4 TIPS FOR FLEXIBLE PHP

FUTUREPROOF PHP

BY ANDREW SHELL

www.FutureProofPHP.com

Hoopla!

I'm Andrew Shell, founder of Madison PHP and creator of FutureProofPHP.com.

Congratulations and thank you for downloading "4 Tips for Flexible PHP"!

In this ebook, you will find my top four tips for writing flexible PHP.

In the 13 years I've been working as a professional PHP developer, I'm embarrassed by the number of mistakes I've made through trial and error.

If I had known about these strategies from the beginning, I would have saved myself a lot of headaches, time, and money.

Now, I am sharing my experiences with you so you can start writing future-proof PHP.

Cheers,
Andrew

Tip #1: Automated Tests

If you want to keep your code flexible over time, you should be writing automated tests using a tool like PHPUnit.

Testing your code as you're writing it (or even before) makes sure it's *testable*. What makes something testable? Smaller classes and methods that do less. After all, you don't want to write a test that has to account for twenty different logic paths.

By making it easier to test, you're actually writing better code. You'll also thank yourself down the road when you have to change code, because you'll know if what you've done
broke anything.

There are also different types of tests that accomplish unique goals.

- **Unit tests** test how specific parts (units) of code work. It makes sure that a specific method in a specific class does what its supposed to do.
- **Integration tests** make sure that the system of classes working together do what they are supposed to. These are very important when writing flexible code. If each class is simple and tested, bugs appear in the interactions between the objects.

  You can use integration tests as you're building to make it easier to identify the next part you have to build. Stub out pieces at first and then, as you work through, build the parts you're stubbing out.

Here is an example of an integration I test I wrote in PHPUnit.

I don't have a version of LazyLoadedListener that I'd actually use in production, so I just created a simple "Mock" object that implements the LazyLoadedListener interface.

I'm testing how the parts work together. There is a LazyLoadedListener, a Dispatcher and PatRun which is an implementation of PatternMatcher.

If this passes we know that the interaction between those three classes work.

```
109     public function testActWithLazyLoadedListener()
110     {
111         $lazyLoadedListener = new MockLazyLoadedListener(function ($msg) {
112             return $msg['value'];
113         });
114
115         $dispatcher = new MemoryDispatcher(new PatRun());
116
117         $dispatcher->add(['role' => 'test'], $lazyLoadedListener);
118
119         $this->assertEquals(
120             'hoopla',
121             $dispatcher->act(['role' => 'test', 'value' => 'hoopla'])
122         );
123     }
```

https://github.com/andrewshell/patrun-dispatcher/blob/1.x/tests/MemoryDispatcherTest.php#L109-L123

Tip #2: Inject Dependencies

If you have a class that needs a database connection, you will want to pass the connection into the class as a dependency in the constructor. Does that tip seem too obvious? Of course you'd pass it in, you don't want to be reconnecting to the database in every object in your application!

What if your class is connecting to a remote API? It couldn't hurt to instantiate a Guzzle[1] client in your class if you're not going to be reusing it, right? Wrong.

Remember tip #1? You're going to want to test this class, and it's not easy to test a class if it's making calls to a remote server. Instead you'll want to pass in a fake Guzzle client for your class to use so you can test that your class is working the way it's supposed to. You don't want your tests waiting to connect to a server or to fail if you're offline.

You'll also want to use a fake client to test how your class responds under different circumstances. What happens if the API is down, or if it returns a response that you don't know how to handle?

Dependency injection is great for code reuse, but it's also essential for writing testable code.

1. Guzzle is a PHP HTTP client that makes it easy to send HTTP requests and trivial to integrate with web services.

Here is an example of a class I created that uses a GuzzleHttp\Client object. As you can see I passed it in as a dependency so that it was easier to test.

```php
use GuzzleHttp\Client;
use GuzzleHttp\Exception\GuzzleException;
use StatusBot2000\Core\Check\Check;
use StatusBot2000\Core\Check\CheckException;

class GuzzleHttp implements Check
{
    private $client;
    private $url;

    public function __construct(Client $client, $url)
    {
        $this->client = $client;
        $this->url = $url;
    }
```

https://github.com/statusbot2000/StatusBot2000.GuzzleHttp/blob/1.x/src/Check/GuzzleHttp.php#L4-L18

Tip #3: Depend on Interfaces

Assume you're writing testable code and injecting your dependencies. How are you defining your dependencies? You will want to use a Guzzle client. In your constructor, you will define a parameter that is expected to be an instance of GuzzleHttp\Client.

