www.phparch.com

php[architect]

March 2016
VOLUME 15 - ISSUE 3

Mutant Testing

Education Station: Hunting Mutants with Humbug

Leveling Up: Ensuring Your Tests are Valuable

ALSO INSIDE

Object-Relational Mapping with Laravel's Eloquent

Integrating Heterogeneous Web Applications with Laravel and Guzzle

An Introduction to Compilers, Interpreters, and JITs

Getting Started with Zend Expressive

Community Corner:
Modern PHP

finally{}:
Growing Programmer Population

Security Corner:
Securing Legacy Applications—Part 1

WE'RE HIRING

QA

SOFTWARE QUALITY ASSURANCE ENGINEER

test
develop
collaborate
improve

self starter
security minded
experienced
attention to detail

APPLY	nexc.es/218WJØY

NEXCESS™
beyond hosting.

NEXCESS.NET | USA 1.866.639.2377 | UK 0.808.120.7609 | AU 1.800.765.472

php[architect]

CONTENTS

MARCH 2016
Volume 15 - Issue 3

Mutant Testing

Columns

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!

Managing Partners
Kevin Bruce, Oscar Merida, Sandy Smith, Eli White

php[**architect**] is published twelve times a year by:
musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

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

Print ISSN 1709-7169
Digital ISSN 2375-3544

Hunting Mutants

Greetings from the sunny mid-Atlantic USA, where the days are finally getting longer. This month, I wanted to write about uncertainty. I've been doing some client work recently and I realized (probably not for the first time) how important reducing uncertainty is especially for our profession. It's important in client communications.

For example, I had a client reach out to us on a Thursday that needed help for "something that was going to be deployed this Monday." The next Friday afternoon, we were in a mild-panic because we still didn't have a signed contract much less any code to review for fixing. After writing the client to tell them the Monday deadline would probably slip, I learned "Oh, you don't have to work the weekend because the previous developer isn't deploying the code to my server until Monday." By jumping to conclusions and not asking for clarification earlier our team created uncertainty and no small amount of anxiety for ourselves.

In the same vein, poor bug reports generate needless uncertainty which requires back-and-forth emails, calls, or comment threads to clarify exactly what "doesn't work" means. On a Drupal project, one bug report I received was "Form validation doesn't work". Of course, I fire up my trusty browser, test the form in question and find that validation is working just as *I* expect. Poking around the modules installed, I found a suite of Client-side validation modules enabled but not functioning. At this point, I had to go back and ask for clarification.

I could rattle off dozens of examples about how poor communications breeds uncertainty. We struggle with it daily, since chat is our main mode of communicating during the work day. We're constantly finding that a short hangout or phone call is much more effective for sharing feedback, explaining alternatives, and finding consensus especially when we need to convey any nuance. I'm sure you're thinking of your own examples, so I'd implore you to think a bit before you send that next email or chat message and review it. Include all the details you can. Look for words or phrases that are vague and offer to talk in person to clarify things.

Unless you've been cryogenically frozen for the past few years, you've no doubt heard about how important automated testing is as well as TDD, BDD, Continuous Deployment, and other acronyms. Either you've read an article in a past issue of this magazine or you've heard it from numerous advocates at conferences, on blogs, and on twitter. This month's issue features two articles on Mutation Testing with Humbug. In *Education Station: Hunting Mutants with Humbug*, Matthew Setter shows you how to install and get started with Humbug. In *Leveling Up: Ensuring Your Tests are Valuable* David Stockton shares how and when he uses it in his work to improve his code and test suite.

Also this month, Chris Tankersley takes a look at *Getting Started with Zend Expressive*. Luis Atencio dives into *Object Relational Mapping with Laravel's Eloquent* and shows two different ways to map object-oriented systems with a relational database. If you need to share authentication between related apps, *Integrating Heterogeneous Web Applications with Laravel and Guzzle* by Alexandros Gougousis will show you one approach. Joshua Thijssen pulls back he curtain on how code gets executed in *An Introduction to Compilers, Interpreters, and JITs*. Cal Evans defines what he sees as *Modern PHP* in *Community Corner* and Eli White wraps things up with what a *Growing Programmer Population* means for us in *finally{}*.

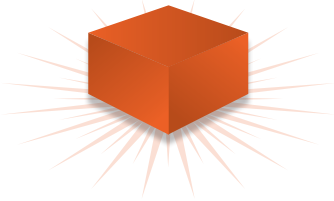
WRITE FOR US

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!

Download this issue's code package:

http://phpa.me/Mar2016_code



Object-Relational Mapping with Laravel's Eloquent

Luis Atencio

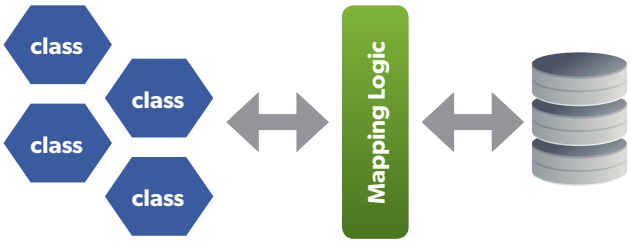
Object-relational mapping—ORM for short—has been a challenge in software system development for many years. In basic terms, this difficulty arises when attempting to match an object-oriented system to a relational database or RDBMS. Objects are very free and lenient structures that can contain many different types of properties, including other objects. While object graphs are free to grow organically via inheritance, relational entities are rather flat and contrived, defining a very limited set of types. So can you transpose an object-oriented model onto a relational model? While this remains a very hard problem to solve, there are ORM solutions that can emulate the same effect by creating a virtual object database. One such solution is Laravel's Eloquent ORM framework. Eloquent provides the abstractions needed for you to be able to work with relational data as if it were loaded onto an inherited object model.

Introduction

ORM Mapping is a technique for converting data from the world of objects into the world of relations (and vice versa), or tables in the sense of a typical RDBMS. This eliminates the need for building a very complex adapter layer in charge of reading relational data from, say, a MySQL database, into objects. ORM tools also abstract out the details of mapping logic, i.e., managing reads and writes, as well as one-to-one or one-to-many relationships.

In case you're not familiar with the technology, I'll provide brief introductions to both Laravel and Eloquent.

ORM Mapping **FIGURE 1**



Laravel

Laravel[1] is a PHP web MVC application framework designed to abstract the common pain points associated with applications pertaining to: authentication, routing, middleware, caching, unit testing, and inversion of control. In many ways, it's similar to the Rails platform that dominates Ruby web development. Built into Laravel is a component called Eloquent ORM.

Eloquent ORM

Eloquent ORM[2] is PHP's response to very successful ORM frameworks such as Hibernate and Rails, available in Java and Ruby respectively, for many years. These ORM tools nicely implement the *ActiveRecord*[3] design pattern that treats objects as relational rows in a database. This pattern puts the **M** in MVC—the model—which facilitates the creation of objects whose data need to be persisted and read from a database. In simple terms, in *ActiveRecord* the bulk of the logic for carrying out data access operations are shoved into your model classes, in contrast to having it all reside in a data access object (DAO). In this model, the class definition maps to the relational table per se, while instances of the object constitute the individual rows.

Common ORM Strategies

ORMs, like Eloquent, connect these rich objects to tables using different strategies. In this article, I will go over the two most popular ones:

- Concrete Table Strategy[4]
- Single Table Strategy[5]

Concrete (Class) Table Inheritance (CTI)

Because relational databases do not support inheritance (theoretically speaking), thinking of tables from an object instance point of view is incredibly challenging. Given that there is no automatic way for data to trickle down from "parent" tables to any "derived"

1 *Laravel:* https://laravel.com

2 *Eloquent ORM:* https://laravel.com/docs/5.0/eloquent

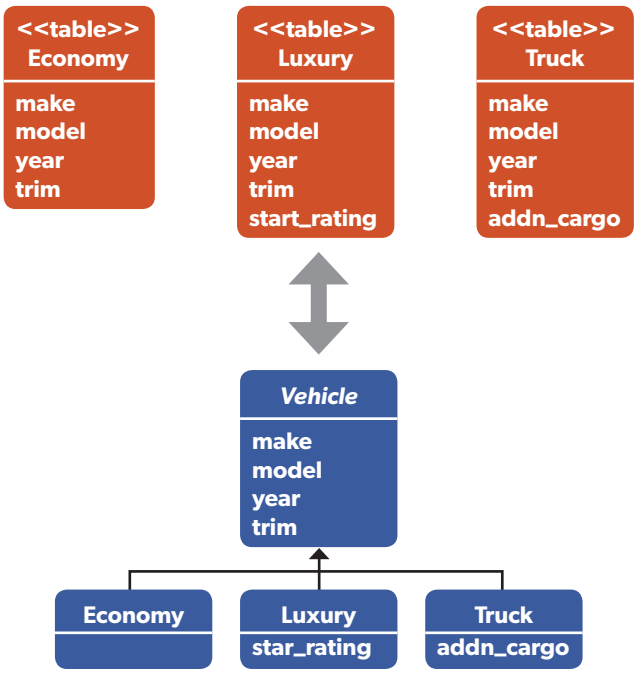
3 *Active Record Patter:* http://phpa.me/fowler-ar

4 *Concrete Table Inheritance:* http://phpa.me/fowler-sti

5 *Single Table Inheritance:* http://phpa.me/fowler-sti

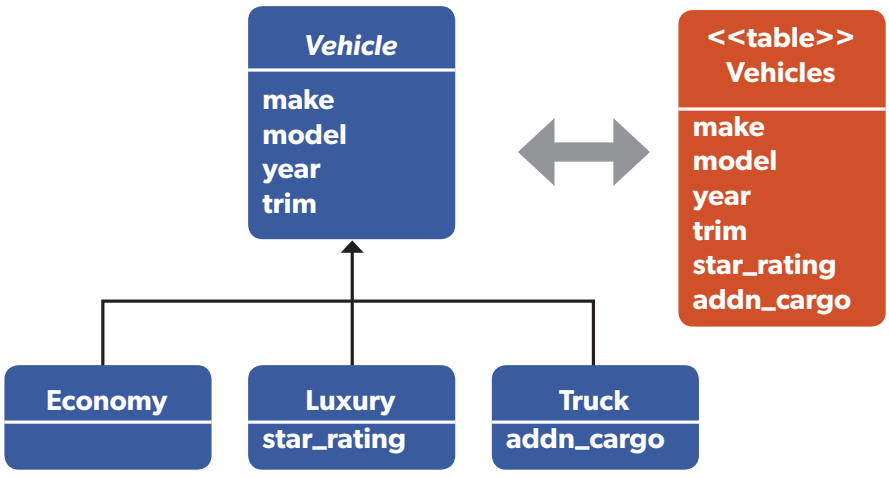
tables, it's natural to think that each child object would map to its own concrete table. Consider this very simple model for a car dealership application:

Concrete Mapping **FIGURE 2**



And this strategy would work well for very diverse classes or data types that share only a minimal set of attributes and contain many specialized child attributes, but results in lots of duplication when the amount of inherited data is greater. As in this case, you can see that I needed to repeat the *make*, *model*, *year*, and *trim* columns in every table. It seems that for this example, since we have very simple child types and flat class hierarchy, I could benefit from consolidating the data of an object model into a single table, known as Single Table Inheritance.

Single-Table Mapping **FIGURE 3**



Single Table Inheritance (STI)

Instead of creating dedicated tables for each child type, a single table is used to house all of the data contained inside an object graph. A *type* column, also known as a *Discriminator* column, is used to discern among the different types of objects in the graph.

Despite storing the data in a single table, ORM tools can easily map this onto different objects at runtime, abstracting the persistence strategy from you and the application. Let's jump into the implementation details.

Implementation

Before we get deep into the object definitions and queries, let's take care of some housekeeping application setup. Because Laravel and all of its packages are loaded via Composer, let's begin by defining the project level `composer.json` file. I will be using Laravel 5.0, see Listing 1.

LISTING 1

```
01. {
02.     "name": "phparch/eloquent-laravel-demo",
03.     "description": "PHPArch Laravel Eloquent Sample Code",
04.     "type": "project",
05.     "require": {
06.         "laravel/framework": "5.0.20",
07.         "phaza/single-table-inheritance": "1.0.1"
08.     },
09.     "require-dev": {
10.         "phpunit/phpunit": "4.6.2",
11.         "phpspec/phpspec": "~2.1"
12.     },
13.     "autoload": {
14.         "classmap": [
15.             "database"
16.         ],
17.         "psr-4": {
18.             "App\\": "app/"
19.         }
20.     }
21. }
```

Configuration

In Listing 1, I left out some of the other configuration parameters and kept the pertinent ones. If you've set up a Laravel application before, you should be familiar with this configuration. As you know, Laravel sets up the web application code under the `app` folder, which corresponds to the `App` namespace. Here you will find all of the code related to controllers, events, handlers, routing, etc.

Instead of sprinkling custom code everywhere, I will create a project directory for all of my custom classes called `EloquentDemo`.

Class Structure

The class structure (model) that drives all of the business logic and persistence will be stored in the `Model` directory, defining the package `App\EloquentDemo\Model`, underneath which my virtual object database will reside.

BaseModel

Every object that will be acting as an *ActiveRecord* must be an instance of Eloquent's `Model`. `Model` defines abstract behavior that applies generally to all instances. Because there's some general behavior that applies to all of my application objects as well, it's a good idea to create an abstract class between my custom classes and Eloquent's `Model` class. This will give you additional flexibility in the future to define shared properties and methods. Suppose I want to use soft deletes instead of hard deletes—I can easily install this trait in the base class (See Listing 2).

LISTING 2

```
<?php
namespace App\EloquentDemo\Model;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

/**
 * Base class for all models in the system
 *
 * @author Luis Atencio
 * @package App\EloquentDemo\Model
 */
abstract class BaseModel extends Model {

    use SoftDeletes;

    /**
     * Return the database record ID for all tables
     *
     * @return integer
     */
    public function getId() {
        return $this->id;
    }

    /**
     * Save the record ID
     *
     * @return mixed
     */
    public function setId($id) {
        $this->id = $id;
    }
}
```

Parent Type = Single Table

With this out of the way, I will create the object hierarchy that takes advantage of single inheritance. I will define an abstract `Vehicle` class, which corresponds to the `vehicles` database table, which I'll create with the Laravel migration script in Listing 3.

To implement the object-table mapping, I will use the **phaza/single-table-inheritance**[6] composer package (configuration shown in Listing 4) to instrument the parent type, via a trait.

This library requires a very minimal set of artifacts: a protected `$singleTableTypeField` field used to define the discriminator column and `$singleTableSubclasses` array containing the path to the model classes whose data will be shared by this single table. This is really clean because you don't have to expose the internal details of the inheritance mapping to Controller layer, allowing the trait to internally take complete control of these objects and their behavior.

6 phaza/single-table-inheritance: http://phpa.me/phaza-sti

LISTING 3

```
<?php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateVehicleTable extends Migration {

    /**
     * Create vehicle table
     *
     * @return void
     */
    public function up() {
        Schema::create('vehicles', function(Blueprint $table) {
            $table->increments('id');
            $table->string('vehicle_type');
            $table->unsignedInteger('make_id');
            $table->unsignedInteger('model_id');
            $table->unsignedInteger('trim_id');
            $table->unsignedInteger('year');
            $table->boolean('addn_cargo')->default(false);
            $table->tinyInteger('star_rating')->nullable();
            $table->timestamps();
            $table->softDeletes();
        });
    }

    /**
     * Drop vehicle table
     *
     * @return void
     */
    public function down() {
        Schema::drop('vehicles');
    }
}
```

LISTING 4

```php
<?php

namespace App\EloquentDemo\Model;

use DB;
use Phaza\SingleTableInheritance\SingleTableInheritanceTrait;

/**
 * Represents parent base model
 *
 * @author luisat
 * @package App\EloquentDemo\Model
 */
class Vehicle extends BaseModel {

    use SingleTableInheritanceTrait;

    /**
     * The database table used by the model.
     *
     * @var string
     */
    protected $table = 'vehicles';

    /**
     * STI table column name
     *
     * @var string
     */
    protected static $singleTableTypeField = 'vehicle_type';

    /**
     * Display name
     *
     * @var string
     */
    protected $displayName;

    /**
     * STI class names
     *
     * @var array
     */
    protected static $singleTableSubclasses = [
        Economy::class, Truck::class, Luxury::class
    ];

    /**
     * The attributes that are mass assignable when reading from DB
     *
     * @var array
     */
    protected $fillable = [
        'vehicle_type',
        'make_id',
        'model_id',
        'trim_id',
        'year',
        'addn_cargo',
        'star_rating'
    ];

    public function getType() {
        return $this->vehicle_type;
    }

    public function getMakeId() {
        return $this->make_id;
    }

    public function getModelId() {
        return $this->model_id;
    }

    public function getTrimId() {
        return $this->trim_id;
    }

    /**
     * Load the Make record for this vehicle
     * @return Make
     */
    public function make() {
        return $this->hasOne(Make::class, 'id', 'make_id');
    }

    /**
     * Load the Model record for this vehicle
     * @return Model
     */
    public function model() {
        return $this->hasOne(Model::class, 'id', 'model_id');
    }

    /**
     * Load the Trim record for this vehicle
     * @return Trim
     */
    public function trim() {
        return $this->hasOne(Trim::class, 'id', 'trim_id');
    }
}
```

Now, let's create the subtypes. To keep things short, I'll just show one of the child objects, `Truck` (see Listing 5); you can decipher the rest easily.

Now that we've set this up, let's show how easy it is to create and read records of any type.

Writing to the Database

I can write a record in the database explicitly by defining type (see Listing 6).

Better yet, I can leverage the functionality of STI to create and read items of any type as shown in Listing 7.

