www.phparch.com

September 2022
Volume 21 - Issue 9

php[architect]

The Magazine For PHP Professionals

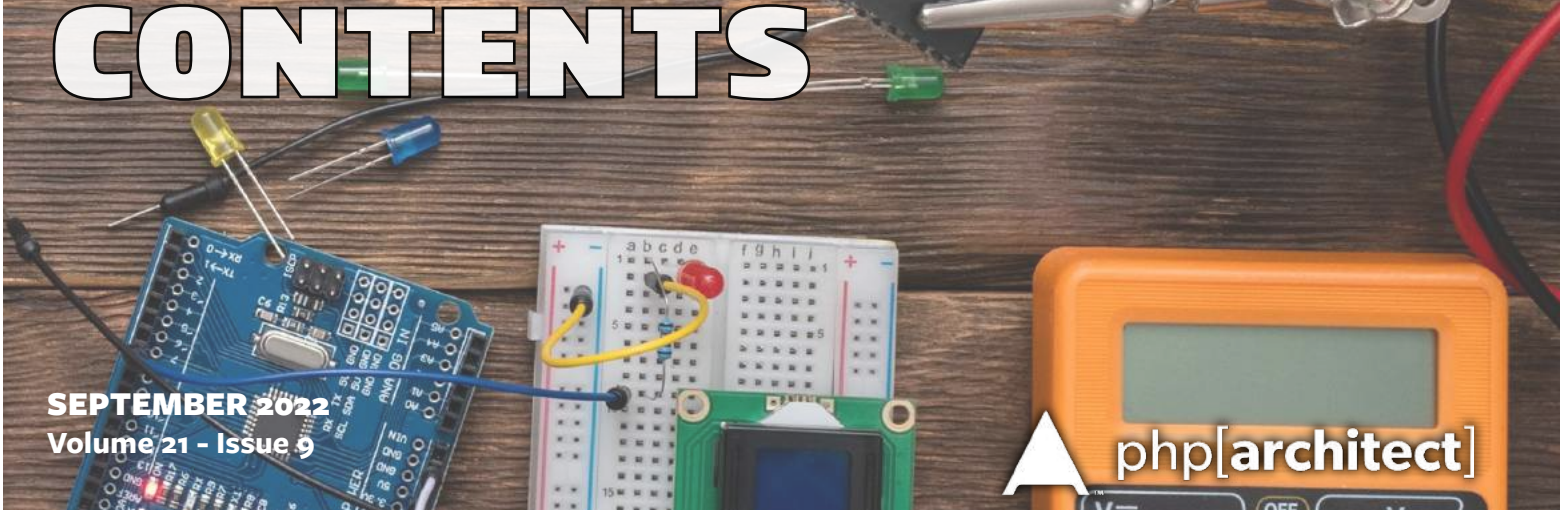
Making Code

Real World Abstract Syntax Trees

Universal Vim Part Two

ALSO INSIDE

CONTENTS



SEPTEMBER 2022
Volume 21 - Issue 9

php[architect]

Edited in a workshop

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

Back To School

John Congdon

Summer time is coming to an end and school's back in session. Let the learning begin, not just for the kiddos, but for the developers in our community as well. We are going to take a deep dive into learning a wide range of topics including Abstract Syntax Trees, Vim, Design Patterns for building code, Domain-Drive Design, Caching, DevOps, and so much more.

This month marks the start of a new school year for most students, at least here in The United States. If you have kids, the summer break can be an amazing time to spend with your family and take the extra time to reconnect before getting back to the hustle and bustle of everyday life. Something that I like to do for my kids is to model continuing education. I try to show them that we never stop learning, no matter how good we may be at the things we do, there's always room for improvement.

Back to school also reminds me of the mission of PHP Architect. Our goal is to provide resources to our fellow coders to further their education in this ever-changing ecosystem. There is so much to learn, and we use this platform to keep learning ourselves. Our magazine is meant to help you be aware of various libraries, best practices, tools, security, and more. Learning about these resources, that may not apply to anything you are working on today, can help you realize a certain solution is possible in the future and give you the lightbulb moment, "I recall reading something about this in the PHP Architect magazine". We also provide conferences to get our friends (that includes you reading this) together to learn from each other. Want to be in the know about potential upcoming conferences? Please follow us on social media and subscribe to our newsletter. Links available on phparch.com.

We are proud to be a resource of continuing education for so many people in our community. We ourselves learn something new every month we publish this magazine.

Are you interested in helping others, including us, learn? Have you recently learned something new that you'd like to share and simultaneously become a "published author"? Write for us here in PHP Architect; it is fun, and you can earn a little extra spending money. Contact us at write@phparch.com with your idea, and we can get a conversation going.

So let's dive into this month's learning adventure with "Real World AST" by Tomas Vortuba, which will start laying the groundwork for teaching us how to parse our own legacy code. Then Andrew Woods returns with "Universal Vim Part Two: Fuzzy Search Fun" to teach us some new plugins for Vim.

Oscar Merida teaches us some "Fractional Math" in this month's PHP Puzzles. We continue learning about the various PHP Standard Recommendations in PSR Pickup with "PSR-6: Caching Interface" by Frank Wallen. Learn about speeding up your Drupal 9 application using Varnish in this month's Drupal Dab by Nicola Pignatelli. Then "Making Things Happen" in The Workshop with Joe Ferguson will teach us how to use a Makefile to simplify our lives. Figure out how to beat the statistics to have a long-lived career in cybersecurity with Eric Mann's Security Corner article, "Surviving Cybersecurity." Continue down the Domain Driven Design road with Edward Barnard's "Domain Event Walkthrough." And finally, looking at things through "Everyone's Eyes and Ears" helps us learn that while lots of input, even on a simple feature, can be overwhelming, everyone has their own life experience determining their own specific goal(s) for the feature.

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
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download the Code Archive:

https://phpa.me/September2022_code

Real World AST

Tomas Votruba

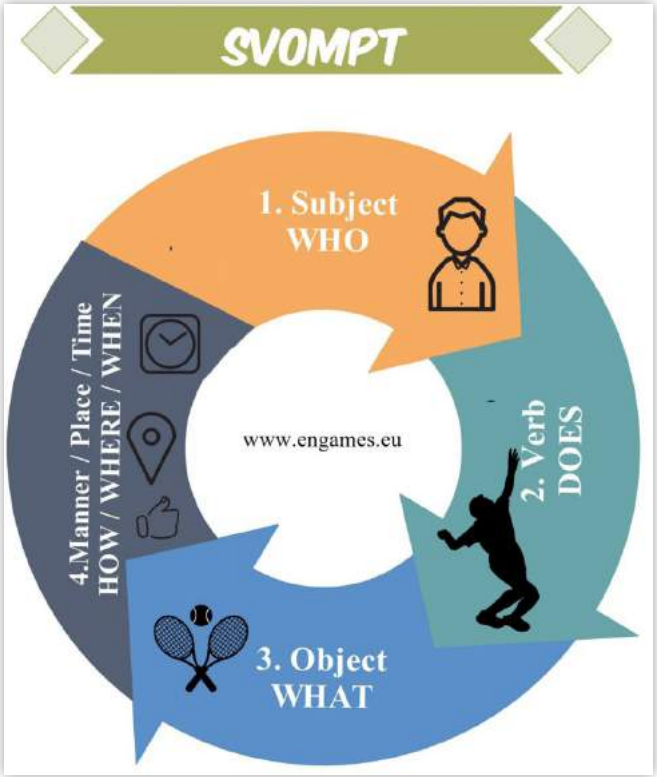
AST is a technology that changed my life and the way I view code. I want to share this way of seeing code so that you can deal with your "impossible problems" in a fancy and lazy way.

AST is a technology that changed my life and the way I view code. How? Let's say a client comes to me with a project. "Hi Tomas, we have 2000 static method calls. We want to migrate to a"2022 framework" and need dependency injection. Everyone tells us it's impossible and that we must rewrite the whole project. What do you think?"

After a few moments, I can reply: "I see a fairly simple algorithm using AST to achieve that. Let's do it".

You don't need years of experience. I'm no more intelligent than you. I don't have any science engineering degree, and my parents were not government employees with early access to information technology. This is only the know-how of a single pattern.

Do you know about dependency injection? It's not magic, right? It's a simple and effective way to get a dependency via a constructor. But if you don't know what dependency injection is and what rules it follows, coding a project with 50 classes might be challenging (and it may become a real mess for the next developer who will take over the project after you). If you know dependency injection pattern, you don't even think about it when writing a constructor.



I want to share this way of seeing code so that you can deal with your "impossible problems" in a fancy and lazy way.

What is an Abstract Syntax Tree?

Abstract syntax tree (AST) is a tree structure that represents the structure of a programming language. It's from a similar group of terms like "dependency injection". It's not limited only to the PHP community or PHP language itself. It's a pattern that can be applied in any language once you know what to look for.

Most people who write about AST try to explain it through a picture of an abstract syntax tree representing PHP expression. But it's not very helpful, as its tautology. It's explaining the term by using the term itself—like describing the movie by playing it.

Instead, it's easier to start with a simple example we learn in school:

S V O M P T

English sentence order.

S V O (Subject Verb Object)

e.g.: Tomas writes a post

S V O P T (Subject Verb Object Place Time)

e.g.: Tomas wrote a post in a coffee shop today.

That is a structure **that splits one branch into a few more detailed ones**:

- Text splits into sentences.
- A sentence splits into words - required ones (S, V, O) and optional ones (M, P, T)
- A subject can have detailed adjectives - extremely lazy Tomas

And so on. When we learn this English sentence structure rule, we **suddenly see every sentence through these glasses**. If a teacher tells us to find all the subjects in the text, we can do it very quickly.

If a programmer created English grammar, it would be called "ASVOT", "Abstract Subject Verb Object Tree".

Seeing the Code As Ast

Okay, now we know how to find all subjects in the text. Let's move to code.

Could you find all **the variables** in the following code? See Listing 1.

This is way easy, right? There are 3—`$article`, `$author`,and `$magazine`.

I have a surprise for you. You've just used AST directly in your brain without even knowing it!

How do you know it's a variable? Let's reflect on what we did—slowly. We look for a pattern that creates variables. Probably a string starting with dollar sign $, then alphanumeric strings, and then by assign =.

Listing 1.

```
$article = new Article('How anyone can learn AST fast?');

$author = new Author('Tomas');
$author->writeContentForArticle($article);

$magazine = new Magazine('PHP Arch');

$author->deliverFinishedArticleToMagazine($article, $magazine);
```

```
$<variable_name> = ...;
```

Anything that matches this pattern is a newly created variable. What we've described in the previous sentence is called *grammar*. What a coincidence with the English language. S V O is the grammar rule:

- Tomas drinks coffee—valid
- Coffee drinks Tomas—invalid (I hope)

Did you know many programming languages use grammar to parse their syntax? You can take a look at what the PHP grammar[1] looks like by checking out the source code.

Traversing the Nodes

When we looked for all the subjects, we already performed a type of operation. We traversed a tree—first, we split the whole text into sentences:

```
* text
|
  * sentence
  * sentence
  * sentence
```

Then one by one, we read a sentence in more detail:

```
* sentence
|
  * word
  * word
  * word
```

We *traversed a tree* of sentences.

As usual, behind any fancy name is nothing special. Traversing a tree is basically reading every big item, then entering all smaller items. Then we read these smaller items and enter all of its even smaller items until we read every single letter.

Traversing a PHP tree is no different. We read from the higher items to the lower ones, then even lower ones until we read every piece of code.

Let's put the example above into a PHP code.

We have three classes, or *nodes*: `Text`, `Sentence` and `Word`. See Listing 2, 3 and 4.

Listing 2.

```
final class Text
{
    /**
     * @param Sentence[] $sentences
     */
    public function __construct(
        public array $sentences
    ) {
    }
}
```

Listing 3.

```
final class Sentence
{
    /**
     * @param Word[] $words
     */
    public function __construct(
        public array $words
    ) {
    }
}
```

Listing 4.

```
final class Word
{
   public function __construct(
       public string $content
   ) {
   }
}
```

1 *PHP grammar: https://phpa.me/ZendLanguageParser*

Now we compose those 3 objects together. What would a sentence such as "Tomas drinks coffee" look like? See Listing 5.

How do we traverse them? With a service called *node traverser*. We already know what traversing is:

- *We read from the higher items to the lower ones, then even lower ones until we read every piece of code.*

What would a generic PHP class that can traverse a whole text look like? See Listing 6.

Listing 5.

```
$words = [
    new Word('Tomas'),
    new Word('drinks'),
    new Word('coffee'),
];

$sentence = new Sentence($words);

$text = new Text([$sentence]);
```

That's it. The `NodeTraverser` visits every `Text`, `Sentence`, and `Word` node. You've probably noticed that the `$this->visit()` method is called thrice on each element. This is to ensure we visit every node of the whole node tree.

This class is generic, and it will not change. You can put in any text, long or short, with few words or a whole book. This class will traverse 100 % of any provided content.

Listing 6.

```
final class NodeTraverser
{
    public function traverse(Text $text): void
    {
        $this->visit($text);

        foreach ($text->sentences as $sentences) {
            $this->visit($sentence);

            foreach ($sentences->words as $word) {
                $this->visit($word);
            }
        }
    }

    private function visit($node)
    {
        // ...
    }
}
```

How to Visit a Node

So what will change? The content of the `visit()` method. It depends on what exactly we need at the moment. Let's say we want to count how many times the word "Tomas" appears in a text. What would the visit method look like? See Listing 7.

Three Basic Elements in an Abstract Syntax Tree

Listing 7.

```
private int $tomasCounter = 0;

public function getTomasCounter(): int
{
    return $this->tomasCounter;
}

private function visit($node)
{
    // we only look for the `Word` object, nothing else
    if (! $node instanceof Word) {
        return;
    }

    // we need the value to be exactly "Tomas"
    if ($node->content !== 'Tomas') {
        return;
    }

    // we found it! let's increase the counter
    $this->tomasCounter += 1;
}
```

Everyone loves acronyms. They help us to remember the basic structure of complex systems. Most PHP frameworks are built with MVC architecture—model, view, and controller. Thanks to this separation, we know where to put what part of application. (*grammar wise 'application' needs 'the' or 'an' in front of it but not sure if it works code wise?)

Now we're going to learn a new acronym that will help us navigate with abstract syntax trees—**NTV**:

- **nodes**
- node **traverser**
- node **visitors**

We've just made them all. This is a generic rule that applies to any language. In PHP, YAML, NEON, TWIG, Latte, HTML, and XML… we always find these three elements.

1. Nodes - Basic Building Bricks

The node is the bare building brick of an abstract syntax tree. The tree is composed only of these node objects, and it's usually a class defined in the specific parser. The amount and names of nodes depend on the language domain and its

complexity. E.g., in PHP, there might be ~100 nodes, but in YAML, there could be as few as 10.

It's very specific and contains the value itself and all its children, another unwritten rule shared by various parsers.

The nodes usually share similar logic, e.g., data about line positions, file names, or the ability to add generic metadata. These are usually extracted in a single `AbstractNode` that every node inherits from.

United Traverse Direction

That's why we say the abstract syntax tree is always **traversed from top to bottom**. We can traverse from text to sentence, then to word. But we can't traverse from a word to the text.

In my experience of 6 years with ASTs, I see there are two reasons for that:

- There must be one direction for recursive calls to work
- The tree is an array of objects that we can modify
  - If we modify the most bottom element than the top element, then again the bottom element - we can get quickly to circular changes, and the script will crash
  - Another result can be missed as an item in the parent array because it was already traversed

We always modify the current node or its children by convention. But never the parent node.

2. Node Traverser - Will See Every Node

The node traverser is quite generic across all platforms. The idea is simple: it goes to every node, then to all its children. This repeat recursively until we reach nodes without children.

The solution for `NodeTraverser` we created above is not very good and is locked to explicit node types. What if we add a new node, e.g., `Letter`? We would also have to extend the node traverser, and that seems wrong, doesn't it?

Instead, the `NodeTraverser` must be closed for modification and opened for extension. How can we make it more generic and independent of specific node classes?

The secret is providing a list of public properties in the node itself. E.g., the `Text` would provide `sentences`, because that is the property with all the sentences. See Listing 8.

Now we only need to modify the `NodeTraverser` to work with `getSubNodeNames()` method: See Listing 9.

That's it! This is how generic `NodeTraverser` works. It can traverse any node class you create later, and it will visit every node.

Listing 8.

```
final class Text
{
    // ...

    public function getSubNodeNames(): array
    {
        return ['sentences'];
    }
}
```

Luckily, you don't need to understand this part in-depth. The node traversers are usually very well handled by the parser package. Unless you write your own parser and traverser, you can save the cognitive capacity for the coolest part of the AST—node visitors.

Listing 9.

```
final class NodeTraverser
{
  public function traverse(Node $node): void
  {
    $this->visit($node);

    foreach ($node->getSubNodeNames() as $subNodeName) {
      foreach ($node->$subNodeName as $subNode) {
        // this recursive call will jump back to the
           // 1st line of this method, but with a subnode
        // that way, we iterate all the
          // child nodes of the child nodes
        $this->traverse($subNode);
      }
    }
  }

  private function visit($node)
  {
    // ...
  }
}
```

3. Node Visitors - a Custom Script That You Write

The node visitors are the fun part of AST:

- They allow us to **extract information from the code**, e.g., how many times a class is used in YAML configs,
- And also **modify the code itself**, based on criteria, e.g., rename class from old to a new one in YAML config

So how do we write the node visitors? Do we put everything to the `NodeTraverser`? That would soon result in a messy god class. Imagine having 30 conditions in a single `visit()` method.

Generic Node Visitors

How can we refactor the `NodeTraverser` again, to be closed for modification but opened for extensions? We'll introduce a `NodeVisitorInterface` that will hook into the specific node. Then we allow to add of any number of node visitors to the `NodeTraverser`: See Listing 10.

Now we can add a new node visitor from the outside and keep `NodeTraverser` architecture untouched and clean:

```
$tomasCountNodeVisitor = new TomasCountNodeVisitor();

$nodeTraverser = new NodeTravser();
$nodeTraverser->addNodeVisitor($tomasCountNodeVisitor);
```

Listing 10.

```
1. final class NodeTraverser
2. {
3.   /**
4.    * @var NodeVisitorInterface
5.    */
6.   private array $nodeVisitors = [];
7.
8.   public function addNodeVisitor(
9.       NodeVisitorInterface $nodeVisitor
10.  ) {
11.    $this->nodeVisitors[] = $nodeVisitor;
12.  }
13.
14.  public function traverse(Node $node): void
15.  {
16.    $this->visit($node);
17.
18.    foreach ($node->getSubNodeNames() as $subNodeName) {
19.      foreach ($node->$subNodeName as $subNode) {
20.        $this->traverse($subNode);
21.      }
22.    }
23.  }
24.
25.  // then, in the node visit() method,iterate the node visitors
26.  private function visit($node): void
27.  {
28.    foreach ($this->nodeVisitors as $nodeVisitor) {
29.      $nodeVisitor->visit($node);
30.    }
31.  }
31. }
```

Then we have to traverse the node to invoke the visitors:

```
$text = new Text(...);

$nodeTraverser->traverse($text);

echo $tomasCountNodeVisitor->getTomasCounter();
```

If everything works, we'll get the output with the value:

```
1
```

It works, yay!

