2]. An engine is the software development environment that helps simplify and optimize the process of game development. Each year, we see these engines become more and more advanced, allowing developers to create new functionality and reduce the time spent on repetitive tasks. A game engine also allows programmers to use various programming languages, such as C++, C#, and Lua to achieve their goals. Furthermore, game engines assist programmers in optimizing their coding process by giving them access to tools such as profiling, debugging, and asset management. (See Figures 1 and 2)

Figure 1.

Figure 2.

Top: Image of the UI layout for Unreal Engine. Showcasing a normal 3rd person game base.

Bottom: Image of the UI layout of Unity, with a small environment example.

Code Editors and Managers

Often, developers will need to use a text editor or Integrated Development Environment (IDE) outside of the game engine they use. Common examples of these tools include Visual Studio Code[3], Sublime Text[4], and IntelliJ IDEA[5] (from the same developers of PhpStorm[6], Jetbrains[7] . These tools provide programmers with a comfortable and efficient workspace for crafting lines of code. They do this with an array of features ranging from syntax highlighting to intelligent code completion to debugging capabilities. Many times, game engines have an integrated feature that allows seamless transitions between working in the engine and writing code. For example, Unity allows easy integration with Visual Studio Code as an IDE.

Version Control Systems (VCS) is a crucial application for collaboration and codebase management. It is often an overlooked and underappreciated tool that programmers and developers use daily. Git[8], a VCS, is used to power platforms like GitHub and GitLab. These systems enable programmers to track code changes, coordinate seamlessly with team members, and manage project versions with ease.

An Artist's Essentials

The toolkit of artists varies widely. In the realm of video game development, it is important to recognize that there is more to it than just 3D modeling. A wide population of very talented and creative individuals breathes life into a sketch. There are animators, rigging artists (artists who apply skeletal structure to 3D models), visual effect artists, and user information artists, who all play pivotal roles in molding that immersive gaming experience. However, we will focus on digital painting and 3D modeling /sculpting to streamline this conversation. It is important to understand that digital painters and 3D artists have different roles with different outcomes; however, the tools that digital painters and 3D artists use can converge with one another, as many of these tools are extremely versatile.

Digital Painting

Digital painting is a versatile art form used extensively in various creative fields. In video game design, concept, environmental, texture, and character artists rely on programs like Adobe Photoshop[9] and Clip Studio Paint[10] (CSP). These software tools enable professional artists to transform ideas into stunning paintings without the mess associated with physical media like paints, inks, or watercolors.

For instance, consider the role of a concept artist. They collaborate closely with writing teams and developers to represent initial ideas visually. Photoshop and CSP grant artists access to digital brush pens, an array of colors, and the freedom to undo their work effortlessly. Mastery of lighting,

1 https://unity.com
2 https://www.unrealengine.com/en-US
3 https://code.visualstudio.com
4 https://www.sublimetext.com
5 https://www.jetbrains.com/idea
6 https://www.jetbrains.com/phpstorm
7 https://www.jetbrains.com
8 https://git-scm.com
9 https://www.adobe.com/products/photoshop.html
10 https://www.clipstudio.net/en/

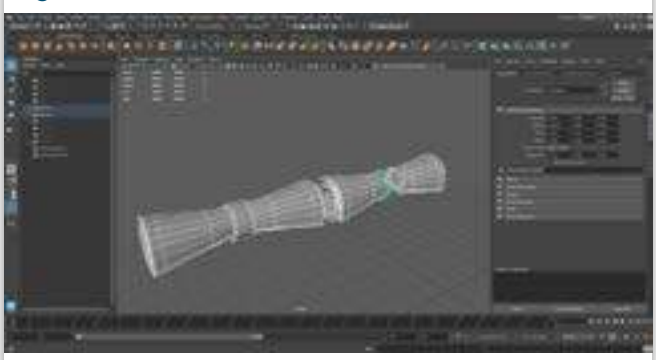
color theory, perspective, and anatomy is crucial, allowing artists to convey the desired emotions in-game art. Suppose writers envision a dark, foreboding swamp with mysterious creatures beneath the water's surface. In that case, it falls upon the concept artist or digital painter to establish these feelings and convey the concept to the rest of the development team.

3d Modeling / Sculpting:

In the realm of 3D art, it's essential to differentiate between modeling and sculpting. 3D modeling, or CAD (Computer-Aided Design), is typically the starting point for 3D modelers when creating assets. 3D modeling is different from digital painting because it is usually a process that happens after the digital painting process. However, when it comes to painting an object, digital painting is reintroduced in the pipeline. This is necessary because when a 3D artist is ready to paint/render their object, they need to understand a program such as Photoshop in order to make accurate textures[11] that will fit the object.

Modeling primarily involves defining basic shapes and structures, often referred to as "low topology." Maya[12], a leading industry program, aids artists in this process. Artists manipulate the shape using tools in programs like Maya without delving into intricate details. Minecraft is a great example of a game that exemplifies 3D modeling's suitability for using basic shapes like cubes to represent various elements. (See Figure 3)

Figure 3.



Maya 3D modeling project of two scrolls. Both show examples of an object with a low amount of faces and vertices.

On the other hand, sculpting is a more intricate process known as "organic modeling." This technique requires a significantly higher number of vertices and faces than normal modeling. ZBrush[13], another industry-leading program, facilitates this action. Sculptors utilize tools to push, pinch, mold, and draw on objects[14], much like working with real-life clay. Complex video games like God of War showcase the exceptional detailing achieved through skilled modeling. Sculpting is best suited for organic subjects like people, animals, and foliage because the high number of vertices and faces allows the artist to make intricate details and shapes.

The Pipeline

The System Development Life Cycle

After reviewing a swath of the tools used by video game programmers and artists, we can now cover the similarities in the pipelines used by them. When it comes to understanding what the creative pipeline process is for both programmers and artists, it is first important to understand what the "System Development Life Cycle[15]" (SDLC) is. The SDLC is a circulatory journey that consists of five steps to achieve efficiency and effectiveness when creating any product. These steps are (1) planning, (2) analysis, (3) design, (4) implementation, and (5) maintenance. As we dive more into details about the pipeline for programmers and artists, the similarities between each pipeline will start to reveal themselves. You will now see how both roles go through an ever-changing cycle of implementation and improvement within the video game industry. (See Figure 4)

Figure 4.

The System Development Life Cycle is a 5 step cycle that many follow for efficiency and effectiveness when working on a product.

Programmers

Planning

In the beginning stages of video game development, programmers collaborate with artists, writers, and other developers. An early discussion arises on the overall idea of a game and what engine the programmers will have to write

11 https://phpa.me/3d-texturing

12 https://phpa.me/autodesk-overview

13 https://www.maxon.net/en/zbrush

14 https://phpa.me/sculpting-brushes

15 https://phpa.me/dev-life-cycle

for. The programming language that developers will use will be determined by the game engine they are working in. Is this game going to be single-player, multiple-player, or a massive multiplayer game online? Will it be in a first-person view or third-person? Answers to these questions will help the programmer better understand what systems within the game they will need to help create or the mechanics that will need to be implemented.

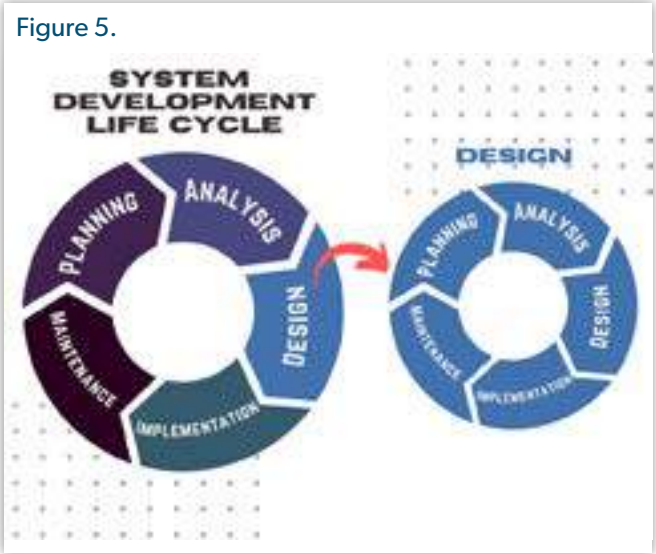
Analysis

With a fundamental understanding of what type of game they will help to create, it is now time to dive into the analysis phase. This will be the research stage for many programmers as they will take the time to fully understand the realm they will be creating in. At this time, programmers work very closely with artists, writers, and producers of the game as they flesh out the game more and more. As the artists are drafting what the world and characters will look like, programmers are thinking about the mechanics. An important aspect of this process is when many programmers begin to question whether they can bring something new to the industry, whether it's in terms of gameplay, the enemies players will come across, and how they might be able to execute a new mechanic never seen before.

Design

The design phase for programming can start with simple building blocks for your game. They will start by making a Unity or Unreal game to begin their development. Many times, these early phases are where programmers establish fundamental files and establish basic systems like movement controls. In many ways, the design phase has its own SDLC within it. (See Figure 5)

Figure 5.



The design stage of the System Development Life Cycle has its own cycle within it before the cycle moves onto the implementation stage.

During this phase, code is regularly tested as a method that will cement the fundamental mechanics that can be further enhanced later on. This phase is also where you see programmers map out interactions between various mechanics, creating a framework of communication within the game's codebase. Once these mechanics are established, a programmer will tackle more advanced game logic. Examples of this advanced game logic may include creating artificial intelligence (AI) systems for non-playable characters (NPC) with which the player can interact. Using a program like GitHub, programmers will be able to implement each iteration they make into a repository that can be shared with other programmers and developers.

Implementation

The implementation stage is when programmers see all of their code come together. All of the programmer's work is brought together simultaneously with the goal that their interactions do not cause any cataclysmic issues. Take for example a system in a game that could be your character's inventory. Implementation of this system might have a programmer see how it will interact with a quest system. This scenario might ask questions about whether receiving an item from a quest will affect the player's inventory. At this point, the implementation stage can be seen as the ultimate test of code. If done incorrectly, programmers can be sent back to the number of issues they need to adjust via refactoring. Or worse, they need to go back to square one. This is where we see something called "stress testing" is used. During stress testing, video game companies will hire individuals to push the game to its limit to ensure no bugs pop up.

Maintenance

The maintenance stage happens once a game has been released; there will inevitably be a flurry of bugs that will arise from the launch. There will always be a player who will attempt to push the limits of the game and find issues. As is common with any software project, maintenance will need to be performed to fix bugs, address changing goals, and improve overall system performance. Many times, it is a small fix, as any large issues are usually ruled out before or very early after launch. When a project is finally launched, that doesn't mean the programmer pipeline is finished. Rather, it starts again. A job that many programmers face is to ensure that the game evolves, adapts, and delivers an ever-more-engaging experience. There is a never-ending list of improvements and new features to be implemented. The program pipeline is a continuous loop of the SDLC. (See Figure 6 on the next page)

Though a project might be considered complete, the improvements and implementations to the project using the SDLC will never stop, making it a continuous cycle.

Artists

Planning

Similar to programmers, artists often work closely with other teams on the project, such as the writing and development

Figure 6.

teams, to understand what is to be created. There are often creative meetings that are held early in the development stage to discuss what a game will be about. Artists will get information from the writers to gain a better understanding of the world and characters they must help visually design. They work with programmers to create a character or environment that will enable player interaction. It often starts with just a single idea that is loosely sketched in a small book or even a napkin! That idea is then transferred to digital painters, who will take it upon themselves to create multiple versions of this idea using digital painting. Whether this be a character or an environment, their role is to help solidify what will be created.

Analysis

We see digital painters and 3D artists do the analysis stage in very similar fashions. Digital painters and 3D artists will usually spend time analyzing other artists' works, real-life examples, and the story of what they are creating art for. The goal here is for the artists to gain a deeper understanding of what exactly they will be creating. There will be questions that have to be answered and ethical considerations to be considered. Many artists make sure they research what they will be creating before they enact the design process.

Design

As digital painters begin adding color to a blank white canvas, 3D modelers begin the process of manipulating spheres and cubes. It starts with very basic shapes, as the artist's only goal at this time is to "block" out the shape. This allows the artists to get a general idea of how their art piece is going. If an artist is drawing or sculpting a human face, this allows them to see if they have the proportions of the face correct, and this early stage allows them to adjust if needed without losing detail or time.

After the blocking-out stage, artists will add details slowly but surely. An artist will never add all the details at once to a single area, as this won't allow them to see the final product as a whole. It is a process of adding and taking away to see what fits best and where they can test the limits of their design. In the final design steps, you will see artists add color, shadows, and highlights. These final touches are what really bring the piece together and are now ready to be implemented.

Implementation

Implementation of art within a video game is quite complex and is a very team-focused aspect of the video game development process. This phase for a digital painter is not as complex as compared to what a 3D artist must do. However, it is important for digital painters in the implementation process to fully see how their artwork fits into the world they are helping create. Final small touch-ups may be made in this section, but this is where their artwork is placed and implemented into something like a loading screen.

As stated earlier, I mentioned how it is important for someone like a 3D artist to keep in mind their topology count. If a 3D asset has a very high topology count, it will cause issues for how the game will run. You will often see 3D artists go through a process called "retopology". Retopology is when you lower the topology count of a model. Once a 3D object has been retopologized correctly, it is now ready to be implemented into the game itself. Usually, this will be the last step in many 3D artists' pipelines as they begin to work on the next asset required.

Maintenance

Every so often, an art asset requires some maintenance. Sometimes, this is done by a completely different artist, as maintenance to an asset means it is essentially getting a facelift. We see this happen quite often with older yet still very popular games such as World of Warcraft. In the past few years, artists have been going in and giving different environments or assets an aesthetic and graphic upgrade compared to their older model that was made back in 2008. Sometimes, this is a requirement as games are ever-evolving and changing. The texture on an object might not load properly, or player interaction with old 3D assets might cause issues that eventually require an artist to go in and solve them. It all comes down to the longevity of a game and what will keep players engaged.

Why is It Important?

I hope, at this point, you can see the similarities between the pipelines of programmers and artists. These similarities can range from working closely with other teams in the early stages to continuously improving an existing project. When both of these groups recognize the similarities between their positions, a previous barrier to effective communication starts to dissolve. A deeper comprehension and appreciation for their pivotal roles in the development process begins to emerge. This can result in a rise of mutual understanding and respect that further fosters collaboration between both entities. A programmer's proof of concept can mirror an artist's sketch. Advances from an initial codebase to an optimized system reflect the progress from an initial concept sketch to a fully rendered 3D model. There is a coalescence between programmers and artists when it comes to their dedication to their craft.

We can see an example of good collaboration between programmer and artist when you level up in a game. A

good feeling should come over the player, especially if they just defeated a high-level boss. Communication between programmers and artists should allow this feeling of triumph to wash over the player through good visuals and code. Both are striving to create a captivating and immersive digital landscape; a shared commitment to this end goal drives the industry forward. This collaboration ensures that players will be able to continue on their gaming journey.