Reading from the Database

This single table strategy allows you to read data polymorphically for any vehicle type, and seamlessly taking care of any associations just like any other model class would. Consider the simple queries in Listing 8.

As you can see, by using Laravel's Eloquent ORM with this configuration there's absolutely nothing else you need to do to support STI. This is the real beauty of a setup like this. In an MVC world, you make your changes at the database and Model levels but your Controllers and Views are handled just like any other model. As far as you are concerned, each object has its own persistent storage.

Pros and Cons

There are many considerations to keep in mind when deciding whether STI is suitable for your application. Here's a short list of the pros and cons of using STI:

LISTING 5

```php
<?php namespace App\EloquentDemo\Model;

/**
 * Derived vehicle type: Truck
 *
 * @author luisat
 * @package App\EloquentDemo\Model
 */
class Truck extends Vehicle
{
    /**
     * Single inheritance table type
     *
     * @var string
     */
    protected static $singleTableType = 'Truck';

    /**
     * Display name for the model
     *
     * @var string
     */
    protected $displayName = 'Truck';


    public function getAdditionalCargo() {
        return $this->addn_cargo;
    }

    public function setAdditionalCargo($addnCargo) {
        $this->addn_cargo = $addnCargo;
    }
}
```

LISTING 6

```php
$data = [
    'vehicle_type' => 'Luxury',
    'make_id'      => Make::firstByAttributes(['name' => 'Toyota'])->id,
    'model_id'     => Model::firstByAttributes(['name'=> 'Camry'])->id,
    'trim_id'      => Trim::firstByAttributes(['name' => 'Camry LE'])->id,
    'year'         => 2015,
    'addn_cargo'   => false,
    'star_rating'  => 4
];

$camry = Vehicle::create($data);

//...

Vehicle::find($camry->id); //-> Vehicle(1)
```

LISTING 7

```php
Truck::create([
    'make_id'  => Make::firstByAttributes(['name' => 'Ford'])->id,
    'model_id' => Model::firstByAttributes(['name'=> 'F150'])->id,
    'trim_id'  => Trim::firstByAttributes(['name' => 'Limited'])->id,
    'year' => 2015,
    'addn_cargo' => true,
    'star_rating' => 5
]);
```

LISTING 8

```php
// Reading and updating a collection of truck records
Truck::where('year', '>', 2014)
      ->update(['trim_id' => Trim::firstByAttributes(
            ['name' => 'XL'])->id]
        );

// Load associations
Luxury::find($camry->id)->model; //-> Model('Camry')
Luxury::find($camry->id)->make;  //-> Make('Toyota')

// Query all vehicles in the database
Vehicle::all(); //-> Collection[Luxury(1), Truck(2), ...]
```

Pros

- Simple to implement, perhaps for quick spiking tasks and simple apps.
- Easy to add new classes by simply adding additional columns.
- Supports polymorphic queries with a simple discriminator column.
- Data access and reporting are quick since all the information is stored in one table.

Cons

- Tight coupling of objects to tables. A major refactoring effort on your classes would cause changes to your table structure as well.
- Space potentially wasted due to jagged data. This can occur when child types evolve with many unique attributes.
- Table quickly grows when supporting deeply nested hierarchies.

When to use

This is ideal for simple and/or shallow class hierarchies that have lots of overlap with parent types and clear inheritance hierarchy.

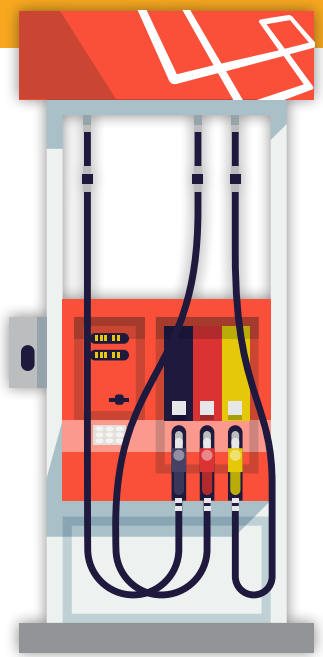
Final Comments

In this article I showed you how to use a Single Table Inheritance (STI) mapping scheme to model a simple object graph. This will allow you to emulate inheritance in relational databases. The caveat to this strategy, though, is that there must be a clear and natural class hierarchy; otherwise, STI may be hard to maintain. A common problem that might occur is that your child types develop too many unique attributes, creating lots of non-global columns in a single table.

ORM frameworks like Laravel's Eloquent can be used to work with several strategies very effectively, but ORMs are not the only solution. Other people gravitate toward the NoSQL databases, such as MongoDB, because it allows them more flexibility as data is stored more freely in a schemaless object form. This avoids having to translate between the contrived, restricted relational form into objects. Nevertheless the strong principles behind relational databases continue to be very appealing for developers and system architects, so ORMs are and will be used very frequently in modern web applications for many years to come.

Luis Atencio is a Staff Software Engineer for Citrix Systems in Ft. Lauderdale, FL. He has a B.S. and an M.S. in Computer Science. He works full time developing and architecting web applications with Java, PHP, and JavaScript. When he is not coding, Luis writes a developer blog at http://luisatencio.net focused on software engineering. Luis is also the author of Functional Programming in JavaScript[7] (Manning 2016). @luijar

7 https://www.manning.com/books/functional-programming-in-javascript



Integrating Heterogeneous Web Applications with Laravel and Guzzle

Alexandros Gougousis

Companies and organizations that offer more than a couple of web applications to their employees or community require some kind of integration, at least for access control (e.g., LDAP, CAS, etc.). But what if the target user group does not belong to the company? What if some applications are not written in PHP or are proprietary and don't make source code available? And what if, on the top of this, we want a higher level of integration (e.g., on a UI level)? This article will describe an integration architecture that provides a solution to this problem.

Introduction

There are often cases where a company, an organization, or just a team develops a bunch of web applications to be used by the public, generally a user group that is not associated with the company. So, solutions like LDAP, CAS, or Shibboleth cannot be applied to unify the authentication. Moreover, there is often a need to make obvious to the user that this group of applications has something in common (e.g., has been developed under the same project, is targeted to the same user group, etc.). This scenario, which is not uncommon, requires integration.

Need for Integration

The architecture that we are going to describe satisfies this need for integration in the following respects.

First of all, access control. All applications use a common user database. In that way, the user does not have to sign up (register) or log in separately for each application. Users sign up once, and after logging in can access all the applications they have permission to use. Going a step further, a central access control mechanism is created. Access to each web application can be controlled (granted or revoked) from a single point. This access can be granted (for each application or generally) by default to a newly registered user or manually to specific users or groups of users. Since this is controlled centrally, we can configure each application separately. Each application could have its own fine-grained access controls as well.

Second, integration on the UI level is achieved by providing a common graphical wrapper to all integrated web apps. This wrapper not only makes it obvious to the user that these applications are provided under the same "umbrella," but also makes certain actions always available to the user, no matter which application the user is using at any moment. The simplest example of such a case is a logout action that is always available at any page of any app.

For the rest of this article, we will use the Laravel Framework and the Guzzle PHP HTTP client.

Access Control Integration

The proposed architecture is based on the development of a *core app* that provides central services to all other *integrated applications*. We need to make this distinction apparent before we proceed to the architectural details. The core app is going to be developed separately and without any dependency to other applications that are going to be integrated. The core app will be responsible for all the functionality that is not relevant to a specific application, like login, logout, user profile management, and so on. Of course, the user need not know about it. Let's assume that the core app is assigned the `myportal.com` domain name. In order to have a common user database among all applications we need to share the user state between the core app and all other apps. This will be done by sharing session information from the core app (the user logs in through the core app, so the core app is the one that holds the user status). This "session sharing" will take place by sharing the session cookie of the core app with all the other apps. How can this be done? In two steps:

1. setting the session cookie to `.myportal.com` , thereby making it available to this domain and all its subdomains, and
2. deploying all other applications to subdomains of this domain (e.g., `app1.myportal.com` , `app2.myportal.com`, and so on). This naming scheme comes almost naturally, since we are talking about applications that are being offered by the same "umbrella."

Each time we want to integrate a new application, we create a relevant record for this application into the core app's database. This record should contain some information about the application under integration, including a short code name, a description, the subdomain this application will run under, and a pair of username/password credentials (the purpose of these credentials will be made obvious in a while).

Sharing the session requires you to trust the integrated applications to a greater degree than other random sites on the network. This can work for applications under the same domain, where presumably you know the other developers and development processes to review code and fix any potential issues. For a real heterogeneous network of sites, you should consider other solutions, like OAuth.

The next step is to set up (in core app) an endpoint that will be called by other applications in order to retrieve information about the user status. The URL can be in the format of:

```
https://myportal.com/api/v1/session/info/<codename>
```

where `<codename>` can be the short code name that we mentioned above. This code name can also be used to form the subdomain of the application (e.g., `app1`).

Don't forget to use SSL for security!

Our URL definition is shown in Listing 1, where `{codename}` is the code name given to every application that has been integrated to the portal. We used Route::group in case we wanted to add more URLs in the future, to provide more functionality through the core app. Requests to this URL will be served by the info() method of SessionController controller. The `'before' => 'auth.api'` part is used to ask for HTTP Basic Authentication, but we will get back to this in a while.

LISTING 1

```
01. Route::group(
02.     array(
03.         'https',
04.         'prefix' => 'api/v1/session',
05.         'before' => 'auth.api'
06.     ),
07.     function(){
08.         /// ...
09.         Route::get('test/{codename}',array(
10.             'uses'=>'SessionController@info'
11.         ));
12.         /// ...
13.     }
14. );
```

Integrated applications should make requests to this URL piggybacking the session cookie of the core app (if one was sent by the browser). A sample request using the Guzzle HTTP client is shown in Listing 2. `SESSION_NAME` is a constant that holds the name given to the session cookie by the core app.

LISTING 2

```
15. $client = new \Guzzle\Service\Client('https://myportal.com/api/v1/session');
16. $request = $client->get('info/app1');
17. $request->setAuth(<app1_username>, <app1_password>);
18. $request->addHeader(
19.     'Cookie',
20.     SESSION_NAME . "=" . $_COOKIE[SESSION_NAME]
21. );
22. $response = $request->send();
```

The `setAuth()` method is used for HTTP Basic Authentication and uses the application's credentials that we mentioned earlier. Again, notice that we use HTTPS, so that traffic between the apps is encrypted. This is used to validate that the application making the request is the one it claims to be. It is one way to ensure that only legitimate web applications are allowed to request information about a user's state. You may ask, why not use the same pair of credentials for all web apps that are integrated and simplify the URL to:

```
https://myportal.com/api/v1/session/info
```

The main reason is security isolation. For example, if one application is compromised, then users' permissions related to other applications will not be available (see the response format later in this article) to the hacker. The same holds for any other information related only to the application that made the request and may be sent as part of the user's state.

Multi Auth

The last obstacle in our access control integration is the need for multiple (in our case, two) tables/models to support multiple authentication types. One is used to authenticate real users (let's call them GUI users) and the other authenticates requests for user state information by an integrated app (let's call them API users—one user for each integrated web app, since each web app needs a pair of username/password credentials). Instead of building our own authentication mechanism from scratch, we can take advantage of Laravel Multi Auth package by Ollie Read[1]. You can easily install this package using Composer.

Next, we need to replace Laravel's default authentication providers with the new ones, modifying `/app/config/app.php`:

```
'providers' => array(
    ...
    //'Illuminate\Auth\AuthServiceProvider',
    'Ollieread\Multiauth\MultiauthServiceProvider',
    ...
)
```

After defining the two user tables (e.g., `api_users` and `gui_users`) and models (e.g., `ApiUser` and `GuiUser`), relevant configuration should be placed in `/app/config/auth.php`. The configuration can be found in Listing 3.

LISTING 3

```
23. //'driver' => 'eloquent',
24. //'model' => 'User',
25. //'table' => 'users',
26.
27. 'multi' => array(
28.     'gui' => array(
29.         'driver' => 'eloquent',
30.         'model' => 'GuiUser'
31.     ),
32.     'api' => array(
33.         'driver' => 'eloquent',
34.         'model' => 'ApiUser'
35.     )
36. )
```

With our new authentication provider set, we can now define authentication filters for our routes (in `/app/routes.php`). Two examples are shown in Listing 4.

LISTING 4

```
37. // Should be an API user
38. Route::filter('auth.api',function(){
39.     Config::set('session.driver', 'array');
40.     return Auth::api()->onceBasic('username');
41. });
42.
43. // Should not be logged in
44. Route::filter('is_visitor',function(){
45.     if(Auth::gui()->check()){
46.         return Redirect::to('home');
47.     }
48. });
```

1 *Laravel Multi Auth:* https://github.com/ollieread/multiauth

The first filter uses a stateless HTTP Basic Authentication. If you go back to the definition of the URL for retrieving user state, you will see that this filter has been used to ensure that user state can be obtained only by API users and after successful authentication. Stateless authentication is a convenient solution when we don't need to set up new sessions or set new cookies. After all, this is supposed to be a one-shot request.

Checking if a GUI user has been authenticated can be easily done inside SessionController controller:

```
// If the request was made in part of a logged in
if(Auth::gui()->check()){
   // ...
}
```

The Core App Response

The method `info()` in the `SessionController` controller of the core app responds to such a request in the following way:

1. Identifies the calling application (this is done automatically by HTTP Authentication). If the authentication fails, a response will not be sent (actually a 401 response will be sent) and the IP source of this request will be logged.
2. Checks if a session cookie was sent (remember that the core app's session cookie is supposed to be sent along with this request and should be accepted by the core app, since it comes from a subdomain). If a session cookie was not sent, the user is a visitor.
3. Checks if the user is logged in. If not (even if a session cookie was sent), the user is a visitor.
4. If the user is not a visitor, retrieve information about the user.

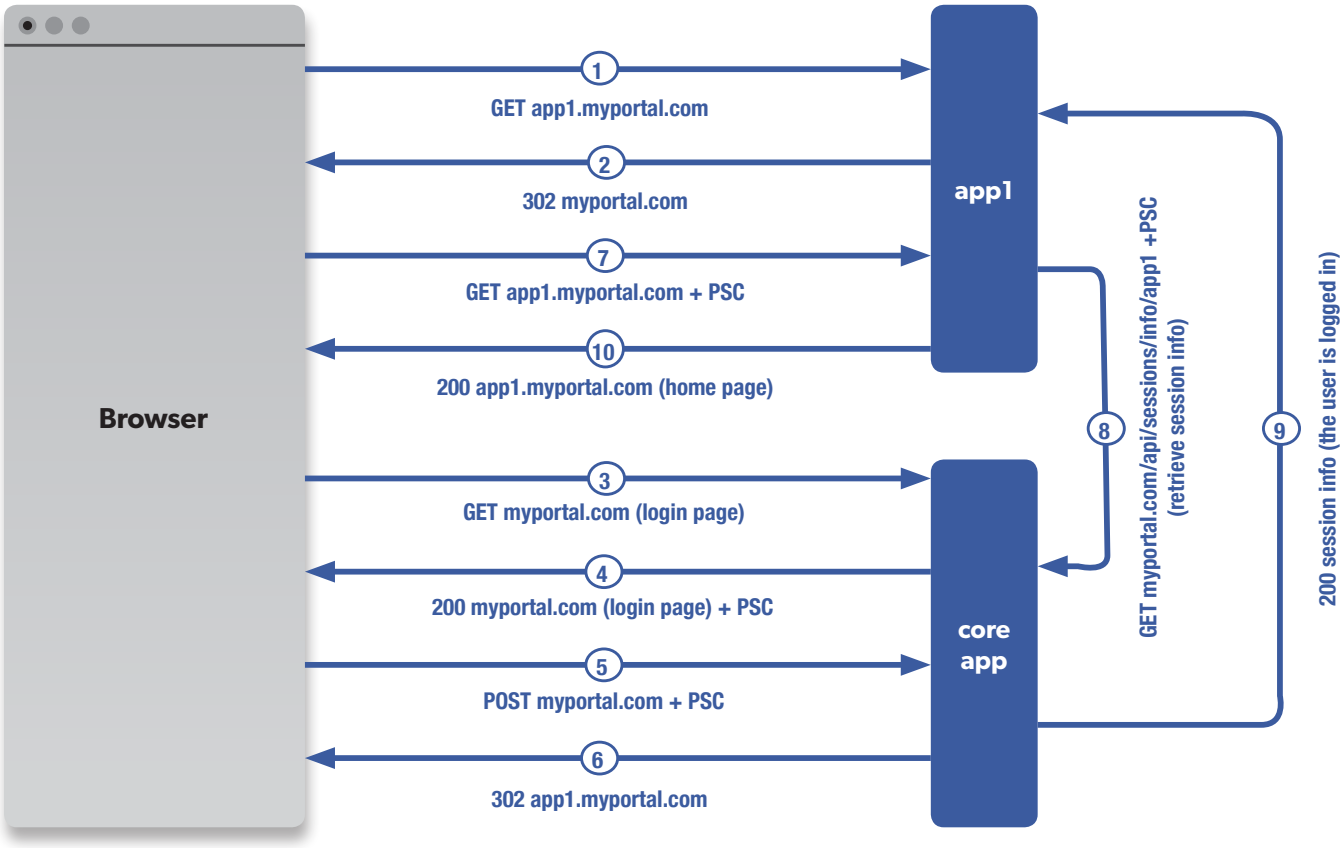
The response should, at the least, include information about user status (logged in, visitor) and the username, so the integrated application can know who this user is.

```
$response = [
   'status' => 'logged',
   'email'  => 'user1@gmail.com'
];
return Response::json($response,200);
```

Each integrated app is responsible for handling the response appropriately. Normally, if the user is a visitor, the user should be redirected to core app's login page. This request should take place in the beginning of every script that is directly accessible to the user (not the scripts that are included/required by other scripts). This is especially easy in applications built using MVC architecture or, generally, applications that use a central file through which all the traffic goes (e.g., an `index.php` file that most CMSs use). A typical flow of requests that takes place when a user who is not logged in tries to access the integrated application app1 is illustrated in Figure 1. For clarity, PSC = core app's session cookie.

