www.phparch.com

July 2022
Volume 21 - Issue 7

php[architect]

The Magazine For PHP Professionals

Database Freedom

MongoDB and PHP---A Perfect Match

Customizing Drupal Feeds For Smooth Migrations

ALSO INSIDE

Education Station:
What is Git Doing?

PHP Puzzles:
The Birthday Paradox

PSR Pickup:
PSR-7 HTTP Message Interface

The Workshop:
PHP from VM to Docker

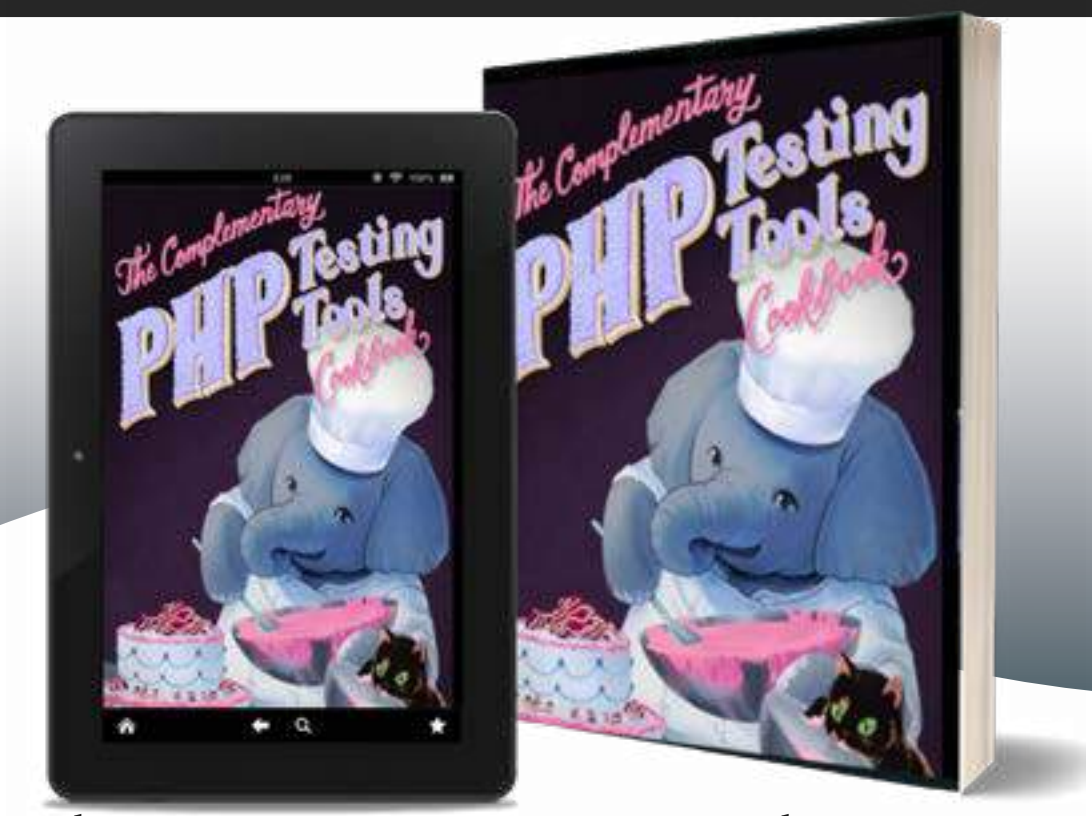
Drupal Dab:
Create a Custom Module

Artisan Way:
Refactor to Enums

Security Corner:
Demystifying Multifactor Auth

DDD Alley:
Structur by Use Case

finally{}:
Blind



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

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

Available in Print+Digital and Digital Editions.

Order Your Copy

phpa.me/grumpy-cookbook

Edited with Freedom of Thought

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

An Ode to Team Building

John Congdon

Team building and continuing education is an underutilized activity by most companies. I believe that it can make the difference between a successful and a failing company. I had the pleasure of taking part in a developer group extended education and team-building meeting in June.

As developers, we often fall into a few different categories of employment.

Some tend to work alone for small companies, often just getting work done and rarely getting the chance to learn from others.

Some work for startups and end their workday feeling overworked and dreading what is to come. These developers are often overworked and are left feeling like they should have done more to help the poor little startup make it to greatness. That carrot is always being dangled in front of them.

And then there are the lucky developers that get to work on a team of like-minded individuals. They learn from each other, care about one another, and want to get work done because they like what they do.

I am fortunate to belong to two such teams. Our developers at DiegoDev have the freedom to make decisions on how to implement the solutions they dream up for our clients. They seem to get along with each other and genuinely care about their teammates (at least from my management view). And we encourage continued education with our team.

The second team at PhoneBurner is similar as well, and in June, they held a developer meeting that I felt brought that team closer together. There was continued education via a few presentations, a mini competition between a few teams playing a fun game called BattleSnake, and a fun gift box for each developer.

I can't rave enough about this experience. If your company is looking to improve your team's relationship, please reach out[1] and let us here at PHP Architect help your team take a few days away from the ordinary to achieve the extraordinary.

This month's issue starts off with two great feature articles, "Customizing Drupal Feeds For Smooth Migrations" by Doug Groene and "MongoDB and PHP—-A Perfect Match" by Joel Lord.

Our monthly columnists have been hard at work as well. Over in Education Station, Chris Tankersley gives us "What is Git Doing"? Then Eric Mann helps with "Demystifying Multifactor Authentication" on Security Corner.

Continuing down through DDD Alley, Edward Barnard brings up "Structure by Use Case".

The Workshop will help improve our development environments with the article "PHP from Virtual Machine to Docker" by Joe Ferguson..

Nicola Pignatelli tells you everything you've ever wanted to know to "Create a Custom Module in Drupal 9" in the Drupal Dab.

If you're a puzzle fan, check out this very counterintuitive PHP Puzzles column, "The Birthday Paradox" by Oscar Merida.

Learn all about the PSR-7 HTTP Message Interface" in Frank Wallen's PSR Pickup.

And finally{}, Beth Tucker Long talks about the pitfalls behind the commonly used "Blind" talk selection process used by many conferences and user groups.

1 mailto: support@phparch.com?subject=Developer+Team+Relationship+Building

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: http://phpa.me/sub-to-updates
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download the Code Archive:

http://phpa.me/July2022_code

MongoDB and PHP—A Perfect Match

Joel Lord

Modern applications require modern tooling. MongoDB has increasingly become a popular choice for building large-scale PHP applications. It is a convenient and intuitive way to use data. This article will show you how to use MongoDB and PHP together.

MongoDB might not be the first database people think about when working with PHP. For years, the go-to database for PHP applications was MySQL. The application duo was often used with Apache and Linux as part of the LAMP (Linux-Apache-MySQL-PHP) stack. However, modern applications require modern tooling. More and more, large PHP applications are deployed at large scales and require databases that can scale with them. MongoDB is a general-purpose database that works perfectly with PHP and can handle massive amounts of data and concurrent traffic. When used with MongoDB Atlas, the cloud application data platform by MongoDB, it can provide a multitude of advanced features that will make your application stand out.

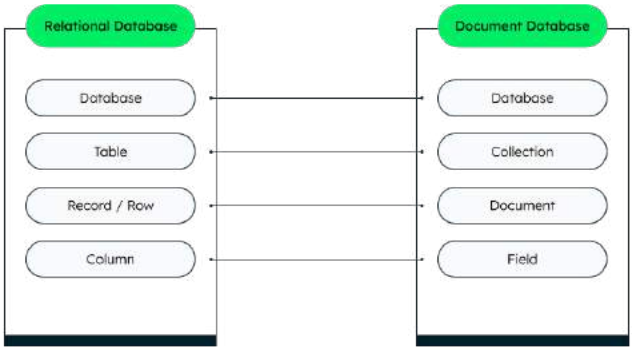
This article covers the main differences between a document database and traditional relational databases and various ways to install and use MongoDB. It provides you with code samples to get started with MongoDB in PHP.

Document Databases Vs. Relational Databases

You might have heard document databases, such as MongoDB, store data without any sort of schema and that you can store any data with no structure whatsoever. While this statement is somewhat true, it is also not the best way to use a document database.

MongoDB is a flexible schemaless database. It can enforce schemas when needed and offers an easy way to change and adjust the format of your documents as your application evolves. By using a particular schema when storing your data, you can create indexes that will provide you with the same performance you can expect from any other database.

Figure 1.



You might wonder, then, why you would use a document database rather than a traditional database. The biggest performance gain will have to do with the way you build your schema. But I'm getting ahead of myself. Let's look at some terminology to understand how MongoDB and other databases compare.

Databases, Collections, and Documents

Because MongoDB introduced a new way to store and retrieve data, they also have a slightly different vocabulary. Thankfully, there is a direct match between the two. See Figure 1.

Optimizing Data Schemas

Now that you know what documents are let's dig a little deeper and understand how they compare to records. For the purpose of this article, let's look at this database schema that stores information about conference speakers. You can see that it takes three tables to describe this data—one for the speakers themselves, another for social links, and a third table for talks.

Table: Speakers

_id	name	bio
1	Joel Lord	Joel Lord is a developer advocate at MongoDB …

Table: Links

_id	speaker_id	social	link
1	1	Twitter	https://twitter.com/joel__lord
2	1	Github	https://github.com/joellord

Table: Talks

_id	speaker_id	conference_id	title	day	time
1	1	1	MongoDB and PHP: A Perfect Match	June 1, 2022	14:15

Listing 1.

```
// Speakers
[
  {
    _id: 1,
    name: "Joel Lord",
    bio: "Joel Lord is a developer advocate at MongoDB…"
  }
]

// Socials
[
  {
    _id: 1,
    speaker_id: 1,
    social: "Twitter",
    link: "<https://twitter.com/joel__lord>"
  },
  {
    _id: 2,
    speaker_id: 1,
    social: "Github",
    link: "<https://github.com/joellord>"
  }
]

// Talks
[
  {
    _id: 1,
    speaker_id: 1,
    conference_id: 1,
    title: "MongoDB and PHP: A Perfect Match",
    day: "June 1, 2022",
    time: "14:15"
  }
]
```

MongoDB is slightly different when it comes to storing data. Rather than using normalized tables, it uses JSON-like documents. More specifically, it uses BSON (Binary JSON), an extended version of JSON that uses binary storage and additional data types. An easy way to convert the tables above to a MongoDB database is to convert them directly into objects as shown in Listing 1.

It is a common misconception that MongoDB does not support joins. They are possible, but it is not always the best approach. In this specific example (Listing 2), you can see a one-to-many relationship between the speakers and the socials. However, as you can imagine, that information is always accessed together. For this reason, you can embed those two documents into an array, part of the speaker document.

Thanks to the flexible schema, you could simplify the speaker object further by creating an object for the speaker's socials. See Listing 3.

By doing so, you avoid doing expensive outer joins every time you need to look up a speaker. You could even go one step further and embed the speaker details in the talks table. However, this would mean duplicating the speaker data if he has multiple talks. In some cases, data duplication is not an issue. Disk space is cheap, and data duplication should be used as a strategy to increase performance when needed.

Listing 2.

```
[
  {
    _id: 1,
    name: "Joel Lord",
    bio: "Joel Lord is a developer advocate at MongoDB…",
    socials: [
      {
        social: "Twitter",
        link: "<https://twitter.com/joel__lord>"
      },
      {
        social: "Github",
        link: "<https://github.com/joellord>"
      }
    ]
  }
]
```

Listing 3.

```
[
  {
    _id: 1,
    name: "Joel Lord",
    bio: "Joel Lord is a developer advocate at MongoDB…",
    socials: {
      Twitter: "<https://twitter.com/joel__lord>",
      Github: "<https://github.com/joellord>"
    }
  }
]
```

Another strategy here could be to embed the talks inside the speaker document. This, however, could cause performance issues when trying to render a schedule, as each document would need to be parsed to list them in a grid on the website.

For this application, the software engineer could decide to use a `$lookup`, MongoDB's left outer join equivalent, when they need to display both tables together and do a partial data duplication to guarantee the best possible performance.

Listing 4.

```
[
  {
    _id: 1,
    speaker: {
      _id: 1,
      name: "Joel Lord"
    },
    conference_id: 1,
    title: "MongoDB and PHP: A Perfect Match",
    day: "June 1, 2022",
    time: "14:15"
  }
]
```

Now only one query needs to be done on the database to display the schedule grid, and only one call is required to display the complete speaker profile. Reducing the number of joins will significantly improve the performance of the queries.

As a rule of thumb, your data should follow this mantra:

> *"Data accessed together should be stored together."*

Don't try to over complicate your data structures by embedding too much information, and don't try to reproduce a traditional database structure. A balance of both is needed to ensure the best possible performance.

A Perfect Match

When data is stored together, it is simpler to fetch all of it at once. Take the previous example. If you wanted to create a page that would display the user profile, you would need to either use a left outer join to get the necessary information or do two queries to your database.

When using PHP with the MongoDB drivers, the code looks like the following.

Listing 5.

```
1. <?php
2. // Get the first speaker from the collection
3. $speaker = $collection->findOne();
4. // Output the speaker profile
5. ?>
6. <h1><?= $speaker->name ?></h1>
7. <h2>Socials</h2>
8. <ul>
9. <?php
10.   foreach ($speaker->socials as $social => $link) { ?>
11.     <li><a href="<?= $link ?>"><?= $social ?></a></li>
12.   <?php
13.   }
14. ?>
```

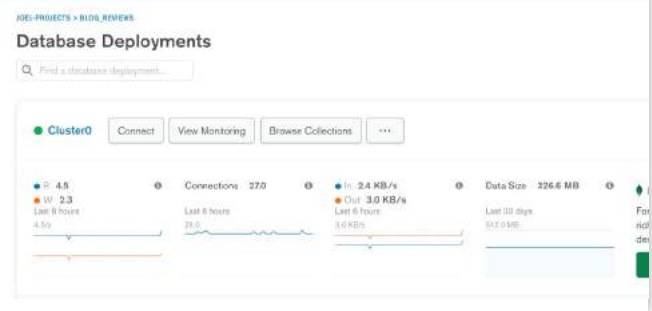
As you can see in this example, the object retrieved from the database maps directly to the entity displayed on this page. This is why MongoDB makes it more natural to work with software developers.

You now know all the MongoDB basics, from understanding the foundational concepts to modeling data. Let's get hands-on and look at some examples.

Setting Up Mongodb and Configuring PHP

The first step will be to install MongoDB. There are many installation possibilities. You can download the community version and install it locally or run the community version directly from the official container maintained by the Docker team. However, the easiest way to use MongoDB, the one used in this article, is through MongoDB Atlas, a free-to-use cloud offering by MongoDB. Once you have a MongoDB instance up and running, you will need to install the PHP driver. Instructions are also provided here, or you can use a PHP-ready container.

Figure 2.



Set Up Your Mongodb Instance

The easiest and cheapest way to get started with MongoDB is to use a free cluster on MongoDB Atlas[1]. By following the instructions on the provided link, you will be able to create your first free (M0) MongoDB instance. This instance is a complete three-node cluster ready to be used. In addition to the MongoDB database, you will have access to all the MongoDB Application Data Platform features, including a full-text search extension (Atlas Search) and advanced visualization tools (Charts). Those are out of the scope of this article, though. Once you have a cluster ready, you should see something similar Figure 2 following on the Atlas UI.

From this page, you will get the connection string necessary to connect to your database by using the Connect button. We'll get to that later.

That's all you need to do for now with your MongoDB Atlas cluster. It is now up and running and waiting for your data.

Install the PHP Driver

The next step to connecting to MongoDB from your server is to set up the PHP native driver. To do so, you will need to use `pecl` from your favorite terminal. On Linux or macOS, you can use the following command to install the drivers. (You might need administrative rights to run the command.)

```
sudo pecl install mongodb
```

Once the installation is complete, you will need to add the MongoDB driver to your `php.ini` file. Add the following line at the end of the file.

```
extension=mongodb.so
```

You now have everything ready to use the PHP driver. You can find further detailed instructions and the instructions for the Windows operating system on the MongoDB documentation website[2].

1 *MongoDB Atlas: https://www.mongodb.com/cloud/atlas/register*

2 *documentation website: https://phpa.me/mongodb-php-library*

If you don't want to install the driver locally and have docker installed, you can use the following `docker` command to run a pre-configured PHP server with Apache and the MongoDB drivers.

```
docker run --rm -d -p 8080:80 -v
       $(pwd)/www:/var/www joellord/php-mongodb
```

If you want to try these examples yourself, you can get all the source code from the repository[3].

Performing Crud Operations

Now that you have an Atlas cluster ready and a PHP server prepared to access your MongoDB database, it's time to write code to connect to the database and perform some basic CRUD (Create-Read-Update-Delete) operations.

These examples will all be using vanilla PHP for the sake of simplicity, but you should be able to adapt these examples to your framework of choice.

Install Dependencies

You can use Composer to install the required packages to connect to your MongoDB database. To install those, run the following command.

```
composer require mongodb/mongodb
```

Once this operation is complete, you will have access to all the necessary dependencies to connect to your cluster. Don't forget to include Composer's autoloader code in your PHP file.

```
<?php
require_once __DIR__ . '/vendor/autoload.php';
```

With this, you are now ready to connect to your database.

Connect to the Server

First, start by declaring a client that will connect to your database with the `MongoDB\Client` object. You will need your connection string from your Atlas cluster. If you go back to *Figure 2*, you will see a Connect button. Using that button will provide various options to get your connection string. That string should start with mongodb+srv, contain your credentials, and have your cluster URL.

```
$client = new MongoDB\Client('YOUR_CONNECTION_STRING');
```

Now that you have a client, you can set up a database. The client object contains all the databases as properties so that you can assign this database to a variable. In this example, we are connecting to a database named perfectMatch.

```
$db = $client->perfectMatch;
```

Next, you can also assign a collection to a variable, making it easier to use the various methods available to you on this specific collection.

```
$collection = $db->speakers;
$collection->drop();
```

Now that you have access to this collection, you will be able to perform those CRUD operations against it. Since all of these examples will run on the same page, the drop method has been used here to ensure that a fresh collection is used every time the page is accessed by dropping the existing collection. MongoDB will recreate the collection upon the first insert.

Create

Let's use the same example as we did at the beginning of this article. We will use the `insertOne` method on this collection to insert the speaker described earlier.

```
$newSpeaker = [
    'name' => 'Joel Lord',
   'bio' => 'Joel Lord is a developer advocate at MongoDB…',
    'socials' => [
        'Twitter' => '<https://twitter.com/joel__lord>',
        'Github' => '<https://github.com/joellord>'
    ]
];
```

The operation will return a result that will include the `insertedId`. MongoDB requires a unique id named `_id` for each document. Since we haven't specified one in the `$newSpeaker` object, it was automatically generated.

```
$result = $collection->insertOne($newSpeaker);
echo('New user inserted with id '.$result->getInsertedId());
```

You have just inserted your first record into your MongoDB database. If you need to insert more than one record at a time, you can alternatively use the `insertMany` method with an array.

```
$otherSpeakers = [
  [ 'name' => 'William Wright' ],
  [ 'name' => 'Amanda Ryan' ]
];

$result = $collection->insertMany($otherSpeakers);
echo('Added '.$result->getInsertedCount().' documents');
```

The code above will insert two new documents into the database.

If you go back to the Atlas UI, you will see a "Browse Collections" button. From this screen, you can browse the documents currently in your database.

Figure 3. The Atlas Data Explorer lets you browse your documents directly from the web UI.

3 *repository:* https://github.com/joellord/mongodb-php

Read

Now that you have at least one document in your collection, it's time to read this data. You'll use the `findOne` method to read from the collection. You can pass an argument to the function to specify the criteria to use. If left empty, this should return the first entry in your collection.

```
$speaker = $collection->findOne();
echo('The first speaker found in the collection is '.
                $speaker->name);
foreach ($speaker->socials as $social => $link) {
  echo($social.' ('.$link.')');
  echo('<br/>');
}
```

As you can see here, using MongoDB's capabilities to embed arrays inside your documents makes it really easy to display content that is meant to be displayed together. In this case, you can immediately iterate through the speaker socials without the need for complex joins.