Wait, we missed quite an important part. What does the `TomasCountNodeVisitor` look like? And what about the `NodeVisitorInterface`?

The latter is straightforward:

```
interface NodeVisitorInterface
{
    public function visit(AbstractNode $node);
}
```

It receives any node as an argument, and the destiny of the node awaits the specific implementation of every node visitor.

Then we move the `visit()` class method contents that we wrote above to the standalone `TomasCountNodeVisitor` class: See Listing 11. That's it. We're good to go.

Listing 11.

```
1. class TomasCountNodeVisitor implements NodeVisitorInterface
2. {
3.     private int $tomasCounter = 0;
4.
5.     public function getTomasCounter(): int
6.     {
7.         return $this->tomasCounter;
8.     }
9.
10.    public functio visit(AbstractNode $node)
11.    {
12.        // we only look for the `Word` object, nothing else
13.        if (! $node instanceof Word) {
14.            return;
15.        }
16.
17.        // we need the value to be exactly "Tomas"
18.        if ($node->content !== 'Tomas') {
19.            return;
20.        }
21.
22.        // we found it! let's increase the counter
23.        $this->tomasCounter += 1;
24.    }
25. }
```

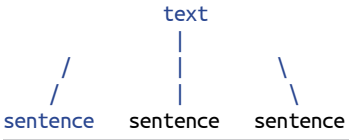
Summary: Basic Elements of Ast, How It Works

Before we go to the next part, let's summarize what we've learned.

The abstract syntax tree is a **way of seeing code**. In the same way English grammar helps us to see sentences, subjects, verbs, and objects; language grammar helps to create logical elements. In the abstract syntax tree area, these elements are called nodes.

The **nodes are essential building elements** of the node tree. The connections between nodes are possible thanks to **child nodes**.

E.g., one text node contains many child sentence nodes.



How can we access nodes? Through a node visitor. **The node visitor is a service with one job—go through all nodes and their children**. It uses simple recursion. If a node has children, the traverser will traverse them. If those children

nodes have children, the traverser will traverse them too. This will repeat recursively until the traverser reaches the node without children.

The nodes and node traverser are usually defined by the parser package, e.g., for PHP, it is `nikic/php-parser`; for TWIG, it's `twig/twig` package, and so on. We do not touch them, just use them.

On the other hand, the node and node traverser themselves don't do anything. It's like reading every sentence of this article and forgetting everything. But in real life, we have a goal in mind. That's why we read. We want to learn new knowledge to solve our problems at work or read a link that can give us more detailed information on topics we love.

That's where node visitors come to play. **Node visitor is a service we write from scratch and always has a goal.**, E.g., find the number of "Tomas" used or rename the class. Node visitor visits every node, and then we can read or modify it.

Coming in the Next Article

The understanding of three components of AST is really important because in the upcoming 2nd post, I'll show you **how to build real scripts that will modify code for you**. Not just one-time scratch code that we run and delete, but scripts that will have your back in the CI on every commit.

We'll look at how we can use AST to modify not only PHP, but also YAML or TWIG. From that, you'll see how to create such an AST and how to parse any file in your PHP project. By the end, you'll see how **effortless it is to work with legacy code** when you know the right tool and identify the pattern you look for.

Happy coding!

Tomas Votruba is a regular speaker at meetups and conferences and writes regularly. He created the Rector project and founded the PHP community in the Czech Republic in 2015. He loves meeting people from the PHP family so he created FriendOfPHP which is updated daily with meetups all over the world. Connect with him on twitter votrubaT. @votrubaT

Universal Vim Part Two: Fuzzy Search Fun

Andrew Woods

We continue our quest to craft a universal vim experience by enhancing your ability to search your content using FZF, a performant fuzzy finder.

In part one of the Universal Vim series, we created the base of our universal vimrc configuration. We kept it general enough to be easily adapted for various purposes. It built the UI and provided settings to create the beginning of an IDE-like user experience. If you're a Neovim user, you'll eventually want to check out Telescope, which can do much of what will be discussed here. However, Telescope is not available for Vim, as it's built on features in the Neovim core. Now that that's out of the way, let's continue.

In this article, we continue improving upon our universal configuration. This time focused on search. We'll further our pursuit of creating an IDE-like experience with a great command line tool, **FZF**. Just like no man is an island, no tool works entirely in isolation. FZF uses other command line tools behind the scenes, namely RipGrep, fd, and bat. RipGrep (rg) is a highly performant replacement for grep. The `fd` command is an enhanced replacement for the `find` command. The `bat` command is used to preview the contents of files.

What exactly is FZF? FZF is a fuzzy finder. It's an application that makes it easy to filter a list of text in an imprecise manner. Naturally, more precise results "float to the top," but it's the lack of perfection that makes FZF user-friendly and powerful. FZF uses smart case searches by default. You'll remember from last month's issue that we configured Vim to use smart case searches. If lowercase letters are used, the search is treated as insensitive, but when uppercase letters are used, the search becomes case-sensitive. FZF is both a command line tool and a Vim plugin.

Before we continue, let's create a `bin` directory under your home directory. This is great to have on your system. It provides you with a place to find all your executable scripts and programs. We'll use it later. It's not enough to create the directory. We also need to update our path. Let's do this now.

```
$ mkdir -p $HOME/bin
$ vi ~/.bash_profile
```

In your `~/.bash_profile`, you'll want to add `PATH=$HOME/bin:$PATH` below your current PATH variable. This will put your new bin directory at the front of your PATH. Usually, you can source a bash file. However, since you're updating with the PATH, creating a new shell session is the safest thing to do.

FZF On the Command Line

FZF was created by Korean software developer Junegunn Choi, who actively maintains it. FZF takes a list of content and filters the list by fuzzy matching each line against your search term, removing what doesn't match. While the idea of FZF may be unfamiliar, if you've never heard of a fuzzy finder, it's fantastic. It will enhance your workflow. The beauty of FZF comes from its simplicity. FZF narrows those results to focus on what's important to you.

The project documentation is available on the FZF GitHub repo at https://github.com/junegunn/fzf[1]. While this article will get you started with FZF, you'll likely find a gem or two when you dive deeper by exploring the documentation.

Command Line Installation

FZF has been around for a while, so it's available in many package managers. However, if you're the type who likes to stay up-to-date, you can clone the git repo. FZF has support for Bash, ZSH, and Fish. Say yes to the key bindings. You're going to want them.

```
$ git clone --depth 1 <https://github.com/junegunn/fzf.git> ~/.fzf
$ ~/.fzf/install
```

Earlier I mentioned tools to make FZF more effective. Each one has its own documentation on how to install it. Please refer to each project's documentation.

- RipGrep https://github.com/BurntSushi/ripgrep[2]
- fd https://github.com/sharkdp/fd[3]
- bat https://github.com/sharkdp/bat[4]

Command-line Usage

The command line experience includes more than the binary. It also contains three key bindings. These key bindings are great ways to trigger FZF on the command line. Once you learn them, you'll find ways to use them. They add flexibility and efficiency to your command line experience.

1 *https://github.com/junegunn/fzf:* https://github.com/junegunn/fzf

2 *https://github.com/BurntSushi/ripgrep:* https://github.com/BurntSushi/ripgrep

3 *https://github.com/sharkdp/fd:* https://github.com/sharkdp/fd

4 *https://github.com/sharkdp/bat:* https://github.com/sharkdp/bat

Here's the summary of what they do.

```
CTRL+R - FZF enhanced shell history
CTRL+T - FZF search for a command argument
ALT+C - Change to the directory you fuzzy searched
```

Each of these key bindings has shell variables that affect its operation. This is just one of the many reasons why you'll want to read the documentation. It will further help you to enhance your experience.

Searching Your History

If you've used the command line for any length of time, you've already used CTRL+R to search through your bash command history. It's not the best experience, however. It shows you the next result each time CTRL+R is pressed. Another option is to use the history command and pipe it to less, so you can use the / command to search through the results. *That's without FZF!* You can do this right now. However, by using FZF, you combine these ideas with FZF's ability to filter the results elegantly.

```
# Without FZF
history | less

# With FZF
CTRL+R
```

Selecting Files and Directories

The `CTRL+T` keybinding is powerful and quite useful. It will include files and directories and allows you to select multiple items. The thing that makes a tool like FZF great is being able to adapt it to your workflow easily. FZF works by looking in your current directory and below. To leverage its power, you'll want to change your current directory to your project root directory. An easy way to start is to type `vi` then `CTRL+T` to fuzzy search through your project files. When working on a CMS, type a module name if you can't remember a specific file. That will get you a result set you can further filter. Remember in our previous article when we enabled `set exrc`? If you have a vimrc in your project root, vim will also load that to enable your project-specific settings. That's 2 great features enabled by starting vim from your project root.

Changing Directories

Depending upon how deeply nested your desired directory is, it's usually faster to fuzzy find the directory instead of using tab completion. You might think of typing `cd $(fzf)`, but that will cause an error. Not to worry though, it's even easier than that. `ALT+C`, That's it. FZF will create the `cd` command once you choose your directory. FZF has the advantage of showing you all the directories as a list ahead of time, whereas tab completion requires you to select the directories one level at a time.

Ripgrep Integration

FZF is the belle of the ball in this article. But before FZF can filter its results, that means being able to find the correct files first. If you only remember the content you changed but not the file's name, FZF alone can't help you. RipGrep, however, will rip through all your files for that incantation you so elegantly crafted in your source code. RipGrep ignores files and directories specified by .gitignore. It's also written in Rust. These things make RipGrep very performant.

rfv—RipGrep FZF Viewer[5]

Here's a script that comes directly from the FZF author Junegunn. Remember that bin directory we created earlier. Here's where we make use of it. Copy the code for the "rfv—RipGrep FZF Viewer" and paste it into your ~/bin directory. I'll refer to it here as `rfv`. The script takes one argument and sends it to RipGrep. Then RipGrep takes each matching line, including the filename and line number, and passes it to FZF. As you scroll the results, FZF passes the info to the previewer. In our case, it's bat. This shows you the matching line and the surrounding content. When you press Enter, the `rfv` script will open that file in Vim to that line number. **NOTE:** If you're a Neovim user, you'll need to tweak the last line of the `rfv` script to use Neovim instead.

FZF Command Examples

The easiest command to type is `fzf` without any parameters. Without any other content, FZF will find all the files in the current directory and below. When FZF displays its results, the bottom file is highlighted, with the search prompt below the results, and on the left half of the window. The preview displays on the right. The position of the prompt is configurable, as is the preview display. If you're a PhpStorm user, you may prefer seeing the file search prompt above the file list and the content preview below the file list.

```
# Use the default command and options
$ fzf
```

You can also pipe content to FZF from another command. Anything that generates a list is a good candidate, whether it's a list of processes, records, or even `php --info` output. When you want to select multiple items, use the tab key.
See Listing 1.

Listing 1.

```
1. # Display processes
2. $ ps -ef | fzf
3.
4. # Choose one or more processes to kill
5. kill $(fzf --multi)
6.
7. # Filter RipGrep results
8. # search for "function" in the bash files of your dotfiles
9. rg --type sh ^function  ~/dotfiles | fzf
```

5 *rfv—RipGrep FZF Viewer: https://github.com/junegunn/fzf*

FZF has many settings you can play with. Since we took the time to install some additional tools, let's make use of them. Update your bashrc file with the following snippet. See Listing 2.

The `fzf` command has a `--preview` option that allows you to specify the command you want to use for previewing your files. If you want to use the UNIX standard `cat` program, that's possible. Check the man page on your system. The `-n` displays line numbers for all lines. The curly braces {} represent the filename.

```
$ fzf --preview 'cat -n {}'
```

Instead, if you want to use the `bat` command we just installed, we can do that too. The `bat` command has more options than the typical cat command. To leverage bat's abilities, pass the bat command and its options in the preview option of FZF.

```
$ fzf --preview 'bat --style=numbers \
            --color=always --tabs=4 {}'
```

You may be tempted to add the `--preview` option to the `FZF_DEFAULT_OPTS` variable. However, the FZF documentation *explicitly tells you not to*. You don't want to type all that out. I get it. So I suggest making an alias so you use it when you want. I'll call it `fzfp` to keep it short.

```
# Put this in your ~/.bashrc
alias fzfp="fzf --preview 'bat --style=numbers \
            --color=always --tabs=4 {}'"
```

The FZF Vim Plugin

It was important to explain FZF from the command line perspective. If you're anything like me, you probably live in the terminal. So it only makes sense to enhance your CLI experience as much as possible. However, we want to strengthen our Vim experience with FZF. The vim plugin makes use of command line FZF, particularly the environment variables.

Vim Plug Installation

You can skip this section if you already have a plugin manager for Vim. However, If you still need a plugin manager, follow along and **install Vim Plug**. FZF author Junegunn Choi also created this project. We'll use Vim-Plug to manage all our plugins—for this article and the next.

The Vim Plug project is on GitHub at https://github.com/junegunn/vim-plug[6].

Download plug.vim and put it in the "autoload" directory. The following command will do that for you.

```
$ curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
    https://phpa.me/junegunn-vim
```

6 https://github.com/junegunn/vim-plug

Listing 2.

```
1. export FZF_DEFAULT_COMMAND='fd --type f'
2.
3. FZF_DEFAULT_OPTS=""
4. FZF_DEFAULT_OPTS+=' --multi'
5. FZF_DEFAULT_OPTS+=' --tabstop=4'
6. FZF_DEFAULT_OPTS+=' --border=sharp'
7. FZF_DEFAULT_OPTS+=' --layout=default'
8.
9. export FZF_DEFAULT_OPTS
```

If that doesn't work for you for some reason. You can download it from the GitHub repo with your browser to your Downloads directory. Then execute these commands.

```
$ mkdir -p ~/.vim/autoload
$ mv ~/Downloads/plug.vim ~/.vim/autoload
```

Restart vim for the changes to take effect. Now that Vim Plug is installed, you'll need to update your vimrc file to specify your plugins. In your ~/.vimrc (or ~/.config/nvim/init.vim for Neovim users), find the plugins section header we created in the previous article, and add the following code. See Listing 3.

Listing 3.

```
" ==== Plugins ======================================
"

call plug#begin('~/.vim/plugged')

" add plugins here

call plug#end()
```

Vim Plug is entirely configurable while being easy to use. For the purposes of this article, the Vim Plug aspect will be minimal. One thing worth noting is that by default Vim Plug assumes that your desired plugin is on GitHub. For example, the line in your vimrc `Plug 'junegunn/fzf.vim'` will get its code from https://github.com/junegunn/fzf.vim[7]. You're encouraged to read through the docs on its github repo.

Fzf.vim Plugin Installation

The fzf.vim plugin is on GitHub at https://github.com/junegunn/fzf.vim[8], and it's chock full of information in its documentation. To install the `fzf` and `fzf.vim` plugins, you'll want to update your vimrc plugin section to look like the following. See Listing 4 on the next page.

Reload your vimrc and run `:PlugInstall` to complete the installation. Occasionally, I've needed to run the `:PlugInstall` function a second time for the changes to take effect. To

7 https://github.com/junegunn/fzf.vim

8 https://github.com/junegunn/fzf.vim

Listing 4.

```
" ═══ Plugins ═══════════════════════════════
"

call plug#begin('~/.vim/plugged')

Plug 'junegunn/fzf', {'dir':'~/.fzf', 'do':'./install --all'}
Plug 'junegunn/fzf.vim'

call plug#end()
```

ensure it works, hit ESC to enter normal mode and run `:Files`, and you'll see FZF bring up a list of files.

FZF provides you with many settings to control your experience. There are several variables, but there are four key variables that we'll discuss four key variables.

The `g:fzf_action` provides customizable extra key bindings for opening selected files in different ways. When FZF displays our results, the easiest thing to do is press enter to make the file our current buffer. However, we have more options available to us. You may want to split your screen into two buffers. By pressing CTRL+X, you can split your buffers horizontally. Alternatively, you can press CTRL+V to split your buffer vertically. You may decide that you don't want to split your buffer at all. Instead, use CTRL+T if you'd rather open the file in a new tab. These are the default settings. However, I've included them in the `g:fzf_action` variable to make them more discoverable—in case you want to change them.

The actions triggered are as follows.

```
let g:fzf_action = { \
    'ctrl-t': 'tab split', \
    'ctrl-x': 'split', \
    'ctrl-v': 'vsplit' \
}
```

If you don't typically split your buffers in Vim, you may be surprised that it didn't split in the way you wanted. If you saw two files—one on the left of your screen and one on the right you might assume they were split horizontally. You would be wrong. Vim uses the swordsman method (TM). Imagine yourself a Celtic warrior with a broadsword. An enemy on the battlefield steps toward you. You swing your sword horizontally along their waist. You've divided them into two pieces in one swing—top and bottom. Another enemy attacks, but this time you raise your sword above your head and bring it down upon them, dividing them into their left and right halves. So remember, when you're splitting buffers in Vim—it's not about where the buffers sit but how you swing your sword.

The `g:fzf_layout` variable determines the size and position of the FZF window. With this variable, you can determine how much space you want it to consume. You can control its height and width, among other aspects.

```
let g:fzf_layout = { 'window': { \
    'width': 0.9, \
    'height': 0.9, \
    'relative': v:true \
    } \
}
```

The `g:fzf_colors` variable customizes FZF colors to let you match the current color scheme. The `g:fzf_history_dir` variable enables the history feature. The fzf_history_dir setting is a "set it and forget it" kind of setting. It's important, though, because the default function of FZF uses it to display results for files, and it helps manage FZF's history.

```
let g:fzf_history_dir = '~/.local/share/fzf-history'
```

The `g:fzf_preview_window` setting determines how you want to display a preview of the file results. Earlier, we talked about using the `bat` command. The FZF Vim plugin makes use of it too. As you highlight a file, the contents of that file will be previewed on the right half of the FZF window. You may want to temporarily hide the content preview. You can press `CTRL+/` to hide it, then again to re-enable it.

```
" Default Preview
"let g:fzf_preview_window = ['right:50%', 'ctrl-/']
```

You might decide that you prefer to hide the file preview by default. You can do that by updating the first item of the `g:fzf_preview_window` variable.

```
" Default Preview Hidden by default
"let g:fzf_preview_window = ['right:50%:hidden', 'ctrl-/']
```

The Heart of Fzf in Vim

There are several functions provided by the `fzf.vim` plugin. Earlier, you learned that on the command line, FZF looks for files in the current directory and below. The `:Files` function is the equivalent in Vim. You have the option of selecting multiple files with the tab key and splitting your buffers all from this file list. Depending on your directory, you may be using Git to manage your files. You can use the `:GFiles` function to limit your search to files that have been checked in to Git. Visually, the list of files won't look different between `:Files` and `:GFiles`, other than possibly fewer files in the `:GFiles` output.