Conclusion

In the ever-changing world of game development, programmers and artists share an unspoken connection that each plays a pivotal role in crafting an immersive virtual experience. Both use the SDLC, which involves comprehensive planning, analysis, design, implementation, and maintenance that assists the team in effectively and efficiently crafting an otherworldly experience. They both work in the ever-evolving world of technology, and their work evolves with advances in the industry. A shared dedication to detail brings a vision closer to reality. A mutual understanding fosters effective communication, deeper appreciation for one another, and collaboration between teams. A good connection between the two can produce high-quality products with polished details and innovative thinking.

Sydney Reinert, a fourth-year student at Boise State University, is currently working towards her undergraduate degree in Games, Interactive, Media, and Mobile (GIMM). By day, she is a graphic designer at a student-led engineering firm at Boise State. Her academic journey in GIMM has immersed her in the worlds of game design, 3D art, accessible technology, and the intricacies of mobile and web applications.

Software Archaeology - Part 2

Chris Tankersley

This month, we are going to look at some tools, tips, and tricks to help you along as you dig through the ancient past of the historic project that you need to work on.

> *Previously, I discussed the topic of Software Archeology, which is the practice of diving into a codebase to figure out the how and sometimes the why. Not only is this a good practice for when you need to join projects that have either been finished by other teams, but also when you start a new job and are unfamiliar with the codebase. You may even revisit a previous project you worked on and curse your former self for not leaving enough notes.*

Get Our Hands Dirty

Reading code can only get you so far. At some point, you will need to make a change to the system, and it is often those first few forays into the code that are the scariest. We might have a bit of an idea of where we need to look, but how do we start to probe the system to find out what is actually happening under the hood?

Find People with First-hand Knowledge

If you are lucky, there will be people that you can rely on to answer questions for you. Do not discount the advice and knowledge of those who came before you, as they may be the only reliable source of information. It is a bit of a joke at work, but my unofficial title is "Oracle" since, in many cases, I am the only one left who remembers why we did something.

Along those same lines, just because someone is not working on a project anymore does not mean you can't consult with them when questions arise. Using my work as an example, at one point, I was the dedicated maintainer of the PHP SDK that we provided to customers. I rewrote a significant portion of the code but no longer directly maintained it. My coworker, who is now in charge of it, regularly questions me on the how and why of the code (I don't know why, though; it's perfect and excellent code).

If you are not sure who to talk to, look at the commit history. If you use GitHub, you can go to the project page and click on "Contributors" to see who worked on the project in the past. If you are using the GitHub command-line client, you can run:

```
git shortlog -sne
```

while inside of a repository to see a list of committers sorted by who has the most commits (not necessarily who has committed the largest amount of changes). It may be possible to contact some of those people for help.

I would caution against reaching out to people who have left the company. You never know the reasons or situation surrounding their departure, so I would stick with talking to people still in the company. I have found that most people are generally happy to give some insight, especially if it is clear you are using them for background information instead of a constant source of support.

Searching Through Files

If we are lucky, someone will be able to point us in the right direction, but what happens when we do not have that luxury? We can resort to good old searching. Thankfully there are plenty of tools we can use to find what we are looking for.

The first thing we need to do is find an appropriate signpost. A good first place is to look for routes in the code since routes can directly point to running code. If looking for a route is too hard, maybe find some text on the page to find the view file (or, so help us, the PHP file that has the HTML and business logic) that gets called. Maybe we need to look for a property name from an API result, which leads us to a class name.

If your IDE has searching capabilities, take a look at what it offers. For Visual Studio Code, which is what I use, you can search specific directories using regex, wildcards, case sensitivity, and whole-word matching. Start with everything turned off, and narrow down as needed. Many times, I will turn on case sensitivity, especially if I know I am looking for a class name or a specific line of text from a page.

Many times, I also drop to the command line to search for things since there are a plethora of very powerful text-searching tools available. The one I default to is `egrep`, which is a variation on the `grep` program.

Both of them allow searching through files via pattern matching, but `egrep` allows automatic usage of regex patterns. You pass it an express to match against as well as a path, and it will return all of the files and matching lines from the files.

```
// Search for "AbstractField" with:
//   - case-insensitivity (-i)
//   - in the src/ folder
//   - recursively (-r),
//   - and return the filename and
//   - line number (-nH)
egrep -nHir "AbstractField" src/
```

If you get too many results, you can drop options like `-i` to turn case sensitivity back on. `egrep` is very fast at what it does, and the plethora of flags allows you to get the desired output.

Once we know where the code we are looking for might be located, we can start to dig in and find out what is going on.

Debugging with Var_dump and Die

When working on a piece of code, one of the quickest ways to see what is happening is to just `var_dump()` and `die()`. By placing `var_dump($variable); die();` in your code, you can print out the contents of a variable and halt the execution of the program at that point. I find this useful when you try to figure out the contents of a variable during the lifecycle of a request, especially complex objects or arrays.

```
$someVariable = "Hello, debugging!";
var_dump($someVariable);
die();
// Code execution halts, and the contents
//  of $someVariable are printed
```

I am also a fan of using `die()` to see where conditional flows go. When you have a block of code for routing or just complex logic that you are unsure about, `die()` can let you find where you are during a request, and then you can augment it with additional `var_dump()` calls to get more information.

```
if ($condition) {
    die('Reached inside the condition.');
} else if ($someOtherCondition) {
    die('We are here now')
}
```

While the two functions work well together, you can also sprinkle in `var_dump()` wherever you need some additional context. (See Listing 1)

Listing 1.

```
1. function complexCalculation($input) {
2.     // Complex calculations...
3.     return $result;
4. }
5.
6. $inputValue = 42;
7. $calculationResult = complexCalculation($inputValue);
8. var_dump($calculationResult);
9. // Output: Contents of $calculationResult are printed
```

You just have to be careful that this additional output does not break other systems. If you are working on the API section of a single-page application, the additional text can break the JavaScript code running on the front end.

Using Symfony/var-dumper for Enhanced Debugging

The built-in `var_dump()` and `die()` methods are nice, but you may want to also check out the Symfony `var-dumper`[1] package. This package provides some additional capabilities that the internal `var_dump()` method lacks, like configurable output and working with output buffers. It installs a new global method called `dump()` that can be called from anywhere in your codebase.

You can install the package using Composer:

```
composer require symfony/var-dumper
```

Using `dump` from `symfony/var-dumper`:

```
require_once '/path/to/vendor/autoload.php';

$complexData = [/* ... */];
dump($complexData);
```

`symfony/var-dumper` also has the ability to do things like capture the output and redirect to the CLI so that you can avoid the issues of `var_dump()` outputting directly in the response. The package also has a helper called `dd()`, which will perform a "dump and die" operation for you, saving some typing.

Step Debugging with Xdebug

`var_dump()`, `die()`, and tools like `symfony/var-dumper` are good tools for what they do, but they can be clunky to use. They dirty the output of your code and completely stop execution when you throw in `die()` or `dd()`. Thankfully, there is a much more powerful option; it is just not included with PHP.

Xdebug[2] is a debugging tool that opens up the ability to do step debugging. Step debugging is a process where a program halts at a specific line of code, called a breakpoint, and the developer can "step" through each line of code and inspect it. Step debuggers, including Xdebug, allow you to see everything else going on, like call stacks and other variables and memory usage.

Installing Xdebug

Installing Xdebug can vary from system to system, but in general, you might be able to find it in your package manager with some variation on `php-xdebug`, `php-8.1-xdebug` (or your PHP version), or just `xdebug`. If your package manager does not supply it, you can probably install it by running:

```
pecl install xdebug
```

If you use a container, check with whoever supplies your base image to find the best way to install Xdebug. In the worst case, you can check the installation page[3] for instructions on installing it.

Once Xdebug is installed, you can verify it loaded by running `php -v`:

```
$ php -v
PHP 8.2.6 (cli) (built: Jun  2 2023 23:59:34) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.2.6, Copyright (c) Zend Technologies
    with Zend OPcache v8.2.6, Copyright (c), by Zend
```

1 https://phpa.me/var_dumper

2 https://xdebug.org

3 https://xdebug.org/docs/install

If it is not showing up after installation, fear not. You may need to enable it manually. We just need to create an `ini` file with the proper settings. The first thing we need to do is find where your additional config files are located. One way to do this is to run the following command:

```
php -i | egrep "Scan this"
```

This will return a line of our PHPInfo output pointing to a folder. In my case, my PHP installation is controlled by Herd[4], so it is located under `/Users/ctankersley/Library/Application Support/Herd/config/php/82/`. Your folder will vary depending on your setup.

If you already saw that Xdebug was loaded, run the following instead to make sure we find the file we will need to edit:

```
php -i | egrep xdebug | egrep ini
```

This will list the file you need to edit, as we will need to tell Xdebug what mode to run in.

Inside that folder we need to make a new file called `xdebug.ini` (or edit the file we just found), and put the following lines:

```
zend_extension=xdebug.so
xdebug.mode=debug
xdebug.start_with_request=yes
xdebug.client_port=9003
```

Save that file, and re-run the `php -v` command again. You should see Xdebug show up. This makes sure that the Xdebug extension is loaded, and that it is running in debug mode.

A> Herd requires a slight change to how you install Xdebug and requires additional files from Homebrew. If you are using Herd, check out the documentation page for Xdebug[5]

Set Up Your Ide

Now we just need to have your IDE talk to Xdebug. This can be different from program to program and can change from version to version, so I would suggest checking how to do this for your editor.

For Visual Studio Code, download the PHP Debug[6] extension and follow the directions there. If you are using PHPStorm, check out their documentation on Configuring Xdebug[7].

Using Xdebug

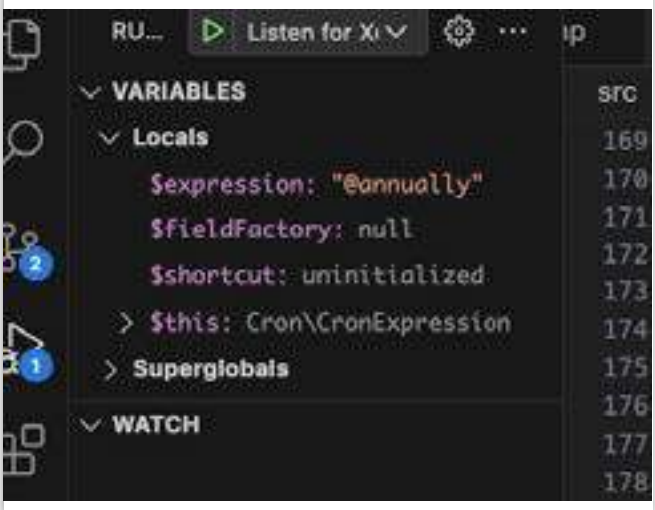
Once you have Xdebug configured, it's time to start using it. I will be showing off Visual Studio Code as that is what I use day-to-day, but PHPStorm is similar.

The first thing you will want to do is set up a breakpoint. In PHPStorm, you can actually have it break automatically at the start, or you can open up a file and click in the gutter, or area to the immediate left of the line numbers, next to the line number you want to stop on. You should see a little red dot appear. This is your breakpoint. (See Figure 1)

Figure 1.

When you run your code, the engine will stop on the breakpoint itself. At this point, we can inspect the variables that are in use. (See Figure 2)

Figure 2.



Here, we have four variables, `$expression`, `$fieldFactory`, `$shortcut`, and `$this`, representing the object we are currently stopped in as I set a breakpoint inside a class. We get a real-time feed of our application's data.

We could do this with `var_dump()` and such, but the power here is that the program is paused instead of stopped. The PHP engine is just waiting for us to tell it to continue, and we can control what we do using the step controller buttons.

In order, these controls allow us to:

- Execute up to the next breakpoint
- "Step Over" to the next line
- "Step Into" a method/function
- "Step Out" of the current method/function
- Stop and start listening again
- Stop debugging completely

4 https://herd.laravel.com
5 https://phpa.me/xdebug-herd
6 https://phpa.me/vscode-xdebug
7 https://phpa.me/phpstorm-config-xdebug

You will use the controls in Figure 3 to dive deeper into the code and control its execution. You will often use the "Step Over" command to go line-by-line when trying to understand exactly what is going on.

Figure 3.

You can set multiple breakpoints as you figure out more of the code. You can set breakpoints in places you want to inspect code, like before and after loops, so you do not have to step through every iteration of the loop.

Xdebug is an awesome tool, and I highly encourage you to learn it. The only downside is that, in many legacy cases, you may not be able to run your code locally or cannot install Xdebug on the location your site is running. It is also not recommended to run Xdebug in production since it can have a performance impact—it is a debugging tool after all.

Careful Digging with Tests

Hopefully, we have found what we were looking for, but we need to be careful. We want to make sure that any changes we make will not impact the rest of the system, so we need to start writing appropriate tests. You have three main avenues to go down when it comes to tests:

- Unit tests
- Functional tests
- Behavioral tests

Unit Tests are designed to test individual blocks of code, like methods on classes. This involves calling the method or function you want to test, giving it different inputs, and making sure that the outputs are what you expect. (See Listing 2)

Listing 2.

```
1. public function testGetParts(): void
2. {
3.     $e = CronExpression::factory('0 22 * * 1-5');
4.     $parts = $e->getParts();
5.
6.     $this->assertSame('0', $parts[0]);
7.     $this->assertSame('22', $parts[1]);
8.     $this->assertSame('*', $parts[2]);
9.     $this->assertSame('*', $parts[3]);
10.    $this->assertSame('1-5', $parts[4]);
11. }
```

In this test, we are making sure that the expression passed in is properly put into the correct parts. We do nothing else but test that one specific expectation. You can use tools like PHPUnit[8] or Pest[9] to write your tests.

Functional tests make sure that multiple systems work together. In this case, you may actually run the full application and test various routes or API calls to see how the entire system itself functions. These are usually slower and more complex tests, but very useful to watch for unintended consequences. Codeception[10] is a popular functional testing suite.

The last type of testing is Behavioral testing, which is a different way to write tests compared to a lot of functional systems. You write "stories" using a specific format, and those stories are turned into tests. I use Behat[11], but Codeception can also run behavioral tests.

For more information on testing, I would check out past issues of php[architect] as well as PHP The Right Way's testing section[12].

Writing tests in a legacy application can be hard. If you have no testing, I highly suggest setting up something like Behat or Codeception for functional/behavioral testing. These can be easier to set up compared to trying to wrangle unit tests into a system that may have many couplings or design decisions that actually make it hard to unit test. When delving into legacy PHP applications, both unit tests and integration tests are vital tools. While integration tests provide broader insights and may be more accessible initially, unit tests offer a focused, detailed understanding necessary for software archeology.

Tests also serve as a pseudo-documentation for how the system is expected to work. Unit tests can show how the developers expected pieces to work together using mocks, and behavioral tests can document how you expect users to interact with the system. No matter what tests you write, you will better understand the overall system as you write more tests.