You can also find a specific document by passing an associative array to the `findOne` function.

```
$speaker = $collection->
                findOne([ 'name' => 'William Wright' ]);
echo('Found a match with id '.$speaker->_id);
```

This will return the document for the specified filter. If no match is found, you will get a null value.

```
$speaker = $collection->
                findOne([ 'name' => 'Unknown Speaker' ]);
echo(is_null($speaker));
```

Just like you can insert multiple documents, you can also retrieve various documents. The find method works like its counterpart `findOne` but will return a cursor with numerous entries. You can then iterate through the entries with a `foreach` or use the `toArray` method to convert it to an array.

```
$speakers = $collection->find()->toArray();
echo('This collection contains '.
                count($speakers).' documents');
```

Update

Now that you have multiple entries in your database and know how to query those, you can perform an update operation on one of your documents. In this case, we'll change the first speaker's bio using the `updateOne` method.

The first argument for the `updateOne` method is the filter to use, and it works just like the filter for the `findOne` method. The second argument takes an operator and the update to execute. In this case, we are using the `$set` operator to update a specific field.

```
$result = $collection->updateOne(
  [ 'name' => 'Joel Lord' ],
  ['$set' => [ 'bio' => 'New bio for Joel' ]]
);
$speaker = $collection->
                findOne([ 'name' => 'Joel Lord' ]);
echo('Speaker: '.$speaker->name);
echo('Bio: '.$speaker->bio);
```

After the update, the `findOne` call will fetch the updated speaker and display its name and new bio.

If you need to update more than one document at once, you can also use the `updateMany` method, which works the same way.

Delete

The next and final operation of the CRUD series is a delete operation enabling you to delete a document from the database. The syntax for the `deleteOne` method should already be familiar. It uses a filter like `findOne` and `updateOne`. Below, we delete an entry from the collection and then fetch all the remaining entries again:

```
$result = $collection->
                deleteOne([ 'name' => 'William Wright' ]);
$speakers = $collection->find()->toArray();
echo('This collection contains '.
                count($speakers).' documents');
```

And just like for the other operations, a `deleteMany` method exists to perform multiple delete operations at once.

Advanced Querying

There will be times when you want to perform more complex operations against your data. While the document model helps simplify your queries, sometimes it is not enough. You might want to transform some data before displaying it or perform grouping functions such as a count. This is where aggregation pipelines come into play.

Aggregation pipelines are a series of stages through which the data is piped to provide you with the desired output. We will use an aggregation pipeline to gather the first five speakers alphabetically and only list their names and socials.

First, let's start by populating the collection with some more documents. Some sample data is available on the Github repository, so you can use the PHP function `file_get_contents` to retrieve that JSON document, convert it to an array, and use an `insertMany` to insert those into the database.

```
$data = file_get_contents('<https://raw.githubuser
                'content.com/joellord/mongodb-php/main/'.
                'sample_data.json>');
$collection->insertMany(json_decode($data));
$speakers = $collection->find()->toArray();
echo('This collection now contains '.count($speakers).
                ' documents');
```

Your collection should now have 27 documents. Let's create a pipeline. The first stage will be a `$project` operation that will

Listing 6.

```
$pipeline = [
  ['$project' => [
    '_id' => 0,
    'name' => 1,
    'socials' => 1
  ]],
  ['$sort' => [
    'name' => 1
  ]],
  ['$limit' => 5]
];
```

filter out only the fields we want to display. After that, you will use a `$sort` stage to specify which field you want to order by. Finally, a `$limit` stage will limit the number of documents returned by the pipeline. See Listing 6.

You can run the pipeline using the aggregate method on the collection.

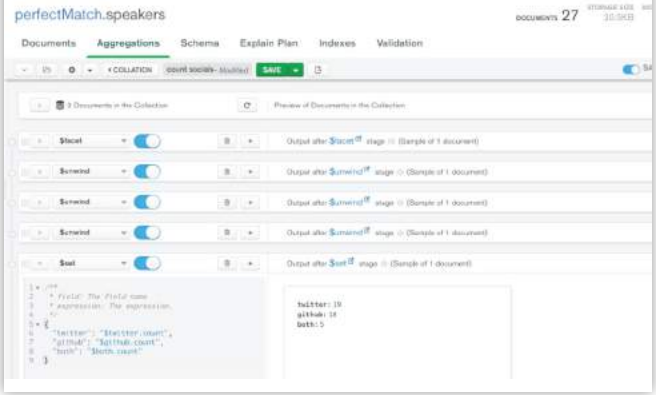
```
$speakers = $collection->aggregate($pipeline);
foreach($speakers as $speaker) {
  echo($speaker->name);
}
```

Listing 7.

```
[
  [
    '$facet' => [
      'twitter' => [

       ['$match' => ['socials.Twitter'=> ['$exists'=> 1]]],
        ['$count'=> 'count']
      ],
      'github'=> [
        ['$match'=> ['socials.Github'=> ['$exists'=> 1]]],
        ['$count'=> 'count']
      ],
      'both'=> [
        ['$match'=> ['$and'=> [
          ['socials.Github'=> ['$exists'=> 1]],
          ['socials.Twitter'=> ['$exists'=> 1]]
        ]]],
        ['$count'=> 'count']
      ]
    ]
  ],
  ['$unwind'=> ['path'=> '$twitter']],
  ['$unwind'=> ['path'=> '$github']],
  ['$unwind'=> ['path'=> '$both']],
  ['$set'=> [
      'twitter'=> '$twitter.count',
      'github'=> '$github.count',
      'both'=> '$both.count'
    ]
  ]
]
```

Figure 4.



Doing so will show you the first five speakers when ordered alphabetically. Some aggregation pipelines can get much more complex (Listing 7).

Can you guess what this aggregation pipeline does? Figure 4 shows the aggregation pipeline built with the MongoDB Compass aggregation builder. This builder is an excellent and easy way to explore your data. Each stage provides you with a data sample to see results.

As you experiment with MongoDB, you will find more ways to manipulate and transform your data using those pipelines.

What next? You now have all the necessary tools to use MongoDB and PHP. Hopefully, you now understand why they are a perfect match. Using a document database lets you store and query data naturally, without the need to build complex queries that join data from multiple tables. This data can then be used directly in your application. When you need more complex data transformations, you can always use the aggregation pipeline framework to transform the data to suit your needs.

Many operators filter and transform your data—much more than we could present in an article. After you've created your free MongoDB Atlas cluster, check out the documentation[4] and the community forums[5] if you need help.

Joel Lord is passionate about the web and technology in general. He likes to learn new things, but most of all, he wants to share his discoveries. He does so by traveling to various conferences all across the globe. He graduated from college in computer programming in the last millennium. Apart from a little break to get his BSc in computational astrophysics, he was always in the industry. In his daily job, Joel is a developer advocate with MongoDB, where he connects with software engineers to help them make the web better by using best practices in web development. During his free time, he can be found stargazing at a campground somewhere or brewing a fresh batch of beer in his garage. @joel__lord

4 *documentation: https://docs.mongodb.com*

5 *community forums: https://mongodb.com/community/forums*

Customizing Drupal Feeds For Smooth Migrations

Doug Groene

With custom Feeds Tamper plugins, you can easily build a set of reusable data manipulation tools to fit the quirks of your data and greatly simplify some Drupal migration and data import projects.

Like most developers who worked with the Feeds module in Drupal 7, I became a huge fan of its robust functionality and elegant UI. For me, it became a go-to solution whenever I had to import some content that could map to fields on my content types, perhaps with some alteration from the powerful Feeds Tamper module. While the typical use case for Feeds is importing, perhaps on a cron schedule, from a periodically updating external feed, Feeds also does spectacularly well with one-time imports.

Unfortunately, Feeds was a bit late to the party when Drupal 8 came out, and for years I had to look to other solutions. In the case of migrations, that solution was the core Migrate module and related ecosystem of contrib modules, such as Migrate Plus. These modules form a swiss-army knife type toolset that can pull in highly complex data using various methods: direct database connection, CSV, JSON, XML, etc. There are infinite possibilities for fetching, transforming, and depositing data exactly how you need to, using source, process, and destination plugins. For complicated architecture with paragraphs or layers of nested entity references, Migrate is still the way.

For more straightforward migration tasks, however, I can now turn to my old friend, the Feeds module, with its friendly UI and easy, often code-free settings, mappings, and tampers.

The Challenge

Recently I was tasked with migrating a relatively simple content type called Article from Drupal 7 to Drupal 9. One of the fields, called Internal Link, is meant to hold a reference to another piece of content. In Drupal 9, we wanted this to be an entity reference. Unfortunately, in Drupal 7, this field was a link field that contained nothing but the aliased URL for the content, with no other information about the destination. The path aliases were preserved in Drupal 9, but how could I leverage those URLs to point my entity reference field to the correct target?

My first thought was to check the mappings on my Feed settings. For an entity reference field, you can map based on several different aspects of the target, not just ID. You can map based on the title or any of the fields. Unfortunately, URL alias is not a field, and it is not one of the reference options.

Next, I thought of using Feeds Tamper to convert the URL into the correct target ID. Feeds Tamper comes with so many options for manipulating data prior to saving that a custom Tamper plugin is usually unnecessary. Options include:

- converting dates to timestamps and vice versa
- making the field required, so the item does not get processed if empty or setting a default value
- exploding and imploding strings; array filtering or making unique
- applying math operations or formatting numbers
- converting to numbers or booleans
- formatting strings by changing cases, trimming, padding, or truncating
- finding and replacing, either with string manipulation or regular expressions
- Keyword filtering
- encoding and decoding (url encoding, serialization, HTML entities, strip tags, and JSON encoding)
- converting country names to ISO codes

While Feeds Tamper can solve almost any data transformation challenge, in this case, I could not find any pre-baked options that could solve my use case. I needed to look up content based on an alias URL and fetch a target ID to pass to an entity reference field. I had no choice but to create my own manipulation through a custom tamper plugin. There are many examples of tamper plugins in the Feeds Tamper module itself. Luckily, the process is dead simple!

Creating the Tamper Plugin

To create a plugin, you need to do three things:

1. **Create a custom module and enable it**: At a minimum, you will need to create a folder with your module name somewhere in /modules, such as /modules/custom/my_tamper. Inside this folder, you will need a .info.yml file describing the module. For example, my_tamper.info.yml could look like this:

```
name: 'My Tamper'
type: module
description: 'Provides custom feeds tamper plugins'
core_version_requirement: ^9.3 || ^10
package: 'Custom'
```

2. **Create a Plugin class in the correct namespace**: You will need to create a class extending TamperBase, in the namespace Drupal/my_tamper/Plugin/Tamper. Because Drupal uses the PSR-4 namespace conventions, this translates to the file location /modules/custom/my_tamper/src/Plugin/Tamper.
3. **Add the correct plugin annotation**: Drupal uses this annotation to discover the plugins. It should contain the plugin type (in this case, `@Tamper`) with a block specifying a unique plugin id, such as "url_to_article_ref." A label, description, and category will be used to place the tamper in the UI.

Listing 1.

```
1. <?php
2.
3. namespace Drupal\my_tamper\Plugin\Tamper;
4.
5. use Drupal\tamper\TamperBase;
6.
7. /**
8.  * Plugin implementation for converting URL into Article
                                          field_id.
9.  *
10. * @Tamper(
11. *   id = "url_to_article_ref",
12. *   label = @Translation("Convert URL into Article
                                          field_id"),
13. *   description = @Translation("Convert URL into Article
                                          field_id"),
14. *   category = "Text"
15. * )
16. */
17. class UrlToID extends TamperBase {}
```

That's all! These three steps are sufficient to create the plugin. So far, it doesn't actually do anything. Still, you should be able to enable the my_tamper module (either through the Extend UI or using `drush pm-enable my_tamper`) and find the new tamper in the list of plugins you can add for a field when editing a feed type.

Injecting Services

I like to begin by injecting the services I will need. Our tamper will need to convert a path alias into an actual node ID in order to create an entity reference. A quick google search (or perhaps an assist from GitHub Copilot) reveals that the service that does this is PathAliasManager.

Injecting the service is not technically necessary, but it is a best practice. You could use it statically by running `\Drupal::service('path_alias.manager')->getPathByAlias($data)`. However, a better approach is always to use dependency injection when possible. One of the many benefits of dependency injection when unit testing your class is you can simply mock the service and pass it to your constructor because you don't need to test core code, only your own.

For plugins, injecting a service is relatively straightforward. First, adjust your class to implement *ContainerFactoryPluginInterface*. This interface requires that you add a constructor and a *create* function. When the class gets instantiated, the create function (which has access to the service container) will be the function making the object, passing to your constructor both the original parameters required by the Plugin base class and the services your custom plugin class requires. In the constructor, you will first use the original parameters to call parent::construct, then assign the services to your own class properties for later use.

Note that the constructor does not take an object of type AliasManager, but rather AliasManagerInterface. That is a key feature of dependency injection. It allows you not to be bound to one specific class- you could use a different alias manager, or a mock one, as long as the object you use implements the AliasManagerInterface. The code looks like this:

Listing 2.

```
1. class UrlToArticleRef extends TamperBase implements
                          ContainerFactoryPluginInterface {
2.
3.   /**
4.    * Path Alias.
5.    *
6.    * @var Drupal\path_alias\AliasManagerInterface
7.    */
8.   protected AliasManagerInterface $pathAliasManager;
9.
10.  /**
11.   * Constructs a new URL To Article Ref Tamper object.
12.   *
13.   * @inheritDoc
14.   */
15.  public function __construct(array $configuration,
                $plugin_id, $plugin_definition,
                SourceDefinitionInterface$source_definition,
                AliasManagerInterface $path_alias_manager) {
16.    parent::__construct($configuration, $plugin_id,
                          $plugin_definition,$source_definition);
17.    $this->pathAliasManager = $path_alias_manager;
18.  }
19.
20.  /**
21.   * Dependency injection for Article Ref tamper object.
22.   *
23.   * @inheritDoc
24.   */
25.  public static function create(ContainerInterface
                $container, array $configuration,
                $plugin_id, $plugin_definition) {
26.    return new static(
27.      $configuration,
28.      $plugin_id,
29.      $plugin_definition,
30.      $configuration['source_definition'],
31.      $container->get('path_alias.manager')
32.    );
33.  }
```

Using PHP 8? In that case, you don't need to bother with creating the class property or assigning it in the constructor. Instead, you can rely on constructor property promotion. Simply by declaring the parameter as protected in the constructor, PHP will automatically create the class property and assign the value for you. By using property promotion, we can reduce the code to:

Listing 3.

```
1. class UrlToArticleRef extends TamperBase implements
                          ContainerFactoryPluginInterface {
2.
3.   /**
4.    * Constructs a new URL To Article Ref Tamper object.
5.    *
6.    * @inheritDoc
7.    */
8.   public function __construct(array $configuration,
           $plugin_id, $plugin_definition,
           SourceDefinitionInterface $source_definition,
         private AliasManagerInterface $path_alias_manager) {
9.     parent::__construct($configuration, $plugin_id,
           $plugin_definition, $source_definition);
10.  }
11.
12.  /**
13.   * Dependency injection for Article Ref tamper object.
14.   *
15.   * @inheritDoc
16.   */
17.  public static function create(
           ContainerInterface $container,
           array $configuration, $plugin_id,
           $plugin_definition) {
18.    return new static(
19.      $configuration,
20.      $plugin_id,
21.      $plugin_definition,
22.      $configuration['source_definition'],
23.      $container->get('path_alias.manager')
24.    );
25.  }
```

Overriding the Tamper Function

You now have an active plugin with an injected AliasManager. The final step is to perform the data manipulation to convert a path alias URL into a target node ID. For this, you must override the tamper function. This function hands you the data as a parameter. Simply manipulate it as needed and return the adjusted data. In this case, you can use pathAliasManager to load the unaliased path based on the alias. That unaliased path, in the form **/node/[ID]**, will give you the node ID you can hand off to the entity reference field.

What should you return if the ID cannot be found? Since Internal Link is a required field, you do not want to import the article if you can't create the entity reference. In this case, we want to skip importing the entire record. We *could* throw a *SkipTamperItemException*. The Required tamper in the Feeds Tamper plugin folder provides a great example of doing just that. But you may want to reuse this tamper on other fields or content types, and throwing an exception may not always be the desired behavior. Instead, you can simply return an empty string so that feeds that want to process the item can do so. And for this particular internal_link field, you can add an additional "required" tamper plugin to make sure the item won't import without the field.

Because the data source might contain messy URLs with *GET* parameters that would cause a mismatch with the stored path alias, some initial cleanup is called for:

```
$data = strtok($data, '?');
```

Then you can match the clean url to a node alias in the system and get the Drupal path by calling AliasManager's *getPathByAlias* function:

```
$path = $this->pathAliasManager->getPathByAlias($data);
```

Finally, the node ID can be extracted from the unaliased Drupal path with regular expression matching, taking advantage of the fact that unaliased Drupal paths are in the format **/node/[ID]** where ID is one or more digits. As a regular expression, one or more digits is , and we can add parenthesis around that part to "capture" it:

```
if (preg_match('/node\\/(\\d+)/', $path, $matches)) {
      return $matches[1];
    }
```

Note that preg_match always returns the full match as array index 0, so to get the first captured portion, the node ID, you return $matches[1].

The complete code for the tamper function might look like this:

Listing 4.

```
1.   /**
2.   * {@inheritdoc}
3.   */
4.   public function tamper($data,
                TamperableItemInterface $item = NULL) {

5.   // Dump any get parameters on the URL for a clean alias.
6.    $data = strtok($data, '?');
7.    $path = $this->pathAliasManager->getPathByAlias($data);
8.    if (preg_match('/node\\/(\\d+)/', $path, $matches)) {
9.      return $matches[1];
10.   }
11.
12.   return '';
13.  }
```

Our Feeds Tamper plugin is an extremely simple one. For more complex tampers, you may want to add configuration options. To do so, you would implement *buildConfigurationForm*, *submitConfigurationForm*, and *defaultConfiguration*. Again, there are many great examples of this in the Feeds Tamper module plugin folder.

Here is the final code for the entire plugin:

The real power of the plugin system comes from reusability. If you used this type of URL field in one Drupal 7 content type, chances are you used it in others, and you will want to apply the same tamper to those other imports. Using this plugin as a template, you can easily build your own tool kit of tamper plugins to supplement the already-comprehensive set of tampers baked into the module.