Of course, not all integrated applications need to require that the user log in first. Many access levels can be used. For example, some of them can be publicly accessible, others can be, by default, accessible to a logged in user, and still others may require that we manually assign that privilege to a user. Such functionality can be easily built into the core app, which will embody this piece of information in the user state returned by `SessionController` controller.

Typical flow of HTTP requests **FIGURE 1**



LISTING 5

```
// /views/ui_wrapper/head.php

<link rel="stylesheet" href".../bootstrap.css" />
<script src".../jquery.js"></script>
<script src".../bootstrap.js"></script>

// /views/ui_wrapper/body_top.php

<div class="container">
   <div class="panel panel-default">
      <div class="panel-heading">
         Common Header
      </div>
      <div class="panel-body">

// /views/ui_wrapper/body_bottom.php

      </div>
      <div class="panel-footer">
         Common Footer
      </div>
   </div>
</div>
```

LISTING 6

```
Listing 6
$head = View::make('ui_wrapper.head');
$body_top = View::make('ui_wrapper.body_top');
$body_bottom = View::make('ui_wrapper.body_bottom');

$response = array(
   'status' => 'logged',
   'email'  => 'user1@gmail.com',
   'head' => "$head",
   'body_top' => "$body_top",
   'body_bottom' => "$body_bottom"
);
```

UI Integration

Integrating the UI can be achieved by using a common UI wrapper to all web applications. An easy way to do this is to supply the wrapping HTML parts from the core app through the response that we described earlier. You can build, for example, three templates like the ones in Listing 5 and load the template HTML to responses, like in Listing 6.

Of course, a number of template groups can be used for various cases (we may have applications that are accessible without requiring the user to log in, and so the UI wrapper should be different when the user is not logged in) and selected based on the user's state or other information.

The integrated application can embed the wrapping part to its pages following the format shown in Listing 7.

LISTING 7

```
<!DOCTYPE html>
<html lang="en">
   <head>
      <?= $response['head'] ?>
      <!-- HEAD: YOUR CODE GOES HERE -->

      <!-- HEAD: END OF YOUR CODE -->
   </head>
   <body>
      <?= $response['body_top'] ?>
      <!-- BODY: YOUR CODE GOES HERE -->

      <!-- BODY: END OF YOUR CODE -->
      <?= $response['body_bottom'] ?>
   </body>
</html>
```

If the UI wrapper uses some JavaScript or CSS libraries, these libraries will be loaded to the integrated app through `$head`. In that case, the developer of the integrated app should be notified in advance in order not to use other libraries that may create conflicts. Imagine a case where jQuery is used for the UI wrapper and the app developer builds the interface using a different version of jQuery or even MooTools. I think you get the point.

Advantages

This integration architecture has a number of advantages. An overview of the most important ones follows.

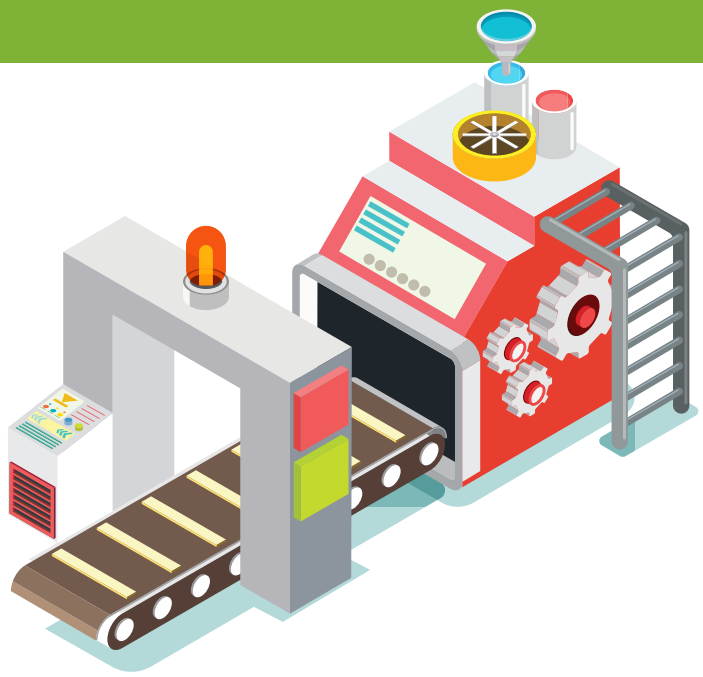
- **language-agnostic architecture:** The web applications that will be integrated, as well as the core app, can be developed in any programming language (PHP, Java EE, .NET, Python, etc.). You may need to use a custom session handler and a portable format for serializing and unserializing session information.
- **independent development:** The design, development, and maintenance of each web application does not require the involvement of the developers of other applications or the core app. The core app developer needs to be involved only during the integration for applications that were developed before the core app and not for other ones, since guidelines and documentation to be followed should have been given to other developers in advance.
- **web app security isolation:** The security of the core app is separated from the integrated applications. Each applications can handle authorization independently. If one application is compromised, the effect on others is minimized. Even if you're an *Admin* user on one application, this shouldn't grant you the same level of access across all applications.
- **integration of open-source apps:** Integration is not limited only to custom applications. Open-source applications, especially those that follow an MVC architecture, can also be integrated with small-scale changes both in access level and interface level.
- **workspace development made possible:** The ability of the core app to offer centralized services to all integrated applications makes the development of a user workspace possible.
- **location independent development:** Each application behaves as an independent web application as far as the execution environment is concerned. So, if network traffic starts to build due to the high number of integrated applications residing on the same server, some of them can be moved to different VMs or hosts.

Alexandros Gougousis is a seasoned contract developer with an Electrical and Computer Engineering degree and a system administration background. He mostly develops in PHP and sometimes in J2EE, likes team working, and hates debugging undocumented web applications that someone else wrote. In his free time, he likes to play bouzouki and read history.

Requirements:

- PHP: 5.4+
- Laravel 4.2
- Guzzle 3 (Guzzle 6 is already available)

An Introduction to Compilers, Interpreters, and JITs



Joshua Thijssen

With the birth of PHP 7, there is and has been a lot of talk about abstract syntax trees, just-in-time compilers, static analysis, etc. But what are these terms about? Are they magic features that make PHP go much faster? And if so, how does it all work? In this article, I will discuss the basics of how computer languages work and explain the processes that must be in place before a computer can actually run, for instance, your PHP script.

Interpreting Code

Before we talk about how things work, let's dive into a simple example. Imagine we have a new language (you can choose its own name). This language is fairly simple:

- each line forms a *statement*,
- and each statement consists of a *command* (an operator),
- and zero or more *values* (operands) to operate on.

An example:

```
set a 1
set b 2
add a b c
print c
```

It's a simple language, so we can safely assume that this program does nothing more than print the number "3" on our screen. The `set` operator takes a variable and a number to assign it (just as `$a=1` would in PHP). The `add` operator takes two variables to add and stores the result in the third variable, and the `print` operator takes a variable and prints it to the screen.

Now, let's write a program that reads each "statement," finds the operator and the operands, and does "something" with it, according to the operator. In fact, it's quite easy to write in PHP as you can see in Listing 1.

This program is very basic, and you are not going to write your next web application in our new language, but it's a start and gives you an idea of how easy it is to set up a new language and have a program that can actually read and execute this new language. Our program reads source files line by line, parses the operator and operands, and executes some code based on the operator. We don't need to deal with converting our application into weird languages like assembly or binary code in order for it to run: it runs perfectly as-is. This method of executing programs is called "interpreting" and is, for example, the way languages like BASIC often run: each statement is read and executed immediately on a high level.

But there are problems, the major one being the fact that writing a language processor this way might be easy, but the execution of our new language will be very slow. We must process each line and check:

- Which operator needs to be executed?
- Is the operator actually a valid one?
- Does it have the right number of operands?

LISTING 1

```
01. <?php
02.
03. $lines = file($argv[1]);
04.
05. $linenr = 0;
06. foreach ($lines as $line) {
07.     $linenr++;
08.     $operands = explode(" ", trim($line));
09.     $command = array_shift($operands);
10.
11.     switch ($command) {
12.         case 'set' :
13.             $vars[$operands[0]] = $operands[1];
14.             break;
15.         case 'add' :
16.             $vars[$operands[2]] = $vars[$operands[0]] + $vars[$operands[1]];
17.             break;
18.         case 'print' :
19.             print $vars[$operands[0]] . "\n";
20.             break;
21.         default :
22.             throw new Exception(sprintf("Unknown command in line %s\n", $linenr));
23.     }
24. }
```

And we must take care of other tasks, too. For example, can the `SET` operator only assign numerical values to variables, or could it also assign strings, or even other variables? Each statement must deal with these questions and handle appropriately. What happens when somebody types `set 1 4`? Writing a fast application this way is practically impossible.

Even though interpreting is relatively slow, it does have one advantage: every time we change something in our program, we can run that program immediately. For those paying attention: when I change something in a PHP script, I can run that script immediately and I will see the changes, so does that mean that PHP is an interpreted language, too? For now, let's assume the answer is yes, your PHP script is interpreted in pretty much the same way as our simple invented language—but we will revise that answer in the next sections!

Transcompiling

How can we make our program "run fast"? There are multiple ways to do this, and one is the way HipHop from Facebook worked (note: I'm talking about the "old" deprecated HipHop system, not the HHVM that is available now). What HipHop did was convert one language (PHP) into another language (C++). Once the PHP source is converted to C++, it can be compiled by a regular C++ compiler directly into binary code. At this point, a computer directly understands the code and can execute it without the overhead of an interpreter. This saves a HUGE amount of the work that a computer needs to do, resulting in a much-faster application.

This method is called source-to-source compiling, or transcompiling or even transpiling. It does not really compile anything to binary, but rather converts it into something that can be compiled to machine code by existing systems like C or C++ compilers.

By using transcompiling we get the benefit of directly running binary code, which is fast. The downside of this method is that we have a transcompilation phase plus the actual compilation phase before we can actually run our code. However, these two steps only need to be executed when your application changes, something that happens only during development.

Another usage for transcompilation is to make "hard" languages easy and more dynamic. For instance, LESS, SASS, and SCSS are languages that cannot be understood by a web browser, but are compiled into CSS by a transpiler. This makes maintaining CSS much easier, but requires the extra "transcompilation" step.

Compiling

The next step in making things fast is to see if we can skip the transcompilation phase, and compile our language directly into binary code. If we could compile our source code directly into binary code, then this binary code could run instantly without any interpreter overhead, just as with transcompiling. Plus, we don't have the overhead of the source-to-source compilation.

Unfortunately, writing a compiler is one of the hardest things to do (and do right) in computer science. One thing, for instance, we need to take into account when compiling something into binary code is on what kind of computer our binary code will be running: will it be running on a 32-bit Linux machine, or a 64-bit Windows machine, or maybe even on OS X? Our interpreted script can run just as easily on a Linux machine as it can on a Windows machine. Just as in PHP, we do not need to worry if our script is run on a Windows or Linux machine (there could be operating-system-specific code that might make running your script on another system impossible, but that is not the interpreter's fault).

And even if we could get rid of the transcompiling phase, we would still have the compilation phase. Large programs written in C (a compiled language), for instance, might take minutes—even close to an hour—before they are completely compiled. Can you imagine writing a PHP application where you need to wait 10 minutes before you can actually see if your change worked?

Using the Best of Both Worlds

If interpreting results in slow execution, and compilation is too difficult to implement and would result in slower development time, how do languages like PHP, Python, Ruby, and such work? They seem to be pretty fast!

In order to get this fast, they will actually use **both** interpreting **and** compiling. Let's explain how that works.

What if we could convert our own language we created above not directly to binary code, but to something that looks awfully close to binary code (we call this the "bytecode"). And what if this bytecode was so close to how a computer internally works that interpreting this bytecode could be done at a fast-enough speed (it could, for instance, interpret millions of these bytecodes per second). That would give our applications the speed remarkably close to the speed of compiled code, while allowing us to keep all the advantages of an interpreted language. Primarily, we would not need to compile our scripts each time we made a change in them.

It seems like a win-win, and in fact, this is how many languages actually operate: PHP, Ruby, Python, and even Java work in similar ways. Instead of reading source lines and interpreting them one by one, these languages use a different approach:

- Step 1: Read the complete (PHP) script into memory.
- Step 2: Convert/Compile the complete script into bytecode.
- Step 3: Execute this bytecode through the (PHP) interpreter.

There are actually more steps and in practice it's much harder than it sounds, but generally these three steps are all that is needed to run your script from the command line or when executing a request through your web server.

You might notice that this flow can be easily optimized: Let's assume that we are running a web server and each request will execute the script `index.php`. Do we need to load that `index.php` into memory each and every time? Wouldn't it be wiser to actually have that file cached in memory so we can quickly convert it the next time we need to execute it?

And another optimization: Once we have generated this bytecode, for a second request to that file we could use the same bytecode. So we could just as well cache the bytecode instead (but make sure that when the source file changes, we recompile the source into bytecode). This is exactly what opcode caches like the OPCache extension in PHP do: caching of compiled scripts so we can execute them quickly in next requests without the overhead of loading and compiling them to bytecode first.

The final step of gaining speed is where we execute our bytecode in our PHP interpreter. We will discuss in the next section how this will speed things up in contrast to a regular interpreter. To avoid confusion, such a bytecode interpreter is often called a "virtual machine": it mimics a machine (a computer, actually) on a certain level. This might be confused with the virtual machines we run on our computers like VirtualBox or VMWare, but many

people are familiar with the JVM, which stands for Java Virtual Machine, and in the PHP world, HHVM, which stands for HipHop Virtual Machine, is a known virtual machine. In a sense, our virtual machines are highly specialized and fast interpreters of bytecode. Python has its own virtual machine, as do Ruby and PHP. All these machines can run their own special bytecode generated from original Python, Ruby, or PHP code, but they are not compatible. You cannot run PHP bytecode on a Python VM and vice versa. You COULD, however, create a program that compiles PHP scripts into bytecode understandable to a Python VM, so in theory you could run your PHP scripts in Python that way! (Awesome challenge detected here!)

Bytecode

Before we talk about how bytecode gets generated from source code, let's talk about what bytecode looks like and how it works. Let's take a look at two examples. First, this PHP code:

```
$a = 3;
echo "hello world";
print $a + 1;
```

You can see PHP bytecode through http://3v4l.org or if you're inclined, by installing the VLD extension[1]. Our example above, generates the following:

```
line     #* E I O op    fetch   ext  return  operands
-------------------------------------------------------
   2     0  E >   ASSIGN                     !0, 3
   3     1        ECHO                       'hello+world'~
   4     2        ADD                ~1      !0, 1
         3        PRINT              ~2      ~1
         4        FREE               ~2
         5      > RETURN
```

If you look at a similar example in Python:

```
def foobar():
    a = 1
    print "hello world",
    print a + 4
```

The (C)python opcodes can be generated by python directly: `dis.dis(func)`[2]:

```
1           0 LOAD_CONST               1 (1)
            3 STORE_FAST               0 (a)

2           6 LOAD_CONST               2 ('hello world')
            9 PRINT_ITEM

3          10 LOAD_FAST                0 (a)
           13 LOAD_CONST               3 (4)
           16 BINARY_ADD
           17 PRINT_ITEM
           18 PRINT_NEWLINE
           19 LOAD_CONST               0 (None)
           22 RETURN_VALUE
```

We have two simple scripts, one PHP and one Python, together with their generated bytecodes. Note that both the PHP and Python bytecodes look similar to the language we created at the start of the article: each line is an operator with zero or more operands. In the case of the PHP bytecode, variables are actually prefixed with a `!`, so `!0` means the variable `0`. The bytecode does not care that your script uses the variable `$a`: in the compilation process, variable names lose their meaning and are converted into numbers. That's ok, since it will make it actually easier and faster for the virtual machines to execute. Many of the "checks" that need to be done are completed during the compilation phase. This means that we can skip a lot of checks in our virtual machine, and thus speed things up.

Because bytecode consists of just simple instructions, interpreting bytecode is very fast by itself. Instead of the maybe thousands of binary instructions a computer must process for each statement in our interpreted language, inside a bytecode virtual machine, it may only need a few hundred instructions per statement (sometimes even less). This is how virtual machines can operate with much higher speeds than interpreted languages.

So when using these virtual machines, we get the best of both worlds. We don't have to worry about compilation; it still needs to compile source code to bytecode, but this process is quick and transparent. And when we have our bytecode, a virtual machine can interpret it quickly and efficiently without too much overhead, resulting in a performant application.

From Source to Bytecode

Now that we are able to execute generated bytecode efficiently, the task remains of compiling our source code into this bytecode. Consider the following three PHP statements:

```
$a = 1;

$a=1;

$a
=
1;
```

All three statement are equally valid, and should ultimately be converted into the same bytecode. But how do we read them? In our own interpreter we actually parse each command by splitting them on spaces. This must mean that a programmer has a single way to write the code, unlike PHP where you can argue if you want to use tabs or spaces, brackets on the same line, or the second line, etc. The first step a compiler will take is to try to convert your written source code into tokens in a process called lexing (also known as tokenizing).