Related Reading

- *Education Station: Software Archaeology - Part 1* by Chris Tankersley, August 2023. https://www.phparch.com/article/2023-08-education/
- *Education Station: Serverless PHP with Bref* by Chris Tankersley, July 2023. https://phpa.me/education-jul-23
- *Education Station: We are Losing the Browser War* by Chris Tankersley, September 2023. https://www.phparch.com/article/2023-09-education/

8 https://phpunit.de/
9 https://pestphp.com
10 https://codeception.com
11 https://behat.org
12 https://phptherightway.com/#testing

Conclusion

Exploring legacy PHP code is not merely a task; it's an adventure filled with discoveries, challenges, and rewards. With the right approach, tools, and a curious mindset, what appears as an impenetrable labyrinth turns into an exciting exploration.

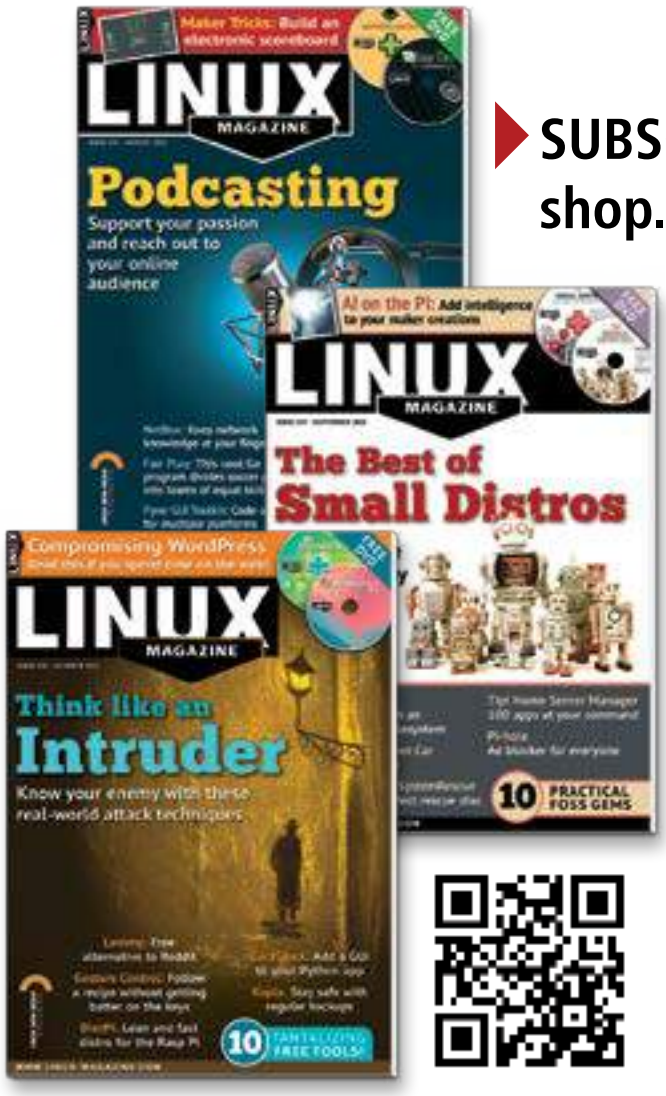
From basic debugging techniques to understanding complex patterns, from employing modern testing frameworks to learning from real-world examples, this guide equips you for the journey ahead.

So, grab your virtual shovel and embark on this exciting expedition. Happy digging!

Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. Chris has worked with many different frameworks and languages throughout his twelve years of programming but spends most of his day working in PHP and Python. He is the author of Docker for Developers and works with companies and developers for integrating containers into their workflows. @dragonmantank

Linux Magazine Subscription

Print and digital options
12 issues per year



SUBSCRIBE
shop.linuxnewmedia.com

Expand your Linux skills:

- In-depth articles on trending topics, including Bitcoin, ransomware, cloud computing, and more!
- How-tos and tutorials on useful tools that will save you time and protect your data
- Troubleshooting and optimization tips
- Insightful news on crucial developments in the world of open source
- Cool projects for Raspberry Pi, Arduino, and other maker-board systems

Go farther and do more with Linux, subscribe today and never miss another issue!

Follow us

@linux_pro
Linux Magazine
@linuxpromagazine
@linuxmagazine

Need more Linux?

Subscribe free to Linux Update

Our free Linux Update newsletter delivers insightful articles and tech tips to your inbox every week.

bit.ly/Linux-Update

The Meaning of "High Trust"

Eric Mann

When many people think about security, they naturally think about entities attacking from the outside. This might be the outside of their application, network, or even organization. We often fail to realize that the most critical threat is often users already inside your system.

> *Never. Trust. The. User.*

I fully recognize that even while saying that, I am a user myself and often placed in a role requiring a high level of trust from my direct team end employer. In these circumstances, I work diligently to ensure that no one needs to *actually* trust me because the system enforces checks and balances against my activity automatically. Said another way—I trust myself not to damage the organization; however, the organization should be able to independently verify the righteousness of my actions without implicit or even explicit trust.

Ultimately, any insider can be a threat to the organization. Any individual who otherwise possesses legitimate access to and is already inside the system could leverage that access—intentionally or otherwise—to bring the said system down. It's easy to assume that all of these individuals have your best interest in mind. Unfortunately, that's sometimes not the case.

Authenticated users are, by definition, inside the application. If you've attended any of my security talks in the past, you know I always tell you never to trust the user. Never trust the user. *Never trust the user.* This isn't because I want you to be in any way rude to the user; you merely have little way to ensure that the user is acting in your best interest when logged into the system. Assume users are malicious unless proven otherwise and take the steps necessary to prevent damage should they go rogue and abuse the system.

Can You Really Trust Your Team?

In September, several Las Vegas casinos were targeted and breached[1] by a malicious actor. These hackers were able to gain access to customer databases through a social engineering attack—they looked up an employee online and impersonated them to IT support in order to gain access to their accounts. The system acted as expected—unknown parties couldn't access the data or harm customer information. However, a human weakness in the system allowed the hackers to leverage that one class of users that still has privileged access to the system—the team.

In general, your team is the first line of defense against an attack. They have access to backups. They have deep knowledge of your codebase. More often than not, they have direct access to the credentials used by your system to interact with remote computers. They can read and edit logs. They control duty schedules and incident notification. In short, your engineering team is the master of all things related to your application.

What happens when a member of that team goes bad? What happens when a disgruntled employee decides to lob a grenade back at the team on their way out the door? What happens when a malicious third party successfully impersonates a team member and gains access to their accounts? What happens when the biggest threat to your system comes from within the very team tasked to protect it?

How Do You Minimize Damage?

The single best way to protect your business from a rogue employee is to diligently and completely embrace the principle of least privilege[2]: An employee (user) should be able to access the resources and information they need to perform their legitimate job action. A programmer likely does not need production access to the system database. Likewise, a DBA doesn't need control over the organization's network infrastructure.

These are convoluted examples, true, but to minimize the impact of any one person trying to damage the system, you need to minimize the amount of the system over which they have control.

Once control is limited to known, well-scoped actions, you must systematically track every action performed in the system. A simple, append-only log can keep a forward-looking record of every event or action within the system. Leveraging a cryptographic signing algorithm, similar to the one used by Chronicle[3], ensures records cannot be removed or changed after they've been

1 https://phpa.me/caesars-mgm

2 https://phpa.me/least-privilege

3 https://github.com/paragonie/chronicle

written to the system. Should anything go wrong, you have a proven record of everything that happened and, in theory, a path to undo any damage.

Is This Really What It's Come To?

It's merely good business to protect your enterprise from the risk of an employee who breaks down and uses their access to take control of your system for nefarious purposes. As your team and company grow, you'll eventually have engineers you have to let go. Some of these dismissals will be mutually beneficial separations. Others will lead to hurt feelings and, potentially, a misguided act of revenge. Yet other times, someone on your staff will see the writing on the wall and take their vengeance before you have "the talk."

Thankfully, this is rare.

The more present threat to your system from existing employees is that of laziness. Someone logs into a staging server from a coffee shop and leaks their password over Wi-Fi. A busy executive falls for a phishing email scam[4] and coughs up their administrative credentials. Someone puts off encrypting their hard drive and leaves their laptop on the bus.

None of these scenarios are situations where an employee goes rogue and attacks the platform, but you have no way to know for sure until you dig in and investigate. On the surface, it looks like a remote developer uploaded malware through a staging environment, an executive embezzles funds, or a new employee is selling your intellectual property to a competitor. I mention these outcomes because I've seen every single scenario here in the wild.

Your team is your most valuable asset. They make the company go. They build your products. They respond to midnight outages and respond to customer complaints. All the same, every member of your team is also a user of your system. As we've said before and will say time and time again:

Never. Trust. The. User.

Give your team the access they need to get the job done. Log everything. Enforce strong passwords, non-repudiation on company emails, hard drive encryption, and strong multi-factor authentication. Do everything you can, not just to enable your team to function but to enable the team to track down an employee who potentially went rogue.

The easiest attack is one that begins inside the city walls—don't ignore the ever-present threat insiders pose to your system.

4 https://www.owasp.org/index.php/Phishing

Eric is a seasoned web developer experienced with multiple languages and platforms. He's been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. You can reach out to him directly via Twitter: @EricMann

HTMX: The Simple Markup Extension We've Been Waiting For

Matt Lantz

Enhanced simplicity and efficiency, what's not to love? HTMX offers both; however, it also comes with some disadvantages. Let's review the pros and cons together and determine how to make it work best for you.

HTMX, or Hypertext Markup Extension, is an external JavaScript library used to allow developers to use AJAX (Asynchronous JavaScript and XML), CSS (Cascading Style Sheets), and HTML (HyperText Markup Language) to manage web pages without necessarily writing JavaScript code. It does this by adding various attributes to DOM elements handled by the 14KB package.

Since its introduction, HTMX has won the hearts of many developers for its simplicity and efficiency. However, as with any other technology, it has advantages and disadvantages. HTMX is leaning into removing as much JavaScript as possible from a codebase like AlpineJS. The largest value gain for many is that all the action logic is nested in your HTML templates. Unlike JavaScript frameworks such as Vue and React, it's not component-oriented; it is simply a processor that parses attributes defined in various DOM elements.

Enhanced Simplicity:

One of the most significant advantages of HTMX is its straightforwardness. It allows developers to work with only HTML for managing AJAX, speedily prototyping, and managing web pages, reducing the complexity of the learning process. It removes build processes we've become accustomed to for frameworks like Vue and React. Its integration is as simple as adding a few attributes to DOM elements.

A strict Content-Security-Policy structure encourages developers not to use default attributes like onclick in the DOM. HTMX overcomes this by adding custom attributes with event handlers defined by the attribute's content. In the case below, we can see how this button will perform a POST request to /example endpoint, and as its *on* attribute states, when the htmx configRequest is performed, its event will have the following parameter added to it.

```
<button hx-post="/example"
hx-on="htmx:configRequest:
event.detail.parameters.example = 'Hello Scripting!'">
Post Me!
</button>
```

Below is a simple display of an alert on click binding.

```
<button hx-on="click: alert('You clicked me!')">
Click Me!
</button>
```

Developing with Livewire, Vue, or even React, you would have at least one component file configuration to handle a search bar engine. If you're using Livewire, you've either contained the search logic in the component or moved it to a separate file, and for Vue/React, you would need an API endpoint. Similarly, with HTMX, you would need an endpoint for your search results. However, it could output more raw HTML, similar to how Livewire works.

Below is a complete search engine component, which would interact with an API endpoint that outputs some HTML. (See Listing 1)

Listing 1.

```
<h3>
    Search Contacts
    <span class="htmx-indicator">
            <img src="/img/bars.svg"/> Searching...
        </span>
</h3>
<input class="form-control" type="search"
       name="search"
       placeholder="Begin Typing To Search Users..."
       hx-post="/search"
       hx-trigger="keyup changed delay:500ms, search"
       hx-target="#search-results"
       hx-indicator=".htmx-indicator">
<table class="table">
    <thead>
    <tr>
        <th>First Name</th>
        <th>Last Name</th>
        <th>Email</th>
    </tr>
    </thead>
    <tbody id="search-results">
    </tbody>
</table>
```

Reduced Complexity:

There are general reductions in complexity when using HTMX since you're no longer having to perform JavaScript builds. Also, you can output HTML rather than creating stripped-down API endpoints outputting JSON, which you

need to convert back to HTML. Overall, for any developer, this means less complexity management across their systems and release cycles. Like Tailwind in the developer experience, while exploring your HTML templates, you also encounter the actual action bindings and logic that your code uses rather than potentially dissecting multiple files.

HTMX will perform adjustments within the DOM itself rather than a Virtual DOM, similar to how Alpine works. The significant difference is that with HTMX, you can bind Ajax actions. The code below will perform a put request to contact/1. This means that rather than having a universal 'axios' binding for forms or having to maintain the logic of various AJAX forms vs. standard forms; you can add an 'hx-put' like attribute to your DOM element. (See Listing 2)

Listing 2.

```
<form hx-put="/contact/1"
      hx-target="this" hx-swap="outerHTML">
  <div>
    <label>First Name</label>
    <input type="text" name="firstName" value="Joe">
  </div>
  <div class="form-group">
    <label>Last Name</label>
    <input type="text" name="lastName" value="Blow">
  </div>
  <div class="form-group">
    <label>Email Address</label>
    <input type="email" name="email"
           value="joe@blow.com">
  </div>
  <button class="btn">Submit</button>
  <button class="btn"
          hx-get="/contact/1">Cancel</button>
</form>
```

Though the above code doesn't handle error responses well, and HTMX documentation recommends using the HTML validation rules in conjunction with backend rules, the above code is simple and easy to replicate. It can elegantly handle a general user form request.

Improved Performance:

HTMX boosts the performance of web applications by controlling specific parts of the webpage that need to be updated via AJAX without refreshing the complete page. As we all understand, users want fast, streamlined experiences and in general, the less page loading we do, the better for many user experiences. Similar to Livewire, HTMX will handle DOM swaps of content based on responses from the server. This is different from Virtual DOM frameworks, which require reprocessing whole sections sometimes and are distinct from a SPA, which will render once and swap on demand. (See Listing 3)

The above code will load from the hx-get route and then perform a DOM content swap. It's simple, but the gain is that no subcomponent files are needed, and you only load what is necessary for the initial page experience.

Seamless Integration:

HTMX can be easily integrated with any backend language or framework, providing high versatility, unlike Livewire, which is entirely Laravel-bound, and Turbo with Ruby on Rails. HTMX is framework agnostic, much like AlpineJS. You could easily interchange the components across any framework and have endpoints driven by any backend. Though there is a slim chance your project will ever switch frameworks, independence is optimal since it reduces the chance of version change impacts between the frameworks you use.

