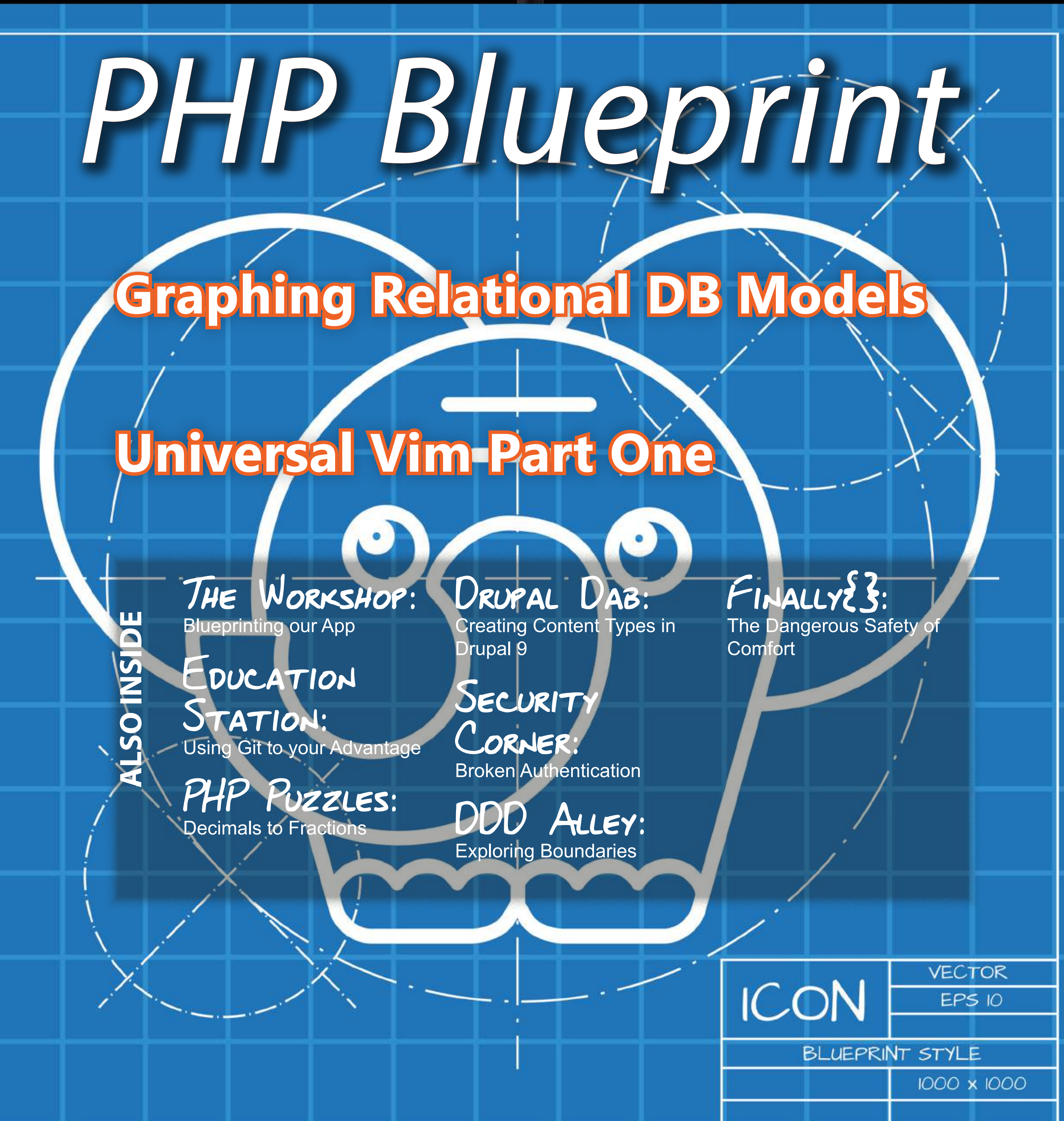
www.phparch.com

August 2022
Volume 21 - Issue 8

php[architect]

The Magazine For PHP Professionals

PHP Blueprint



Graphing Relational DB Models

Universal Vim Part One

Honeybadger provides all the context you need to understand what is causing an exception

Honeybadger.io

The Web Developer's

SECRET WEAPON!

Exception, uptime, and cron monitoring, all in one place and easily installed in your web app. Deploy with confidence and be your team's devops hero.

Are exceptions *all* that keep you up at night?

Honeybadger gives you full confidence in the health of your production systems.

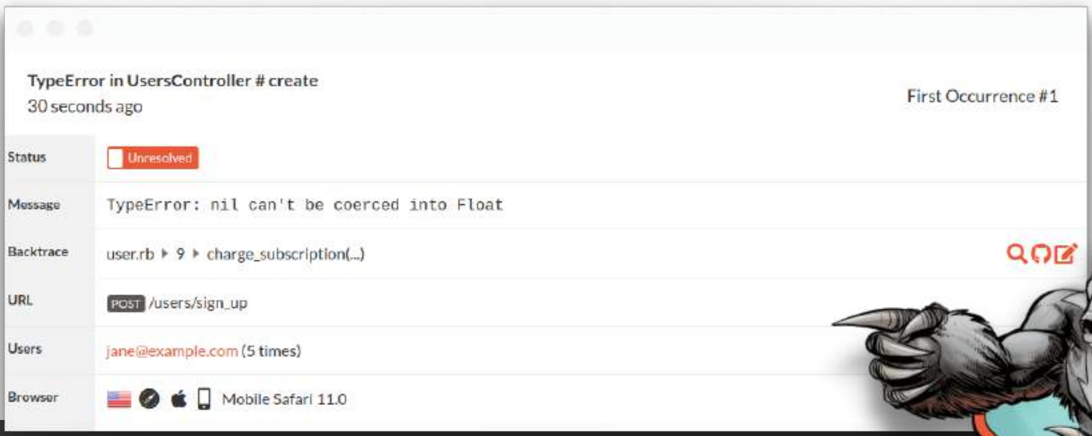
DevOps monitoring, for developers. *gasp!*

Deploying web applications at scale is easier than it has ever been, but monitoring them is hard, and it's easy to lose sight of your users. Honeybadger simplifies your production stack by combining three of the most common types of monitoring into a single, easy to use platform.

Exception Monitoring
Delight your users by proactively monitoring for and fixing errors.

Uptime Monitoring
Know when your external services go down or have other problems.

Check-In Monitoring
Know when your background jobs and services go missing or silently fail.



TypeError in UsersController # create
30 seconds ago

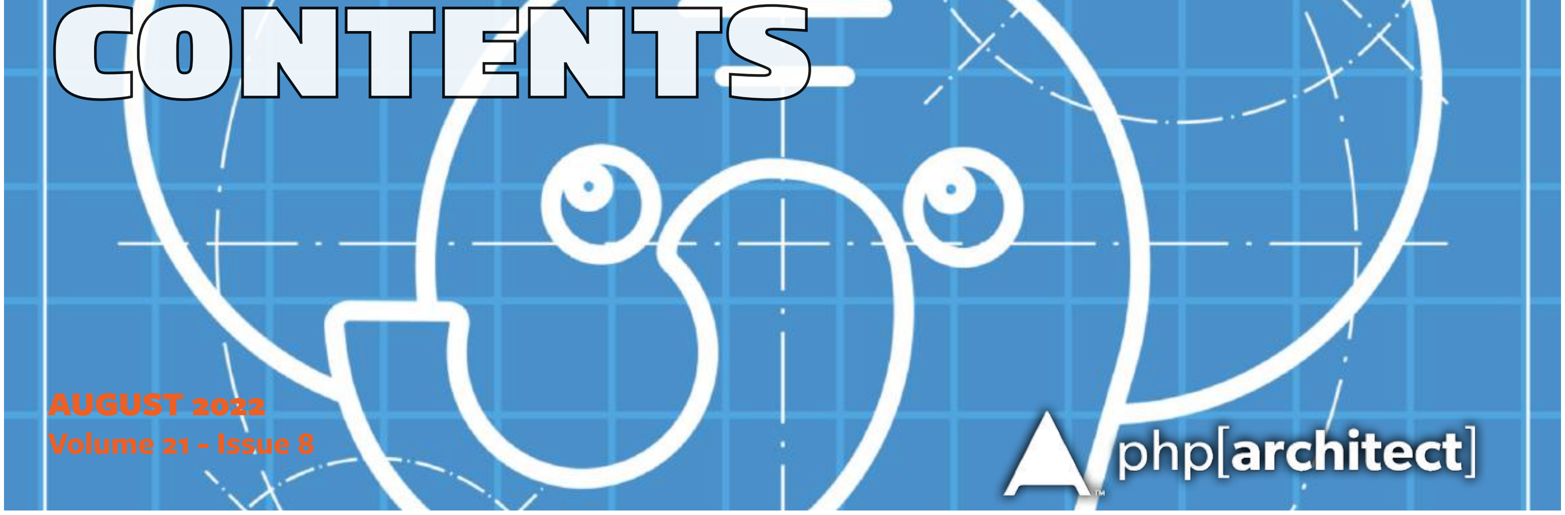
First Occurrence #1

Status	Unresolved
Message	`TypeError: nil can't be coerced into Float`
Backtrace	user.rb ▸ 9 ▸ charge_subscription(...)
URL	POST /users/sign_up
Users	jane@example.com (5 times)
Browser	Mobile Safari 11.0

Start Your Free Trial Today

https://www.honeybadger.io/

CONTENTS



AUGUST 2022
Volume 21 - Issue 8

Edited on Graph paper

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

A Left Turn

Carolyn Miller

As I got older, I expected to decrease my hours but had always planned to write software part-time. After all, I use my brain, not my body, for work, so I should be set as I age. I could travel and work part-time from anywhere, forever. Right?

Long covid changed those plans. Overnight, my brain basically aged 20 years and didn't work as expected, and no amount of problem-solving could make it so. Eventually, with a lot of retraining, I could read again using special glasses. Now, I could learn again, albeit much slower than before. I'm telling you this because, with the prevalence of covid, it's highly likely that you will know or work with someone who gets long covid, and they may need support from you - their friend, coworker, or boss.

I voluntarily stepped away from active coding and client meetings as I could no longer perform to my expectations. Editing and layout of the magazine for the last 11 months has been a privilege and allowed me to end my paid employment on my terms, stepping away much earlier than anticipated.

I tend to embrace change; even look for change. I wasn't one of those coders who worked on a system then sat back and maintained it. Delivered; What's next? That attribute has been helpful with the sudden turn from my expected path. Having outside interests and a social life has also made the turn easier.

When I first started studying Computer Science, I was often the only female in my classes, and I've spent most of my career working exclusively with men. After many advances in the industry, which have opened up many specialties, the percentage of women entering the field has not increased. The peak was in 1984, just as I finished schooling and started my career? Coding is a great career allowing women to support themselves, often while working from home. 1099 work allows you to set your own hours and work around outside interests or a family. The essential attributes are to be logical, curious, and enjoy solving problems. Please encourage girls and women to enter the industry, advance them when they do, and treat them respectfully.

Lastly, I'd like to share an exercise that was very helpful in me choosing what was most important in a job because no path is straight. Knowing what's important when interviewing allows you to be interviewing the employer, which could help you relax.

Fold a piece of paper to make 16 rectangles. Write the requirements you would like to have in your ideal job in those rectangles. Separate the rectangles. Take away 4, leaving only the most important. Take away another 4. Then another. Now, the 4 left will guide you when looking for a new job or career.

Thanks to DiegoDev Group for meeting all my requirements and excelling in my top requirement—being an honest, ethical employer.

Our features this month start with Ghlen Nagels guiding us through "Converting Relational Database Models Into a Graph." Next, Andrew Woods begins a series enlightening us on Vim and Neovim, starting with "Universal Vim Part One—No Plugins Required."

Our columnists are not on vacation this summer. The Workshop has us building some scaffolding with foreman Joe Ferguson in "Blueprinting our Application." Drupal Dab will programmatically generate content in "Create Content Type in Drupal 9" with Nicola Pignatelli.

Chris Tankersley teaches us how we can rewrite history and time in Education Station's "Using Git to Your Advantage." DDD Alley has us "Exploring Boundaries" with Edward Barnard building on the Bounded Context pattern introduced last month. And we're working with floating-point values with Oscar Merida in PHP Puzzles, "Decimals to Fractions."

How good is our communication? Eric Mann's Security Corner explores language in "Broken Authentication." Lastly, Beth Tucker Long has us finally{} talking about muscle memory in "The Dangerous Safety of Comfort."

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/August2022_code

Graphing Relational DB Models Graph

Ghlen Nagels

In this article, we will explore the wonderful world of Graph Databases. After discussing the myriad reasons that might prod you to change paradigms, we will actually do it with PHP!

Introduction

Every PHP developer knows the classic LAMP[1] stack: Linux, Apache, MySQL, and PHP. The term was first coined in the German computing magazine *Computertechnik* in April 1998, making it 24 years old; coincidentally, the term web 2.0 appeared one year later (DiNucci 32)[2] . As powerful as it is, SQL technology has its limits like any other.

PHP developers tend to be accustomed to JavaScript and a subset of the ecosystem built on top of it. We might have worked with Nginx or serverless hosting solutions. Even the machine we work on might be running Windows or Mac. The only part of LAMP that PHP devs tend to stick to is SQL.

Yet, I've had many clients who have hit the limits of their SQL database and want to start migrating to a database that performs better when working with highly relational data or wants to add data science to their stack.

This article will first explain the symptoms when hitting the limits of SQL, so you can decide whether it is the right moment to change or migrate database paradigms. Then we will explain the core differences between the two paradigms and provide hands-on code examples of migrating the data. All code samples[3] and migrations are available online for free under the MIT license.

Growing Pains

Speaking from experience: performance issues are the biggest trigger for people to move from SQL to Graph. Once they finally migrate, they tend to realize they had a two-for-one deal: the modeling of the data and structure of the application become much smoother, with fewer hacks, workarounds, and caching.

I've also had clients that already used Graph Databases beforehand; they usually had a good vision ahead and knew they needed graphs to query the data efficiently, or they already had a big workflow setup for data analysis.

I've compiled a small list of typical symptoms that, in isolation, are pretty manageable but, when combined, can halt the growth of your application to a snail's pace:

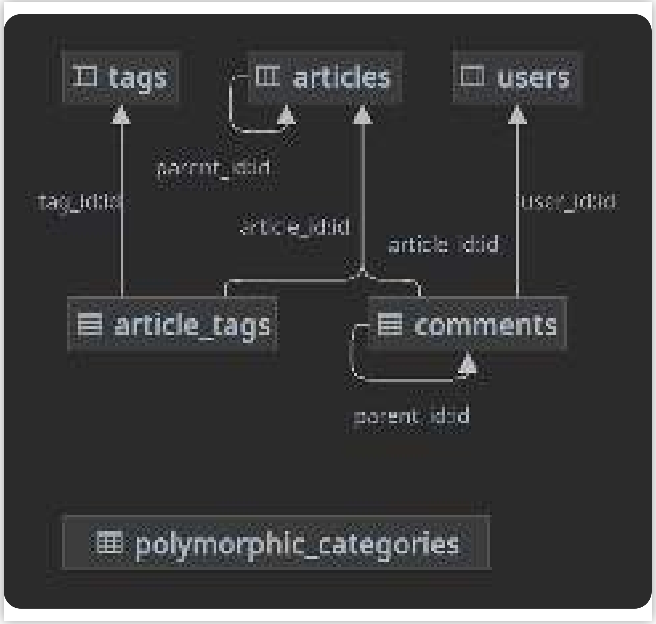
- **Slow join operations**. Tables in SQL have no way to treat relationships as first-class citizens. They rely on foreign key relations to emulate this behavior. Once the dataset grows, SQL can become the limiting factor in your app.
- **Caches for everything.** Caching can be an excellent tool for optimizing performance. Still, it is also a significant source of bugs and developer confusion, as it is one more layer in the application that people tend to forget quickly. The caching often happens due to complex queries and the computation needed to build the actual dataset in PHP.
- **Esoteric solutions to simple problems.** I've seen people create their indexing system to get around the limitations of thinking in tables.
- **A consciously non-normalized table.** Because joins are expensive, it becomes common practice to duplicate small pieces of information in the table instead of extracting the data from it and using foreign keys.
- **Lots of database migrations.** One of the biggest strengths of SQL can also be its biggest weakness. If you find yourself continuously migrating, changing, creating, and updating database schemas, it might be too soon to work with predefined tables.
- **Multiple foreign keys on a table when only one relationship exists.** You introduce redundancy through multiple foreign key columns if a table can point to two or more different tables, but only one per row. This practice can become challenging to manage in the application layer.
- **Introducing data science.** Gartner predicts that data and analytics innovation will use graph technologies in more than 80 percent of data and analytics innovations (King[4] . The chances are that you'll be looking at this technology if you want to introduce data science to your tech stack.

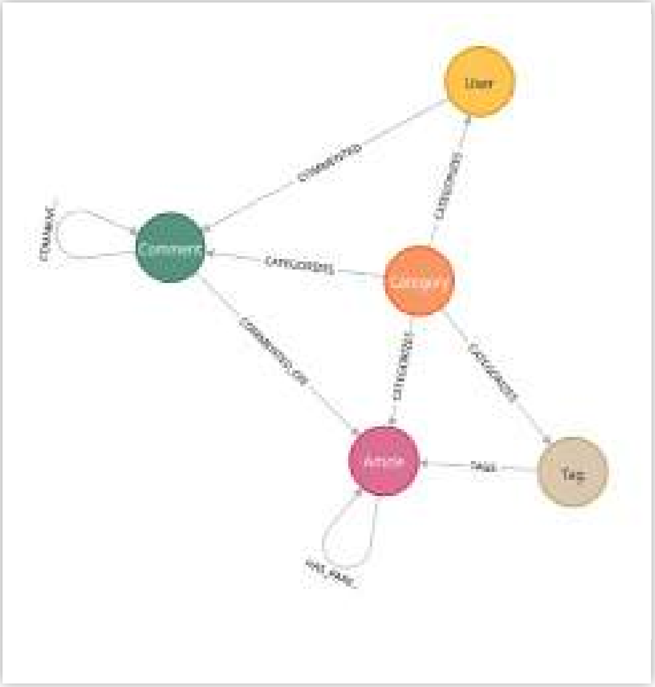
1 *LAMP: https://w.wiki/5S2e*

2 *DiNucci 32: http://darcyd.com/fragmented_future.pdf*

3 *code samples: https://github.com/transistive/book-example*

4 *King: https://phpa.me/gartner-top-10-data-analytic-2021*





Modeling the Data

Modeling the data between the two database paradigms is surprisingly simple!

The imaginary application we are working with is a book collaboration platform. It allows people to write articles and combine them in a tree structure to form a book eventually. Each article can have tags to hint about the subject. Users can also comment on each article or comment on another comment to create what is colloquially known as sub-comments. There is also a bigger picture in this use case: some esoteric deep learning algorithm that categorizes comments, articles, users, and tags. I'm not a data scientist, but it's an excellent example of broad polymorphic associations in the database.