Lexing

The lexing step consists of converting your PHP source code—without understanding its meaning—into a long list of tokens. It's a difficult process, but you can do something similar to this in PHP quite easily. The code in Listing 2 produces the following output:

```
T_OPEN_TAG              <?php
T_VARIABLE              $a
T_WHITESPACE
=
T_WHITESPACE
T_LNUMBER               3
;
T_WHITESPACE
T_ECHO                  echo
T_WHITESPACE
T_CONSTANT_ENCAPSED_STRING "hello world"
;
```

As you can see, it processes a string and converts it into tokens: the `<?php` is converted into a `T_OPEN_TAG` token. The `$a` is converted to a `T_VARIABLE` token, and consists of the value `$a`. The tokenizer knows that when reading source code, when it encounters a `$` followed by an alpha character followed by zero or more alphanumerical characters. Numbers are tokenized as `T_LNUMBER` and

1 Vulcan Logic Dumper: https://derickrethans.nl/projects.html#vld

2 Python Disassembler: https://docs.python.org/2/library/dis.html

consist of one or more digits.

This tokenizing allows us to get source code into a more-structured form without worrying about how a programmer has written it. But as said before, it will not understand the meaning of the tokens: it will perfectly tokenize `$a = 1;` as well as `1 = $a`. That is ok though, as we will define meaning to the stream of tokens in the next stage: the parsing.

Parsing

When we parse our tokens, we must come up with some "rules" that make up our language. One rule could be: the first token we encounter in a program must be a `T_OPEN_TAG` token (which we saw in the example above corresponds to `<?php`). Another rule could be: an assignment is any `T_VARIABLE` followed by an `=` and then followed by a `T_LNUMBER`, a `T_VARIABLE`, or a `T_CONSTANT_ENCAPSED_STRING`. In essence, this would define that we allow `$a = 1;` or `$a = $b;` or `$a = 'foobar'`, but not `1 = $a`, as that does not adhere that rule.

When the parser finds a series of tokens that do not match any of our rules, it will automatically result in a `syntax error`. There is a lot more to parsing and lexing, but this process is what defines a language and allows us to create its syntax rules.

If you are curious about the list of rules that PHP uses, you can see them yourself at http://bit.ly/zend-language-parser. Besides testing if your PHP script adheres to the syntax rules, other checks are added to make sure the syntax is not only valid, but also makes sense: defining `public abstract final final private class foo() {}` may be syntactically correct, but does not make any sense PHP-wise.

Both tokenizing and parsing are tricky businesses, and often third-party applications are used to actually do this. In many cases, tools like `flex` and `bison` are used (PHP uses these tools, too). These tools can be seen as transcompilers also: they convert our rules into actual C source code that automatically gets compiled when you compile PHP.

Parsers and tokenizers are also useful in other areas. For instance, they are used for parsing SQL statements in database systems, and there are even many different parsers and tokenizers written in PHP as well. The Doctrine object-relational mapper has its own parser for DQL statements and a "transcompiler" for converting DQL into SQL, and many template engines, including Twig, use their own tokenizer and parser to "compile" template files back to PHP scripts. (In a sense, these template engines are transcompilers, too!)

Abstract Syntax Trees

When we have tokenized and parsed our language, we can generate our bytecode. Up to version 5.6 of PHP, this is exactly what happened: the bytecode was directly generated in the parser phase. However, it's more common to add another step in the process: instead of generating the actual bytecode, the parser generates something called an Abstract Syntax Tree, or AST. This is a treelike structure in which a whole program is built up in an abstract manner. An AST makes it easier to generate bytecode, but another benefit is that before converting to bytecode, we can actually make changes inside the AST since the tree is always generated in a specific way. A node in the tree that represents an "if" statement always has tree elements underneath: the first element is the condition (like `$a == true`), the second element is the statements that need to be executed when the condition is `true`, and the third element holds the statements that will be executed when the condition is `false` (the `else` clause). Even if there is no else clause, there are always three elements, except the `else` element is empty.

This way, we can "rewrite" the program before it's converted to bytecode. Sometimes this could be used for optimizing the code. If we could detect that a developer was recalculating a variable over and over again inside a for-loop, and we knew that this variable would always be the same, an optimizer could rewrite the AST to create a temporary variable so we didn't need to do this recalculation every time. An AST could be used to rearrange code a bit to make it run faster, it could remove unneeded variables, etc. This is not always possible to do, but when we have an AST of the complete program, it makes it much easier to create such checks and optimizations.

Inside an AST, we can check to see if variables are declared before usage, or to see if we use assignments in conditional block (`if ($a = 1) {}`) and emit warnings when we encounter code structures that might be error-prone. It's even possible to analyze code for security issues and warn users as soon as they execute the script. This kind of analysis is called static analysis; it opens the opportunity for creation of new features, optimizations, and validation systems that will help developers in writing solid, secure, and faster code.

With PHP 7.0, a new parsing engine has been introduced (Zend engine 3.0) that also generates an AST during the parsing phase. It's still relatively new, so we cannot do a lot with it yet, but the fact that it's there means we can expect all kinds of new features in the near future. In fact, the `token_get_all()` function accepts a new, undocumented `TOKEN_PARSE` constant that might be used in the future to return not only tokens, but the parsed AST as well. Third-party extensions like the `php-ast` extension already allow us to view and edit the AST directly in PHP. This complete rewrite of the Zend engine and implementation of the AST will open PHP up for so much more potential in the near future!

JIT

Have we found the sweet spot between interpreting and compiling with virtual machines running highly optimized bytecode generated from ASTs? Well, there is another technique that is gaining more ground, but it is one of the most complex systems that we can implement.

Think of how applications are executed: a lot of time is spent in setting up your application (for instance, a framework needs booting, routes need to be parsed, environment variables need to be processed, filtered, sanitized, etc). Once that's completed, it usually isn't executed anymore. In fact, most of the time is spent on only a fraction of your application. What if we could identify the parts that will run often and be able to convert those small pieces of code (say, just a few methods) into binary code? Sure, it might take a relatively long time to do this compiling, but since we are only compiling a method instead of a whole application, this process is still quick. You might experience a small delay on the initial call to a function, but all other calls will be blazingly fast, skipping the virtual machine altogether, and will run directly as binary code.

We get the speed of compiled code, while enjoying the benefit of interpreted code. It turns out that such systems can outperform regular interpreted bytecode, sometimes even by a large number.

This is exactly what a JIT compiler does. JIT stands for *just-in-time*, and the name fits perfectly. The system detects which parts of the bytecode might be a good candidate for compilation to binary code, and does this as soon as that code needs to be executed, i.e., just in time. Our program can start immediately, and there is no need to wait for compilation. Only the most efficient parts of your code are converted to binary code, making the compilation process fast and automatic.

Not all JIT compilers work the same way, though: some compile all methods on the fly, some will only try to detect which functions need to be compiled in an early stage, and some, for instance, will only compile functions when they are called two or more times, but all JITS are based on the same premise: compile small parts, only when really needed.

Another advantage of JIT compiling in contrast to the more regular compilation is the fact that the JIT compiler is able to make better guesses and optimizations based on the current state of the application: it can analyze code dynamically at runtime, and make guesses that a regular compiler cannot do. This is because information about the current state of these programs is missing when we are in the compilation phase, while a JIT compiler compiles during the execution phase.

If you've ever worked with HHVM, you've already used a JIT compiler: PHP code (and the superset language Hack) are converted into bytecode that is run on the HHVM virtual machine. This machine identifies blocks that can be (safely) converted to binary code, does so if not done yet, and runs it. When done running, the virtual machine will continue with the next bytecodes, which may or may not be converted to binary code.

PHP 7 does not run on a JIT, but its new system outperforms any previous release. Right now, all the pieces to actually experiment with static analysis, dynamic/runtime optimization, and even simple JIT systems are in place, so maybe one day we can even outperform PHP 7!

Joshua Thijssen is a freelance consultant and trainer. His daily work consists of maintaining code bases, working on different projects, and helping other to achieve higher standards in both coding and thinking. He is author of the php[architect] book Mastering the SPL library and the Symfony Rainbow Series, founder of the Dutch Web Alliance, and regular speaker at national and international conferences. @JayTaph

Getting Started with Zend Expressive

Chris Tankersley

Many frameworks are now starting to adopt PSR-7 and the ideas of middleware. Middleware helps move some common code out of our business logic and into reusable objects that can be injected into a request. Zend Expressive is one of the new micro frameworks that are built totally around PSR-7, and is designed fully around middleware as a first-class citizen.

Way, way back in the dark and ancient times of late 2011 and early 2012, PHP was changing in a way that was a shift not only back to the pre-framework days, but also heralding a new direction from the massive monolithic frameworks of the time. Ed Finkler published the *MicroPHP Manifesto*[1], which railed against the monolithic frameworks and their bulk. It contained these core tenets:

- I am a PHP Developer
- I like building small things
- I want less code, not more
- I like simple, readable code

I was fully behind this drive. I loved using the monolithic frameworks for my day-to-day job because it provided a nice, structured layout and design principle for my coworkers and me. I did not have to explain why things were in places, and I did not have to maintain my own custom framework anymore. It was a nice departure from the old "We run our own custom framework that was cobbled together from various Internet snippets."

I did not need these giant frameworks for small projects. I wanted less code, not more code. The more code in the framework, the more code I had to depend on. The more code, the more complex the system. Ain't nobody got time for that, especially when you are not getting paid.

Composer changed all of that. Small packages that excelled in their area of duty were incredibly easy to install now. With the advent of The PHP Framework Interoperability group's passing of PSR-0 for autoloading, we got rid of massive `require` files. With PSR-3 for logging we started to see the emergence of standardized interfaces for generic systems. PSR-6 for caching and PSR-7 for HTTP messaging provides us with the ability to swap out these components at a moment's notice. If your application supports talking to a PSR-7 compliant package, it does not matter who it is written by.

That leads us to today. In August 2015, Zend released a new PSR-7 middleware framework that, in the vein of the *MicroPHP Manifesto* from 2012, allows you to write smaller applications with less code, and allows you to swap out many of the components for whatever your needs are. At its heart, Zend Expressive[2] is a wrapper for routing, dependency injection, templating, and error handling. You supply all of those pieces and Zend Expressive will make it all work using common interfaces.

It's Like a Micro Framework

Zend Expressive installs just like any package now would—through Composer. You can add it to any existing project or start a new one by just requiring at least three different packages: Zend Expressive itself, a compliant router, and a compliant dependency injection system. Let's install just the basics to get going:

```
composer require zendframework/zend-expressive \
   zendframework/zend-expressive-fastroute \
   zendframework/zend-servicemanager
```

I am going to use the FastRoute package for routing, and the Zend ServiceManager for dependency injection. Zend Expressive supports the following different packages:

1. Routing
   - Aura.Router `zendframework/zend-expressive-aurarouter`
   - FastRoute `zendframework/zend-expressive-fastroute`
   - Zend Framework 2 MVC Router `zendframework/zend-expressive-zendrouter`
2. Dependency Injection
   - Aura.Di `aura/di:3.0.*@beta`
   - Pimple `xtreamwayz/pimple-container-interop`
   - Zend Framework Service Manager 2.x or 3.x `zendframework/zend-servicemanager`
   - Anything that supports `container-interop`
3. Templating
   - Plates `zendframework/zend-expressive-platesrenderer`
   - Twig `zendframework/zend-expressive-twigrenderer`
   - Zend Framework 2 PhpRenderer `zendframework/zend-expressive-zendviewrenderer`

Your choice of exactly what packages you use will be determined by your own needs. I tend to use the FastRoute router for no particular reason other than it is pretty fast, the Zend Framework Service Manager because I'm familiar with it, and Twig because I've been using Twig for years. Don't like Service Manager and want to use Pimple? Awesome! Use it instead.

1 *The MicroPHP Manifesto:* http://microphp.org

2 *Zend Expressive Docs:* http://zend-expressive.readthedocs.org/en/latest/

With at least the core, a router, and a dependency injector, we can start to build an application just like any other micro framework.

```
use Zend\Expressive\AppFactory;
require 'vendor/autoload.php';

$app = AppFactory::create();
$app->route('/', function ($request, $response, $next) {
   $response->getBody()->write('Hello php[architect]!');
   return $response;
});
$app->run();
```

If you have used Slim or Silex, this will all look really, really familiar. We create an application, add a route, and run the application. Zend Expressive can then take in a request, match it to one of the routes that have been registered, and determine a response. It is not terribly exciting, but hey, there's more work to be done.

Organizing through Dependency Injection

Anyone who has written a small application in a micro framework quickly learns that keeping all of your routes and code in a single `index.php` file is not maintainable. Each one has its own way of handling how a project scales, and so does Zend Expressive. This is done almost all through configuration and the dependency injection layer. There are a few main sections of the configuration that we can take a look at.

You can organize your configuration any way you like. The Zend Expressive Skeleton[3] project splits configuration in multiple files, but you can easily just use one.

Making Objects—Invokables and Factories

The first section we need to understand is the `dependencies` key. This section will contain a list of Invokable classes and Factory classes that will be used to help create objects. If you are familiar with Zend Framework 2, these two terms should look familiar and you will need to decide which section to use for your objects. A sample of this configuration is in Listing 1.

LISTING 1

```
01. return [
02.    'dependencies' => [
03.       'invokables' => [
04.          MyApp\String\Slugger::class => MyApp\String\Slugger::class,
05.       ],
06.       'factories' => [
07.          MyApp\Model\Users::class => MyApp\Model\UsersFactory::class,
08.          MyApp\Controller\IndexController::class  =>
09.             MyApp\Controller\IndexControllerFactory::class,
10.          Zend\Expressive\Application::class =>
11.             'Zend\Expressive\Container\ApplicationFactory',
12.       ]
13.    ]
14. ];
```

Each section is an array of key/value pairs, with the key being the name of the "service" that you want to be able to find, and the value is the name of the class that will be used to instantiate the object. Most of the time you will be using the name of the class you want as the "service," such as `MyApp\String\Slugger`, so using the class name is most appropriate. If you are using PHP 5.5 or higher you can use the method I use in Listing 1, where you use `::class` static property; otherwise, just use the class name as a string.

This section provides two benefits. One is that we let the dependency injection layer handle how to create the objects so your application does not need to bother with it. The second is that there is now a catalog, or Service Locator, which we can pluck objects out of. All of this is kept in one single area, reducing the spread of code.

Invokable classes are classes that do not need any sort of dependency injected, either through constructor injection (passing dependencies in through `__construct()`) or through Setter methods. If you can do `$foo = new Foo()` and have a fully usable class, your class is considered an Invokable. We let the dependency injection layer know about these classes so we can keep track of then for Service Location. Your key and value in the config will most likely be the name of the class.

Factory classes are classes that need some help getting set up, either by needing data passed into them through their constructor or through Setter methods. If we have a class which depends on a database adapter, we need to create a factory to handle the creation. In the configuration we will set the service we want to locate, say, `MyApp\Model\Users`, as the key, and the thing that helps make it, `MyApp\Model\UsersFactory`, as the object we want to instantiate. The `UsersFactory` will create the `Users` object for us.

All the objects your application needs, be it a database model, a controller, a form, any object you create, should be pushed through the Dependency Injection layer.

Factories

Many, many objects have dependencies, so you will need to get comfortable writing factories. This looks like a lot of work, but it really is not. Factories will help reduce bad coupling and practices like Service Locator injection and give you a cleaner object interface.

For Zend Expressive, a Factory is any class that implements `__invoke()` and takes the Service Locator, generally denoted as `$container`, as a parameter. You then use the `$container` to look up your dependencies and do anything you need to do to create your object.

For example, let us say that `MyApp\Model\Users` requires us to pass in a database adapter through the constructor, since it talks to the database. We will use a factory, `MyApp\Model\UsersFactory`, that takes in the service locator, finds the database adapter, and creates a Users object with it.

```
use Interop\Container\ContainerInterface;
class UsersFactory
{
   public function __invoke(ContainerInterface $container) {
      $db = $container->get('Zend\Db\Adapter\Adapter');
      return new Users($db);
   }
}
```

The above code is a fully working factory. There does not need to be a lot of code here for your factory, just whatever is needed to create your object.

Routing and Controllers

Now that we can create objects, we need to link them up to some URI. When the user visits `/`, what code should fire? What about `/profile`? This is where the routing configuration comes into play.

3 Zend Expressive Skeleton App: http://phpa.me/zend-expressive-skel

The first thing we need is a controller. Much like the factory, a controller is a plain old PHP object that implements `__invoke()` and takes in three parameters—the server request, the pending response, and a thing to call next. The controller will modify the request and go on from there. The next block of code shows a very basic controller that just replicates what our little inline route did at the beginning of the article.

```
class IndexController
{
   public function __invoke($req, $resp, $out = null) {
      $resp->getBody()->write('Hello php[architect]!');
      return $response;
   }
}
```

By default, each controller will only handle a single route. There are ways around that, though, by using the `__invoke()` method to check the route being called. You can read more about that online in the official documentation. For now we'll just handle one route per controller.

```
return [
   'routes' => [[
      'name' => 'homepage',
      'path' => '/',
      'middleware' => 'MyApp\Controller\IndexController',
      'allowed_methods' => ['GET'],
]]];
```

As you can see, each route is just an array with four keys: `name`, `path`, `middleware`, and `allowed_methods`. The newcomer here is `middleware`. This is the class that the router system will invoke to help generate a response. It uses the Dependency Injection layer, so make sure all of your middleware controllers are registered as an invokable or a factory.

That's all routing is—just a collection of arrays that point to a controller.

Middleware

Middleware, traditionally, is software that extends an existing suite of software in a somewhat pluggable way. They will take some point of data, pass it through a list of registered middlewares that transform or work with the data, and then pass it along to the application. For PHP applications, we generally use the term *middleware* to talk about processes that need to interact with an HTTP request before or after our main logic has finished working.

Middleware helps take some of the routine application logic out of the business logic of an application. A good example of middleware is an authorization layer. You can have middleware run after routing that checks to see if your user has access to a certain route. If not, change the response to redirect them somewhere else. Your business logic does not need to worry about authorization anymore.

