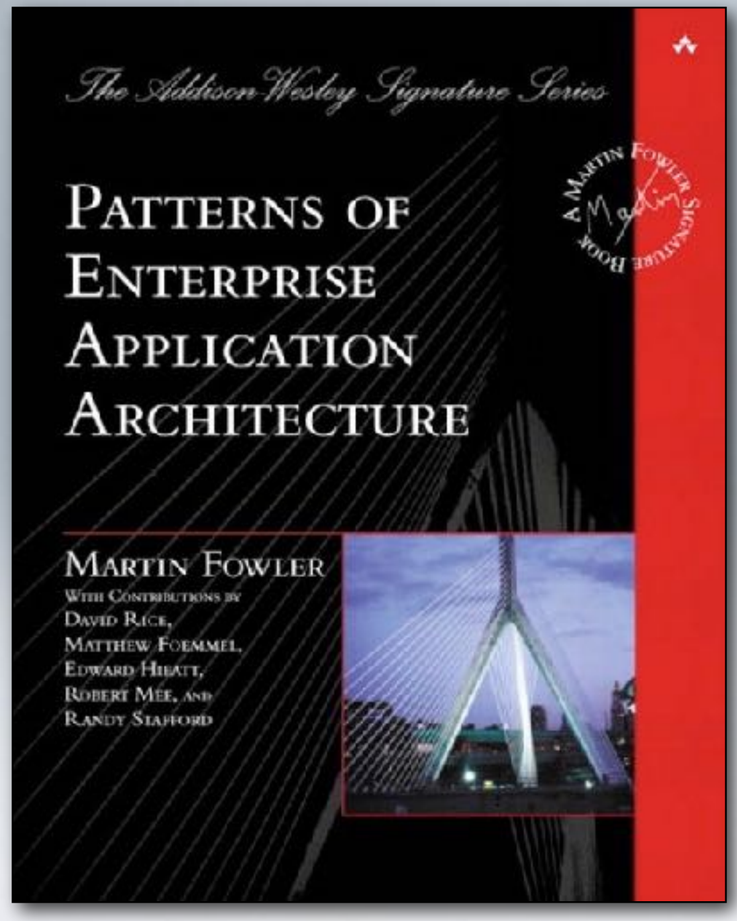
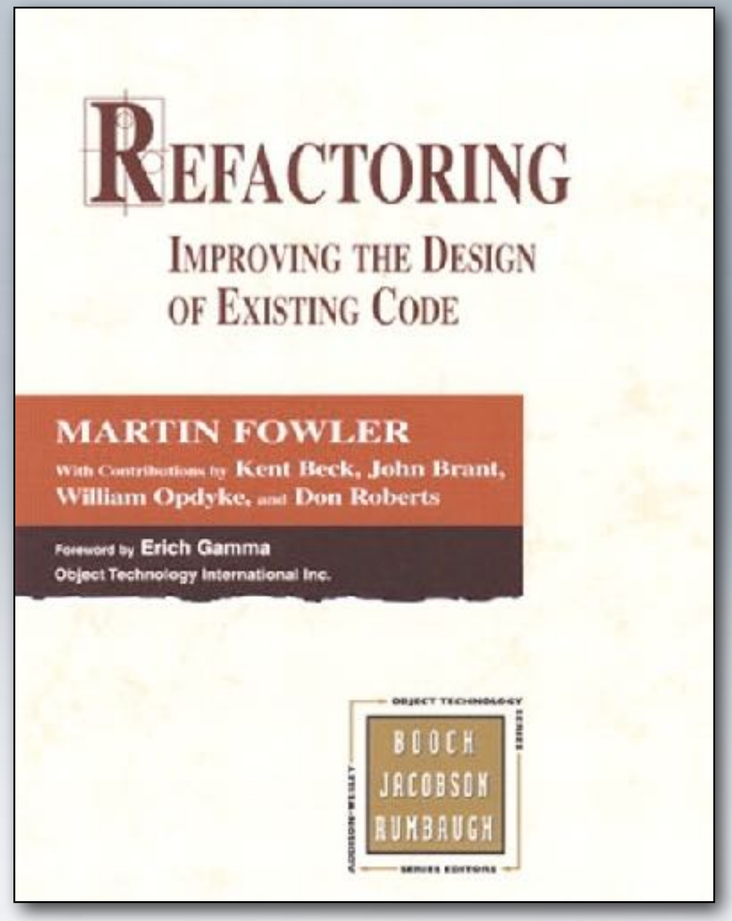
Decoupled Libraries for PHP

joind.in/10510

paul-m-jones.com
@pmjones

Read These


The Addison Wesley Signature Series
A Martin Fowler Signature Book
PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE
MARTIN FOWLER
WITH CONTRIBUTIONS BY
DAVID RICE,
MATTHEW FOEMMEL,
EDWARD HIEATT,
ROBERT MEE, AND
RANDY STAFFORD