When looking at the schema for SQL, we can identify four different cases to solve in our PHP script. If you understand these, you can migrate anything from SQL to Neo4j!

Using the example in GitHub, you can quickly get up and running if you have PHP 8.1 or up and docker installed. Just run the Listing 1 commands in your working directory to run all the scripts.

Listing 1.

```
git clone git@github.com^[git@github.com:
      <mailto:git@github.com>]:transistive/book-example.git
cd book-example

composer install # install PHP all libraries

docker-compose up -d # set up MariaDB and Neo4j
vendor/bin/phinx migrate # migrate the SQL schema
vendor/bin/phinx seed:run # generate random MariaDB dataset
php migrate_to_neo4j.php # migrate the data to Neo4j
```

For those interested in working with the driver immediately, you can create it like this:

```
// Create a driver to connect to Neo4j with user
                 Neo4j and password test
$driver = Driver::create('neo4j://neo4j:test@localhost');
// Verify the connectivity beforehand to make sure the
                 connection was successful.
$driver->verifyConnectivity() ?? throw new Error(
                 'Cannot connect to database');
// You can run queries on a session
$session = $driver->createSession();
```

A great blog post[5] is available for people to read a more in-depth explanation of how the driver actually works. You can create PDO with a DSN like this:

```
$pdo = new PDO('mysql:host=127.0.0.1;port=3306;
                         dbname=test', 'test', 'sql');
```

Now that we've got our main tools to access both databases, we can start the actual work!

Inserting Rows As Nodes

Most rows directly translate to nodes. While there are certain edge cases, this makes the process of loading rows as nodes into Neo4j deceptively easy. Consider this naive approach:

```
$articles = $pdo->query('SELECT * FROM articles')
                         ->fetchAll(PDO::FETCH_ASSOC);
$session->run(<<<'CYPHER'
UNWIND $articles AS row
MERGE (a:Article {id: row['id']})
ON CREATE SET a.row = row
CYPHER, compact('articles'));
```

5 bpost: https://neo4j.com/developer-blog/connect-to-neo4j-with-php/

We are off to a great start! Notice how we requested a row of arrays from PDO. All we did was pass them through the Neo4j driver and let Neo4j do the heavy lifting by unwinding the array of articles and merging them as nodes, which in this case means only creating the node if there isn't already one with the same ID.

But the critical point: it's just a start. We're not done yet. There are so many potential bugs in this code!

In order of obviousness, we can make the query a lot more generic by parameterizing the tag and moving it to a function or method:

```
public function storeRowsAsNodes(string $tag,
                                  iterable $nodes): void {
    $session->run(<<<CYPHER
    UNWIND \$nodes AS node
    MERGE (x:$tag {id: node.id})
    ON CREATE SET x = node
    CYPHER, compact('nodes'));
}
```

A couple of essential changes make this example worth looking at—we typed the nodes as an iterable, not an array. The driver accepts an iterable instead of an array, making our lives much easier in the next step. We also moved from NOWDOC to HEREDOC. It requires us to escape the dollar sign if we want to pass the variable reference to CYPHER, as it will try to interpolate a PHP variable otherwise.

While this example is excellent from a code reusability standpoint, the biggest red flag in the code is memory! We are pulling all the data from the disc into memory, potentially causing memory overflow. Let's use generators for this instead to keep memory usage low as shown in Listing 2.

By simply chunking the generator, we can have fine-grained control over memory usage:

```
foreach (Helper::chunk($pdo->yieldRows('articles'), 25000)
              as $chunk) {
    Helper::logCreatedNodes(
        TablesEnum::ARTICLES,
        $nodes->storeRowsAsNodes('articles', $chunk)
    );
}
```

You can find the chunking algorithm in the example repository[6] but might already have access to one through another library or framework you are using.

These two methods can be potent as it allows you to easily migrate over any row in any table to any node in a highly performant and memory-efficient manner!

Basic Foreign Key Relations

While relationships get treated as second-class citizens in SQL, they are the bread and butter of any Graph Database. The primary foreign key example uses a single column in SQL. To connect the data, we need to generate cipher queries that go over all the nodes, find the node with the corresponding ID, and connect them. It is genuinely trivial to do in PHP and Neo4j!

Listing 2.

```
public function yieldRows(string $table): Generator {
    $statement = $this->pdo->query(sprintf(
                                    'SELECT * FROM %s', $table));

    while ($row = $statement->fetch(PDO::FETCH_ASSOC)) {
        // sneaky way to translate dates to DateTime
        // objects to reduce headaches later
        $row['created_at'] = new DateTime($row['created_at']);
        $row['updated_at'] = $row['updated_at'] === null ?
                             null :
                            new DateTime($row['updated_at']);

        yield $row;
    }
}
```

```
public function connectArticles(): ResultSummary
{
    return $this->session->run(<<<'CYPHER'
    MATCH (child:Article),
          (parent:Article {id: child['parent_id']})
    MERGE (child) - [:HAS_PARENT] -> (parent)
    CYPHER)->getSummary();
}
```

Notice how in this example, we return the summary of the result. It allows us to access some exciting information. It holds information about query timing, created and updated nodes and relationships, etcetera.

Multiple Foreign Keys

Having multiple foreign keys in a single table is probably the first big hurdle a Neo4j novice will encounter. The naive and wrong approach is usually something along the lines of trying to create multiple relationships in a single query:

```
public function connectCommentToArticles(): ResultSummary {
        return $this->session->run(<<<'CYPHER'
        MATCH (c:Comment), (a:Article {id: c.article_id})
        MERGE (c) - [:COMMENTED_ON] -> (a)
        MATCH (c), (p:Comment {id: c.comment_id})
        MERGE (c) - [:COMMENTED_ON] -> (p)
        CYPHER)->getSummary();
}
```

What's wrong about this is that you are reusing the original comment match. Because in the initial application, we decided that a comment can point to another comment or an article, the initially matched comments will never match another comment through their comment_id as they will all have null as their value.

6 *example repository: https://phpa.me/github-book-ex-helper*

Therefore, it is necessary to match a new comment completely, or better and clearer yet, split them into different queries entirely as shown in Listing 3.

Splitting them into separate queries and methods is much clearer in general. While it is possible to create complex all-in-one queries, the consensus in the Neo4j community is almost always to prefer smaller, simple queries over big ones. One reason to move to complex queries is to strike a balance between IO performance and code maintainability. Many smaller queries in a real-time application can become problematic when they block everything every time.

Pivot Tables

When it comes to pivot tables, one important epiphany is that they are relationships in disguise! These tables exist only to define a many-to-many connection between two different tables.

One subtlety to take into consideration is that pivot tables are bidirectional. Their foreign keys point both ways. In Neo4j, every relationship is unidirectional. You can still query a relationship without defining a direction, though.

When defining the relationship type, it should also convey the direction. For example, a Tag TAGS an Article. TAGS is the type of relationship that points from a tag to an article. If the verb becomes passive, the direction inverts: an Article is TAGGED_BY a Tag.

While it might seem pedantic to some, studies have shown that the active voice improves clarity (*Active and Passive Voice—Scholarly Voice—Academic Guides at Walden University*, n.d.)[7] . So when picking a direction for pivot tables, consider which table acts on the other one when designing the direction. Here is a code example:

```
public function connectTags(): ResultSummary {
     return $this->session->run(<<<'CYPHER'
     MATCH (at:ArticleTag), (t:Tag {id: at['tag_id']}),
                      (a:Article {id: at['article_id']})
     MERGE (t) - [ta:TAGS] -> (a)
     ON CREATE SET ta = at
     CYPHER)->getSummary();
}
```

Notice how we can also define relationships' attributes, just as we can add columns to our pivot table. To keep our queries simple, we copy and paste all the column data to the relationship attributes, just as we did with the Nodes.

We've already discussed the most common patterns, but for completeness, we'll move on to the final and most complicated stretch: polymorphic relations.

Polymorphism

Starting off this section on a slight tangent—tables and rows are often interchangeably used when comparing them to nodes in Graph Databases. It can be confusing as a Node directly translates to a row, not a table. A row is an instance of information, while a table concerns meta information such as typings, columns, etc.

For this reason, we tend to name node labels with singular nouns while we name tables with plural nouns. Language and other ideas defined by millennia of human interaction and culture are notoriously difficult to deal with in code (I'm looking at dates and your Gregorian customs). I often find it helpful to use a translation table:

```
$translationTable = [
    'article_tags' => 'ArticleTag',
    'articles' => 'Article',
    'comments' => 'Comment',
    'polymorphic_categories' => 'Category',
    'tags' => 'Tag',
    'users' => 'User',
];
```

Doing so solves one problem—translating the table name reference to the node label, but we still need to store it in the database. Because we are using iterables and generators, we can map the translations to the array before storing them into Neo4j.

```
$categories = $this->yieldRows('polymorphic_categories');
$categories = Helper::map($categories,
                          static fn (array $x) => [
    ...$x,
    ...['label' => TablesEnum::from(
                         $x['resource_table'])->asTag()]
]);
$this->storeRowsAsNodes('Category', $categories);
```

Let's drool some more over these underused generators! We've deferred the mapping of each node until the Neo4j driver sends it over the network. Using arrow functions and array packing is for bonus style points.

Listing 3.

```
public function connectCommentToArticles(): ResultSummary
{
        return $this->session->run(<<<'CYPHER'
        MATCH (c:Comment), (a:Article {id: c.article_id})
        MERGE (c) - [:COMMENTED_ON] -> (a)
        CYPHER)->getSummary();
}

public function connectParentComments(): ResultSummary
{
        return $this->session->run(<<<'CYPHER'
        MATCH (c:Comment), (p:Comment {id: c.parent_id})
        MERGE (c) - [:COMMENTED_ON] -> (p)
        CYPHER)->getSummary();
}
```

7 *Academic Guides at Walden University*, n.d.:*
https://phpa.me/waldenu-active-passive-voice

Listing 4.

```
public function listAllTags(int $articleId): array
{
    return $this->session->run(<<<'CYPHER'
    MATCH p = (:Article {id: $articleId}) <-
              [:HAS_PARENT*0..] - (:Article)
    UNWIND nodes(p) AS article
    WITH DISTINCT article
    MATCH (article) <- [:TAGS] - (tag:Tag)
    RETURN tag.name AS tag
    CYPHER, compact('articleId'))
        ->pluck('tag')
        ->toArray();
}
```

Listing 5.

```
public function topCategoryNode(int $articleId): void
{
        $node =  $this->session->run(<<<'CYPHER'
        MATCH (c:Category) - [:CATEGORIZES] -> (node)
        WITH node, collect(c) AS categoryDegree
        RETURN node
        ORDER BY categoryDegree DESC
        LIMIT 1
        CYPHER, compact('articleId'))
            ->getAsCypherMap(0)
            ->getAsNode('node');

        echo 'LABEL: ' . $node->getLabels()->first() .
              PHP_EOL;
        echo 'ID: ' . $node->getProperty('id') . PHP_EOL;
}
```

Listing 6.

```
public function doubleCommenters(int $articleId): array {
        return $this->session->run(<<<'CYPHER'
        MATCH (b:Article) <- [:COMMENTED_ON*1..] -
              (:Comment) <- [:Commented] - (u:User),
              (u) - [:COMMENTED] -> (:Comment) -
              [:COMMENTED_ON*1..] -> (a:Article)
        WHERE a <> b
        RETURN DISTINCT u AS user
        CYPHER)
            ->pluck('user')
            ->toArray();
}
```

Now that they are available in the database with their mappings included, we can define a simple query to wire the relationships together:

```
public function connectCategories(): ResultSummary
{
    return $this->session->run(<<<'CYPHER'
    MATCH (c:Category), (x {id: c.resource_id})
    WHERE c.label IN labels(x)
    MERGE (c) - [:CATEGORIZES] -> (x)
    CYPHER)->getSummary();
}
```

Some Fun Real-time Queries

All right! This has been great stuff. Now that we've discussed every major migration case, you can refer to the example project and dive deeper. It also uses basic logging functionality to provide insights into what is happening as the script goes along.

Many exciting tools are available to you with the driver and Neo4j in PHP! Let's go over some examples:

- Listing 4 shows all the tags of an article, including all its subtags
- Get the node with the highest amount of categories attached to it, as shown in Listing 5
- Find all users that commented on at least two different articles. See Listing 6.

Conclusion

We went through a lot of information in a short period. We've discussed complex database issues and how they can be solved using different tools. While impressive, it can also seem daunting. The entire article assumes quite a bit of prior knowledge. You can follow a free online interactive course[8] if you've just done a cursory glance, like what you see, and want to get started with a graph database.

Articles about database migrations tend to display the problem as straightforward to solve, with one-off solutions that will magically make all of your problems disappear—a pipe dream. It is often a massive undertaking with many subtleties that are impossible to discuss in a short 2800-word article.

You can always contact me if you want to discuss migrations or experiment with these remarkable technologies. I love to hear about your story and help out where I can! You can reach me at ghlen@nagels.tech or go to my homepage[9]. Even posting an issue on the driver GitHub[10] or StackOverflow will have someone from the community come in to look at the problem.

Ghlen Nagels is a Web Developer and Graph Database expert. He loves to keep an open mind and involve himself in multiple projects and start-ups. You'll likely find him running on the trails or enjoying some of his local favorite Belgian beers with his friends when he is not behind his keyboard. https://nagels.tech.

8 *a free online interactive course: https://graphacademy.neo4j.com*

9 *homepage: https://nagels.tech*

10 *driver GitHub: https://github.com/neo4j-php/neo4j-php-client*

LONGHORN
PHP CONFERENCE

Nov 3-5, 2022 Austin, TX

We want you to speak at Longhorn PHP!

Talk proposals due by August 11

cfp.longhornphp.com

Universal Vim Part One—No Plugins Required

Andrew Woods

Vim is a powerful editor with a long history. It's the most popular of the descendants of Vi. Over the last couple of years, a fork of Vim was born called Neovim. This article will provide you with a Vim configuration file that can be used across both Vim and Neovim.

Vim. It's the editor whose name is on the lips of every developer—whether they love it or hate it. If you're reading this article, you probably love Vim—or at least you're open to learning more about it. Vim is the most popular editor of Vi's descendants—of which there are many. However, a new contender called Neovim has emerged over the last several years. Neovim is currently fully compatible with Vim's features. As time progresses, new features may distinguish Vim and Neovim from each other. For the moment, though, we can create a single configuration file you can use with either one.

This article is about Vim *and* NeoVim. However, for brevity, I'll be using Vim to refer to both. When their needs diverge, I'll specify them individually. Case in point—Vim and NeoVim have different configuration files. Each Vim and Neovim have multiple locations where you can manage the settings. I recommend reading the help page—`:help vimrc`—to get the finer details. In fact, you should do that with every setting discussed in this article. You'll be amazed at what you learn. Neovim uses `$HOME/.config/nvim/init.vim` and Vim uses `$HOME/.vimrc` to manage config settings.

Before we go much further, let's discuss what is meant by universal. In this context, it means the configuration will work for both Vim and Neovim to help you choose the editor you want. Maybe you'd like to try Neovim, or maybe you prefer standard Vim. It's up to you. The second way this setup is universal is that it's equally useful if you want to write code, research papers, or blog posts. Perhaps you'd like to try your hand at contributing some PHP Architect articles. No matter what you want to write, the goal is to provide you with a solid foundation from which to build.

Part of that foundation is showing you what's possible without requiring any plugins. So for this article, you won't have to worry about which package manager you need to use. That's why this article is titled "Universal Vim Part One: No Plugins Required"—I'm only providing settings for what Vim provides. Don't worry; I'll be introducing some useful plugins in upcoming articles.

Universal Vim Configuration

There are many ways you can organize your vimrc file. Let's start with some high-level structure by dividing the vimrc into 6 sections:

- Settings
- Plugins
- Variables
- Leaders
- Remaps
- Auto Commands

The first thing we do in our settings is use the `set nocompatible` configuration, which refers to being compatible with Vi, Vim's predecessor, and can affect the default values for multiple options. We don't want to inherit that behavior, so setting `nocompatible` provides us with more predictable behavior. Neovim always sets it to `nocompatible`, so this one is for Vim.

The rest of the Settings section can be split into 2 main sections—user interface and user experience. User Interface relates to the visual aspects, whereas the User Experience relates to how Vim feels. Here is the structure of the vimrc

Listing 1.

```
" ==== Settings ==================
set nocompatible

" **** User Interface *************

" **** User Experience ************

" ==== Plugins ===================

" ==== Variables =================

" ==== Remaps ====================

" ==== Auto Commands =============
```

that we'll be using for the rest of the series.

User Interface Settings

Vim isn't a GUI editor in the same way that PhpStorm is. We need to build up the interface we want. Let's start by adding some settings to the User Interface section. Who doesn't love some instant gratification? As you follow along, don't forget to source your vimrc after changing it to see the changes take effect.

We begin with line numbers. As you can see, there are a few settings related to displaying line numbers. The `number` setting enables the display of real line numbers. The fifth line of content resides on line five, which is pretty straightforward. However, the next

Listing 2.

```
" **** User Interface ***************************

" Show real line numbers
set number

" Number the lines, relative to where your cursor is
set norelativenumber

" Control the left edge of the window
set numberwidth=6

" Always display the status bar
set laststatus=2

" custom status line, comment if you use a status plugin
if has("statusline")
    " Reset to empty string
    set statusline=
    " Display the buffer number
    set statusline+=\<%n\>
    " Base filename
    set statusline+=\ %t
    " Various flags
    set statusline+=\ %m%r%h%w

    " Separate left side from the right side
    set statusline+=%=

    " File format/type
    set statusline+=(%{&ff})
    " current line number / total lines
    set statusline+=\ line:%l\/%L
    " Percentage within file
    set statusline+=\ (%p%%)
    " Column number
    set statusline+=\ col:%c
endif

" ~~~~ Display Guide ~~~~~~~~~~

" Remember to add 1 to your desired maximum line length
set colorcolumn=65
" 235 is a very dark grey, good for dark backgrounds
hi ColorColumn ctermbg=235
```

setting `relativenumber` is highly preferential. You'll note that I've turned it off. However, I encourage you to try it out.

When `relativenumber` is enabled, every line is renumbered according to where the cursor is. The cursor represents line zero. As you navigate up and down, you'll see the lines get renumbered. This is very useful, particularly with Vim's motion commands. It helps guide you to the right lines when you want to say things like `6j`, `5dd`, or `12yy`. It's worth noting that since we enabled `number`, you won't see line zero. Instead, the 0 is replaced with the objective line number.

There are a couple of situations you might want to consider when deciding to enable the `relativenumber` setting. The first is pair programming. Most people aren't used to it because other editors don't have it. If the person at the keyboard is quickly navigating up and down, the other person loses their context. The other situation is file navigation. Objective line numbers inform you of your position in the file—they provide landmarks for content. If you're on line 50 of a 300-line file, that tells you about your place in the file. You lose that sense with relative line numbers because now everything is 1 through 12 (or thereabout), depending upon the height of your buffer. I encourage you to try relative line numbers, at least for a little while. You can toggle it on/off by using command mode and typing `set relativenumber!`. Note the exclamation point at the end—*that's the toggle*. Any binary setting can be toggled.