You may wonder—*how can you tell if Git manages a file from within Vim?* The `:GFiles?` function does just that. The output should look familiar. It's an abbreviated output of the `git status` command. Note the question mark at the end of the `:GFiles?`, as it's part of the name. I don't believe that was the best design choice, but you know what they say about naming things. Remember, FZF's primary job is fuzzy

searching text. You'll have to wait until the next article to read about managing Git from within Vim.

> *There are only two hard things in Computer Science: cache invalidation, naming things, and off-by-one errors*

You can do this with Vim when editing just one file. However, some features only become useful when you are editing multiple files. One of those features is listing your open buffers. FZF provides the `:Buffers` function, which makes it fast to switch between buffers. This is a more interactive version of Vim's `:ls` command.

The thing I haven't mentioned is how you're going to trigger these functions. Before FZF was created, some people used the older ctrlp.vim plugin, which also performed fuzzy searching. They tend to map CTRL+P to the `:Files` function. There are only so many single-letter CTRL (or ALT) key combinations between Vim and your operating system, so some existing functionality is bound to get clobbered. The use of function keys is even more precarious. I recommend making use of Vim's leader key. In the previous article of this series, we mapped it to the space character.

Let's start mapping FZF functions to our leader key. When you have just a few leader mappings, it doesn't really matter how you name them. But remember that this is where you have control over your muscle memory. I like to use a "verb noun" naming convention. It focuses you more on what you want to do—like opening something or displaying things. The other common naming convention is by topic. This can feel natural because then you can say, "All my Git mappings start with `g`". Choose what works for you. There's a danger of making your mappings bind too tightly to the action you want to take. How can you tell if you've done that? One way to tell is to ask yourself, "if you changed which command is called, does the mapping still make sense?". Here are some good defaults for FZF mappings. See Listing 5.

Without FZF, `nnoremap <Leader>ob :ls<CR>` could be your leader mapping for opening the buffer list. Should the day come when you change to another tool, you can update your mapping to call that function, yet still be able to trigger it with `<leader>ob`.

It's also helpful to keep your categorizations broad. Search, for example, could mean looking for content within a single file or within multiple files. In the future, you may also want to include file-type specific searches. You can't know what that search might be. Hopefully, you're convention will flexible enough to incorporate it. Remember, we installed RipGrep earlier. RipGrep will search our files. But we'll call it two different ways—the standard search that appears in the window and full-screen mode. It's tempting to use `<Leader>s` to mean 'search', but that breaks the "verb noun" convention. For this article, let's use 'search common', which makes our mapping `<Leader>sc`, and we'll use `<Leader>sf` to mean search full-screen.

Sometimes you don't always need to search the entire codebase. You may know the answer lies within your open buffers. FZF has provided two functions—`:Lines` and `:BLines`—to use for such occasions. The `:Lines` function searches all the open buffers, whereas `:BLines` limits your fuzzy search to the current buffer. See Listing 6.

There's a third and final grouping—display. These leader mappings are a little less intuitive to name. To create a mapping for displaying the Git status of your files, it was a coin toss. You could use `dg` or `ds` since it could mean "display git (status)" or "display status (for git)." It's worth noting that you can create a three letter leading mapping. So using `dgs` to mean "display git status" is valid. It just breaks the convention I established. The choice is yours.

The `:Marks` and `:Maps` functions begin with the same first two letters. I decided that "marks" means bookmarks, which is kinda how they are used. So I assigned the `:Marks` function to the `<Leader>db` mapping. This allowed me to use the `<Leader>dm` mapping. What Vim calls color schemes, most other applications call themes. Even in Vim, I call it a theme. So that's where the `<Leader>dt` mapping originates. The `:Colors` function is interesting. It demonstrates the FZF can deal with more than just files. The items you choose can be anything. You'll need to add some configuration, but FZF enables you to customize it to suit your needs. The great thing about Vim is that it's easy to change these mappings. So as you create new mappings, don't be afraid to change things. Nothing is set in stone. It's all about finding what works best for you. See Listing 7.

Listing 5.

```
" Open the buffers list
nnoremap <Leader>ob :Buffers<CR>

" Open any of our project files
nnoremap <Leader>of :Files<CR>

" Open any of our Git managed files
nnoremap <Leader>og :GFiles<CR>
```

Listing 6.

```
1. " Search
2. nnoremap <Leader>sc :Rg
3.
4. " Search in full screen
5. nnoremap <Leader>sf :Rg!
6.
7. " Search lines in All Buffers
8. nnoremap <Leader>sa :Lines<CR>
9.
10. " Search lines in Current Buffer
11. nnoremap <Leader>sb :BLines<CR>
```

For your convenience, you can view the entire vimrc on GitHub https://github.com/andrewwoods/universal-vim[9]

Conclusion

The thing that's great about FZF is that it feels fun to use while making you more productive. It reminds you that fun and productivity don't have to be adversaries. FZF has been around for a few years, but it feels like many people still don't know about it. That's part of why I wrote this article. Also, there's so much information in the documentation of the various repos that it felt necessary to provide a simple path to help get you started.

Listing 7.

```
" Git status
nnoremap <Leader>dg :GFiles?<CR>

" Display Marks
nnoremap <Leader>db :Marks<CR>

" Display Mappings
nnoremap <Leader>dm :Maps<CR>

" Choose from the themes (colorschemes) list
nnoremap <Leader>dt :Colors<CR>
```

While FZF isn't the only fuzzy finder, it does feel like the most comprehensive. It provides a great experience—both inside of Vim and out. One thing I noticed in the journey of learning to use FZF is the need to use a file explorer in Vim is greatly diminished. That's how powerful FZF is in improving your workflow. We've covered quite a bit here. However, there's still much left to learn. Albert Einstein said, "Imagination is more important than knowledge." How are you going to use FZF? Use your imagination. Go and explore FZF … and have fun doing it.

> *Imagination is more important than knowledge. For knowledge is limited to all we now know and understand, while imagination embraces the entire world, and all there will ever be to know and understand.—Albert Einstein*

Footnotes

Phil Karlton said, "There are only two hard things in Computer Science: cache invalidation and naming things[10]" and Roger Pate expanded it to include the "off-by-one errors[11]" clause

Andrew Woods is a Software Developer at Paramount. He's been developing for the web since 1999. When not coding or playing guitar, he enjoys films, music, and exploring the city. You can find him at andrewwoods.net, his online home, or on Twitter @awoods

9 *Universal Vim:* https://github.com/andrewwoods/universal-vim

10 *Phil Karlton Quote:* https://phpa.me/only-2-hard-things

11 *Roger Pate:* https://phpa.me/only-2-one-off

Building Code

Chris Tankersley

Patterns help us to put objects together. Two popular ways to instantiate new objects are via the Factory and the Builder patterns. Let's take a closer look at the impact these patterns have on our code.

Humans are pretty good at determining patterns. This is one of the many reasons that we exist at all. Early humans were able to detect changes in the visual and auditory patterns around them and be alerted to danger. Staring out across a vast sea of grass and noticing that a part of it is now moving? That's probably something coming to eat you.

While we no longer have to worry about the dangers of a lion wanting to eat us for lunch, that pattern recognition never went away. In the developer world, as we start to write more and more code, we see those patterns. Some of those patterns come from copy-and-pasted code, and others come from a similarity in design.

There are two patterns that I want to talk about today, and both of them come from recognizing patterns in how you put objects together. We often start writing a block of code and then find a use for it in other places. Sometimes our needs expand for that original code, but only just slightly.

A common place for that is instantiating objects. Our applications may spend a huge amount of time just instantiating new objects, and many of those objects may be very similar. While it is easy to copy the new code and alter it, we can use those patterns to our advantage.

Two popular ways to instantiate new objects are via the Factory and the Builder patterns. These patterns have evolved and appeared in many different languages, but they help serve two different creation problems—creating objects in multiple places and creating a variety of similar but slightly different objects. While they do the same thing (instantiate objects), their differences shine in their specializations.

Factories

Factories are one of the first types of "building" patterns that most developers encounter. The idea of a factory is to supply it with some data, and it will return a fully functional object. While you can do this in your business logic, a factory helps encapsulate that logic into one place and provides a clean API for building any number of similar objects.

Let's pretend that we have an application that returns some data to the user. In a perfect world, all we ever have to deal with is JSON, so lets have our API return a JSON response.

There is nothing special here. We output a header saying that we are returning `application/json` content, we declare we are returning a `200 OK` HTTP response code, and then we output the JSON itself. If this is all we ever need to do, we are all set.

```
class OurController {
  public function someAction() {
    // Magical business logic that creates a $returnData array
    header('Content-Type: application/json');
    http_response_code(200);
    echo json_encode($returnData);
  }
}
```

The first major refactor to this is that we probably have multiple places we need to return a response. So let's turn all of that into a small `JsonResponse` object and use that instead. See Listing 1.

Listing 1.

```
1. class JsonResponse {
2.   public function __construct(
3.     protected array $data = [],
4.     protected int $returnCode = 200,
5.   ) { }
6.
7.   public function output() {
8.     header('Content-Type: application/json');
9.     http_response_code($this->returnCode);
10.    echo json_encode($this->data);
11.  }
12. }
13.
14. class OurController {
15.   public function someAction() {
16.     // Magical business logic that
17.     // creates a $returnData array
18.     $resp = JsonResponse($returnData);
19.     $resp->output();
20.   }
21. }
```

This is fine as long as all applications accessing our API want JSON. A few weeks later, the client turns around and tells you that one of their older back office systems also needs to access the API, but it only speaks XML. How can we handle this?

The first thing that sticks out is to make an XMLResponse object, and then we can switch based on the header that is passed. See Listing 2 on the next page.

Listing 2.

```
1. class XMLResponse {
2.   public function __construct(
3.     protected array $data = [],
4.     protected int $returnCode = 200,
5.   ) { }
6.
7.   public function output() {
8.     header('Content-Type: application/xml');
9.     http_response_code($this->returnCode);
10.    $xml = new SimpleXMLElement('<root/>');
11.    array_walk_recursive(
12.      array_flip($this->data),
13.      array ($xml, 'addChild')
14.    );
15.    print $xml->asXML();
16.  }
17. }
18.
19. class OurController {
20.   public function someAction() {
21.     // Magical business logic that
22.     // creates a $returnData array
23.     switch(getallheaders()['Accept']) {
24.       case 'application/xml':
25.         $resp = new XMLResponse($returnData);
26.         break;
27.       default:
28.       case 'application/json':
29.         $resp = new JsonResponse($returnData);
30.         break;
31.     }
32.     $resp->output();
33.   }
34. }
```

In this case, we create an `XMLResponse` object that functions like our previous `JSONResponse`, but this time it should output everything needed for XML. Where we started to get complicated and error-prone is the switch statement in our controller. Not only do we end up with the same problem that we originally had, where we have to duplicate this in every endpoint, but this will only grow as we need to support more and more response types.

The answer here to all of this is a Factory. We can move all this decision logic to a central place and let the Factory determine what it is we need. We will give the factory all the basic information it needs to decide what kind of response we need. Let's sketch out what that will look like.

```
class ResponseFactory {
    static public function create(
        mixed $data,
        int $returnCode = 200
    ): ResponseInterface {

    }
}
```

Much like our response objects, we want to be able to pass in the data to be displayed as well as the response code. We should also declare what kind of thing we are returning. We want to be a bit forward-thinking, so instead of just an array, we want to take any sort of output. We change the type hint on `$data` from `array` to `mixed` to allow us to pass in any data. We also need to create an interface for our responses, `ResponseInterface`, our return type.

```
interface ResponseInterface {
    public function __construct(mixed $data);
    public function output(): void;
    public function setReturnCode(int $code): static;
}
```

Now we need to make sure our classes implement this interface. Since much of the setup logic is the same, and only the output logic differs, we will also create an `AbstractResponse` class for our responses to extend from. While we could have our existing classes implement the interface, we will still end up with copy-and-pasted logic when it comes to setting up the objects. See Listing 3.

Listing 3.

```
1. abstract class AbstractResponse
2.             implements ResponseInterface {
3.   protected int $returnCode = 200;
4.
5.   public function __construct(
6.     protected mixed $data,
7.   ) { }
8.
9.   public function setReturnCode(int $code): static {
10.    $this->returnCode = $code;
11.    return $this;
12.  }
13.
14.  abstract function output(): void;
15. }
16.
17. class JsonResponse extends AbstractResponse {
18.   public function output(): void {
19.     header('Content-Type: application/json');
20.     http_response_code($this->returnCode);
21.     echo json_encode($this->data);
22.   }
23. }
24.
25. class XMLResponse extends AbstractResponse {
26.   public function output(): void {
27.     header('Content-Type: application/xml');
28.     http_response_code($this->returnCode);
29.     $xml = new SimpleXMLElement('<root/>');
30.     array_walk_recursive(
31.       array_flip($this->data),
32.       array ($xml, 'addChild')
33.     );
34.     print $xml->asXML();
35.   }
36. }
```

Now our responses are nice and type-hintable, and all work basically the same. We can move that switch statement to our factory and clean up the business logic in our controller. The only difference that our Factory will do is set the return code explicitly to whatever was passed, but the net result is the same. The factory will return a response appropriate to the headers passed. See Listing 4.

Listing 4.

```
1. class ResponseFactory {
2.   static public function create(
3.     mixed $data,
4.     int $returnCode = 200
5.   ): ResponseInterface {
6.     switch(getallheaders()['Accept']) {
7.       case 'application/xml':
8.         $resp = new XMLResponse($data);
9.         break;
10.      default:
11.      case 'application/json':
12.        $resp = new JsonResponse($data);
13.        break;
14.    }
15.    $resp->setReturnCode($returnCode);
16.    return $resp;
17.  }
18. }
19.
20. class OurController {
21.   public function someAction() {
22.     // Magical business logic that
23.     // creates a $returnData array
24.     $resp = ResponseFactory::create($returnData);
25.     $resp->output();
26.   }
27. }
```

Here we can see the factory in action. We pass it the data that we want to return, and it handles determining what we want to get back. If we need to add additional return types, like an HTML version of the response, we can create a new Response class and add all the logic to the Factory. Our controllers will never know the difference. Factories themselves can come in many different implementations, but all of them share a basic idea—they take all the data needed to construct an object and actually create the object. Depending on how your objects need to be created and your application structure, your factories may look different, but all factories serve the same purpose. Our example above is an example of a factory that uses a static method. This is useful when your factory has no dependencies in and of itself, or the object the factory is creating can have all the relevant data passed in as part of the method call. Since our Response objects only need to know about the return code and the data to display, there's no need to ever really create a `ResponseFactory` object. Not all factories need to be based on static methods. If you have a factory that is used many times throughout your application and it has some configuration that never changes between calls, it may be helpful to pass in some of those dependencies via the constructor for the factory. For example, you may have a factory that creates database-backed services. They themselves all follow the repository pattern and have different configuration options, but they all require some connection back to your application's database. You may make a factory that looks like this: See Listing 5.

Listing 5.

```
1. class RepositoryFactory {
2.     public function __construct(protected \\PDO $pdo)
3.     { }
4.
5.     public function create(
6.         string $entityName,
7.         string $idColumn = 'id',
8.         ?string $tableName = null
9.     ) {
10.        if (is_null($tableName)) {
11.            $tableName = $entityName;
12.        }
13.
14.        if (class_exists($entityName . 'Repository')) {
15.            $classname = $entityName . 'Repository';
16.            return new $classname($this->pdo);
17.        }
18.
19.        $repo = new GenericRepository($this->pdo);
20.        $repo->setId($idColumn);
21.        $repo->setTableName($tableName);
22.        return $repo;
23.    }
24. }
```

Since the PDO dependency is shared amongst all the repository objects, we can pass it into the factory via the factory's constructor. When we make an individual repository via the `create()` method, we pass in all the variable information. If we couple this with a service locator, we reduce the number of objects we need to pass around by retrieving the factory from the service locator. Then we can pass in just the options we need for individual repositories, which in this case are the entity class they are tied to, an optional ID column name, and the name of the DB table if it differs from the entity. The factory nicely wraps all the logic around checking to see if we have a bespoke class already created or create a new generic repository object. We could use it in a controller similar to the following: See Listing 6 on the next page.

The Builder Pattern

Similar to the Factory pattern is the Builder pattern. They both do very much the same thing, which is create objects, but they differ in their construction. Where a factory generally has a single method to take all the information needed at once, a Builder tends to have a fluent interface where the developer sets various options on the new object.

The Builder pattern will come into play when you have objects that have many different options, and many of those options are not mandatory, and when many of the objects that you are instantiating are the same at a conceptual level (a `Car`, for example) but rarely have the same options.

While not strictly a requirement, the Builder pattern also tends to have a fluent interface where the developer can string multiple method calls together. You can create a Builder without a fluent interface, but the ability to chain together sets of function calls for options does make it a bit cleaner to group options that are always set and options that are only sometimes set.

Unlike Factories, Builders tend to be classes with many methods that collect data that end up creating the final product. This means you may see many "setter" methods on the Builder. A Builder is also very sensitive to state—where a Factory requires you to pass in the information each time. The Builder can keep the already entered information and use it to instantiate more objects.

Let's say we are allowing someone to write their own book. At the end of the process, we want to have a `Book` object that contains all the options we need to send information to the printer. We will need to know the size of the book itself, the type of cover, the title, the author, the cover image, and the book's contents. We can then use this object to create a `PrintOrder` for the actual printer.

First, we will declare some options that we can pass into the Builder. Thanks to PHP 8.1's new `enum` type, we can ensure that we strongly type against correct values. See Listing 7.

Listing 7.

```
enum BookCover: string {
    case HARDCOVER = 'hardcover';
    case PAPERBACK = 'paperback';
}

enum BookSize: string {
    case TRADE_5x8 = 't58';
    case TRADE_6x9 = 't69';
    case TRADE_8x10 = 't810';
    CASE MAGAZINE = 'mag';
}
```

Next, we'll look at the Builder itself. This class will allow us to set all the options that we need for the individual books. We know some basic information we need to capture and can set some sane defaults. See Listing 8.