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
ADDISON-WESLEY
SERIES EDITORS

About Me

- 8 years USAF Intelligence
- Programming since 1983,
  PHP since 1999
- Developer, Senior Developer,
  Team Lead, Architect, VP Engineering
- Aura project, benchmarking series,
  Zend_DB, Zend_View
- ZCE Advisory Board, PHP-FIG,
  PSR-1, PSR-2, PSR-4 (php-fig.org)

Overview

- Background: Libraries, Frameworks, Components
- Principles of decoupled library packages
- Examples: individual Aura library packages
- Limits to decoupling, direction of dependencies

Background: Libraries, Frameworks, Components (oh my)

Frameworks: Bad!

- PHP 3, PHP 4, and early PHP 5: "framework" a bad word ("content management system" was ok)
- Libraries and collections: phpLib, Horde, PEAR, PhpClasses, FreshMeat
- Not unified in operation: different constructor signatures, different method verbiage, different usage idioms, tough to combine
- Started Solar (solarphp.com) in late 2004 as a library collection (first of the PHP 5 E_STRICT collections)

Frameworks: Good! (Round 1)

- Ruby on Rails (2004/5): “framework” suddenly acceptable
- Agavi, Cake, CodeIgniter, ezComponents, Mojavi, PhpOnTrax, Symfony, Zend Framework, many others
- Tapped into developer needs, including tribal belonging and plugins

Frameworks: Good! (Round 2)

- PHP 5.3 “full stack”: Lithium, Symfony 2, Zend Framework 2, others
- Micro-frameworks: Glue, Limonade, Silex, Slim
  - Context, router+dispatcher, HTTP request/response, middleware

Frameworks: Good?

- Delivered as a monolithic whole
- Want to use just part of a framework? Difficult.
- Download entire framework and try to use one part ...
- ... except it's coupled to dependencies within the framework.
- Have to set up parts you don't care about.
- Components (kind of): Symfony 2, Zend Framework 2

Definitions of "Decoupled"

- *In formal design, decoupling means to make the features of a formal system as independent as possible from each other.* — Ioannis T. Kassios, http://link.springer.com/chapter/10.1007/11526841_5

- *Decoupling refers to careful controls that separate code modules from particular use cases, which increases code re-usability.* — https://en.wikipedia.org/wiki/Object-oriented_programming#Decoupling

- Cf. http://www.coldewey.com/publikationen/Decoupling.1.1.PDF

- **Framework-level** and **package-level** decoupling, not class-level

Zend Framework 2

```
install:
    zend-inputfilter

depends:
    zend-stdlib
    zend-servicemanager
    zend-filter
    zend-i18n
    zend-validator

suggest:
    pecl-weakref
    zendframework/zend-di
    zendframework/zend-crypt
    zendframework/zend-db
    zendframework/zend-math
```

Symfony 2

- *Symfony Components implement common features needed to develop websites. They are the foundation of the Symfony full-stack framework, but they can also be used standalone even if you don't use the framework as they don't have any mandatory dependencies.* — http://symfony.com/components

- *Symfony's claim … is* ***clearly true*** *for 11 of those packages …* ***clearly false*** *for 4 of them …* ***debatable for the remaining 6*** *…*
  — http://paul-m-jones.com/archives/4263

How To Tell

- Composer.json has "require" for another package? Not decoupled.
- Composer.json has "require-dev"? *Might* or *might not* be decoupled. (Allowance for interface packages.)
- "Optional behavior"? Still coupled. Could say the same for framework.

```
/** Symfony/Component/Routing/Loader/AnnotationClassLoader.php */
namespace Symfony\Component\Routing\Loader;

use Doctrine\Common\Annotations\Reader;
use Symfony\Component\Config\Resource\FileResource;
use Symfony\Component\Routing\Route;
use Symfony\Component\Routing\RouteCollection;
use Symfony\Component\Config\Loader\LoaderInterface;
use Symfony\Component\Config\Loader\LoaderResolverInterface;
```

Decoupled Components?