One essential part of an editor is a status bar. By default, Vim won't show you a status bar—unless you have more than one window. That is less than desirable. It's the inconsistency that bothers me. I suppose Vim is trying to help you conserve space so you have more room for your content. By using `set laststatus=2`, we ensure the status line is always displayed.

The custom `statusline` provided is useful without being overwhelming. It's more like what you'd see in an IDE. There are some good Vim status bar plugins. In the future, if you decide to use one of them, this can be removed from your vimrc.

The "display guide" is that thin line you see in the other editors, usually set to column 80. In Vim, we use the `colorcolumn` setting to determine which column(s) you want it to be. The `hi` setting determines which background color to use. The value `235` is dark gray, assuming you're using a dark theme. You want it to be subtle. If it's a high contrast color like bright red, it'll be obnoxious, and you'll likely shut it off. Vim uses the width of a column instead of just a thin line. Should you decide that you don't like it, commenting out these two settings will disable the display guide.

How long should your lines be? When you're using line wrapping, it doesn't matter as much. However, let's say you want your desired line length to be 80 characters. You might try setting colorcolumn to 80. However, you'll soon discover that if your content is exactly 80 characters, the last character will be sitting on the guide, which is probably not the desired outcome. Increase the colorcolumn value by 1 to be the first column in the "overflow area."

```
set colorcolumn=81
```

There are times when you may want to have more than one column. For example, your team's coding standard might say something like, "lines are recommended to be 80 characters long, and **must not** be longer than 120 characters." You'll be happy to know that Vim supports this. Use a comma to separate the desired column values.

```
set colorcolumn=81,121
```

Listing 3.

```
syntax on

" Pick a theme
colorscheme default

" Look for project specific config file
set exrc

" Ability to change buffers without saving first
set hidden

" Error settings
set noerrorbells
set novisualbell

" Briefly jump to a matching bracket, then come back
set showmatch

" Useful when you split your current buffer
set splitbelow
set splitright
```

User Experience Settings

The visual aspect of an editor is just the tip of the iceberg. So much lies just below the surface. These user experience settings are less visual in nature. While these settings sometimes have a visual aspect, they're more about how Vim behaves.

The `syntax on` ensures that syntax highlighting is enabled, while the `colorscheme` setting determines which theme should be used to syntax highlight the file. The use of `default` here is redundant. It's just a placeholder for the colorscheme you decide to use.

The `exrc` setting, when enabled, looks in the current directory for a vimrc file. Why would you want to do this? In a word, control. It is useful for creating project-specific configurations.

The `hidden` setting is another setting worth using. It allows you to change buffers without having to save them first. It can be quite annoying to have to save the buffer first—particularly if you want to jump between files quickly. Enabling the `hidden` setting is helpful in reducing friction in your developer experience.

There are two error settings related to how to notify a user when they've made an error. The `errorbells` setting is an auditory notification, while `visualbell` is a visual notification that causes the screen to flash. They've both been disabled, but try them out. See if there's one you prefer.

The `showmatch` setting is good for reminding you of the location of your matching brace when you insert it. Vim will jump you to your matching brace, then bring you back. This setting is separate from highlighting the braces and your ability to bounce between matching braces with the percent key.

Finally, we have two settings for splitting a window. The `splitbelow` setting is used when opening a horizontal split. You can create a split by running `:sp file.txt`. By default, it's not enabled, so this will place the new buffer above the current one. If this seems unnatural to you, having `splitbelow` enabled will do the right thing. The `splitright` setting is a corresponding setting for vertical splits. You can create a vertical split by running `:vsp file.txt`. By default, it's disabled—which places the new buffer to the left of your current buffer. Hopefully, you'll enjoy my preference for `splitbelow` and `splitright`, but if not, they are easy enough to change.

Content Settings

As developers, we're an opinionated group. We love our flame wars. Vim vs. Emacs is one of them. If you've read this far, you've already decided that Vim is the winner. Another famous one is tabs vs. spaces. While I'm not going to decide for you, I am going to discuss some indentation-related settings. Since you're reading this in PHP Architect, the PHP coding standard PSR-12 makes the most sense to use as a default.

Vim has multiple settings related to indentation, beginning with the `tabstop` setting. This setting determines how many spaces a tab character represents with a default value of 8 spaces. Vim isn't the only UNIX tool to use this value. The `less` command also uses a value of 8 spaces. I often thought it was a large value considering the lines should also be a maximum of 80 characters wide. So, why? Wikipedia says, "horizontal tab size of eight evolved because as a power of two it was easier to calculate with the limited digital electronics available." That seems like a reasonable answer for 1976 when Bill Joy released the first version of Vi. However, it makes much less sense when Vim was initially released in 1991 and even less when Neovim was released in 2015. It occurred to me that it's an affordance. It's meant to call out the structure of your program. That idea is supported by Linux kernel coding style documentation on indentation.

> *"In short, 8-char indents make things easier to read and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning."—Linux Kernel Documentation*[1]

Since the goal of this article is to provide a universal configuration, we're going to stick with 4 spaces since you might also want to write some prose-like documentation. You are keeping your documentation up-to-date, right?

The shiftwidth setting is typically a "set it and forget it" setting. People set the value equal to `tabstop`. That's what we did here. Did you know that if `shiftwidth` is set to zero, the `tabstop` value will be used? So they'll always be in sync, which could be useful if you need to update the value of tabstop while editing a document. I'm sure it can be useful to have them be different values. I've just never had the occasion to do so.

1 *Linux Kernel Documentation: https://phpa.me/linux-coding-style*

Listing 4.

```
" ~~~~ Content Settings ~~~~~~~

set tabstop=4
set softtabstop=4
set shiftwidth=4
set smartindent

" Convert tabs to spaces
set expandtab

set nojoinspaces
set textwidth=0
set wrapmargin=0
set nowrap
set linebreak

" Min no of screen lines to keep above and below the cursor
set scrolloff=8

set sidescroll=12
set sidescrolloff=12

" Determine how you want special characters to be displayed
set listchars=""
set listchars+=tab:>¬
set listchars+=eol:¶
set listchars+=trail:·
set listchars+=extends:»
set listchars+=precedes:«
set listchars+=nbsp:¤
```

Typically, when you've indented a line and you hit return, you also want the next line indented. That's where `autoindent` comes in. The `smartindent` is helpful when braces are used—for things like C-like languages, which includes PHP.

When you use >> you may expect the indentation to match the tabstop value. However, the smarttab setting affects this behavior. So it's disabled to provide this behavior. It's not a big deal at the moment since I've set the tabstop and shiftwidth to the same value. It's just something to keep in mind should you make them different values.

The final indentation setting—`expandtab`—is responsible for whether or not the tabs get converted (expanded) to spaces. Of course, we're going to use spaces. These are the indentation settings for a more civilized age.

Miscellaneous Content

Have you noticed that Vim puts 2 spaces after the period when you join two lines with `J`? Annoying, isn't it? Do people do that anymore? By setting `nojoinspaces`, that behavior will be disabled, and you'll go back to a single space after the period.

To wrap or not to wrap your content—that is the question. The answer lies with whether you're OK with line returns in your content. Wrapping provides you flexibility, and not wrapping offers you control. These settings ensure that you do want to wrap your text. If that's not what you want, it's easy to change with just a couple of minor tweaks.

The `textwidth` is one of those settings that will put line returns into your text. It helps limit the maximum number of characters in a line. When the line length reaches that textwidth value, it adds a line return. This config has it set to 0, which disables it.

The `wrapmargin` acts like a fallback when `textwidth` is disabled, except it uses the width of the window to determine where the line return will be added. This means that if you resize the width of the window as you write your text, you can have some inconsistent paragraph width. Using `textwidth` is the better way.

The `linebreak` provides some sanity to line wrapping. Without this enabled, your text will wrap the entire width of the window, at individual characters, and break in the middle of words. As you can guess, that is a terrible reading experience. The `linebreak` setting works in concert with another called `breakat` to determine where the lines should break. By default, this includes spaces, slashes, and hyphens.

There are 3 settings related to scrolling—scrolloff, sidescroll, and sidescrolloff—that determine how much content the user sees when the cursor approaches the edge of the window. The `scrolloff` setting determines the number of lines to display between the cursor and the top/bottom of the window.

The `sidescroll` determines the minimum number of columns, whereas `sidescrolloff` uses screen columns to display when scrolling horizontally. These settings will only matter when the `wrap` option is off.

The final miscellaneous content setting is listchars. Remember back in the day when you'd write term papers with WordPerfect? Inevitably, you'd have to turn on Reveal Codes to find out where the formatting went wrong. That's what `listchars` reminds me of whenever I switch it on. It shows you tabs, EOL, trailing spaces, and non-breaking spaces. Note how I've used the `+=` operator to append the `listchars` value. I prefer to write **multi-value** settings in this manner in the event that it's necessary to disable one of the values by commenting it out.

Search Settings

As you edit your file, eventually, you'll need to look for something. Vim's search is quite quick. Search, in this context, is confined to the current file. It's a basic substring match, just like a CMD+F in other editors. This is a nice set of settings to make search in Vim feel more natural.

Searching starts from your current position in the file. This works well with the `incsearch` setting. As you add characters to your search term, Vim searches incrementally, highlighting the first matching substring. If you're happy with the current match, press Enter to retain that position in the file. To find the next match, press **CTRL-g** to search forward (down the file) or press **CTRL-t** to search backward. If you need a way to remember, look down at your QWERTY keyboard and note that the T key is above the G key, making them `up` and `down`, respectively.

The `hlsearch` setting will highlight *every* match of your search term, which can be overwhelming if you have a lot of matches. The other thing about this is the matches will remain highlighted—even after performing other editing operations. To clear the matches, you need to run `:noh`. If you like this feature, you can enable it. It's disabled in this config, as the clarity of looking at a single match that you don't have to clear feels like a better user experience.

Are you sure the text you're looking for is ahead of you? If not, the `wrapscan` will be quite useful. When you reach the end of the file, Vim wraps to the other end of the file and continues searching for your term each time you press **CTRL-g** or **CTRL-t**.

Vim performs case-sensitive searches by default when you search for a term. If you don't know if there is any capitalization in the word you're looking for, this can be too limiting. So the `ignorecase` setting will provide a case-insensitive search when you type your search term in lowercase letters. The `smartcase` setting works together with `ignorecase`. If you use capital letters in your search, Vim will respect them and limit your search to only those matches, allowing Vim to be smart about the searches you make. Vim assumes that when you include capital letters, you know what you're doing. If you don't include them, Vim tries to be helpful. It's the best of both options.

Final Touches

We just went through many user experience-related settings. However, there are a couple of things you expect from an IDE to feel complete. The first thing is a file explorer. If you use PhpStorm or VSCode, it can feel necessary to have a file explorer visible at all times. However, it takes up valuable space—especially if you want to split your window to edit multiple files. What if I told you that it didn't always need to take up that valuable space? The other thing you'll probably want is a terminal *inside* of Vim. Sometimes you don't want to leave the editor just to execute a few commands. To help you implement these commands, we'll add a couple of variables and leader key mappings to the config file. Let's wrap these things up—shall we?

File Explorer

Vim provides a few different ways to explore files, each which its own command. However, `:Lexplore` makes a good option for the typical IDE user by placing the file explorer on the left edge of your window.

Let's begin with the file explorer. You'll notice it's off when Vim starts. We need a way to make it easy to turn it on and off. The best way to do that is to create a mapping using your leader key. We'll dive into the leader key more in the next article, but it's a single key that acts like a command prefix. Our leader key is the space bar. I chose the abbreviation `fe` to mean "file explorer," so if you change the `:Lexplore` to a different command, you can keep the mapping. It's nice when abstractions are flexible.

Add these settings to the "Leader Key Mappings" section of your vimrc.

```
let mapleader=" "

nnoremap <Leader>fe :Lexplore<CR>
```

By default, the file explorer will probably use 50% of the window—which I'm guessing is not what you want. Fortunately, we can tweak it with a couple of settings. That's what the two variables below do. Add them to the `Variables` section of the vimrc file. The netrw plugin is built into Vim. So I'm kinda cheating with the "no plugins" approach. It doesn't count, though, since it came with Vim.

```
let g:netrw_banner = 0
let g:netrw_winsize = 25
```

The `g:netrw_banner` setting disables a banner that appears at the top of the file explorer. When you're getting started with it, the banner is useful. However, it gets annoying quickly, so it's useful to be able to turn it off. The `g:netrw_winsize` determines the percentage of the window. A value between 25 and 30 is probably what you want. The value of 25 is used here to give more room for your content.

The Terminal

Sometimes you need a terminal in your IDE. PhpStorm has this built in. Vim should have it too, don't you think? It does. You can use a terminal in a horizontal or vertical split. Sure, you can always type the command. However, creating a couple of leader key mappings will make it faster and less error-prone to open a terminal. Just copy these settings to your vimrc.

```
nnoremap <Leader>th :terminal<CR>
nnoremap <Leader>tv :vertical terminal<CR>
```

Conclusion

We covered a lot in this article to create a universal vimrc that suits multiple types of users, whether you want to write code or prose. I did my best to inform you about the different settings to help you decide how to make it your own. For your convenience, you can access the entire vimrc on GitHub https://github.com/andrewwoods/universal-vim[2]. Future articles will make updates to this universal vimrc. So stay tuned! There's more Vim and Neovim goodness to come.

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

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

Broken Authentication

Eric Mann

One of the most foundational elements of security is clear communication. If we fail to use the correct language to communicate, we risk being misunderstood and making critical software mistakes.

Several years ago, I had the opportunity to travel to LA to meet face-to-face with a client. My team was building a strong, cryptographic authentication system for an embedded hardware component they had designed. We took an Uber from the airport and used the trip as an opportunity to talk shop. My colleague started asking about requirements our client had for the "crypto" components.

That was a mistake.

Our Uber driver was an *expert* in "crypto" and spent the rest of the ride giving us the rundown on how he'd liquidated his 401k to invest in Bitcoin and how we could do the same. It was an interesting experience, both in terms of operational security (don't discuss client projects in a car with a random stranger driving) and in terms of ambiguous terminology.

"Crypto" meant one thing to us. It meant a very different thing to him.

In 2021, OWASP redefined one of its top risks as *Identification and Authentication Failures*[1]. This risk was previously known as *Broken Authentication*[2] but expanded to include related identification failures that can lead to similar exploits. In most of those cases, the failure stems from an inability to identify which kind of "auth" a system is implementing—**authentication** or **authorization**.

Different Flavors of Auth

Most developers recognize the importance of "auth" within their application. At a very basic level, any multi-user application needs to discretely identify one user versus another. Usernames, email addresses, identification numbers—all of these are ways to describe uniqueness among many different users.

The fatal mistake many of us make, though, is confusing authentication with authorization. Just because you know who is logging into your app doesn't mean you immediately understand what they're allowed to do within it.

Last month[3], we clarified that authentication is *not* authorization when we covered multi-factor authentication. Specifically, your face (or other biometric) helps identify you to a system, but it does not prove your intent to perform an action within that system. You need to keep both concepts completely separate in your mind - authentication is for proving identity while authorization is for proving access or intent.

1 *Authentication Failures: https://phpa.me/owasp-id-auth-failures*

2 *Broken Authentication: https://phpa.me/owasp-A2-2017-broken-authentication*

3 *Last month: https://phpa.me/security-demystifying-mfa*

OpenID Connect and Authentication

Many modern applications rely on tools like OpenID Connect (OIDC)[4] for both authentication *and* authorization. With this system, a user authenticates not to your system but to some federated authentication provider. That provider returns to them a token that represents the user's identity as well as specific "scopes" for which the token is valid.

For example, assume you publish a website that powers a comprehensive movie database. Most users might only interact with the website anonymously—they pass by, view content, and never log in. Certain users, though, might log into the website using their Google credentials to edit content.

In this workflow, a user clicks a "Log in with Google" button on your website. Doing so pushes them over to Google, where they log in with a password you can't see and are given an OIDC token in exchange. This token references their identity according to Google and carries a signature stating that Google has verified that identity.

This token alone means a user is logged in. But there's currently no notable difference between you logging in with Google or me using my own Gmail account to do the same.

Scopes and Authorization

One approach to handling authorization is to define custom "scopes" within which a token is valid. On our movie website, some users might be able to edit movies or actors. Others might be able to delete content. Yet others are able to manage users.

Each of these actions is represented by a specific action against a particular resource. For example, `movie:edit` or `actor:create` or `user:delete`. Within our application, we want to validate that a given user has a specific scope before performing an associated action.

To do so, the user must request a token from our identity provider (Google, in this example) for that scope. If you log in and request the `movie:edit` scope, Google will check that

4 *OpenID Connect (OIDC): https://openid.net/specs/openid-connect-core-1_0.html*

Listing 1.

```php
function checkAuthentication(string $token, array $scopes): void
{
    global $idp;

    try {
        $idp->validateToken($token, $scopes);
    } catch (Exception $error) {
        error_log('Auth failed: ' . $error->getMessage());
    }
}
```

you have this permission before adding it as a "claim"[5] to the resulting token.

Within your application, it's then important to check two things:

- Is the signature on the token valid (according to a public key published by Google)?
- Does the token contain a valid claim on the scoped action the user is trying to perform?

Failing to do either of these will invalidate the security of your application.

In the Wild

Authentication is hard. Authorization is even harder. Many apps leverage built in systems for role-based access control or permissions lists, but these can easily become complicated and difficult to manage. Leveraging a third-party identity provider for authentication *drastically* simplifies an application stack, but it is only part of the story.

The code in Listing 1, based on something I've actually seen in the wild, was configured to properly check both the signature on and contents of an authentication token. If the signature were invalid or the required scope(s) were missing on the token, the identity provider implementation would throw an `Exception`. The only problem was that the `try`/`catch` immediately threw that exception away rather than triggering an error.

The result: valid tokens passed the check, and the application ran. Invalid (or even `null`) tokens failed the check, but the *application still ran*. It's an easy mistake to make; this kind of security lapse is exactly the reason why we delegate authentication to third-party platforms. It's also a strong way to ensure *authorization* is properly cataloged outside of error-prone code.

It's still remarkably easy to break auth, regardless of the flavor of auth we're using.

5 as a "claim": https://phpa.me/scopeclaims

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

Blueprinting our Application

Joe Ferguson

This month we're exploring a programming concept known as Scaffolding. Scaffolding is the process of using an application to generate code based on some input or configuration file. Just as construction workers raise temporary structures to help perform their work, we can expect similarly. Automatically generating code is a wonderful idea but questions always arise with "how good is the code?"

I was first introduced to the idea of scaffolding[1] parts of your application via the `bake` command with CakePHP[2]. Even back in 2014, frameworks were generating ready-to-run code. You would use `bake` in a similar fashion to how we use `Artisan` or `Symfony` to create controllers, models, and more. Scaffolding is now a foundational part of many large frameworks but the code may not be as feature-rich as you'd like. The Blueprint[3] package adds a mountain of scaffolding options to our application. Blueprint can create Models, Controllers, Routes, Form Requests, and Tests for our Laravel applications.