Doug Groene switched careers from attorney to web developer and never looked back. He has been working with Drupal CMS for over a decade and is currently a lead software developer with College Board. When not coding, Doug enjoys gaming, trips to the Space Coast, watching hockey, and spending time with his wife and son. @DougDevPHP

Listing 5.

```
<?php

namespace Drupal\apcentral_course\Plugin\Tamper;

use Drupal\Core\Entity\EntityTypeManagerInterface;
use Drupal\Core\Plugin\ContainerFactoryPluginInterface;
use Drupal\path_alias\AliasManagerInterface;
use Drupal\tamper\SourceDefinitionInterface;
use Drupal\tamper\TamperableItemInterface;
use Drupal\tamper\TamperBase;
use Symfony\Component\DependencyInjection\ContainerInterface;

/**
 * Plugin implementation for converting URL into
                        Article field_id.
 *
 * @Tamper(
 *   id = "url_to_article_ref",
 *   label = @Translation("Convert URL into
                        Article field_id"),
 *   description = @Translation("Convert URL into
                        Article field_id"),
 *   category = "Text"
 * )
 */
class UrlToArticleRef extends TamperBase implements
                        ContainerFactoryPluginInterface {

  /**
   * Constructs a new URL To Article Ref Tamper object.
   *
   * @inheritDoc
   */
  public function __construct(array $configuration,
                        $plugin_id, $plugin_definition,
  SourceDefinitionInterface $source_definition,
       private AliasManagerInterface $path_alias_manager) {
    parent::__construct($configuration, $plugin_id,
              $plugin_definition, $source_definition);
  }

```

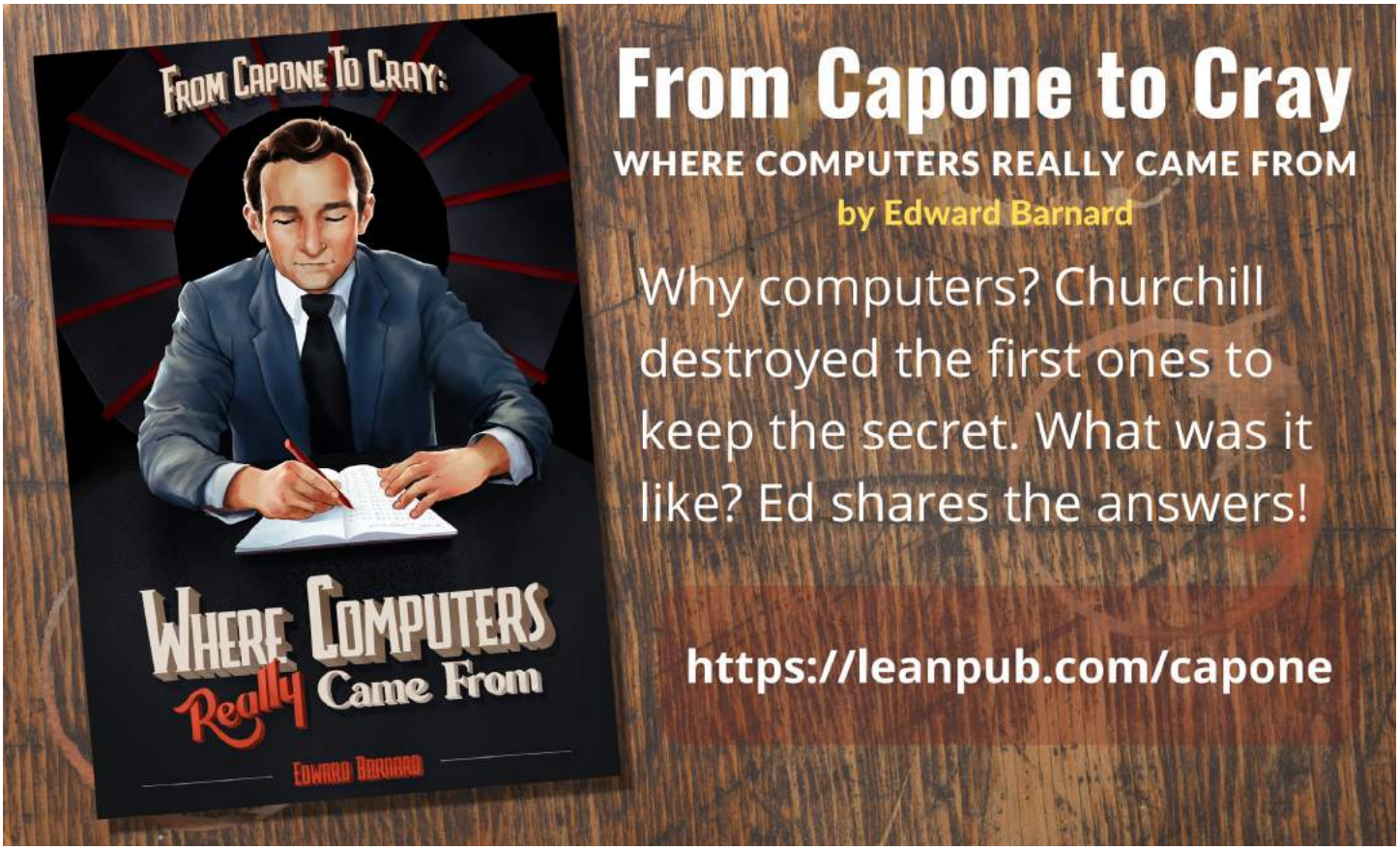
Listing 5 continued.

```
  /**
   * Dependency injection for Article Ref tamper object.
   *
   * @inheritDoc
   */
  public static function create(
              ContainerInterface $container,
              array $configuration, $plugin_id,
              $plugin_definition) {
    return new static(
      $configuration,
      $plugin_id,
      $plugin_definition,
      $configuration['source_definition'],
      $container->get('path_alias.manager')
    );
  }

  /**
   * {@inheritdoc}
   */
  public function tamper($data,
              TamperableItemInterface $item = NULL) {
    // Dump any Get paramaters on the URL to form a clean alias.
    $data = strtok($data, '?');
    $path = $this->pathAliasManager->getPathByAlias($data);
    if (preg_match('/node\\/(\\d+)/', $path, $matches)) {
      return $matches[1];
    }

    return '';
  }

}
```


From Capone to Cray:
Where Computers Really Came From
Edward Barnard
From Capone to Cray
WHERE COMPUTERS REALLY CAME FROM
by Edward Barnard
Why computers? Churchill destroyed the first ones to keep the secret. What was it like? Ed shares the answers!
https://leanpub.com/capone


Listen to Episode 74
Another Bright Idea
In this month's podcast, Eric and John review the articles in the June 2022 release including Event-Driven Programming and A Night with Symfony
Hosted By:
Eric Van Johnson and John Congdon
php[podcast]
php[architect]
Another Bright Idea
Illuminating Smart Light Bulbs with PHP
Building Solid and Maintainable PHP Apps
https://phpa.me/podcast-ep-74

What is Git Doing?

Chris Tankersley

When developers get started, the first thing they want to do is write code. Writing code is the fun part of the job or hobby, and as developers, we love to type in some text, hit refresh, and see our changes. Most of the time, the code we write does not work, so we have to keep trying until we get it right. Once it works, we cherish that code.

Then comes the inevitable time that we need to change that code. When new developers are starting out, the first thing many of us do is just edit the code. As luck has it, something breaks, and we might not know what exact change broke things. We CTRL-Z to start undoing things, but can we undo enough to get back?

The next time we find ourselves in this situation, we make a backup. `index.php` becomes `index-20000101.php`, then make our fix. Then something else needs to be fixed, so we get `index-20000512.php`, then `index-2000051201.php`, and so on and so on. Our workspace is littered with ghosts of backups past.

At some point, we learn about version control. Version control is a system that keeps track of changes that are made to files. We get to move away from keeping multiple copies of a file to allowing a system to manage the changes. The first time you have to revert a change, and it works properly, is glorious.

Today the most dominant version control system is Git[1]. We do not think about it much, but what exactly is it doing while we write all this wonderous code?

CVS and RCS

For me, I first learned about version control through the website Sourceforge. Sourceforge was a website that allowed open-source software hosting alongside the source code. You could store and maintain your source code through them using CVS, the Concurrent Versions System. CVS was a front-end for an older system called RCS, or Revision Control System.

RCS managed local versioning for individual files, and CVS expanded on that to allow project-level versioning as well as a client-server model that allowed changes to be stored remotely. Under the hood, RCS was keeping track of changes, and CVS helped make it "easy" to share the files and maintain project-level histories. A developer could easily "tag" a series of files as a version, and users and developers could just as easily jump back to those tags.

How did this all work? RCS itself used a reverse-delta system where it would store the current version of a file and the delta, or difference, between the previous version and the current file. The delta could be applied to turn the file into an older version. Multiple deltas could be stored and re-applied to go back further in history.

Since CVS was just a front-end for RCS, CVS worked this same way but allowed a user to work with multiple files at the same time and had a more sophisticated way for multiple users to change the same file. RCS was intended to only allow for a single person to change a file.

So, in either case, when you needed to revert a file, a changeset was applied to the current file. If you needed to go back multiple versions, it was possible by applying multiple deltas to go backward. This system allowed saving space while preserving speed, as the authors assumed that most reverts would be against the current version and only go back a few revisions.

RCS (and SCCS, the Source Code Control System it replaced) was considered a First Generation revision control system. There were others, but they showed that the idea of version control systems had merit. First Generation systems also tended to just work on individual files.

CVS was a Second Generation system. Second-generation systems were more identified with features like the ability to work on multiple files at once, as well as the addition of a centralized system. Where RCS stored everything locally, CVS allowed you to have the main copy on one machine and mechanisms for downloading the copies to other systems. As development moved first to the network and eventually the internet, second-generation systems took hold.

SVN

Another extremely popular second-generation system was Subversion, commonly called SVN. While the Subversion of today looks and acts much more like Git, Subversion originally took many cues from CVS. The main draw for SVN was the fact it supported atomic commits.

A major drawback to CVS was that it allowed partial commits. If you had a commit that contained twelve files, but during the save, your network dropped, and only four of the files had updated, CVS did not care. Your central repository would have the four updated files, and it would be on you to figure out how to reconcile this.

SVN considered commit operations as atomic. If this same situation happened with SVN, the central server would just revert the entire change. Doing so was essential as SVN

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

required you to have network access when you committed files. Your local copy was strictly available for you to edit the files, and the storage of the changes was done remotely.

SVN handled changes by keeping track of changes through various special folders. Individual file deltas were handled by storing the original copy of a file, and subsequent changes were stored as deltas. This design helped save space but increased checkout times as the current version of a file needed to be rebuilt. Every 1023 revision of a file was stored as a full copy of the file again to help balance out checkout times.

SVN became one of the most popular version control systems throughout the early 2000s. It, and other second-generation systems like CVS, grew in popularity as development moved toward network-based collaboration. Centralized repositories could easily be backed up, and developers could easily get copies of the code no matter where they were.

Git and the Third Generation

We've now arrived at Git. Third Generation version control systems bring in the idea of decentralization and a greater emphasis on copies of repositories, called forks, rather than a single centralized repository (which GitHub has done a great job making us all forget). Users download and keep full copies of the repositories locally, and each collaborator brings in changes from other collaborators.

Git has had explosive growth since it was declared the official version control system for the Linux Kernel despite its reputation among many users as being very complex. The inherent simplicity of forking, branching, and sharing code that is built into Git gave GitHub an advantage over other source control hosting systems. This growth has also been helped by a massive push by GitHub to court open source developers and, honestly, a much better interface for collaboration compared to any other system.

Git was designed for decentralized collaboration. Unlike SVN and CVS, where there is a single canonical source for a project, projects using Git have users download entire copies of each repository, called forks. Changes to the repository are stored locally, and then collaborators can push or pull the changes to other forks.

Unlike earlier systems, Git does not use deltas for file history. Git stores entire snapshots of a file when a file is committed to history. These snapshots are called "blobs" and are treated as just raw data. They are referenced only by a hash that Git generates based on the compressed data. When a file is updated, a new blob is created.

Doesn't Git show changes as diffs? Whenever you do a `git diff` command, the diff you see is actually generated on demand instead of using a saved diff. While Git will compress individual files to save some space, Git sacrifices storage space for easier manipulation.

Blob information is stored as a "tree." These trees are stored as snapshots of the repository and contain more information about the structure of the project and its files and folders. These trees are turned into "commits," which contain additional metadata about what trees are part of the commit, as well as notes and author information. It is at this level that Git stores parent information for a commit as well.

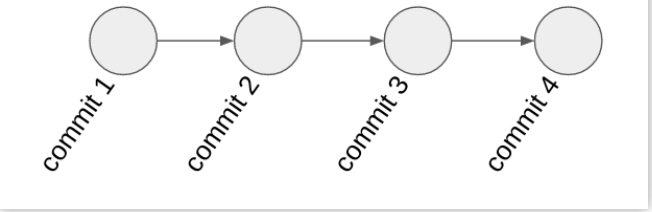
A Git repository is just a bunch of snapshots of files. So what happens when you actually do stuff in Git?

Git is a History Book

Whenever someone shows things like branching and histories, you will see something like Figure 1.

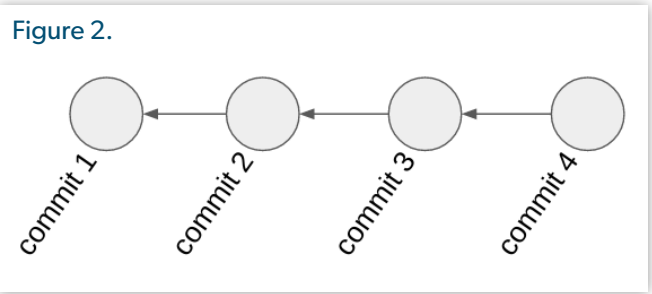
This is because we describe and then illustrate the actions

Figure 1.



we are taking. You create a Git repository, then add a commit ("commit 1"). Then you add "commit 2", then "commit 3", and finally "commit 4". It becomes natural to show "commit 1", pointing to "commit 2", and so on. The only problem is that, structurally, this is backward. See Figure 2.

Figure 2.



In reality, Git stores where the repository currently is. This location is called `HEAD`. This commit, like all other commits, keeps track of its parent. Each commit has no knowledge of any child commits, nor any parent commits further back in time. Git pieces all of this together by following the parent chain.

When you commit a series of changes, Git collects the current versions of all the files (or blobs) and the tree structure to represent the current file and folder structure into a commit. This new commit then points back to the parent commit from which this new commit was generated. Remember, this system is not based on any sort of diff or delta. Git just simply attaches the parent commit hash to the new commit.

When we create "commit 1", this commit has no parent. Git understands this is the oldest commit in the history. When we make "commit 2", we record its parent as "commit 1." Git then marks "commit 2" as HEAD. The same thing happens with "commit 3":

- Package the tree and blobs as a commit
- Mark this commit's parent as "commit 2"
- Mark "commit 3" as the new HEAD

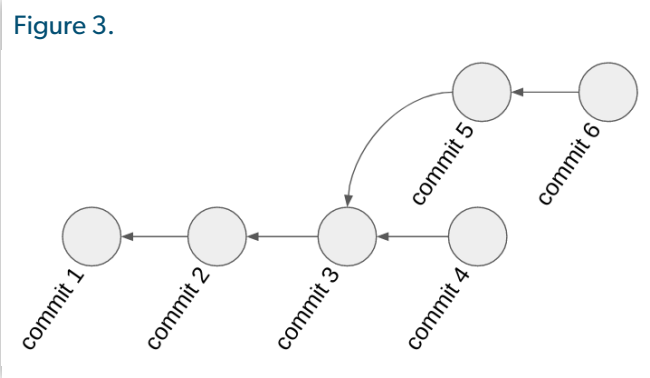
When you do `git log`, Git looks at the current commit and then moves to the parent commit. This commit is then looked at, and then Git navigates to the next parent commit. This pattern keeps repeating until you find a commit with no parent, which means it reached the original commit.

Most tools represent this as an ascending timeline, but internally it is actually a descending timeline.

Branches

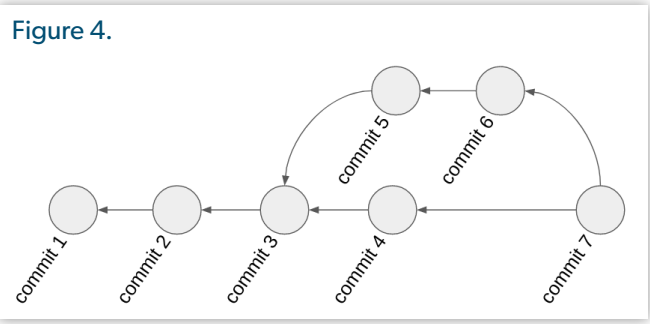
This setup is one of the reasons that branching in Git is so easy. All you have to do is keep track of parents, and multiple commits can point to a single commit. A branch is essentially just a commit that points to a parent that has other children as shown in Figure 3.

Figure 3.



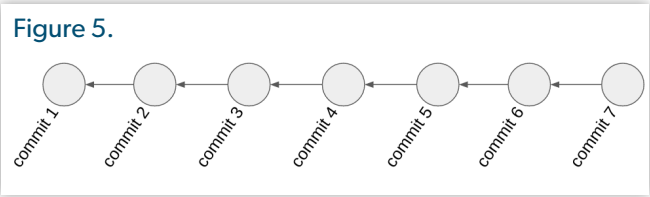
When you do a merge, two things can happen. In projects where branches are constantly updated, you can end up with a "merge commit."—a special commit that takes two branches, determines the best way to merge files together, and creates a special commit with multiple parents. This type of commit happens when multiple changes to common files need to be reconciled. A merge commit is also where you may run into the dreaded "merge conflict." See Figure 4.

Figure 4.



The other type of merge is called a "fast forward merge." In this case, Git determines that no file reconciliation needs to happen, and it is safe just to move the branch commits behind the commits in the branch being merged into (Figure 5). The parent of the first branch commit is changed to the last commit in the branch it is merging into. The branch is simply absorbed into the branch it was being merged into.

Figure 5.



In either case, Git's setup of using the parent as a reference point instead of a child makes it very easy for it to handle branches. The second part that makes this work is that since each commit is a snapshot of the files, Git can use various algorithms to try and merge files together instead of trying to reconcile a bunch of diffs. Git can take each version of the file, compare them to their parents to see what lines changed, and then figure out if it is safe to merge those changes.

Git is a Time Machine

Git is an incredibly powerful tool that has a very simple concept. Since it stores full copies of files, it is very easy to show you an older version of a file. The way Git references commits by a parent instead of a child makes it very easy for Git to show you the history of any file or branch in the system.

This setup allows us to manipulate the history of our repository without anyone noticing and without losing any work. Next month we will take a deep dive into some of the more complicated but very useful things we can do by messing with these commit histories.

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

Demystifying Multifactor Authentication

Eric Mann

Authentication by way of a username and password is well understood. Adding an extra authentication factor—like a smartphone—to the mix helps strengthen a login flow. But what exactly is an authentication factor, and what are the trade-offs between each one?

A few weeks ago, I spent an inordinate amount of time debating multifactor authentication (MFA) on Mastodon. The conversation had a fair amount of friction, mostly due to those of us involved failing to work from the same basic foundation. In the end, we actually *agreed* with one another; we'd spent a few days debating in circles before that was evident.

MFA is apparently difficult for even those who focus on security for a living to understand.

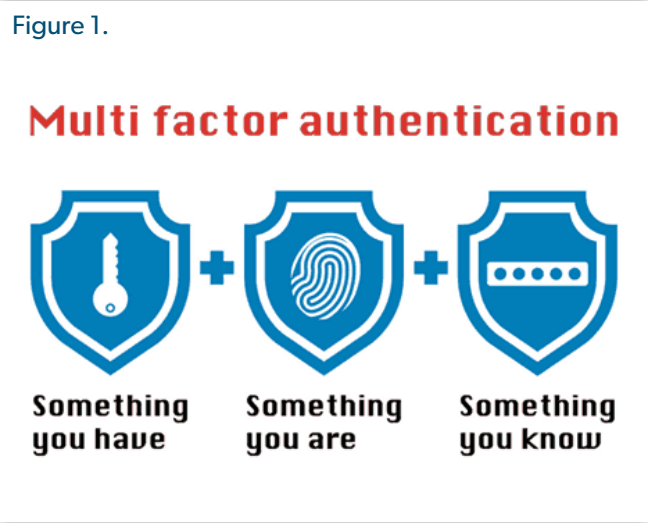
Rather than live through yet another public disagreement, I want to clarify what exactly multifactor authentication really is for those new to the field. This way, you can look at any authentication scheme with a certain level of discernment and understanding.

Basic Authentication

There are (at least) two factors involved in any form of authentication: identification and an indication of intent.

Identification is the "something you are" in the traditional authentication triad—your username or some other identifier that distinguishes you from any other user or participant in a system.

Figure 1.



The authentication triad consists of three components: something you have, something you are, and something you know. All three are required for strong authentication.

Indication of intent is a way to verify that you *meant* to authenticate. Most commonly, it is leveraging some hidden knowledge you possess that you must proactively retrieve and provide to the system performing the authentication—like your password. In traditional systems, this is the "something you know" component of the triad.

Most basic authentication systems we use today feature these two components: identification and verification of intent. To make them stronger, we add a *third* factor, a redundant verification of intent. Often this additional factor provides the "something you have" component of the triad by proving you have a physical device (i.e., your phone) and can actively verify your intent to authenticate.

When used together, these multiple factors build the foundation of a strong authentication scheme. But there's still a mountain of confusion about what each really represents and how many factors are necessary.

Magic Links and Email

In another life[1], I built software that powered simple authentication schemes for users leveraging so-called "magic links." These links were one-time use URLs that an application would email to you to allow you to authenticate. If you use Slack, you might have used a similar mechanism to log in to your workspace.

The power of magic links is that you don't need to remember a password. Purportedly this makes an application *more* secure by preventing you from reusing a password from another service or selecting a too-weak password easily guessed by an attacker. In reality, it merely trades one component of the triad for another.

You are no longer leveraging the "something you know" component. You've swapped it for "something you have": access to a particular email account.

In reality, magic links are no more or less secure than a password. You likely still use a password to log in to your email server. A magic link shifts the burden of keeping users' accounts secure to the email provider; a weak email password is just as dangerous as a weak application one. However, the system makes authentication *easier* for many users to log in by identifying themselves and then clicking a link to proactively assert they intend to authenticate.

Magic links leverage multiple authentication factors—but usually only two. Unless you add an extra factor to

1 *In another life:* https://speakerdeck.com/ericmann/going-password-free

redundantly verify intent, you haven't necessarily made your application any more secure.

Identification Is Not Authentication

Biometric verification is one of the more popular features of modern smartphones and computers. However, I want to be perfectly clear here; **biometrics are not authentication!**

Fingerprints are static, as is your retina. In the overwhelming majority of cases, so is the structure of your face. Biometrics are a way to *identify* you, but they are entirely passive. There is nothing about identification that verifies your *intent* to do something.