- Not dependent on **framework** but still dependent on **each other**
- Ed Finkler, "The Micro-PHP Manifesto" (microphp.org)

Principles of Decoupled Packages

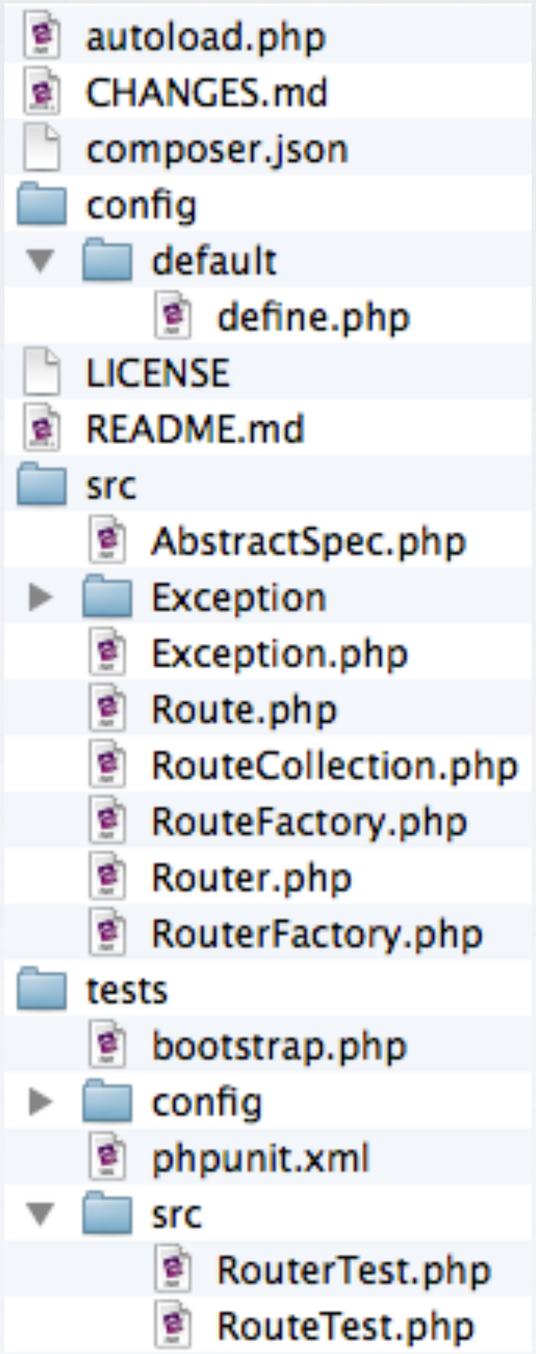
Rewrite Solar as Aura

- Solar Framework: 5+ years old at the time (Oct 2010)
- Monolithic; tough to use just parts of it
- Independent, decoupled library packages
- V1 (Oct 2010): extract components, PSR-0
- V2 (Sep 2013): split packages even further, PSR-4

Driving Principles (1)

- **Libraries first**, framework later
- **No use of globals** within packages (e.g., pass in $_SERVER)
- **No dependencies** on any other package
- Tests and assets **encapsulated** within package
- No “composer install" to run tests or get extra/optional functionality
- Each has its **own repository** (no subtree splits or extract-and-build)

Library Package Organization



Driving Principles (2): Dependency Injection

- Solar used static Service Locator and universal constructor
- In Aura, a shared Service Locator would mean a package dependency
- All packages are set up for **dependency injection**
- You can use any container you like (Aura.Di is nice ;-) or none
- Pass **Factory** objects instead of using `new` (reveals dependencies)

new Example

```
class Foo
{
    protected $db;
    public function __construct()
    {
        $this->db = new Database(...);
    }
}
```

Service Locator Examples

```
class Foo
{
    protected $db;
    public function __construct()
    {
        $this->db = Locator::get('db');
    }
}
```

```php
class Foo
{
    protected $db;
    public function __construct(Locator $locator)
    {
        $this->db = $locator->get('db');
    }
}
```

