21 DEC 2014

Onion Architecture and Zend Framework 2

"Onion Architecture" is probably the fanciest and the most discussed architectural design pattern proposed on the Internet these days. The term was first introduced by Jeffery Palermo back in 2008 in his series of blogs. The idea behind it is the separation of business logic from its dependencies by placing core modules in the middle and the rest of the code around it. It's been said a lot about it on the Internet and proposed many different implementations and directory structures. However, most of the proposed solutions are implemented in Java or ASP.Net/C#, and there's nothing worthwhile in PHP.

In this post, I want to highlight some important rules of the architecture and introduce the directory structure which I like to have in my PHP projects.

First of all, most of my projects are built with Zend Framework 2, and all the examples in this post will be based on this framework as well. I like Zend Framework 2, since it provides that kind of flexibility which I couldn't find in any other PHP frameworks, thanks to its modular architecture.

Core, Infrastructure, Di and Application

These are the most important concepts which I like to keep separate in my Zend Framework 2 projects.

– **Core** – this is the place where you should put all your business logic, entities, specifications and services which will be used in application layer. It doesn't know anything about the technologies are used in the application such as email sender, database engine or even the framework. Here, you just define the interfaces for things which should be handled by technologies.

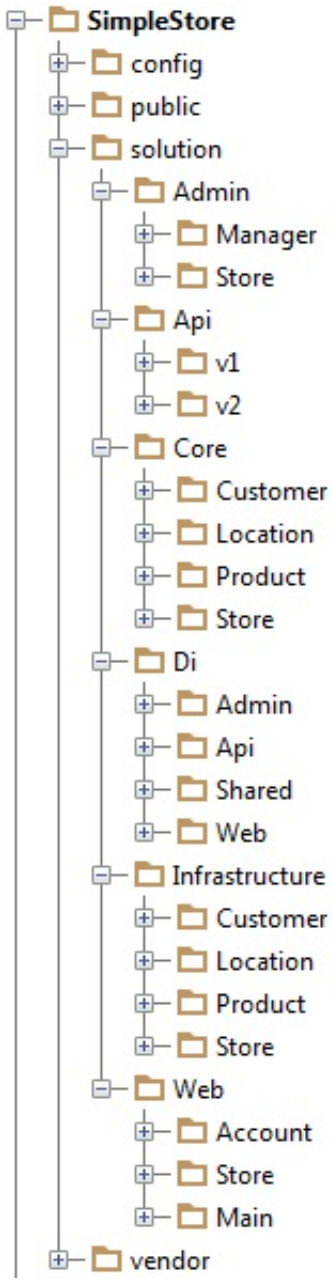
– **Infrastructure** – this is the place where you should put the code which is not allowed in core layer. Here, I usually implement interfaces which defined in core layer. For instance, repository interfaces or email sender interface.

– **Di** – this the place where the things from core layer and infrastructure layer are

put together in order to serve your application needs. Different applications may have different strategies of putting the things together. So, it is a common practice in my projects to have separate modules for each application.

– **Application** - this the place where you call services defined in core layer. Application layer doesn't know anything about infrastructure layer though, so you should not call infrastructure services directly from this layer. It is possible to have many applications using the same infrastructure and core layer. For instance, we can have a website for the customers to buy goods, the admin panel to allow managers control the website, and the web services to expose APIs for the mobile apps. All these applications utilize the same infrastructure layer and core layer.

Directory structure



Tot

2

2

2

1

1

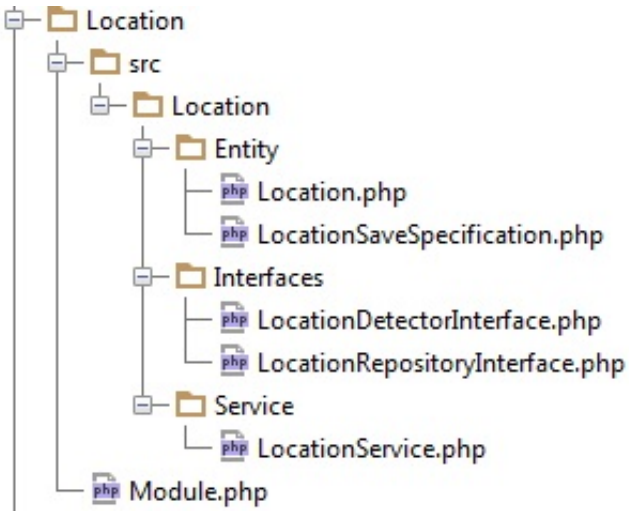
This is how the directory structure looks like in Zend Framework 2 if we follow Onion Architecture. You may noticed that there's no "module" folder in the above example as we accustomed to see in the normal ZF2 application, but instead, we got "solution" folder which contains bunch of other folders which actually hold the expected modules. The reason behind it is a pretty much explainable.

The directory structure of "Onion Architecture" project based on layers where each layer holds a number of modules. With this in mind, each folder inside "solution" folder can be considered as a single layer which together with other layers create a single solution. However, it doesn't mean that all the layers in the solution are run together every time the application serves the request.