Security:

You may be concerned about security, but there is reasonable documentation covering critical issues like XSS, etc. Overall, HTMX has many aspects covered, but in general, like most security issues, it's up to the developer to know what risks they're taking with their application structure. Adding this to your body will ensure that you can have your CSRF tokens present.

```
hx-headers='{"X-CSRFToken":
  "v1qfudoZYKDQx31jT9yK5R2S9XUNbmZVb8K5RRLN"}'
```

Furthermore, you should review this before diving too deeply, as, in some cases, you could provide too much data in the history engine of HTMX, making your app more open to bad actors. The link https://htmx.org/docs/#security[1] details how to disable history and other features via the HTMX configurations.

Like any new framework or tool, there are some downsides, but thus far, HTMX is demonstrating only a few, but that depends on the type of development you like to focus on. HTMX depends on Server-Side Rendering, which can be a significant drawback, especially for applications that require creating numerous virtual DOM elements with event bindings. This can sometimes get ugly as you'd need to run your bindings after various lifecycle steps of HTMX.

Listing 3.

```
<tr id="replaceMe">
  <td colspan="3">
    <button class='btn' hx-get="/contacts/?page=2"
            hx-target="#replaceMe"
            hx-swap="outerHTML">
      Load More Agents...
      <img class="htmx-indicator" src="/img/bars.svg">
    </button>
  </td>
</tr>
```

1 https://htmx.org/docs/#security

Although HTMX is steadily gaining traction, its community support is still early. The consequence is that certain complex issues might have yet to have immediate solutions or workarounds. If you like pioneering new methods of coding, then this is undoubtedly a framework for you; if not, you may want to give the community more time to grow.

Given the nature of how HTMX is structured, it is likely to have few incompatibilities with other frameworks; as mentioned, you're more likely to have issues loading DOM elements from the response and handling their JavaScript bindings than you will incompatibilities. However, there is a learning curve even to the simplicity of HTMX. Developers focused on SPAs and JavaScript frameworks may have issues with responses returning HTML vs. JSON.

Arguably, HTMX is a form or style of inline scripting in alignment with Alpine, which is generally popular in the Laravel community. Unlike Livewire, it's not very "component" oriented, and naturally, its structure is to be placed into various areas of an application. If you were building an app with Laravel, HTMX would pair wonderfully with Blade components, given there is no real JavaScript to load or propagate in the template layering.

HTMX currently has 20,000+ stars on GitHub, is growing in interest and popularity, and is also getting some attention from decent-sized sponsors. Its tiny footprint of 14KB makes adding and tinkering with components for any web application easy. Will it become more popular than Livewire or Turbo? If they continue simplifying complex component interaction between the live page and backend systems, then most likely.

In summary, HTMX's advantages lie in its simplicity, reduced complexity, and enhanced performance. Its drawbacks revolve around its heavy dependability on server-side rendering and limited community support. If your team is building small to mid-size applications, then HTMX could be a great toolset to adopt and make your standard for handling many interactive aspects of applications. This is even more true for teams that handle multiple projects with varying backend systems. Full-stack developers looking to remove more maintenance work from their projects should also highly consider adopting HTMX. Given the framework-agnostic nature of HTMX, I see it becoming a common standard for applications of all sizes in the next few years.

Matt has been developing software for over 13 years. He started his career as a developer working for a small marketing firm, but has since worked for a few Fortune 500 companies, lead a couple teams of developers and is currently working as a Cloud Architect. He's contributed to the open source community on projects such as Cordova and Laravel. He also made numerous packages and helped maintain a few. He's worked with Start Ups and sub-teams of big teams within large divisions of companies. He's a passionate developer who often fills his weekend with extra freelance projects, and code experiments. @MattyLantz

Emoticons, Stickers, and GIFs

Maxwell Ivey

Images aren't so clear for me. I rely on my screen reader to explain the image, whether an emoticon, sticker, or gif. I'm hoping that this process could be much easier for me and others with a little bit of understanding on the development side.

One of my goals for this column is to make it a place where readers feel comfortable asking questions, including sometimes uncomfortable ones. I am open to helping answer those questions about people with disabilities that you have always wanted to ask.

Eric recently asked me about how the blind deal with emoticons, stickers, and gifs. He wondered if I used them at all. And he asked how my screen reader handles them.

As you might know, he is one of the owners of this magazine. I told him that close friends often ask that same question, and he suggested we address the question here.

Hopefully, you will follow Eric's example and reach out to get your own nagging questions answered.

We talk a lot about alternative text tags in online accessibility, but we rarely discuss these small images that play such a significant role in our daily conversations.

First, I need to break the discussion up into two groups. This is because my experience with stickers, emoticons, and emojis is different than with gifs.

Starting with Gifs

I'm going to start with gifs because they are the least accessible and because I want to discourage you from using them.

Usually, I do know they are there. I just don't know what they are an image of.

Whenever I have tried to search for a gif to add to a social media post, I have been unable to do so.

I imagine this is because gifs are a sort of small moving image. But it could also be that very little effort has been put into describing them.

I would respectfully ask that you not use them on a website, blog, or social media post.

Emoticons and Emojis

Before writing this post, I checked to see if there is a difference between emoticons and emojis. I found out there is a small difference in how they are created. But I'm going to use the terms interchangeably.

The conventions behind the creation of these little images have resulted largely in inclusive descriptions.

The descriptions include things like skin color, gender, age, and number of people in the image.

They tell me things like light-skinned man levitating in business suit or dark-skinned woman juggling colored balls.

Because of those descriptions, I can navigate through the same menu you use to select the emojis I think fit my message best.

I can also enter keywords into the search engine to help me find new applicable emoticons.

I am not always sure of the message I am sending with them, but that's a whole other story.

If someone puts multiple emoticons next to each other without spaces, then my computer will read that as a number followed by the description.

It might say something like three red hearts, five musical scores, or two smiling faces with sunglasses.

However, if the emoticons are different, my screen reader has to announce each individually.

I hope you see how this gets annoying when too many emoticons are included in a single text message, direct message, or email.

It can get even worse with social media posts when multiple emojis are included in your screen name or handle. This is because my screen reader has to announce those images every time it reads a new post.

While I don't know of any standard for the maximum number of emoticons in a piece of content, my personal preference is three or fewer.

Stickers

With stickers, things become a little less predictable. Descriptions of stickers are more reliable than gifs; however, they are not as dependable as emoticons.

This is because these images are more involved, and the description depends on the effort of the designer.

I am able to navigate stickers and select them. I can use the search engine in the store to download sticker packs. I am also able to move between the packs by choosing the most recent set before picking the individual sticker I want.

Sadly, not all the descriptions are reliable. And, in my opinion, some of them exaggerate. Although that usually involves cartoon characters or superheroes.

For this reason, I prefer to use stickers with more expressive descriptions. I also like asking friends what a sticker looks like before I add it to my favorites.

I'm a Peanuts fan, so some of my favorites are Snoopy, a black and white beagle in a plane with goggles titled Adventurous, or Charlie Brown with his arm around Linus titled Friendship.

I haven't dealt with any personalized stickers yet. I would imagine that those people who have blind friends or who have been exposed to people with disabilities will write good descriptions.

Conclusion

While most of the time, people will probably just write something more minimal, like my company logo with a smiley face.

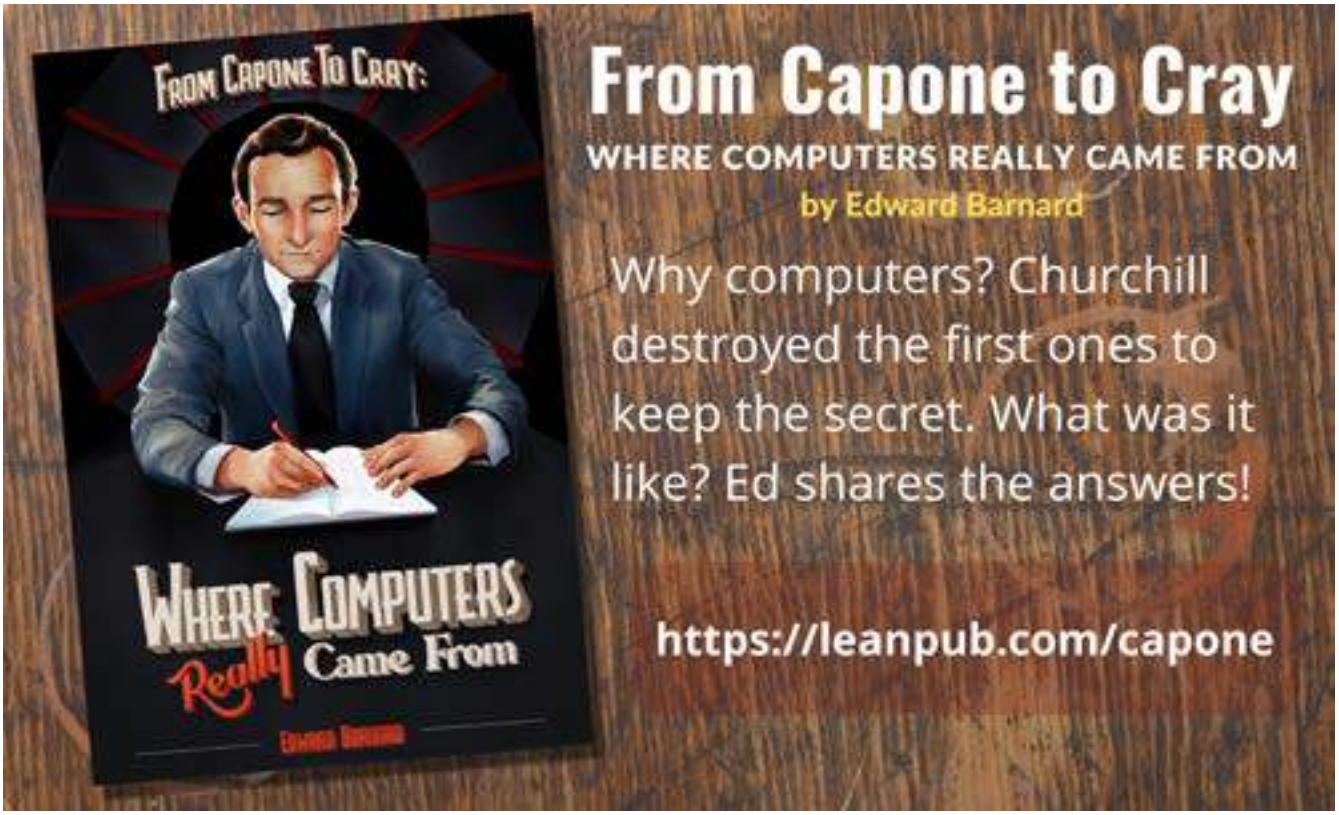
Maybe some of you have created your own stickers or emoticons and could send me a couple to try out.

Just like with emoticons, stickers can gum up my screen reader, announcing all that additional text. So, please use them sparingly.

Emoticons, emojis, and stickers are part of modern-day play. But they're more than that. They are a key part of how we express ourselves in the digital world we live in.

It's great that I can participate right along with sighted people. Hopefully, this post will free you from the fear of using them in your conversations with people like me.

Maxwell Ivey is known around the world as The Blind Blogger. He has gone from failed carnival owner to respected amusement equipment broker to award-winning author, motivational speaker, online media publicist, and host of the What's Your Excuse podcast. @maxwellivey



LONGHORN PHP 2023

Schedule is Live!

3 Days, 3 Tracks

- **Thursday: Tutorials**
- **Friday & Saturday: Talks**

Awesome Speaker Lineup!

- **3 Keynote Speakers**
- **31 Talk & Tutorial Presenters**
- **42 Tutorials & Talks**
- **Open Spaces Discussions**

Plus:

- **Networking Opportunities**
- **Lunch Provided**

AUSTIN TX
NOVEMBER 2-4, 2023

Holiday Inn Austin Midtown
6000 Middle Fiskville Road

Buy tickets @
longhornphp.com

Insertion Sort

Oscar Merida

There's more than one way to sort things? We've looked at two basic sorting algorithms and will continue that exploration this month. Instead of comparing pairs of values in our unsorted arrays, we'll take a different approach and use an insertion sort.

> *Recap*
>
> *For next month, write an implementation of the Insertion Sort, one of the fastest algorithms for sorting "small" arrays. Generate an array of N random numbers. Pick as large a range as you want to work with, but don't make the range too small, and use your comb sort function to order it.*
>
> *Again, generate many such arrays; collect and present data on how long it takes to sort. Show the shortest, longest, and mean times.*

Algorithm

The bubble sort and comb sort are useful as instructional sorts because of a straightforward explanation for how they work. We compare two values, eventually adjacent values, and swap them if the higher one needs to shift towards the end of the array. Do this enough times, and we'll eventually compare all items with their neighbors and stop when we don't need to make any swaps. Now, they're not very efficient for most use cases, so computer science researchers have developed others. The Insertion Sort is efficient for small data sets, relatively simple to implement, and doesn't require more memory than the original list.

How does insertion sort work? Instead of swapping adjacent elements blindly, it builds up a sorted array with each loop through the elements. The first pass looks at the first two elements and then swaps them if needed. Then, it looks at the first three elements, and instead of swapping the third element with the second, we'll look through the first and second elements and insert them "in the right spot." We'll come back to that in a moment. On the next iteration, we loop at the first four elements, find "the right spot" for the fourth element, and insert it between those two elements.

Can we generalize that part? I think so. On the `n`th scan of the array, elements `1` to `n-1` will be sorted in ascending order. We need to insert the value at position `n` in "the right spot" before moving on to look at the `n+1`th element. How do we find the correct place for it? Here, we can use another loop to scan our sorted array from `n-1` to the first element, shifting items in our array to the right until we find an element smaller. Then we go on to the next loop.

See the Wikipedia article for insertion sort[1] for an explanation of the pseudocode below and a helpful visual example.

Listing 1.

```
1. i ← 1
2. while i < length(A)
3.     j ← i
4.     while j > 0 and A[j-1] > A[j]
5.         swap A[j] and A[j-1]
6.         j ← j - 1
7.     end while
8.     i ← i + 1
9. end while
```

The state of the array as it's being sorted should look like this. Note that the front of the array is sorted with each pass.

```
6 12 8 3 10 5 11 9
6 8 12 3 10 5 11 9
3 6 8 12 10 5 11 9
3 6 8 10 12 5 11 9
3 5 6 8 10 12 11 9
3 5 6 8 10 11 12 9
3 5 6 8 9 10 11 12
```

1 https://en.wikipedia.org/wiki/Insertion_sort

Solution

I wrote the solution in Listing 2 based on the pseudocode on Wikipedia. Is that cheating? I don't think so. The pseudocode helped me understand how the algorithm works; ignoring it after that would be wasteful. We're learning about a well-known algorithm and shouldn't be expected to come up with the same answer from scratch. I didn't have pseudocode for swapping items in our array, so I turned to the same `swap_elements()` function used with the bubble and comb sorts.