Dependency Injection Examples

```
class Foo
{
    protected $db;
    public function __construct(Database $db)
    {
        $this->db = $db;
    }
}
```

```php
class Foo
{
    protected $db;
    public function setDb(Database $db)
    {
        $this->db = $db;
    }
}
```

How To Tell Dependency Injection from Service Locator

- Both build and retain service objects: usage, not implementation
- Any DI container instance can be injected into an object
- Any SL container instance can be kept outside an object
- Providing static methods on `Container`: Service Locator
- `implements ContainerAware`: Service Locator
- `Container` parameter typehint: Service Locator

Dependency Injection > Service Locator

- Service Locator **hides dependencies**
- Service Locator **is itself a dependency** (all libraries need it)
- Harder to write tests for objects using the Service Locator
- DI **reveals** dependencies (especially with **Factories**)
- Is not itself a dependency
- Easier to write tests, fakes, etc.
- **Service Locator** instance, if needed, on a per-package basis

Aura.Router v2

Description

Provides a web router implementation: given a URL path and a copy of $_SERVER, it will extract path-info parameters and $_SERVER values for a specific route.

Instantiation

```php
require '/path/to/Aura.Router/autoload.php';

use Aura\Router\RouterFactory;
$router_factory = new RouterFactory;
$router = $router_factory->newInstance();

use Aura\Router\Router;
use Aura\Router\RouteCollection;
use Aura\Router\RouteFactory;
$router = new Router(new RouteCollection(new RouteFactory));
```

Adding Routes

```php
// add an unnamed route with params
$router->add(null, '/{controller}/{action}/{id}');

// add a named route with optional params
$router->add('archive', '/archive{/year,month,day}');

// add a named route with an extended specification
$router->add('read', '/blog/read/{id}{format}')
    ->addTokens(array(
        'id'     => '\d+',
        'format' => '(\.[^/]+)?',
    ))
    ->addValues(array(
        'controller' => 'blog',
        'action'     => 'read',
        'format'     => '.html',
    ));

// add a REST route: BREAD+CUR
$router->attachResource('users', '/users');
```

Matching Routes

```
// get the incoming request URI path
$path = parse_url($_SERVER['REQUEST_URI'], PHP_URL_PATH);

// get the route based on the path and server
$route = $router->match($path, $_SERVER);
```

The match() method does not parse the URI or use $_SERVER internally. This is because different systems may have different ways of representing that information; e.g., through a URI object or a context object. As long as you can pass the string path and a server array, you can use Aura.Router in your application foundation or framework.

Route Parameters

```php
$route = $router->match('/blog/read/42.json', $_SERVER);
var_export($route->params);

// shows these values:
[
    'controller' => 'blog',
    'action'     => 'read',
    'id'         => '42',
    'format'     => '.json',
]
```

Dispatching Routes

```php
$params = $route->params;
$class = ucfirst($params['controller']) . 'Page';
$method = $params['action'] . 'Action';
$object = new $class();
echo $object->$method($params);
```

Micro-Framework Route

```
$router->add('read', '/blog/read/{id}')
    ->addTokens(array(
        'id' => '\d+',
    ))
    ->addValues(array(
        'controller' => function ($params) {
            $id = (int) $params['id'];
            header('Content-Type: application/json');
            echo json_encode(['id' => $id]);
        },
));
```

Micro-Framework Dispatcher

```php
$controller = $route->params['controller'];
echo $controller($route->params);
```

Aura.Web (v1 to v2)

Old (v1) Description

Provides tools to build web page controllers, including an `AbstractPage` for action methods, a `Context` class for discovering the request environment, and a `Response` transfer object that describes the eventual HTTP response.

Instantiation and Calling

```php
use Vendor\Package\Web\Page;
use Aura\Web\Context;
use Aura\Web\Accept;
use Aura\Web\Response;
use Aura\Web\Signal;
use Aura\Web\Renderer\None as Renderer;

$params = [
    'action' => 'hello',
    'format' => '.html',
    'noun'   => 'world',
];

$page = new Page(
    new Context($GLOBALS),
    new Accept($_SERVER),
    new Response,
    new Signal,
    new Renderer,
    $params
);

$response = $page->exec();
```

Important Parts

- `$this->params` for incoming parameters
- `$this->context` for get, post, files, etc.
- `$this->accept` for content-type, language, encoding, etc
- `$this->response` for headers, cookies, content *(data transfer object)*
- `$this->signal` for signals/events/notifiers *(separated interface)*
- `$this->renderer` for rendering strategy (default "none")
- `$this->data` for data to be rendered
- `(pre|post)_(exec|action|render)` hooks, and `catch_exception` hook