Blueprint uses a `draft.yaml` file as the configuration file. We'll use YAML to describe what we want Blueprint to build for us. We can jump directly into a fresh Laravel & Blueprint project by running the following commands:

```
$ composer create-project \
        laravel/laravel fresh
$ cd fresh
$ composer require --dev \
        laravel-shift/blueprint
$ touch draft.yaml
```

We'll start by adding two model definitions with fields specifying what we want on each model. See Listing 1. We'll also use `relationships:` to instruct Blueprint to automatically create the model relationships for us.

Listing 2 shows how we can use `php artisan blueprint:build` to ready our `draft.yaml` configuration and generate our files.

We now have our two models created, `app/Models/Game.php` and `app/Models/GameTag.php`, as well as migrations and database factories to support our models. Here's the moment of truth, how *good* is the code Blueprint generates for us? Let's take a dive into `app/Models/Game.php` shown in Listing 3.

Listing 1.

```
models:
  Game:
    name: string index
    slug: string index
    site_url: string
    publisher_url: string
    publisher: string
    description: longtext
    thumbnail_url: string
    screen_url: string
    published: boolean
    relationships:
      hasMany: GameTag
  GameTag:
    name: string
    game_id: id foreign:games
```

Listing 2.

```
$php artisan blueprint:build
Created:
- database/factories/GameFactory.php
- database/factories/GameTagFactory.php
- database/migrations/1973_01_22_153133_create_games_table.php
- database/migrations/1973_01_22_153134_create_game_tags_table.php
- app/Models/Game.php
- app/Models/GameTag.php
```

Listing 3.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\
     Factories\HasFactory;
use Illuminate\Database\Eloquent\
     Model;

class Game extends Model
{
    use HasFactory;

    protected $fillable = [
        'name',
        'slug',
        'site_url',
        'publisher_url',
        'publisher',
        'description',
        'thumbnail_url',
        'screen_url',
        'published',
    ];

    protected $casts = [
        'id' => 'integer',
        'published' => 'boolean',
    ];

    public function gameTags()
    {
        return $this->hasMany(
               GameTag::class);
    }
}
```

1 scaffolding: https://w.wiki/5Rz5

2 CakePHP: https://phpa.me/cakephp-code-gen-bake

3 Blueprint: https://github.com/laravel-shift/blueprint

We can see Blueprint has added each of our fields into the `fillable()` array to support mass assignment elsewhere in our controller. We also see the `$casts` array ensures our `id` is an `integer` type, and our `published` field is a `boolean`, which will represent if the game has been published or released. Blueprint also created the `gameTags()` method for us, which defines our `hasMany` relationship to the `GameTag` model.

In our `GameTag` model (Listing 4), we have a belongsTo[4] relationship configured, and it is represented by the `game()` method returning `$this->belongsTo(Game::class)`, which allows us to easily reference which tag belongs to which game. This is a basic tagging solution that solves the problem for our application. To use a more complex relationship configuration with Blueprint, refer to the documentation[5] describing how to configure model relationships.

Doesn't `artisan:make model Game` do the same thing? The power that Blueprint brings to our project is the ability to explicitly describe not only the names of our model classes but also to populate the contents of the files with our fields and relationship information. Blueprint generates ready-to-use code we can immediately run using `php artisan migrate` to test out our generated migrations.

```
$ php artisan migrate
Migrating: 2022_07_09_200447_create_games_table
Migrated:  2022_07_09_200447_create_games_table (53.67ms)
Migrating: 2022_07_09_200448_create_game_tags_table
Migrated:  2022_07_09_200448_create_game_tags_table (21.86ms)
```

Using MySQL on the command line in Figure 5, we can see our tables have been created:

We can see from `describe games;` that the games table has the columns we specified as well as the `updated_at`, `created_at` columns for Eloquent. We also see our indexes applied to the `name` and `slug` columns. We can inspect our `game_tags` table and see our columns in Listing 6.

Blueprint does a great job creating model classes, and I'm now ready to dive into my application. What about the migrations? We can inspect our `games` table migration (Listing 7), and the code looks nearly identical to what you'd see in the official Laravel documentation.

One area of code generated by Blueprint that we'll want to review and improve is the Model Factories. No library is able to assume intent and understand our application's domain logic, so it's up to us to visit our model factories and review the faker content. See Listing 8.

Once we have tweaked the Faker library methods (Listing 9), we can get a better representation of our production data.

How Does This Magic Work?

If you haven't already realized, the magic under the hood is using stubs and using them as templates for the contents

4 belongsTo: https://phpa.me/laravel-one-to-many-inverse

5 documentation: https://phpa.me/laravelshift-model-relationships

Listing 5.

```
mysql> show tables;
+------------------------+
| Tables_in_fresh        |
+------------------------+
| failed_jobs            |
| game_tags              |
| games                  |
| migrations             |
| password_resets        |
| personal_access_tokens |
| users                  |
+------------------------+
mysql> describe games;
+---------------+-----------------+------+-----+---------+
| Field         | Type            | Null | Key | Default |
+---------------+-----------------+------+-----+---------+
| id            | bigint unsigned | NO   | PRI | NULL    |
| name          | varchar(255)    | NO   | MUL | NULL    |
| slug          | varchar(255)    | NO   | MUL | NULL    |
| site_url      | varchar(255)    | NO   |     | NULL    |
| publisher_url | varchar(255)    | NO   |     | NULL    |
| publisher     | varchar(255)    | NO   |     | NULL    |
| description   | longtext        | NO   |     | NULL    |
| thumbnail_url | varchar(255)    | NO   |     | NULL    |
| screen_url    | varchar(255)    | NO   |     | NULL    |
| published     | tinyint(1)      | NO   |     | NULL    |
| created_at    | timestamp       | YES  |     | NULL    |
| updated_at    | timestamp       | YES  |     | NULL    |
+---------------+-----------------+------+-----+---------+
```

Listing 4.

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class GameTag extends Model
{
    use HasFactory;

    protected $fillable = [
        'name',
        'game_id',
    ];

    protected $casts = [
        'id' => 'integer',
        'game_id' => 'integer',
    ];

    public function game()
    {
        return $this->belongsTo(Game::class);
    }
}
```

Listing 6.

```
mysql> describe game_tags;
+------------+-----------------+------+-----+---------+
| Field      | Type            | Null | Key | Default |
+------------+-----------------+------+-----+---------+
| id         | bigint unsigned | NO   | PRI | NULL    |
| name       | varchar(255)    | NO   |     | NULL    |
| game_id    | bigint unsigned | NO   | MUL | NULL    |
| created_at | timestamp       | YES  |     | NULL    |
| updated_at | timestamp       | YES  |     | NULL    |
+------------+-----------------+------+-----+---------+
mysql> describe games;
+---------------+-----------------+------+-----+---------+
| Field         | Type            | Null | Key | Default |
+---------------+-----------------+------+-----+---------+
| id            | bigint unsigned | NO   | PRI | NULL    |
| name          | varchar(255)    | NO   | MUL | NULL    |
| slug          | varchar(255)    | NO   | MUL | NULL    |
| site_url      | varchar(255)    | NO   |     | NULL    |
| publisher_url | varchar(255)    | NO   |     | NULL    |
| publisher     | varchar(255)    | NO   |     | NULL    |
| description   | longtext        | NO   |     | NULL    |
| thumbnail_url | varchar(255)    | NO   |     | NULL    |
| screen_url    | varchar(255)    | NO   |     | NULL    |
| published     | tinyint(1)      | NO   |     | NULL    |
| created_at    | timestamp       | YES  |     | NULL    |
| updated_at    | timestamp       | YES  |     | NULL    |
+---------------+-----------------+------+-----+---------+
```

Listing 7.

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateGamesTable extends Migration
{
    public function up()
    {
        Schema::create('games', function (Blueprint $table) {
            $table->id();
            $table->string('name')->index();
            $table->string('slug')->index();
            $table->string('site_url');
            $table->string('publisher_url');
            $table->string('publisher');
            $table->longText('description');
            $table->string('thumbnail_url');
            $table->string('screen_url');
            $table->boolean('published');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('games');
    }
}
```

of the destination files. If you're curious, we can inspect `vendor/laravel-shift/blueprint/stubs/model.class.stub`, which is the stub file used to render or create our models based on the values we've configured in our YAML. See Listing 10.

These stub files allow Blueprint to build out fully functional code for our application based on our configuration.

Blueprinting "Real" Code

Generating models and migrations is all fun and games, but what about controllers and requests? Do we have to write our own tests for all this generated code? Don't worry; Blueprint also has test generation covered as well. We've started with Game and GameTag models, so we'll want to add the ability to submit new game information to be saved in our database. We'll start by expanding our `draft.yaml` configuration by adding a `controllers:` section as shown in Listing 11.

We want to create a controller for our `Game` model that has an `index` method to return a list of all the games we have in the database. We'll want to continue our configuration using `create` and `store` methods to describe how we want our methods to function and what fields we want to validate in our Form Request. To apply our changes, we'll use `php artisan blueprint:build` again and get an update of the files created, and we now have an update to our `routes/web.php` to route to our new `GameController` class.

The blueprint build command also builds a `GameStoreRequest` Form Request class to use to validate the list of fields we specified. We also have data being flashed to the redirect when we store a new instance of our model. Form Requests allow us to abstract all of our validation logic into another class and keep it from bloating our controller methods. Lastly, we have a `destroy` method that allows the deletion of a Game instance.

Listing 8.

```php
public function definition()
{
    return [
        'name' => $this->faker->name,
        'slug' => $this->faker->slug,
        'site_url' => $this->faker->word,
        'publisher_url' => $this->faker->word,
        'publisher' => $this->faker->word,
        'description' => $this->faker->text,
        'thumbnail_url' => $this->faker->word,
        'screen_url' => $this->faker->word,
        'published' => $this->faker->boolean,
    ];
}
```

Listing 9.

```
public function definition()
{
    return [
        'name' => $this->faker->name,
        'slug' => $this->faker->slug,
        'site_url' => $this->faker->url,
        'publisher_url' => $this->faker->url,
        'publisher' => $this->faker->company,
        'description' => $this->faker->text,
        'thumbnail_url' => $this->faker->imageUrl(240,240),
        'screen_url' => $this->faker->imageUrl(1280,720),
        'published' => $this->faker->boolean,
    ];
}
```

One of my favorite features of Blueprint is generating tests as well as the controller class. Before we can run the tests, we'll want to install a package that supports the custom asserts used by Blueprint via:

```
$ composer require --dev jasonmccreary/laravel-test-assertions
```

We can run the tests with PHPUnit.

We can open `app/Http/Controllers/GameController.php` and take a look at the `index()` method so we understand what's happening and verify that the test is sufficient for our needs.

Our method queries the database for all Game models and returns the `game.index` view (located at `resources/views/games/index.blade.php`) with the data returned in `$games`. We should expect our test to perform a request and verify our method works as expected.

We can inspect our `tests/Feature/Http/Controllers/GameControllerTest.php` class to ensure Blueprint isn't doing anything silly like `assertTrue(1);`. As expected, we see a well-formatted and thought-out test class that contains the basic exercises for generated code.

Listing 11.

```
controllers:
  Game:
    index:
      query: all
      render: game.index with:games
    create:
      render: game.create
    store:
      validate: name, slug, site_url, publisher_url,
               publisher, description,
               thumbnail_url, screen_url, published
      save: game
      flash: game.name
      redirect: game.index
    destroy:
      delete: game
      redirect: game.index
```

Listing 10.

```
<?php

namespace {{ namespace }};

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class {{ class }} extends Model
{
    use HasFactory;
}
```

Our test in the example `GameControllerTest` class contains an `index_displays_view()` method to verify our index page loads the view. We're going to create a Game factory using the `database/factories/GameFactory.php` that was created for us based on our model's fields. The test creates three instances of our model and then creates a request to our game index route (Using named routes). Then assertions are made that the response code was successful, the view in the response matches the view template we expect, and finally that the

Listing 12.

```
<?php

namespace App\Http\Controllers;

use App\Http\Requests\GameStoreRequest;
use App\Models\Game;
use Illuminate\Http\Request;

class GameController extends Controller
{
    public function index(Request $request)
    {
        $games = Game::all();

        return view('game.index', compact('games'));
    }

    public function create(Request $request)
    {
        return view('game.create');
    }

    public function store(GameStoreRequest $request)
    {
        $request->session()->flash('game.name', $game->name);

        $game = Game::create($request->validated());

        return redirect()->route('game.index');
    }

    public function destroy(Request $request, Game $game)
    {
        $game->delete();

        return redirect()->route('game.index');
    }
}
```

view contains a `games` object containing the games from our database previously created by the factory.

Conclusion

Controllers, Models, and Tests! (oh my!) Before you get *too* excited, let's remember we haven't written *any* code in our views. Blueprint created view stubs for us, but it's up to us to build our frontend and form to return for the `create()` method from our controller. We don't want to forget about the quality of what we're building. Blueprint isn't going to print our application for us. It's a tool that allows us to create parts of our application more rapidly, including PHPUnit (or Pest) test classes. This library is a superpower for anyone just getting started with Laravel as it generates well-formatted code following industry standards and examples.

Listing 13.

```
$ php vendor/bin/phpunit
PHPUnit 9.5.21 #StandWithUkraine

....... 7 / 7 (100%)

Time: 00:00.498, Memory: 32.00 MB

OK (7 tests, 17 assertions)
```

I hope our quick tour of using Blueprint to generate code for your application has inspired you to check it out in your own projects. Blueprint is the tool I reach for when I want to not only quickly generate ready-to-use code but also experiment with different options to find which solves the problems best. When you generate code, you don't have the time investment in writing everything yourself. Blueprint helps get even more of the boring stuff out of your way so you can focus on solving problems for your clients.

Listing 14.

```
/**
 * @param \Illuminate\Http\Request $request
 * @return \Illuminate\Http\Response
 */
public function index(Request $request)
{
    $games = Game::all();

    return view('game.index', compact('games'));
}
```

Listing 15.

```
<?php

namespace Tests\Feature\Http\Controllers;

use App\Models\Game;
use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\\WithFaker;
use JMac\Testing\Traits\AdditionalAssertions;
use Tests\TestCase;

/**
 * @see \\App\\Http\\Controllers\\GameController
 */
class GameControllerTest extends TestCase
{
    use AdditionalAssertions, RefreshDatabase, WithFaker;

    /**
     * @test
     */
    public function index_displays_view()
    {
        $games = Game::factory()->count(3)->create();

        $response = $this->get(route('game.index'));