Writing an API? Write middleware that checks the token to make sure it's valid before you even process routing. No token, no access.

Zend Expressive is built on the idea of using middleware to work with a request and formulate a response. At the most basic level, a request will run through a route middleware (as seen earlier) that generates a response. You throw in different middleware either globally or on a route and you start to build up your application.

As with everything else, we define this in configuration through the use of a `middleware_pipeline` config key. The below code shows a basic middleware declaration for a Session middleware, which sets up a PHP session and makes it available to the rest of the system.

```
return [
   'middleware_pipeline' => [
      'always' => [
         'middleware' => ['SessionMiddleware'],
         'priority' => 100000,
]]];
```

In Zend Expressive, you have two types of middleware. One is Pipe middleware, which runs at various times during the application lifecycle to do things like authorization, error handling, handling sessions, etc. The other is Route middleware, which is used when you need to react to a specific route that is being called. The routes we defined above set up the latter type, where our `SessionMiddleware` sets up a pipe-style middleware.

Any sort of pipe middleware will need to go into the `middleware_pipeline` key, not in `routes`.

Pipe middleware works exactly like the Controllers we created earlier. It just needs to implement `__invoke()` and take in the request, response, and next callable. Listing 2 shows a pipe middleware that injects a session into the request.

Templating

If you need to render templates in your application, we have one final configuration to look at, which is `templating`. Setting up templating is actually two parts. The first is initializing a middleware that turns on templating, and then setting up which template system to use. Zend Expressive supports Zend Framework's PhpRenderer, Twig, and Plates, so we need to tell it which one to use.

We do this by adding in two entries into our `factories` array. Add the following two snippets of code to the `factories` section of your config.

```
'Zend\Expressive\Template\TemplateRendererInterface'
   => 'Zend\Expressive\Twig\TwigRendererFactory',
'Zend\Expressive\FinalHandler' =>
   'Zend\Expressive\Container\TemplatedErrorHandlerFactory',
```

LISTING 2

```
01. <?php
02. // Registered as a Factory for dependency injection,
03. // which passes in the session object
04.
05. class SessionMiddleware
06. {
07.    protected $sessionContainer;
08.
09.    public function __construct($sessionContainer) {
10.       $this->sessionContainer = $sessionContainer;
11.    }
12.
13.    public function __invoke($req, $resp, $next = null) {
14.       $req = $req->withAttribute('session', $this->sessionContainer->getSession());
15.
16.       if ($next) {
17.          return $next($req, $resp);
18.       }
19.
20.       return $response;
21.    }
22. }
```

We first need to register a `TemplateRendererInterface` with Zend Expressive. This is the templating system that we want to use. I'm using Twig for our example, so we set the `TemplateRendererInterface` key to the class that will wrap Twig. The template system will invoke the template system we supply from here on out.

The second key, `'Zend\Expressive\FinalHandler'`, is a key that tells Zend Expressive to render errors in the template system. This key is not needed, but stuff like 404s will not be passed through to the template system without it. Notice that this service is surrounded by quotes, as `'Zend\Expressive\FinalHandler'` is not a real class—it is a service that is just looked up.

Once we have enabled templates, we need to configure it a bit. We need to tell the template system what extension our templates use and where they live. Figure 1 shows the directory structure.

```
'templates' => [
    'extension' => 'twig',
    'paths' => [
        '__main__' => ['views', 'views/layouts'],
        'account' => ['views/account'],
        'error' => ['views/error'],
        'index' => ['views/index'],
]];
```

With this configuration in mind, we can now start to render HTML! Well, almost. Everything in Zend Express is pretty well decoupled, so we need to inject a template renderer into our controllers. In our `IndexControllerFactory`, we will grab the `TemplateRendererInterface` from the service locator and pass it into the `IndexController`. We can then use that to render a template and return an HTML response. The following code shows the new factory, and Listing 3 shows our controller with code to render the layout.

```
class IndexControllerFactory
{
    public function __invoke(ContainerInterface $container) {
        $template = $container->get(
            'Zend\Expressive\Template\TemplateRendererInterface'
            )
        return new IndexController($template);
    }
}
```

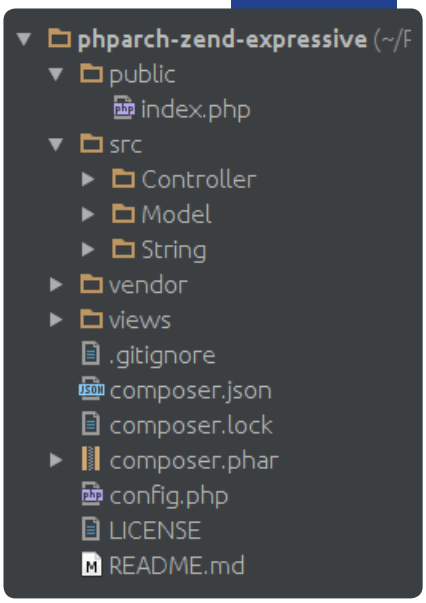
LISTING 3

```
01. <?php
02. use Zend\Diactoros\Response\HtmlResponse;
03.
04. class IndexController
05. {
06.     protected $template;
07.
08.     public function __construct($template) {
09.         $this->template = $template;
10.     }
11.
12.     public function __invoke($req, $resp, $out = null) {
13.         return new HtmlResponse($this->template->render('index::homepage'));
14.     }
15. }
```

The `render()` function takes two parameters. The first is the template we want to render, which is in a `[path]::[filename]` format. Remember the `paths` we set up in our configuration? Those are used in the part before the `::`. The second is the name of the template without the extension. So we will look for a template located at `views/index/homepage.twig`.

The second parameter is an array of variables to pass into the template. These variables will then be available for you to work with in the template itself.

FIGURE 1



Putting it All Together

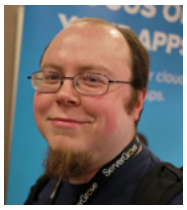
So we now have a set of configuration, some controllers, and some templates. How do we put it together? Assuming the file structure of Figure 1, Listing 4 shows a simple bootstrap that will read in a single config file and execute the application, with `index.php` living in the `public/` folder.

There are various things you can do from here. For example, split out the configuration into multiple files like the Zend Expressive skeleton app. Add in more routes and controllers. Almost all of your additions will just need to be done in the configuration; there is very little you will need to do in this `index.php` bootstrap file. Start to mix and match the various types of components that Zend Expressive supports, and see what you can build.

LISTING 4

```
01. <?php
02. chdir(dirname(__DIR__));
03. require 'vendor/autoload.php';
04.
05. use Zend\ServiceManager\Config;
06. use Zend\ServiceManager\ServiceManager;
07.
08. $conf = require 'config.php';
09. $serviceManagerConfig = new Config($conf['dependencies']);
10. $container = new ServiceManager($serviceManagerConfig);
11. $container->setService('config', $conf);
12.
13. $app = $container->get('Zend\Expressive\Application');
14. $app->run();
```

If you are interested in working more with Zend Expressive, check out the Zend Expressive Skeleton App, or the demo app[4] from which most of the code here was taken.

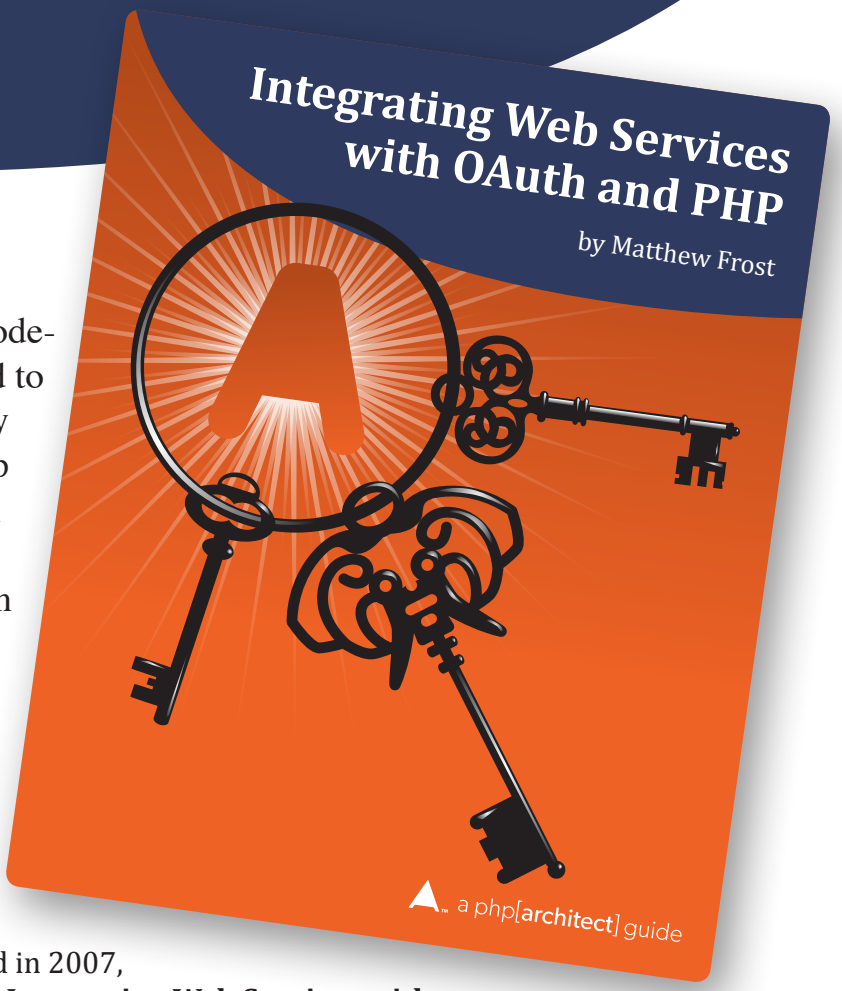


Chris Tankersley is a husband, father, and PHP developer in Northwest Ohio. He works as a programming consultant for ZF2, Drupal, WordPress, and Symfony projects. He founded the Northwest Ohio PHP User Group, and helps local developers with programming and server administration. He works with PHP primarily, with some work done in Node.js and Python for personal projects. @dragonmantank

4 *Demo Code:* https://github.com/dragonmantank/phparch-zend-expressive

Integrating Web Services with OAuth and PHP

by Matthew Frost



Modern web applications are no longer standalone, monolithic codebases. Instead, they are expected to integrate with external, 3rd party applications to allow users to tap into new features, integrate with their social networks, and to easily migrate their data between systems. Many services afford these integrations by building web services that use the OAuth standard to authenticate users and allow "secure delegated access" on their behalf.

There are two versions of OAuth. Version 1.0 as introduced introduced in 2007, and OAuth 2.0 was released in 2012. **Integrating Web Services with OAuth and PHP** describes the differences between the two versions, explains the jargon associated with each, and–most importantly–provides working PHP examples for integrating with popular web services such as Twitter, Tumblr, Instagram, and others. This book also includes a primer on the HTTP protocol, highlights open-source resources for OAuth clients and servers, and discusses issues with OAuth and application security.

Written by PHP professional Matt Frost, this book is an indispensable resource for any PHP developer that builds or integrates with online applications.

Purchase

http://phpa.me/oauthbook

Modern PHP

Cal Evans

I've been thinking a lot lately about the topic of *Modern PHP*. (Those of you who know me probably know why.) It is an interesting topic these days and a very fluid one as well.

I remember when I first started programming in PHP. Every page was self-contained. Most of us had a library of functions that we had developed over the years that we included in each page, but for the most part, everything you needed for the page was on the page. Forms called themselves, and at the top of each form was a piece of PHP code to check to see if the form was submitting to itself. If so, you processed the info passed in and then usually redirected to another page.

Speaking of pages, there were really only three types:

- Static page
- List page
- Form page

Almost every page in any application was one of those three.

Then we got objects. Not totally, but PHP could be "object-based." We started building objects instead of just procedural code, but since PHP 4's object model was … well, let's be generous and say flawed, objects were largely just containers for procedure code and data.

Buried deep in the bowels of the Internet—but not deep enough to keep Elizabeth Naramore from finding it and using it in my bio as part of the infamous CodeWorks '08 tour—is a framework I wrote around the concept that any page in an application should be an object. I didn't stray too far from the "every page is self-contained" model; each page defined its own object, instantiated it, and then called `run()` or `process()` depending on what data was being passed it. (Way to think outside of the box while keeping firmly pressed against it.)

With PHP 5.2 came an explosion of PHP frameworks. PHP 5.3-5.6 gave framework authors even more tools to work with, and we saw not only the first version of most frameworks, but also v2 and beyond. Depending on your skill level and the framework you chose, coding PHP got infinitely easier. You had an explosion of options and tools to play with.

These days, many frameworks are pulling back. Microframeworks are the rage and, in many cases, no framework at all is the proper answer; instead, we use Composer to pull in just the components we need.

The way we code PHP has evolved over the years, but Modern PHP is more than just features and coding style. Modern PHP has as much to do with the community as it does with the language itself. The PHP community has grown, matured, and self-organized. Whereas, it used to be that there were few canonical sources of information that a PHP developer had to keep up with—like this very magazine—now there are a multitude of blogs, podcasts, books, screencasts, and other learning resources available. User groups are popping up like weeds these days, scattered all over the globe, and there is even a thriving community of developers who either don't have or can't participate in a local PHP user group.

As we have grown, as we have matured, we have refined our goals as a community. Even from its early days as just a few people hanging out on IRC, the PHP community has always been about giving, helping, and paying it forward. We take our cue from the countless developers who have contributed to the core, and from the countless developers who maintain our extensive manual. We have been shown that to succeed, we all have to help each other.

These days the hashtag #communityWorks can be seen on Twitter. Every time we want to celebrate someone being helped by the community, we tag the post #communityWorks. Additionally, at many conferences, you will see speakers talking about the wonderful "pay it forward" attitude of the community as people write blogs, speak at conferences, mentor others, and share what they know—what they have been taught by others—to help repay the debt they owe.

Modern PHP is about building things—cool things, things that solve problems for others. We do that with code, and we do that by sharing. Move into the modern era, step up, and share something you know. Be a part of Modern PHP.

Past Events

SunshinePHP 2016
February 4–6, Miami, FL
http://sunshinephp.com

Joomla Day UK
February 13, London, Victoria, UK
http://www.joomla-day.uk

PHP UK
February 18–19, London, UK
http://phpconference.co.uk

DrupalCon Asia
February 18–21, Mumbai, India
https://events.drupal.org/asia2016

ConFoo 2016
February 24–26, Montreal, Canada
http://confoo.ca

Upcoming Events

March

Drupalcamp London 2016
March 4–6, London, U.K.
http://drupalcamp.london

Midwest PHP 2016
March 4–5, Minneapolis, MN
http://2016.midwestphp.org

PHP Darkmira Tour
March 18–20, Brasilia, Brazil
https://br.darkmiratour.com

April

Lone Star PHP
April 7–9, Dallas, TX
http://lonestarphp.com

SymfonyLive Cologne 2016
April 27–29, Cologne, Germany
http://cologne2016.live.symfony.com

May

New Orleans DrupalCon
May 9–13, New Orleans, LA
https://events.drupal.org/neworleans2016

phpDay 2016
May 12–14, Verona, Italy
http://2016.phpday.it

PHPKonf
May 21–22, Istanbul, Turkey
http://phpkonf.org

php[tek]
May 23–27, St. Louis, MO
http://tek.phparch.com

CakeFest 2016
May 26-29, Amsterdam, Netherlands
http://cakefest.org

International PHP Conference 2016
May 29–June 2, Berlin, Germany
https://phpconference.com

June

PHP South Coast 2016
June 10-11, Portshmouth, UK
https://cfp.phpsouthcoast.co.uk

Dutch PHP Conference
June 23–25, Amsterdam, The Netherlands
http://www.phpconference.nl

July

php[cruise]
July 17–24, Baltimore, MD (leaving from)
https://cruise.phparch.com

LaraCon US
July 27–29, Louisville, KY
http://laracon.us

August

Northeast PHP
August 4–5, Charlottetown, Prince Edward Island, Canada
http://2016.northeastphp.org

October

LoopConf
October 5–7, Ft. Lauderdale, FL
https://loopconf.com

These days, when not working with PHP, Cal can be found working on a variety of projects like Nomad PHP. He speaks at conferences around the world on topics ranging from technical talks to motivational talks for developers @calevans.

Things We Sponsor

Developers Hangout
Listen to developers discuss topics about coding and all that comes with it.
www.developershangout.io

NEPHP
NorthEast PHP & UX Conference. August 11-12, 2016
2016.northeastphp.org

NomadPHP
Start a habit of Continuous Learning. Check out the talks lined up for this month.
nomadphp.com

MidwestPHP
Take a break from the cold to enjoy some sunshine and talk about PHP!
2016.midwestphp.org

DC PHP
The PHP user group for the DC Metropolitan area
meetup.com/DC-PHP

FredWebTech
The Frederick Web Technology Group
meetup.com/FredWebTech

Hunting Mutants with Humbug

Matthew Setter

Welcome back to another month of Education Station. This month, I want to take another dive into testing, specifically looking at a way of validating and verifying the tests which we write. That might seem like a funny thing to say. And perhaps it's just because I'm not as experienced at testing as you, or those in the community, are.

But while I love the rigor of testing, how it improves the quality, and reliability of the applications I'm developing (or helping to develop), I often feel a certain shadow hanging over me. It's the shadow of uncertainty—an uncertainty that the tests I'm writing are truly covering, validating, and testing the code properly, and in its entirety.

Are My Tests Really Working?

You think your world is safe? It is an illusion, a comforting lie told to protect you.