Rendering Strategy

```php
class NaiveRenderer extends AbstractRenderer
{
    public function exec()
    {
        // get data from controller
        $data = (array) $this->controller->getData();

        // pick a template file based on controller action
        $action   = $this->controller->getAction();
        $__file__ = "/path/to/templates/{$action}.php";

        // closure to execute template file
        $template = function () use (array $data, $__file__) {
            ob_start();
            extract($data);
            require $__file__;
            return ob_get_clean();
        };

        // invoke closure
        $content = $template();

        // set content on response, and done!
        $response = $this->controller->getResponse();
        $response->setContent($content);
    }
}
```

Way Too Much In v1

- At first, all seemed to go together: base controller, page controller, action method dispatch, event signals, request, response, rendering
- Even with separated interfaces, all coupled to each other
- Extract Aura.Dispatcher from Aura.Web, Aura.Cli, front-controller
- No more need for "controller" or "rendering" code
- All that remains is request and response for your own controllers

New (v2) Description

Provides web Request and Response objects for use by web controllers. These are representations of the PHP web environment, not HTTP request and response objects proper.

Request Object

- *Not* an HTTP request, but a web execution context representation
- If you have `$_SESSION` or `$_ENV` in your request object, it's not HTTP
- Read superglobals, headers, cookies, negotiate accept values, etc.
- Read/write on "params"

Response Object

- *Not* an HTTP response, but a **Data Transfer Object**
- Must convert it to a real HTTP response (does not "send itself")
- Allows any HTTP library, or none

```php
// typical full-stack controller, dependency injection
class MyAppController
{
    public function __construct(
        Request $request,
        Response $response
    ) {
        $this->request = $request;
        $this->response = $response;
    }

    public function foo()
    {
        $bar = $this->request->post->get('bar');
        $this->response->content->setType('application/json');
    }
}

// typical micro-framework route+dispatch+logic, service locator
$app->addGet('/foo', function () use ($app) {
    $bar = $app->request->post->get('bar');
    $app->response->content->setType('application/json');
});
```

Delivery Code

```php
// send status line
header($response->status->get(), true, $response->status->getCode());

// send non-cookie headers
foreach ($response->headers->get() as $label => $value) {
    header("{$label}: {$value}");
}

// send cookies
foreach ($response->cookies->get() as $name => $cookie) {
    setcookie(
        $name,
        $cookie['value'],
        $cookie['expire'],
        $cookie['path'],
        $cookie['domain'],
        $cookie['secure'],
        $cookie['httponly']
    );
}

// send content
echo $response->content->get();
```