        $response->assertOk();
        $response->assertViewIs('game.index');
        $response->assertViewHas('games');
    }
```

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](https://twitter.com/JoePFerguson)

Related Reading

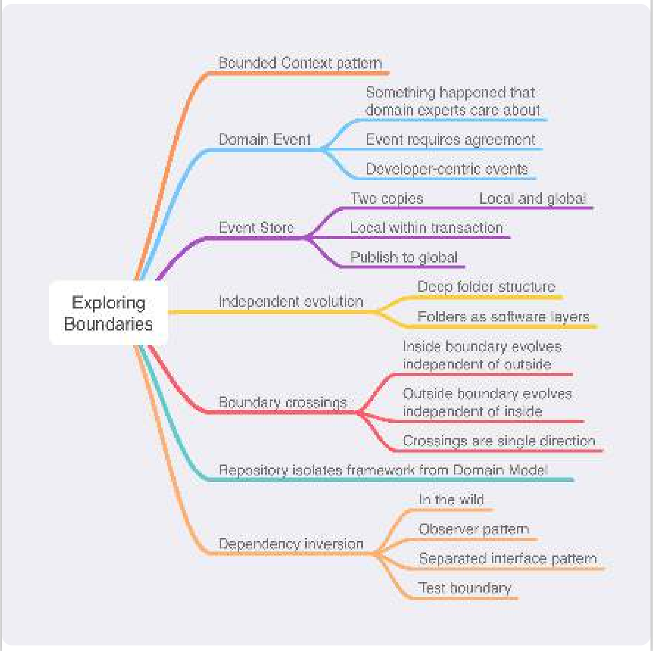
- *The Workshop: PHP Development with Homestead in WSL* by Joe Ferguson, October 2020.
  http://phpa.me/workshop-oct-20
- *The Workshop: Refactoring to an Object Store* by Joe Ferguson, April 2021.
  http://phpa.me/workshop-apr-21
- *Boosting User Perceived Performance with Laravel Horizon* by Scott Keck-Warren, August 2021.
  http://phpa.me/boost-performance

Exploring Boundaries

Edward Barnard

Last month "Structure by Use Case" introduced the Bounded Context pattern we'll be using over and over as we structure our software by use case. This month we're diving more deeply into the boundaries, the crossing points, and the software layers created by this pattern.

Figure 1.



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

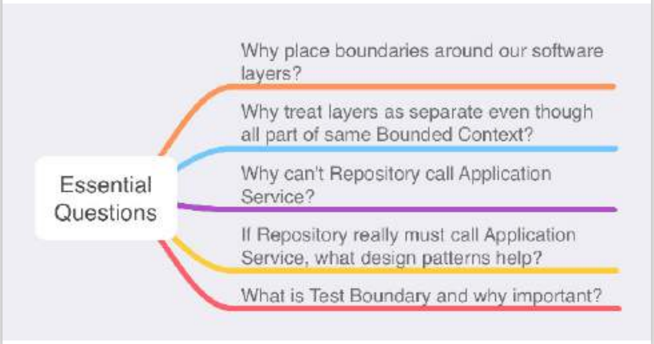
Essential Questions

Upon surviving this article, you should be able to answer the following questions (Figure 2):

- Why is it important to place boundaries around our software layers?
- Why do we treat Application Service, Repository, Domain Model, and Factory as separate layers even though they are all part of the same Bounded Context?
- Why don't we allow the Repository to call the Application Service?
- If the Repository really does need to call the Application Service, what design pattern(s) can help?
- What is the Test Boundary and why is it important?

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

Figure 2.


Essential Questions
Why place boundaries around our software layers?
Why treat layers as separate even though all part of same Bounded Context?
Why can't Repository call Application Service?
If Repository really must call Application Service, what design patterns help?
What is Test Boundary and why important?

New Bounded Context

The Domain Event is a fundamental Domain-Driven Design (DDD) pattern. Eric Evans, in *Domain-Driven Design Reference: Definitions and Pattern Summaries*, page 17, has the definition:

> *Something happened that domain experts care about.*

Vaughn Vernon, in *Implementing Domain-Driven Design*, page 285, hints at the difficulty awaiting the unwary:

> *Use a domain event to capture an occurrence of something that happened in the domain. This is an extremely powerful modeling tool. Once you get the hang of using Domain Events, you will be addicted and wonder how you survived without them until now. To get started with them, all you have to do is find agreement on what your Events actually are.*

Do you see the problem? All we have to do is find an agreement! Instead, we're doing an alternative. We'll be creating events that we care about as developers. This gives us the infrastructure so that if an agreement is ever found, it will be easy to drop-in support of "real" Domain Events.

We'll be implementing the event store Vernon describes on page 304 as option 3 (condensed for clarity):

> *Create a special storage area (a database table) in the same persistence store that is used to store your domain model. This is an Event Store; this storage area is not owned and controlled by your messaging mechanism but instead by your Bounded Context.*
>
> *An out-of-band component that you create uses the Event Store to publish all stored, unpublished Events through the messaging mechanism. This has the advantage that your model and your Events are guaranteed to be consistent within a single, local transaction.*

From a database standpoint, we have two copies of the event. The local store receives a record I'm calling the Application Event. The global store stores that record as a Domain Event. We're building Domain Event support first. Note that the local store is what Vernon calls the Event Store; in Vernon's terms, those events are then published to our global Domain Event store. We're building the latter infrastructure first.

Where should we put the code? The global event store should probably sit outside any particular Bounded Context. That is, it should be its own Bounded Context.

The local event store should also sit outside any particular Bounded Context. What if the local event store turns out to be specific to a certain Bounded Context? We don't, at this point, know if that's the case or not. It's easier to make it separate now and fold it in later, as compared to trying to extract it from some other Bounded Context later.

Another way to look at the question is this. When I'm adding a feature and don't have any particular guidance as to what constitutes the appropriate Bounded Context, I take a "fine-grained" approach. It doesn't really hurt to make a bunch of smaller bounded contexts, even if we choose to combine them into a larger structure later. In fact, we plan to move things around (i.e., refactor and restructure the codebase) as we gain deeper insight into the business. It's much easier to move several small folders than to extract and move parts of a large folder.

I created the new folder `src/BoundedContexts/Infrastructure` for code supporting multiple Bounded Contexts. We've already seen one usage. We saw that the traits for loading framework Model classes are within `src/BoundedContexts/Infrastructure/LoadTableModels`. See Figure 3.

Independent Evolution

We'll now create the new folder `Events` within `Infrastructure`. We're perfectly free to subdivide as deeply as we care to go. In this case, we're creating the Bounded Context `AppEvent` with its folder structure and Bounded Context `DomainEvent` with its similar folder structure. See Figure 4.

Figure 3.

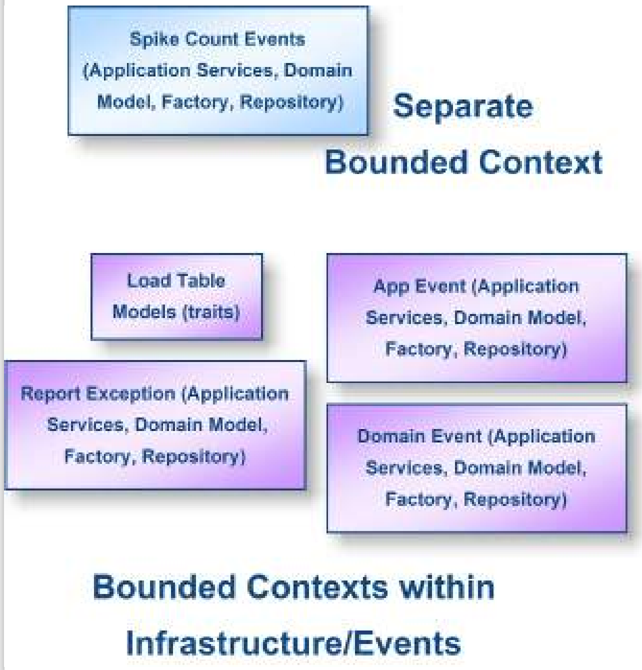
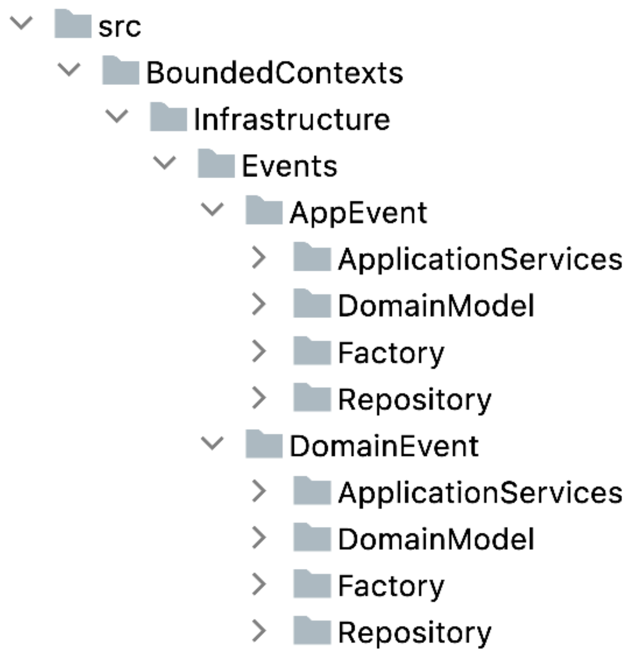

Spike Count Events (Application Services, Domain Model, Factory, Repository)
Separate Bounded Context
Load Table Models (traits)
App Event (Application Services, Domain Model, Factory, Repository)
Report Exception (Application Services, Domain Model, Factory, Repository)
Domain Event (Application Services, Domain Model, Factory, Repository)
Bounded Contexts within Infrastructure/Events

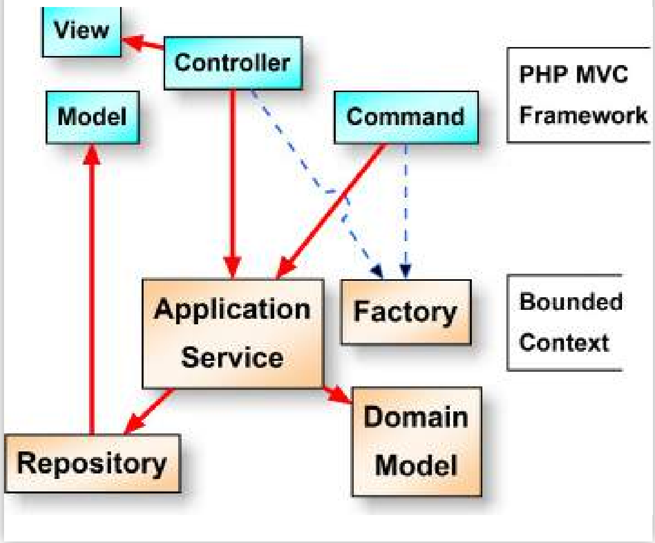
Figure 4.



Our structure is deceptively simple. Don't let it fool you. Each folder represents a software layer[2] of the architecture. You have created a boundary between what's inside the folder and what's outside the folder.

2 *layer: https://en.wikipedia.org/wiki/Layer_(object-oriented_design)*

Figure 5.



Our folder structure creates a separation of concerns[3] which itself is a good thing. But there's something far more important here directly related to Strategic Domain-Driven Design.

Think of each of these folders shown in Figure 4 as a container with a boundary as in Figure 5.

Use some sort of explicit mechanism when crossing the boundary, such as:

- Use the factory to obtain the needed object.
- Use the public methods without making any assumptions as to how the internals work.
- Use an interface defining what public methods are available.

Your "boundary-crossing protocol" doesn't need to be formal or complex. You don't need to create a PHP `interface` for every boundary to be crossed. Nor do you need to create a Factory for every little thing. The point here is to create the boundary and *respect* the boundary in some way.

What does that boundary gain us?

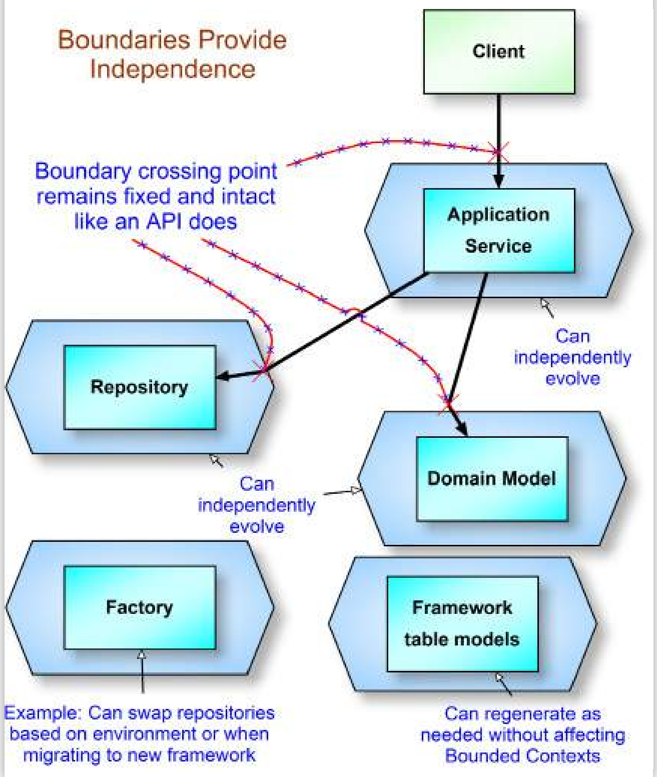
- Everything inside the boundary can freely evolve without respect to anything outside the boundary.
- Everything outside the boundary can also freely evolve without affecting anything inside the boundary.

So long as the crossing point remains intact, everything on one side can shift without affecting anything on the other side.

Here's an example. Suppose that, for whatever reason, we changed a column name in one of our tables in the database. It's a column currently being used, and the name change breaks the code using that column.

Let's further assume that we remain on excellent terms with the database administrator. She is willing to take these two initial steps for us:

Figure 6.


Boundaries Provide Independence
Client
Boundary crossing point remains fixed and intact like an API does
Application Service
Repository
Can independently evolve
Domain Model
Can independently evolve
Factory
Framework table models
Example: Can swap repositories based on environment or when migrating to new framework
Can regenerate as needed without affecting Bounded Contexts

- Alter the table to create the new column name with default null.
- Update each table row to set the new column's value to the same as the old column's value.

At that point, in our development environment, we can re-bake the framework table models to pick up the new column name and characteristics. Note that, in effect, our table models are inside a boundary of their own. We can regenerate the models any time we want without affecting anything else.

Next, we can update any Repository classes that know about the old column name to use the new column name instead. There's no need to change the Repository's API. So long as the repository maps the new column name so that the repository's output continues to look the same, we've kept the boundary crossing the same. In other words, any class using that repository will continue to achieve the same result.

The Repository has hidden our database change within the Repository. There's no domino effect of cascading dependencies. We can allow the Repository to evolve as our data design evolves without affecting any of the higher-level layers (use cases, business rules, etc.).

Once our Repository changes (and regenerated table models) have reached production, our database administrator can drop the obsolete table column.

The same "boundary crossing" strategy applies when we see opportunities for refactoring. See Figure 6. Keep each small step (a refactoring[4] that changes the code without changing

3 *separation of concerns: https://phpa.me/wikip-concerns*

4 *refactoring: https://phpa.me/refactoring*

the code's observable behavior) within a boundary, and we need not worry about affecting anything on the other side of the boundary.

The Dependency Inversion Principle

Brett L. Schuchert wrote a super-useful article based on his experience with the Dependency Inversion Principle in the Wild[5]. After describing the principle itself, he explains:

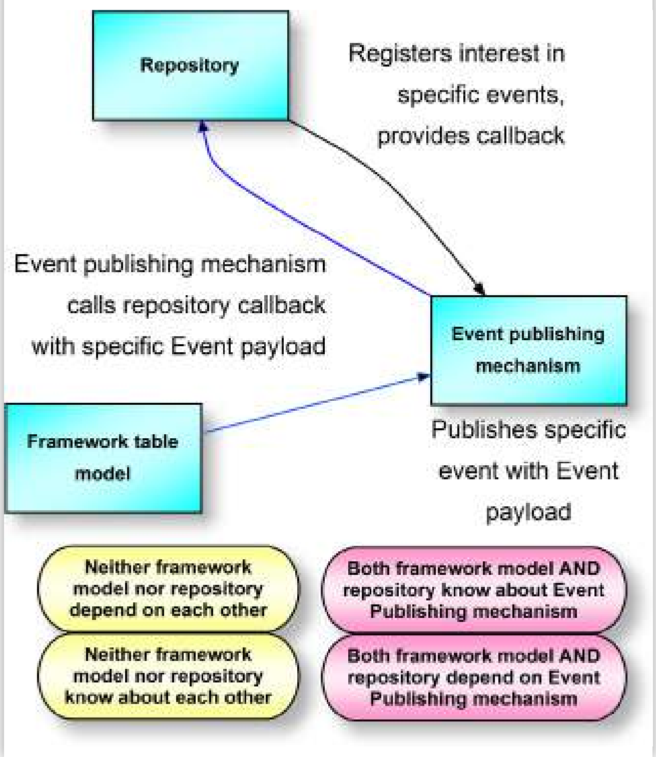
> *The common thread throughout all of these is about the view from one part of your system to another; strive to have dependencies move towards higher-level (closer to your domain) abstractions.*

In other words, when we consider crossing a boundary, remember that the crossing point is limited to one-way traffic. A response can come back the other direction, to be sure, but requests only flow in one direction.

For example, the use case handler, the Application Service, makes requests of its Repository. However, the Repository does not make requests of the Application Service.

The Repository invokes the framework table models, but the framework table models do not invoke the Repository. However, the framework table models come with a caveat.

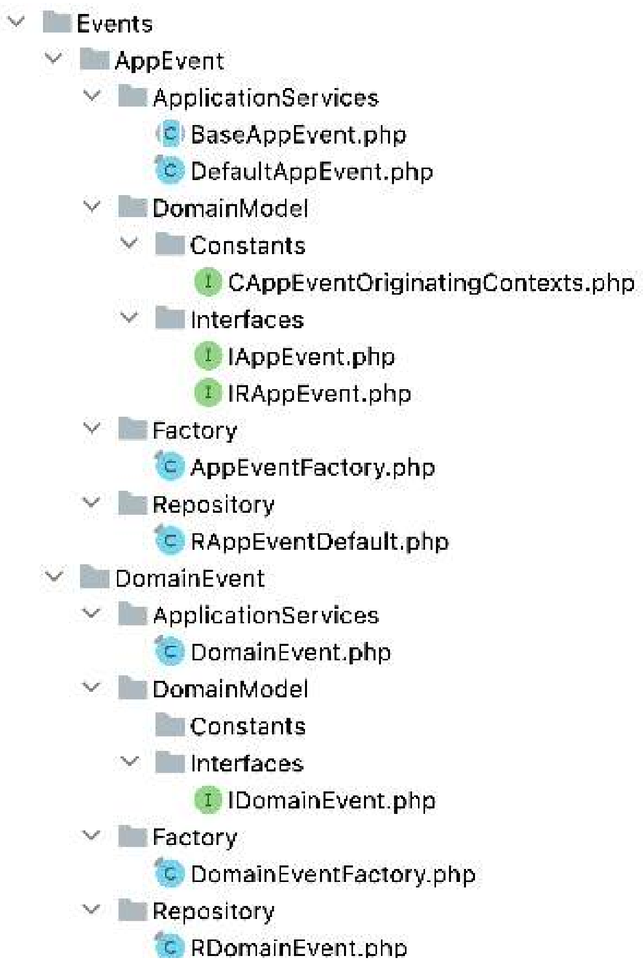
Figure 7.


Repository
Registers interest in specific events, provides callback
Event publishing mechanism calls repository callback with specific Event payload
Event publishing mechanism
Framework table model
Publishes specific event with Event payload
Neither framework model nor repository depend on each other
Both framework model AND repository know about Event Publishing mechanism
Neither framework model nor repository know about each other
Both framework model AND repository depend on Event Publishing mechanism

They can, in fact, invoke the Repository because the models publish events using the Observer pattern[6]. The Repository can register to become an Observer. With this pattern, both the Repository and framework model know about the event publishing mechanism, which is an example of the Dependency Inversion Principle at work. See Figure 7.

Figure 8 shows the individual class files we'll be placing in those folders. Within the Domain Model, we have a file (a PHP `interface`) containing class constants, `CAppEventOriginatingContexts.php`. We also have two traditional interface files (containing methods and no constants), `IAppEvent.php` and `IRAppEvent.php`. Note that all three files are within the Domain Model boundary.

Figure 8.



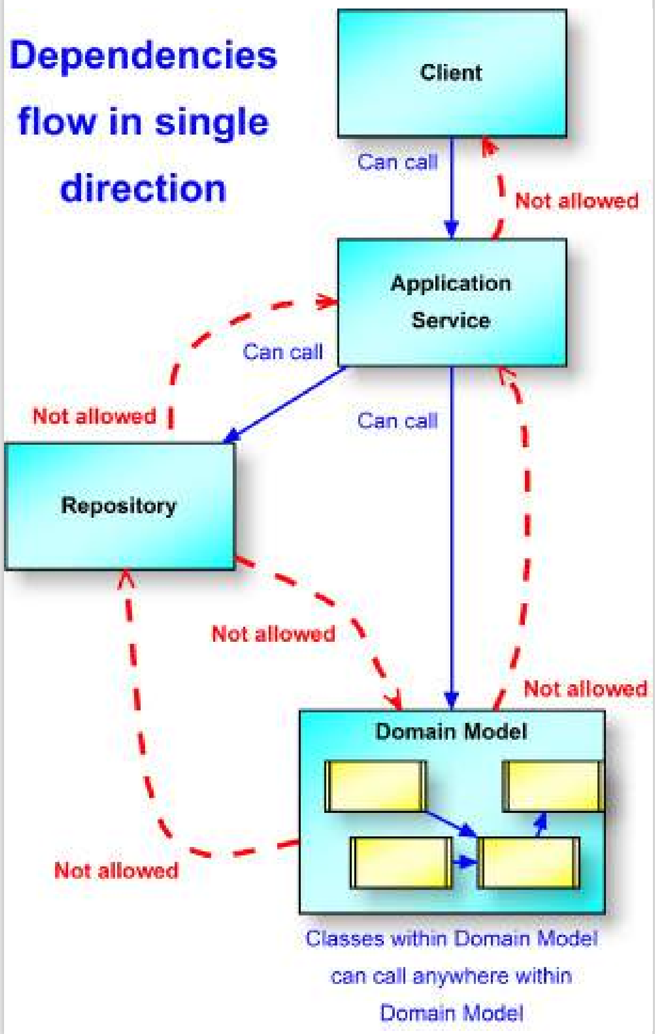
```
Events
  AppEvent
    ApplicationServices
      BaseAppEvent.php
      DefaultAppEvent.php
    DomainModel
      Constants
        CAppEventOriginatingContexts.php
      Interfaces
        IAppEvent.php
        IRAppEvent.php
    Factory
      AppEventFactory.php
    Repository
      RAppEventDefault.php
  DomainEvent
    ApplicationServices
      DomainEvent.php
    DomainModel
      Constants
      Interfaces
        IDomainEvent.php
    Factory
      DomainEventFactory.php
    Repository
      RDomainEvent.php
```

Note the direction of dependencies in Figure 9:

5 *DI Principle in the Wild:* https://phpa.me/di-wild

6 *Observer pattern:* https://en.wikipedia.org/wiki/Observer_pattern

Figure 9.

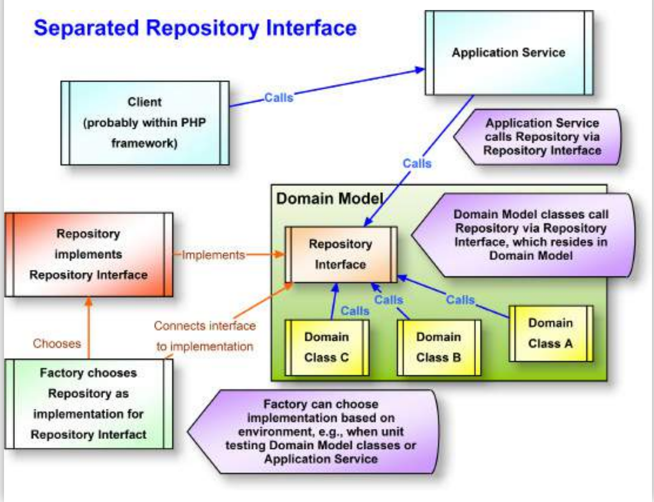


- Application Services can make use of the Repository but not the other way around.
- Application Services can make use of the Domain Model but not the other way around.
- The Repository can make use of the framework models but not the other way around.
- The Repository can make use of the Domain Model but not the other way around.
- The Domain Model can make use of anything within its own Domain Model boundary.
- The Factory is a special case in that it gathers together knowledge of all available boundary crossings and how the pieces fit together.

Martin Fowler explains what we're doing as the "Separated Interface" pattern on page 476 of *Patterns of Enterprise Application Architecture* (Figure 10):

> *As you develop a system, you can improve the quality of its design by reducing the coupling between the system's parts. A good way to do this is to group the classes into packages and control the dependencies between them. You can then follow rules about how classes in one package can call classes in another–for example, one that says classes in the domain layer may not call classes in the presentation package.*
>
> *However, you might need to invoke methods that contradict the general dependency structure. If so, use Separated Interface to define an interface in one package but implement it in another. This way a client that needs the dependency to the interface can be completely unaware of the implementation.*

Figure 10.


Separated Repository Interface
Application Service
Client (probably within PHP framework)
Calls
Application Service calls Repository via Repository Interface
Calls
Domain Model
Repository implements Repository Interface
Implements
Repository Interface
Domain Model classes call Repository via Repository Interface, which resides in Domain Model
Calls
Calls
Calls
Domain Class C
Domain Class B
Domain Class A
Connects interface to implementation
Chooses
Factory chooses Repository as implementation for Repository Interfact
Factory can choose implementation based on environment, e.g., when unit testing Domain Model classes or Application Service

One of Fowler's examples of when to use it is (p. 478):

> *You have some code in one layer that needs to call code in another layer that it shouldn't see, such as domain code calling a Data Mapper.*
>
> *For applications I recommend using a separate interface only if you want to break a dependency or you want to have multiple independent implementations.*

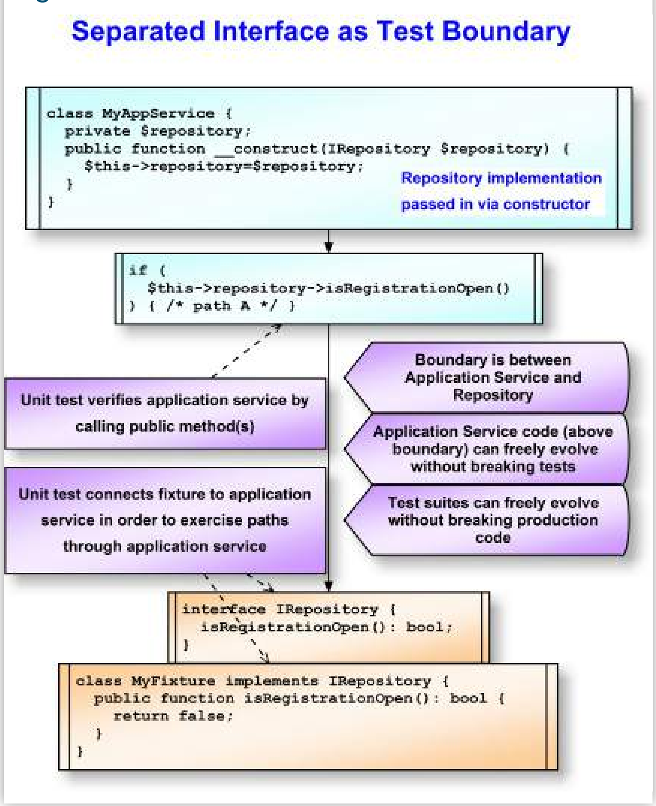
Test Boundary

There is a non-obvious dependency that sometimes needs to be broken. Robert C. Martin explains on page 252 of *Clean Architecture: A Craftsman's Guide to Software Structure and Design* that unit tests are often written to be closely coupled to the system under test.

For example, when I write PHP class A, which contains public methods a, b, and c, I normally write unit tests that directly call `a()`, `b()`, and `c()` in class A. That pretty much destroys my ability to refactor class A as needed. Class A has become answerable to my unit tests as a client.

When this issue becomes a problem, I create a separate interface, even if the interface will only be used by the test suite. The interface then locks down what boundary crossings must remain supported regardless of the underlying implementation.

Figure 11.