Listing 6.

```
class OurController {
    protected RepositoryInstance $userRepo;

    public function __construct(RepositoryFactory $factory) {
        $this->userRepo = $factory->create(User::class);
    }

    public function indexAction() {
        $user = $this->userRepo->find(1);
        // Do something with the user
    }
}
```

As I mentioned before, with a Builder, we do not have to have a single method that takes the information about the object we need to instantiate, so we need to supplement our properties with ways to set those properties. The Builder will let the developer customize any portion of the `Book` object. See Listing 9 on the next page.

The last thing we need to do is have a way for us to instantiate the object we want. We will add a `build()` method to our builder that takes all the information we have collected so far and will instantiate and return a new `Book` object to us. See Listing 10 on the next page.

Now we can use the Builder to make books! This comes in handy when some options are the same between objects and only some of the options change. Let's create a few magazines using our builder. We will create a new builder and set the cover as a softcover and the size as a traditional magazine size. See Listing 11 on the next page.

No objects are created until we call the `build()` method. This allows us to set a few "static" options and then only alter the options we need to make the individual objects. In the case of the magazine, we simply update the title, contents, and cover image for each month. We call `build()` all the currently set options are used, including the cover and size we set initially. What happens if we want to create a more book-looking version of February 2022? Since that was the last set of options passed, we can just update the size and `build()` a new book.

```
$bookFeb = $magazineBuilder
    ->setSize(BookSize::TRADE_5x8)
    ->build();
```

Listing 8.

```
class BookBuilder {
protected string $author = 'Anonymous';
    protected string $contents;
    protected BookCover $cover = BookCover::PAPERBACK;
    protected string $coverImage = 'generic.png';
    protected BookSize $size = BookSize::TRADE_5x8;
    protected string $title;
}
```

Listing 9.

```
1. class BookBuilder {
2.   // Our properties from before
3.
4.   public function setAuthor(string $author): static {
5.     $this->author = $author;
6.     return $this;
7.   }
8.
9.   public function setContents(string $contents): static {
10.    $this->contents = $contents;
11.    return $this;
12.  }
13.
14.  public function setCover(BookCover $cover): static {
15.    $this->cover = $cover;
16.    return $this;
17.  }
18.
19.  public function setCoverImage(string $image): static {
20.    $this->coverImage = $image;
21.    return $this;
22.  }
23.
24.  public function setSize(BookSize $size): static {
25.    $this->size = $size;
26.    return $this;
27.  }
28.
29.  public function setTitle(string $title): static {
30.    $this->title = $title;
31.    return $this;
32.  }
33. }
```

Listing 10.

```
1. class BookBuilder {
2.     // Our properties
3.
4.     public function build(): Book {
5.         return new Book(
6.             title: $this->title,
7.             contents: $this->contents,
8.             author: $this->author,
9.             cover: $this->cover,
10.            coverImage: $this->coverImage,
11.            size: $this->size,
12.        );
13.    }
14.
15.    // Our setter methods
16. }
```

The important thing to remember with a Builder pattern is that the current state, or options, are preserved between `build()` calls. On the one hand, this makes it very easy to create many objects with slight differences between them very quickly without repeating yourself, but it can be easy to forget to update an option. See Listing 12.

Listing 11.

```
1. $magazineBuilder = (new BookBuilder())
2.     ->setCover(BookCover::PAPERBACK)
3.     ->setSize(BookSize::MAGAZINE);
4.
5. $jan2022 = $magazineBuilder
6.     ->setTitle('php[architect] January 2022')
7.     ->setCoverImage('jan2022.png')
8.     ->setContents($archJan2022)
9.     ->build();
10. $feb2022 = $magazineBuilder
11.     ->setTitle('php[architect] February 2022')
12.     ->setCoverImage('feb2022.png')
13.     ->setContents($archFeb2022)
14.     ->build();
```

Listing 12.

```
1. $magazineBuilder = (new BookBuilder())
2.     ->setCover(BookCover::PAPERBACK)
3.     ->setSize(BookSize::MAGAZINE);
4.
5. $jan2022 = $magazineBuilder
6.     ->setTitle('php[architect] January 2022')
7.     ->setCoverImage('jan2022.png')
8.     ->setContents($archJan2022)
9.     ->build();
10. $devLeadBook = $magazineBuilder
11.     ->setTitle('The Dev Lead Trenches')
12.     ->setContents($devTrenches)
13.     ->build();
14. $feb2022 = $magazineBuilder
15.     ->setTitle('php[architect] February 2022')
16.     ->setCoverImage('feb2022.png')
17.     ->setContents($archFeb2022)
18.     ->build();
```

Here we added a new book to the build process, but because one option was not updated, it will be printed with a flaw. Can you tell what it is? I did not reset the cover image, so this copy of "The Dev Lead Trenches" will be printed with the January 2022 cover for php[architect].

This is a simple mistake to make, but as with many technical decisions, developers make the power of the Builder pattern does come with this downside. That does not make it bad by any means, but the statefulness of the Builder is something to keep in mind.

If you are retrieving the Builder from a service locator, make sure that you set your service locator to always return a fresh instance. This is one of the easiest ways for the state to accidentally be kept between builder invocations without the developer noticing. How you do this depends on your service locator but keep this in mind.

Go Build Stuff

There are a few other patterns that help build objects as well. One is the Abstract Factory, which are factories that take in other factories to generate objects. If we wanted to be able to represent pages on our site in various ways, we might make a `PageFactory`, which has a `createDigitalView()` method and a `createPrintView()` method. We may make a `MobileDeviceViewFactory` that creates a mobile-specific view and printable PDF for a user, a `DesktopViewFactory` that does the same thing but for larger screens, and a `TerminalViewFactory` that creates raw text output as well as physically prints to an attached printer.

Another is the Prototype pattern. This is where a prototype object is created and can be copied and modified for use. In PHP, we can use the `clone` keyword to copy an existing object to a new copy and edit it. This is different than `$objectB = $objectA`, as PHP assigns objects to variables via reference by default. In this case, `$objectA` and `$objectB` are not copies but are the same object. `$objectB = clone $objectA` makes a discreet copy.

Singletons are also considered part of this "object creation" batch of patterns. It enforces one global instantiation of an object, but for the sake of what we are talking about, it does not help you "build new objects."

Take a look at your code and see where Factories or Builders may help out. If you see a pattern when it comes to creating objects, maybe one of these patterns will be a better fit. At least until the lion gets you.

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

Fractional Math

Oscar Merida

Recap

Build on the `Fraction` class, included in the code download as listings/fraction.php. Given two fractions, provide a way to add them together, subtract one from the other, multiply the two, and divide one by the other. For example, if you're given `2/3` and `9/16`, your code should perform the following calculations and output the results in its simplest form. Mixed fractions should be output when the numerator is greater than the denominator.

- `2/3 + 9/16`
- `2/3 - 9/16`
- `2/3 × 9/16`
- `2/3 ÷ 9/16`

Designing the Api

If we're working with instances of the `Fraction` class, let's consider how we'd want to approach one of the operations above. How would working with two fractions look from the outside? It could look like this:

```
$first = new \\Puzzles\\Fraction(2, 3);
$second = new \\Puzzles\\Fraction(9, 16);

$calc = new FractionalCalculator();
$sum = $calc->add($first, $second);
$difference = $calc->subtract($first, $second);
$product = $calc->multiply($first, $second);
$quotient = $calc->divide($first, $second);
```

That's a reasonable API.

Adding Fractions

To add two fractions, we need to know the lowest common multiple of our two denominators. Consider the case where the denominators are 4 and 6.

```
 1     5
--- + --- =  ?
 4     6
```

We can list out common multiples until we find one they share. In this case, we don't need that many, the third multiple of 4 is 12, and the second multiple of 6 is 12. So we can use that to rewrite our fractions with our common denominator.

```
4:  4   8  12  16  20  24
6:  6  12  18  24  30
```

Then we can sum the two together.

```
 1     5       3     10      13
--- + --- =  ---- + ---- =  ----
 4     6      12     12      12
```

What do we need to implement this? We need a function that returns the lowest common multiple (LCM) given two positive integers. Once we have the LCM, we can convert the numerators and add them together. Let's start with the LCM function. I made some assumptions and ran into a few edge cases to consider while writing the solution.

1. I assume denominators are always positive and non-zero.
2. If either argument is 1, then the LCM is the other argument.
3. To find the LCM, I realized that we need to check the first N multiples of our numbers, where N is the largest of the two denominators.

In the worst case, we effectively end up multiplying our two arguments together. But, if the LCM occurs earlier (like for 4 and 6), we can terminate as soon as we have one. It may not be the most efficient solution, but it works, and if it's slow, well, we can improve it later. The function is shown in Listing 2 on the next page.

We can make that a private method of our `FractionCalculator` class and then use it to add two fractions. See Listing 1.

Listing 1.

```
1. <?php namespace Puzzles;
2.
3. use \Puzzles\Fraction;
4.
5. class FractionCalculator
6. {
7.   public function add(
8.     Fraction $a,
9.     Fraction $b
10.  ): Fraction {
11.    $lcm = $this->getLCM($a->denominator, $b->denominator);
12.
13.    // convert the numerators
14.    $newA = $a->numerator * ($lcm / $a->denominator);
15.    $newB = $b->numerator * ($lcm / $b->denominator);
16.
17.    $sum = new Fraction($newA + $newB, $lcm);
18.    return $sum->simplify();
19.  }
20.  // * private method shown in Listing 2 */
21. }
```

Listing 2.

```php
<?php

function getLCM(int $a, int $b): int
{
  // we assume $a and $b are positive
  $a = abs($a);
  $b = abs($b);

  if ($a === 0 || $b === 0) {
    throw new \InvalidArgumentException("Do not use zero");
  }

  // skip the loop if we can
  if ($a === $b) {
    return $a;
  }

  $max = max($a, $b);

  // if one of the numbers is 1, the other is the LCM
  if ($a === 1 || $b === 1) {
    return $max;
  }

  $lcm = null;
  // worst case we have to try all the multiples between
  // 1 and the largest of ($a or $b) to find a common one.
  for ($i = 1; $i <= $max; $i++) {
    $multsA[] = $i * $a;
    $multsB[] = $i * $b;

    $common = array_intersect($multsA, $multsB);
    if ($common) {
      $lcm = array_shift($common);
      break;
    }
  }

  return $lcm;
}
```

We can now add two fractions like so:

```
$calc = new FractionCalculator();

$first = new Fraction(2, 3);
$second = new Fraction(9, 16);
$sum = $calc->add($first, $second);
echo $sum;
```

Subtracting Fractions

We did the bulk of the work for subtracting when we wrote the `add()` method. The one major change I had to make was to the `Fraction` class, to allow it to handle simplifying negative fractions. Otherwise, the `subtract()` method shown in Listing 3 only changes the operator.

Listing 3.

```php
public function subtract(Fraction $a, Fraction $b) : Fraction
{
    $lcm = $this->getLCM($a->denominator, $b->denominator);

    // convert the numerators
    $newA = $a->numerator * ($lcm / $a->denominator);
    $newB = $b->numerator * ($lcm / $b->denominator);

    $sum = new Fraction($newA - $newB, $lcm);
    return $sum->simplify();
}
```

Multiplying Fractions

We don't need to find any common factors or multiples to find the product of two fractions. We go straight to multiplying the numerators and denominators and then using `Fraction::simplify()` to return a simplified form.

```php
public function multiply(Fraction $a, Fraction $b):Fraction
{
    $numProduct = $a->numerator * $b->numerator;
    $numDenominator = $a->denominator * $b->denominator;

    $prod = new Fraction($numProduct, $numDenominator);
    return $prod->simplify();
}
```

Dividing Fractions

Dividing fractions is similar to multiplication, except we invert the second fraction. That is, instead of multiple numerators and denominators together. We multiply the first numerator by the second denominator and the second numerator by the first denominator as shown below.

```php
public function divide(Fraction $a, Fraction $b): Fraction
{
  $numProduct = $a->numerator * $b->denominator;
  $numDenominator = $a->denominator * $b->numerator;

  $quotient = new Fraction($numProduct, $numDenominator);
  return $quotient->simplify();
}
```

Pretty Printing

One thing I noticed was that we could improve how we output the fraction. The simplified form is fine for internal workings, but we can do better when we output a value. For example, if we divide the specified fractions, we get `32/27`.

```
$calc = new FractionCalculator();

$quot = $calc->divide($first, $second);
echo $quot; // outputs 32/27
```

We can tweak `Fraction::__toString()` to handle negative fractions and to output the whole and fractional parts to improve readability. It's certainly more code, but the output is markedly improved. See Listing 4.

Here's a sample of the improved output:

```
echo $first . " plus " . $second . " = " . $sum;
// Outputs "2/3 plus 9/16 = 1 and 11/48"
```

Listing 4.

```
1. public function __toString(): string
2. {
3.     if ($this->numerator === 0) {
4.         return '0';
5.     }
6.     $out = '';
7.     if ($this->numerator < 0) {
8.         $out = '-';
9.     }
10.
11.    $whole = intdiv($this->numerator, $this->denominator);
12.    if ($whole > 0) {
13.        $out .= $whole;
14.    }
15.
16.    $remainder = $this->numerator % $this->denominator;
17.    if ($remainder > 0) {
18.        if ($whole > 0) {
19.            $out .= ' and ';
20.        }
21.        $out .= $remainder . '/' . $this->denominator;
22.    }
23.
24.    return $out;
25. }
```

Converting Float Strings

Write a function that takes a currency string for dollars and converts it to the equivalent amount in pennies. The String may be formatted like "$100" or "$100.00". The dollar sign is optional, Confirm it works with all the following values "105.00", "$ 128.76", "$2,487.47", '11,135.65', and $71,135.71.

We're sticking with floating point values as I recently ran into this oddity in real-world code at my job.

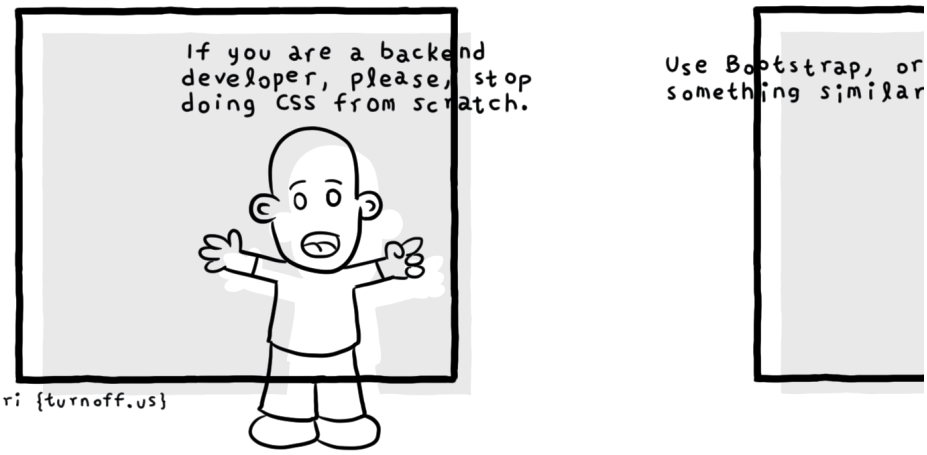
Some Guidelines And Tips

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[1] 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.*

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

1 *PsySH: https://psysh.org*

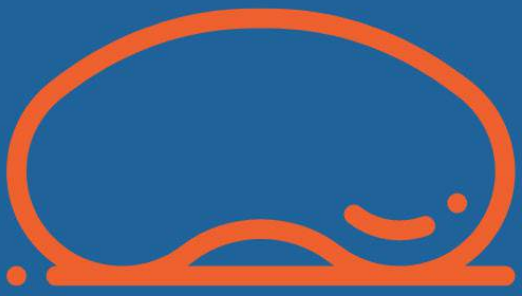


turnoff.us | Daniel Stori
Shared with permission from the artist

Call for Speakers

PHP[TEK]
2023

Share your amazing stories about Tech Leadership, PHP Development and Web Technologies. We'd love for you to apply to our Call for Speakers for the 15th annual php[tek] conference. Proposals are accepted through Jan 3rd and full travel support is available.



May 16-18, 2023

Sheraton Suites Chicago O'Hare

tek.phparch.com

Brought to you by the good people at

Surviving Cybersecurity

Eric Mann

Engineers don't often last as long in a cybersecurity focus as they do in other disciplines. If this is your path, you should understand why and how to beat the odds.

A common statistic I quote to folks asking about starting in cybersecurity[1] is their life expectancy in the role. This isn't a matter of their *actual* life in the role, but more a level setting for how long the average practitioner remains in the field.

Less than four years.

Nearly half of cybersecurity specialists will change careers or trajectories in two years or less. Almost two-thirds will have left the profession in under four. This is a field where individual contributors face a significantly high level of burnout and experience equally high turnover rates. If it's a field in which you're passionate, you should understand both why and how to survive.

Alert Fatigue

Security experts often see the worst part of our profession. It's not a field where we focus much on greenfield engineering or innovative architectures. Instead, we try to find a system's weak parts to ensure things are reliable, stable, and secure. We focus on protecting users from mistakes and establishing the "pit of success" such that innocent engineering oversights result in the correct behavior by default.

When we do our jobs well, it means people have to go out of their way to break something.

Unfortunately, that's exactly what happens.

Despite our best efforts and focus, we will make a mistake. We will miss a detail in an API that exposes a potential weakness in the system. An employee is sick during training and mistakes a phishing attack for a legitimate request. The phone rings at 2 am to alert us that a rogue actor has taken down or defaced a critical website.

Being constantly on alert can be utterly draining.

Fatalistic Forensics

If we're lucky, most alerts are false positives. An MFA[2] failure because someone mistyped their code rather than an attacker surveilling the system. A vulnerability detected during code review well in advance of a remote exploit. False positives are good things—they show the alerting system is working, help the team practice incident response, and strengthen any automation that might be triggered in the event of a real compromise.

There will come a day when an alert is a valid incident. In those situations, you and your team will be called upon to react—both in terms of immediate responses to end the threat and ongoing forensics to understand what went wrong and how to fix it.

Forensics can definitely be exciting. If you enjoy puzzles, finding a common thread between asynchronous requests across microservices can present an invigorating challenge. Actually, *proving* that something happened and being able to explain every stage of the kill chain[3] in an attack is thrilling.

Until you realize the ramifications of what you've found, in many cases, someone intentionally set out to do harm. They made a concerted effort to steal, break, violate, or otherwise compromise a system. There will be repercussions for their actions—likely legal. And your work will directly contribute to the consequences they face.