Listing 2.

```
<?php

function swap_elements(
    array &$list, int $i, int $j
): void {
    // safety check
    if (
        array_key_exists($i, $list)
        && array_key_exists($j, $list)
    ) {
        $tmp = $list[$i];
        $list[$i] = $list[$j];
        $list[$j] = $tmp;
        return;
    }

    throw new \InvalidArgumentException(
        "Invalid index specified"
    );
}

/**
 * We're assuming sequential integer keys
 * @param array<int, scalar> $list
 */
function insertion_sort(array &$list): void
{
    $i = 1;
    while ($i < count($list)) {
        $j = $i;
        while ($j > 0 and $list[$j - 1] > $list[$j]) {
            swap_elements($list, $j, $j - 1);
            --$j;
        }
        ++$i;
    }
}
```

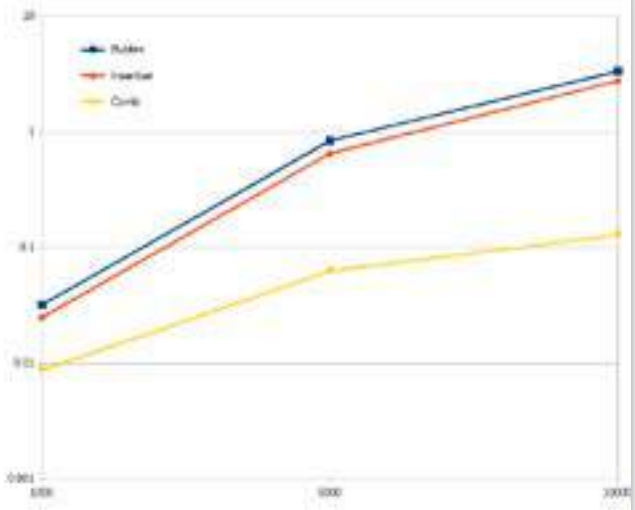
Benchmarks

Now that we know the "best" shrink factor to use, we can benchmark and compare performance to the bubble sort.

Insertion Sort	1000	5000	10000
Fastest	0.023444	0.62636	2.6110
Slowest	0.030364	0.67991	3.0507
Mean	0.024899	0.65251	2.7604

The table below compares the mean performance for the three sorting algorithms we've looked at so far. The insertion sort is faster than bubble sort, but not by much. The comb sort is still a lot faster? Why might this be? I suspect it also suffers from having to make too many comparisons like the bubble sort does. Just not as bad. In fact, for next month, let's look at Shell's sort, which improves on the insertion sort by comparing elements some distance apart. Figure plots the performance on a log scale. In PHP, the comb sort with fewer operations is the clear winner.

Figure 1.



Sort	1000	5000	10000
Bubble	0.0321	0.8414	3.3420
Insertion	0.0249	0.6525	2.7604
Comb^*^	0.00873	0.06275	0.1286

^*^ *The Comb sort numbers from last month looked a bit too good to be true. I re-ran the benchmarks and updated the row with new values. It looks like I didn't pay close attention to where the decimal point was in the older benchmarks. My apologies. (See Figure 1)*

Shellsort

For next month, write an implementation of the Shell's sort, a variant of the insertion sort. Generate an array of N random numbers. Pick as large a range as you want to work with, but don't make the range too small, and use your shell sort function to order it.

Again, generate many such arrays and then collect and present data on how long it takes to sort. Show the shortest, longest, and mean times.

Some Guidelines And Tips

- *The puzzles can be solved with pure PHP. No frameworks or libraries are required.*
- *Each solution is encapsulated in a function or a class, and I'll give sample output to test your solution against.*
- *You're encouraged to make your first attempt at solving the problem without using the web for clues.*
- *Refactoring is encouraged.*
- *I'm not looking for speed, cleverness, or elegance in the solutions. I'm looking for solutions that work.*
- *Go ahead and try many solutions if you like.*
- *PHP's interactive shell (`php -a` at the command line) or 3rd party tools like PsySH[2] can be helpful when working on your solution.*
- *To keep solutions brief, we'll omit the handling of out-of-range inputs or exception checking.*

2 https://psysh.org

Oscar Merida has been working with PHP since version 4 was released and is constantly learning new things about it and still remembers installing an early version on Apache. When not coding or writing, he enjoys RPGs, soccer, and drawing. @omerida

DDD Alley: Create Observability, Part 5: Offline Processes

Edward Barnard

This month completes our "Create Observability" series. We will design, in flowchart form, each of the queue workers. We will describe each event to be emitted by each worker. We'll see a webpage useful for observing queues and events, thus providing observability. As we saw last month, the flowcharts document the business process so that it does not become unknown "tribal knowledge". Finally, we'll discover that the final queue contains precisely what our stakeholders need for customer service resolution.

> *Last month, we presented the "Post Reserve Week Scores" feature. This feature needs to be rewritten due to the unacceptably long "hang time" during webpage load. This hang time is primarily due to sending out dozens or hundreds of email notifications during the page processing.*

This "hang time" is an issue with several different business processes that send out large numbers of email notifications. We, therefore, have several issues to consider as part of the rewrite:

- The "hang time" noted above
- Move ad-hoc PHP generation of email to Twig templates
- Move to an offline processing model
- Discover commonalities between several of our processes generating mass email notifications
- Prepare for better control of email-send bursts

At the same time, we need to design for Observability:

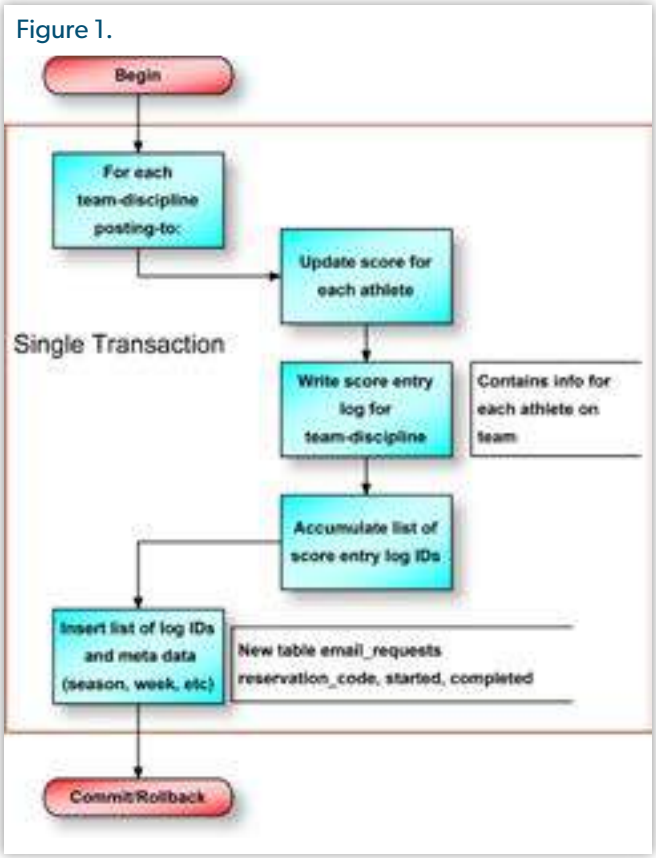
- What events or results are of interest to stakeholders?
- What events or results are of assistance to customer support? (We receive many calls to the effect of, "You never sent my email.")
- What events or results assist developers in performance or problem analysis?

Note that these three questions should be considered independent of each other. All three questions represent requirements. It's important not to conflate domain events (stakeholder events) with developer events needed for problem/performance solving.

Emitted Events

Let's consider developer events first. Why? Because those events can help us during feature development. Figure 1, carried forward from last month, shows our first stage of processing.

Figure 1.



Generally speaking, developer events should show the running process's perspective. These events are like a history trace, showing each worker's step.

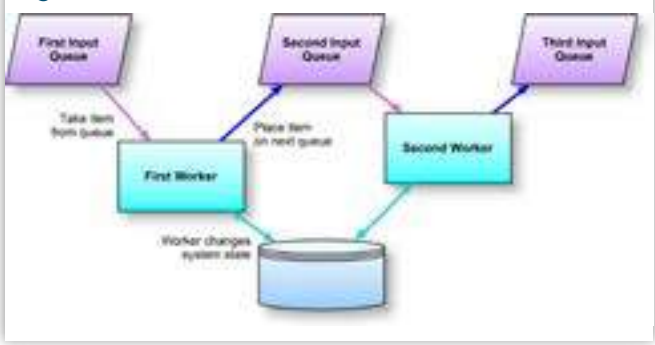
Here are the events to be emitted by the webpage posting Reserve Week scores:

Event	Description	Activity
PostReserveWeek.ProcessingBegan	Form data processing began	league_type_id, season_id, week_id
PostReserveWeek.ProcessingCompleted	Form data processing completed	league_type_id, season_id, week_id
PostReserveWeek.TeamDisciplinePosted	Reserve week scores posted for team-discipline	season_id, week_id, team_id, discipline_id, score_entry_log_id
PostReserveWeek.EmailRequestCreated	Email request with team-discipline list created	league_type_id, season_id, week_id
PostReserveWeek.TransactionRolledBack	Form data processing failed	league_type_id, season_id, week_id, message

The event name and description are both expressed in past tense. The event reports the fact that something happened. The "activity" field lists values that should be captured as part of the event.

Figure 2, also carried forward from last month, shows our "assembly line" approach. The above process (post Reserve Week scores via the web page) places a record in the first input queue. That's the event "EmailRequestCreated".

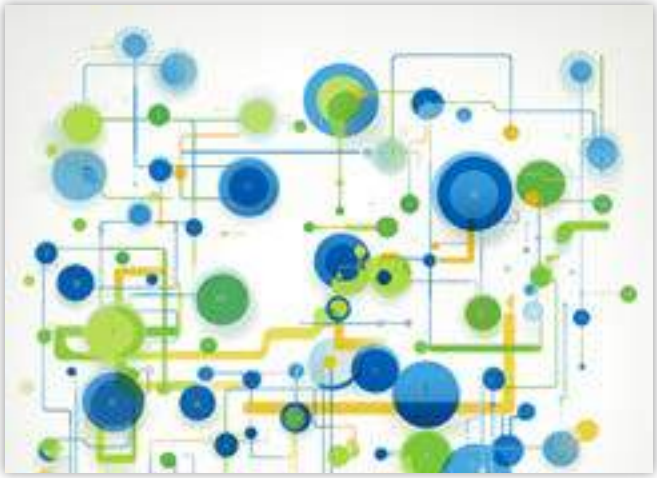
Figure 2.



Email Funnel

While designing this feature, we realized that we needed to send email to one recipient at a time. Rather than blast one message to 20 recipients (for example), we prefer to make 20 copies of the email, one per recipient. That's so we can address the customer-support issue of *one* recipient reporting the email was never received. This way, we can record separate gateway status, timestamp, etc., for each intended recipient.

Given this design, we can send all such emails with a single worker. All email has "to", "from", "subject", and "body". Given



those input parameters, the Simple Worker can take each input, send out the email via our email gateway, mark that input as complete, and move to the next input request. This approach also ensures we capture a copy of what was sent in each case.

However, we have several different business processes generating email. How do we get all emails to a common format for sending? Create an input record containing:

- To, From, Subject
- The twig templates to be used in rendering the message body
- The parameters Twig should use in rendering the message body

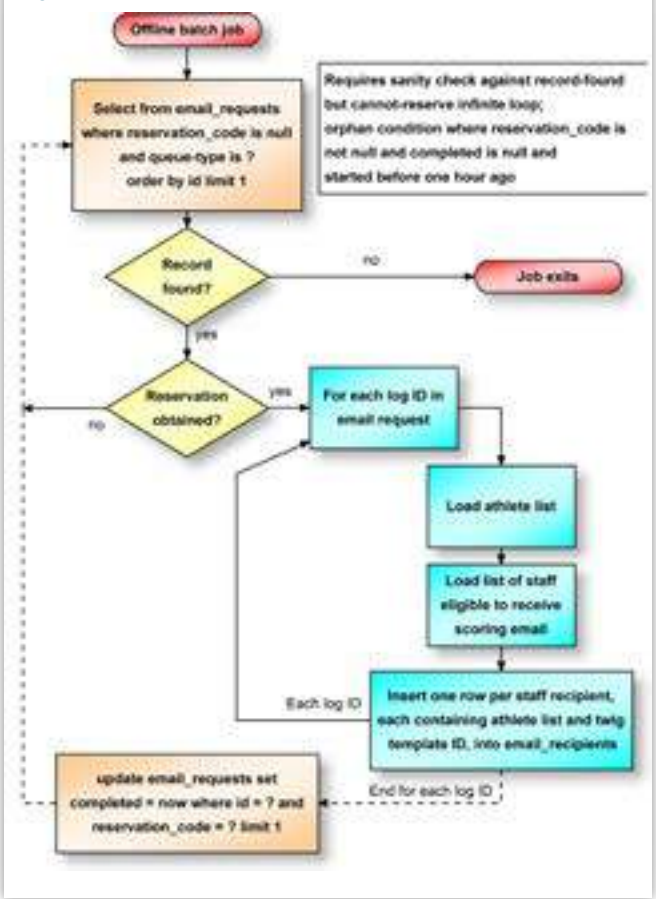
We've just described the third and fourth steps of the email funnel (that is, the second and third workers). Let's put it together:

1. The business process places one or more requests into the first input queue.
2. The first worker, specific to each business process, splits the request into one request per recipient. These output requests, one per recipient, contain: To, From, Subject; the twig templates to use; the Twig parameters for rendering the templates.
3. The second worker renders the Twig templates for each input record and creates a record in the next queue containing the rendered email.
4. The third worker sends out the email, recording the gateway status of that send.

Split Email Request Into Recipients

The first worker is specific to posting Reserve Week scores. The process shown in Figure 1 will have placed one record in the first input queue, which is the table `email_requests`. See Figure 3 on the next page. The first worker takes that record as input and creates one `email_recipients` record for each team staff member (i.e., one for each recipient according to our business process).

Figure 3.



Based on the flowchart in Figure 3, we can list the events this worker should emit. Note that these events are the items of interest to you as the developer. At this point, we're making no attempt to design for stakeholder or customer service needs.

Where do we record these events? That's a great question! We decided that, initially, we were recording them in database tables. Ultimately, we intend to feed them into New Relic, Splunk, or some similar tool. Rather than trying to solve the entire monitoring question at the same time as designing the email queues, we're recording the events in database tables. That gets the "hooks" into the PHP code. Later, it will be easy to find and update the hooks to send events elsewhere. We will have already designed-in Observability.

Figure 4.

Figure 4 shows the page I use to observe both logged events and queue status. When I click on a queue name or log name, the page shows the latest records in that queue or event log.

Figure 5 (on the next page) shows the logged events emitted by the Post Reserve Week webpage. The event names and descriptions should match the table above.