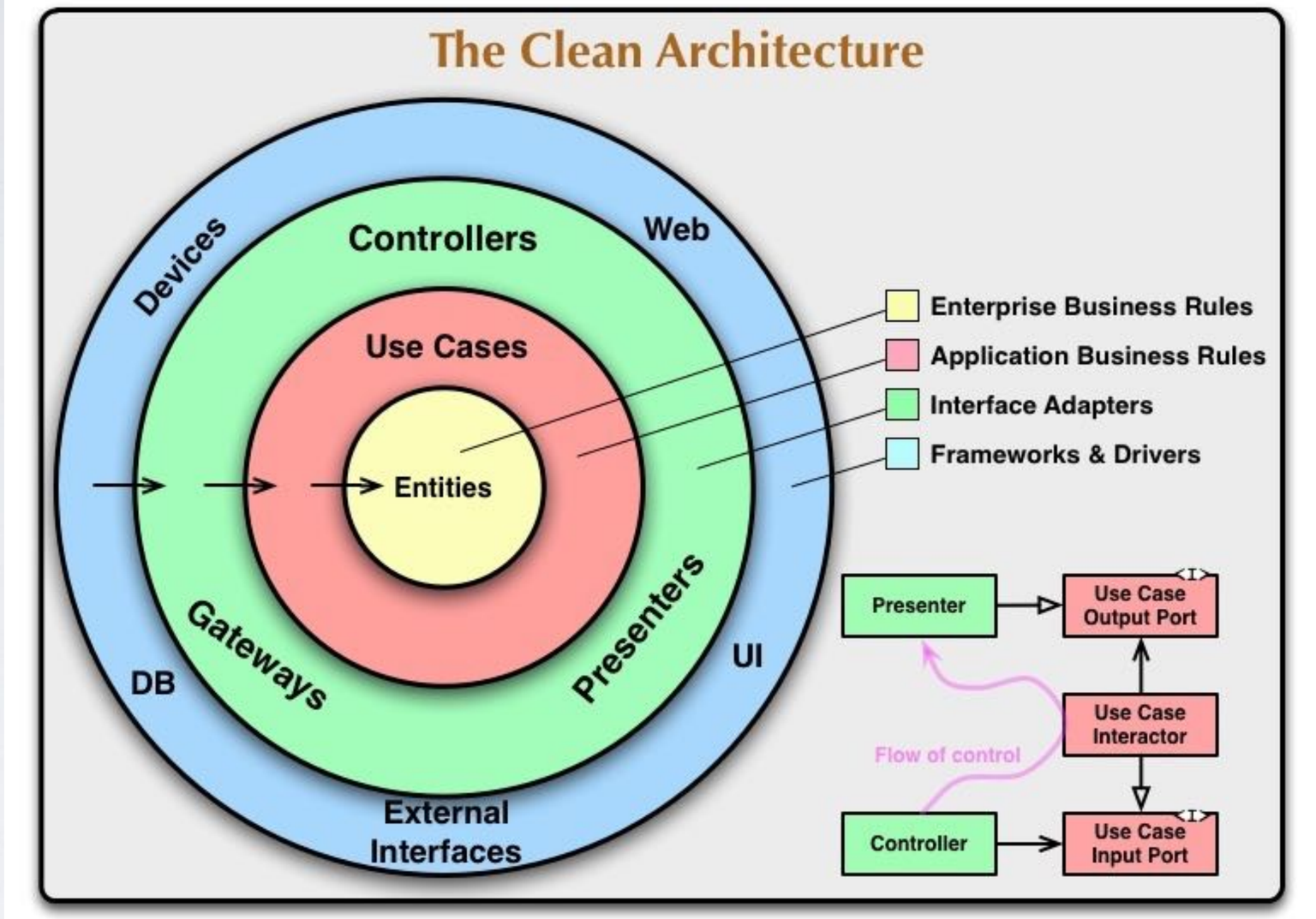
Limits To Decoupling

Library Packages

- **Aura.Autoload**
- **Aura.Cli**
- **Aura.Di**
- **Aura.Dispatcher**
- Aura.Filter
- **Aura.Html**
- Aura.Http
- **Aura.Includer**
- Aura.Input
- Aura.Intl
- Aura.Marshal
- **Aura.Router**
- Aura.Session
- Aura.Signal
- **Aura.Sql**
- **Aura.Sql_Query**
- **Aura.Sql_Schema**
- Aura.Uri
- **Aura.View**
- **Aura.Web**