This is a hard-hitting one-two punch in this field. There's the immediate rush of concretely identifying a bad actor, followed by the sudden realization that your actions might send someone to jail (or perhaps through expensive litigation). It's enough to make anyone second guess themself.

I've also seen firsthand how this kind of scenario contributes directly to burnout and turnover on a team.

Staying Positive

Every Friday, I ask my team what plans they have for the weekend. What hobbies will they practice? What trips will they take? Who will they spend time with? It's critically important that security engineers be able to leave the job "at work," disconnect from the grind, and spend time enjoying life away from their regular security responsibilities. Taking time to focus and refresh yourself is vital to maintaining balance and bringing your best to the job, whether Monday during working hours or at 2 am when an alarm goes off.

It's also important to cultivate the right mindset about the work you do. We use terms like "threat hunter" that conjure images of tracking a ferocious beast of a vulnerability through the jungles of legacy code. Or perhaps chasing down a ne'er-do-well in a Fawkesian mask intent on harming your team. These images play well in film and are often the reason folks are attracted to this industry to begin with.

But tread carefully.

You are a defender. A protector. A positive influence on your team. Your job isn't to hunt down the bad guys—it's

1 *starting in cybersecurity:* https://phpa.me/security-corner-cybersecurity

2 *MFA:* https://phpa.me/security-corner-demystifying-mfa

3 *kill chain:* https://en.wikipedia.org/wiki/Kill_chain

to help craft a system resilient to their attacks. When you conduct forensics, it's to identify what broke and how to fix it—that is (and should be) your only objective. Focus on the positive impact your work has. Stop yourself from dwelling on any potential negative consequences the bad actor might face later because you did your job well.

The keys to success in this field are balance and positivity. Know when and how best to disconnect from work and focus on a well-balanced personal life. Understand where the line is between being a positive force on your team and dwelling on the negative—then keep as far from that line as you possibly can.

it's Worth It

The industry is evolving quickly. In the past, we had strict silos between development, operations, and security, making it hard to accomplish anything out of a rigid waterfall project management cycle. Newer evolutions in DevOps help link those two fields, empowering engineers to iterate and ship quickly while spreading the burden of operations over a diverse support staff.

An even newer paradigm—DevSecOps[4] - merges all three disciplines and makes security a shared responsibility across the entire team. This makes security *everyone's* problem and reduces the cognitive and emotional burden on the security team. It also encourages faster iterations on software with security well integrated into any changes to preempt and prevent potential breaches.

Knowing everyone is collaborating to build secure software makes the practice much more rewarding. It minimizes the things that can go wrong in the first place and promotes a culture of trust between the engineering and security teams.

Protecting your team and solving forensic puzzles is exciting, rewarding, and highly valued. If you can establish and maintain a positive attitude, your presence will have a huge, lasting impact on both the team and on your own job satisfaction. The work is hard, but it's a noble cause and always pays off in the long run.

4 *DevSecOps:* https://phpa.me/devsecops

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

Making Things Happen

Joe Ferguson

This month we're straying slightly from our usual PHP tooling to cover Make and how we can leverage it to simplify our lives as developers.

This month we're straying slightly from our usual PHP tooling to cover Make[1]#Makefile) and how we can leverage it to simplify our lives as developers. Make is an application common to Linux and macOS systems that can be used to build *just* about anything we need. A Makefile allows us to create commands to execute long complex operations such as managing dependencies, creating our test environment and executing test suites, and even deploying our application.

The idea for this article came from this interaction with Daniel Abernathy[2], where he asked for example Makefile uses. I'm always excited to share how I use tools, so I came up with a few examples off the cuff but felt like this topic was too good to stay buried in a thread on the bird site.

The `Makefile` should be in the project's root, checked into version control, and we should ensure users or CI/CD runner users have permissions to pull any private repositories or dependencies. We'll want to have SSH configured to deploy and push artifacts to remote registries. Configuring SSH is beyond the scope of this article; however, you can find a great introduction on GitHub.com[3]. Makefiles were originally intended to operate on files that existed in the project structure. Because we're using Make to run commands instead of build files, we'll use the `.PHONY` on all of our tasks so Make knows we're not compiling or generating files from source. We can add a `Makefile` to an existing PHP application and create our first *target,* which will run a command or set of controls for us: See Listing 1.

You might have noticed the comment, but Makefiles are expected to have Tab indention, not spaces! If you are struggling to figure out why targets are not working, this could be the cause. With our basic `Makefile` established, we can run `make setup` at our project root to run `install` commands for Composer and NPM and then run our NPM development command. The first line establishes a default target if the user runs make without any targets. For our case, we want to run `setup`. We've also created a `clean` command which cleans up our `vendor` and `node_modules` folders. We're able to leverage the (most likely) already existing make binary on your system to create explicit repeatable shortcuts to save us from having to type the entire command each time. See Listing 2.

We can add another target to help with everyday tasks such as resetting our local database. Traditionally we would want to empty our database, apply our migrations, and then run our database seeders. Whichever steps your application may need to get the development environment to a running state is what we should be building into our make target.

Listing 1.

```
.DEFAULT_GOAL := setup

.PHONY: setup
setup:
cp .env.example .env
    composer install # must be tab indent
    npm install
    npm run development
    mysql -u root -e "CREATE DATABASE IF NOT EXISTS app;"
    php artisan key:generate
    php artisan migrate

.PHONY: clean
clean:
    rm -rf vendor/
    rm -rf node_modules/
```

Listing 2.

```
$ make clean
rm -rf vendor/
rm -rf node_modules/

$ make setup
composer install
Installing dependencies from lock file (including require-dev)
Verifying lock file contents can be installed on current platform.
// ...
webpack compiled successfully
php artisan migrate
Nothing to migrate.
```

```
.PHONY: clean_db
clean_db:
    php artisan migrate:fresh
    php artisan db:seed
```

1 *Make:* https://en.wikipedia.org/wiki/Make_(software

2 *Daniel Abernathy:* https://twitter.com/dabernathy89

3 *GitHub.com:* https://phpa.me/SSHagent

Instead of having to type out the `artisan` commands each time we want to reset the database, we can use `make clean_db` to run the commands for us.

```
$ make clean_db
php artisan migrate:fresh
Dropped all tables successfully.
Migration table created successfully.
// ...
php artisan db:seed
Database seeding completed successfully.
```

We're making things happen! Moving right along, we want to create a target that runs our test suite. You can likely guess we're going to use `make test` to keep our syntax familiar with verbs because we're making something happen with the `make` command. In some large enough projects where context becomes important, you can also use a verb and state combination such as `setup_test_environment`, which may be used to configure the application to run in a specified environment.

Listing 3.

```
$ make test
php vendor/bin/phpunit
PHPUnit 9.5.20 #StandWithUkraine

..........................................  41 / 41 (100%)

Time: 00:01.827, Memory: 48.50 MB
OK (41 tests, 250 assertions)
```

The more complex actions being taken should justify more complex target names. See Listing 3.

Using `make test` will run PHPUnit to execute our application's test suite. Conveniently we've built everything needed to set up and test our application via our `Makefile`. We can now refactor our existing GitHub Actions steps to use our make targets. See Listing 4.

Listing 4.

```
1. name: PHP App
2. on: [push, pull_request]
3. jobs:
4.   build:
5.     runs-on: ubuntu-18.04
6.     strategy:
7.       fail-fast: false
8.       matrix:
9.         php: ['8.0']
10.    name: PHP ${{ matrix.php }}
11.    services:
12.      mysql:
13.        image: mysql:5.7
14.        env:
15.          MYSQL_ALLOW_EMPTY_PASSWORD: yes
16.          MYSQL_DATABASE: app
17.        ports:
18.          - 3306
19.        options: --health-cmd="mysqladmin ping" --health-interval=10s --health-timeout=5s --health-retries=3
20.
21.    steps:
22.      - name: Checkout Code
23.        uses: actions/checkout@v1
24.      - name: Setup PHP
25.        uses: shivammathur/setup-php@2.9.0
26.        with:
27.          php-version: ${{ matrix.php }}
28.          extensions: dom, curl, libxml, mbstring, zip, pcntl, ...
29.          coverage: none
30.      - name: Copy Env
31.        run: cp .env.example .env
32.      - name: Install dependencies & Setup App
33.        run: |
34.          composer install && php artisan key:generate --ansi && php artisan migrate --force && php artisan db:seed --force
35.        env:
36.            DB_PORT: ${{ job.services.mysql.ports[3306] }}
37.      - name: Execute tests
38.        run: vendor/bin/phpunit
39.        env:
40.          DB_PORT: ${{ job.services.mysql.ports[3306] }}
41.
```

Our GitHub action checks out our code, configures the application, and runs our test suite. We can simplify the steps by replacing them with our make targets. Our refactored action steps might look like this: See Listing 5.

Listing 5.

```
steps:
  - name: Checkout Code
    uses: actions/checkout@v1

  - name: Setup PHP
    uses: shivammathur/setup-php@2.9.0
    with:
      php-version: ${{ matrix.php }}

    extensions: dom, curl, libxml, mbstring, zip, pcntl, ...
      coverage: none

  - name: Install dependencies & Setup App
    run: make setup
    env:
      DB_PORT: ${{ job.services.mysql.ports[3306] }}

  - name: Execute tests
    run: make test
    env:
      DB_PORT: ${{ job.services.mysql.ports[3306] }}
```

With our action steps now utilizing our make targets, we can easily reproduce what happens in our Continuous Integration / Continuous Development (CI/CD) test runners. Even if you're not using GitHub actions, the same concept applies to any CI/CD platform such as GitLab, Jenkins, or TeamCity. The habit of leveraging make targets in CD/CD environments has saved what feels like a huge amount of time by not trying to use a tool to reproduce what the runner is doing. If the runner is only executing our make targets, debugging whatever goes wrong is much less of a headache. One way to measure the impact on your team would be to notice how much less often anyone struggles with not being able to reproduce what happens in CI/CD locally on their own machines.

A pattern emerges, allowing us to take snippets of shell script code and create make targets. Complex setup operations become `make setup`. Running test suites is done the same way in CI/CD as it is local development using `make test`, and developers no longer struggle with being unable to reliably reproduce CI/CD failures in their local environments. The pattern of make and verb allows us to push complex tasks behind easy-to-remember, boring commands. If you swap between multiple projects frequently, there is a lot of comfort knowing you built all the special things for each project in its Makefile and don't have to remember if this project is using NPM, Yarn, Vue, or React. We just need to refer to our `Makefile` in the project root to quickly understand what the individual project's configuration requires. Still not convinced about Make? Think of targets as command line aliases or shortcuts for complex operations. Laravel users often alias `artisan` to `php artisan` to skip typing `p h p <SPACE> a <TAB>`; the alias allows us to type `ar<TAB>` to autocomplete `artisan`. These types of small shortcuts on the command line are similar to our everyday project tasks and can be created as make targets.

Another use case for moving commands to Makefile targets is for deploying our application. Like most traditional PHP applications, the deployment process involves:

- pulling down the latest changes from source control
- installing/updating dependencies
- applying any database migrations
- running any front end or other setup commands
- any other tasks that may need to be completed

For many Laravel applications, I enjoy using the Envoy package[4] as my deployment method. The power of Envoy is the ability to run shell commands from a Blade template on remote Linux systems. We can see an example `Envoy.blade.php` example below where we specify a server and a task to run on the specified server. See Listing 6.

Listing 6.

```
@servers(['prod' => 'phparch@phparch.com'])

@task('deploy', ['on' => 'prod'])
cd /home/phparch/awesome-php
git pull origin erics-best-branch
/usr/bin/php8.1 /usr/bin/composer install --no-dev
/usr/bin/php8.1 artisan migrate --force
@endtask
```

To deploy our application, we would run `php vendor/bin/envoy run deploy` to trigger Envoy to deploy our application. If you're already guessing what our next Makefile target is, you're absolutely correct; we're going to add `make deploy`.

```
.PHONY: deploy
deploy:
    php vendor/bin/envoy run deploy
```

With our `make deploy` target created, we can update our GitHub Actions file and replace our envoy run command. See Listing 7.

We have now eliminated all of the raw commands running in our CI/CD pipeline with Makefile targets allowing us to reproduce behaviors in our action runners locally on our own machines without extra steps or hassle. A successful CI/CD workflow would be to run `make setup`, `make test`, and `make deploy` to push our changes to production. Remember, we have to have configured SSH on the remote server as well as GitHub secrets, so we keep the private deployment secret. Always put sensitive values behind GitHub secrets to avoid

4 *Envoy package:* https://laravel.com/docs/envoy

accidentally leaking API keys or passwords outside of your organization.

The last example we're going to cover this month is building Docker images. Even veteran Docker building experts can agree that simplifying build commands is a great value for any sized team. We're able to consolidate long, expressive commands and arguments into easy to follow make targets which we can also leverage in our CI/CD pipelines. If you're just getting started building containers, you should check out The Workshop: Just Use Docker[5] to get a better understanding of what's happening with Docker. For our examples, we'll assume you have a basic working `Dockerfile` capable of building an image.

We can utilize environment variables in our Makefiles to use for tagging artifacts such as our container tag. We can also apply dynamic labels to our container images to contain the git hash of the commit that the repository was on when the container was built. We use `HASH := $(shell git rev-parse HEAD)` to save the hash in the `HASH` variable and apply the label in our build command via a build argument `--build-arg HASH=$(HASH)`. Later we'll be able to `docker inspect...` our image name, and we'll see `HASH` among the labels. Our build command starts with `DOCKER_BUILDKIT=1 docker build --build-arg HASH=$(HASH)` to instruct Docker to use the newer buildkit features when building the container as well as pass our hash build argument. We'll add on to this command in our `docker build` target to construct the full build command: See Listing 8.

You can see we're creating variables so we can easily reuse our target with other projects. We specify the repository, and container, fetch the hash from git, then check for `GITHUB_RUN`. GITHUB_RUN is an environment variable that contains a unique run in our CI/CD environment that doesn't exist outside of GitHub Actions. To accommodate the missing value, we add it here and set it to `latest`, which is a safe default for building images on our local machines. Because we don't use `latest` in production, we only run specifically tagged images that have passed the building and test processes via GitHub actions. We've greatly simplified building and pushing our tagged container image to our remote registry, so our project is just about complete when it comes to Makefile targets. We can set up our application, run tests, deploy, and build container images simply by running `make TARGET` on our command line.

Make is one of those tools that more developers should learn how to use to increase the developer experience of repetitive and boring everyday tasks. It can increase your productivity by making complex commands simple and boring, allowing you to get back to solving real problems. Make can solve a lot of problems and especially help new project contributors get their environment up and running without having to be an expert in the local development practices.

Listing 7.

```
- name: Install SSH key
  uses: shimataro/ssh-key-action@v2
  with:
    key: ${{ secrets.SSH_KEY }}
    known_hosts: ${{ secrets.KNOWN_HOSTS }}
  if: github.ref == 'refs/heads/master'

- name: Deploy
  run: make deploy
  if: github.ref == 'refs/heads/master'
```

Listing 8.

```
REGISTRY := registry.phparch.com
CONTAINER := phparch/awesome-php
HASH := $(shell git rev-parse HEAD)
BUILD = DOCKER_BUILDKIT=1 docker build --build-arg HASH=$(HASH)

ifndef GITHUB_RUN
GITHUB_RUN=latest
endif

.PHONY: docker_build
docker_build:
    $(BUILD) --label=GIT_HASH=$(GIT_HASH) \\
    -t $(REGISTRY)/$(REPOSITORY):$(GITHUB_RUN) .

.PHONY: docker_push
docker_push:
    docker push $(REGISTRY)/$(REPOSITORY):$(GITHUB_RUN)
```

Related Reading

- *The Workshop: Easy CLI PHP with Symfony Console 5* by Joe Ferguson, April 2020. https://phpa.me/workshop-apr-2020
- *The Workshop: Git Hooks with CaptainHook* by Joe Ferguson, December 2020. https://phpa.me/2020-12-workshop
- *Jenkins Automation* by Toni Van de Voorde, January 2019. https://phparch.com/article/jenkins-automation/

Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They live in the Memphis suburbs with their two cats. @JoePFerguson

5 *The Workshop: Just Use Docker:* https://phpa.me/just-use-docker

PSR-6 Caching Interface

Frank Wallen

Caching is very important, so let's take a look at some of the issues that can be associated with caching and practices to help better optimize performance.

Caching is a crucial component of the modern web for a number of reasons, but mostly for performance. It allows delivering repeatedly requested data or pages almost immediately, which is important for user satisfaction when they do not need to wait those few extra seconds to receive a response (which feels like a long time when staring at the browser). The other side is the reduction of stress and load on servers to re-process and render a response when all that is needed is to pull the response from the cache and return it. Imagine thousands or hundreds of thousands of visitors or APIs requesting the team stats of their favorite sport, financial data, or a master report. The code and the server can run those queries on a database or generate datasets, then compose the data and present it as requested. However, that could trigger load balancing servers to spin up, increase costs, or bottle-neck the database. Returning cached results on those repeated requests will result in savings, satisfied visitors, and faster API calls.

While 'front-end' caching handled by a service like a reverse proxy or content delivery network can be very effective, they don't really help with retrieving and presenting large datasets. Caching within the design and implementation of the application is crucial for scaling and should be taken into consideration as part of the architecture. Data can be cached in many different ways such as a file or in memory (like an array, maybe the session, or in the early days of PHP, in a `static` variable or property). In-memory caches are fast and made easier with a service like `redis`, a key-value store, and often have built-in support for time-based expiration of the data.

Before PSR-6[1], frameworks and libraries were implementing their own caching, an excellent decision to be sure. However, it led to a familiar situation where the implementation was tightly coupled to the code and could not be easily shared or integrated with a library. Developers of caching libraries found themselves supporting only a few frameworks or having to create many adapters specific to other frameworks and libraries. Using a common interface saves developers a lot of time and effort when implementing their own caching solution or integrating a caching library into the application. PSR-6 defines item interfaces for a caching system's components rather than interfaces for the entire mechanism, allowing frameworks and libraries to apply their own architectural strategies. What is important is how we understand and handle the data being cached and retrieved.

Let's review some important caching concepts from PSR-6 definitions[2]:

- **Time To Live (TTL)**— How long, usually in seconds, before the stored data is considered stale.
- **Key** — A unique identifier that represents the stored data.
- **Hit** — A hit means that the key and data are found in storage, and the data is not expired or stale and is valid.
- **Miss** — A miss means that the key was not found, or if it was, the data is expired, stale, or invalid.

What is critical to caching is expecting to get the exact data retrieved from the cache pool that was originally stored. Therefore we use any serializable PHP data type:

- **Strings** — Arbitrary lengths in PHP-compatible encoding.
- **Integers** — Up to 64-bit signed.
- **Floats** — Signed floating point values.
- **Boolean** — True or False.
- **Null** — Actual null value.
- **Object** — Must be serializable, deserializable, and lossless. Meaning that this evaluates to true: `$object == unserialize(serialize($object))`.