Using biometrics for authentication merely proves your face is in front of a particular screen (or your finger is on the phone). It does nothing to prove you intend to authenticate to any one system.

Although, as a convenience factor, it's quite reliable for unlocking a machine that otherwise houses a secure vault of passwords. Using a fingerprint to unlock a 1Password vault doesn't log you in directly—it decrypts the vault and presents you the opportunity to assert your intention to log in.

Likewise, leveraging facial recognition to unlock a MacBook to access a Fido2 token stored on the machine[2] is *still* leveraging multiple factors to authenticate you. Your face provides the "something you are" factor while the Fido2 token (and the proactive interaction with the system to use it) leverages a "something you have" factor.

Unfortunately, you're still only leveraging two factors in the authentication.

2FA vs. MFA

Many developers use the acronyms "2FA" and "MFA" interchangeably. They accept that a username (or email) and password are the default for authentication, and a second factor is required beyond your password to log in. This oversimplification makes the multiple factors involved hard to understand.

The fact is that each authentication factor has its strengths and its weaknesses. Multiple factors working together yield a truly strong and secure system

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

2 Fido2 token stored on the machine: https://phpa.me/malwarebytes-passwordless

PHP from Virtual Machine to Docker

Joe Ferguson

Containers are here to stay, and if you've been putting off learning how to migrate a PHP application from a VM to containers, we have you covered this month as we migrate a long-standing Laravel application from running on Linux via Apache and MariaDB. Our application also utilizes Redis for caching, so we need to include this in our container plan.

Docker Compose[1] is a tool for defining how containers should be run. As soon as you get past the `docker run -it` basics, it becomes extraordinarily easier to manage container services via a compose file than arbitrary docker run commands. Why not jump straight into Kubernetes[2] (k8s)? While k8s has become the de facto standard for container orchestration at scale, and while k8s is a great container orchestration tool, we're going to focus on Docker Compose as it's the less complex option; unless you are building the next big app. Docker Compose is a great starting place to get your feet wet with real-world container usage and understand the basics of shifting these services from running directly on a host together to running each in their own container, which we can easily pick up and move to any host capable of running our compose file and images.

Our application is a user management system that serves an API consumed by an on-site authentication system guarding physical access to a shared workshop. Admins will associate RFID tokens with users, and then users can manage their information in the web application. Currently, the application runs on a Ubuntu virtual machine running PHP 7.4 (LTS Ubuntu), MariaDB, and Redis. We want to migrate from this VM host into another host running Docker Compose. We'll need to build a container image for our application which installs and configures PHP, Apache, and our application. Next, we'll use a generic MariaDB container image to act as our database service.

Additionally, we will configure Redis to be available for our application to be used as a caching layer. While the source code of the application I'm using isn't publicly available, these instructions will support any PHP application already using MariaDB and Apache. Don't have a project? Create a new Laravel project with `composer create-project laravel/laravel app` or a new Symfony project with `composer create-project symfony/skeleton app`.

How do we containerize Apache and our application? You *could* start from scratch and build your container by copying your VM configuration steps. If you have existing configuration management tooling, this will be straightforward. You would start with a base Ubuntu image (or your preferred Linux flavor) and build up from there. If you don't have a place to start, the Snipe-IT[3] asset management project has an extensive `Dockerfile` and associated configuration files. These files represent the build steps to produce ready-to-run container images using PHP and Apache on top of Ubuntu Linux. Snipe-IT happens to be a modern PHP application built on the Laravel framework, so we can easily follow their examples for our application.

The Dockerfile

The source of the container image is the `Dockerfile`. The `Dockerfile` reads similar to shell scripts but uses a specific syntax to take actions on existing containers to provision them just as you would a traditional VM or server. This explanation sounds more complicated than it is; the task is to install the Ubuntu packages you need for PHP with Apache and build our application as we normally would.

As shown in Listing 1, our `Dockerfile` starts by extending the `ubuntu:20.04` image, which gives us a traditional Ubuntu environment in our container. We'll start by marking our changes as non-interactive so that we're not prompted for any questions; then, we will proceed to install packages for PHP and Apache. After package installation, we install Pear and install the `gd` and `bcmath` PHP extensions. Then we update our `php.ini` configuration files and set up our Apache sites. To keep our application's folders organized, we place our supporting files, such as our Apache virtual host configurations and supporting shell scripts, in `docker/`. At the very end of our `Dockerfile`, we state our `CMD` and expose ports 80 and 443, telling the container to run the `startup.sh` script by default when we run the container. All of these files come together as a container image with our application, Apache, and PHP ready to run.

We can use `` `docker build . -t svpernova09/rfid` `` to build our application image. `` `docker build .` instructs` `` Docker to build in the current directory, which will read our `Dockerfile`, and `-t svpernova09/rfid` is what we're going to *tag* the container image we build. Tagging our container is important because this is how we share container images with registries. A Docker registry is a service that tracks docker images and their versions. Think of it like a Packagist for your images. You can use the default DockerHub registry as I do, or you

1 *Docker Compose:* https://docs.docker.com/compose/

2 *Kubernetes:* https://kubernetes.io/

3 *Snipe-IT:* https://github.com/snipe/snipe-it/tree/master/docker

can use your own by using the full path to your registry in addition to the user and image name.

Note: It's incredibly important not to publish your container images to a public registry as you would be exposing source code and possibly configuration secrets. Take caution!

Listing 1.

```
1. FROM ubuntu:20.04
2.
3. RUN export DEBIAN_FRONTEND=noninteractive; \\
4.     export DEBCONF_NONINTERACTIVE_SEEN=true; \\
5.     echo 'tzdata tzdata/Areas select Etc' | debconf-set-selections; \\
6.     echo 'tzdata tzdata/Zones/Etc select UTC' | debconf-set-selections; \\
7.     apt-get update -qqy && apt-get install -qqy software-properties-common \\
8.     && add-apt-repository -y ppa:ondrej/php \\
9.  && apt-get install -qqy --no-install-recommends apt-utils apache2 apache2-bin \\
10. libapache2-mod-php8.0 php8.0-curl php8.0-ldap php8.0-mysql php8.0-gd php8.0-xml php8.0-mbstring \\
11. php8.0-zip php8.0-bcmath patch curl wget vim git cron mysql-client supervisor cron gcc make \\
12. autoconf libc-dev pkg-config libmcrypt-dev php8.0-dev ca-certificates unzip \\
13.  && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
14.
15. RUN curl -L -O <https://github.com/pear/pearweb_phars/raw/master/go-pear.phar>
16. RUN php go-pear.phar
17.
18. RUN phpenmod gd
19. RUN phpenmod bcmath
20.
21. RUN sed -i 's/variables_order = .*/variables_order = "EGPCS"/' /etc/php/8.0/apache2/php.ini
22. RUN sed -i 's/variables_order = .*/variables_order = "EGPCS"/' /etc/php/8.0/cli/php.ini
23.
24. RUN useradd -m --uid 1000 --gid 50 docker
25.
26. RUN echo export APACHE_RUN_USER=docker >> /etc/apache2/envvars
27. RUN echo export APACHE_RUN_GROUP=staff >> /etc/apache2/envvars
28.
29. COPY docker/000-default.conf /etc/apache2/sites-enabled/000-default.conf
30.
31. RUN mkdir -p /var/lib/ssl
32. COPY docker/001-default-ssl.conf /etc/apache2/sites-available/001-default-ssl.conf
33.
34. RUN a2enmod ssl
35. RUN a2ensite 001-default-ssl.conf
36.
37. COPY . /var/www/html
38. RUN a2enmod rewrite
39. WORKDIR /var/www/html
40.
41. COPY docker/docker.env /var/www/html/.env
42. RUN chown -R docker /var/www/html
43.
44. COPY --from=composer:latest /usr/bin/composer /usr/bin/composer
45.
46. USER docker
47. RUN composer install --no-dev --working-dir=/var/www/html
48. USER root
49.
50. COPY docker/startup.sh docker/supervisord.conf /
51. COPY docker/supervisor-exit-event-listener /usr/bin/supervisor-exit-event-listener
52. RUN chmod +x /startup.sh /usr/bin/supervisor-exit-event-listener
53.
54. CMD ["/startup.sh"]
55.
56. EXPOSE 80
57. EXPOSE 443
```

Figure 1.

```
┌─(~/Code/rfid)(container ??:2 M:3 S:10)──────────────────────────────────────────────────────────────────────────────────────────────────┐
└─(1)─ docker build .                                                                                                                    ┘
[+] Building 61.5s (6/34)
 => [internal] load build definition from Dockerfile                                                                                0.0s
 => => transferring dockerfile: 2.23kB                                                                                              0.0s
 => [internal] load .dockerignore                                                                                                   0.0s
 => => transferring context: 35B                                                                                                    0.0s
 => [internal] load metadata for docker.io/library/ubuntu:20.04                                                                     6.3s
 => CACHED FROM docker.io/library/composer:latest                                                                                   0.0s
 => => resolve docker.io/library/composer:latest                                                                                    1.0s
 => CACHED [stage-0  1/29] FROM docker.io/library/ubuntu:20.04@sha256:fd92c36d3cb9b1d027c4d2a72c6bf0125da82425fc2ca37c414d4f010180dc19  0.0s
 => [internal] load build context                                                                                                   0.0s
 => => transferring context: 37.82kB                                                                                                0.0s
 => [stage-0  2/29] RUN export DEBIAN_FRONTEND=noninteractive;     export DEBCONF_NONINTERACTIVE_SEEN=true;     echo 'tzdata tzdata/Areas select Etc  55.1s
```

Figure 1 illustrates using docker build to build our application image.

Once our image build is complete, we can inspect our image by running `docker run -it svpernova09/rfid /bin/bash` as shown in Figure 2.

We can see our application is in `/var/www/html` and we can check our framework version with `artisan --version`:

```
root@9b5ee6c28b85:/var/www/html# php artisan --version
Laravel Framework 9.8.1
root@9b5ee6c28b85:/var/www/html# php -v
PHP 8.0.19 (cli) (built: May 17 2022 18:48:59) ( NTS )
Copyright (c) The PHP Group
```

Figure 2. Running our freshly built image and inspecting our application folder

```
┌─(~/Code/rfid)(container ??:2 M:3 S:10)──────────────────────────────────────────────────────────────────────────────────────────────────┐
└─ docker run -it svpernova09/rfid /bin/bash                                                                                             ┘
root@9b5ee6c28b85:/var/www/html# ls -alh
total 1.9M
drwxr-xr-x 1 docker root 4.0K Jun  7 18:10 .
drwxr-xr-x 1 root   root 4.0K Jun  7 17:05 ..
-rw-r--r-- 1 docker root 1.7K Jun  7 14:36 .env
-rw-r--r-- 1 docker root  901 Feb 11 14:33 .env.example
-rw-r--r-- 1 docker root 213K Apr 23 04:16 .phpstorm.meta.php
-rw-r--r-- 1 docker root 2.0K Jun  7 17:23 Dockerfile
-rw-r--r-- 1 docker root  300 Feb 11 14:33 Envoy.blade.php
-rw-r--r-- 1 docker root 1.1K Feb 11 14:33 LICENSE.md
-rw-r--r-- 1 docker root 1.3K Feb 11 14:33 Vagrantfile
-rw-r--r-- 1 docker root 758K Apr 23 04:16 _ide_helper.php
-rw-r--r-- 1 docker root  177 Feb 11 14:33 after.sh
-rw-r--r-- 1 docker root 2.4K Feb 11 14:33 aliases
drwxr-xr-x 1 docker root 4.0K Jun  7 16:27 app
```

```
Zend Engine v4.0.19, Copyright (c) Zend Technologies
    with Zend OPcache v8.0.19, Copyright (c), by Zend Technologies
```

Supporting Services

With our application image built, we're ready to move on to our supporting services—MariaDB and Redis. We're going to introduce our `docker-compose.yml` file (Listing 2), which will act as our service definitions. We'll start with MariaDB, and we want to use a volume[4] with a specific name so we can easily identify *where* our database data is located. We could also map a specific directory on the host to serve as our storage location.

We're also utilizing `healthcheck`, which tells Docker how to test the container to see if it's working correctly. We use

4 *volume:* https://docs.docker.com/storage/volumes/

Listing 2.

```
1. mariadb:
2.     image: mariadb:10.7.4-focal
3.     volumes:
4.       - type: volume
5.         source: mariadb-data
6.         target: /var/lib/mysql
7.         volume:
8.           nocopy: true
9.     ports:
10.      - "33060:3306"
11.    healthcheck:
12.      test: mysqladmin ping -h 127.0.0.1 -u $$MYSQL_USER
                --password=$$MYSQL_PASSWORD
13.      interval: 3s
14.      timeout: 1s
15.      retries: 5
16.    env_file:
17.      - docker/docker.env
```

the `mysqladmin ping` command against our service every 3 seconds with a 1-second timeout, and we'll retry 5 times. This command will allow us to configure our application's image to wait for the database service to be healthy before launching, thus preventing our application from attempting to connect to a database service that isn't ready or healthy in this case. We'll pass our `docker.env` file containing all of our

Listing 3.

```
1. # MariaDB Parameters
2. MYSQL_ROOT_PASSWORD=ERIC_WAS_ROBBED_AS_RM
3. MYSQL_DATABASE=rfid
4. MYSQL_USER=rfid
5. MYSQL_PASSWORD=SUBSCRIBE_TO_PHP_ARCH
6. # Application Environment
7. DB_CONNECTION=mysql
8. DB_HOST=mariadb
9. DB_DATABASE=rfid
10. DB_USERNAME=rfid
11. DB_PASSWORD=SUBSCRIBE_TO_PHP_ARCH
12. DB_PREFIX=null
13. DB_DUMP_PATH='/usr/bin'
14. DB_CHARSET=utf8mb4
15. DB_COLLATION=utf8mb4_unicode_ci
16. APP_URL=https://rfid.secret.tld
17. APP_KEY=BENS_A_GOOD_RM
```

specific environment variables. This file would be your `.env` file updated for the container services.

We've also configured port `33060` on the host to `3306` in the container so we can connect to MariaDB from our host via `mysql -h localhost -P 33060`.

Docker Environment

Our `docker.env` file sets the root user's password and adds a user and database to be created on startup. We then need to pass the database credentials to our application and any other additional variables that we may need.

Up next is our Redis service, which will serve as our application cache. This service could easily be something like MongoDB or any other supporting service your application may need. If the services need to be run alongside the application, they should be defined in our `docker-compose.yml`. Alternatively, if you have a dedicated database host, you can skip the MariaDB portion of our compose file and pass in the credentials to your application as we have in `docker.env`.

Our Redis service definition uses a similar volume mount that will persist the contents to disk at specific intervals, so if we restart our containers, we won't lose the contents of Redis.

We're also exposing Redis' port at `6379`, which we can connect to if needed, similarly to how we can connect to MariaDB.

Listing 4.

```
redis:
    image: redis:6.2.5-buster
    ports:
      - "6379:6379"
    volumes:
      - type: volume
        source: redis-data
        target: /data
        volume:
          nocopy: true
```

Listing 5.

```
rfid_app:
    image: svpernova09/rfid:latest
    volumes:
      - ./logs:/var/www/html/storage/logs
    ports:
      - "8000:80"
    depends_on:
      mariadb:
        condition: service_healthy
      redis:
        condition: service_started
    env_file:
      - docker/docker.env
```

Listing 6.

```
version: '3'

services:
  rfid_app:
    image: svpernova09/rfid:latest
    volumes:
      - ./logs:/var/www/html/storage/logs
    ports:
      - "8000:80"
    depends_on:
      mariadb:
        condition: service_healthy
      redis:
        condition: service_started
    env_file:
      - docker/docker.env

  mariadb:
    image: mariadb:10.7.4-focal
    volumes:
      - type: volume
        source: mariadb-data
        target: /var/lib/mysql
        volume:
          nocopy: true
    ports:
      - "33060:3306"
    healthcheck:
      test: mysqladmin ping -h 127.0.0.1 -u
              $$MYSQL_USER --password=$$MYSQL_PASSWORD
      interval: 3s
      timeout: 1s
      retries: 5
    env_file:
      - docker/docker.env

  redis:
    image: redis:6.2.5-buster
    ports:
      - "6379:6379"
    volumes:
      - type: volume
        source: redis-data
        target: /data
        volume:
          nocopy: true

volumes:
  mariadb-data:
  redis-data:
```

Finally, we'll add our application service to the `docker-compose.yml` and notice we use `depends_on:` and have set MariaDB to `service_healthy`, which means our defined `healthcheck` must pass *before* the application service is started. The Redis service starts much faster than our database, so we set its condition to `service_started`.

Listing 7.

```
mysql -h 127.0.0.1 -u root -pERIC_WAS_ROBBED_AS_RM -P 33060
mysql: [Warning] Using a password on the command line interface can be insecure.
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 100
Server version: 5.5.5-10.7.4-MariaDB-1:10.7.4+maria~focal mariadb.org binary

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Running Our Services

Putting it all together, Listing 6 shows our complete `docker-compose.yml` file.

You should start to get the idea now with all of our services laid out next to each other. If you were expecting fancy networking statements and definitions, luckily, most of the networking complexity has been removed from current docker versions. By default, each service will have access to other services in the compose file. Because we're using named volumes, we need to define those in the `volumes:` section; otherwise, we're ready to test drive our compose file.

To start the services, we can use `docker-compose up` as shown in Figure 3.

Because we've configured a `healthcheck`, our RFID application service waits for MariaDB to be fully functional before starting. Since we're using a similar `startup.sh` script, we can check to ensure files are owned by the correct users, directories or paths exist, and we can run migrations to ensure the database is up to date with our application. In Figure 3, we can see the RFID application running our database migrations as the database was empty on our initial run.

Figure 3. docker-compose up showing Redis starting, MariaDB, then our RFID application

```
└─(130)─ docker-compose up
[+] Running 3/3
 ⠿ Container rfid-mariadb-1   Recreated
 ⠿ Container rfid-redis-1     Created
 ⠿ Container rfid-rfid_app-1  Recreated
Attaching to rfid-mariadb-1, rfid-redis-1, rfid-rfid_app-1
rfid-redis-1      | 1:C 07 Jun 2022 20:39:21.871 # oO0OoO0OoO0Oo Redis is starting oO
rfid-redis-1      | 1:C 07 Jun 2022 20:39:21.871 # Redis version=6.2.5, bits=64, comm
rfid-redis-1      | 1:C 07 Jun 2022 20:39:21.871 # Warning: no config file specified,
erver /path/to/redis.conf
rfid-redis-1      | 1:M 07 Jun 2022 20:39:21.871 * monotonic clock: POSIX clock_getti
rfid-redis-1      | 1:M 07 Jun 2022 20:39:21.871 * Running mode=standalone, port=6379
rfid-redis-1      | 1:M 07 Jun 2022 20:39:21.871 # Server initialized
rfid-redis-1      | 1:M 07 Jun 2022 20:39:21.871 # WARNING overcommit_memory is set t
sue add 'vm.overcommit_memory = 1' to /etc/sysctl.conf and then reboot or run the co
rfid-redis-1      | 1:M 07 Jun 2022 20:39:21.872 * Loading RDB produced by version 6.
rfid-redis-1      | 1:M 07 Jun 2022 20:39:21.872 * RDB age 57 seconds
rfid-redis-1      | 1:M 07 Jun 2022 20:39:21.872 * RDB memory usage when created 0.77
rfid-redis-1      | 1:M 07 Jun 2022 20:39:21.872 * DB loaded from disk: 0.000 seconds
rfid-redis-1      | 1:M 07 Jun 2022 20:39:21.872 * Ready to accept connections
rfid-mariadb-1    | 2022-06-07 20:39:21+00:00 [Note] [Entrypoint]: Entrypoint script
rfid-mariadb-1    | 2022-06-07 20:39:22+00:00 [Note] [Entrypoint]: Switching to dedic
rfid-mariadb-1    | 2022-06-07 20:39:22+00:00 [Note] [Entrypoint]: Entrypoint script
rfid-mariadb-1    | 2022-06-07 20:39:22+00:00 [Note] [Entrypoint]: MariaDB upgrade no
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] mariadbd (server 10.7.4-MariaDB-1:10
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Compressed tables use zlib 1
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Number of transaction pools:
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Using crc32 + pclmulqdq inst
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] mariadbd: O_TMPFILE is not supported
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Using Linux native AIO
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Initializing buffer pool, to
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Completed initialization of
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: 128 rollback segments are ac
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Creating shared tablespace f
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Setting file './ibtmp1' size
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: File './ibtmp1' size is now
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: 10.7.4 started; log sequence
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] Plugin 'FEEDBACK' is disabled.
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Loading buffer pool(s) from
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Warning] You need to use --log-bin to make
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] InnoDB: Buffer pool(s) load complete
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] Server socket created on IP: '0.0.0.
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] Server socket created on IP: '::'.
rfid-mariadb-1    | 2022-06-07 20:39:22 0 [Note] mariadbd: ready for connections.
rfid-mariadb-1    | Version: '10.7.4-MariaDB-1:10.7.4+maria~focal'  socket: '/run/mys
rfid-rfid_app-1   | Module ssl disabled.
rfid-rfid_app-1   | To activate the new configuration, you need to run:
rfid-rfid_app-1   |   service apache2 restart
rfid-rfid_app-1   | Migration table created successfully.
rfid-rfid_app-1   | Migrating: 2014_10_12_000000_create_users_table
rfid-rfid_app-1   | Migrated:  2014_10_12_000000_create_users_table (31.08ms)
rfid-rfid_app-1   | Migrating: 2014_10_12_100000_create_password_resets_table
```

With our services running, you can connect to the database directly as shown in Listing 7.

Now that we have all of our services configured and we're able to access our application at `https://rfid.secret.tld`, what's next? Where do we go from here? First things first, realize this is still somewhat early in the process. You still need to harden your passwords and sort out URLs. You also likely want to use something like Traefik[5] to route traffic to your application as well as handle your SSL certificates. You should also implement a reliable database backup and restoration plan; if something happens to the `MariaDB data`, you'll be able to recover from a backup. The next step would be running our `docker-compose.yml` as a Docker Stack[6], which gives the services a bit of stability instead of requiring a user to run `docker-compose up`. These are all fantastic next steps, and I hope you're inspired to test drive running your PHP applications with Docker Compose!

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

5 *Traefik:* https://traefik.io/

6 *Docker Stack:* https://docs.docker.com/engine/reference/commandline/stack_deploy/

PSR-7 HTTP Message Interface

Frank Wallen

This month we'll be looking at PSR-7, the HTTP Message Interface. Handling HTTP messages is crucial to web development. Request messages come into the server, are processed, and content is constructed and packaged into a response message and then sent back to the requesting entity. It sounds relatively simple on the surface, but it drives our internet experience. Requests and responses must be appropriately formed and follow the essential requirements and protocols. The intention of PHP-FIG's HTTP Message Interface[1] is exactly that, building structure around those requirements, so developers know what to expect and how to respond.

Let's take a quick look at a simple request:

```
POST /path HTTP/1.1
Host: foo.com

bar=baz&boz=100
```

The first line is the request line, starting with the method (POST), the path ('foo.com,' which can be a web server path or URI), and the protocol version (HTTP/1.1). The request line is followed by one or more header lines, an empty line, and finally, the message body. All this information is parsed and populated into a request object implementing `RequestInterface`.

A response looks like this:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Cache-Control: max-age=100

The body of the response.
```

The status line is the first line, describing the protocol version (HTTP/1.1), status code (200), and the 'reason' (OK), which is a readable description of the status code. Following the status line will be headers followed by a blank line and the actual content for display. This response would be properly generated by a response object implementing `ResponseInterface`.

The request and response HTTP messages can be complex when it comes to handling headers, URIs, file uploads, etc. In those components alone, there are many requirements and expectations described in a couple of RFCs:

- RFC7230[2] — The message syntax and routing rules for an HTTP Request
- RFC7231[3] — The rules, semantics, and contents for an HTTP Response

1 *Message Interface:* https://www.php-fig.org/psr/psr-7/

2 *RFC7230:* https://datatracker.ietf.org/doc/html/rfc7230

3 *RFC7231:* https://datatracker.ietf.org/doc/html/rfc7231

4 *PSR-7 package:* https://github.com/php-fig/http-message

5 *value objects:* https://en.wikipedia.org/wiki/Value_object

There is a lot of information there, and it is overwhelming. It would also very quickly fill the space for this column and could still be incomplete if we attempted to associate those requirements with the interface definitions within the PSR-7 package[4]. For that reason, we'll be talking about the reasoning behind the creation of the interfaces.

To start, the reason PHP-FIG decided to define the Message Interface is primarily the same as the other PSRs (before the popularity of today's MVC frameworks). Projects created their own implementations from scratch, which tended to make interoperability impossible or labor-intensive as one would have to create adapters to bridge the different request/response methods. When it comes to server-side responses, it's important to control the flow. Content cannot be emitted before a `header()` call and could result in an error or incorrect headers/responses. A common solution is to aggregate headers and content before emitting them to avoid those problems, but this often results in incompatible abstractions.

When reviewing the package, one will see the implementation of value objects[5] throughout the interfaces (such as `RequestInterface::getUri()`). Value objects are very useful to maintain immutability so as not to inadvertently alter the state of the messages between the requester and responder when constructing new request and response objects. A new request object can be instantiated from an existing one using a `with*()` method to modify a single value. For example:

A URI object was created with our destination and passed to a new `Request` object that can be used as a base request object

```
$uri = new Uri('http://foo.dev');
$baseRequest = new Request($uri,null, [
   //headers ...
]);

$newRequest = $baseRequest->
        withUri($uri->withPath('/list'))->withMethod('GET');
$anotherRequest = $baseRequest->
        withUri($uri->withPath('/users'))->withMethod('GET');
```

in the flow. A new request object is essentially copied from `$baseRequest` but using a new URI ('http:://food.dev/list'). Another request object is then composed using `$baseRequest` again, but this time with a new URI ('http://food.dev/users'). Any other data in `$baseRequest` is maintained throughout the new objects (like important headers).

When building a request or response object using `Psr\Http\Message\RequestInterface` or `Psr\Http\Message\ResponseInterface`, the `with*()` methods can safely return `$this` as it is usually more performant without a cloning operation. The interfaces only require that an instance is returned so that a cloning operation *can* be used, but it's important that the developer maintain immutability. It can be noted, however, that when responding with large data, a stream response may be required, which essentially cannot be immutable. PHP-FIG, therefore, recommends "… that implementations use read-only streams for server-side requests and client-side responses."

When reviewing the `psr/http-message` package, one will likely notice that, along with `RequestInterface` and `ResponseInterface`, there is also a `ServerRequestInterface`. While request and response have a one-to-one correlation with RFC 7320, a `ServerRequest` object can encapsulate other considerations:

- Access to server parameters derived from the request or from the `$_SERVER` superglobal.
- Access to query parameters, usually from the `$_GET` superglobal.
- Access to uploaded files, usually from the `$_FILES` superglobal.
- Access to other attributes derived from the request.
- Access to the parsed body (more on this later).

The additional access is very convenient when handling a request, and by identifying that the request object implements `ServerRequestInterface`, the developer knows they have access to all these values.

The Parsed Body of ServerRequestInterface

Typically, the body of a request is contained in the `$_POST` superglobal, but it could also be serialized data or non-form-encoded data like an array or object (JSON or XML). The `getParsedBody()` method will have an ambiguous return and could be *anything*, but it is dependent upon the domain of the application, and its format or structure would be expected. It could then be handled and processed appropriately by the application.

Many web applications transmit and submit data using forms and POST, but the trend is shifting towards being more API-centric, and they are using PUT and PATCH as request methods more often. While data can certainly be converted into arrays, it doesn't mean that it *should* be and depends on the systems or platforms communicating with each other. This avoids adding additional specifications and rules that only increase the developer's work to retrieve the data and allows the domains to decide what conventions they should be using.

Validating Header Data

This validation is an important rule to be aware of in classes implementing `MessageInterface`. PHP-FIG recommends the "… implementation SHOULD reject invalid values and SHOULD NOT make any attempt to automatically correct the provided values." At a minimum, header names and values should reject the following values:

- NUL (0x00)
- `\r` (0x0D)
- `\n` (0x0A)

Additionally, header names should also reject "Any character less than or equal to 0x20."

Header fields names should be case-insensitive, whether setting or reading them. For example:

```
$request = $request->withHeader('Foo', 'bar');

$request->getHeaderLine('Foo'); // returns 'bar'
$request->getHeaderLine('foo'); // returns 'bar'
$request->getHeaderLine('fOo'); // also returns 'bar'
```

Conclusion

I hope this has presented an understanding of what is going on under the hood of request and response objects in popular frameworks and libraries. Even if these specific interfaces are not implemented, many of the decisions behind them are in use (specifically in the `guzzlehttp/guzzle` package and Laravel's `Illuminate/Http/Client/Request`).

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

Related Reading

- *PSR Pickup: PSRs - Improving the Developer Experience* by Frank Wallen, March 2022. https://phpa.me/wallen-mar-2022
- *PSR Pickup: PSR 12 Extended Coding Style Standard* by Frank Wallen, April 2022. https://phpa.me/2022-04-psr
- *PSR Pickup: Psr-3 Logger Interface* by Frank Wallen, May 2022. https://phpa.me/2022-05-psr

New and Noteworthy

PHP Releases

PHP 8.2.0 Alpha 2 available for testing:

https://www.php.net/archive/2022.php#2022-06-23-1

PHP 8.1.7 Released!:

https://www.php.net/archive/2022.php#2022-06-09-2

PHP 7.4.30 Released!:

https://www.php.net/archive/2022.php#2022-05-12-1

News

Symfony 4.4.43 released

The latest release of Symfony has dropped with lots of fixes.

https://symfony.com/blog/symfony-4-4-43-released

PHPRoundtable Episode 85: The State of PHP User Groups

The PHPRoundtable brings together a panel of PHP User Groups Organizers and members from around the globe to discuss the state of their User Groups and how they are moving forward.

https://www.youtube.com/watch?v=I9aDTqwhG6g&t=107s

Zoom in on Symfony 5/6: the fast track workshop at SymfonyCon Disneyland Paris 2022

Discover the Symfony 5/6: the fast track 2-day workshop organized at SymfonyCon Disneyland Paris 2022 on November 15-16.

https://phpa.me/symfony-con-wkshop-paris-2022

Jetbrains: Addressing the New UI Comments

Last week, we announced a preview program for the new UI that we're working on for IntelliJ-based IDEs. We've received an overwhelming number of comments, and we can't reply to each one individually, so we'd like to address some of the most common concerns.

https://phpa.me/jetbrains-new-ui-comments

PHP Internals: [RFC] [Under Discussion] Auto-implement Stringable for string backed enums

I'd like to open a discussion on this RFC, to auto-implement Stringable for string-backed enums

https://externals.io/message/118040

PHPWatch: New `composer bump` Command in Composer 2.4

Composer version 2.4 adds a new command called `bump` that increases the requirements listed in the `composer.json` file with the currently installed version numbers. When the version numbers are bumped in the `composer.json` file, it effectively prevents Composer from installing a lower version of the required packages.

https://php.watch/articles/composer-bump

DEVDOJO: ****Laravel 9 WhereNotIn Database Query Examples****

This tutorial provides you with simple examples of where Not In Laravel Query Builder. And as well as how to use Laravel Eloquent WhereNotIn with arrays.

https://phpa.me/devdojo-wherenotin-examples

Symfony Station: ****Frontend Madness: SPAs, MPAs, PWAs, Decoupled, Hybrid, Monolithic, Libraries, Frameworks! WTF for your PHP backend?****

A look at how JavaScript chaos on the front end is impacting PHP CMSs and frameworks.

https://phpa.me/symfonystation-JS-PHP-backend

Structure by Use Case

Edward Barnard

This month we're introducing the Strategic Domain-Driven Design pattern that we'll be repeating over and over as we build out our project.

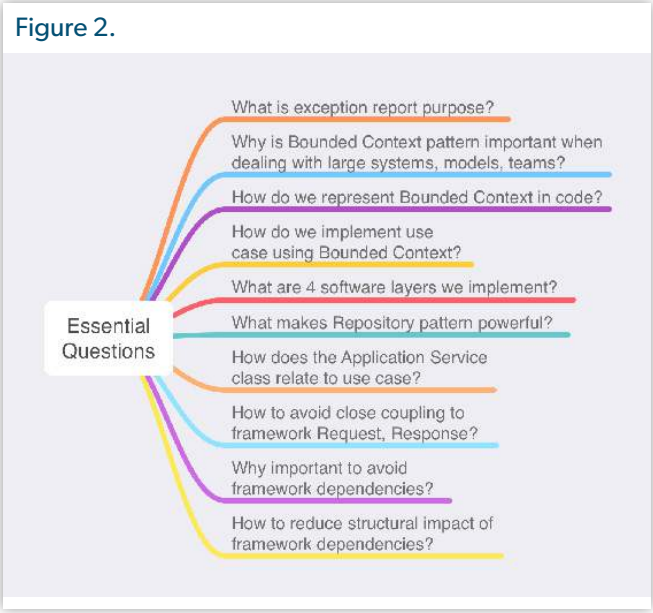
All source code is available on GitHub at ewbarnard/strategic-ddd[1].

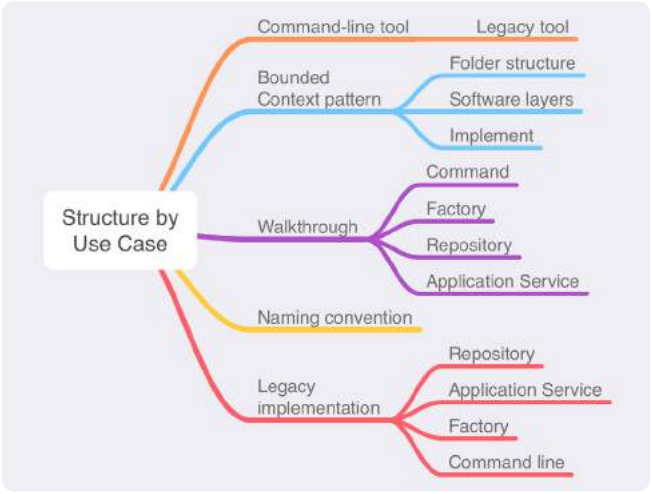
Essential Questions

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

- What is the purpose of the Exception Report feature?
- Why is the Bounded Context pattern important when dealing with large systems, models, or teams?
- How do we represent a Bounded Context in the codebase?
- How do we implement a use case using the Bounded Context pattern?
- What are the four software layers we implement with every Bounded Context, even if just an empty folder?
- What makes the Repository pattern so powerful?

Figure 2.





- How does the Application Service class relate to the use case being implemented?
- How do we avoid closely coupling to the PHP framework's server request and response objects?
- Why is it important to avoid framework dependencies wherever possible?
- How do we reduce the structural impact of framework dependencies?

Command-line Tool

We're about to develop a feature that writes to the `exception_reports` table. We want to exercise that feature as we develop it. I personally prefer to exercise via the Unix/Linux command line, so that's what we'll do. You might prefer to invoke our use case by some other method. By all means, do so! The nature of the wrapper, as you'll soon see, doesn't matter. It's the functionality within that wrapper that matters.

Generate the command-line tool skeleton with CakePHP's `bake` command:

```
bin/cake bake command --no-test CountEvents./annotate.sh
```

The `annotate.sh` script below invokes the CakePHP IDE Helper to update files as needed.

```
#!/bin/bash -xv

bin/cake illuminator illuminate -v
bin/cake annotate all -v
bin/cake code_completion generate
bin/cake phpstorm generate
```

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

Listing 1.

```php
<?php

declare(strict_types=1);

namespace App\Command;

use Cake\Command\Command;
use Cake\Console\Arguments;
use Cake\Console\ConsoleIo;
use Cake\Console\ConsoleOptionParser;

class CountEventsCommand extends Command
{
    public function buildOptionParser(
        ConsoleOptionParser $parser
    ): ConsoleOptionParser {
        $parser = parent::buildOptionParser($parser);

        return $parser;
    }

    public function execute(Arguments $args, ConsoleIo $io)
    { }
}
```

Listing 1 shows the generated command. We'll be stripping most of it out to focus directly on our use case.

Legacy Command-line Tool

Here's how I am building out our feature in the legacy codebase. We'll be using the command-line tool as a test harness. I, therefore, created a minimal PHP script within the `test/` directory. See Listing 2.

We are essentially duplicating the unit test (Listing 3) we created as part of the legacy setup. Our purpose is the same. We need to ensure the setup is correct and operational.

Below see the result of running the command-line tool. My current development environment runs PHP 8.0 as the default PHP interpreter. However, the legacy codebase requires PHP 7.3. I, therefore, specified the full path to PHP 7.3 when running our new command-line tool. Our test successfully passed the string `'Road Trip'` through class `ContextRoot`.

Listing 3.

```php
<?php

declare(strict_types=1);

namespace Test\LegacyBoundedContexts;

use LegacyBoundedContexts\ContextRoot;
use PHPUnit\Framework\TestCase;

class ContextRootTest extends TestCase
{
    public function testEchoBack(): void
    {
        $target = new ContextRoot();
        $expected = 'test input string';
        $actual = $target->echoBack($expected);

        static::assertSame($expected, $actual);
    }
}
```

Listing 2.

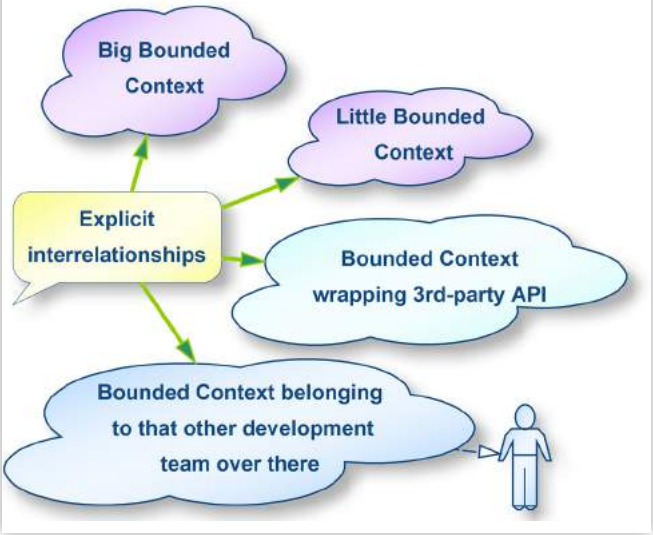
```php
<?php

use LegacyBoundedContexts\ContextRoot;

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

echo 'Road Trip' . PHP_EOL;
$expected = 'Road Trip';
$target = new ContextRoot();
$actual = $target->echoBack($expected);
echo "Expected $expected, actual $actual" . PHP_EOL;
```

Figure 3.




```
/Applications/MAMP/bin/php/php7.3.29/bin/php
                              test/exercise_count_events.php
Road Trip
Expected Road Trip, actual Road Trip
```

Bounded Context Pattern

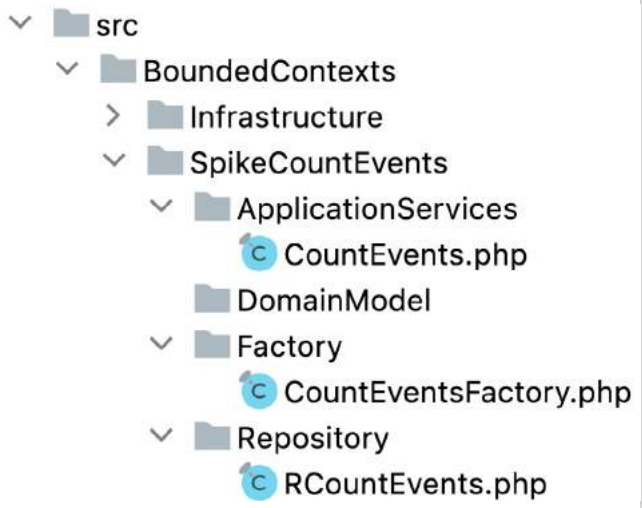
The *Bounded Context* is an important concept fundamental to Domain-Driven Design. Martin Fowler, writing about Eric Evans' 2003 book *Domain-Driven Design: Tackling Complexity in the Heart of Software*, explains[2]:

> *A particularly important part of DDD is the notion of Strategic Design–how to organize large domains into a network of Bounded Contexts. Until that point, I'd not seen anyone tackle this issue in any compelling way.*

See Figure 3. Fowler begins his Bounded Context article (linked in the quote above):

2 *explains:* *https://martinfowler.com/bliki/DomainDrivenDesign.html*

Figure 4.



> *Bounded Context is a central pattern in Domain-Driven Design. It is the focus of DDD's strategic design section which is all about dealing with large models and teams. DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.*

The idea of the Bounded Context is so fundamental, so important, that it literally prevented me from getting any code written as I was learning about Domain-Driven Design. Is the Bounded Context big or small, fine-grained or coarse-grained? Should separate Bounded Contexts be separate repositories? Separate plugins?