Event	Description	Activity
PostReserveEmail-Split.WorkerBegan	Worker began processing	null
PostReserveE-mailSplit. WorkerCompleted	Worker completed processing	null
PostReserveEmail-Split.RecordFound	Worker found record to process	email_requests_ id
PostReserveE-mailSplit. ReservationOb-tained	Worker reserved record to process	email_requests_ id
PostReserveE-mailSplit. NoRecordsFound	Worker found no more work	null
PostReserveE-mailSplit. ReservationNotOb-tained	Worker failed to reserve found record	email_requests_ id
PostReserveE-mailSplit. ScoreEntryLog-Loaded	Worker loaded team-discipline data	email_ requests_id, score_entry_ log_id
PostReserveE-mailSplit. EmailRecipientCre-ated	Worker created email request for one recip-ient	email_ requests_id, score_entry_ log_id, email_address, email_recipi-ent_id
PostReserveE-mailSplit. EmailRequestCom-pleted	Worker completed processing reserved record	email_requests_ id
PostReserveE-mailSplit. EmailRequestFailed	Worker processing failed	email_requests_ id, message

Figure 5.

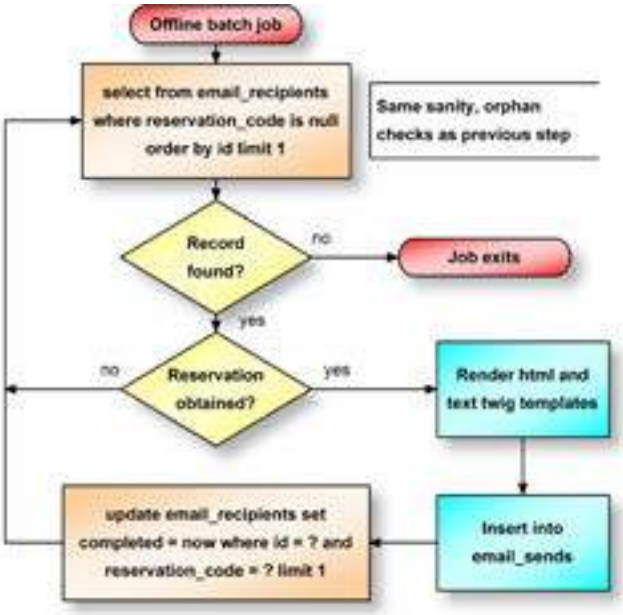
Render Twig Templates

The next step is common to all business processes emitting email notifications. It takes the request with a single recipient as input, renders the email body using the twig templates, and places the fully rendered email into its output queue (the input queue for the next step). See Figure 6.

This worker emits a similar set of events.

Event	Description	Activity
EmailRender.WorkerBegan	Worker began processing	null
EmailRender.WorkerCompleted	Worker completed processing	null
EmailRender.RecordFound	Worker found record to process	email_recipient_id
EmailRender.ReservationObtained	Worker reserved record to process	email_recipient_id
EmailRender.NoRecordsFound	Worker found no more work	null
EmailRender.ReservationNotObtained	Worker failed to reserve found record	email_recipient_id
EmailRender.RenderCompleted	Worker rendered html and plain-text email body	email_recipient_id, template_prefix
EmailRender.EmailSendCreated	Work created full email record to be sent	email_recipient_id, email_send_id
EmailRender.EmailRecipientCompleted	Worker completed processing reserved record	email_recipient_id
EmailRender.EmailRecipientFailed	Worker processing failed	email_recipient_id, message

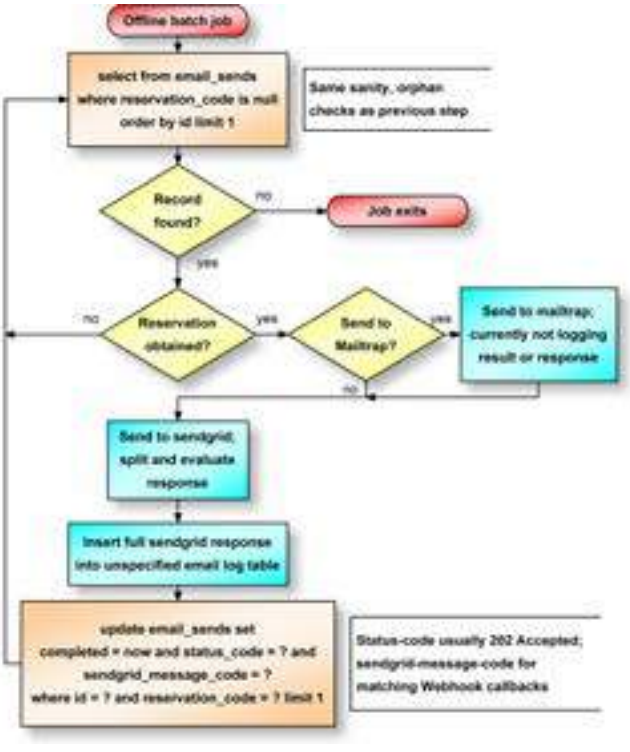
Figure 6.



Send Email

The final step is, again, common to all business processes emitting email notifications. This worker contains extra processing for working with developer environments, throttling requirements, and so on. If, in the future, we need to provide priority for certain types of email, this worker will be the place to make those adjustments. See Figure 7.

Figure 7.



This worker, again, emits a similar set of events.

Event	Description	Activity
EmailSend.WorkerBegan	Worker began processing	null
EmailSend.WorkerCompleted	Worker completed processing	null
EmailSend.RecordFound	Worker found record to process	email_send_id
EmailSend.ReservationObtained	Worker reserved record to process	email_send_id
EmailSend.NoRecordsFound	Worker found no more work	null
EmailSend.ReservationNotObtained	Worker failed to reserve found record	email_send_id
EmailSend.MailTrapSendCompleted	Worker sent email to MailTrap	email_send_id
EmailSend.MailTrapSendFailed	MailTrap send failed	email_send_id
EmailSend.SendGridSendCompleted	Worker sent email to SendGrid	email_send_id, status_code, sendgrid_message_code
EmailSend.SendGridSendFailed	SendGrid send failed	email_send_id, status_code (possibly null)
EmailSend.EmailSendCompleted	Worker completed processing reserved record	email_send_id
EmailSend.EmailSendFailed	Worker processing failed	email_send_id, message

Stakeholder Events

We still need to answer the following of questions:

- What events or results are of interest to stakeholders?
- What events or results are of assistance to customer support?

The answer is easy! The only item of interest is the email itself and information about sending it. The final queue, containing a copy of what was sent and to whom, is what we need. From that, we can investigate whether it was or was not sent and when. We can re-send the email if desired. The queue itself represents a record of what was sent for audit purposes.

That's why it was important not to combine the questions about what events were of interest. Solve each of those problems individually so that one answer does not get in the way of keeping the other answer simple.

Separation of Concerns

Everything here represents transitional architecture. We may need to change how we do email. We expect to change the destination of emitted events (to, e.g., New Relic). We might move to a different queueing system, such as RabbitMQ. However, it's best to only change one "variable" at a time. By separating things, we can change our queueing mechanism without affecting anything else, and so on.

Summary

We continued to use flowcharts as a means of describing business processes. To be sure, PHP code can be written to match the flowchart, but that need not be the case. The flowchart's purpose is to capture "tribal knowledge". We designed an email queue system as a series of small steps. We described the events that the system should emit. We separated various concerns, making adjusting one area easier without affecting others.

Related Reading

- *DDD Alley: Create Observability, Part 1: Local Timing* by Edward Barnard, June 2023. https://phpa.me/ddd-jun-23
- *DDD Alley: Create Observability, Part 2: Capture "Tribal Knowledge"* by Edward Barnard, July 2023. https://phpa.me/ddd-jul-23
- *DDD Alley: Create Observability, Part 3: Rewrite Business Process* by Edward Barnard, August 2023. https://www.phparch.com/article/2023-08-ddd/
- *DDD Alley: Create Observability, Part 4: Simple Queue System* by Edward Barnard, September 2023. https://www.phparch.com/article/2023-09-ddd/

Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others. @ewbarnard

Taking Laravel To The Orchestra; Building A Symfony Inspired Application.

Steve McDougall

Clean code. Something not often associated with a typical Laravel application is it. With that Active Record database approach and the magic static calls everywhere—those used to stricter frameworks, such as Symfony, often shudder when they open a Laravel project. What if I told you it didn't have to be like that?

What I love about Laravel is that I can go as in-depth as I like—pushing the strictness as far up or as low down as I want. I can get a proof of concept thrown together over a weekend and retrospectively increase my code strictness over time (presuming I have a good test suite, of course).

Now, when we talk about strict code, we think of the classic "Clean Code" book that many of us have likely read before. We think about that clunky old Java code where there are abstractions on abstractions. Adapters and ports. Onions and Hexagons. You get the idea. A lot of terminology and a lot of code in a lot of different places.

So, how can we relate this back to PHP—and, more specifically, Laravel? How can we take the lessons we learned from these books and apply the principles to a standard Laravel project? Surprisingly we do not have to look too far, as Laravel has friends in the right places. By that, I mean, of course, Symfony.

Symfony is a fantastic framework that you, as a developer, can do a lot with. However, it is not as forgiving as Laravel and requires you to know what you are actually doing a little more. Some might say that if you want a truly clean code, just use Symfony. I would be inclined to agree as the work required to get Laravel doing things the typical "clean" way—you are stripping out a lot of what makes Laravel, well, Laravel. However, with that being said, Laravel has such a rich ecosystem and community—that you don't really want to miss out on it. So, the question still remains: how can we make our Laravel code clean?

For me, it starts with data retrieval—a pivotal part in any application of any size. How can we effectively talk to our database in a Laravel application with as little "magic" as possible? Another good question is: how can we leverage the power of what Eloquent ORM offers while keeping our code clean? The answer is simpler than you think—use the repository pattern, but use it wisely.

To properly understand this approach, we need to start with what a typical Laravel application might look like, and go through the stages of how we might change this code to make it closer to what is classed as "clean". For the sake of keeping it all together, I will write the entire process in one code block at a time. (See Listing 1)

Listing 1.

```
1. Route::get('/articles', IndexController::class)
2.     ->name('articles:index');
3.
4. final class IndexController
5. {
6.     public function __invoke(Request $request)
7.     {
8.         return Article::all()
9.     }
10. }
```

So, minimal code is required to get the request through to the controller, make a query, and return a JSON Response. This is handy when we are prototyping and need the rapid application approach. However, as our platform scales—or we want to increase our testability we need to go beyond just getting it done. This is when we can start leaning on design principles and software architecture to improve the testability and reliability of our code. It is never enough just to be done because we all know code isn't really done until it is tested. How can we go about testing the code above properly? We can't because everything is too tightly bound to Laravel magic.

Now, if I were a start-up, and when I was a solo developer at a start-up, I would write code like this. It allowed me to move quickly, test ideas, and have a low time investment, simply so I could move onto the pile of other things I was being asked to do. But, once you have hit that MVP or proof of concept marker, you need to re-think your approach; otherwise, you will end up going through rebuild after rebuild instead of revolution after revolution of your product.

Now that it is out of the way, we can move forward. How can we improve upon this code to make it more testable and cleaner? The first thing we want to do is remove the "magic" that is harder to test. We then want to move towards a more declarative response. (See Listing 2 on the next page)

We have swapped out the helper class for creating and returning the JSON response directly. The response and JSON helpers will only do this in the background anyway. We also add the status code to be explicit. If we do not add this, we do

Listing 2.

```
1. Route::get('/articles', IndexController::class)
2.             ->name('articles:index');
3.
4. final class IndexController
5. {
6.     public function __invoke(
7.         Request $request
8.     ): JsonResponse {
9.         return new JsonResponse(
10.            data: Article::all(),
11.            status: Response::HTTP_OK,
12.        );
13.    }
14. }
```

not know that we will return the correct status code. However, we are still facing the problem of the Eloquent magic being stuck in there.

Eloquent is a fantastic ORM, and I use it all the time—even now. But, it does a lot of magic for you—especially if you are just calling a magic static `all` method. We can incrementally improve this, which will show the different options available to you. (See Listing 3)

Listing 3.

```
1. Route::get('/articles', IndexController::class)
2.         ->name('articles:index');
3.
4. final class IndexController
5. {
6.     public function __invoke(
7.         Request $request
8.     ): JsonResponse {
9.         return new JsonResponse(
10.            data: Article::query()->get(),
11.            status: Response::HTTP_OK,
12.        );
13.    }
14. }
```

Now, we are using the static `query` method, which is an actual method available on the Eloquent Model class—with no static call magic going on behind the scenes. Then, we call the `get` method to get all of the results, which is a method available on the Query Builder that gets returned from the `query` method. This makes your code more reliable, but it isn't any more testable than it was before. When building and scaling an application, we want to make sure that our code is both reliable and testable.

Where can we go from here? Design patterns are our friend here, and we can start by implementing the Repository Design Pattern, which will make our code more testable while still utilizing the Eloquent ORM. (See Listing 4)

Listing 4.

```
1. Route::get('/articles', IndexController::class)
2.     ->name('articles:index');
3.
4. interface ArticleRepository
5. {
6.     public function all(): Collection;
7. }
8.
9. final class EloquentArticleRepository
10.                implements ArticleRepository
11. {
12.    public function all(): Collection
13.    {
14.        return Article::query()->get();
15.    }
16. }
17.
18. final readonly class IndexController
19. {
20.    public function __construct(
21.        private ArticleRepository $articleRepository,
22.    ) {}
23.
24.    public function __invoke(
25.        Request $request
26.    ): JsonResponse {
27.        return new JsonResponse(
28.            data: $this->articleRepository->all(),
29.            status: Response::HTTP_OK,
30.        );
31.    }
32. }
```

Now, we are getting into the area where our code is a bit cleaner and a lot more testable. What I haven't added here is the container bindings that we would usually add in a service provider. Typically, we would bind the interface to a concrete implementation so that when we inject the interface, we get a resolved implementation injected. I mean, we know how a container works, right?

From here, there are a few different approaches depending on the software architecture approach you may want to take. Right now, we are using something reliable and testable. However, we can go further. If we are looking for clean code, we are a long way off. From a testing perspective, we can create a mock repository for our test environment right now. We can also test the implemented version to make sure that there are no issues with our actual repository.