What will happen if a new version of Guzzle comes out that you want to use, but it's backwards incompatible with the version you're using? Or maybe you want to use a different library?

You'd have to rewrite every class that depended on it. Or possibly abandon any hopes of using the new library in the near future.

Instead you could have created your own interface, in your own namespace for HttpClient. Doing this exposes a few methods that will define what you want to do with an HTTP Client. For example, maybe a fetchUrl method that takes a URL and returns an array with keys statusCode, header, and body.

Or, in other words, create a class GuzzleHttpClient that implements your HttpClient interface and does the work and returns the array.

All of your classes depend on HttpClient (not GuzzleHttp\Client). When you want to switch to something else, you can create a new class FancyNewHttpClient and all of your classes can seamlessly start using the new library because they don't care how the interface is implemented.

If you look at the example I included in Tip #2 you'll see that the class I build actually implements the interface StatusBot2000\Core\Check\Check.

That is because the main application can handle any sort of Check as long as it implements the interface.

```
24      public function addCheck($name, Check $check, array $notifiers)
25      {
26          if (isset($this->checks[$name])) {
27              throw new Exception("Check {$name} already defined.");
28          }
29          $this->checks[$name] = array($check, $notifiers);
30      }
```

https://github.com/statusbot2000/StatusBot2000.Core/blob/1.x/src/Application.php#L24-L30

The core library defines the Check interface but does not depend in any way on Guzzle.

Tip #4: Use Cases

You might be used to Model-View-Controller or MVC architecture, because popular frameworks like Laravel, Symfony and Yii use this architecture. If you want a flexible application, however, you need to realize that the framework isn't your application. It's a *delivery mechanism* for your application.

Sure, you can build your app on a framework. It makes writing code really easy. You put logic in your controllers, some logic goes into your models (which get data from the database), and you have a great template engine so you can keep display logic from your business logic.

In the same way that you don't want to depend on a third-party library (see Tip #3), you don't want to be too dependent on your framework.

Instead, you can create classes that define the use cases in your application. These could be things like creating a user, publishing a blog post, or retrieving a list of products that are for sale. They contain the application specific business logic.

These classes would have well defined inputs and outputs. You'd use these objects in your frameworks controllers. The controllers would do the bare minimum needed to load and use these objects and interact with the user.

These classes in turn would use classes that encapsulate enterprise business rules. You might call these classes entities or domain logic. These are classes that could be reused across related applications.

The delivery mechanism, on the other hand, could be a web app, a web api, a command line tool, unit tests, or a message queue to name a few.

You would then follow the practices from the prior tips.

You can use the Object-relational mapper (ORM) from the framework to connect to the database but you wouldn't depend on it directly. Instead you'd create an interface like UserGateway and implement an OrmUserGateway that did the work using the ORM from your framework.

Your tests would run quickly and with minimal dependencies.

If there was a major update to your framework or you wanted to switch frameworks, you would have a clear path because the majority of your application wouldn't need to be changed. Just the objects that wrap the framework objects or libraries that have changed.

In this example I've created a Use Case called CreateNewPost. It leverages a PostGateway which can be implemented using any tools I care to use.

This Use Case can be used with any delivery mechanism we want. There is nothing in this class that cares if we're being called by a command line tool, a web API or something else entirely.

```php
<?php
namespace Blog\Domain\Interactor;

use Blog\Domain\Gateway\Post as PostGateway;
use Blog\Domain\Entity\Post as PostEntity;
use Blog\Domain\Interactor\CreateNewPost\Request as CreateNewPostRequest;

class CreateNewPost
{
    protected $postGateway;

    public function __construct(PostGateway $postGateway)
    {
        $this->postGateway = $postGateway;
    }

    public function __invoke(CreateNewPostRequest $request)
    {
        $post = new PostEntity(
            $request->getTitle(),
            $request->getContent(),
            $request->getExcerpt()
        );
        $this->postGateway->savePost($post);
        return ['success' => true];
    }
}
```

https://github.com/andrewshell/radar-fig-demo/blob/master/src/Blog/Domain/Interactor/CreateNewPost.php#L1-L27

Resources

Testing

PHPUnit	https://phpunit.de/
Codeception	http://codeception.com/

Libraries

Guzzle	http://docs.guzzlephp.org/en/latest/

Architecture

Clean Architecture	https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html
Hexagonal Architecture	http://alistair.cockburn.us/Hexagonal+architecture

Videos

FutureProof Your Code	https://youtu.be/yEc48QfkFcY