That's right, are my tests really working? You see, I don't know about you, but, perhaps ironically, the better I feel I get at testing, the more the question nags at me, seeking an answer. Sure, the coverage level rises ever higher with each application, with each line of code written.

But does that metric, the higher it goes, truly correlate with quality and certainty? Perhaps it does and I'm just worrying over nothing. But what if it doesn't? What if we're resting on false foundations of certainty? What if they are an illusion?

Now that's all a bit dramatic, I'll admit. But then, I like a bit of drama, and I'm a bit prone to the dramatic. But it's a fair perspective to consider, if only for a moment. What if there are paths through our code which aren't obvious to see, ones we've missed, which haven't been verified?

What if, as a result of not testing for them, we leave security vulnerabilities in our code, memory leaks, and so on? But the question is, though, if we don't know about them, how do we find them? Good question. But it's one I have an answer for already. An answer which is the subject of this month's column. It's called Mutation Testing.

What is Mutation Testing?

The idea was proposed as far back as 1971, by Richard Lipton, with the first implementation being done by Timothy Budd. Unfortunately, while groundbreaking in its scope, mutation testing was considered too computationally expensive to be practically implemented.

Fast-forward to today however, and modern computing power has largely caught up. As a result, mutation testing has seen a resurgence, with implementations for a host of functional and object-oriented languages, along with non-procedural languages, including C, PHP, Ruby, Java, C#, Python, and Haskell.

Here's a description which I pulled from Wikipedia, one which is fairly apt:

> *Mutation testing (or Mutation analysis or Program mutation) is used to design new software tests and evaluate the quality of existing software tests. Mutation testing involves modifying a program in small ways. Each mutated version is called a mutant and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant.*
>
> *This is called killing the mutant. Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants. Mutants are based on well-defined mutation operators that either mimic typical programming errors (such as using the wrong operator or variable name) or force the creation of valuable tests (such as dividing each expression by zero).*
>
> *The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution.*

Introducing Humbug

And that's how you find bugs which you didn't know existed. But how do we use it in PHP, and test our code there? Thankfully, Padraic Brady has written the de facto implementation, called Humbug[1]. It's the subject of this month's column.

Here's a short version of how it works, gently paraphrasing the Humbug documentation:

Humbug injects small defects into your source code and checks if the unit tests noticed. If they do, your unit tests have "killed" the mutation. If not, the mutation has escaped undetected. As unit tests are intended to prevent regressions, having a real regression pass unnoticed would be a bad thing!

More specifically, Humbug scans your code and determines if a PHP token can be mutated, or changed. It makes a mutation based on its standard set of mutators, which you can find at https://github.com/padraic/humbug#mutators.

For example, it could mutate a `true` to be `false`, a logical "and" (`&&`) to be a logical "or" (`||`) and so on. It does these one at a time when it believes it should and tests the results. Given that, and the complexity of performing mutation testing, the time taken to run tests is likely to be noticeably larger than your standard set of unit tests. But let's not let that put us off.

1 *Humbug:* https://github.com/padraic/humbug

Install

Like any good PHP library, there are multiple ways to install it. We can download a Phar file, we can install it from source, or we can install it via Composer. My preferred method is, naturally, Composer. Given that, run the following in your project directory to install it:

```
composer require humbug/humbug
```

If you don't have a `composer.json` file, one will be created, with Humbug as its sole dependency. Otherwise it will be added. After that, the project's dependencies will be updated, and you'll be ready to go. One thing to note, though: when you run Humbug, you have to have Xdebug[2] installed and enabled. Otherwise you'll encounter the error:

You need to install and enable Xdebug in order to allow for code coverage generation.

Setup

Once you've installed Humbug, you need to configure it. You could create the configuration file yourself, but Humbug comes with a command line switch to do it all for you. From the terminal, run:

```
bin/humbug --configure
```

This will guide you through the process, asking six questions. These are:

- What source directories do you want to include
- Any directories to exclude from within your source directories
- Single test suite timeout in seconds (defaults to 10)
- Where do you want to store the text log (defaults to `humbuglog.txt`)
- Where do you want to store the json log
- Generate `humbug.json.dist`

For the first two questions, you can add as many directories as you like, and the command will continue to ask you until you don't provide a directory name. For the remaining questions, it's simplest to accept the defaults. You can see sample output in Figure 1.

The one thing I did after config was finished running was to rename `humbug.json.dist` to `humbug.json`. The reason being, editing it later in PhpStorm was simpler, as it would use the JSON plugin and ensure that syntax and indentation were correct. Don't feel obliged to rename yours, though. With that, we're ready to go and hunt some mutants.

Running It–Hunting Mutants

To run it, as with configuring it, from your terminal run `bin/humbug`, no extra arguments are required. What will happen are the following two steps:

- First, your PHPUnit tests will be run so that logs and code coverage data can be generated
- Then, Humbug will run and make some changes to your code and check to see if your tests noticed

When Humbug's finished running, it will report mutations across five categories. These are:

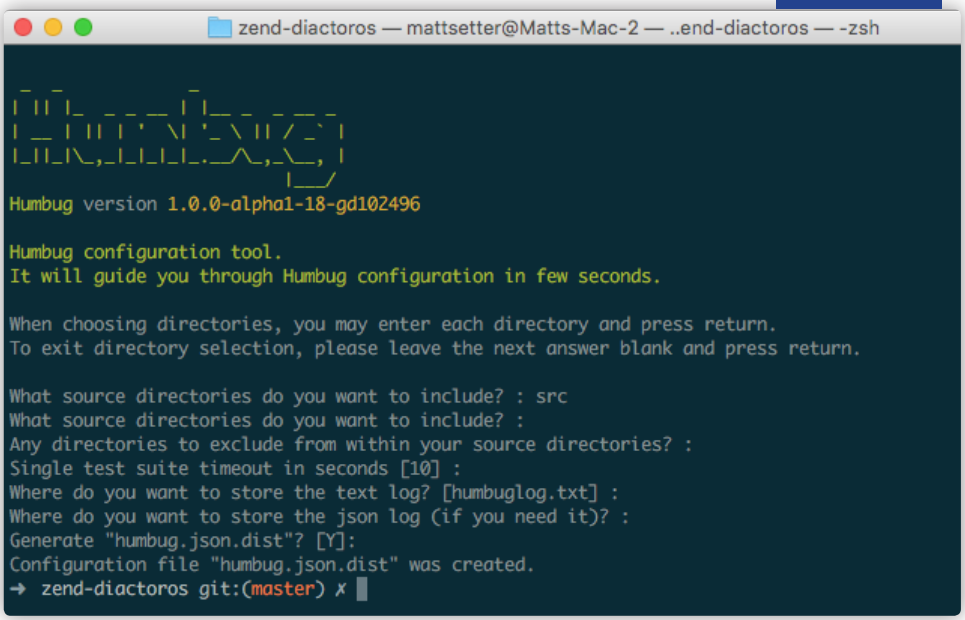
- **Killed Mutation (.):** A mutation that caused unit tests to fail which is a positive outcome.
- **Escaped Mutation (M):** A mutation where the unit tests still passed, which is not what we want! Our unit tests should detect any behavior changes.
- **Uncovered Mutation (S):** A mutation which occurs on a line not covered by any unit test. Since there are no unit tests, this is another undesirable result.
- **Fatal Error (E):** A mutation created a fatal error. Usually a positive result since it's obviously going to be noticed. In some cases, however, it might be a Humbug problem that needs fixing.
- **Timeout (T):** This is where unit tests exceed the allowed timeout configured for Humbug. Likely a positive result if your timeout is appropriate, and often occurs when a mutation ends up creating an infinite loop.

You can see in Figure 2, which shows the result of running Humbug on the latest clone of the Piston framework, that of the 48 tests, two mutants were killed, 33 weren't covered by tests, and one covered mutant was not detected. Now this is, partly, to be expected, as Humbug mainly works with PHPUnit tests, and the majority of Piston is covered by PHPSpec tests.

The reason why it has so many uncovered mutations is because the repository is primarily covered by tests written with phpspec, not PHPUnit. And as we used the default test adapter, which is PHPUnit, it only referenced those tests. I'm OK, for now, with how high that percentage is. The Uncovered Mutation, on the other hand, was unexpected. Let's dig further and see what's going on there.

To find the specifics of the report, you have to look in the log file, `humbuglog.txt`, which you can see a sample of in Figure 3. Doing so revealed the following patch, produced from

FIGURE 1



2 *Xdebug:* https://xdebug.org/

altering what Humbug thought was a suspicious line code segment.

```
{
 -    if ($this->offset || $this->limit) {
    +    if ($this->offset && $this->limit) {
         return ['offset' => $this->offset,
                 'limit' => $this->limit];
     }

      return [];
    }
```

This patch was made to the following function, one I added to Piston:

```
public function getOffsetLimit()
{
   if ($this->offset || $this->limit) {
     return ['offset' => $this->offset, 'limit' => $this->limit];
   }

   return [];
}
```

The function returns an associative array, containing the value of offset and limit, if at least one of them has been set. Not too sophisticated a function, but enough to be a worthwhile example. To test that it would work correctly, I wrote three covering tests.

The first one is to test the case where neither offset nor limit had been initialized. The second and third tested setting and retrieving both in different ways and that the values of both could be successfully retrieved.

I thought I'd covered the function sufficiently. At least at the time of writing, I thought I'd gone far enough. Turns out, I might not have. So let's consider what Humbug uncovered.

What if, instead of using a logical "or," there was a logical "and"—would my tests have covered this scenario? Well, technically, no. Given that, I decided to add another test, one to test when only one of the two variables was set. Here's how I did it.

I first created a new function which would be invoked by PHPUnit's `@dataProvider` annotation. As you can see in Listing 1, for each invocation, it will return a value for one of the two variables, but `null` for the other.

LISTING 1

```
01. public function offsetOrLimitProvider()
02. {
03.     $faker = $this->getFaker();
04.
05.     return [
06.         [
07.             'offset' => $faker->numberBetween(0, 9000),
08.             'limit' => null,
09.         ],
10.         [
11.             'offset' => null,
12.             'limit' => $faker->numberBetween(1, 9000),
13.         ],
14.     ];
15. }
```

I then added a new test to use the data provider, which then instantiated a new object and called one of the methods to set the two variables. With the object instantiated, I then used the `assertSame` function to check that what was returned was correct. See Listing 2.

FIGURE 2

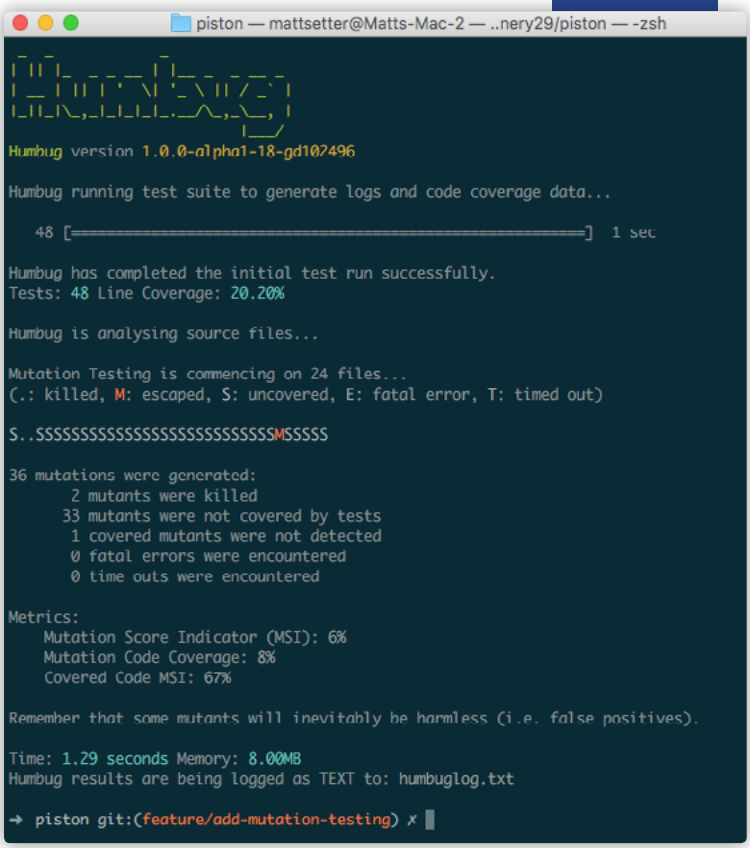
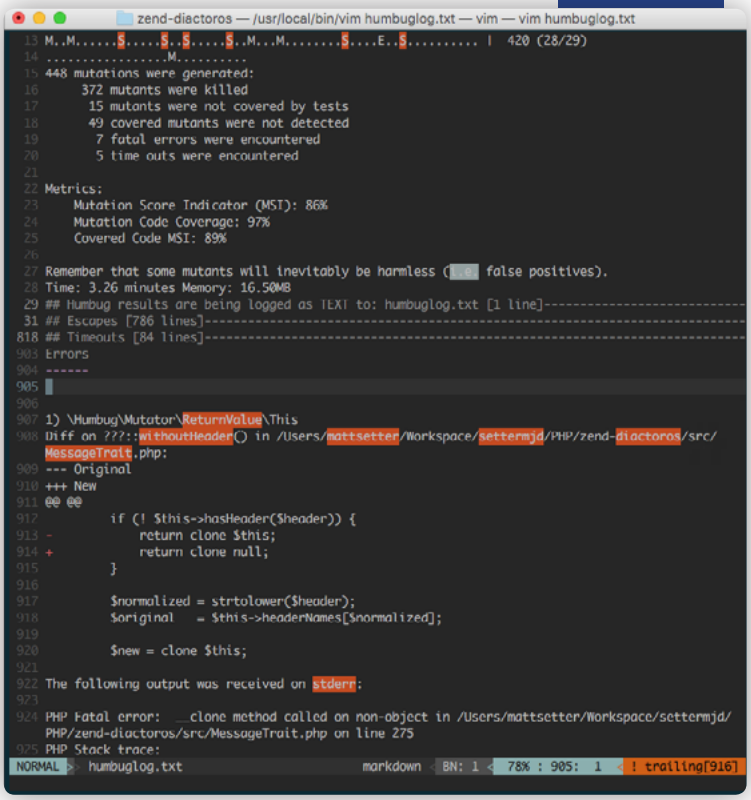


FIGURE 3



LISTING 2

```
01. /** @dataProvider offsetOrLimitProvider */
02. public function testCanSetEitherOffsetOrLimitAndReturnAResult($offset, $limit)
03. {
04.     $request = (new Request())->withOffsetLimit($offset, $limit);
05.
06.     $this->assertSame(
07.         [
08.             'offset' => $offset,
09.             'limit' => $limit
10.         ],
11.         $request->getOffsetLimit()
12.     );
13. }
```

With the test in place, I then ran Humbug again and found, as you can see below, that the test results were better:

```
Mutation Testing is commencing on 24 files...
(.: killed, M: escaped, S: uncovered, E: fatal error, T:
timed out)

S..SSSSSSSSSSSSSSSSSSSSSSSSS.SSSSS

36 mutations were generated:
       3 mutants were killed
      33 mutants were not covered by tests
       0 covered mutants were not detected
       0 fatal errors were encountered
       0 time outs were encountered

Metrics:
    Mutation Score Indicator (MSI): 8%
    Mutation Code Coverage: 8%
    Covered Code MSI: 100%

Remember that some mutants will inevitably be harmless
(i.e. false positives).

Time: 4.78 seconds Memory: 6.50MB
Humbug results are being logged as TEXT to: humbuglog.txt
```

With that extra test in place, the mutant was killed. But I just kept wondering what the problem was. To me it seemed like a false positive. I really didn't see what the problem was. But I kept on at it, thinking about it for some time. Eventually, I started to wonder if I was really understanding it all correctly. Given that, I got in touch with Padraic, seeking further input. Here's what he said:

It's likely what we'd call a "false positive." Humbug follows a particular strategy of not analysing the context of code, so it sometimes reads undetected mutations incorrectly. In this case, any OR condition which evaluates to TRUE, can also evaluate to TRUE if the mutated AND condition is also met by coincidence, i.e., both sides happen to be TRUE for that particular set of test input.

TL;DR: `TRUE || TRUE` *and a (mutated)* `TRUE && TRUE` *both lead to the code block executing as expected, passes your tests, and this makes Humbug suspicious. I'm assuming that both sides are evaluating as TRUE in this case—it could be something else entirely.*

As to why Humbug does this, it boils down to performance. Eradicating even a small proportion of possible false positives, by analysing code context, is prohibitively expensive and unreliable. The benefit is not worth the cost for the time being. A fast Humbug with false positives is worth more than a slow Humbug without…by a few orders of magnitude probably.

There is a Catch

While it's an excellent addition to helping to improve both your test coverage and test quality, it's definitely not a panacea, not a silver bullet. You can't just run it and take it for granted. "But why introduce a library like this?" you're probably asking. If it's not complete, is it worth using?

That's a good and fair question and one I want to raise with you, and not sweep it under the proverbial carpet. While it might seem like I've given an excellent introduction to mutation testing and Humbug, only to now say that it's not perfect, don't misunderstand my intent.

While mutation testing is still a relatively new addition to software testing, and there's still a lot of work yet to come before it's at the level of other aspects of the greater discipline, it's one worth getting on the bandwagon of now.

Getting started now and learning to determine whether your the results are false positives or not will, in itself, both educate you and improve your overall skill with testing.

You'll increasingly develop a more savvy eye for spotting what should and shouldn't be tested, for what paths you've either not tested properly, or completely overlooked.

Further Information

I've only recently come across mutation testing as I continue to build my knowledge and affinity with using test-driven development to create code to a high standard of quality. After adding mutation testing, I really feel that it has begun to help me out, rounding out the gray areas of my understanding.