Let's take this a step forward by implementing service classes. To me, adding a service class makes a lot of sense when you are building something relatively large, where you might want to have articles accessed across many areas of your application, as well as having different approaches to accessing the database layer. (See Listing 5 on the next page)

All we have really done here is add an additional layer between our user-land code and our business code. This is admittedly a relatively simplistic example. However, if we

Listing 5.

```php
Route::get('/articles', IndexController::class)
    ->name('articles:index');

interface ArticleRepository
{
    public function all(): Collection;
}

final class EloquentArticleRepository
            implements ArticleRepository
{
    public function all(): Collection
    {
        return Article::query()->get();
    }
}

interface LaravelService {}

final class ArticleService implements LaravelService
{
    public function __construct(
        private readonly ArticleRepository $repository,
    ) {}

    public function all(): Collection
    {
        return $this->repository->all();
    }
}

final readonly class IndexController
{
    public function __construct(
        private ArticleService $service,
    ) {}

    public function __invoke(
        Request $request
    ): JsonResponse {
        return new JsonResponse(
            data: $this->service->all(),
            status: Response::HTTP_OK,
        );
    }
}
```

were to look at a more complex example with user roles and potentially other business rules—the benefits of a service class in-front of your repositories become quite clear. The service class gives you a clear separation of concerns, both reusability and testability, and maintainability is increased. (See Listing 6)

What we see here is the requirement for three different repositories in our service. Our Financial service could need to access information on Accounts, Invoices, and Sales. Then, in our method, we perform several checks. Firstly, if the authenticated user can view invoices, then if the user's account has tokens available to spend in getting the information. This

Listing 6.

```php
final class FinanceService implements LaravelService
{
    public function __construct(
        private readonly AccountRepository $account,
        private readonly InvoiceRepository $invoice,
        private readonly SalesRepository $sales,
    ) {}

    public function latest(
        CarbonInterval $between
    ): Collection {
        if (! Gate::allows('view-invoices')) {
            throw new MissingPermissionsException(
       'You do not have permissions to view invoices.',
            );
        }

        if (! $this->account->hasTokensAvailable()) {
            throw new RanOutOfTokensException(
  'Your account has ran out of tokens, please top up.',
            );
        }

        return $this->invoice->between($between);
    }
}
```

is, of course, a very hypothetical service class—but I feel it highlights where this could be beneficial.

If we take a step back and think about our software architecture a little, what is important to us is being able to model our business processes and rules in code easily and adjust and amend these when required or even extend them. This is where patterns such as service and repository patterns really shine. Because each service is an entry point into your business logic, which is abstracted away from your data access layer, having that separation in a business-oriented application is vital as complexity and team size grows.

This is also highly testable, since magic code is hard to test. Instead, you end up relying on magic test helpers that are provided—which massively limits the control you have on your test cases. This isn't always a problem, but it does limit the confidence you can have in your tests. In the past, I have had issues where magic wasn't properly understood, which led to fragile tests that sometimes randomly failed in CI. Let's compare two test cases, one magic and one controlled. (See Listing 7 on the next page)

The code above starts by putting Laravel's HTTP facade into "fake" mode, which will fake the request. We then send a post request to one of our endpoints and assert that we sent one HTTP request externally. This is minimal magic, but we have minimal control here. You can take a few different approaches to improve this magic, but the main one is to wrap your HTTP communication in a library that is not hidden

Listing 7.

```php
/** @test */
public function it_can_send_a_http_request(): void
{
  Http::fake([
    '*' => Http::response(['status' => 'ok'], 200),
  ]);

  $this->postJson('some-internal-url', [
    'foo' => 'bar',
  ])->assertStatus(201);

  Http::assertCount(1);
}
```

behind layers—instead, allowing you to create test doubles and mock where required. Your future self will thank you!

One approach I like to do is: (See Listing 8)

Listing 8.

```php
final class GitHub
{
  public function __construct(
    public function ClientInterface $client,
  ) {}

  public static function new(
    ClientInterface $client
  ): GitHub {
    return new GitHub(
      client: $client,
    );
  }

  public static function discover(): GitHub
  {
    return new GitHub(
      client: Psr18ClientDiscovery::find(),
    );
  }
}
```

Allowing me to either:

- Bind my implementation in my DI Container
- Use a factory approach to create a new client and use PSR-18 auto-discovery to find my HTTP client.
- Or manually passing a client instance in - allowing me to use the HTTP PHP Mock client for testing.

These are things that we will all get to a point of having to think about; it may not be now—but you will need to at some point in your career. The reason is that you will want to tighten control of the magic going on in your application as you scale in some way. As you squeeze as much control and performance out of your application—you will also want to start squeezing the performance and control out of your tests, as you will want to prove reliability.

Of course, you could take this further, but there is a fine line between well-written and over-engineered, and finding that line can sometimes be harder than building the application in general. Let's discuss the options for taking this further.

The initial step that comes to mind for me is to use a pattern such as CQRS—Command and Query Responsibility Segregation. This leans on a global command and query bus to map commands (write action) and queries (read actions) through the bus to handlers, which will handle what is being asked to do.

Another option is to use the Action pattern, which was made popular a few years ago in the Laravel community. What this does is split what you would typically have in a Service or Repository class into single classes with one job. This is a great pattern to look into. However, I find in larger applications, this approach creates cognitive overload when trying to find a specific action to debug.

The main question to ask yourself is, what is going to work for you? There are so many different design patterns and architectural approaches you could take. But if it doesn't feel natural to you and your team, it isn't worth the time investment to refactor to this approach.

This begs the question, "Should you worry about clean code in your Laravel application?" Should you replicate the Symfony approach in Laravel? In my humble opinion, the answer to both questions is no. If you want to write Laravel like a Symfony application, just use Symfony. The Developer Experience is why you should choose Laravel, and trying to adopt these other patterns is going against what the framework is trying to give you. Yes, you could adjust your architecture to fit your business needs, but it doesn't mean you should fight the framework at every turn.

Laravel, when you accept its benefits and flaws, is a pleasure to use. When you are only trying to get it to do something that isn't the Laravel way, you start to feel frustrated. The Laravel team has poured their souls into making the developer experience of Laravel the best it can be, and as a community, we should embrace that instead of fighting it.

So, in terms of taking Laravel to the orchestra. You shouldn't. Symfony might fit well in this scenario, but Laravel is happiest going to a rave; in fact, it works best when it's at a rave. You can take inspiration from wherever you like, but please don't try to fight against Laravel's nature. Celebrate Laravel for what it is.

Steve McDougall is a conference speaker, technical writer, and YouTube livestreamer. During the day he works on building API tools for Treblle, and in the evenings spends most of his time writing content, or contributing to the PHP open source community. Whatever you do, don't ask him his opinion on twitter/X `@JustSteveKing`
@JustSteveKing

Is Your Plan Extensive Enough To Help Someone Else?

Christopher Miller

Do you remember your first introduction to computing? How about what happened next? My life would be forever changed by the experience and for the better. This article will walk you through what happened next and the overall development of my code.

Who are You? Why Can You Talk About This?

I Remember getting my first introduction to computing in 1985, around age three. My dad came home from work one day with a box – it contained a soldering iron, a book, and some bits. We sat at a table together and built a computer. It was a ZX Spectrum 48k. I vividly remember the feelings of excitement and joy as I watched this piece of technology come together, followed by fascination as it was plugged into the television for the first time and came to life – the sheer ecstasy of watching some blocky graphics move on a screen in eight different colors, the sense of anticipation as the tape loaded a game for me and dad to play.

That day changed my life for good. I started learning the concept of typing things to make other things happen on screen and found a love for making computers do what I wanted. I learned this was called programming, and I liked it. I wrote games that were published in Spectrum magazines and knew I was destined to one day work with computers and create software.

Today, I am a Lead Full Stack Developer Working with Laravel, CakePHP, React, Vue, React Native, and many other languages. I am often found preparing the next talk, learning about the next new technology, or mentoring newcomers to the technology scene.

I take great pride in producing highly efficient code, logical debugging, seeing the bigger picture when looking at a feature request from a client, and suggesting changes that would improve customer process efficiency. Customers often comment on my ability to turn tech jargon into easy-to-understand concepts, helping them to understand the problem and the solution, often helping them to see the future possibilities of their applications.

What's All This About?

Over the next sections, we will investigate a few questions that will give you a great overview of how I think about all the code I write, with code examples. You'll get a PHP Object Oriented example, which should give you enough scope to cover any language.

I Got It Wrong

Let me tell you a story. One day, I was hired by a company to create inventory software. I was young, cocky, and absolutely sure I was infallible in building this piece of software. I didn't plan anything out—I just got stuck into writing. I didn't even spend more than ten minutes discussing the concept with the customer, believing they basically wanted a spreadsheet. I quoted two days to do the job and thought I was 100% right. I Gave them the price; they were over the moon! I was less than a tenth of the cost of my closest competitor. As my first freelance job, this was awesome! Not only can I come in cheaper than my competitors, but I'm faster, can produce it easier, and don't even need to waste time planning.

Off I went to build this software. I fired up a quick SQL database—it had the following structure:

Field Name	Field Type	Constraints
id	integer	Primary Key, Auto Increment
name	string	100 characters
price	float	8,2
qty	integer	

I quickly created my table, calling it products. I then got to writing my CRUD routes:

- Create Product
- Update Product
- Get Product
- Delete Product
- Sell Product
- Buy Product

It took me about a day, and I only spent a couple of hours testing it the next day, delivering it early! I was so happy to deliver early on this. At this point, I went to see the customer. Then came the hardest conversation of my life so far—`THE CUSTOMER WAS NOT HAPPY!`

I didn't understand what was wrong! He could add products, add a price, and track the quantity. That's an inventory app, right? Well, it turned out that the customer wanted WAAAY more than that. He wanted something that could be used to track suppliers, customers, sales, and purchases, and give an audit log. He wanted to be able to track everything to do with his inventory, not just a figure.

I got burned badly here, and with some careful negotiations explaining I'd totally misunderstood the requirements, some quick thinking on my part, and a lot of sleepless nights, as well as a good, friendly customer, we agreed that I would continue and add to the project the bits that were missing. The customer agreed to meet me at 75% of the cost I said it would be, mainly because he felt sorry for me, I think!

I Got It Wrong... Again!

Well, after that, my next customer came to me. They wanted a much different thing—a website that could detail their company and have customers contact them (this was back when the web was little more than pretty text). Having learned from the last events, we went through a different process. I met with the customer over three days (free of charge) to plan out the design they wanted and provided this to them. Then, over another five days, I planned in intricate detail the CSS code, the Javascript, the HTML—every mortal thing. I spent a full day producing a detailed quote. I then went back for another meeting with the customer—at this point, we were at nine days free of charge, a twenty-page design document, a thirty-page plan document, and a ten-page quote showing every tiny detail of what I'd be doing. The final meeting was to go through these documents together. After half a month of planning, this time, I was 100% sure I was okay. Nothing could get in the way of this, right?

WRONG—the customer thanked me for the design, plan document, and the quote but said no—they didn't want to go with me.

I was horrified! I'd got two customers in a row wrong with my planning! By this point, I almost didn't want to code anymore! Clearly, customers are idiots because no plan is wrong, and a big plan can't be wrong.

Then I Got It Right

The third customer came to me. This time, they wanted an application allowing them to take in some images in bulk and then output each image in three formats—grayscale, transparent background, and blue background (they gave me the exact hex code).

Now, I had already tried no plan, and that wasn't right. I had tried an extensive plan, and that wasn't right. This time around, I spent a couple of hours with the client, with a few images they had, and manually did the tasks they wanted to check that my understanding was correct. I then went home, wrote a one-page plan that detailed what would be needed, a one-page quote detailing my price and timing, and then delivered that back in a day. The customer was happy! The customer said yes, and everything went *almost* perfectly—a few minor bugs that were down to images not being 100% perfect for the job, but I delivered on budget, on time, and on plan.

Then I Worked On a Bigger Project

Well, as you can see, I quickly learned about plans for a customer being one thing, but let me tell you another story.

Jump forward a few years, and I was asked to be part of a team to build a much larger application. This application was my first adventure in working with a team, and I set out planning as I did for customers, with quick, short descriptions with a time estimate against it. That was okay, but I noticed that the team seemed to be asking many questions. Hey *Chris, you said you would do this—but how are you doing it? I need to make the front end of that* kind of question. This was intriguing, but I thought nothing of it. At that time, my father had a heart attack, which stopped me from working on this application for about two weeks, and I told the team my plans should be okay for them to pick up the slack for a while.

Well, when I got back, there was an email from the manager asking me to call him as soon as possible. I did exactly that. He did not sound happy; in fact, he sounded angry! He asked me to come see him in person. I thought everything was going great, so I assumed this was anger at someone else, and he just wanted me to pick up some more work to get it done. That day, I met up with him, and we sat down. I remember the start of him talking to me word-for-word!

Chris, I need to be 100% honest with you. The team was really unhappy while you were away. They felt like they didn't know what was happening with the code or the application on the bits you were doing. We had to take two days to work out what you were up to and to put together the next steps you were heading towards. I get that you had some plans, but no one could marry the plans to the code. They felt like different things. I'm sorry to have to do this; we can't continue like this—I need you to be much clearer in your plans about the code.

That was devastating—I'd been working like this for years! Never had anyone told me my plans weren't good enough; in fact, just the reverse! Most customers commented on how helpful my plans were by now. I was shocked! Our conversation continued for a few hours after that as I tried to understand how this was so different!

Well What Makes for Good Customer Requirements?

So, what does a good plan look like? Why don't we take a thought experiment here: let's look at a requirement you might get from a customer and make a plan for the customer vs. the team working on the code together.

First, let's look at the user's requirements that come in for a mortgage amortization application:

```
As a user, I want to be able to create a secure account with a unique username and password,so that I can access
     and manage my mortgage information securely.
As a user, I want to be able to log in to my account using my username and password, so that I can access my
     mortgage information.
As a user, I want to have the option to use two-factor authentication for added security, such as receiving a
     one-time code via email or text message.
As a system administrator, I want to have the ability to lock or disable accounts that show signs of unauthorised
     access or potential security threats, to protect the security of user information.
As a user, I want to be able to reset my password if I forget it, using a secure process such as answering security
     questions or receiving a password reset link via email.
As a user, I want to be able to input my mortgage loan amount, interest rate, and loan term in years, so that I can
     see my monthly payment and amortisation schedule.
As a user, I want to be able to select from different payment frequencies (e.g. monthly, biweekly, weekly), so that
     I can choose the option that works best for me.
As a user, I want to be able to make additional principal payments on my loan, and see how this affects my
     amortisation schedule and total interest paid.
As a user, I want to be able to view a graph or chart that shows my loan balance and interest paid over time, so
     that I can better understand how my mortgage is amortising.
As a user, I want to be able to save and access my amortization information, so that I can track my progress and
     update my information as needed.
As a user, I want to be notified via email or in-app notification when important events or updates occur, such as
     when my monthly payment is due or when I have made significant progress towards paying off my mortgage.
As a user, I want to be able to customize my notification settings, such as choosing which types of notifications
     I want to receive and how I want to receive them (e.g. email, in-app, push notification).
As a user, I want to be able to view the history of my notifications, so that I can review important information
     and track the progress of my mortgage.
As a system administrator, I want to be able to send targeted notifications to specific groups of users or
     individual users, such as when there is a change to interest rates or other relevant information.
As a user, I want to have the option to opt out of receiving notifications altogether,
     I do not want to receive them.
As a user, I want to be able to adjust the amount of my monthly payment, and see how this affects my amortisation
     schedule and total interest paid.
As a user, I want to be able to enter a target loan payoff date, and see how making additional payments or
     increasing my monthly payment would affect my ability to reach this goal.
As a user, I want to be able to see the impact of making a one-time lump sum payment on my loan, such as how much
     interest I would save and how it would affect my amortisation schedule.
As a user, I want to be able to compare the costs and benefits of refinancing my loan, including the potential
     savings on interest and the impact on my loan term and monthly payment.
As a user, I want to be able to see a summary of the potential savings I could achieve by making changes to my
     loan payments, such as the total interest saved and the number of payments reduced.
As an admin, I want to be able to view a dashboard that provides an overview of key metrics and information
     related to the program, such as the number of active users, the total loan amount outstanding, and the
     average interest rate.
As an admin, I want to be able to generate and download reports that provide detailed information about the
     program, such as a list of all users and their loan details, or a breakdown of loan balances by loan term
     and interest rate.
As an admin, I want to be able to filter and customize the data included in reports, so that I can easily access
     the information I need.
As an admin, I want to be able to view graphs and charts that visualise key trends and patterns in the program
     data, such as the distribution of loan amounts or the growth of the user base over time.
As an admin, I want to be able to set up automated reports that are generated and delivered on a regular schedule,
     such as daily, weekly, or monthly. This will allow me to easily track key metrics and monitor the program's
     performance without having to manually generate reports.
```

And What Makes a Good Customer Plan?

so here, we have what feels like 25 requirements from the customer, however Let's take a look at what we show to the customer:

Phase 1: Initial development

In this phase, we will focus on building the core functionality of the mortgage amortization program, including the ability for users to input their loan information and view their monthly payment and amortization schedule.