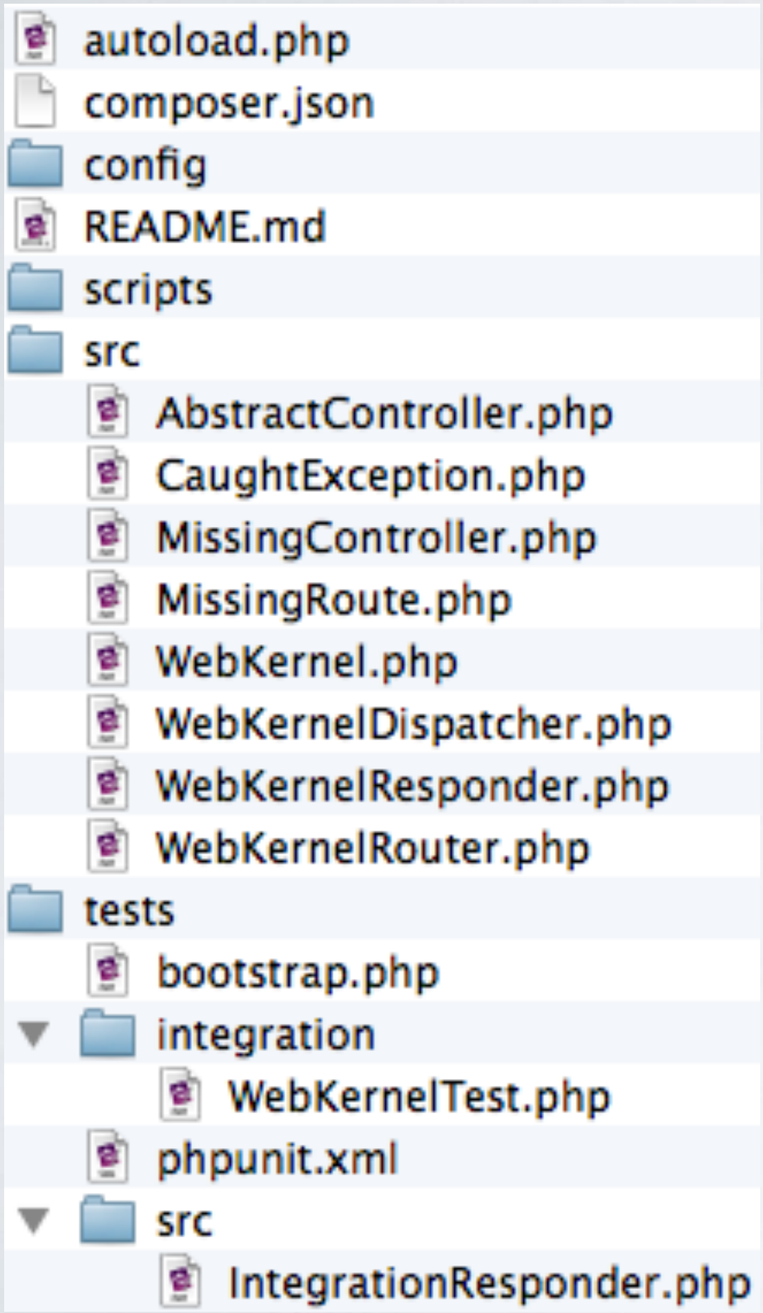
Combinations Mean Dependencies

- Front controller will need router, dispatcher, responder
- Action controller will need services, request/response
- Gateway services will need data source connection

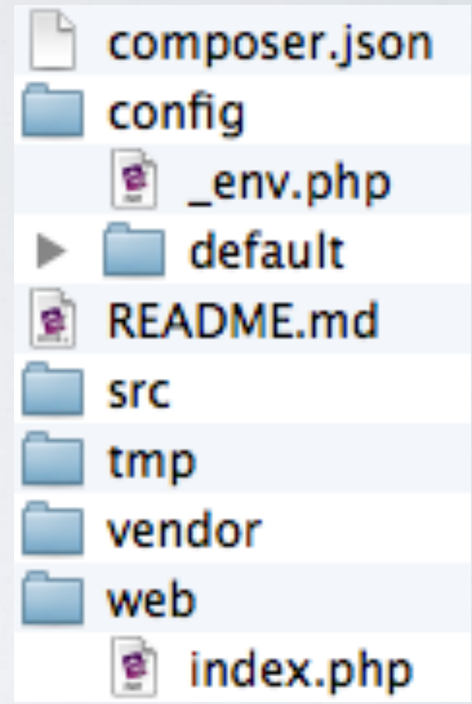


Clean Code (Robert C. Martin)

*_Kernel, *_Project



- "Kernel" of packages and glue
- "Project" (framework) skeleton
- `composer create-project`



Watch For Dependency/Coupling

- Controller should probably be independent of routing, dispatching, external base class (and vice versa)
- Service Locators are especially binding
- Micro-frameworks are antithesis of decoupling: closure is bound to routing, dispatching, locator (and probably middleware system)

Conclusion

- Background: Libraries, Frameworks, Components
- Principles of decoupled library packages
- Examples: individual Aura library packages
- Limits to decoupling, direction of dependencies

Modernizing Legacy Applications in PHP

Are you overwhelmed by a legacy application full of page scripts, spaghetti includes, and global variables? Do you want to improve the quality of the code, but don't know where to begin or how to proceed?

My new book, Modernizing Legacy Applications in PHP, gives step-by-step instructions on how to get your code under control and keep your application running the whole time.

http://mlaphp.com

Buy the early access version now and get free updates as it is completed, or sign up on the mailing list below for more information and a free sample chapter.

Thanks!

auraphp.com

paul-m-jones.com
@pmjones

mlaphp.com

joind.in/10510