I ultimately decided to simply place a new folder in the source tree called `BoundedContexts`. Figure 4 shows that folder containing two folders: `Infrastructure` and `SpikeCountEvents`.

Figure 6.

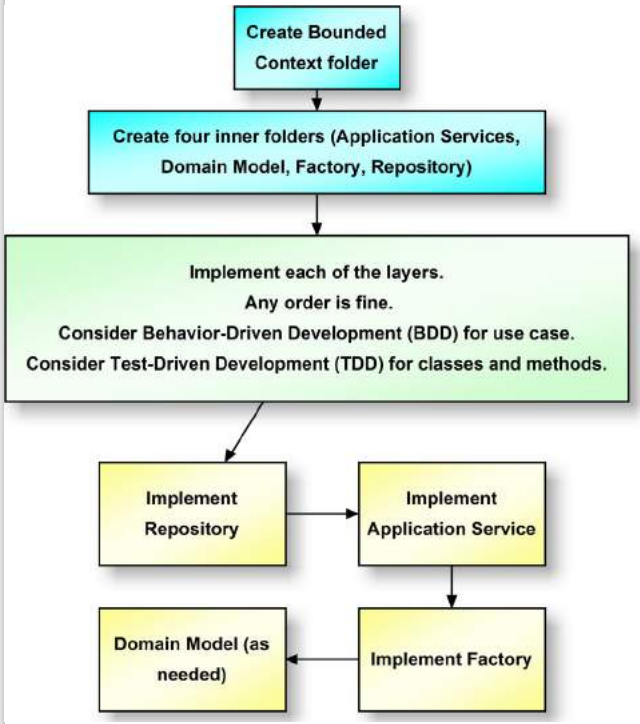
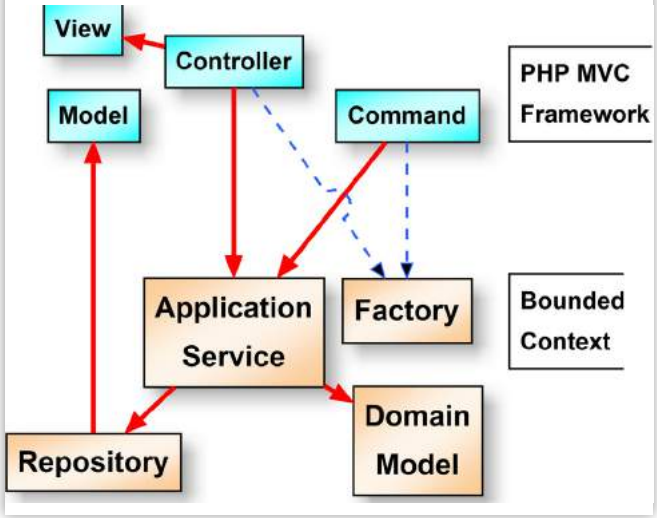


Figure 5.



We'll be writing our features and use cases in such a way that the Bounded Contexts can themselves evolve as we gain deeper insight. We'll gain a better understanding of where the lines should be drawn based on conversations with our stakeholders. For now, the definition of a Bounded Context is simple: it's whatever is in a top-level folder within `src/BoundedContexts`.

I'm calling our "count events" feature a spike because we don't intend to retain the event-counting code. It merely exists to exercise the production code we're developing for exception reports.

Figure Figure 4 shows the folder structure we'll be repeating every time we create a new Bounded Context. Every Bounded Context includes the top-level folders:

- `ApplicationServices`
- `DomainModel`
- `Factory`
- `Repository`

I create those four folders every time to show the pattern, even though some of those folders might remain empty. Given that "service" is such an overused word within our world of software development, what does "Application Service" mean here?

Patterns, Principles, and Practices of Domain-Driven Design[3] by Scott Millett with Nick Tune [Millet] introduces the term (page 108):

> *The application service layer represents the use cases and behavior of the application. Use cases are implemented as application services that contain application logic to coordinate the fulfillment of a use case by delegating to the domain and infrastructural layers.*

Figure 5 shows the Bounded Context and its relation to whatever PHP framework we're using. Figure 6 shows the steps we'll be following over and over again as we implement

3 *Patterns, Principles, and Practices of Domain-Driven Design:* https://phpa.me/amazon-ppp-of-ddd

Listing 4.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Command;
6.
7. use App\BoundedContexts\SpikeCountEvents\Factory\CountEventsFactory;
8. use Cake\Command\Command;
9. use Cake\Console\Arguments;
10. use Cake\Console\ConsoleIo;
11.
12. final class CountEventsCommand extends Command
13. {
14.     public function execute(Arguments $args, ConsoleIo $io): ?int
15.     {
16.         $service = CountEventsFactory::countEvents();
17.         $service->insertCurrentCount();
18.         $io->out('Count complete');
19.
20.         return 0;
21.     }
22. }
```

Listing 5.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\BoundedContexts\SpikeCountEvents\Factory;
6.
7. use App\BoundedContexts\SpikeCountEvents
                            \ApplicationServices\CountEvents;
8. use App\BoundedContexts\SpikeCountEvents\Repository
                            \RCountEvents;
9. use JetBrains\PhpStorm\Pure;
10.
11. final class CountEventsFactory
12. {
13.     private function __construct()
14.     {
15.     }
16.
17.     #[Pure]
18.     public static function countEvents(): CountEvents
19.     {
20.         $repository = new RCountEvents();
21.         return new CountEvents($repository);
22.     }
23. }
```

use case after use case. You are completely free add, subtract, or adjust to fit the situation of the moment.

Command Line

Let's walk through an actual implementation to see how everything fits together. Listing 4 shows the completed command-line handler.

Line 16 uses a factory[4] to produce the application service that will handle our use case. Line 17 invokes the use-case processing. Here we are invoking the use case from a CakePHP command-line wrapper; we could have called the factory from anywhere and invoked the use case from any point in our PHP source code.

That's an important point. The application service (handed to us by the factory) is the entry point to our software handling that use case (Figure 7). As we're about to see, this is an exceedingly small use case. It could just as well have been an exceedingly large use case. The pattern remains the same.

Factory

Listing 5 shows the factory itself is only two lines long. We don't yet know what a repository is, but the factory creates one here and passes it into the Application Service constructor.

Repository

Listing 6 shows our tiny repository. It's small but represents a set of useful concepts.

Before looking closely inside the repository, I should explain a naming convention. You may prefer a different way of naming things, and that's fine. There's almost certainly a better convention than what I'm showing here.

The problem is this. Early on in a project, I'm implementing many use cases one by one. Each use case will normally get its own set of files. The `ApplicationServices` folder for this Bounded Context may well contain two dozen use cases represented by two dozen classes. With more than a few use cases, I will likely have grouped them into subfolders, but either way, we'll have a lot of use cases. That's why we are here.

In addition to the use case (the Application Services class), I may have a similarly named command-line invocation, a webpage controller, a webpage view, a table model with a similar name, the repository, the factory, class constants as a separate file, and perhaps an interface class supporting the unit tests. I need to keep the names similar to make it clear they all fit together and that they do not belong to some other use case.

I, therefore, adopted the naming convention:

- Use case "CountEvents" is the bare class name `CountEvents`.
- The factory has "Factory" as a suffix, `CountEventsFactory`.
- The repository has "R" as a prefix, `RCountEvents`.
- The class constants (such as magic numbers or strings) file has a "C" prefix, `interface CCountEvents`. It contains only `public const` declarations.
- The interface file has an "I" prefix, `interface ICountEvents`. It contains only `public function` declarations.
- Framework-specific classes such as `CountEventsCommand` follow the framework's naming conventions.

4 *factory:* *https://en.wikipedia.org/wiki/Factory_(object-oriented_programming*

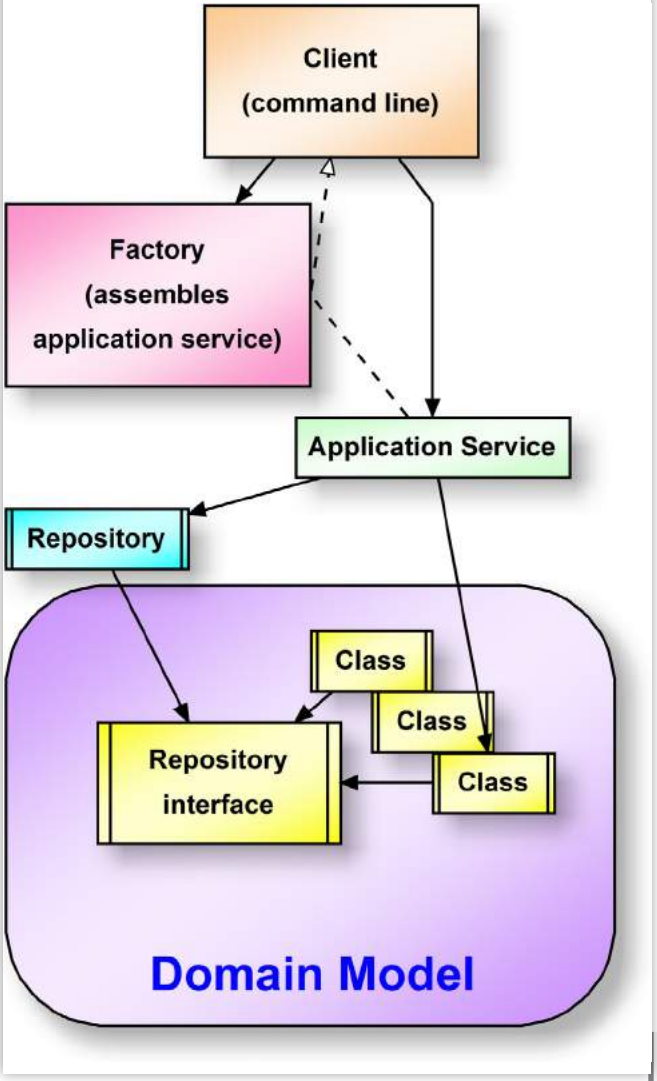
The repository, Listing 6, is the intermediate layer between our feature (use case) processing and the framework's database access layer. Frameworks generally assume you'll put database-related code inside the framework's table models. We're not doing that. By keeping our feature code outside of the models, we can freely regenerate those models as we evolve the database design.

The repository method `loadModels()` instantiates the two framework models we'll be using. Method `collectCount()` counts the number of rows in table `local_app_events`. The count, of course, could be zero, indicating an empty table.

Method `storeCount()` uses the framework model to insert that count as a new row in table `event_counts`.

Separating `collectCount()` and `storeCount()` might seem a bit weird at this point. We could put them both into the same method and save a function call. What we see here is a careful separation of concerns.

Figure 7.



Listing 6.

```php
<?php

declare(strict_types=1);

namespace App\BoundedContexts\SpikeCountEvents\Repository;

use App\BoundedContexts\Infrastructure\LoadTableModels\
                PrimaryDatabase\Events\AppEventsPrimaryTrait;
use App\BoundedContexts\Infrastructure\LoadTableModels\
                PrimaryDatabase\Events\EventCountsTrait;
use App\Model\Entity\EventCount;
use Cake\I18n\FrozenTime;

final class RCountEvents
{
    use AppEventsPrimaryTrait;
    use EventCountsTrait;

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

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

    public function collectCount(): int
    {
        return $this->localAppEventsTable->find()->count();
    }

    public function storeCount(int $count): void
    {
        $data = [
            EventCount::FIELD_WHEN_COUNTED =>
                        FrozenTime::now(),
            EventCount::FIELD_EVENT_COUNT => $count,
        ];
        $entity = $this->eventCountsTable->newEntity($data);
        $this->eventCountsTable->saveOrFail($entity);
    }
}
```

It's the application service that orchestrates the workflow. The repository is only responsible for database-specific aspects. The repository class needs to know how CakePHP works; the application service does not. Database-specific code (i.e., the repository) can be difficult to unit test. By extracting the database-specific code into a separate class, the application service becomes far easier to unit test.

[Millett] explains the repository pattern's power (p. 481):

> *Many developers have negatively blogged that a repository is an antipattern because it hides the capabilities of*

Listing 7.

```php
<?php

declare(strict_types=1);

namespace App\BoundedContexts\SpikeCountEvents
                                    \ApplicationServices;

use App\BoundedContexts\SpikeCountEvents\Repository
                                    \RCountEvents;

final class CountEvents
{
    private RCountEvents $repository;

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

    public function insertCurrentCount(): void
    {
        $count = $this->repository->collectCount();
        $this->repository->storeCount($count);
    }
}
```

the underlying persistence framework. This is actually the point of the repository.

Instead of offering an open interface into the data model that supports any query or modification, the repository makes retrieval explicit by using named query methods and limiting access to the aggregate level. By making retrieval explicit, it becomes easy to tune queries, and more importantly express the intent of the query in terms a domain expert can understand rather than in SQL.

Application Service

Listing 7 shows our application service. It should not, at this point, contain any surprises. The factory passed the repository into the constructor, and our single handler `insertCurrentCount()` makes the two repository calls as expected.

Legacy Count Events

With the listings right in front of us, I find it easiest to work from the inside out. We'll build things in this order:

- Repository
- Application Service
- Factory
- Test harness

Listing 8.

```php
<?php

declare(strict_types=1);

namespace LegacyBoundedContexts\SpikeCountEvents\
                                Repository;

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

use function is_array;

use const PHP_EOL;

/**
 * Non-production code, intentionally exposes SQL
 */
final class RCountEvents extends BaseModel
{
    public function collectCount(): int
    {
        $sql = 'select count(*) row_count
                from local_app_events
                limit 1';
        try {
            $statement = $this->sql->executeQuery($sql,[]);
            $rows = $statement->fetchAll();
        } catch (DBALException $e) {
            return 0;
        }
        if (is_array($rows) && (1 === count($rows))) {
            return $rows[0]['row_count'];
        }
        return 0;
    }

    public function storeCount(int $count): void
    {
        $sql = 'insert into event_counts
    (when_counted, event_count, created, modified)
VALUES (now(), ?, now(), now())';
        $parms = [$count];
        try {
            $this->sql->executeUpdate($sql, $parms);
        } catch (DBALException $e) {
            echo 'Query failed: ' .
                $e->getMessage() . PHP_EOL .
                $sql . PHP_EOL;
        }
    }
}
```

Listing 8 shows the legacy repository. Our legacy codebase uses the Symfony/Doctrine Database Abstraction Layer[5] (DBAL). I wrote the queries as raw SQL because, for now, this is likely the most portable. Most frameworks support low-level SQL queries like this.

5 *Symfony/Doctrine Database Abstraction Layer:* https://symfony.com/doc/current/doctrine/dbal.html

Listing 9.

```
<?php

declare(strict_types=1);

namespace LegacyBoundedContexts\SpikeCountEvents
                                    \ApplicationServices;

use LegacyBoundedContexts\SpikeCountEvents\Repository\
                                    RCountEvents;

final class CountEvents
{
    /** @var RCountEvents */
    private $repository;

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

    public function insertCurrentCount(): void
    {
        $count = $this->repository->collectCount();
        $this->repository->storeCount($count);
    }
}
```

To be sure, portability is not a major consideration at this point. If we were moving features or use cases between frameworks, we would expect each repository class to need rework to be compatible with the target framework.

Listing 10.

```
<?php

declare(strict_types=1);

namespace LegacyBoundedContexts\SpikeCountEvents\Factory;

use LegacyBoundedContexts\SpikeCountEvents
                        \ApplicationServices\CountEvents;
use LegacyBoundedContexts\SpikeCountEvents\Repository
                        \RCountEvents;

final class CountEventsFactory
{
    private function __construct()
    {
    }

    public static function countEvents(): CountEvents
    {
        $repository = new RCountEvents();
        return new CountEvents($repository);
    }
}
```

Listing 9 shows the application service. It's essentially identical to Listing 7 except for being compliant with PHP 7.3. The private `$repository` property isn't allowed to have a type hint.

Every Application Service should, in theory, be free of any framework dependencies; in actual practice, that might not be the case. I have found, for example, that it's useful to have access to the HTTP/HTTPS request object. If your framework implements the PSR-7 request and response interfaces[6], your application service can indeed remain free of the framework request/response objects.

If there is some specific framework feature that your Application Service needs, consider creating a repository method that accesses (or wraps) that feature. Doing so keeps your Application Service free of entanglements and easier to test.

The Count Events Factory (Listing 10) is likewise identical to Listing 5 except for code annotations.

The command-line test harness is shown in Listing 11. It works the same as the command-line example in Listing 4.

Listing 11.

```
<?php

use LegacyBoundedContexts\SpikeCountEvents
                    \Factory\CountEventsFactory;

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

$service = CountEventsFactory::countEvents();
$service->insertCurrentCount();
echo 'Count complete' . PHP_EOL;
```

The result below is not terribly informative, but all we need to know is that the script has finished. The database `event_counts` table now contains one row (not shown).

```
/Applications/MAMP/bin/php/php7.3.29/bin/php
                                test/exercise_count_events.php
Count complete
```

Essential Questions Answered

- *What is the purpose of the Exception Report feature?* Provide visibility to "random and rare" failures.
- *Why is the Bounded Context pattern important when dealing with large systems, models, or teams?* As Martin Fowler explains, Strategic Domain-Driven Design organizes large domains into a network of Bounded Contexts. DDD deals with large models and teams by dividing them into different Bounded Contexts and being explicit about their interrelationships.

6 *PSR-7 request and response interfaces:* https://www.php-fig.org/psr/psr-7/

Figure 8.

- *How do we represent a Bounded Context in the codebase?* We create a folder within the top-level Bounded Context folder.
- *How do we implement a use case using the Bounded Context pattern?* Create an Application Service class specific to that use case. The Application Service orchestrates the use-case processing, delegating to other software layers as appropriate.
- *What are the four software layers we implement with every Bounded Context, even if just an empty folder?* Application Service, Repository, Domain Model, Factory.
- *What makes the Repository pattern so powerful?* It is laser-focused and specific, never generalized. It expresses purpose in terms a domain expert can understand rather than SQL.
- *How does the Application Service class relate to the use case being implemented?* It is the handler, the point of entry. The actual use-case implementation hides behind or within the Application Service.
- *How do we avoid closely coupling to the PHP framework's server request and response objects?* Use the PSR-7 interface declarations if the framework implements those interfaces. The PSR-7 interfaces are designed for precisely this purpose.
- *Why is it important to avoid framework dependencies wherever possible?* Code becomes less portable and more difficult to test when closely tied to a PHP framework. This same code can become tied to obsolete versions of the framework, • no longer easy to upgrade due to incompatibility with newer versions.
- *How do we reduce the structural impact of framework dependencies?* Use the Repository pattern to isolate the framework dependencies. Since the repository should be small and relatively free of business logic, it will be relatively easy to replace the repository with a different repository tied to a different framework or framework version.

Summary

We introduced our PHP implementation of the Bounded Context pattern fundamental to Strategic Domain-Driven Design. We used a simple folder structure to represent the different software layers within the Bounded Context:

- Application Services
- Repository
- Factory
- Domain Model

We'll be repeating this structure over and over as we implement our features and use cases.

Each Application Service, generally speaking, represents a single use case. The repository is carefully limited to providing access to databases (or other external services) based on what the Application Service and Domain Model actually need.

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

Related Reading

- *DDD Alley: Random and Rare Failures* by Edward Barnard, June 2022. https://phpa.me/2022-06-ddd
- *DDD Alley: Get Organized and Get Started* by Edward Barnard, May 2022. https://phpa.me/2022-05-ddd
- *DDD Alley: When the New Requirement Arrives* by Edward Barnard, April 2022. https://phpa.me/2022-04-ddd
- *DDD Alley: Better Late Than Never_* by Edward Barnard, March 2022. https://phpa.me/2022-03-ddd

Create a Custom Module in Drupal 9

Nicola Pignatelli

This series of articles will teach you how to create a custom module and use its main features. It assumes you installed Drupal using composer, as I explained in the previous article (PHP Architect April 2022).

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.

Sites are down? Use Health sites to verify

This module allows us to verify the health of a hostname/URL list.

In this series of articles, you will learn how to:

- write a basic custom module
- import configuration files
- create admin pages
- create administration items menu
- create local items task
- use Controller to render the page
- create Forms

I will need a content type called "History Url." I will use YAML configuration files previously exported to create a new content type. I will import the configuration using drush.

Configuration files will be in the folder: `[rdp]/config/[site name]/sync/historical_url/historical_url_content_type`. See Figure 1.