We will also implement basic authentication and account management features, allowing users to create and log in to their accounts securely.

Additionally, we will develop the necessary infrastructure and user interface for the program, including a responsive web design that works across different devices and browsers.

Phase 2: Advanced features and customization

In this phase, we will expand the functionality of the program by implementing the additional requirements provided, such as the ability for users to make additional payments and view their loan balance and interest paid over time.

We will also add more advanced features for users to customise their experience, such as the ability to select different payment frequencies and adjust their notification settings.

Additionally, we will improve the program's user interface and user experience based on initial user feedback and testing.

Phase 3: Reporting and analysis

In this phase, we will develop the admin reporting panel and related features, allowing administrators to view key metrics and generate reports about the program.

We will also implement the ability for users to compare different payment scenarios and see the potential savings and costs involved.

Additionally, we will continue to gather feedback and make refinements to the program based on user needs and requests.

Overall, the development process will take approximately 3-6 months to complete, depending on the scope and complexity of the requirements. We will work closely with the customer to gather feedback and ensure that the program meets their needs and expectations.

But Developers Want Something Different

So now we have our customer's plan; what might a developer plan look like?

Well, we'd start with a database schema: (See Listing 1 on the next page)

With this, we've got a great starting place. Now, let's get our API routes set up for our developer plan: (See Listing 2 on the next page)

The next step would be to look at the third-party APIs we might need to use:

```
Auth0: Auth0 is a popular platform for authentication
  and authorization, providing secure and scalable
  solutions for managing user accounts, logins, and
  permissions. It supports a wide range of
  authentication methods, including two-factor
  authentication, and offers easy integration with
  various languages and frameworks.
Twilio: Twilio is a cloud-based communication platform,
  providing APIs and tools for building messaging,
  voice, and video applications. It offers support for
  two-factor authentication via SMS and voice, and
  allows for easy integration with various languages
  and frameworks.
DocRaptor: DocRaptor is a cloud-based HTML to PDF
  conversion platform, providing APIs and tools for
  generating PDFs from HTML, CSS, and JavaScript. It
  offers support for various features, such as custom
  headers and footers, and allows for easy integration
  with various languages and frameworks.
```

As you can see, I will often go into a lot of depth with my planning, but all this is designed to make it helpful for other people—not just me!

What About Handovers and Handoffs?

Before we finish, though, we need to look at a couple more things—handovers and handoffs.

Handovers are generally done when a developer takes a break from a project but will return to the task—they should give enough information for someone to pick up on their work and continue from where they are. There are a couple of types I like—based on how long I will be off the project.

If it's for a day or so, I'll use the following style:

- Notify the team and the manager about the absence and provide the reasons and the expected duration to ensure that the team is aware of the situation and can plan accordingly.
- Identify the tasks and the deliverables that the developer is currently working on and provide the status and the progress to help the team understand what needs to be done and how much has been done.
- Identify the tasks and the deliverables that the developer is supposed to work on in the next few days and provide the details and the requirements to help the team understand what needs to be done and how it needs to be done.
- Identify the dependencies and the risks that the developer is aware of and provide the details and the potential solutions to help the team understand what needs to be done and what challenges might arise.
- Provide the contact information and availability to ensure that the team can reach the developer if necessary and that the developer can respond to any urgent or important requests.

Listing 1.

```
users:
    id: UUID (primary key)
    username: VARCHAR (unique)
    password: VARCHAR
    email: VARCHAR
    phone: VARCHAR
    two_factor_enabled: BOOLEAN
    role: VARCHAR
loans:
    id: UUID (primary key)
    user_id: FOREIGN_UUID (foreign key)
    loan_amount: DECIMAL
    interest_rate: DECIMAL
    term: INTEGER
    start_date: DATE
payments:
    id: UUID (primary key)
    loan_id: FOREIGN_UUID (foreign key)
    payment_date: DATE
    principal: DECIMAL
    interest: DECIMAL
    balance: DECIMAL
notifications:
    id: UUID (primary key)
    user_id: FOREIGN_UUID (foreign key)
    notification_type: VARCHAR
    notification_date: DATE
    message: VARCHAR
reports:
    id: UUID (primary key)
    report_type: VARCHAR
    report_date: DATE
    data: JSON
audit_log:
    id: UUID (primary key)
    user_id: FOREIGN_UUID (foreign key)
    action: VARCHAR
    timestamp: DATETIME
    data: JSON
permissions:
    id: UUID (primary key)
    role: VARCHAR
    permission: VARCHAR
```

- Provide any necessary documents, files, or resources the team might need to access or use during the developer's absence to ensure that the team has everything it needs to continue working on the tasks.
- Provide any necessary instructions or guidelines the team might need to follow during the developer's absence to ensure that the team knows what to do and how to do it.

If it's for a week or more, I'll use the following style:

- Identify the tasks and the deliverables that the developer is currently working on and provide the status and the progress to help the team understand what needs to be done and how much has been done.

Listing 2.

```
POST /login: Log in to an existing account.
POST /logout: Log out of the current account.
POST /register: Register a new account.
POST /forgot_password: Initiate the process of
    resetting a forgotten password.
POST /reset_password: Complete the process of
    resetting a forgotten password.
POST /enable_two_factor: Enable two-factor
    authentication for the current account.
POST /disable_two_factor: Disable two-factor
    authentication for the current account.
POST /verify_two_factor: Verify the two-factor
    authentication code for the current account.
GET /users: Retrieve a list of users.
POST /users: Create a new user.
GET /users/id: Retrieve the details of a specific user.
POST /users/id: Update the details of a specific user.
DELETE /users/id: Delete a specific user.
POST /users/id/loans: Create a new loan for a user.
GET /users/id/loans: Retrieve loans for a user.
GET /users/id/loans/loan_id: Retrieve the details of a
    specific loan for a user.
POST /users/id/loans/loan_id: Update the details of a
    specific loan for a user.
POST /users/id/loans/loan_id/payments: Create a new
    payment for a specific loan for a user.
GET /users/id/loans/loan_id/payments: Retrieve a list
    of payments for a specific loan for a user.
GET /users/id/loans/loan_id/payments/payment_id:
    Retrieve the details of a specific payment for
    a loan for a user.
POST /users/id/loans/loan_id/payments/payment_id:
    Update the details of a specific payment for a
    loan for a user.
DELETE /users/id/loans/loan_id/payments/payment_id:
    Delete a specific payment for a loan for a user.
POST /users/id/notifications: Create a new
    notification for a user.
GET /users/id/notifications: Retrieve a list of
    notifications for a user.
GET /users/id/notifications/notification_id: Retrieve
    the details of a specific notification for a user.
DELETE /users/id/notifications/notification_id: Delete
    a specific notification for a user.
POST /reports: Create a new report.
GET /reports: Retrieve a list
```

- Identify the tasks and the deliverables that the developer is supposed to work on in the next few weeks and provide the details and the requirements to help the team understand what needs to be done and how it needs to be done.
- Identify the dependencies and the risks that the developer is aware of and provide the details and the potential solutions to help the team understand what needs to be done and what challenges might arise.

- Provide the contact information and availability to ensure that the team can reach the developer if necessary and that the developer can respond to any urgent or important requests.
- Provide any necessary documents, files, or resources the team might need to access or use during the developer's absence to ensure that the team has everything it needs to continue working on the tasks.
- Provide any necessary instructions or guidelines the team might need to follow during the developer's absence to ensure that the team knows what to do and how to do it.
- Arrange for a temporary replacement or a backup developer, if necessary, to ensure that the team has the necessary skills and expertise to continue working on the tasks and to handle any challenges or issues that might arise.
- Review the project plan, the project schedule, and the project budget, and update them if necessary to reflect the changes in the team composition and the project progress.

The goal of the handover process is to ensure that the team can continue working on the tasks without any interruptions or delays and that the developer can return to the tasks without any difficulties or challenges.

A handoff, on the other hand, is if the developer is not expected to return to the project. That would look like this:

- Identify the tasks and the deliverables that the developer was working on and provide the status and the progress to help the team understand what has been done and how much is left to do.
- Identify the tasks and deliverables the developer was supposed to work on and provide the details and requirements to help the team understand what needs to be done and how it needs to be done.
- Identify the dependencies and the risks that the developer was aware of and provide the details and the potential solutions to help the team understand what needs to be done and what challenges might arise.
- Provide any necessary documents, files, or resources that the team might need to access or use, to ensure that the team has everything it needs to continue working on the tasks. This could include the source code, the design documents, the user manuals, the test cases, the user stories, and any other relevant materials.
- Provide any necessary instructions or guidelines the team might need to follow, to ensure that the team knows what to do and how to do it. This could include the coding standards, the testing procedures, the debugging techniques, the deployment processes, and any other relevant guidelines.
- Arrange for a permanent replacement or a new developer, if necessary, to ensure that the team has the necessary skills and expertise to continue working on the tasks and to handle any challenges or issues that might arise. This could involve recruiting a new developer, training a new developer, or assigning tasks to another developer with the necessary skills and experience.
- Review the project plan, the project schedule, and the project budget, and update them if necessary to reflect the changes in the team composition and the project progress. This could involve revising the project scope, the project milestones, the project deliverables, and the project costs, to ensure that the project can be completed successfully within the constraints of the project.
- Ensure that the developer's privileged accesses are revoked where relevant and that the replacement developer has new credentials
- Detail availability for support for the new developer, and expected timescales for this.

Next month, we're going to delve into our next question, "Is Your Code Abstracted Enough To Minimize Load?" and that's where we get past the bit that most people hate, planning, and into the bit most people love: coding!

In 1983, Christopher was introduced to computers by his dad, at the tender age of 3. now, over 40 years later, he has been working in the industry for over 20 years making an impact across multiple sectors of the industry. Starting with launching the first web development company in Staffordshire, Christopher dealt with the web - when the web was little more than just pretty text. He established the websites for many different businesses in their first inception, before moving onto web applications a little while later. Illness prevented Christopher from working in the industry full time for some considerable time - but recovery meant he could tackle once again the joys of code - but he soon found that his skills had become out of date, so thanks to the School of Code in the UK he was able to return to the workplace with revitalised skills ready to tackle the next wave. Specialising since the School of Code in Readable Code, He has worked with a large number of languages, specialising in supporting businesses to grow standards for their code base, and now he is ready to share his processes with the world.
@ccmiller2018

@phptek
@tek@phparch.social
https://tek.phparch.com

PHP[TEK] 2024

CONNECT WITH THE COMMUNITY, LEVEL UP YOUR SKILLS, LEARN THE LATEST TRENDS, AND MAYBE EVEN FIND YOUR NEXT JOB. RESERVE YOUR SPOT TODAY.

PHP[TEK] 2024
16TH ANNUAL PHP[TEK] CONFERENCE FOCUSED ON PHP AND WEB DEVELOPMENT.

CFP IS CURRENTLY OPEN. IF YOU WOULD LIKE TO HELP SHAPE THE FUTURE OF PHP DEVELOPMENT AND WEB DEVELOPMENT, SUBMIT YOUR TALKS TODAY.
HTTPS://TEK.PHPARCH.COM/SPEAKERS

The Heart of the Code

Beth Tucker Long

We spend a lot of time talking about the best architecture, the most efficient design patterns, the proper way to test, and the safest security measures. These are all essential elements, but when you get down to it, what is the true heart of the project?

A simple search will pull up hundreds, if not thousands, of articles extolling the virtues of "The Best" way to build an application. Don't reinvent the wheel, use a framework. Frameworks are bloated, only use the libraries you need. Take your time and write your tests first. Don't waste time on being perfect, fail forward. The most important part of your app is the security. No, it's the user interface. No, it's the power of the back-end processing. No, it's the speed of the data storage and retrieval. No, it's…well, it's complicated.

At least, it's complicated if you are just looking at the articles and blog posts about theories, best practices, and programmer opinions. What if you take a step back? No, a really big step back. The back-end code, the database, the user interface, they are all just pieces working together. They each have an important place in the app, but they are all just components put together to facilitate sharing. Sharing data. Sharing money. Sharing ideas. Sharing time. Everything we build is helping people share something.

With that as our core focus, it shifts the focus on all of the other components. The user interface isn't just something you have to do to give non-programmers a way to access your system. It is a gateway for people to share data with others and with you. Security isn't just a way to keep hackers out. It is a double-check on how much we are willing to share and who we will share it with. The API is no longer just an easy way for your code to move data around. It is now a gateway for your application to share data with other entities and services.

Anyone can write some code. It takes a true programmer to see the bigger picture, looking outside the lines of code at what the application is facilitating between the humans interacting with it. So make sure you are taking the time to look beyond the changed lines in the latest PR and recognize what that code is making happen. Don't forget that there is a human side of everything we do. It's the piece that moves your app from functional to fantastic, so please, don't forget to share.

Beth Tucker Long is a developer and owner at Treeline Design[1], a web development company, and runs Exploricon[2], a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development[3] and Full Stack Madison[4] user groups. You can find her on her blog (http://www.alittleofboth.com) or on Twitter @e3BethT

1 *Treeline Design: http://www.treelinedesign.com*

2 *Exploricon: http://www.exploricon.com*

3 *Madison Web Design & Development: http://madwebdev.com*

4 *Full Stack Madison: http://www.fullstackmadison.com*