However, as it's not quite as black-and-white as traditional unit testing, I don't want to leave you with nowhere to go. Given that, I've done my best to add a decent set of links for you to peruse as your time permits.

- The Humbug Project https://github.com/padraic/humbug
- Wikipedia Entry, http://bit.ly/wikipedia-mutation-testing
- Mutation Testing tutorial, http://bit.ly/mutation-testing-tutorialspoint
- PIT, http://pitest.org
- The Major Mutation Framework, http://mutation-testing.org
- What the Heck Is Mutation Testing? http://phpa.me/codeafffine-mutation
- Mutation Testing PDF by Stuart Anderson, http://phpa.me/anderson-mutation-pdf

Conclusion

I hope that you see value in mutation testing, and how it can help you improve your testing skills. I hope that it's one which has a bright future, one I hope to write about more in the future. I encourage you to add Humbug to your testing toolkit, experiment with it, and see how it helps you improve your applications. And until next month—happy testing.

Matthew Setter is a software developer specializing in PHP, Zend Framework, and JavaScript. He's also the host of http://FreeTheGeek.fm, the podcast about the business of freelancing as a software developer and technical writer, and editor of Master Zend Framework, dedicated to helping you become a Zend Framework master? Find out more http://www.masterzendframework.com.

Ensuring Your Tests are Valuable

David Stockton

Writing tests for our code and applications is critical. It gives us a repeatable, reliable way to ensure the code we write does what it should do and doesn't do what it shouldn't. Often, code coverage is used as a way to measure how much of a codebase is tested, but it only tells part of the story. More importantly than reaching the fabled 100 percent code coverage metric, we should be concentrating on building valuable tests.

What is Code Coverage?

Code coverage in unit tests is a measurement of which lines of code are executed during a run of tests. If you have XDebug installed and run your test suite, you have the option of outputting code coverage data in a number of formats, including XML, HTML, or even PHP arrays with each line of each file listed along with an indication of whether it was executed or not. In the HTML report, if the line was run, it is colored green and if it was not executed, it has a red/pink background. If your tests run every line of code in your application, you'll see 100 percent code coverage. The coverage report will give you numbers and percentages by file, class, and line.

There's a very important distinction, however, between ensuring that 100% of the lines of code are executed and 100%of the lines of code are tested. Relatively speaking, it's "easy" to get to 100% code coverage, but it's very difficult to ensure that 100% of the lines of code are **tested**. In this article we'll be talking about ensuring that the lines of code that are covered are covered because they've been effectively tested.

What Makes a Valuable Test?

There are a couple of aspects that make tests valuable. If the test helps ensure that the code does what it should do and doesn't do what it shouldn't, then it is valuable. If the test provides a safety net to ensure that refactoring doesn't break things, then it's valuable. So the benefits we want in the test are that:

1. Changes to our code that don't break things should be easier, since we'll know if we've goofed and broken things
2. Changes that do break things should cause test failures, so we know when important behavior that we rely on is no longer happening.

Unfortunately, not all tests are valuable. Some tests make it hard to change code for no good reason. Typically these are tests that get too deep into the implementation of the code under test. Ideally, most of our tests will send input into a method and ensure the output or return value is what is expected. For simple "pure function" tests, this isn't too hard. But in the real world, we need to test things that have side effects or dependencies. Walking the line between providing valuable assurances and tying ourselves to a specific implementation can be very difficult without making the test fragile or removing any assertions that cause the test to actually test something.

Other aspects that make a test valuable is that it must be consistent. What this means is that if you run the test against the same code over and over, the result should always be the same. If the test fails, it should fail every time it is run until we either change the code or we change the test. Tests that inconsistently pass or fail on subsequent runs are less valuable because they introduce doubt into the process of running the tests. If there is doubt then developers learn to mistrust the tests or ignore build failures. This isn't to say that all your tests should necessarily use static input. I'm a big fan of using appropriate random inputs in some tests. I know not everyone agrees, so let's take a look at some examples and reasons.

Random Input for Tests

When following the standard TDD approach to building software, the normal loop is to write a test, run it, see it fail, then write the minimum code to make sure all the tests are passing. What that means is that once you've been writing code like this for a bit, you end up with a first test that is trivial, and code that is equally trivial. For instance, in the following snippet from a PHPUnit test:

```
public function testAddCanAddTwoAndTwo() {
   $this->assertEquals(4, $this->calculator->add(2, 2));
}
```

If we're building a simple calculator using TDD, and this is our first test, then the simplest code that makes it work is something like this:

```
class Calculator
{
   public function add($x, $y) {
      return 4;
   }
}
```

At this point, we have a fully functional calculator that works correctly and returns 100 percent correct results for adding any two numbers—as long as the two values we add total to four. However, it fails pretty badly on all the other sets of input. In order to get to the next level, where we can add up other values besides those that add to four, we create another test:

```
public function testAddCanAddTwoAndThree() {
   $this->assertEquals(5, $this->calculator(3, 2));
}
```

With this test in place, the simple `return 4;` code no longer suffices. But, it doesn't completely guarantee that the code will be correct. In fact, the following code will pass these two tests, but it's an even worse calculator adding function:

```
public function add($x, $y) {
    if ($x == 3) {
        return 5;
    }
    return 4;
}
```

Now, clearly, going from what we had to this is ridiculous, but it illustrates why I do like to use random inputs on occasion, and when I know the inputs will be within an acceptable range. Let's take a look at the test with random inputs:

```
public function testTheCalculatorCanAddInputs() {
   $x = rand(1, 10000000);
   $y = rand(1, 10000000);
   $this->assertEquals($x + $y,
                       $this->calculator->add($x, $y));
}
```

With this code in place, in all likelihood our calculator `add` method will end up being `return $x + $y;`. If we know the inputs are going to be integers or numeric values, then this method is likely all we'll need. Of course, if we can't be sure of the incoming values, then we'll need more tests and more results to be able to ensure that code behaves as expected. Perhaps the calculator needs to throw an exception if it is passed non-numeric strings, or objects, or arrays. Or maybe we need to ensure that we have some expected behavior if the two values add up to something larger than `PHP_MAX_INT`. Each of these situations would require additional tests.

Now, clearly, the "add" is trivial and silly, and we're really duplicating all the work the method does right there in the test. But randomly generated values can also be a good way to ensure that your parameters are being used appropriately. If you've built a class that performs a SQL query, perhaps one of your incoming parameters will be an id. By randomizing that id, and then asserting that the mock database object sees that same random value, you're pretty close to guaranteeing that the incoming id is being sent into the query.

Let's explore another situation that has a more limited input scope for what would be considered valid and invalid. Suppose we want to build a validator that ensures that some incoming value is a string containing between 12 and 16 digits. In order to thoroughly test, we'd want to test that the validator returns an invalid response for inputs that are empty or up to eleven characters, as well as an invalid response for inputs over 17 characters. Next we want to ensure that inputs that are between 12 and 16 characters but are not just digits will be invalid.

The following test is an example of a test with random inputs that will pass most of the time but not always and, unfortunately, closely mirrors some real tests I have seen in real project code. These are the types of tests we want to avoid because they cannot be trusted.

```
public function testStringsWithLettersAreInvalid() {
   $length = rand(12, 16);
   $string = substr(md5(uniqid(), 0, $length);

   $this->assertFalse($this->validator->isValid($string));
}
```

On the surface, it looks decent. We're testing a string with the valid length range of 12 to 16, and we're building the random string from the result of md5 on a random string. The output of `md5` is a 32-character string made up of lowercase hexadecimal digits (essentially, 0-9 and a-f). Most of the time, this test will work. However, a problem occurs because sometimes md5 will return a hash that is entirely digits, or even more commonly, the first 12–16 characters could consist of just numbers. Whenever this happens, this test will fail because the resulting string will be counted as valid when the test expects an invalid response. Tests that include random values that do not always fall within the correct or expected domain are bad and should be avoided. What I mean by this is that the test is always expecting to have a generated value that is invalid. However, the code that generates this input will sometimes generate a valid input. So we've now violated the domain of inputs by generating an input that crosses the boundary between invalid and valid.

Ensuring Tests are Effective

At this point, I'm ready to start talking about the tool and concept that I hinted at last month. The concept is known as mutation testing. If our tests are supposed to let us know when the code is broken, then making certain small logic changes to the code should result in a test failure. If it doesn't then we can be reasonably certain that our test suite doesn't adequately test some parts of our code.

Let's take a look at a simple setter/getter and a test.

```
public function setFoo($foo) {
   $this->foo = $foo;
   return $this;
}

public function getFoo() {
    return $this->foo;
}
```

The test for this could look like this:

```
public testClassCanStoreAFooValue() {
    $value = uniqid('foo_value_');
    $this->thingToTest->setFoo($value);
    $this->assertEquals($value, $this->thingToTest->getFoo());
}
```

If we were to run this, we'd see that we've managed to get 100% code coverage on the `setFoo` and `getFoo` methods. However, not all the lines of code are actually tested. There's nothing in that test that ensures that the `setFoo` method is providing a fluent interface and there may be code that relies on it. If we removed the `return $this` line of code from the setter, the test would continue to pass.

Enter Humbug

In late 2014, Pádraic Brady made the first commits to his mutation testing tool known as Humbug. The tool inspects the tests and code under test and determines a number of changes that can be made. For instance, Humbug will find the `return $this;` line and change it to `return null;`. It will then run the unit tests and if nothing fails, it counts that run as an 'escaped mutant'. If the test fails, then the mutant has been killed and the test was effective.

Installation instructions for Humbug can be found on the Humbug GitHub page[1]. In order to execute Humbug, you must first have a working (read passing) unit test suite. Humbug will execute it, figuring out what code is covered. It will then run through and determine what mutations are possible. Then, one at a time, it will change the code with each mutation for the lines of code that are

1 Humbug GitHub page: http://phpa.me/humbug-installation

covered and run the test suite. If running the test suite results in a test failure, a fatal error, or a timeout, it concludes that the mutation has been detected by the test suite. If the test suite passes, then it considers the mutant as having escaped.

Let's take a look at how it can work. First of all, you need a project with some sort of PHPUnit test suite. After you install Humbug, you'll need to generate a configuration file for Humbug. This is done with `humbug configure`. It will prompt you to enter a few values: the source directories you want to include, any you want to exclude, how long Humbug should wait before saying the test timed out, and where to store the logs, both text and JSON.

Once that's in place, you run Humbug with `humbug run`. Let's take a look at some code samples to go along with all of this.

Listing 1 is the `ValueObject` class. For right now, it's just a simple class that holds some value in a variable named `foo`. There are a setter and getter for `foo`, which will be the subject of our tests.

Listing 2 is the test for the `ValueObject` class. It's testing that whatever is passed into the setter for foo is what we get back out from the getter. With these two files in place, along with the appropriate autoloading and `phpunit.xml` config, we have a passing unit test suite with a single test. We **must** have a passing unit test suite for Humbug to work.

Listing 3 shows the result of `humbug run`. When you initially run Humbug, it executes your test suite and looks at all the code that is mentioned in the coverage report. If you've enabled phpunit to process all uncovered files from your whitelist, then you'll potentially see a whole lot of work being done. Humbug looks at the lines of code listed and looks for potential mutations. We'll talk about what those mutations are shortly. Once it has identified all the mutations, it will only run the ones that involve code that is covered. For this example, that's all of them because there is so little code. The `M` in the output indicates that the mutation that was generated was not caught by our tests. Humbug also includes a few metrics to let us know how things went. The Mutation Score Indicator (MSI) indicates how many of the mutations are caught or killed. In this example, it's 0 percent, since they all escaped. Mutation code coverage indicates how much of the codebase is covered by the mutations that are executed. Finally, the Covered Code MSI indicates how many of the executed mutants were killed or detected.

LISTING 1

```php
<?php

namespace LevelUp;

class ValueObject
{
    private $foo;

    /**
     * @return mixed|null
     */
    public function getFoo()
    {
        return $this->foo;
    }

    /**
     * Sets the foo value for the object
     *
     * @param mixed $foo
     *
     * @return void
     */
    public function setFoo($foo)
    {
        $this->foo = $foo;
        return $this;
    }
}
```

LISTING 2

```php
<?php
namespace LevelUpTest;

use LevelUp\ValueObject;

class ValueObjectTest extends \PHPUnit_Framework_TestCase
{
    /** @var ValueObject */
    private $valueObject;

    public function setUp()
    {
        $this->valueObject = new ValueObject();
    }

    public function testItCanStoreAFoo()
    {
        $foo = uniqid('something_');
        $this->valueObject->setFoo($foo);
        $this->assertEquals($foo, $this->valueObject->getFoo());
    }
}
```

LISTING 3

```
 _  _                   _
| || |_  _ _ __  | |__ _  _ __ _
| __ | || | '  \| '_ \ || / _` |
|_||_|\_,_|_|_|_|_.__/\_,_\__, |
                           |___/
Humbug version 1.0.0-alpha1-18-gd102496

Humbug running test suite to generate logs and code coverage data...

    1 [============================================================]  1 sec

Humbug has completed the initial test run successfully.
Tests: 1 Line Coverage: 100.00%

Humbug is analysing source files...

Mutation Testing is commencing on 2 files...
(.: killed, M: escaped, S: uncovered, E: fatal error, T: timed out)

M

1 mutations were generated:
       0 mutants were killed
       0 mutants were not covered by tests
       1 covered mutants were not detected
       0 fatal errors were encountered
       0 time outs were encountered

Metrics:
    Mutation Score Indicator (MSI): 0%
    Mutation Code Coverage: 100%
    Covered Code MSI: 0%

Remember that some mutants will inevitably be harmless (i.e. false positives).

Time: 232 milliseconds Memory: 7.00MB
Humbug results are being logged as JSON to: humbuglog.json
Humbug results are being logged as TEXT to: humbuglog.txt
```

LISTING 4

```
------
Escapes
------


1) \Humbug\Mutator\ReturnValue\This
Diff on \LevelUp\ValueObject::setFoo() in /Users/davidstockton/Projects/humbug_demo/src/LevelUp/ValueObject.php:
--- Original
+++ New
@@ @@
         $this->foo = $foo;
-        return $this;
+        return null;
     }
 }
```

The log files that are generated are very useful as well. While they don't show every potential mutation found or even run, they do show any of the mutations that escaped and what the change was. Listing 4 shows the log for our first run with the sample code. Or rather, it shows part of the log. The text in Listing 3 showed what appears on the command line when Humbug is run. That same output also appears in the Humbug log, but I've removed it here in the interest of brevity. In Listing 4, we find that the mutation that was not detected or killed was intended to transform the `return $this;` from our setter to `return null;`. Since the test didn't test that the method is fluent, changing the setter to be non-fluent did not induce a test failure, and the mutation was not caught.

Unlike a PHPUnit run, where dots are typically the only good thing and all you want to see, in a Humbug run, it's different. Of course, dots are the best, but `E` and `T` may be acceptable as well. `E` indicates that the mutation resulted in a fatal error when the tests were run. A `T` indicates that the test timed out. This can happen because a mutation could introduce an infinite loop. There are two other letters that may appear. An `S` indicates that none of the tests covered that line of code (or specifically the mutation). An `M` means that a mutation was found, the code was changed, but the tests still passed. We want to avoid escaped mutants. For our simple value object, we need to add another test or assertion that ensures that the setter is fluent.

We can update the setter test to this:

```
public function testItCanStoreAFoo() {
    $foo = uniqid('something_');
    $result = $this->valueObject->setFoo($foo);
    $this->assertEquals($foo, $this->valueObject->getFoo());
    $this->assertSame($this->valueObject, $result);
}
```

Now if we run Humbug again, we get a single dot. We also achieve 100% in all of the metrics, indicating that we've got 100% for traditional code coverage as well coverage that ensures that all mutations were caught and killed.

Humbug Mutations

Humbug looks for a number of code patterns where it will introduce mutations. We've already talked about changing a `return $this;` to `return null;`. Additionally, Humbug will change `+` signs to `-` and vice versa. It will exchange `*` and `/`. The modulus operator (`%`) will become multiplication. Raising to a power with `**` will turn into a division operation. It will change `+=` to `-=` and `*=` to `/=`. In bitwise operators, it will exchange `&` and `|` and swap the direction of the `>>` and `<<` bitshift operators.

For logic changes, `true` and `false`, `&&` and `||` are never safe when Humbug is around. It will swap them for the other wherever they are found. The same goes for `and` and `or`. If it finds the not (`!`) operator, it will simply remove it and see if your tests notice. For inequalities, a strict greater than (`>`) is transformed to a `>=`, while a `<` turns into a `<=`. The opposite directions also apply. It will change conditionals like `==` to their negated `!=` versions. The increment (`++`) and decrement (`--`) operators will be swapped too.

It will mutate return values as we saw before, but it will swap `return true` and `return false`, as well as changing things like `return 0` to `return 1`. In addition, any `return (anything);` will be changed to `(anything); return;`. Some literal numbers like `0` and `1` will be changed whenever found. And there are more mutations that I haven't listed. In short, Humbug will identify and potentially make many changes to the code you're testing to see if your tests are really testing that the code does what it should. What this means is, the more code you're testing, especially if it falls into any of the above categories or samples, the more mutations can be identified and potentially executed. While this isn't slow, it's definitely not fast. If your code takes perhaps 5 or 10 seconds to run your tests normally, running the mutation tests could take 5 or 10 times longer to execute.

Using Humbug to Improve Your Tests

The code I work with most often is a Zend Framework 2 application. Each module has its own test suite separated from all the other test suites. One thing that I found helpful when working with Humbug was to pick a module, run Humbug, and then use the output to improve the tests. For a module with no actual tests, I might have an entire block of just `S` tests indicating that there's no code coverage to start. So I can then pick a class and set about writing tests for the methods in the class. Then I'll run Humbug again and see how it changes the output. Whatever lines of code I've managed to cover with PHPUnit will now change from an `S` in Humbug to something else. In some cases, I may get lucky and get a dot (`.`) from the start. In other cases, I may see some escaped mutant `M` tests. There may also be some `E` or `T` tests as well.