Download the config_split module and enable it. Then enter the drush command to import the configuration.

```
cd [rdp]
composer require drupal/config_split -n
cd [rdp]/web
../vendor/drush/drush/drush en config_split -y
mkdir -p \
   sites/default/config/split/historical_url_content_type
../vendor/drush/drush/drush \
   config-split-import historical_url_content_type
```

Now you see content type into admin/structure/types.

Navigate through file system Structures

Figure 2 shows the structure of the module.

Figure 1.

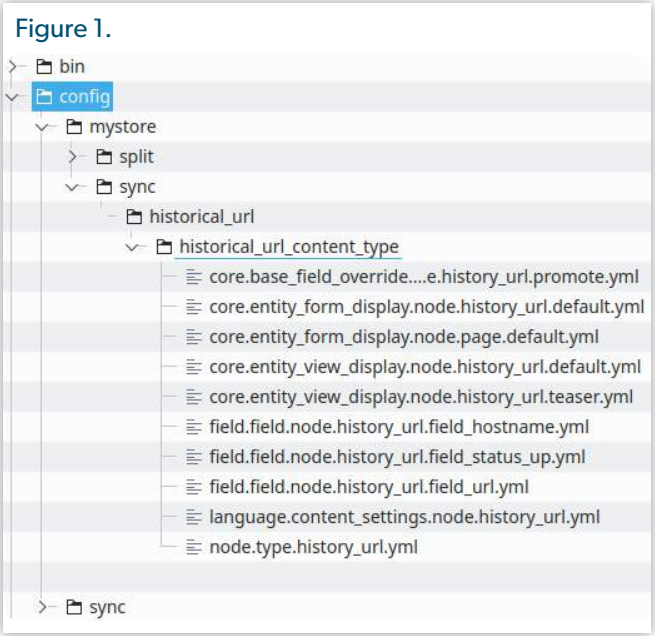
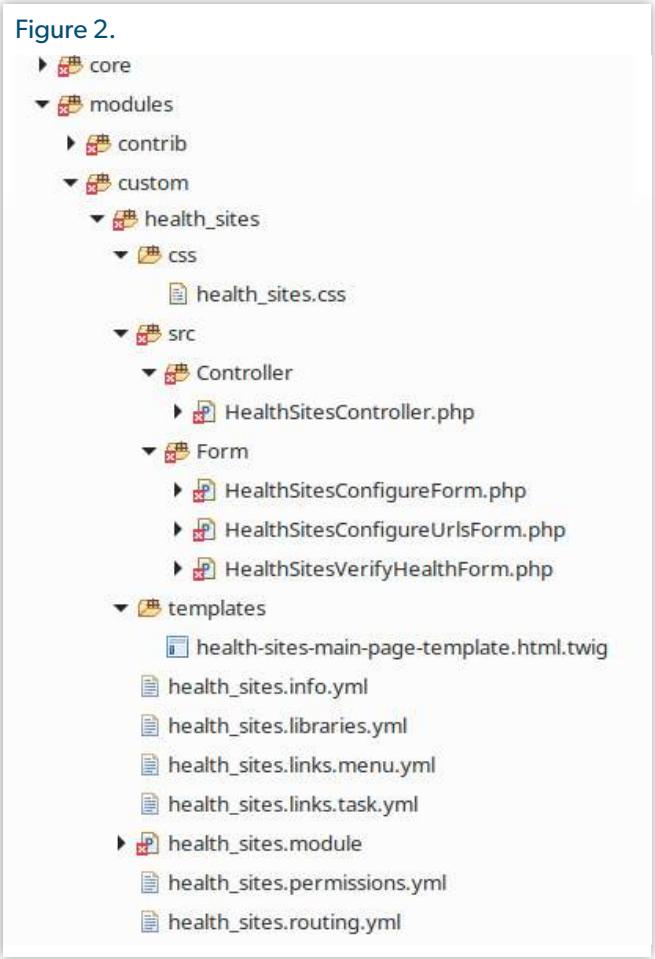


Figure 2.



health_sites.info.yml

This is the first file that you create for a custom module.

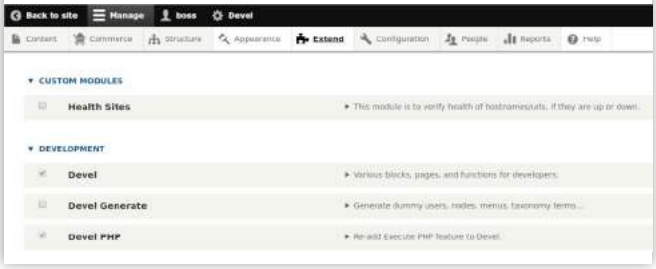
```
name: Health Sites
description: This module is to verify the health of
    hostnames/URLs if they are up or down.
type: module
core_version_requirement: 9.x
package: Custom modules
```

"name" and "description" parameters are visible on the Extend webpage (Figure 3) and allow you to select the module. The "type" parameter identifies typology; in this example, `module`. "core_version_requirement" verifies compatibility between 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 3).

You can enable the module using the web interface with the flag (Figure 3) or with drush with the following command:

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

Figure 3.



A walk in the routing system

The Drupal 8 Routing system is very different from previous Drupal versions. In Drupal 7, one specific hook, "hook_menu," managed routes and specific hook_permission to create new permissions to assign. Now the routing system is divided between many files. Our example includes:

- health_sites.routing.yml
- health_sites.permissions.yml
- health_sites.links.menu.yml
- health_sites.links.task.yml

health_sites.routing.yml

Listing 1, `health_sites.routing.yml`, contains the webpage links.

The route name is the key used in Drupal code, so if the URL changes, you don't need to make changes to the project but only in this file.

`path` indicates the real URL of the webpage. `_controller` indicates the class Controller function to render HTML page or return JSON string. `_title` is the HTML title of the page. `_form` identifies the class that manages a web form on the page. `_permission` indicates whether the page has public access or is restricted by the user's possession of the relevant permission.

From line 1 to line 7, I define the root page for the "Health Sites" section. It contains subpages links to that section. From line 9 to line 15, I define the administration page where you insert hostnames to verify. Lines 17 thru 23 define the administration page where you insert hostname URLs to verify. From line 25 to line 31, I define the administration page where you verify the health of hostname URLs.

Listing 1.

```
1. health_sites.main:
2.   path: '/admin/health_sites'
3.   defaults:
4.     _controller: '\Drupal\health_sites\Controller\HealthSitesController::index'
5.     _title: 'Health Sites'
6.   module_name: 'health_sites'
7.   requirements:
8.     _permission: 'health sites configure'
9.
10. health_sites.configure:
11.   path: '/admin/health_sites/configure'
12.   defaults:
13.     _form: '\Drupal\health_sites\Form\HealthSitesConfigureForm'
14.     _title: 'Health Sites configure'
15.   requirements:
16.     _permission: 'health sites configure'
17.
18. health_sites.configure_urls:
19.   path: '/admin/health_sites/configure-urls'
20.   defaults:
21.     _form: '\Drupal\health_sites\Form\HealthSitesConfigureUrlsForm'
22.     _title: 'Health Sites - Urls configure'
23.   requirements:
24.     _permission: 'health sites configure'
25.
26. health_sites.verify:
27.   path: '/admin/health_sites/verify-health'
28.   defaults:
29.     _form: '\Drupal\health_sites\Form\HealthSitesVerifyHealthForm'
30.     _title: 'Health Sites - Verify Urls'
31.   requirements:
32.     _permission: 'health sites verify urls'
```

Figure 4.

health_sites.permissions.yml

hook_permission has been replaced by `health_sites.permissions.yml`. See Listing 2.

"health sites configure" is permission to insert the hostnames and URLs to verify. "health sites verify URLs" is permission to insert the status of hostnames and URLs.

"title" and "description" keys are visible on the permissions page (/admin/people/permissions). "restrict access" key permits to display of a warning on the page about giving this permission to someone.

Permissions are displayed in Figure 4.

Listing 2.

```
1. health sites configure:
2.   title: 'Configure health sites'
3.   description: 'This permission is to configure hostnames
       and urls to verify'
4.   restrict access: true
5.
6. health sites verify urls:
7.   title: 'Verify health of hostnames/urls'
8.   description: 'This permission is to see if status urls a
       re UP or DOWN'
9.   restrict access: true
```

Listing 3.

```
1. health_sites.main:
2.   title: Healt Sites
3.   description: 'Health Sites'
4.   parent: 'system.admin'
5.   route_name: health_sites.main
6.
7. health_sites.configure_sites:
8.   title: Configure Healt Sites list
9.   description: 'Configure Health Sites list'
10.  parent: 'health_sites.main'
11.  route_name: health_sites.configure
12.
13. health_sites.configure_urls:
14.  title: Health Sites - Urls configure
15.  description: 'Health Sites - Urls configure'
16.  parent: 'health_sites.main'
17.  route_name: health_sites.configure_urls
18.
19. health_sites.verify_sites:
20.   title: Verify Healt Sites
21.   description: 'Verify Healt Sites'
22.   parent: 'health_sites.main'
23.   route_name: health_sites.verify
```

health_sites.links.menu.yml

Listing 3, `health_sites.links.menu.yml`, contains an items menu that matches the old Drupal 7 MENU_NORMAL_ITEM type.

health_sites.main is the key that renders the main page, as shown in Figure 5.

`health_sites.configure_sites` is the key to configuring the hostnames page (Figure 6). Insert the hostnames to verify, one per line.

health_sites.configure_urls is key to go to configure hostnames URLs page (Figure 7). Insert/update/delete URLs, one per line, after selecting the hostname from the select field.

health_sites.verify_sites is key to go to verify the status of hostname URLs (Figure 8). Select a hostname URL where you want to verify the health status.

The "parent" key identifies the optional parent link in the menu. The "route name" key identifies the route associated with the menu item.

Figure 5.

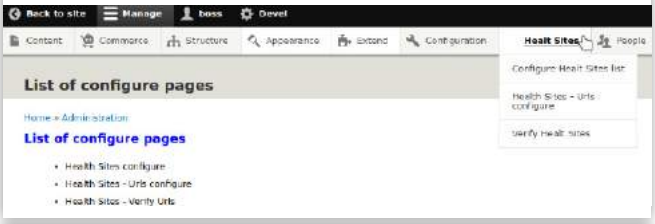
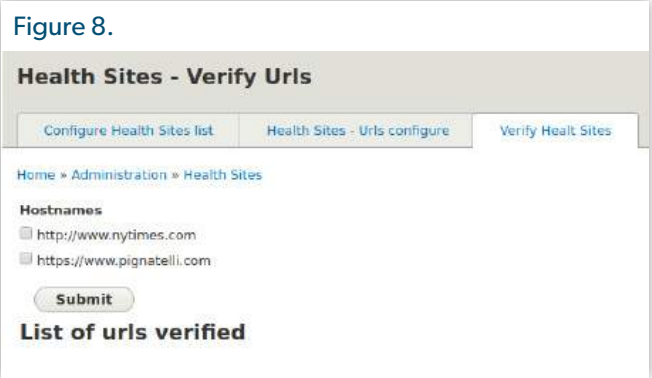


Figure 6.

Figure 7.

From lines 1 to 5, I define the main item menu for the "health sites" section. From lines 7 to line 11, I define the item menu to insert hostnames to verify. Lines 13 thru 17 define the item menu to insert hostname URLs to verify. From lines 19 to 23, I define an item menu to verify the status of hostname URLs.

Figure 8.



health_sites.links.task.yml

This file contains an item menu that matches the old Drupal 7 MENU_DEFAULT_LOCAL_TASK and MENU_LOCAL_TASK types.

health_sites.links.task.yml file is in Listing 4.

health_sites.configure_sites refers to the main tab where we insert hostnames to verify. health_sites.configure_urls refers to the tab where we insert hostnames URLs to verify. health_sites.verify_sites refers to the tab where we test the status of hostnames URLs to verify.

"route_name" is the key associated with a route. "base_name" is the key associated with the default tab.

The output of these tabs is shown in Figure 8.

Add JavaScript and stylesheet files

To import default or custom javascript and stylesheet libraries, you must use the `health_sites.libraries.yml` file.

The `drupal_add_js()` and `drupal_add_css()` functions for manual inclusions of JS/CSS files have been removed in favor of library definitions that properly declare their dependencies. As a result, all modules and themes have to declare their libraries to include JS and CSS files in [name module/name theme].libraries.yml.

Code in health_sites.libraries.yml is shown below. The last line contains the custom CSS of the module. Between braces should be inserted conditions of use of this file, for example, weight, browser type, and others. For a complete list of commands, go to Adding CSS and js to a module[1].

1 *Adding CSS and js to a module:* https://www.drupal.org/node/2274843

Listing 4.

```
1. health_sites.configure_sites:
2.   title: Configure Health Sites list
3.   route_name: health_sites.configure
4.   base_route: health_sites.configure
5.
6. health_sites.configure_urls:
7.   title: Health Sites - Urls configure
8.   route_name: health_sites.configure_urls
9.   base_route: health_sites.configure
10.
11. health_sites.verify_sites:
12.    title: Verify Healt Sites
13.    description: 'Verify Healt Sites'
14.    route_name: health_sites.verify
15.    base_route: health_sites.configure
```

```
health_sites:
  version: 1.x

  css:
    layout:
      css/health_sites.css: {}
```

health_sites.module

Listing 5 contains hooks and any callback functions.

health_sites_help

hook_help provides online user help. The page-specific help information provided by this hook appears in the Help block (provided by the core Help module) if the block is displayed on that page.

health_sites_cron

`hook_cron` performs periodic actions. Modules that require some commands to be executed periodically can implement `hook_cron()`.

Listing 5

```
...
7. function health_sites_help($route_name,
                RouteMatchInterface $route_match) {
8.   switch ($route_name) {
9.     case 'help.page.health_sites':
10.      $output = '';
11.      $output .= '<h3>' . t('About') . '</h3>';
12.      $output .= '<p>' . t('This is health_sites module.') .
                  '</p>';
13.      $output .= '<p>' . t(
                  'This custom module permits to verify
                  health of hostname/urls list.') . '</p>';
14.
15.      return $output;
16.
17.    default:
18.  }
19. }
```

Listing 5 continued

```
...
24. function health_sites_cron() {
25.
26.   $config = \Drupal::config('health_sites.settings');
27.
28.   $hostnames = $config->get('hostnames_to_verify');
29.   $urls_to_verify = $config->get('urls_to_verify');
30.
31.   $urls = [];
32.   foreach ($hostnames as $key => $hostname) {
33.
34.     if (!isset($urls_to_verify[$key])) {
35.       $urls[] = $hostname;
36.     }
37.     else {
38.       foreach ($urls_to_verify[$key] as $url) {
39.         $hostname = strpos($hostname, 'http')===0 ?
                $hostname : 'http://'.$hostname;
40.         $urls[$hostname][] = $hostname . $url;
41.       }
42.     }
43.   }
44.
```

From lines 28 to 31, I recover saved values of hostnames and URLs that need to be verified. From lines 33 to 45, I add a hostname to associated URLs. Next, URLs are verified, and History URL Contents are created or updated to store the status of URLs. If the content is updated, a new revision is created.

health_sites_theme

`hook_theme` is used to pass variables into a TWIG template. 'health_sites_main_page' key is used in the module to use the TWIG template defined in the 'template' key. 'variables' key contains variables switch to TWIG template.

Since we don't define the 'path' key, the template will be in the 'templates' module subdirectory.

Use Controllers to render pages

Let's create a HealthSitesController controller to render a webpage with a list of module sub-sections.

We'll link the Controller to the URL page through the routing.yml file. Back in Listing 1 on line 2 I defined the url and on line 4 I attached the Controller to configuration page.

HealthSitesController.php

`HealthSitesController` code is included in src/Controller/HealthSitesController.php in Listing 6.

Line 3 defines the namespace for the Controller, and line 12 defines the Controller class. I define the public member function to render the page on line 20. From line 21 to line 23, I restore route URLs from the routing file. From line 25 to line 30, I create an array of URLs to render. I return the array structure to render into a TWIG template on lines 32 thru 36.

```
45. //  \Drupal::logger('my_moduleXXX')->alert(print_r($urls,1));
46.   foreach ($urls as $hostname => $list_urls) {
47.     foreach ($list_urls as $url) {
48.       $ch = curl_init($url);
49.       curl_setopt($ch, CURLOPT_HEADER, true);
50.       curl_setopt($ch, CURLOPT_NOBODY, true);
51.       curl_setopt($ch, CURLOPT_RETURNTRANSFER,1);
52.       curl_setopt($ch, CURLOPT_FOLLOWLOCATION, true);
53.       curl_setopt($ch, CURLOPT_TIMEOUT,10);
54.       curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
55.       curl_setopt($ch, CURLOPT_USERAGENT,
                'Mozilla/5.0 (Windows NT 6.2; WOW64; rv:17.0)
                Gecko/20100101 Firefox/17.0');
56.       curl_setopt($ch, CURLOPT_REFERER,
                'https://www.pignatelli.com/');
57.       curl_exec($ch);
58.       $httpcode = curl_getinfo($ch, CURLINFO_HTTP_CODE);
59.       curl_close($ch);
60.       $query = \Drupal::entityQuery('node')
61.       ->condition('status', 1)
62.       ->condition('type', 'history_url')
63.       ->condition('field_hostname', $hostname)
64.       ->condition('field_url',
                str_replace($hostname, "", $url));
65.       $nid = $query->execute();
66.       if (empty($nid)) {
67.         $node = \Drupal::entityTypeManager()->
                getStorage('node')->create([
68.           'type' => 'history_url',
69.           'title' => 'Status of ' . $url,
70.           'field_hostname' => $hostname,
71.           'field_url' => str_replace($hostname, "", $url),
72.           'langcode' => 'en',
73.           'status' => 1,
74.           'revision' => 1
75.         ]);
76.       }
77.       else {
78.         $nid = array_pop($nid);
79.         $node = Node::load($nid);
80.         $node->setNewRevision();
81.         $node->revision_log = 'Created revision for node' .
                $nid . ' programmatically';
82.         $node->setRevisionCreationTime(REQUEST_TIME);
83.         $node->setRevisionUserId(1);
84.       }
85.
86.       if ($httpcode!=200) {
87.         $node->set('field_status_up', 0);
88.       } else {
89.         $node->set('field_status_up', 1);
90.       }
91.
92.       $node->save();
93.     }
94.   }
95. }
```

Listing 6.

```
1. <?php
2.
3. namespace Drupal\health_sites\Controller;
4.
5. use Drupal\Core\Serialization\Yaml;
6. use Drupal\Core\Controller\ControllerBase;
7. use Symfony\Component\HttpFoundation\Request;
8.
9. /*
10.  * Controller for Order content type.
11.  */
12. class HealthSitesController extends ControllerBase {
13.
14.   /**
15.    * Display the markup of Hostnames list webpage.
16.    *
17.    * @return array
18.    *   Return markup array.
19.    */
20.   public function index(string $module_name, Request $request) {
21.     $routingFilePath = DRUPAL_ROOT . '/' .
                drupal_get_path('module', $module_name) .
                '/health_sites.routing.yml';
22.     $routingFileContents = file_get_contents($routingFilePath);
23.     $results = \Drupal\Core\Serialization\Yaml::
                decode($routingFileContents);
24.
25.     $links = [];
26.     foreach ($results as $key => $item) {
27.       if ($key != 'health_sites.main') {
28.         $links[$item['path']] = $item['defaults']['_title'];
29.       }
30.     }
31.
32.     return [
33.       '#theme' => 'health_sites_main_page',
34.       '#title' => $this->t('List of configure pages'),
35.       '#links' => $links,
36.     ];
37.   }
38. }
```

Must you configure sites? Build Forms

To configure a hostname and URLs to verify, I use forms.

Routes are defined in Listing 1. From lines 10 to 16, I define a route to configure the form to add the hostnames to verify. Lines 18 to 24 define a route to configure the form to add the hostname URLs to verify. From line 26 to line 32, I define a route to configure the form to verify hostname URLs.

HealthSitesConfigureForm.php

Complete code to implement a form to add the hostnames to verify is included in src/Form/HealthSitesConfigureForm.php in Listing 7 (downloadable). This form is displayed in Figure 6.