Implementing libraries MUST return values exactly as passed, including variable type. Integers and floats cannot be returned as strings, and booleans cannot be returned as `1` or `0`. If the value cannot be returned exactly, the library MUST respond with a *miss*. The data must be considered immutable once passed to the caching system. Imagine expecting to retrieve an object with methods and receiving a `std object`.

It is also important that a caching system error should not result in an application error; it should be trapping the error and stop bubbling. However, the error should be logged, of course, so that any bugs can be fixed. When clearing data from the pool, if it's multiple items or one, and the key does not exist, it MUST NOT be considered an error condition. It is also not an error condition if a key does not exist or the pool is empty.

PHP-FIG maintains a repo[3] for the PSR-6 interfaces on github. While there are two exceptions defined, we'll be looking at the two interfaces: `CacheItemInterface` and `CacheItemPoolInterface`.

1 *PSR-6:* https://www.php-fig.org/psr/psr-6/

2 *PSR-6 definitions:* https://phpa.me/caching-php-fig

3 *repo:* https://github.com/php-fig/cache

`CacheItemPoolInterface` follows a datamapper (or repository) design pattern centered around a data model that implements `CacheItemInterface`. It is the responsibility of the `Pool` to provide a `CacheItem`, either returning an existing one or creating a new one. The implementing system's responsibility is to provide a unique key that it recognizes to be associated with the `CacheItem`. The `Pool::getItem` method should, in the event of no existing item, create a new `CacheItem` with the provided key. Valid keys are up to 64 characters in length, UTF-8 encoded, and comprised of the following characters: `0-9`, `a-z`, `A-Z`, `_`, and `.`. It is allowed to use additional encodings, longer lengths, and additional characters, but the minimum requirements must be supported. Also, the following characters are reserved and cannot be used: `{}()/\@`.

Listing 1.

```
1. $pool = getCachePool('my_caching_pool');
2. $item = $pool->getItem('PHPArch_is_awesome_1');
3.
4. if ( ! $item->isHit() ) {
5.     $item->set( generateDataArray() );
6.     $pool->save($item);
7. }
8.
9. return $item;
```

Here's a simple example of getting an item from a cache pool:

`$pool` will contain our caching pool implementing `CacheItemPoolInterface`. The pool looks for the key in storage and returns a `CacheItem`. We check if `$item->isHit()` is `false`, meaning there is no value, or it's expired, and populate the value if it is. Our `generateDataArray` function has the potential to take as long as 10 seconds to generate the array. The first missed hit means we wait until we get the array and cache it, but every subsequent hit is an immediately returned value. If 1000 visitors are requesting the same data, you have saved them all at least 9 seconds of wait time, and your data source can sit coolly waiting for other heavy requests without bottlenecking.

A challenging aspect of caching is expiration; how long do we hold on to the data? Obviously, for 'hot' items that change often, that expiration should be low to help ease bottle-necking of the data source or avoid spinning up a load balancer. Another strategy is expiring the cached item when supporting data has been changed, such as new team stats have become available. When that data has changed, an event could be issued that will request the pool to clear the cached data or retrieve it and update it. Expiration strategies will be dependent on the application architecture and usage.

Conclusion

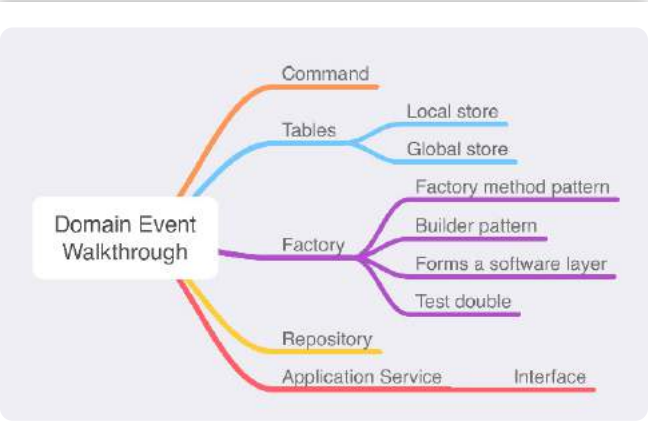
When applications begin scaling, as user count or api usage increases, caching will be critical to performance, visitor satisfaction, and cost. It's worth the investment of resources (time, effort, money) to prepare code or the application to use a caching mechanism. If the reader needed convincing to use a cache, I hope this column provided that. Otherwise, I hope that I have offered some understanding of how a caching system works.

Frank Wallen is a PHP developer, musician, and tabletop gaming geek (really a gamer geek in general). He is also the proud father of two amazing young men and grandfather to two beautiful children who light up his life. He is from Southern and Baja California and dreams of having his own Rancherito where he can live with his family, have a dog, and, of course, a cat. @frank_wallen

Domain Event Walkthrough

Edward Barnard

The July 2022 article "Structure by Use Case" introduced the Bounded Context pattern, which we'll be using over and over as we structure our software by use case. This month we'll continue exploring these concepts as we implement the global Domain Event store.

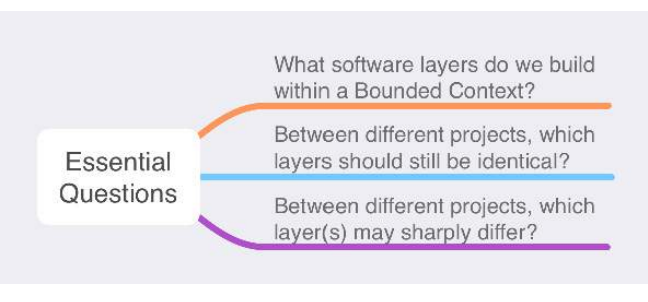


All source code is available on GitHub at ewbarnard/strategic-ddd[1].

Essential Questions

Upon surviving this article, you should be able to answer (see Figure 2:

- What are the software layers we build within a Bounded Context?
- Which layers should be essentially identical when implementing a use case in both a new "greenfield" project and a legacy codebase?
- When implementing a use case in different framework projects (e.g., legacy and greenfield), which layer(s) may sharply differ?



Domain Event Command

Let's walk through some code. We're starting at the top of the stack, so to speak, with the command-line handler. We'll then follow wherever it leads.

Install Ben Ramsey's uuid package[2]:

```
composer require ramsey/uuid
```

Listing 1 shows `public function execute()`, which was generated by CakePHP's `bake` tool. This method is temporary, just providing a way to exercise the code as we develop it.

The steps are:

- Load (i.e., instantiate) the framework table model for table `domain_events`.
- Use the Domain Event factory to create the Domain Event application service.
- Create a keyed array that matches the table schema. See Listing 2 for the table schema. The array keys come from the `local_app_events` table because we are storing one of its table rows in the global `domain_events` table.
- Create a CakePHP table row entity suitable for inserting into the `domain_events` table.
- Set the copied primary key to a random value because we are merely simulating the `local_app_events` row.
- Call `notifyDomainEvent()`. We are exercising the code being developed.
- Write a completion message to the console.

Listing 2 shows the destination table.

Listing 3 shows the originating table.

What do we already know about the Domain Event factory? We know that calling the static method `domainEvent()` produces an application service. Bear that in mind as we look at the factory source code next. I can't count the number of times I've mis-navigated or clicked on the wrong editor tab without realizing I'm then staring at the wrong file! I formed the habit of keeping a mental image of what I expect to see so that I know I'm in the right place when I get there.

1 *ewbarnard/strategic-ddd:* https://github.com/ewbarnard/strategic-ddd

2 *uuid package:* https://github.com/ramsey/uuid

Domain Event Factory

Listing 4 shows that the Domain Event factory works exactly as we'd expect. It instantiates the Domain Event repository and passes the repository into the Domain Event application service's constructor. Note that the return type is the *interface* rather than the service. We don't know why as yet, but it's worth noting that this is so.

The Domain Event factory is small. Is it even worth bothering with something so trivial? Possibly not; however, our initial purpose is to show the pattern without any extra "fluff."

Generally speaking, it's probably worth creating the factory when you're using this same Bounded Context pattern over and over. People new to the project will quickly be able to see how objects are composed, which in turn shows how and where boundaries are crossed.

A given feature will normally encompass several use cases or user stories. They're normally built one at a time over the course of a sprint or several sprints. The use cases for that feature will go into the same Application Services folder as separate classes, but their construction can go into the same Factory class. The factory methods might each remain trivial, but if there are several such methods, it's worth keeping them all in one class.

What is a Factory? Since we saw the implementation first, we know exactly what our Factory is doing. Wikipedia defines a factory[3] as "an object for creating other objects," which does match what we're doing here. However, Wikipedia continues its definition:

> *…formally a factory is a function or method that returns objects of varying prototype or class from some method call, which is assumed to be "new."*

Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the "Gang of Four") introduces the "Factory Method" pattern (page 107):

> *Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*

The Factory Method pattern is definitely not what we're doing here. *Design Patterns* also explains the "Builder" pattern (page 97):

> *Separate the construction of a complex object from its representation so that the same construction process can create different representations.*
>
> *Use the Builder pattern when the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled, [and/or when] the construction process must allow different representations for the object that's constructed.*

Listing 1.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Command;
6.
7. //use statements removed for brevity
8. //see code download for full version
9.
10. final class DomainEventCommand extends Command
11. {
12.   use GlobalEventsTrait;
13.
14.   /**
15.    * @throws \Exception
16.    */
17.   public function execute(Arguments $args, ConsoleIo $io): ?int
18.   {
19.     $this->loadDomainEventsTable();
20.     $service = DomainEventFactory::domainEvent();
21.     $data = [
22.       LocalAppEvent::FIELD_ACTION => 'Test',
23.       LocalAppEvent::FIELD_SUBSYSTEM => 'Command Line',
24.       LocalAppEvent::FIELD_DESCRIPTION =>
25.                   sprintf('%.6f', microtime(true)),
26.       LocalAppEvent::FIELD_DETAIL => null,
27.       LocalAppEvent::FIELD_EVENT_UUID => Uuid::uuid4()->toString(),
28.       LocalAppEvent::FIELD_WHEN_OCCURRED => FrozenTime::now(),
29.       LocalAppEvent::FIELD_CREATED => FrozenTime::now(),
30.       LocalAppEvent::FIELD_MODIFIED => FrozenTime::now(),
31.     ];
32.     $event = $this->domainEventsTable->newEntity($data)->toArray();
33.     $event[LocalAppEvent::FIELD_ID] = random_int(1, 999999999);
34.     $service->notifyDomainEvent('command line', $event);
35.     $io->out('Domain event complete.');
36.
37.     return 0;
38.   }
39. }
```

I mention all this because we're not really using our Factory for its intended purpose. The Factory pattern is designed to allow different internals and different implementations, depending on the circumstance. It's quite handy within unit tests for swapping in a test double or other test fixture.

3 *factory:* https://phpa.me/wikip_oop

Our purpose is to create a boundary. That's why, in the "Independent Evolution" section of August 2022 "DDD Alley," we called "Factory" a layer, just like Application Services, Domain Model, and Repository.

Design Patterns explains (page 100):

> *It isolates code for construction and representation. The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes that define the product's internal structure; such classes don't appear in Builder's interface.*

What do we know about a layer and its boundary? The Factory, as a software layer within our Bounded Context, has the same characteristics as the other layers we've been discussing. In other words, the Factory can evolve independently of whatever is using its product (the Application Service in this case). We also know that the rest of the world can evolve independently of the Factory.

Does this really make a difference in the case of our two-line factory method? It does! Here's an example showing why. Our Application Service does not use the database. On the other hand, our Repository does. We've gone to great pains to make this careful separation.

Listing 2.

```
CREATE TABLE `domain_events`
(
    `id`            bigint unsigned NOT NULL AUTO_INCREMENT,
    `id_of_source`  bigint unsigned NOT NULL,
    `source_table`  varchar(255)    NOT NULL,
    `action`        varchar(255)    NOT NULL DEFAULT '',
    `subsystem`     varchar(255)    NOT NULL DEFAULT '',
    `description`   varchar(255)    NOT NULL DEFAULT '',
    `detail`        json                    DEFAULT NULL,
    `event_uuid`    char(36)        NOT NULL,
    `when_occurred` timestamp(6)    NOT NULL,
    `created`       datetime        NOT NULL,
    `modified`      datetime        NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `event_uuid` (`event_uuid`),
    UNIQUE KEY `source_id` (`id_of_source`, `source_table`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4
  COLLATE = utf8mb4_0900_ai_ci;
```

Listing 3.

```
CREATE TABLE `local_app_events`
(
    `id`            bigint unsigned NOT NULL AUTO_INCREMENT,
    `action`        varchar(255)    NOT NULL DEFAULT '',
    `subsystem`     varchar(255)    NOT NULL DEFAULT '',
    `description`   varchar(255)    NOT NULL DEFAULT '',
    `detail`        json                    DEFAULT NULL,
    `event_uuid`    char(36)        NOT NULL,
    `when_occurred` timestamp(6)    NOT NULL,
    `created`       datetime        NOT NULL,
    `modified`      datetime        NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `event_uuid` (`event_uuid`)
) ENGINE = InnoDB
  DEFAULT CHARSET = utf8mb4
  COLLATE = utf8mb4_0900_ai_ci;
```

Listing 4.

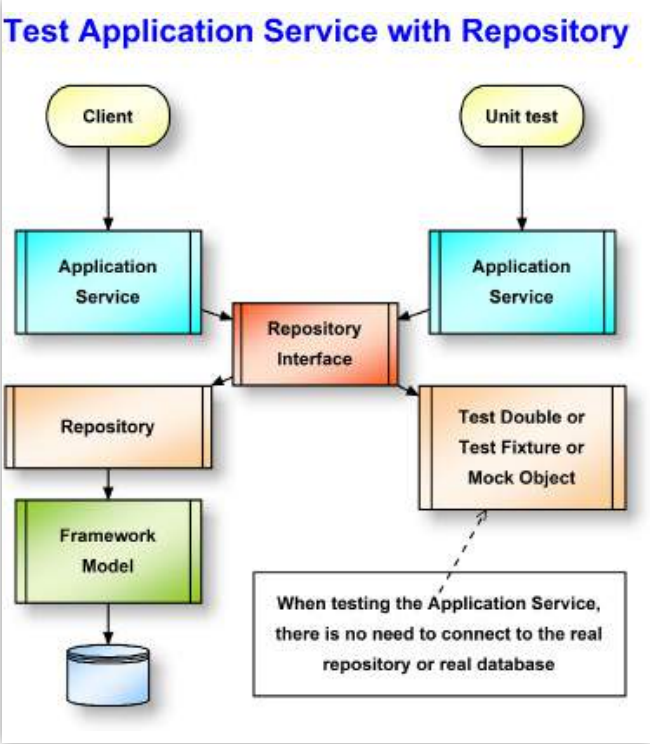
```
<?php

declare(strict_types=1);

namespace App\BoundedContexts\Infrastructure\Events\DomainEvent\Factory;

use App\BoundedContexts\Infrastructure\Events\DomainEvent\ApplicationServices\DomainEvent;
use App\BoundedContexts\Infrastructure\Events\DomainEvent\DomainModel\Interfaces\IDomainEvent;
use App\BoundedContexts\Infrastructure\Events\DomainEvent\Repository\RDomainEvent;

final class DomainEventFactory
{
    private function __construct()
    {
    }

    public static function domainEvent(): IDomainEvent
    {
        $repository = new RDomainEvent();
        return new DomainEvent($repository);
    }
}
```

It's relatively difficult to test an application service that's connecting to a database. However, it's dead simple if we replace the repository with a test fixture. Remember that, at that point, we're focused on testing the application service and *not* on testing the repository. See Figure 3.

Domain Event Repository

Listing 5 shows the Domain Event repository. It's the one piece of code that is CakePHP-specific. The repository, in theory, should be the *only* place that's sensitive to framework changes. In actual practice, I take advantage of various utility classes such as validating or truncating text, foreign language support, and so on.

A repository should be small and fine-grained. Chances are it will start with a single method, like this. The CakePHP `saveOrFail()` method throws an exception if anything went wrong with the row insert. Since there's no try/catch here, we should annotate the method `saveDomainEvent()` to warn that this method throws an exception.



Domain Event Service

You will recall that the factory passes the repository object to our application service, see Listing 6, where we save it in the constructor. You will also recall that the Domain Event command calls `notifyDomainEvent()` with a string and an array. And finally, we know that the repository's `saveDomainEvent()` command saves a row to the global Domain Event store.

This application service, as shown, has two problems. CakePHP's `debug()` function does nothing in production, so it's safe to leave it there, but we would want to delete that line before sending this feature to production. We also know that the repository can throw an exception. Since we forgot to annotate that fact in the repository, it's perhaps no surprise that we forgot to handle the exception here.

We definitely need to make a decision on failures before we reach production. That's actually part of our "random and rare failures" article, so this feature is correct for now. We should include a `TODO` or `FIXME` or `@THROWS` annotation to be sure we don't forget, though!

Finally, note that this application service does implement an interface. We'll walk through the interface next.

Domain Event Interface

Listing 7 shows the interface declaring the single method. Everything we've declared here, we've seen before.

The one item of note is the annoyingly-long `namespace` declaration line. I find it's quite useful to create deeply-nested folder structures as we implement our Bounded Context pattern. When a folder contains only one or two or a very few closely-connected items, it's easy to move an entire folder when refactoring. (For PhpStorm, on the help page, Move Refactorings[4] scroll down to "Move a PHP namespace" for instructions.)

On the other hand, if you have a "flat" folder containing many classes and only want to move a few of them, that's more difficult. Generally speaking, the moment I get three or more files in the same folder, I consider whether I should separate them into subfolders before proceeding further. It's easy to move three classes, even if it means copy/pasting and manually editing the namespace and all references, but doing that with twenty classes gets tedious. Thus my preference is for deep-and-narrow folder structures rather than shallow-and-wide.

Legacy Domain Event Walkthrough

This walkthrough will be similar to "Legacy Count Events" in July 2022 "Structure by Use Case."[5] I implemented the legacy version using the listings we just walked through, but in reverse order:

- Domain Event Application Service Interface
- Domain Event Repository
- Domain Event Application Service
- Domain Event Factory
- Test harness

Listing 8 shows the application service interface. Up to this point, we have mostly been talking about repository interfaces. This interface is not for the repository but for the application service. With so many class files having similar

4 *Move Refactorings:* https://phpa.me/phpstorm-refactoring

5 *July 2022 "Structure by Use Case.":* https://phpa.me/2022-08-ddd

Listing 5.