I then concentrate on killing as many mutants as I can. Ideally, I don't want to stop while there are any `M` tests left. The log file is very useful because it will show a diff format on the bit of code that changed for each escaped mutant and I can use that to move forward and devise a test that will catch and kill the mutant. In some cases, the best I can do is an `E` or a `T` test. Some of the time, I may make an effort to restructure my original code in such a way that I can convert those to a `.`. After I've had enough for the day, I'll remove my Humbug configs and logs and commit the test and code changes into source control.

I haven't been automatically running Humbug on the continuous integration servers because I feel it would dramatically increase the run time of the builds and we're not always going to be paying attention to the Humbug output. Instead, I'd rather focus on improving the tests a bit at a time and then ensuring that the result of those exercises are run automatically by the CI server. That seems to bring the biggest bang for the buck with this type of testing.

Conclusion

Mutation testing with Humbug is a powerful way to ensure that your tests are worthwhile and that they are actually testing your code in a meaningful way. If the tests you have are not actually able to detect bugs when they are introduced then they may be giving you a false sense of security. Even though Humbug is still technically alpha, I'd recommend trying it out on your own code and see how well your tests actually test your code. If you're not running PHPUnit but do have tests in another framework like PHPSpec, unfortunately, at least for right now, you're out of luck. There is an effort to expand Humbug outside of just PHPUnit, but for now that's what is there. If you're interested in reading more on the topic, please check out Pádraic Brady's blog[2] on the topic. See you next month.

David Stockton is a husband, father and Software Engineer and builds software in Colorado, leading a few teams of software developers. He's a conference speaker and an active proponent of TDD, APIs and elegant PHP. He's on twitter as @dstockto, YouTube at http://youtube.com/dstockto, and can be reached by email at levelingup@davidstockton.com.

2 *AstrumFutura: Mutation Testing: http://phpa.me/astrum-futura-mutation-testing*

Securing Legacy Applications–Part 1

Chris Cornutt

I want to take some time in the next few issues to give you some things to think about when refactoring legacy applications and tips on how to make it more secure as you go. In this first part of the series I'm going to look at a few "quick hits" that you can look for in your current codebase. These are things that are pretty easy to fix, too, making them relatively simple to implement.

Most developers involuntarily cringe when they hear the words "legacy application," and with good reason. More often than not, a legacy application usually comes with outdated and poorly architected code that's a mess to try to work with. I've seen cases where the "if it's not broken, don't fix it" mentality always wins out, even if the code they're adding has to conform to other broken standards already in the codebase.

One of the key factors that makes legacy applications tough to deal with is their complexity. Usually this is caused by a developer (maybe even you!) who, at some time in the past , made some poor architecture decisions—or none at all—and pieced together a system that works but is extremely fragile. In these systems, a single change could result in a chain reaction of failures, sometimes without the developer even knowing until some hapless user stumbles across it.

Legacy applications also come with their share of bugs. Sometimes these bugs are "features," and other times they're just annoyances that people just know to work around. These bugs come in many shapes and sizes and cover a wide range of topics inside the same application. However, in this series of columns I want to reflect on a certain kind of issue in legacy applications—security problems. While not all bugs in legacy applications are security issues, these can be some of the more serious. For example, if a user is able to bypass your login using a simple SQL injection because your queries use string concatenation, that's a pretty major hole.

As with any development advice, take these suggestions with a grain of salt. All codebases are different and there will be different challenges in each. Sometimes fixes that are quick for one codebase can be a nightmare for another…it just depends on how the feature is used.

Finding Places Where Superglobals are Incorrectly Used

Superglobals[1] are super-handy in PHP but they can also cause a lot of trouble if used incorrectly. Since they're global everywhere, there are all sorts of issues around decoupling and global state, but I'm not going to focus on that stuff here. That's more for a good old-fashioned software engineering discussion. Instead, I want to help you look around your code and find the places where these superglobals might be used and abused in a way that could compromise your application's security.

One of the most common places they're abused is in building SQL statements and using them completely unfiltered. If you're poking around your code and see this happening, this is an *immediate* refactor point. While the data from the superglobals should always be considered tainted, there is a simple fix that can help protect from SQL injection issues: using prepared statements and bound parameters. In PHP both mysqli and PDO support this functionality, and I highly suggest looking into it. Making the switch usually requires only a few lines different from how you might be using the `mysql_` functions right now, but it's worth it in the long run.

Avoid Using $_REQUEST

The worst usual offender is the use of `$_REQUEST`. So, why is using this variable a bad thing? By its nature it doesn't mark any difference between which source the data is coming from. It combines the information from `$_GET`, `$_POST` and `$_COOKIE` into one data set. The real tick here is that, if there are values with the same key name (say a `GET` and `POST` variable both named `userId`) one would overwrite the other. The order of this is governed by a `php.ini` setting that is completely in the control of the server you're using to run your code: `variables_order`. Since this isn't something that you can rely on as always being consistent, you can't trust the data that's in `$_REQUEST` as it's presented.

1 *PHP Superglobals:* http://php.net/language.variables.superglobals

Replace Hard-coded Credentials

Another major issue I've seen in older codebases is the hard-coding of credentials. It's common to look at classes that connect to external services or even your own databases and see properties for a `username` and `password` right there in the code. While it might have made sense at the time, there's a major issue here—if someone ever managed to get a copy of the source code, either from the site or version control, they'd have immediate access to those credentials.

To solve this issue, there are a few steps to take:

1. First, you want to change those credentials and get fresh values. This means refreshing the tokens you may be using on that API or updating your database username and password.
2. Next, you need to figure out the best storage method. Since the goal here is simplicity, I usually recommend using something like a `.env` file that's not checked in to your repository but lives somewhere else on the server. You can then use a library like phpdotenv[2] to pull those values in and use them directly. This prevents those credentials from falling into the wrong hands.
3. Then the hard part: deploying those changes. It's tricky because you have to make the push at the same time as you update the values. Ideally, you would schedule some kind of downtime to help with this as it's a pretty major change.

These are just a few suggestions to help you get started in refactoring your application to make it more secure. The things I've discussed here are more specific to security concerns, but refactoring for simplicity is a good idea no matter what part of the code you're working with. Legacy code, with all its complexity, can be less secure than some of its more well-architected cousins, but with a little work you can quickly bring it up to speed.

For the last 10+ years, Chris has been involved in the PHP community. These days he's the Senior Editor of PHPDeveloper.org and lead author for Websec.io, a site dedicated to teaching developers about security and the Securing PHP ebook series. He's also an organizer of the DallasPHP User Group and the Lone Star PHP Conference and works as an Application Security Engineer for Salesforce.

2 *phpdotenv:* https://github.com/vlucas/phpdotenv

February Happenings

PHP Releases

- PHP 7.0.3: http://php.net/archive/2016.php#id2016-02-04-1
- PHP 5.6.18: http://php.net/archive/2016.php#id2016-02-04-3
- PHP 5.5.32: http://php.net/archive/2016.php#id2016-02-04-2

News

Alejandro Celaya: How to Properly Implement Persistent Login

In his latest post to his site Alejandro Celaya shares some suggestions about how to make a good, safe persistent login feature for your application. This is usually referred to as the "remember me" handling and is widely used to help improve the overall user experience. He also shares some other security concerns to think about in this setup including the use of one-time tokens, potential multiple persistent sessions and when it might be good to re-prompt for the password.
http://phpdeveloper.org/news/23652

Paragon Initiative: How to Safely Store a Password in 2016

On the Paragon Initiative site they've posted a new article showing you how to safely store a password (in 2016) that discusses both the concepts around good password hashing and how to do it in several languages (including PHP). He advises using libsodium for some of the best protection but points out that it's not widely supported yet.
http://phpdeveloper.org/news/23672

Johannes Schlüter: References—Still Bad in PHP 7

Johannes Schlüter has a post to his site that talks about references in PHP 7 and how they're "still bad" based on some of his previous findings. He includes his testing code that calls a function (`strlen`) in a loop and compares the handling against two methods, one passing by reference the other not. The results are shown in time taken to execute.
http://phpdeveloper.org/news/23684

Magium Blog: 3 Best Practices for Selenium Testing when Constructing Your Page

In a new post to the Magium site Kevin Schroeder shares three helpful tips you can use for the Selenium testing of your application based on some of his recent development on the project. His three tips avoid things like long XPath expressions to locate single items and favor consistency and simplicity.
http://phpdeveloper.org/news/23675

Symfony Finland: Going Async in Symfony Controllers

On the Symfony Finland site Jani Tarvainen has posted a tutorial showing you how to create asynchronous controller handling in a Symfony-based application. Thanks to some other projects, however, asynchronous development with PHP has become more of a reality. He shows how to use one of these projects, Icicle, and its coroutines functionality to make a Symfony controller that handles calls to a sayHello method asynchronously, returning messages in a fraction of the normal processing time.
http://phpdeveloper.org/news/23688

Drupal.ovh: Why is Drupal Still a Gated Community?

Yurit looks at how Drupal could become a better contributor to the overall PHP ecosystem by making bits and pieces of Drupal usable outside of Drupal. Despite the heavy usage of Symfony and other components in Drupal 8, the author still sees a big opportunity for the Drupal community to reciprocate.
http://drupal.ovh/node/39

PHP RFC: Deprecations for PHP 7.1

Nikita Popov has authored an RFC looking at multiple functionality that should be deprecated for PHP 7.1 and removed by PHP 8.0. Some items on the proposed list include the `__autoload` function, `$php_errormsg`, and the functions `rand()`, `srand()`, and `getrandmax()`.
https://wiki.php.net/rfc/deprecations_php_7_1

ThePHP.cc: Questioning PHPUnit Best Practices

In this new post to thePHP.cc blog Sebastian Bergmann (creator of the PHPUnit unit testing tool) questions of some the current "best practices" involved in using the tool. More specifically he looks at the handling for expected exceptions and proposes a new practice to use going forward.
http://phpdeveloper.org/news/23639

Larry Garfield: Composer vs. Linux Distributions: A Mental Model Battle

In his latest post Larry Garfield talks about the Composer problem that was recently brought up by the Gentoo linux project and is related to how Composer packages and system-level shared libraries differ. Larry starts with a bit of history on the subject, pointing out the two methods most developers used PHP code: copy/pasted from the web or installed via PEAR. He talks about the common issues with both approaches. He then talks about how modern PHP development and Composer related and how, from a sysadmin perspective, Composer is the "compile" step of PHP and only supports static links.
http://phpdeveloper.org/news/23702

Loïc Faugeron: The Ultimate Developer Guide to Symfony—Dependency Injection

Loïc Faugeron has posted another of his "Ultimate Developer Guide" series to his site, this time with a focus on the dependency injection component of the Symfony framework. He starts with an introduction to the dependency injection pattern by refactoring a request to an API for its status out into classes with different responsibilities. He includes both the code he started with and what he's refactoring to. He thing brings in the Dependency Injection component, shows how to register the different classes and define a YAML configuration with their relationships.
http://phpdeveloper.org/news/23701

Growing Programmer Population

Eli White

Every month when I write this article, I find a pertinent quote to stick in it once I'm done. This time, I'm going to start the article with a quote:

> *"The number of programmers doubles every five years. That means, at any time, half the world's programmers have less than five years' experience."*
>
> *— Robert C. Martin (Uncle Bob)*

Take a moment to think about that simple statement.

Half of the programmers in the world have less than five years of experience. Doesn't this begin to explain an awful lot? This is especially important to us in the PHP community, because PHP is often seen as a perfect entry language, and the language where many people get started. (Even if they don't realize it, because they are writing "WordPress code" or "Drupal code" in their mind.)

As the number of programmers continues to grow, and as PHP's impressive hold over enterprise web development continues as well, one could even start to imagine that the number might be even a little higher in the PHP community.

Think of the other side of that as well. It means that all other levels of experience fit into that other 50%. People with six to 60 years of experience make up that other 50%. Which means that some of us, like myself, with 20 to 30-plus years of experience, are in the extreme minority.

Consider this anecdote: I was at a polyglot conference over last summer, and found a group of programmers who self-identified at a PHP programming round table. As I got to talking to them, I was shocked to discover that they not only didn't know about php[architect], but they didn't know about our conferences, or any PHP-specific conferences. (They were at the polyglot conference because it was the only one that had some PHP content that they knew of). They didn't know that PHP User Groups existed. When I asked why they didn't assume so, the response was: "Are there C programming user groups? Are there Fortran groups?" Well, no. So why would they assume that PHP had an ecosystem around it? It was "just another programming language," and one of those foundational ones that everyone uses.

These programmers, all in their early twenties, had graduated college, found jobs fresh out of school as junior devs, and the jobs just "happened" to be using PHP. It turns out this was very typical in the circles that they knew.

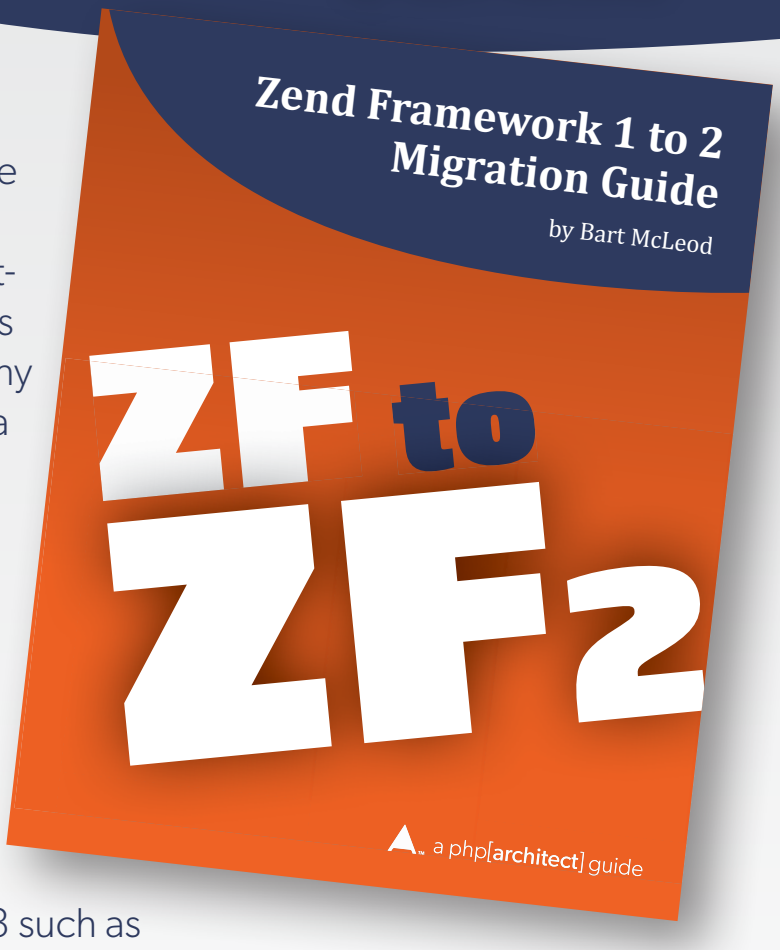
This eye-opening fact really helped explain some things to me. It's why we need *Basic Web Security* sessions at every conference, because there will be numerous people who have never heard of web security. It's why we need to not only focus on advanced topics, but to make sure the basics of the PHP language are still covered. It's why training and content are so important as well.

But to me it's also a huge callout to the PHP community in general. How welcoming are we (are you?) to someone fresh out of college who happens to be writing in PHP? Someone who doesn't seem to want (yet) to buck up, join a user group, and contribute to open source? Realize that 50% of people are in that situation—fresh to programming, just getting a start on this life. We need to remember that; be open and welcoming to them. Show them all what the PHP community has to offer. (And the fact that there **is** a PHP community.)

Eli White is the Managing Editor & Conference Chair for php[architect] and a Founding Partner of musketeers.me, LLC (php[architect]'s parent company). He is an old fart and has been programming professionally for 22 years now, and as a hobbyist for 32 years. @EliW

Zend Framework 1 to 2 Migration Guide

by Bart McLeod



Zend Framework 1 was one of the first major frameworks for PHP 5 and, for many, introduced object-oriented programming principles for writing PHP applications. Many developers looking to embrace a well-architected and supported framework chose to use it as the foundation for their applications. Zend Framework 2 is a significant improvement over its predecessor. It re-designed key components, promotes the re-use of code through modules, and takes advantage of features introduced in PHP 5.3 such as namespaces.

The first release of ZF1 was in 2006. If you're maintaining an application built on it, this practical guide will help you to plan how to migrate to ZF2. This book addresses common issues that you'll encounter and provides advice on how best to update your application to take advantage of ZF2's features. It also compares how key components—including Views, Database Access, Forms, Validation, and Controllers—have been updated and how to address these changes in your application code.

Written by PHP professional and Zend Framework contributor, coach, and consultant Bart McLeod, this book leverages his expertise to ease your application's transition to Zend Framework 2.

Purchase

http://phpa.me/ZFtoZF2

PHP SWAG

PHP Drinkware

Tumbler

PHPye Shirts

php[architect]™

Visit our Swag Store where you can buy your own plush friend or other PHP branded gear for yourself.

As always, we offer free shipping to anyone in the USA, and the cheapest shipping costs possible to the rest of the world.

Get yours today!

www.phparch.com/swag

Borrowed this magazine?

Get php[**architect**] delivered to your doorstep or digitally every month!

Each issue of php[**architect**] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.

php[architect]

magazine
books
conferences
training

www.phparch.com

Digital and Print+Digital Subscriptions Starting at $49/Year

http://phpa.me/mag_subscribe