```
class MyAppService {
  private $repository;
  public function __construct(IRepository $repository) {
    $this->repository=$repository;
  }
}
```

```
if (
  $this->repository->isRegistrationOpen()
) { /* path A */ }
```

```
interface IRepository {
  isRegistrationOpen(): bool;
}
```

```
class MyFixture implements IRepository {
  public function isRegistrationOpen(): bool {
    return false;
  }
}
```

Separated Interface as Test Boundary
Repository implementation passed in via constructor
Unit test verifies application service by calling public method(s)
Unit test connects fixture to application service in order to exercise paths through application service
Boundary is between Application Service and Repository
Application Service code (above boundary) can freely evolve without breaking tests
Test suites can freely evolve without breaking production code

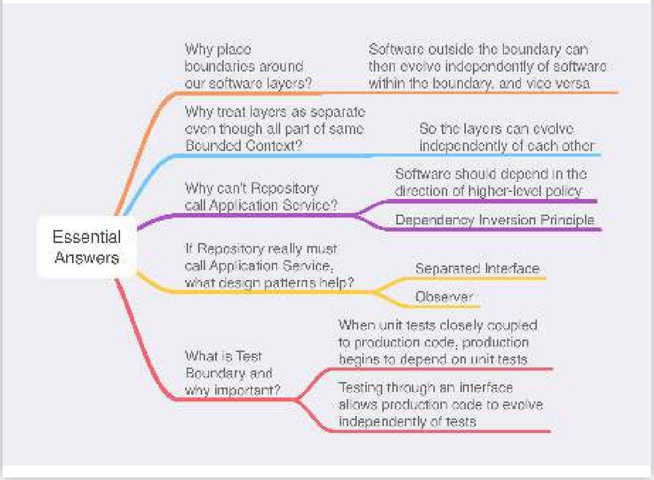
For me, this issue is fairly rare. When I want to do some refactoring, but unit tests will break if I do, that is the time to consider testing through an interface. See Figure 11.

Essential Questions Answered

See Figure 12.

- *Why is it important to place boundaries around our software layers?* Software within the boundary can evolve without the risk of breaking anything outside the boundary. Software outside the boundary can evolve without breaking anything inside the boundary.
- *Why do we treat Application Service, Repository, Domain Model, and Factory as separate layers even though they are all part of the same Bounded Context?* So that they can evolve independently of each other.
- *Why don't we allow the Repository to call the Application Service?* We strive to have dependencies move toward higher-level abstractions - the Dependency Inversion Principle.
- *If the Repository really does need to call the Application Service, what design pattern(s) can help?* Both the Separated Interface and Observer patterns break dependencies flowing in the wrong direction.
- *What is the Test Boundary, and why is it important?* When unit tests are closely coupled to production code, they force the production code to be more rigid. The production code, in effect, depends on the unit tests. Testing through an interface allows the production code to evolve independently of the unit tests.

Figure 12.


Essential Answers
Why place boundaries around our software layers?
Software outside the boundary can then evolve independently of software within the boundary, and vice versa
Why treat layers as separate even though all part of same Bounded Context?
So the layers can evolve independently of each other
Why can't Repository call Application Service?
Software should depend in the direction of higher-level policy
Dependency Inversion Principle
If Repository really must call Application Service, what design patterns help?
Separated Interface
Observer
What is Test Boundary and why important?
When unit tests closely coupled to production code, production begins to depend on unit tests
Testing through an interface allows production code to evolve independently of tests

Summary

What is a Bounded Context, and how big should it be? The official answer is, "it depends on the context." That's not terribly helpful. Instead, in the PHP code, we just set boundaries where they make sense at the time. We can shift things later as we gain a deeper understanding. I personally found this a very difficult question at first, and it prevented me from getting started.

Just do it! Create a folder somewhere called `BoundedContexts` and create a folder inside it to declare your first Bounded Context. The top-level folders within your Bounded Context declare the layers of your architecture within that Bounded Context. The pattern we've shown here includes Application Services, the Domain Model, Factory, and Repository. You're free to structure things differently, but this is the pattern we'll use with our examples.

Each layer, and each Bounded Context as a whole, has boundaries. That boundary is protective. The boundary allows everything within the boundary to evolve independently of anything outside the boundary and vice versa. In fact, it's even possible to decouple unit tests from the production code being tested, allowing the production code to become more abstract and general while the unit tests become more concrete and specific.

Communication between layers should pass in a single direction. An application service can use its Repository, but the Repository should not be making calls to the application service. The Dependency Inversion Principle and the Separated Interface pattern relate to working with the needed direction of dependencies.

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

Using Git to Your Advantage

Chris Tankersley

In the last article, "What is Git Doing?" we explored the details of how Git[1] works. Now we will see how we can rewrite history and time using Git.

For anyone needing a refresher, Git is a decentralized, network-capable version control system for files which means that all of the work that you do and tracking for changes to files is done locally, on your machine. Still, Git can work with other copies of the repository over various network protocols to share that information. Most importantly, Git can reconcile those changes between multiple people and make sense of all of these different changes to form a coherent change history.

Much of this power comes from exactly how Git itself works. Complete copies of the files are stored when changes occur, and Git keeps track of changes using a parent reference model. This means that any individual commit only references and cares about where it came from. Branches can be created, merged, or destroyed without much overhead on the repository.

All of this, taken together, grants the user a vast amount of power. Users can manipulate this history to "clean up" messes and work more efficiently. The downside to this is what can be, on the surface, a very complicated and scary-looking set of tools that can make your life as a developer very hard if you do not understand what the tools do.

Alternate History with Worktrees

Git allows for a large amount of experimentation with editing files. At a basic level, you can edit a bunch of files, and if whatever you are working on does not work, you can just `git restore` those files to their previous versions. Git also allows you to quickly create a branch off of your current version of files, and you can experiment in a branch.

As you start to work on multiple branches, you will probably come across the command known as `git stash`. This command allows you to store (stash) a set of changes and reverts the files to their previous version. Doing so is extremely useful when you may edit a file locally for debugging, but you do not want those changes committed to the final product. You can stash those changes away and return them later with `git stash pop`.

Stashing works for small changes, but what happens when you have a large set of changes you need to track? Stashing also has a blind spot when it comes to untracked files and handling files that have moved around. Git has a solution for this—worktrees.

Worktrees are a linked, separate copy of your repository. Each worktree, while they are in a separate folder, knows of and interacts with each other as if they are part of a single repository. They provide much more flexibility when working with branches compared to making multiple clones of your repository.

Let's say you are working on a new feature and encounter a bug. Instead of incorporating the bug fix as part of your new feature, you want to push the bug fix out before your feature is finished, but you are knee-deep in working on your new feature. You could commit all this partial code as a commit and create a new branch, or you could use a worktree.

Creating a new worktree is the cleaner solution. The `git worktree add` command will take a branch name, folder location, and a pointer to a commit (be that an actual commit hash, tag, or branch name). Let's create a new branch called `bugfix/fix-file-reads`, put it in a folder located at `~/Projects/fix-file-reads`, and base it on our `main` branch.

```
$ git worktree add -b bugfix/fix-file-reads \
                 ~/Projects/fix-file-reads main

Preparing worktree (new branch 'bugfix/fix-file-reads')

HEAD is now at af207a5 Updated composer support for PHP
 8.1 (#478)
```

Git takes care of creating the new folder and branch for us. We can go into that new folder, do all of our work, and follow our processes like normal. For example, once we are done with the hotfix, we can push it up to something like GitHub and submit a pull request. The nice thing about this is that, short of this being done in a different folder, there is no change to how you handle your code.

As I mentioned before, all of the worktrees are linked together. So that `bugfix/fix-file-reads` branch is also available in your original checkout. You can merge it like any other branch, and if you make any other branches in your main checkout, those branches are available to the other worktrees.

Worktrees do add an additional maintenance task on the user's end. They take up space and should be cleaned up every so often.

1 Git: *https://git-scm.com*

You can check and see which worktrees are available using `git worktree list`, which shows the following:

```
/home/ctankersley/Projects/test-project  0c62e8c [newfeature]
/home/ctankersley/Projects/fix-file-reads      0c62e8c [bugfix/fix-file-reads]
/home/ctankersley/Projects/array-oob      0c62e8c [bugfix/array-out-of-bounds]
/home/ctankersley/Projects/web-ui      0c62e8c [feature/web-ui]
```

Once you are finished with a worktree, you can remove it with `git worktree remove <path>`. So if we are finished with our `feature/web-ui` worktree, we can get rid of it with:

```
$ git worktree remove /home/ctankersley/Projects/web-ui
```

If you manually delete a folder that contained a worktree, you can run `git worktree prune` to delete worktrees that no longer are accessible.

Using worktrees helps keep a lot of my daily work much cleaner. I have one project which has three versions that get worked on simultaneously—the mainline code, a beta version, and a rewrite. I keep three worktrees going so that I can not spoil any one instance with accidental code. If I bugfix something in the main code, I also have to update the beta version. Since each worktree can still see the other worktrees and branches, it makes it much easier to incorporate code from one tree to another.

Rewriting Time with `git Rebase`

Worktrees are a great way to keep concurrent histories while working on them, but what happens when you need to change what was in history? That is where rebasing comes into play.

The core idea of rebasing is rewriting the history of a repository by changing the order in which changes are merged and therefore changing what commits exist in a branch's timeline. You are directly manipulating individual commits after they have been added to the history.

There tend to be two camps of developers—those that consider the git timeline a historical representation of what happened in your repository. Any individual change, including all the interstitial, commits when building complex features or commits fixing typos from other commits, should be kept. The accuracy of that timeline is incredibly important, and therefore rebasing should be kept to a minimum and potentially not happen at all.

The other camp considers the timeline a "story," as the git website describes it[2]. All of the messy, internal steps that it took to create your application are not important, and it is better to keep atomic and clean changesets. Other developers do not care that you misspelled a variable name; all that matters is that the variable name is correct now.

I will fully admit I fall into that second camp. I find that at the end of the day, all I want to have to dig through is finished code, not backtrack through a bunch of partially complete commits. Rebasing allows me to work at my own pace and commit when I need to from a work standpoint but deliver a single, cohesive pull request.

Cleaner Pulls

Code changes all the time, especially on the mainline branches like `main`. You need to keep your copy of this in sync with the remote copies, and one of the most common ways is via `git pull`.

`git pull` copies down the remote changes and stores them off into their own branch and then merges that remote branch with yours. In most cases, you are probably not changing the `main` branch, so you end up with what is known as a "fast forward" merge. Git will recognize you have nothing new locally and just point your `main` branch at the same thing as the remote `main`.

Sometimes you may get a situation where this creates a merge commit, and you merge the remote `main` branch into your local `main` branch. This merge commit pollutes the timeline with useless "Merged main" entries.

You can avoid this by using `git pull --rebase origin main` to pull down the remote `main` branch and then rebase your local `main` on the remote version pulled down. Rebasing when pulling will ensure that there are now additional merge commits created, even if you have added commits to your local `main` branch.

I find this keeps these important branches cleaner, and making new branches from a clean `main` that 100% matches the remote `main` leads to fewer merge conflicts down the road.

Cleaner Updates from the Base Branch

One of the selling points of Git is the ease of making branches, and a typical workflow is to create a branch when working on changes. If your change takes a while to complete, you run into an issue where the branch you started from, like `main`, changes while you are still working. You may need the changes that now exist in `main`, or you may have changed some of the same files. It's now time to bring in those changes.

Let's say you have a local copy of a centrally stored repository on GitHub. It contained two commits, with hashes `21420248` and `a9cad80d`. Remember, hashes reference individual commits, so we can verify that these two commits are the same in both repositories. We branch after `a9cad80d` and make our first commit at `c23d5765`.

2 *Rebasing: https://git-scm.com/book/en/v2/Git-Branching-Rebasing*

In the meantime, someone adds a new time feature and merges it. That means the remote/GitHub branch now has two new commits: `a7ec8a44` and `244426b0`.

Now we could just switch back to main, do a `git pull`, and then go back to our branch and `git merge main` to get the updates, but we get what I consider an unclean change history:

```
// Remote/GitHub
244426b0 2 hours ago - Merged branch feature/time-feature
a7ec8a44 3 days ago - Added Time class
a9cad80d 1 month ago - Deleted unneeded dependency on Vendor\Baz
21420248 1 month ago - Initial Commit

// Local on our own branch
c23d5765 10 hours ago - Updated dependencies
a9cad80d 1 month ago - Deleted unneeded dependency on Vendor\Baz
21420248 1 month ago - Initial Commit
```

Since all of the changes were clean, we see the `Added Time Class` merge gets put before our commit, and we end up with two merge commits at the end. As time goes on and more things get merged into `main`, you will end up with more and more merge commits in your own branch. From the perspective of the feature, all of this is unneeded information.

A cleaner way is to rebase your changes on top of the newest copy of `main`. You can accomplish this by doing a `git rebase main` instead of `git merge main`, like above.

```
// Remote/GitHub
244426b0 2 hours ago - Merged branch feature/time-feature
a7ec8a44 3 days ago - Added Time class
a9cad80d 1 month ago - Deleted unneeded dependency on Vendor\Baz
21420248 1 month ago - Initial Commit

// Local on our own branch
63ac1c56 Now - Merged branch main
244426b0 2 hours ago - Merged branch feature/time-feature
c23d5765 10 hours ago - Updated dependencies
a7ec8a44 3 days ago - Added Time class
a9cad80d 1 month ago - Deleted unneeded dependency on Vendor\Baz
21420248 1 month ago - Initial Commit
```

What Git does is pull off the changes you made, in this case, `c23d5765`, and appends them to the newest version of `main`, which is currently at `244426b0`. Your changes are merged with `244426b0`, which creates a new commit, `cbc1eff2`, with a timestamp of "now."

In this example, we cleaned up the history of changes by removing that additional merge from `main`. If your branch is long-lived, this may cut out a bunch of additional commits like merges that don't actually pertain to your changes. Since your changes are appended to the new "base" branch, all your commits also stay grouped together in a cohesive unit.

Why did your commit hash change? It does not just put your old commit at the end, as commits are based on full sets of files, and the underlying files have changed. Rebasing re-merges the changes together, and you end up with a new tree and, therefore, a new commit hash.