```php
<?php

declare(strict_types=1);

namespace App\BoundedContexts\Infrastructure\Events\DomainEvent\Repository;

use App\BoundedContexts\Infrastructure\LoadTableModels\PrimaryDatabase\Events\GlobalEventsTrait;
use App\Model\Entity\DomainEvent;

final class RDomainEvent
{
    use GlobalEventsTrait;

    public function __construct()
    {
        $this->loadModels();
    }

    private function loadModels(): void
    {
        $this->loadDomainEventsTable();
    }

    public function saveDomainEvent(string $sourceTable, array $localEvent): void
    {
        $localEvent[DomainEvent::FIELD_ID_OF_SOURCE] = $localEvent[DomainEvent::FIELD_ID];
        unset($localEvent[DomainEvent::FIELD_ID]);
        $localEvent[DomainEvent::FIELD_SOURCE_TABLE] = $sourceTable;
        $entity = $this->domainEventsTable->newEntity($localEvent);
        $entity->setDirty(DomainEvent::FIELD_WHEN_OCCURRED, true);
        $entity->setDirty(DomainEvent::FIELD_CREATED, true);
        $entity->setDirty(DomainEvent::FIELD_MODIFIED, true);
        $this->domainEventsTable->saveOrFail($entity);
    }
}
```

Listing 6.

```php
<?php

declare(strict_types=1);

namespace App\BoundedContexts\Infrastructure\Events\DomainEvent\ApplicationServices;

use App\BoundedContexts\Infrastructure\Events\DomainEvent\DomainModel\Interfaces\IDomainEvent;
use App\BoundedContexts\Infrastructure\Events\DomainEvent\Repository\RDomainEvent;

final class DomainEvent implements IDomainEvent
{
    private RDomainEvent $repository;

    public function __construct(RDomainEvent $repository)
    {
        $this->repository = $repository;
    }

    public function notifyDomainEvent(string $sourceTable, array $localEvent): void
    {
        debug(compact('sourceTable', 'localEvent'));
        $this->repository->saveDomainEvent($sourceTable, $localEvent);
    }
}
```

Listing 7.

```php
<?php

declare(strict_types=1);

namespace App\...\DomainModel\Interfaces;

interface IDomainEvent
{
  public function notifyDomainEvent(
    string $sourceTable,
    array $localEvent
  ): void;
}
```

names, it might be worth a comment at the top of the interface to clarify that point.

Listing 9 shows the repository. It uses Doctrine's Database Abstraction Layer. Besides the test harness, this will be the one class that differs from the above example.

Listing 10 shows the application service. It's essentially identical to the example above.

Listing 11 shows the factory. It, too, is essentially identical to the example above.

Listing 12 the test harness, is a PHP script sitting in the `test` directory.

Below shows the test harness output. It's concise. The actual output is the row inserted into the `domain_events` table (not shown).

```
$> php  test/exercise_domain_event.php
Domain event complete
```

Listing 8.

```php
<?php

declare(strict_types=1);

namespace LegacyBoundedContexts\...\DomainModel\Interfaces;

interface IDomainEvent
{
   public function notifyDomainEvent(
     string $sourceTable,
     array $localEvent
   ): void;
}
```

Listing 9.

```php
<?php

declare(strict_types=1);

namespace LegacyBoundedContexts\...\Repository;

use Doctrine\DBAL\DBALException;
use Models\Common\BaseModel;

final class RDomainEvent extends BaseModel
{
  public function saveDomainEvent
        (string $sourceTable, array $localEvent): void
  {
      $sql = 'insert into domain_events
(id_of_source, source_table, action, subsystem, description,
 detail, event_uuid, when_occurred, created, modified)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)';
      $parms = [
        $localEvent['id'],
        $sourceTable,
        $localEvent['action'],
        $localEvent['subsystem'],
        $localEvent['description'],
        $localEvent['detail'],
        $localEvent['event_uuid'],
        $localEvent['when_occurred'],
        $localEvent['created'],
        $localEvent['modified'],
      ];
      try {
        $this->sql->executeUpdate($sql, $parms);
      } catch (DBALException $e) {
        // Fail silently for now
      }
  }
}
```

Essential Questions Answered

- *What are the software layers we build within a Bounded Context?* Application Service, Repository, Domain Model, Factory.
- *Which layers should be essentially identical when implementing a use case in both a new "greenfield" project and a legacy codebase?* Application Service and Domain Model. The factory could be similar or different depending on the frameworks, auto-wiring, etc.

Listing 10.

```php
<?php

declare(strict_types=1);

namespace LegacyBoundedContexts\...\ApplicationServices;

use LegacyBoundedContexts\...\IDomainEvent;
use LegacyBoundedContexts\...\RDomainEvent;

final class DomainEvent implements IDomainEvent
{
  /** @var RDomainEvent */
  private $repository;

  public function __construct(RDomainEvent $repository)
  {
    $this->repository = $repository;
  }

  public function notifyDomainEvent
      (string $sourceTable, array $localEvent): void
  {
    $this->repository->saveDomainEvent($sourceTable, $localEvent);
  }
}
```

Listing 11.

```php
<?php

declare(strict_types=1);

namespace LegacyBoundedContexts\...\Factory;

use LegacyBoundedContexts\...\ApplicationServices\DomainEvent;
use LegacyBoundedContexts\...\Interfaces\IDomainEvent;
use LegacyBoundedContexts\...\Repository\RDomainEvent;

final class DomainEventFactory
{
    private function __construct()
    {
    }

    public static function domainEvent(): IDomainEvent
    {
        $repository = new RDomainEvent();
        return new DomainEvent($repository);
    }
}
```

Listing 12.

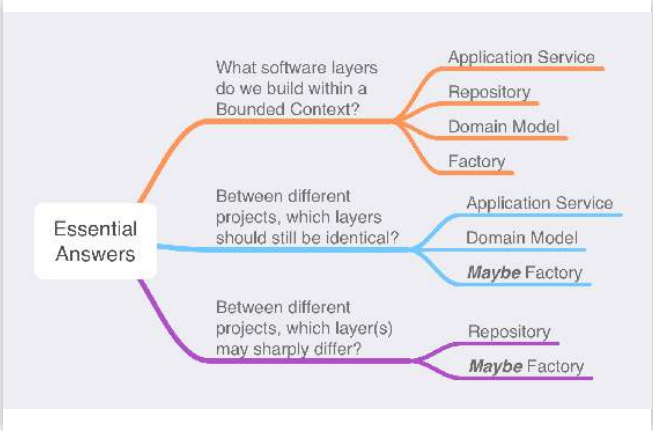
```php
<?php

use LegacyBoundedContexts\...\Factory\DomainEventFactory;
use Ramsey\Uuid\Uuid;

require_once(__DIR__ . '/bootstrap.php');

$service = DomainEventFactory::domainEvent();
try {
  $event = [
    'id' => random_int(1, 99999999),
    'action' => 'Legacy Test',
    'subsystem' => 'Command Line',
    'description' => sprintf('%.6f', microtime(true)),
    'detail' => null,
    'event_uuid' => Uuid::uuid4()->toString(),
    'when_occurred' => date('Y-m-d H:i:s'),
    'created' => date('Y-m-d H:i:s'),
    'modified' => date('Y-m-d H:i:s'),
  ];
  $service->notifyDomainEvent('legacy command line', $event);
} catch (Exception $e) {
  echo $e->getMessage() . PHP_EOL;
}
echo 'Domain event complete' . PHP_EOL;
```

- *When implementing a use case in different framework projects (e.g., legacy and greenfield), which layer(s) may sharply differ?* Repository. The factory could be similar or different.

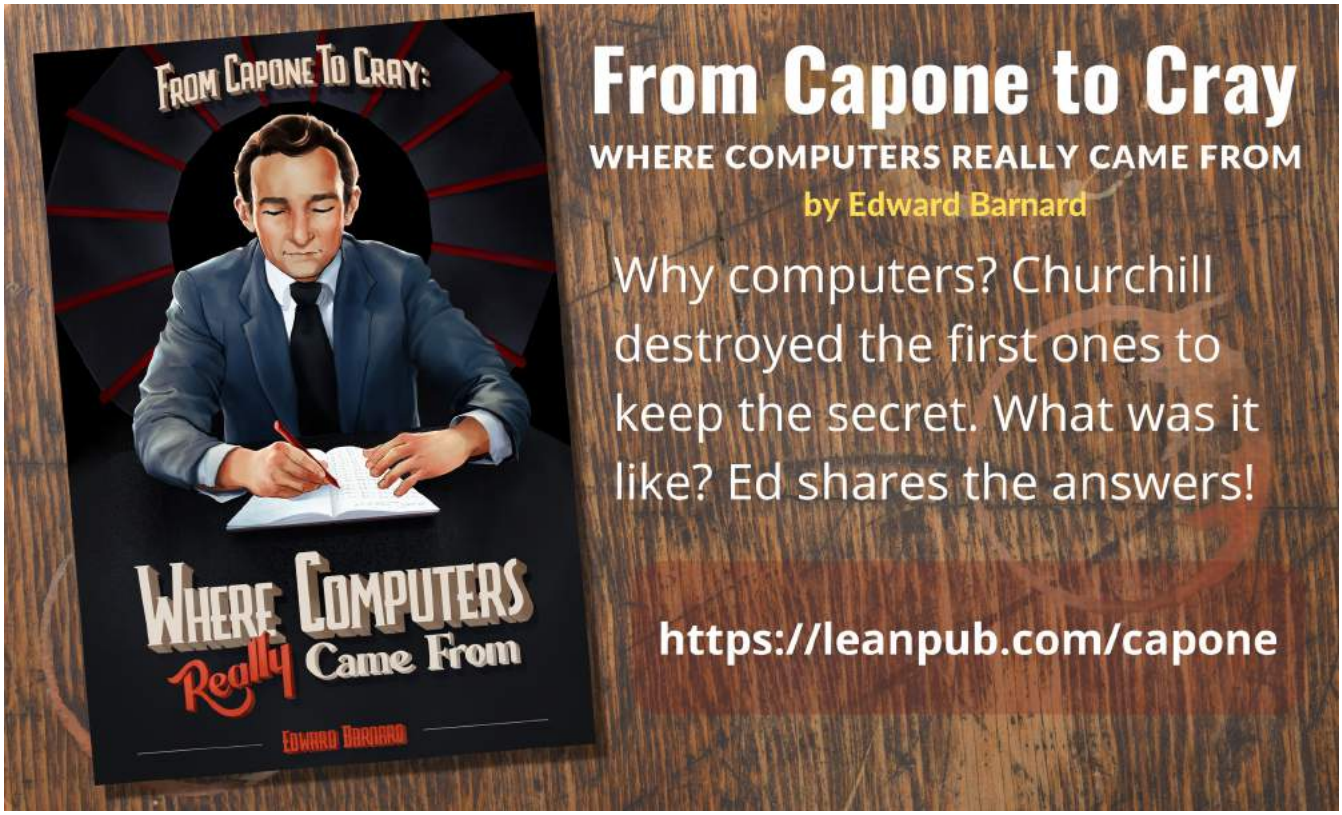
Summary

- The Domain Event command acts as a test harness to exercise the feature during development.
- The Domain Event factory connects up the application service.
- The Domain Event repository contains any database-specific and framework-specific code.
- The Domain Event application service is our entry point, the use case handler.
- The Domain Event application service interface shows an example of the Separated Interface pattern.

We don't yet know why the Separated Interface is necessary here, but this example does show where to place it within the Bounded Context pattern.



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



Drupal 9 and Varnish

Nicola Pignatelli

In this article, we'll take a closer look at the interaction between Drupal and Varnish, one of the most popular Cache Servers.

How many times have you looked at a portal that has just been completed and wondered, "why is it so slow?" Unfortunately, this problem affects many CMSs, e-commerce, and portals.

Fortunately, there is software that solves this problem. Today I will talk about the interaction between Drupal and Varnish, one of the most popular Cache Servers.

In this article, you will learn how to install Varnish and use it to accelerate Drupal pages. I suppose you installed Drupal through composer, as I explained in my previous article (PHP Architect April 2022 / Drupal 9 – Introduction and Installation[1] .

Install Varnish

I will install Varnish version 6.x on Ubuntu 21.10. I will assume that Varnish is installed on the same server of Apache2.

To verify the Varnish version, type the next command:

```
apt info varnish
```

Figure 2:

```
root@test-server:/var/www/html/site.dev/web/sites/default# apt info varnish
Package: varnish
Version: 6.5.2-1
Priority: optional
Section: universe/web
Origin: Ubuntu
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Original-Maintainer: Varnish Package Maintainers <team+varnish-team@tracker.d
Bugs: https://bugs.launchpad.net/ubuntu/+filebug
Installed-Size: 2,788 kB
Provides: varnishabi-12.0, varnishabi-strict-e7233b0ad26390433418l9d19a8d2e41
Pre-Depends: init-system-helpers (>= 1.54~)
Depends: libc6 (>= 2.33), libedit2 (>= 2.11-20080614-0), libncursesw6 (>= 6),
 libc-dev, lsb-base
Suggests: varnish-doc
Homepage: https://www.varnish-cache.org/
Download-Size: 966 kB
APT-Sources: http://it.archive.ubuntu.com/ubuntu impish/universe amd64 Packag
Description: state of the art, high-performance web accelerator
 Varnish Cache is a state of the art web accelerator written with
 performance and flexibility in mind.
 .
 Varnish Cache stores web pages in memory so web servers don't have to
 create the same web page over and over again. Varnish serves pages
 much faster than any application server; giving the website a
 significant speed up.
 .
 Some of the features include:
  * A modern design
  * VCL - a very flexible configuration language
  * Load balancing with health checking of backends
  * Partial support for ESI - Edge Side Includes
  * URL rewriting
  * Graceful handling of "dead" backends

root@test-server:/var/www/html/site.dev/web/sites/default#
```

The output shows that the version in the repository is 6.x: See Figure 1.

To install the Varnish version, type the next command:

```
apt install varnish
```

Figure 1:

```
root@test-server:/var/www/html/site.dev/web/sites/default# apt install varnish
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  cpp cpp-11 gcc gcc-11 libasan6 libatomic1 libc-dev-bin libc-devtools libc6-dev
  libquadmath0 libtirpc-dev libtsan0 libubsan1 libvarnishapi2 linux-libc-dev man
Suggested packages:
  cpp-doc gcc-11-locales gcc-multilib make autoconf automake libtool flex bison
The following NEW packages will be installed:
  cpp cpp-11 gcc gcc-11 libasan6 libatomic1 libc-dev-bin libc-devtools libc6-dev
  libquadmath0 libtirpc-dev libtsan0 libubsan1 libvarnishapi2 linux-libc-dev man
0 upgraded, 27 newly installed, 0 to remove and 2 not upgraded.
Need to get 127 MB of archives.
After this operation, 381 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Output for the previous command is: See Figure 2.

Then I click the Y button, and Varnish will be installed on the server.

Now you must change ports for Apache Web Server and Varnish because the browser gets the URL on port 80, so you must invert ports.

Stop Varnish and Apache:

```
service apache stop
service varnish stop
```

Change Apache port from 80 to 8080 into the next file:

```
vim /etc/apache2/sites-enabled/000-default.conf
```

Change Varnish port from 6081 to 80 into the next file: See Figure 3.

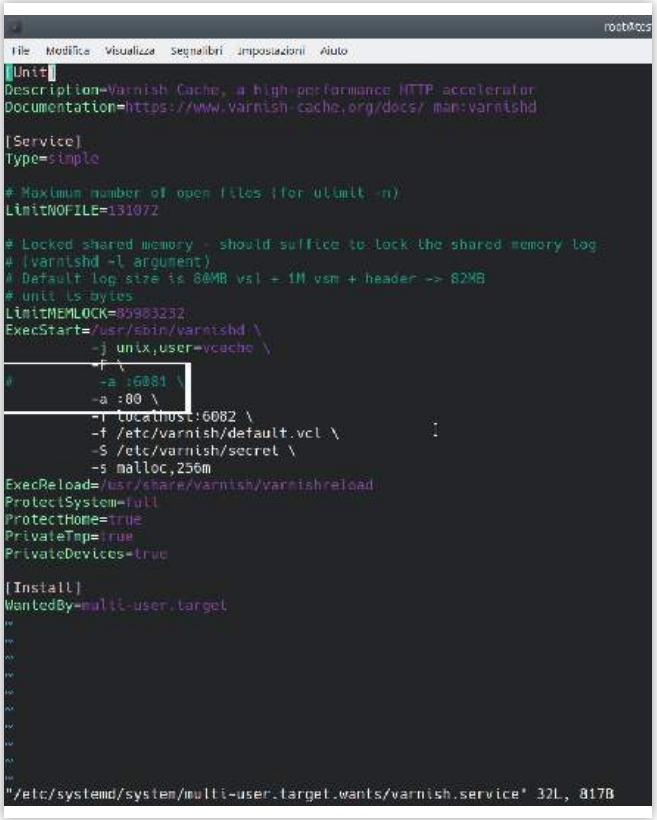
```
vim /etc/systemd/system/multi-user.target.wants/varnish.service
```

Start Varnish and Apache:

```
service apache start
service varnish start
```

1 *PHP Architect April 2022 / Drupal 9 – Introduction and Installation: https://phpa.me/drupal9-Introduction*

Figure 3:



Install Contrib Modules

To invalidate the varnish cache, Drupal provides us with some modules that are a perfect fit for us:

Purge Module

The Purge module[2] is a modular external cache invalidation framework. This module facilitates cleaning external caching systems, reverse proxies and, CDNs as content changes. This allows external caching layers to keep unchanged content cached infinitely, making content delivery more efficient, resilient, and better guarded against traffic spikes.

To download the purge module, use the following composer command:

```
composer require drupal/purge
```

This module contains submodules that must be enabled:

```
drush en purge
drush en purge_drush
drush en purge_tokens
drush en purge_ui
drush en purge_processor_cron
drush en purge_queuer_coretags
drush en purge_processor_lateruntime
```

2 *Purge module:* https://www.drupal.org/project/purge

Generic Http Purger Module

Generic HTTP Purger module[3] provides a generic HTTP-based purger to the Purge project and allows site builders to support caching platforms and CDNs that aren't supported by any other modules.

To download purge_purger_http module, use composer:

```
composer require drupal/purge_purger_http
```

This module contains submodules that must be enabled:

```
drush en purge_purger_http
drush en purge_purger_http_tagsheader
```

Varnish Purger Module

Varnish Purger module[4] is the Varnish purger for the Purge module. It provides a generic HTTP-based purger to the Purge project and allows site builders to support caching platforms and CDNs that aren't supported by any other modules.