Line 9 defines the namespace for the form. Line 16 contains a definition of class, which extends the `FormBase` class. From line 21 to line 23, I generate the form id. Starting at line 28, I implement the abstract function `buildForm`. On line 32, I get configuration by health_sites.settings key.

```
$config = \Drupal::config('health_sites.settings');
```

On line 33, I get the hostnames list to verify if previously saved. `$config` has `get` function member that we use to retrieve string from settings, for example to line 21.

```
$hostnames = $config->get('hostnames_to_verify');
```

Line 34 I transforms the hostnames list array into a string to the Textarea field. I create a form with its elements on lines 36 to 47. Line 49 returns the form object to render into a page. From lines 55 to 59, I implement `validateForm` where I control that I don't submit a form without a hostname. Line 64 to 73 implement `submitForm`, where I save hostnames into `health_sites.settings configuration` key.

```
$config = \Drupal::service('config.factory')->
        getEditable('health_sites.settings');
$config->set('hostnames_to_verify', $hostnames) ->save();
```

`t` is a member function to translate a string in a selected language.

HealthSitesConfigureUrlsForm.php

Code to implement the form to add URLs to the selected hostname is included in `src/Form/HealthSitesConfigureUrlsForm.php` and shown in Listing 8 (downloadable). This form is displayed in Figure 7.

Line 9 defines the namespace for the form. Line 20 contains the definition of class, which extends the `FormBase` class. From lines 25 to 27, I generate the form id. Starting at line 32, I implement the abstract function `buildForm`. On line 34, I get configuration by health_sites.settings key.

```
$config = \Drupal::config('health_sites.settings');
```

On line 35 I get hostnames list to verify, if previously saved.

```
$hostnames = $config->get('hostnames_to_verify');
```

If no hostnames are saved, display a warning message on the screen and link to the `HealthSitesConfigureForm` page.

If hostnames were previously saved, display a form to enter URLs to verify, based on hostnames selected. This selection is possible thanks to the Drupal Ajax framework. In this case, I use this code fragment:

```
'#ajax' => [
  'callback' => '::getUrlsAjaxCallback',
  'disable-refocus' => FALSE,
  'event' => 'change',
  'wrapper' => 'wrapper-urls',
  'progress' => [
    'type' => 'throbber'
  ],
]
```

When you select hostname from the `hostname` field, this field launches an ajax event, which calls the `getUrlsAjaxCallback` member function to return previously URLs saved. Value is returned into the URL field, and all are made through the wrapper key, which is the wrapper div of the URLs field through the #prefix and #suffix key.

Lines 104 to 107 implement `validateForm` where I where I don't allow form submission without a URL.

From line 113 to line 126 I implement `submitForm` where I save urls into `health_sites.settings` configuration key.

```
$config = \Drupal::service('config.factory')->
    getEditable('health_sites.settings');
$config->set('urls_to_verify', $urls_to_verify) ->save();
```

HealthSitesVerifyHealthForm.php

Code to implement the form to verify the status of URLs of hostnames selected is included in `src/Form/HealthSitesConfigureUrlsForm.php` and shown in Listing 9 (downloadable). This form is displayed in Figure 8.

Line 8 defines the namespace for the form. Line 17 contains the definition of class, which extends the FormBase class. The form id is generated in lines 22 to 24. Starting on line 29, I implement the abstract function `buildForm()`.

First I get hostnames list to verify, if previously saved.

```
$config = \Drupal::config('health_sites.settings');
$hostnames = $config->get('hostnames_to_verify');
$hostnames = array_combine($hostnames, $hostnames);
```

This hostnames list populates the #options key of the hostnames field (list of checkboxes).

On click submit field `verifyUrlsAjaxCallback` ajax event is triggered. If you selected some hostnames verifyUrlsAjaxCallback verify status (UP or DOWN) URLs of these hostnames and return result into box field (markup type). If you haven't previously selected any hostname, `validateForm` triggers an error on the form and stops verification. `submitForm` function is implemented empty because it isn't needed, but being abstract not implementing it would trigger an error on the page.

Conclusion

In this tutorial, I have created a module and shown how to use a hook, create a form, and perform periodic actions. You saw that more functions and/or hooks were replaced by the yml configuration files (hook_permission for example) and objects (`\Drupal\node\Entity\Node::load for node_load`). I think the Drupal community will be divided between fans of these new features and people who would have preferred to remain faithful to hook's events and actions that are typical in earlier Drupal versions. I believe that everything evolves, and nothing is immutable, so let's start this new challenge given by the best open source CMS in the World.

Listing 9. Download the complete listing

```
...
106. foreach ($urls as $hostname => $list_urls) {
107.   foreach ($list_urls as $url) {
108.     $ch = curl_init($url);
109.     curl_setopt($ch, CURLOPT_HEADER, true);
...
121.     $query = \Drupal::entityQuery('node')
122.       ->condition('status', 1)
123.        >condition('type', 'history_url')
124.       ->condition('field_hostname', $hostname)
125.        >condition('field_url', str_replace($hostname, "", $url));
126.     $nid = $query->execute();
127.     if (empty($nid)) {
128.       $node = \Drupal::entityTypeManager()->
                          getStorage('node')->create([
129.         'type' => 'history_url',
130.         'title' => 'Status of ' . $url,
131.         'field_hostname' => $hostname,
132.         'field_url' => str_replace($hostname, "", $url),
...
136.       ]);
137.     }
138.     else {
139.       $nid = array_pop($nid);
140.       $node = Node::load($nid);
141.       $node->setNewRevision();
142.       $node->revision_log = 'Created revision for node' .
                          $nid . ' programmatically';
143.       $node->setRevisionCreationTime(REQUEST_TIME);
144.       $node->setRevisionUserId(1);
145.     }
146.
147.     if ($httpcode!=200) {
148.       $node->set('field_status_up', 0);
149.     } else {
150.       $node->set('field_status_up', 1);
151.     }
152.
153.     $node->save();
154.
155.     $form['box']['#markup'] .= t('Status of %url: %status',
156.        ['%url' => $url, '%status' => $node->
157.        get('field_status_up')->getValue()[0]['value']]).'<br>';
```

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

Refactor to Enums in Laravel

Marian Pop

You can create a custom type that is limited to one of a discrete number of potential values by using enumerations, sometimes known as "Enums." Because it allows for "making incorrect states unrepresentable," this is particularly useful when constructing a domain model.

So what are Enums?

Enums can be found in many different languages and have a wide range of functionalities. Enums are a unique object class in PHP. The Enum is a class in and of itself, and all of the possible cases are instances of that class that exist just once. Enum cases are therefore valid objects that can be used anywhere objects are allowed, including type checks. The built-in boolean type, an enumerated type having the permitted values true and false, is the most well-known example of an enumeration. You can create arbitrary, robust enumerations by using Enums.

Enumerations are now supported, starting with PHP 8.1. If you wanted to use Enums before PHP 8.1, you had a few options:

- Defining constants in a class
- Using a Laravel Enum package
- Using a database to store the values

In this article, we will refactor from using the database to store the values to using Enums.

I'm working on a side project where I have a Product model that has two statuses: 'ACTIVE' and 'INACTIVE.' Instead of using the database and a "statuses" table to store the statuses, we can now make use of Enums and only store the actual value that has been set for the model.

You can place your Enum class anywhere you like but for simplicity, let's create a new folder called "Status" and a new PHP file called "Status.php" (App.php).

```
namespace App\Status;
enum Status : string
{
    case ACTIVE = 'Active';
    case INACTIVE = 'Inactive';
}
```

In the snippet above, we moved the status values from the "statuses" table to the Enum class. Let's see how we can use the newly created Enum.

Before, when using the database to store the status values in the products table, I had the "status_id" column where I was storing the ID of the status from the "statuses" table. Since we're now using an Enum instead, we can change the migration as follows:

```
// before
$table->foreignId('status_id')->constrained('statuses');
// after
$table->string('status');
```

In our example, we use and store strings for the status, but you can also use an integer value and map each case to a corresponding int value:

```
enum Status : int
{
    case ACTIVE = 1;
    case INACTIVE = 2;
}
```

If you prefer to use an int value, make sure you define the table column as integer.

Now to fetch and display the statuses, you have to make a minor change to your blade files:

```
// before

{{ $product->status->name }}

// after
{{ $product->status }}
```

If you're using an int value and you want to map an integer to a Enum "name" you can make use of the `tryFrom()` helper:

```
{{ App\Status\Status::tryFrom($product->status)->name }}
```

In case you want to display a different text than the declared name, you can add a new method inside of your Enum class that will match the integer value to a case and then change the displayed string as needed:

```
public function getName() : string
{
    return match($this) {
        self::ACTIVE => 'active',
        self::INACTIVE => 'inactive',
    };
}
```

We can make use of our new method in the view file:

```
{{App\Status\Status::tryFrom($product->status)->getName}}
```

Marian Pop is a PHP / Laravel Developer based in Transylvania. He writes and maintains LaravelMagazine.com and hosts "The Laravel Magazine Podcast". @mvpopuk

The Birthday Paradox

Oscar Merida

We look at a solution to a problem that is not intuitive at first glance—the birthday paradox. Instead of calculating probabilities directly, we'll use a simple simulation to solve the problem.

The birthday paradox is so named because the resulting answer does not seem plausible. Assume that the chance you were born on a particular day of the year is 1/365. The chance that another person shares your same birthday is also 1/365, which is about 0.27%. The possibilities you have to account for increase as you add more people to the group. You'd have to consider the case that Person 2(P2) shares your birthday OR that Person 3 shares your birthday. In these cases, it's usually easier to think of the complementing case. That is, in a group of 3 people, the chance that at least two share a birthday is the complement (or opposite of) of the probability that none of them share a birthday. Mathematically, that is:

```
1-[P(P1 birthday is unique) AND P(P2 birthday is unique)]
1-[(364/365) * (363/365)]
```

The formula and the related probabilities grow in complexity with an increase in the number of people in the group. Moreso, because the chance one other person sharing our birthday is less than 1% (0.27%), most of us assume that we need a very large group of people (100? 200?) before we have a 50% chance of finding a birthday twin. But is that really the case?

Puzzle Recap

> *Assume birthdates are distributed across the calendar year with the same probability. Write a program that calculates the probability of two people sharing a birthday given there are N people in a group. What value of N brings that probability to 50%?*

Simulation

While the birthday problem has a formula[1] you can use to calculate the probability two people share a birthdate in a group of size `N`, I'd like to take another approach.

We can use multiple simulation runs to create any number of N-sized groups of people. We can check if at least two people share a birthdate within each group. If we tally how many groups contain a shared birthdate over a large enough number of runs, we can accurately estimate the probability.

I broke down this approach into the following pieces:

1. Generate a random birthday.
2. Generate a set with `N` birthdays
3. Test if a set has any shared birthdays
4. Loop over `M` sets and count how many have shared birthdays to calculate the frequency of shared birthdays.

Generating Birthdays

One simplification we can make at the start is to randomly generate birthdays within the same year. Doing so should simplify later comparisons. Plus, the year of birth is not relevant for our purposes. We want to compare if two people celebrate on the same day each year.

Generating random dates can be tricky. I tried the following but it didn't work:

```
$day = random_int(1, 365);
$date = new \DateTime($day . 'th of 2022');
```

You may be tempted first pick a random month and then choose a random day within the month. That could certainly work. However, you're slightly changing the probability distribution. If you first pick the month at random, you're ignoring that days with more months, like July and August, should have more birthdays than months with fewer days, like February.

If we pick a random number between 1 and 365 and then turn it into a date, we weigh each day equally. Of course, we are simplifying things here by ignoring leap years and ignoring historical data on how births are distributed in the calendar year based on real-world trends. If we had such data, we could later plug it into our approach to refine our estimate of the probability.

PHP's DateTime library is an unsung gem, particularly because it can interpret DateTime strings[2] which include relative dates. I used it to write a function that can return a random date for this year:

```
function getRandomDate() : \DateTime
{
    $offset = random_int(0, 364);
    return new \DateTime("January 1, 2022 +{$offset} days");
}
```

1 *a formula:* https://www.dcode.fr/birthday-problem

2 *DateTime strings:* https://php.net/datetime.formats.date

Sets of Birthdays

Now that we have a function that gives us a random date, we can use it to create an array of size `N` with that many dates. PHP's array functions are particularly handy for doing so without using loops.

```
function makeGroup(int $n) : array
{
    $dates  = range(0, $n-1);
    return array_map(function() {
        $birth = getRandomDate();
        return $birth->format('m/d');
    }, $dates);
}
```

I anticipate that it'll be simpler to compare a scalar string later to determine if two people share a birthdate. To do so, I've used the `format()` method to `DateTime` to keep only the month and day of the random date.

Finding Duplicates

We have an array with 2 or more strings representing birthdays. Do we need to loop through the array multiple times, comparing one item against the rest until we find a duplicate? Not if we take advantage of another array function.

Listing 1.

```
1. function hasDuplicates(array $dates) : bool
2. {
3.     // remove duplicates
4.     $uniq = array_unique($dates);
5.
6.     // if we didn't remove duplicates, both will have the same size
7.     if (count($uniq) === count($dates)) {
8.         return false;
9.     }
10.
11.    return true;
12. }
```

Multiple Runs

We now have everything we need to collect data and estimate the probability.

The workhorse of this approach is the `doRuns()` function (Listing 2). With it, if we have a group of 4 people, we can create `$runs` number of sets and count how many of them have at least one shared birthdate. If `$runs` is large enough, the number of runs with duplicates should approach the theoretical probability.

I wrapped `doRuns()` in a loop to collect data for various group sizes. See Listing 3.

If we do `10` runs, one estimate of the frequency for group sizes between 2 and 20 looks like the following. Note that the values fluctuate wildly (Output 1) and also are only accurate to the first decimal since we did 10 runs for each group.

Listing 2.

```
1. function doRuns(int $runs, int $size) : int
2. {
3.     $hasDuplicate = 0;
4.
5.     for ($i =0; $i < $runs; $i++) {
6.         $group = makeGroup($size);
7.         if (hasDuplicates($group)) {
8.             $hasDuplicate++;
9.         }
10.    }
11.
12.    return $hasDuplicate;
13. }
```

Listing 3.

```
1. $runs = 10;
2.
3. for ($size = 2; $size < 31; $size++) {
4.     $dupesFound = doRuns($runs, $size);
5.     echo sprintf(
6.         "%3d %12d %12.4f",
7.         $size, $dupesFound, $dupesFound/$runs
8.     ) . PHP_EOL;
9. }
```

Output 1.

```
 2           0       0.0000
 3           0       0.0000
 4           0       0.0000
 5           2       0.2000
 6           0       0.0000
 7           1       0.1000
 8           0       0.0000
 9           0       0.0000
10           0       0.0000
11           3       0.3000
12           3       0.3000
13           4       0.4000
14           2       0.2000
15           3       0.3000
16           3       0.3000
17           1       0.1000
18           3       0.3000
19           3       0.3000
20           3       0.3000
21           3       0.3000
22           6       0.6000
23           5       0.5000
24           6       0.6000
25           5       0.5000
26           9       0.9000
27           6       0.6000
28           8       0.8000
29           9       0.9000
30           6       0.6000
```

We have a tremendous amount of computing power now, so we can really crank up the number of runs we do. The more runs we do for each group size, the better our estimate is. Output 2 is the table for 1000 runs.

And Output3 shows what it looks like for 1,000,000 runs for each. Figure 1 shows the data below in a bar graph.

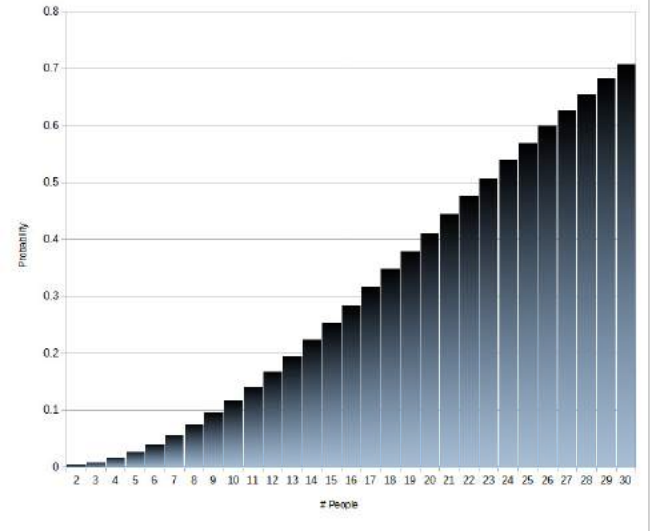
Output 2.

2	1	0.0010
3	5	0.0050
4	17	0.0170
5	28	0.0280
6	30	0.0300
7	63	0.0630
8	64	0.0640
9	89	0.0890
10	98	0.0980
11	118	0.1180
12	150	0.1500
13	197	0.1970
14	212	0.2120
15	268	0.2680
16	260	0.2600
17	300	0.3000
18	335	0.3350
19	360	0.3600
20	397	0.3970
21	423	0.4230
22	481	0.4810
23	498	0.4980
24	544	0.5440
25	589	0.5890
26	577	0.5770
27	624	0.6240
28	620	0.6200
29	686	0.6860
30	734	0.7340

Output 3.

2	2758	0.002758
3	8260	0.008260
4	16285	0.016285
5	27086	0.027086
6	40284	0.040284
7	56113	0.056113
8	74550	0.074550
9	94742	0.094742
10	116798	0.116798
11	140821	0.140821
12	167025	0.167025
13	194985	0.194985
14	223676	0.223676
15	252275	0.252275
16	283437	0.283437
17	315349	0.315349
18	347327	0.347327
19	378619	0.378619
20	411242	0.411242
21	443795	0.443795
22	476225	0.476225
23	507167	0.507167
24	538664	0.538664
25	568134	0.568134
26	598758	0.598758
27	626537	0.626537
28	654185	0.654185
29	681536	0.681536
30	707394	0.707394

Figure 1.



This accuracy is good enough to answer our question. Given our results, when you have 23 people in a group, the chance that at least two people share a birthday is 0.507 or 50.7%.

Decimals to Fractions

For next month's puzzle, let's work again with our favorite scalar type: floating-point values.

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".

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[3] 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

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

Blind

Beth Tucker Long

It has been a while since I've worked on in-person conferences, but with events starting back up, conference planning has started up again as well. I recently chatted with someone about improving diversity in speaker lineups. This is a very difficult topic with no easy answers, but it is one that I love talking about. I'm serious. I want you all to know that if any of you are interested in improving the diversity of your speaker lineup, I'm happy to talk through your process and offer suggestions.

Back to the topic at hand, though, the issue this conference was having was that the speaker selection committee thought they had solved any bias issues by starting to make blind talk selection, and they didn't understand why it wasn't working. They were still ending up with the same old non-diverse lineup.

Blind talk selection is when you do not display the speaker's name, employer, location, etc., while reviewing the talk titles and abstracts for talk selection. The purpose of doing so is to remove any identifying information so the talks can be evaluated solely on the merit of the talk itself. Blind selection is relatively easy to implement and has been a popular solution for addressing complaints of biased talk selection at an event. If you don't know who the speaker is, how can you be biased against them? The answer no one likes to hear from me is, "So very many ways."

There are so many flaws with a blind talk selection process, not the least of which is how many speakers put their own names or identifying information right in their abstracts. Even if you try to strip out speaker names, there's no way to remove all references without having someone manually go through every abstract individually to ensure there are no recognizable references. A speaker's name of "Elizabeth" on the submission form may be listed as "Beth" in the abstract, or you may end up with references like, "As the author of the *Finally* column for php[architect] magazine…" These kinds of references let anyone who is familiar with me know exactly who wrote the abstract, but it is very difficult to strip out programmatically.

Even if you have well-behaved speakers who are perfect at writing abstracts with no personally identifying information in them, selection committee members can recognize the tone, catch-phrases, or even the topics that certain speakers use. Many speakers reuse talks from other conferences, so the title of the talk may give away the speaker. Beyond identifying a certain speaker, abstracts can be used to identify a type of speaker, which can introduce bias. Consider these sentences:

1. Pulling from more than ten years of professional experience writing high-level code for a wide variety of clients, I will teach you the best strategies to use to estimate the time it will take to complete a project and ensure that your projects do not go into the red.
2. I've been a consultant for over ten years, and I have found a fool-proof way to estimate how long it will take to finish a project so you won't be losing money while working.
3. After 10 years of work, I've seen some messed-up projects, but I know exactly how long each job will take so I always get paid right.

These three sentences all describe exactly the same thing, but they are doing so in very different voices. There can be a lot of bias in manners of speaking, tone, and voice without knowing anything about who the speaker is. Blind talk selection just does not work. So as your local user group, trainings, and conferences start back up, make sure you are focused on bringing your biases to light. Everyone has them. It doesn't do any good to refuse to see them. Engage in tough discussions, dig deep into how you are evaluating things, and constantly ask yourself why you think that way. Embrace diversity, not blindness.

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

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

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

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

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