```
// Remote/GitHub
244426b0 2 hours ago - Merged branch feature/time-feature
a7ec8a44 3 days ago - Added Time class
a9cad80d 1 month ago - Deleted unneeded dependency on Vendor\Baz
21420248 1 month ago - Initial Commit

// Local on our own branch
cbc1eff2 now - Updated dependencies
244426b0 2 hours ago - Merged branch feature/time-feature
a7ec8a44 3 days ago - Added Time class
a9cad80d 1 month ago - Deleted unneeded dependency on Vendor\Baz
21420248 1 month ago - Initial Commit
```

Rebasing will not reduce the possibility of merge conflicts. If you change the same file as someone else, you will still need to figure out the best fix. But it may make resolving the conflict easier as your changes are being applied to the final version of the conflicted file, so you normally only end up making one conflict resolution per file.

Cleaning Up History

I use rebasing every single day to clean up the overall history before I push code up for a review. Most developers are familiar with a git history that looks like this:

```
// Local on our own branch

2202910e 2 days ago - Fixed phpstan errors
7eaec88b 2 days ago - Updated header to new logo
475c4c0a 3 days ago - Fixed unit tests
3c76e322 3 days ago - Typo fix
f523a40d 3 days ago - Fixed bug in Service Manager
cbc1eff2 4 days ago - Updated dependencies
244426b0 4 days ago - Merged branch feature/time-feature
a7ec8a44 8 days ago - Added Time class
a9cad80d 1 month ago - Deleted unneeded dependency on Vendor\Baz
21420248 1 month ago - Initial Commit
```

I have a total of six commits, from `cbc1eff2` to `2202910e`, including a few where I fixed a typo, then I fixed unit tests, then phpstan errors. Since I should not have misspelled that typo, and I should have run my unit tests and phpstan, I want to get rid of those commits. I do not want to get rid of work. I just get rid of the commits. At the end of the day, those commits do not provide any additional context or information about my changes.

For these types of changes, you can use an interactive rebase, called via `git rebase -i <hash>`. Doing so tells Git to find the starting commit of `<hash>` and allows us to manipulate the changes *after* that hash. Let's get rid of that typo commit by combining it with `f523a40d`.

We kick off the rebase with:

```
git rebase -i cbc1eff2
```

We chose this commit because it is right before the first commit we want to manipulate, which is `f523a40d`. We do not supply `f5223a40d` as the hash because then we would not be able to modify it.

Git will then supply us with a list of hashes we can manipulate as well as ask us what we want to do with them. See Listing 1.

Listing 1.

```
pick 2202910e Fixed phpstan errors
pick 7eaec88b Updated header to new logo
pick 475c4c0a Fixed unit tests
pick 3c76e322 Typo fix
pick f523a40d Fixed bug in Service Manager
pick b2b99fb Updated dependencies
# Rebase 2202910e..b2b99fb onto cbc1eff2 (6 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the ...
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous...
# f, fixup <commit> = like "squash", but discard this...
```

The first six lines are the commits we can manipulate. The first word of each line is a command, which the commented area at the bottom will detail. I have listed the most common five things you can do. The second column is the commit hash, and the rest of the line is the commit message. This screen allows us to queue up *what* we want to do and will execute it when we save and close this file.

What commands do we normally use?

- `pickup` - Just use the commit as is
- `reword` - Use the commit, but change the commit message
- `edit` - Use the commit, but stop and allow us to amend it with further changes
- `squash` - Use the commit, but merge it and the commit message with the previous commit
- `fixup` - Like squash, but ignore the commit message

We want to create a single unit of work for adding the post, but we do not want to lose the last three commits. We have two options—squash or fixup. Since I want to combine the typo commit with its parent, we change `3c76e322` to be "fixup" instead of "pick." We will edit the lines to look like this:

```
pick 2202910e Fixed phpstan errors
pick 7eaec88b Updated header to new logo
pick 475c4c0a Fixed unit tests
fixup 3c76e322 Typo fix
pick f523a40d Fixed bug in Service Manager
pick b2b99fb Updated dependencies
```

Once we save this command file, Git will run through and apply the commands to each commit. If we look at the log after it is finished, we can see that the typo fix commit is gone:

You can also see that the hash for the bug fix changed and is now `495c9b12`, and all the other hashes after that changed

```
0597e76e A moment ago - Fixed phpstan errors
5e17ff34 A moment ago - Updated header to new logo
58cc8c75 A moment ago - Fixed unit tests
495c9b12 A moment ago - Fixed bug in Service Manager
cbc1eff2 4 days ago - Updated dependencies
244426b0 4 days ago - Merged branch feature/time-feature
a7ec8a44 8 days ago - Added Time class
a9cad80d 1 month ago - Deleted unneeded dependency on Vendor\Baz
21420248 1 month ago - Initial Commit
```

as well. This is because changing an existing commit changes the file tree, which, when saved, creates a new commit. All the later commits are based on a new commit parent and new tree, so we get new commit hashes. These are all new commits, so they also get new timestamps with updated creation times.

If we do another interactive rebase to also fold in the unit tests into `495c9b12`, the same thing will happen. A new file tree is created because the contents of the files change, and all the subsequent commits get rebuilt as new commits because they once again have a new parent commit and file tree.

The Dangers of Rebasing Like This

The above types of rebasing are safe as long as you keep in mind one rule:

NEVER REBASE A BRANCH THAT SOMEONE ELSE HAS CHANGED

Git functions by keeping track of commits and their parents. When you rebase, you create new commits and new parents that don't exist in other copies of the repository. You will notice if you pull down a branch like `main`, rebase it, and then attempt to push it back up; it will complain that the branches are out of sync.

The branches are out of sync as Git can match the same hashes, but as soon as it hits the rebase point and new commits, it does not know what to do. It expects history to match, and because you changed it with a rebase, it actively stops you. Your only recourse is to "force push" a commit, which forces the remote repository to match yours.

Doing so now creates problems for other people using that branch. Their history is now out of sync with the canonical "central" copy. While they can pull down the new copy and reset everything so they match, they may lose commits from their local copy that are hard to recover. If two people are working on a feature branch, this can cause a real loss of code.

As long as you are the only one working on a branch, it is safe to rebase. If other people are working on a branch, never rebase.

Dig Into Advanced Git

You can do a lot if you dig into the power of Git. There are additional tools like the reflog, which is a local copy of all changes ever made that you can use to pull back lost commits. There are ways to completely delete a change or file from a repository as if it never existed. There are more workflows that can help speed up development.

I highly suggest digging through the git Book[3] and the Git Tools section[4]. You can find out how to track down bugs, better ways to search through code, and a whole bunch of other things.

The best thing about Git is that it is designed around experimentation. Play around with the commands and see what helps you work better and more efficiently.

3 *git Book: https://git-scm.com/book/en/v2/*

4 *Git Tools section: https://phpa.me/git-tools-revision*

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

Related Reading

- *Education Station: What is Git Doing?* by Chris Tankersley, July 2022. https://phpa.me/2022-06-education
- *Security Corner: Radical Transparency* by Eric Mann, May 2021. https://phpa.me/security-may-21
- *The Workshop: Git Hooks with CaptainHook* by Joe Ferguson, December 2020. https://phpa.me/2020-12-workshop

New and Noteworthy

PHP Releases

PHP 8.2.0 (Beta 1 available for testing):

https://www.php.net/archive/2022.php#2022-07-21-1

PHP 8.1.8 (Released):

https://www.php.net/archive/2022.php#2022-07-07-3

PHP 8.0.21 (Released):

https://www.php.net/archive/2022.php#2022-07-07-1

News

Serverless PHP Applications on Digital Ocean Functions

Digital Ocean Functions is a serverless compute product Digital Ocean recently launched. It supports PHP as one of the runtimes, along with Node.js, Python, and Go.

https://php.watch/articles/php-serverless-digital-ocean

Longhorn PHP Call For Papers

Call For Papers Now Open! Submissions are accepted until August 11, 2022, at 11:59 pm Central

https://cfp.longhornphp.com

What do you think CSPRNG in PHP

An interesting discussion happening on PHP Internals

https://externals.io/message/118265

Introducing a fresh new look for Artisan

There is an open PR on Laravel's repo (laravel/framework) which will include a ground-up redesign of the artisan output. The author of the PR is Nuno Maduro, who also updated the design of the route:list already. This PR is a logical continuation of that.

https://phpa.me/laravelmagazine-artisan-new-look

A Week of Symfony #812

This week was the most intense in Symfony development activity in months. We completed and merged tens of new features and improvements for the upcoming Symfony 6.2 version, including: a Doctrine entity argument resolver, options to sort files in case insensitive mode and by extension and size, console autocompletion for zsh shells, security logout improvements and new options, etc.

https://phpa.me/symfony-awos-24-07-2022

How to Install Multiple PHP Versions on Ubuntu 22.04

We will use the Ondrej PPA for installing PHP on Ubuntu 22.04 LTS system. Which contains PHP 8.1, 8.0, 7.4, 7.3, 7.2. 7.1, 7.0 & PHP 5.6 packages. You can install any of the versions as required for your application. The new application developers are suggested to use the latest PHP version, i.e., PHP 8.1.

https://phpa.me/devdojo-mult-php-ubuntu

PHP version stats: July 2022

It's that time again: my biyearly summary of which PHP versions are used across the community. I know I'm a little early, but that's because I had some spare time today and wanted to make sure I got it ready in time. You can read the January edition here.

https://stitcher.io/blog/php-version-stats-july-2022

Drupal 7.91, 9.3.19, and 9.4.3 released with several critical security fixes

Drupal, a popular content management system written in PHP, released three new versions on all of the current version series fixing several security vulnerabilities. These releases include fixes for information disclosure, code execution, access bypass, and cross-site scripting vulnerabilities found in earlier versions.

https://php.watch/news/2022/07/drupal-7-9-security-releases

Decimals to Fractions

Oscar Merida

In this installment, we look again at working with floating-point values. We approach how to take a decimal number and represent it as a fraction.

Recap

Given any floating point number, write a function that returns a string representing the number as a fraction in its simplest form. For example, if the decimal is 0.8, the function should return the string "4/5".

Parts of a Fraction

Fractions have two parts, a numerator (the number above the fraction divider) and a denominator (the number below the divider). The numerator is a count of the equal parts of the whole, while the denominator is the total number of parts available.

So, if we have:

```
  7  (numerator)
----
 12  (denominator)
```

We're saying we've taken 7 units of the 12 that are available.

Decimals to Fractions

Decimals are another way of writing fractions, where the denominator is 10, 100, 1000, or another power of ten, depending on our precision. The number we see after the decimal point is the numerator. We can infer the denominator based on how many digits are in it.

The equivalent of `0.025` is:

```
  25
----
1000
```

We could stop there, but fractions are usually reduced to their lowest terms. And we'd simplify the fraction above to

```
  1
---
 40
```

By reviewing what we know about fractions, we've identified the steps of our solution

1. Given a floating-point number with `N` digits, write it as a numerator and denominator.
2. Reduce the fraction to its lowest terms.

One simplification we'll make to our problem is to assume we'll always work with a number greater than 0 and less than 1. If we have a floating point value that is greater than one, we'd first have to break it into the whole number and fractional parts. I'll leave that part to you, dear reader.

Decimal to Fraction

We can use PHP's type juggling to our advantage by treating our decimal as a string. Then we can count the characters after the decimal point.

```php
$input = 0.8;
$parts = explode('.', $input);
$count = strlen($parts[1]);
var_dump($count); // (int)1
```

That's straight-forward, and we've got our numerator and denominator in one step.

```php
$numerator = (int) $parts[1];
$denominator = pow(10, $count);
var_dump($numerator); // (int) 8
var_dump($denominator); // (int) 10
```

So far so good, what edge cases should we consider? What if our input has trailing zero's like `0.4500`?

```php
$input = 0.4500;
$parts = explode('.', $input);
$count = strlen($parts[1]);
$numerator = (int) $parts[1];
$denominator = pow(10, $count);
var_dump($numerator); // (int) 45
var_dump($denominator); // (int) 100
```

Perfect, if we give it a float, the PHP engine cleans it up for us. What if we use a string of `"0.4500"`? Our code gives the following output:

```php
var_dump($numerator); // (int) 4500
var_dump($denominator); // (int) 10000
```

It's not wrong, but the extra zeros are pointless and could lead to issues later. We can use type hints for a function to coerce values into a float. Or to be strict and only accept floats as input. While we're at it, let's make a basic class to represent a function. I have a feeling it'll be useful in the next step. Listing 1 shows our class.

I appreciate how PHP 8 has streamlined constructors via property promotion. If you're not using this yet, give it a try. It reduces boilerplate for defining a class, even if it's unfamiliar at first. Named arguments are also useful for writing clear code. If I were working on PHP 8.1, I'd go so far as to make the properties read-only so that the `Fraction` class is immutable.

We use it as shown here:

```
$frac = Fraction::fromFloat("0.4500");
var_dump($frac);
```

Which does the conversion for us and gives this output:

```
object(Fraction)#1 (2) {
  ["numerator"]=>
  int(45)
  ["denominator"]=>
  int(100)
}
```

Simplifying a Fraction

Another elementary maths review is in order. To simplify a fraction, we have to find the greatest common factor. That is, we want the largest factor that both integers have in common. Didn't we figure out how to find the factors for an integer before? Yes, we did; in February's Puzzle Corner[1], we wrote a function to list them for an integer. Listing 2 reproduces our solution here.

If we can find the factors for each, we can use PHP's array functions to find the largest they have in common. Our `findFactors()` function returns an array of pairs of factors. We're interested in all the factors, so we'll need to flatten the array. I modified it to return a one-dimensional array of the factors it finds, as shown in Listing 3.

Finding the factors is the trickiest part. We can use `array_intersect()`[2] to find all the ones in common, sort that in ascending order, and pop the largest value to find the Greatest Common Divisor. That part looks like this:

```
$numeratorFactors = findFactors($numerator);
$denominatorFactors = findFactors($denominator);
$commonFactors = array_intersect($numeratorFactors,
                          $denominatorFactors);
print_r($commonFactors);
$GCF = array_pop($commonFactors);print_r($GCF);
```

Listing 1.

```
<?php

class Fraction
{
    public function __construct(
        public int $numerator,
        public int $denominator,
    ) {}

    public static function fromFloat(float $input) : self
    {
        $parts = explode('.', $input);
        $count = strlen($parts[1]);

        return new self(
            numerator: (int) $parts[1],
            denominator: pow(10, $count)
        );
    }
}
```

Listing 2.

```
<?php

function findFactors(int $product) : array {
    $factors = range(1, (int) sqrt($product));
    $factors = array_map(
        function($factor) use ($product) {
            if ($product % $factor === 0) {
                return [$factor, $product / $factor];
            }
        }, $factors
    );

    return array_filter($factors);
}
```

Listing 3.

```
<?php

/**
 * @return int[]
 */
function findFactors(int $product) : array {
    $factors = range(1, (int) sqrt($product));
    $flat = [];

    array_walk($factors,
        function($factor) use ($product, &$flat) {
            if ($product % $factor === 0) {
                $flat[] = $factor;
                $flat[] = $product / $factor;
            }
        }
    );

    sort($flat);
    return array_unique($flat);
}
```

1 *February's Puzzle: https://phpa.me/puzzles-finding-integer-factors*

2 *array_intersect(): https://php.net/array_intersect*

Listing 4.

```php
<?php namespace Puzzles;

class Fraction
{
  public function __construct(
    public int $numerator,
    public int $denominator,
  ) {}

  public static function fromFloat(float $input) : self {
    $parts = explode('.', $input);
    $count = strlen($parts[1]);

    return new self(
      numerator: (int) $parts[1],
      denominator: pow(10, $count)
    );
  }

  public function __toString() : string {
    return $this->numerator . '/' . $this->denominator;
  }

  public function simplify() : self
  {
    $numeratorFactors = findFactors($this->numerator);
    $denominatorFactors = findFactors($this->denominator);

    $commonFactors = array_intersect($numeratorFactors,
               $denominatorFactors);
    $GCF = array_pop($commonFactors);

    if ($GCF > 1) {
      return new self(
        $this->numerator / $GCF,
        $this->denominator / $GCF,
      );
    }

    return $this;
  }

  /**
   * @return int[]
   */
  private function findFactors(int $product) : array {
    $factors = range(1, (int) sqrt($product));
    $flat = [];

    array_walk($factors,
      function($factor) use ($product, &$flat) {
        if ($product % $factor === 0) {
          $flat[] = $factor;
          $flat[] = $product / $factor;
        }
      }
    );

    sort($flat);
    return array_unique($flat);
  }
}
```

For `0.8`, the common factors and GCF are:

```
Array // factors
(
    [0] => 1
    [1] => 2
)
2 // GCF
```

If the greatest common factor is greater than 1, we can reduce our fraction by dividing both parts of our fraction by it:

```php
if ($GCF > 1) {
    $numerator = $numerator / $GCF;
    $denominator = $denominator / $GCF;
}
var_dump($numerator, $denominator);
```

Again, for `0.8` the result is:

```
int(4) // numerators
int(5) // denominator
```

Final Solution

We have our algorithm in place, let's update our `Fraction` class to use it. See 4. I added a `simplify()` method, which returns a new object instance, in case the caller wants to keep the original fraction around. I also added a `__toString()` magic method to simplify outputting a fraction. With this code we can work with fractions like this:

```php
$frac = Fraction::fromFloat(0.80);
$simpl = $frac->simplify();
echo $frac . " simplifies to " . $simpl;
```

The code above outputs the string:

```
8/10 simplifies to 4/5
```

Caveats

The solution we looked at may not work for all decimals. Some floating-point values are difficult to represent, but PHP goes a long way to taking care of most of them for us. For a full discussion, check out this discussion on Stack Overflow[3].

Also, the solution doesn't convert fractions which come from repeating numbers. For example, one-third (1/3) is `0.3333` where the `3` repeats until infinity. Our code can not detect that or convert it back correctly. See this example:

```php
$frac = Fraction::fromFloat(1/3);
$simpl = $frac->simplify();
echo $frac . " simplifies to " . $simpl;
```

Our solutions give this output:

3 SO discussion: https://phpa.me/stackoverflow-count-after-float

```
333333333333333/1000000000000000 simplifies to
333333333333333/1000000000000000
```

Fractional Math

For next month, let’s build on the `Fraction` class. 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`

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[4] 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

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

Creating Content Types in Drupal 9

Nicola Pignatelli

Today you will create files to generate content type programmatically. At the end of this article, you will learn to create a simple content type with a text field and another content type with a paragraph field, which contains a set of fields.

Why Choose the Easiest Way?

If you want to create a new content type, you can use the usual Web Interface, but we like the thrill of discovery and not the comfort of the known way. Until Drupal 7, you created code in the module, and if you wanted to transport the code to another site, you had to copy the module and then check that the code didn't conflict with the new software. Forget it!

I will teach you how simple it is to create content type programmatically so you can create a custom module and transport the configuration to every site you want.

Create a Simple Content Type (d9_content_type)

Now let's move to the analysis of the module. In this article, the root folder project will be identified by [rdp], where I will enter the composer command. Drupal will be installed into the [rdp]/web folder, where I will enter the drush command.

D9_content_type.info.yml

First, I create an info module file: d9_content_type.info.yml.

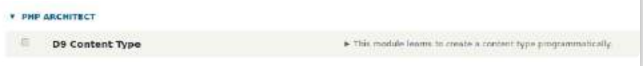
```
name: D9 Content Type
description: This module learns to create a content type
             programmatically.
type: module
core_version_requirement: 9.x
package: Php Architect
```

`name` and `description` parameters are visible on the Extend webpage (figure 1) and permit module selection. `type` parameter identifies typology in this example `module`. `core_version_requirement` verifies compatibility between the module and core. If the Drupal core version is major or equal to the parameter, it is possible to install the module. `package` places the module into a section of Extend webpage (Figure 1).

You can enable module by web interface by flag (Figure 1) or by drush with next command:

```
cd [rdp]/web
../vendor/drush/drush/drush pm:enable d9_content_type
```

Figure 1.



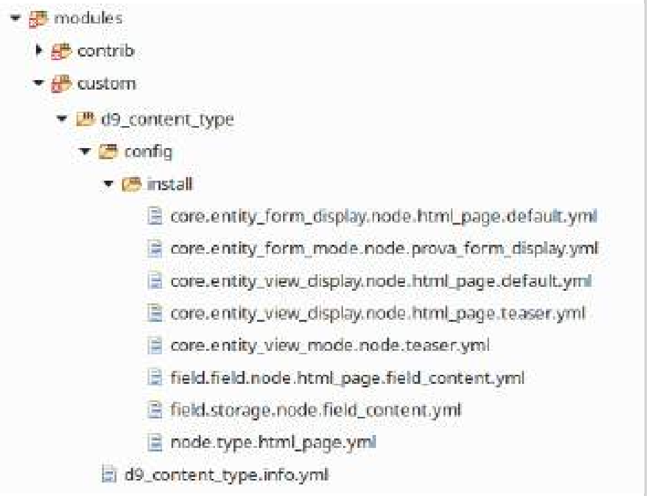
▾ PHP ARCHITECT

D9 Content Type ▸ This module learns to create a content type programmatically

A Walk Into the Structure of the Module

Figure 2 shows the module structure. Besides the `d9_content_type.info.yml` file, you find YAML files to create a content type under the `config/install` folder.

Figure 2.