To download the varnish_purge module, use composer :

```
composer require drupal/varnish_purge
```

This module contains submodules that must be enabled:

```
drush en varnish_purger
drush en varnish_focal_point_purge
drush en varnish_image_purge
drush en varnish_purge_tags
```

Cache and Purge Configuration

To configure Varnish to use with Drupal, go to the Performance section to tune caching and cache invalidation settings. Before we can configure how Drupal invalidates objects from the cache, we must first ensure the objects are properly stored in a cache. So you must set TTL (Time To Live).

Go to Performance Page (/admin/config/development/performance). Select a value from `Browser and proxy cache maximum age` and click the `Save configuration` button. See Figure 4.

You should set the TTL to one year or more. When content is updated, the cache invalidation mechanisms will make sure outdated content is removed from the cache automatically.

Purge

The Purge section is in Figure 5.

First of all, you must add a purger. In this case, we add "Varnish Purger". See Figure 6.

3 *https://phpa.me/purge-purger:* https://phpa.me/purge-purger

4 *https://www.drupal.org/project/varnish_purge:* https://www.drupal.org/project/varnish_purge

Figure 4:

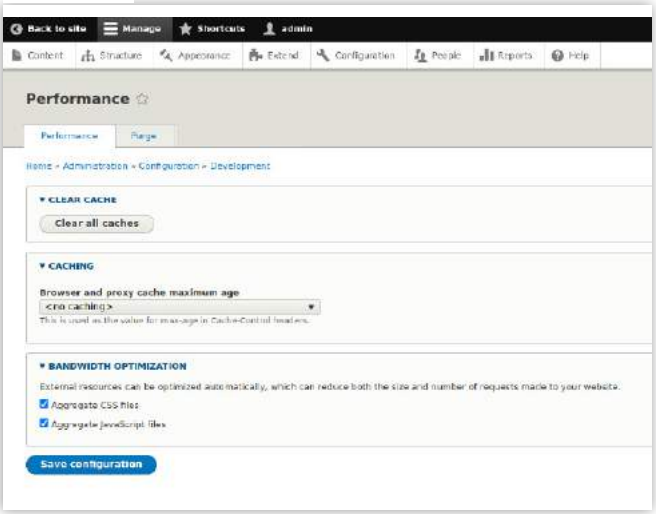


Figure 5:

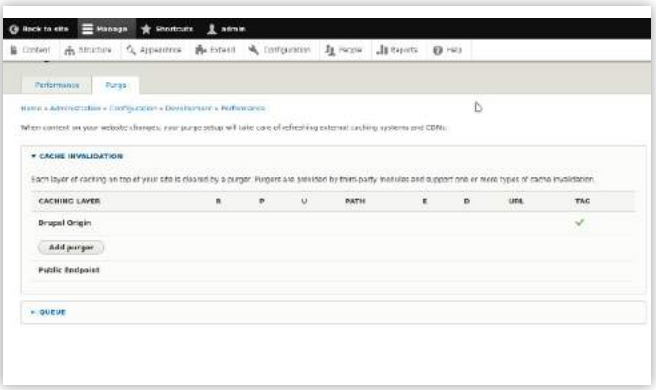
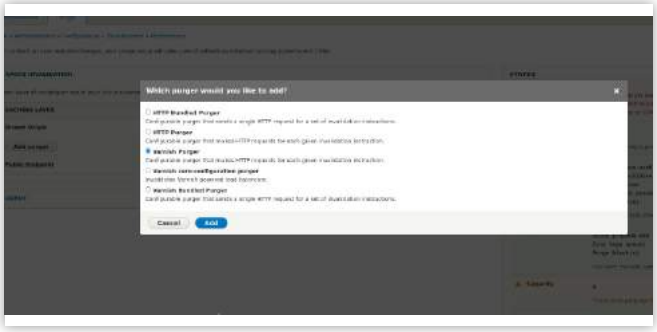


Figure 6: Then we must configure the varnish purger.



Click to configure option. See Figure 7.

Figure 7:

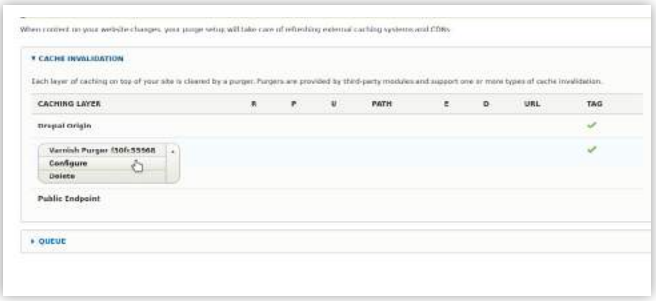


Figure 8: In figure 8, we see the Varnish Purger configuration.

We insert a name and select to purge everything. Other options are at your discretion according to your architecture and your needs.

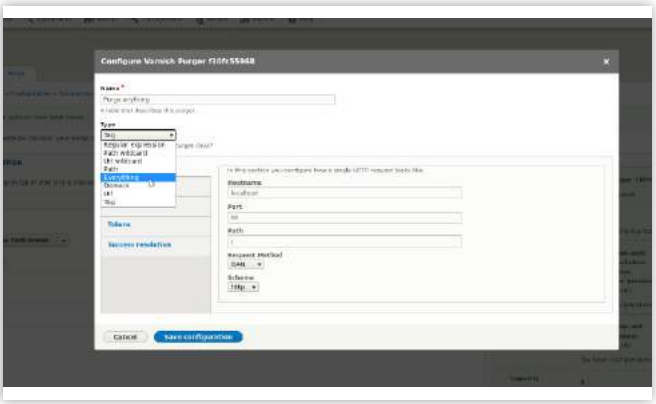
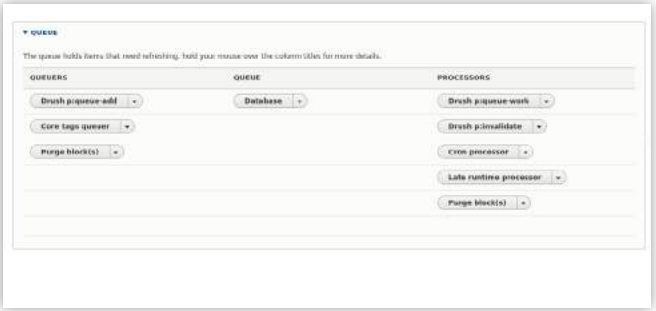


Figure 9:



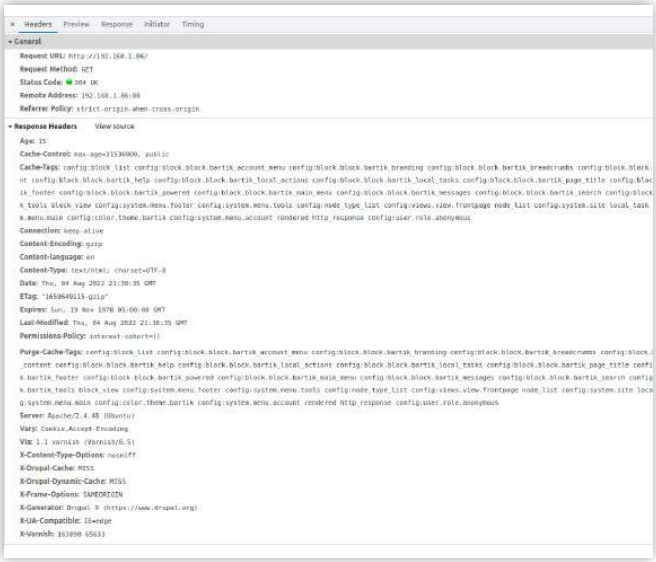
The Queue section permits to hold items that need refreshing and purging when necessary. See Figure 9.

- Queuers add items to the queue upon certain events that processors will process later.
- The queue holds 'invalidation items', which instructs what needs to be invalidated from external caches.

- Processors are responsible for emptying the queue and putting the purgers to work each time they process. Processors can work the queue constantly or at timed intervals; it is up to you to configure a policy that makes sense for the traffic nature of your website. Multiple processors will not lead to parallel processing or conflicts; instead, it simply means the queue is checked more often.

If you request a webpage, you will see firebug headers about Varnish. See Figure 10.

Figure 10:



requirements.markdown

```
### Requirements for Drupal 9:
- PHP 7.3+. PHP 8 is supported from Drupal 9.1.0
- Web Server
    - Apache 2.4.7+
    - Apache 2.4.7+
- Web Server
    - Varnish 6.0
- Drush
    - version 10+ require PHP 7.1+
    - version 11+ require PHP 7.4+

### Other Software:
- Composer

### Related URLs:
- PHP - <http://www.php.net>
- Apache - <http://www.apache.org>
- Varnish - <https://varnish-cache.org>
- Drupal - <http://www.drupal.org>
- Composer - <https://getcomposer.org>
- Drush - <https://www.drush.org/latest/install>
```

Conclusion

In this article, you learned how to install, configure and use Varnish with Drupal to manage content caching.

Varnish permits managing hundreds of simultaneous requests without the system going down.

It is possible to configure Varnish in load balancers or via docker, but this is another story I leave you to discover as young Indiana Jones in search of the lost ark.

Requirements for Drupal 9:

- PHP 7.3+. PHP 8 is supported from Drupal 9.1.0
- Web Server
  - Apache 2.4.7+
  - Apache 2.4.7+
- Web Server
  - Varnish 6.0
- Drush
  - version 10+ require PHP 7.1+
  - version 11+ require PHP 7.4+

Other Software:

- Composer

Related URLs:

- PHP - http://www.php.net
- Apache - http://www.apache.org
- Varnish - https://varnish-cache.org
- Drupal - http://www.drupal.org
- Composer - https://getcomposer.org
- Drush - https://www.drush.org/latest/install

Nicola Pignatelli has been building PHP applications since 2001 for many largest organizations in the field of publishing, mechanics and industrial production, startups, banking, and teaching. Currently, he is a Senior PHP Developer and Drupal Architect. Yes, this photo is of him.
@pignatellicom

New and Noteworthy

PHP Releases

PHP 8.2.0 (RC 1 available for testing):

https://www.php.net/archive/2022.php#2022-09-01-4

PHP 8.1.10 (Released):

https://www.php.net/archive/2022.php#2022-09-01-1

PHP 8.0.23 (Released):

https://www.php.net/archive/2022.php#2022-09-01-3

News

What's New in Composer 2.4

Composer, PHP's de-facto dependency manager. brings several new features in its upcoming Composer 2.4 release. It brings new commands such as audit and bump, support for shell completion on supported shells, suggestions to install a package with --dev flag where appropriate, improved process signal handling, and more.

https://phpa.me/composer-24

Testing Randomness of PHP Random Number Functions

Random number generation is a process that yields numbers that cannot be reasonably predicted. The sequence of numbers should not be predictable, and it plays a significant role in applications that rely on the unpredictability of the random number sequence.

https://phpa.me/testing-php-rand-functions

Asymmetric Visibility

Learn about the desire to make class properties publicly read only, but allowed to be set privately or protectedly. The benefits of a readonly public variable are great in a Idempotent Value Object, but sometimes the object itself needs to be able to set the property while allowing the public to read it without separate getters needing to be defined.

https://phpa.me/externals-asymmetric-visibility

What's new in PHP 8.2

PHP 8.2 will be released on November 24, 2022. In this post, Brent goes through all features, performance improvements, changes and deprecations one by one.

https://phpa.me/new-in-php-82

PHP performance across versions

Do you need a reason besides awesome syntax to update to the latest PHP version? Is performance a good one?

Brent did some casual benchmarks on a WordPress installation from PHP 5.6 to PHP 8.1, here are the results:

https://phpa.me/stitcher-php-performance

Light colour schemes are better

Brent recently released a video about light colour schemes. He's been encouraging people to try them out for a while now, with pretty good results, actually. He's challenged people to try it out for a week, it certainly doesn't work for everyone, but he's gotten lots of positive reactions on it as well. People actually got rid of persisting headaches because of it, they found their code to be more readable, etc.

https://phpa.me/light-color-schemes-are-better

Deprecated dynamic properties in PHP 8.2

As is common with minor releases, PHP 8.2 adds some deprecations. Deprecations often are a source of frustration, though it's important to realise they are actually very helpful. Brent has already written about dealing with deprecations in general, so if you're already feeling frustrated, maybe it's good to take a look at that post first. In this article he wants to focus on one deprecation in particular in PHP 8.2: deprecated dynamic properties.

https://phpa.me/stitcher-deprecation-82

Everyone's Eyes and Ears

Beth Tucker Long

If we want our projects to reach a multitude of people, we need to realize what a multitude of people need from what we build.

I remember the first time I realized that the buildings and houses I was passing by were not just markers on my route from point A to point B. They were full of people living their own lives and having their own experiences. People who had no connection to me. People who saw the world through their own eyes, thinking their own thoughts, being completely separate people from me.

This realization set me on a new course. One of starting to look outward instead of just inward. One of discovering how the same experience can look different depending on the eyes seeing it. Understanding that the history your eyes have seen greatly influences the future you can envision.

It's a journey I am still on and applies to every part of life. Take a simple contact form. Something all of us have dealt with, likely, many times. A few fields like Name, Subject line, Message, and a submit button. Nothing fancy, but it can still be very complicated to discuss. For example, if we ask everyone what we need:

Person A: We need a contact form with a really big message field because we need a way for our customers to tell us everything they want to say.

Person B: We need a contact form with a captcha to prevent spam.

Person C: We need a contact form that asks for two different ways to contact people so we can always get a hold of someone, even if there is a typo in one field.

Person D: We need a contact form with a proper tab order so it's easy to fill out with a keyboard.

Person E: We need a short contact form, so it doesn't take long to fill out.

Person F: We need a contact form with good labels so it's easy to understand what should be in each field.

And so it goes, on and on. Initially, it may seem like this group is working against each other, all wanting something different. They cannot agree on what the form needs to do. This may seem frustrating, but taking a step back, this is happening because each person is bringing their own successes, their own frustrations, and their own lived experiences to this project. While it takes time to work through what each person wants or is concerned about, if we take the time to understand where each person is coming from, we will end up with a far superior end product that will appeal to a broader audience. For example, if we ask everyone to describe the contact form experience that sticks most in their mind:

Person A: I recently tried to submit a bug report to a company and couldn't paste in the full error message because the message field was too small. That was frustrating because I was trying to help them, and they were stopping me.

Person B: I am on a local sports team, and I am so annoyed by all of the spam that I get from our mailing list because they allow anyone to post to the list with no verification.

Person C: I recently tried to respond to a customer service request only to discover that the customer made a typo in their email, and even though I had an easy answer for them, I had no way to get in touch with them to help them solve their issue.

Person D: I hurt my elbow a few months ago, and it was really difficult for me to switch between the keyboard and the mouse, so I really had a lot of trouble with forms that did not support keyboard tabbing between fields.

Person E: I just wanted to contact a company to ask if they provide service x, but before I could even send the message, they made me fill out a six-page questionnaire asking for all sorts of personal information, and I gave up halfway through. I wasted five minutes filling out this form and never even got to the point where I could ask my question.

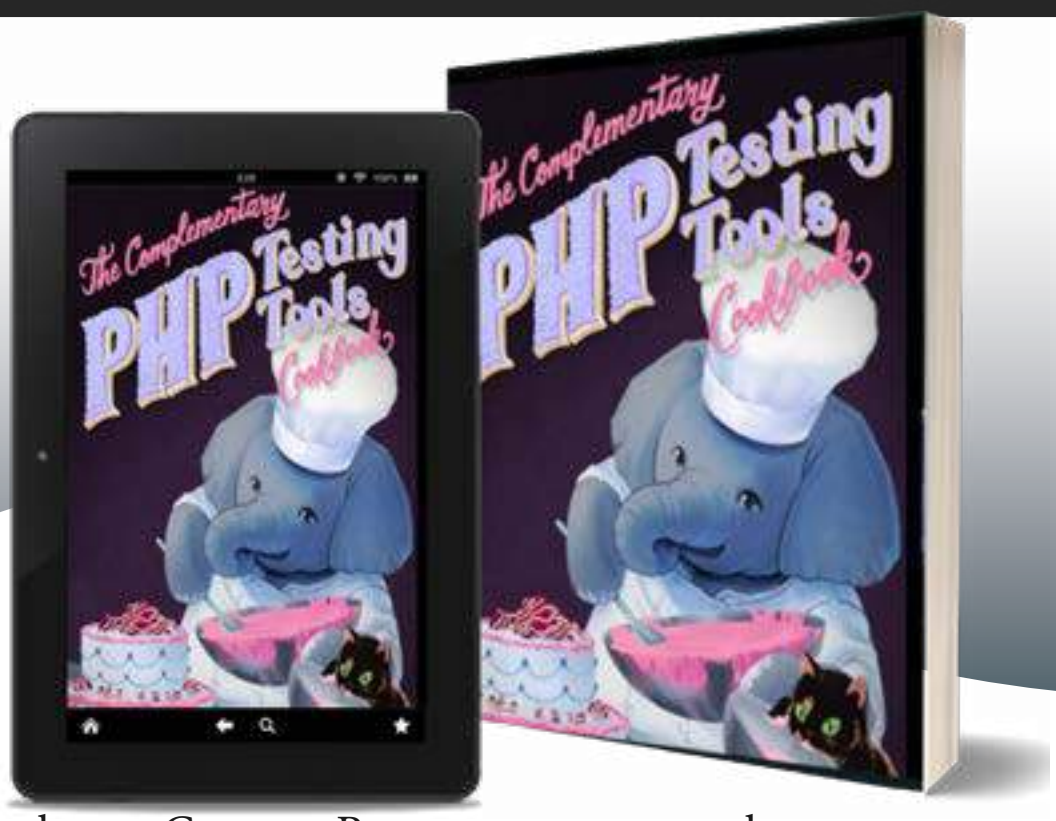
Person F: I was in an accident, and it affected my eyesight, so I have a screen reader that reads forms to me, and it is so frustrating when the field labels are generic, and I have no idea what information they really need.

Different people. Different experiences. Different insights. This is what we need in our projects. If we want to reach a multitude of people, we need to realize what a multitude of people need from what we build. When building your next project, take a moment to reach out to people in your target audience. Find out what experiences they have had that can make your project better. Remember, everyone's eyes have seen different things than yours have, and then use your ears to listen.

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 and Full Stack Madison 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



Learn how a Grumpy Programmer approaches improving his own codebase, including all of the tools used and why.

The Complementary PHP Testing Tools Cookbook is Chris Hartjes' way to try and provide additional tools to PHP programmers who already have experience writing tests but want to improve. He believes that by learning the skills (both technical and core) surrounding testing you will be able to write tests using almost any testing framework and almost any PHP application.

Available in Print+Digital and Digital Editions.

Order Your Copy

phpa.me/grumpy-cookbook