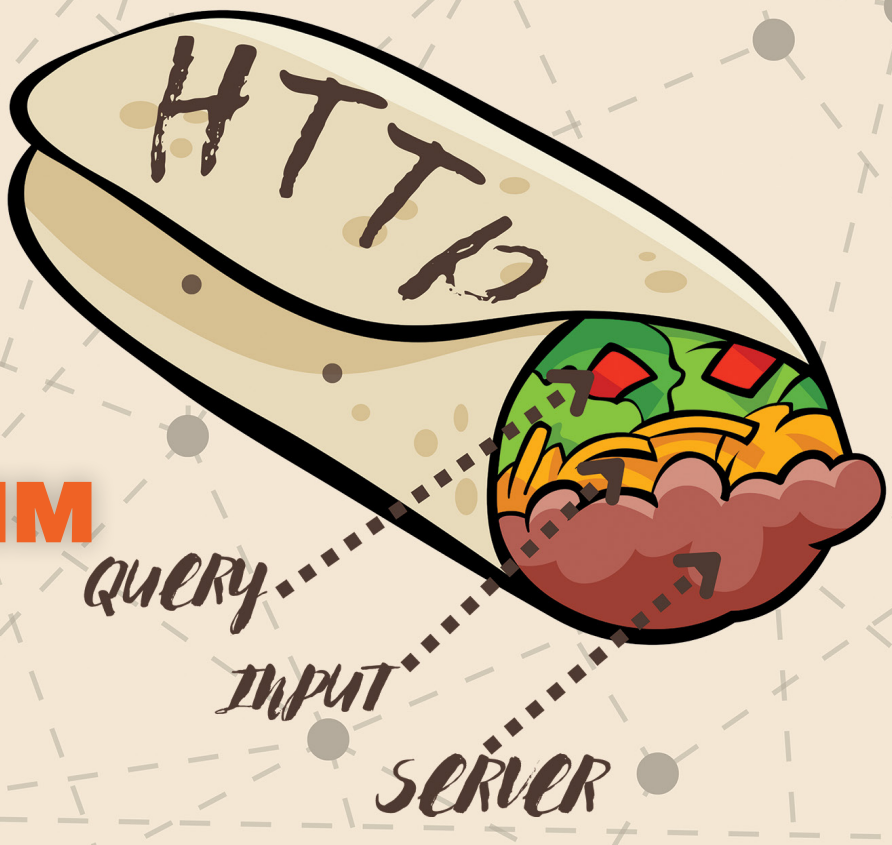
www.phparch.com

May 2023
Volume 22 - Issue 5

php[architect]

The Magazine For PHP Professionals

HTTP Burritos



HTTP Tortilla

Life After GIMM

ALSO INSIDE

infobip

Infobip CPaaS speaks your language

> Integrate feature-rich CPaaS solutions with your tech stack

Integrate any communication channel or module by using a flexible and programmable API stack that supports PHP. Tap into the open-source SDKs designed to get you up and running fast—with just a few lines of code.

Sign-up and get started with Infobip APIs.

[Try for free]

ADR vs MVC

Matt Lantz

In ADR or Action-Domain-Response, we maintain a three-piece pattern that lets us split our responsibilities. The classic MVC structure or Model-View-Controller is prevalent across all languages in the web development industry. Let's take a closer look at both.

> *A refreshing and exciting feeling wafts over each developer when they first write that line in what I suspect is iTerm2.*

```
$ laravel new kickass-site
```

or, more classically

```
$ composer create-project laravel/laravel kickass-site
```

More often than not, the next step is to contemplate which front-end framework to use and begin coding. Most developers will use customized artisan commands, generate their tests and resources, and map complex business entities to elegant models with transparent object layers. Most Laravel developers will build their applications in a classic MVC structure. However, some will explore working with single-responsibility controllers; some will integrate Actions throughout the application. Some will explore event-driven systems, and some will dive deep into domain-driven design.

The classic MVC structure or Model-View-Controller is prevalent across all languages in the web development industry. More often than not, Laravel developers will begin by creating a route in the `web.php` file, pointing it to a Controller, and start tinkering with their view or their Model. Beyond that, in many cases, developers will implement Middleware elements as well as Jobs, Events, and Listeners. What often happens in this classic MVC structure with sprinkles of enhancements is that business logic begins to creep through the various layers making it harder to maintain consistency or enable teams to work on code without impacting multiple layers. This dilemma is why enterprise teams and scaling organizations often begin implementing SPA solutions and breaking code into further and further micro-services, all the while hoping that the breakdown of the monolith is what will resolve the cross-layer contamination of the business logic.

Views often contain permission checks, and Models become somewhat bloated with complex queries. Controllers 10 years ago were bloated with code logic, but these days it's now sprinkled across Services or Models, and in the last few years, we've seen an active rise in the use of Actions. To avoid repetition, developers can use these self-contained code snippets in Commands, Controllers, and Services alike. The value gain is apparent; it enables developers to centralize where the logic is contained. This same pattern has become popular across front-end systems in how Vue and Livewire enable developers to have Components. In these cases, you have a component that can contain all logic of that entity in one file or a small set of files. However, these Components can quickly become bloated and handle all an entity's interactions, thereby becoming spaghetti-like. Each element aims to help developers move logic out of Blade templates and Controllers into reusable spaces.