```
▾ modules
  ▸ contrib
  ▾ custom
    ▾ d9_content_type
      ▾ config
        ▾ install
            core.entity_form_display.node.html_page.default.yml
            core.entity_form_mode.node.prova_form_display.yml
            core.entity_view_display.node.html_page.default.yml
            core.entity_view_display.node.html_page.teaser.yml
            core.entity_view_mode.node.teaser.yml
            field.field.node.html_page.field_content.yml
            field.storage.node.field_content.yml
            node.type.html_page.yml
        d9_content_type.info.yml
```

In the install folder, you may find many files for language configuration, image styles, and others. We'll use the following files for this example:

- `node.type.html_page.yml`
- `field.field.node.html_page.field_content.yml`
- `field.storage.node.field_content.yml`
- `core.entity_form_display.node.html_page.default.yml`
- `core.entity_view_display.node.html_page.default.yml`
- `core.entity_view_mode.node.teaser.yml`
- `core.entity_form_mode.node.prova_form_display.yml`

Node.type.html_page.yml

In this file (Listing 1), you find the definition of a content type:

- `status` define is content type is published to default.
- `dependencies` indicate if the creation of a content type depends on some modules.
- `name` the human name of a content type.
- `type` machine name of a content type.
- `description` indicates description field into content type definition.
- `help` is "Explanation or submission guidelines" field into "Submission form settings" tab to the end of the configuration page.
- `new_revision` indicates if content creates a revision every time is saved.
- `preview_mode` indicates if you see a preview before content is saved.
- `display_submitted` indicates if "Display author and date information" shows on the view display page.

Listing 1.

```
langcode: en
status: true
dependencies:
  enforced:
    module:
      - d9_content_type
name: 'Html Page'
type: html_page
description: 'Content type that can be used to add
              <em>Html Page</em>.'
help: ''
new_revision: false
preview_mode: 1
display_submitted: true
```

Field.storage.node.field_content.yml

This file (Listing 2), contains the definition of the `field_content` field, which can be added to every content type.

- `dependencies` indicate if the field depends on other modules.
- `id` unique id to assign to the field.
- `field_name` machine name of the field.
- `entity_type` entity type to which the field can be associated.
- `type` typology of the field, in this case, long multi rows.
- `cardinality` indicates if the field is single or multi value.
- `translatable` is the field translatable by default.

Listing 2.

```
langcode: en
status: true
dependencies:
  module:
    - node
    - text
id: node.field_content
field_name: field_content
entity_type: node
type: text_long
settings: {  }
module: text
locked: false
cardinality: 1
translatable: true
indexes: {  }
persist_with_no_fields: false
custom_storage: false
```

Field.field.node.html_page.field_content.yml

This file (Listing 3), allows association between field and content type. Important properties are:

- `dependencies` indicate if the field depends on other configuration files and/or modules.
- `id` unique id to assign to the field.
- `field_name` machine name of the field.
- `entity_type` entity type to which the field is associated.
- `bundle` type of content type (html_page) to which the field is associated.
- `label` label field into content type fields list.
- `description` a description of the field.
- `required` indicates if the field is mandatory or not.
- `translatable` field is translatable or not into content type.
- `default_value` there is a default value.
- `field_type` typology of field, in this case, long multi rows.

Listing 3.

```
langcode: en
status: true
dependencies:
  config:
    - field.storage.node.field_content
    - node.type.html_page
  module:
    - text
id: node.html_page.field_content
field_name: field_content
entity_type: node
bundle: html_page
label: Content
description: ''
required: false
translatable: false
default_value: {  }
default_value_callback: ''
settings: {  }
field_type: text_long
```

Display Configuration Files

The last three files in the module allow you to configure the data entry form and data display.

Listing 4.

```
langcode: en
status: true
dependencies:
  config:
    - field.field.node.html_page.field_content
    - node.type.html_page
  module:
    - path
    - text
id: node.html_page.default
targetEntityType: node
bundle: html_page
mode: default
content:
  created:
    type: datetime_timestamp
    weight: 10
    region: content
    settings: {  }
    third_party_settings: {  }
  field_content:
    type: text_textarea
    weight: 121
    region: content
    settings:
      rows: 5
      placeholder: ''
    third_party_settings: {  }
  path:
    type: path
    weight: 30
    region: content
    settings: {  }
    third_party_settings: {  }
  promote:
    type: boolean_checkbox
    weight: 15
    region: content
    settings:
      display_label: true
    third_party_settings: {  }
  status:
    type: boolean_checkbox
    weight: 120
    region: content
    settings:
      display_label: true
    third_party_settings: {  }
  sticky:
    type: boolean_checkbox
    weight: 16
    region: content
    settings:
      display_label: true
    third_party_settings: {  }
  title:
    type: string_textfield
    weight: -5
    region: content
    settings:
      size: 60
      placeholder: ''
    third_party_settings: {  }
  uid:
    type: entity_reference_autocomplete
    weight: 5
    region: content
    settings:
      match_operator: CONTAINS
      match_limit: 10
      size: 60
      placeholder: ''
    third_party_settings: {  }
hidden:
  langcode: true
```

Core.entity_form_display.node.html_page.default.yml

This file (Listing 4), permits to configure form display of content type, namely form to create or update content. This file configures default display. In next paragraph I explain how to create other display to form.

Important properties are:

- `dependencies` indicate if the field depends on other configuration files and/or modules.
- `id` unique id to assign to content type display.
- `field_name` machine name of the field.
- `targetEntityType` entity type to which the field is associated.
- `bundle` type of content type (html_page).
- `mode` what display you define in this file.
- `content` section containing the fields to show on the form and the settings of every default and added field.
- `hidden` list of fields that should not be shown on the form.

Elements below `content` are the fields to show on the form. settings and third_party_settings are complied in base to type field.

Core.entity_view_display.node.html_page.default.yml

This file (Listing 5), permits to configure default display of content type, namely configure full page view. This file configures default display. In next paragraph I explain how to create other display to view.

Important properties are:

- `dependencies` indicate if the field depends on other configuration files and/or modules.
- `id` unique id to assign to content type display.
- `targetEntityType` entity type to which the field is associated.
- `bundle` type of content type (html_page).
- `mode` what display you define in this file.
- `content` section that contains the fields to show on the form and the settings of every default and added field.
- `hidden` list of fields that are not shown on the form.

Elements below `content` are fields to show on the form. settings and third_party_settings are complied in base to type field.

Core.entity_view_mode.node.teaser.yml

This file (Listing 6), is similar to previous file, because is only another display type that you can use into your installation, for example into views.

Important difference with default view is `core.entity_view_mode.node.teaser` file. This file contains the configuration for this display.

Listing 5.

```
langcode: en
status: true
dependencies:
  config:
    - field.field.node.html_page.field_content
    - node.type.html_page
  module:
    - text
    - user
id: node.html_page.default
targetEntityType: node
bundle: html_page
mode: default
content:
  field_content:
    type: text_default
    label: above
    settings: {  }
    third_party_settings: {  }
    weight: 101
    region: content
  links:
    settings: {  }
    third_party_settings: {  }
    weight: 100
    region: content
hidden:
  langcode: true
```

Wizard of Oz: Change Form Display

Sometimes we need to display form and view of an entity, in particular content type, in a different shape. This is possible by displays configuration. For mode display you saw teaser mode. For form mode display I teach you in next paragraph.

Core.entity_form_mode.node.prova_form_display.yml

In this case, to generate a new form display, you create a new file with the same name as this section. The new display id is prova_form_display, and its content is shown in Listing 7.

After you enable the d9_content_type module, you will see a new form mode admin/structure/display-modes/form (Figure 3).

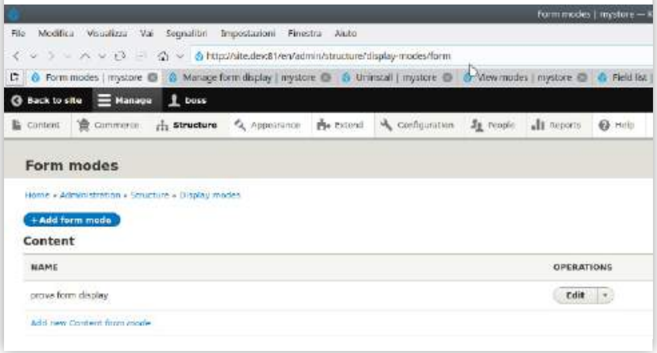
Listing 7.

```
langcode: en
status: true
dependencies:
  module:
    - node
id: node.prova_form_display
label: prova form display
targetEntityType: node
cache: true
```

Listing 6.

```
langcode: en
status: true
dependencies:
  config:
    - core.entity_view_mode.node.teaser
    - field.field.node.html_page.field_content
    - node.type.html_page
  module:
    - user
id: node.html_page.teaser
targetEntityType: node
bundle: html_page
mode: teaser
content:
  links:
    settings: {  }
    third_party_settings: {  }
    weight: 100
    region: content
hidden:
  field_content: true
  langcode: true
```

Figure 3.



http://site.dev:81/en/admin/structure/display-modes/form

Form modes

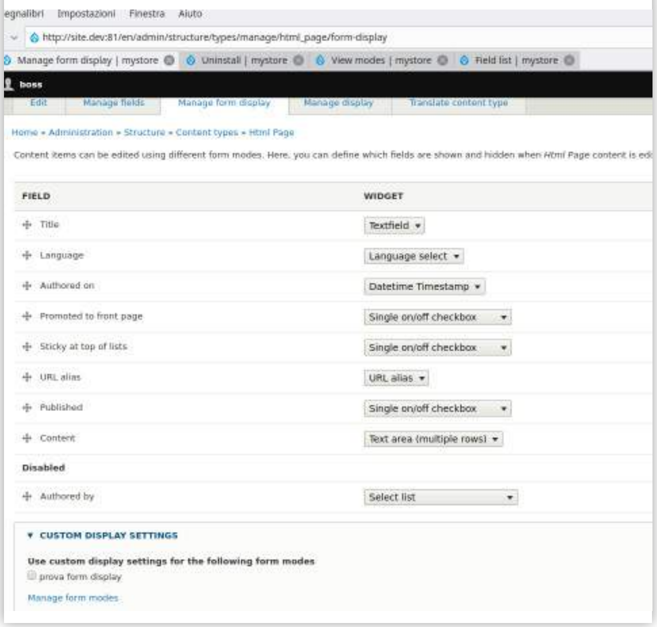
Home » Administration » Structure » Display modes

Content

NAME	OPERATIONS
prova form display	Edit

Add new Content form mode

Figure 4.



http://site.dev:81/en/admin/structure/types/manage/html_page/form-display

Home » Administration » Structure » Content types » Html Page

Content items can be edited using different form modes. Here, you can define which fields are shown and hidden when *Html* Page content is edi

FIELD	WIDGET
Title	Textfield
Language	Language select
Authored on	Datetime Timestamp
Promoted to front page	Single on/off checkbox
Sticky at top of lists	Single on/off checkbox
URL alias	URL alias
Published	Single on/off checkbox
Content	Text area (multiple rows)
Disabled	
Authored by	Select list

▾ CUSTOM DISPLAY SETTINGS

Use custom display settings for the following form modes

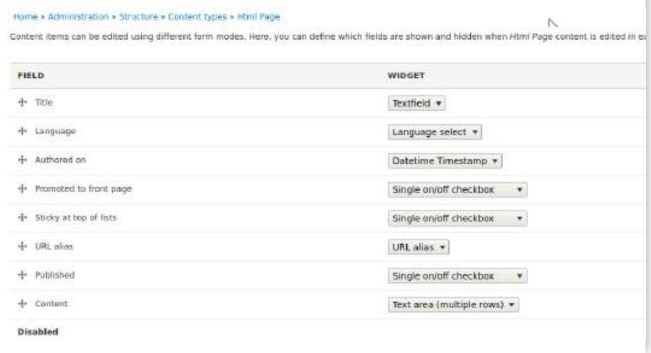
☐ prova form display

Manage form modes

Now in each form configuration page of a content type, it will be possible to select display modes and configure them according to your needs (Figure 4).

After you enable `prova form display`, you can switch between display enabled by clicking on form mode display links (Figure 5).

Figure 5.



Home » Administration » Structure » Content types » Html Page

Content items can be edited using different form modes. Here, you can define which fields are shown and hidden when *Html Page* content is edited in ea

FIELD	WIDGET
Title	Textfield
Language	Language select
Authored on	Datetime Timestamp
Promoted to front page	Single on/off checkbox
Sticky at top of lists	Single on/off checkbox
URL alias	URL alias
Published	Single on/off checkbox
Content	Text area (multiple rows)
Disabled	

Conclusion

In this article, you have seen how to create a content type programmatically and add displays for both the `form mode` and the `view mode`.

There are many other configuration files you can create to enrich both the content and system configurations. An entire year of PHP Architect articles would not be enough to list them all; moreover, such a complete list would not remain in your memory without practicing with some actual cases.

I wanted to show you the potential of the new way of managing configurations and creations of parts of the site in Drupal 9, dividing them from code into various files such as `[module name].module`. Doing so allowed the isolation and transport of configurations without the need to change hook names or anything else.

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](https://twitter.com/pignatellicom)

Requirements for Drupal 9:

- PHP 7.3+. PHP 8 is supported from Drupal 9.1.0
- Web Server
- Apache 2.4.7+
- Nginx 0.7.x+
- Database Server
- MySQL or Percona 5.7.8+
- MariaDB 10.3.7+
- SQLite 3.26+ is required
- PostgreSQL 10+ with the pg_trgm extension
- Drush
- version 10+ require PHP 7.1+
- version 11+ require PHP 7.4+

Other Software:

- Composer

Related URLs:

- MySQL - https://dev.mysql.com/downloads/mysql/
- MariaDB - https://mariadb.org/
- PHP - http://www.php.net
- Apache - http://www.apache.org
- Nginx - http://www.nginx.org
- Drupal - http://www.drupal.org
- Composer - https://getcomposer.org
- Drush - https://www.drush.org/latest/install

found something?
Writings of some species that lived here
Forget... It's in Erlang
Let it go. We will never understand.

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

The Dangerous Safety of Comfort

Beth Tucker Long

“Hey, I need to log in to the app on my phone. What’s the test account password?”

“Oh, here. I’ll type it in for you.”

“I need to log on to multiple devices, so I need to know the actual password.”

“It’s um...”

The Cambridge Dictionary defines “muscle memory[1]” as:

> *“The ability to move a part of your body without thinking about it, learned by repeating the movement many times.”*

In short, muscle memory is our absolutely amazing ability to train our bodies to do things without us needing to think through each movement/action individually. This is super helpful in many ways. In exercise, it makes it easier and quicker to build or rebuild muscle tissue in certain areas of your body. In our day-to-day, it’s how your hand knows exactly which way to turn the sticky front door knob even though you can’t tell someone else which way to do it. At work, it’s how your mouse hand moves the cursor to all the right icons, and you have no idea what those icons look like.

Muscle memory generally takes around 2-4 weeks to build up and, depending on how long you have practiced it, can last for years or even decades without use. It is wonderfully helpful and healthy for our bodies. Building muscle memory can help make exercise more effective. Repetitive practice drills build muscle memories to help people make the right choices in emergencies when they need their body to act, but they don’t have time to think and analyze the situation. Research is even being done to see how muscle memory can help those with dementia. There is so much good that comes from muscle memory.

But muscle memory can also make us complacent and lazy. The already-learned movements are easier than learning new ones, so we avoid anything that takes us out of our comfort zone. Why bring this up? How many of you skipped the Vim article because you already have an IDE of choice and “it would be too hard to switch?” Retraining muscle memory is hard (and so frustrating), but never checking out new tools because of a muscle memory hurdle means missing out on opportunities to make your workflow more efficient. It means wasting time doing things that have been automated. It means wasting brainpower hunting for things the system can find for you automatically.

It means being stuck doing things the way you have always done things just because that is the way they have always been done. If there is one phrase that should make all programmers shudder, it’s “because that’s the way we’ve always done it.” You’ve been trained to watch for this and to fight it when project owners want to ignore security or performance advice. Make sure you are watching out for it in your own workflow too.

And before you mention how much time you’ll waste trying to learn where everything is in a new program when you already know where it is in yours, I challenge you instead to explore how much time you can save yourself by critically evaluating your current workflow and learning about new tools that can help you build better processes.

Don’t stagnate. Get out there and learn something new.

Beth Tucker Long is a developer and owner at Treeline Design[2], a web development company, and runs Explioricon[3], a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development[4] and Full Stack Madison[5] user groups. You can find her on her blog (http://www.alittleofboth.com) or on Twitter @e3BethT

1 *muscle memory: https://phpa.me/cambridge-muscle-memory*

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

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

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

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