Laravel itself has enabled developers to start with a very barren starter structure, and they can set things accordingly within their structure of choice. This benefit is that it allows developers to choose their own architecture. However, as mentioned above, most developers will often build things as an MVC structure and then sprinkle in other features as needed without documenting or designing the architecture as a whole. It becomes an MVC with random add-on patterns.

Paul M. Jones, a well-recognized member of the PHP community, proposed an alternative architecture in 2014 titled ADR or Action-Domain-Response. Its overall structure helps developers remove the repetition of business logic in their applications by drawing much deeper lines in the sand. Implementing the ADR pattern in Laravel would likely increase the total development time while reducing mental

Listing 1.

```
resources/
    templates/
        blog/
            index.php
            create.php
            read.php
            update.php
            delete.php
            _comments.php
src/
    Domain/
        Blog/
            BlogModel.php
                        BlogService.php
    Ui/
        Web/
            Blog/
                Index/
                    BlogIndexAction.php
                    BlogIndexResponder.php
                Create/
                    BlogCreateAction.php
                    BlogCreateResponder.php
                Read/
                    BlogReadAction.php
                    BlogReadResponder.php
                Update/
                    BlogUpdateAction.php
                    BlogUpdateResponder.php
                Delete/
                    BlogDeleteAction.php
                    BlogDeleteResponder.php
```

Listing 2.

```
class BlogCreateAction
{
    public function __construct(
        Request $request,
        BlogCreateResponder $responder,
        BlogService $domain
    ) {
        // ...
    }

    public function __invoke()
    {
        if ($this->request->isPost()) {
            $data = $this->request->getPost('blog');
            $blog = $this->domain->create($data);
        } else {
            $blog = $this->domain->newInstance();
        }

        return $this->responder->response($blog);
    }
}
```

Listing 3.

```
class BlogCreateResponder
{
    public function __construct(
        Response $response,
        TemplateView $view
    ) {
        // ...
    }

    public function response(BlogModel $blog)
    {
        // is there an ID on the blog instance?
        if ($blog->id) {
            // yes, which means it was saved already.
            // redirect to editing.
            $this->response->setHeader(
                'Location',
                '/blog/edit/{$blog->id}'
            );
        } else {
            // no, which means it has not been
            // saved yet. show the creation form with
            // the current data.
            $html = $this->view->render(
                'create.php',
                ['blog' => $blog]
            );
            $this->response->setContent($html);
        }

        return $this->response;
    }
}
```

strain and code complexity, thus enabling less complex maintenance.

Paul has some examples on GitHub of how the ADR pattern can be implemented and provides an example of refactoring an MVC to an ADR approach. Below are his examples: (See Listing 1 on the next page) https://phpa.me/refactoring[1].

In ADR, we maintain a three-piece pattern that lets us split our responsibilities. Actions are simple single tasks. The actions would follow the pattern of `Ui/Web/Blog/Create/CreateBlogAction.php`. Furthermore, you would have the Domain/Blog directory containing the Blog model and Service type classes interacting with the Model. Within the `Ui/Web/Blog`, you would have classes that build the entity's listing or provide a single view of a Form for editing the entity with a Responder. In some cases, we see how Responders can work much like Components since they can contain logic and use templates where needed. We can quickly see how we can also handle something like `Api/Blog/Create/CreateBlogResponder.php` or `Ui/Web/Blog/Create/CreateBlogResponder.php`; each of these can be defined in our action based on the request and

1 https://phpa.me/refactoring

its expected platform, which falls into better alignment with Bob Martin's Clean Architecture. (See Listing 2)

Within this example, we can see clearly the various cases where business logic can be injected into the domain and detached from the Responder and Action. (See Listing 3)

Similarly, we can see here how the response is mitigated, and the template is rendered and injected into the response.

We are seeing more and more systems arise in the Laravel community, which are helping to decouple the business logic from the presentation layer. Things like Livewire Components enable developers to centralize logic into single files. It also allows developers to remove JavaScript coding requirements, resulting in less maintenance. However, it doesn't resolve the issue of business logic being in both the presentation layer and the Component itself. Systems like Inertia enable developers to build applications with a more formal REST backend and have their VueJS components contain all the presentation layers and the corresponding logic. In either of these cases, we still see the persistence of logic crossing multiple application layers. There are some valid critiques of ADR, which Paul M. Jones has, in some cases, addressed in his GitHub repo detailing the pattern. Building applications with this pattern can become verbose with numerous files, many of which are very small and somewhat repetitive. There are also concerns about logic being placed poorly across these layers as well. We can see that we gain the option of more granular testing and overall improvements to the readability of the code within the ADR pattern, but ultimately, it comes at a cost. In the outlined case above, we can see some easy ways to split up our view handling and reduce the probability of injecting logic into the presentation layer.

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