A single solution can have multiple applications. In the above example, we have three applications such as "Web", "Admin" and "Api". Each application is considered as a single entry point and each application may require different modules even though core, infrastructure and di layers are common layers for any application in the solution.

Core

I came up with the following default directory structure for my core modules



While developing applications based on "Onion Architecture" I have noticed that all my core modules contain at least these three directories:

– **Entity** – this is the place where you put all the entity classes which related to the same module. Moreover, I found that sometimes I can put here also the classes which are directly related to the entities like for example, specifications or finders/filters.

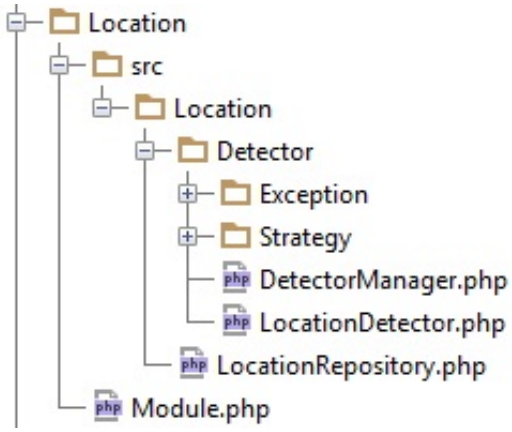
– **Interfaces** – this is the place where you put all the public interfaces which are expected to be implemented in infrastructure layer. For

instance, `LocationRepositoryInterface` or `LocationDetectorInterface` as in the above example.

– **Service** – this is the place where you put all the services which will be consumed in application layer.

Infrastructure

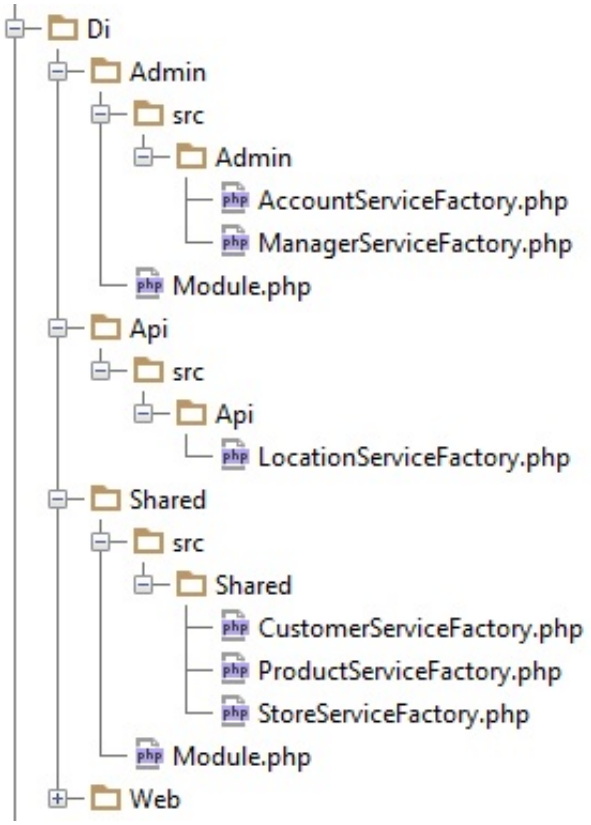
I found that the best way to organize the modules in infrastructure layer is to reflect the modules from core layer. However, using the same module structure in infrastructure layer doesn't make sense though. I usually don't bother too much about it, so I put all the classes into the root directory of the module. If the implementation requires many classes, I put them into a separate directory as in the example below:



Di

The modules in di layer usually reflect the applications in the solution. Each module represents a separate application and stores service factories which implemented in core layer and which are used only by the represented application. However, most of the service factories are suitable to any other applications in the solution. Therefore, I usually have a common module called "Shared" which contain shared service factories and other classes for all applications. The idea behind it is to have more control over what to include and what to not include in the specific application.

This is how the basic di layer structure would look like:



As you can see, in the common module we have factories for services `CustomerService`, `ProductService` and `StoreService`. These services intend to be consumed in all applications in the project, so that's why they are in "Shared" module. We also have some factories in "Admin" module and in "Api" module. Those services will be used only in the respective applications "Admin" and "Api".

"Product" module

So far, I have shown the basic directory structure applied in each layer and tried to explain the purpose of some folders. In this section, I would like to demonstrate how the module "Product" can be implemented and consumed in each layer by following "Onion Architecture" principles.

Don't forget to tell Zend Framework 2 to look for the modules in "solution" folder

```
return  [
    'module_listener_options' => [
        'module_paths' => [
            './vendor',
            'SimpleStore\\*' => './solution'
        ],

        'config_glob_paths' => [
            'config/autoload/{,*.}{global,local}.php',
        ]
    ]
];
```

Let's begin from core layer by creating "Product" entity

```php
namespace SimpleStore\Core\Product\Entity;

class Product
{
    private $id;
    private $title;
    private $price;

    public function setId($id)
    {
        $this->id = $id;
    }

    public function getId()
    {
        return $this->id;
    }

    public function setTitle($title)
    {
        $this->title = $title;
    }

    public function getTitle()
    {
        return $this->title;
    }

    public function setPrice($price)
    {
        $this->price = $price;
    }

    public  function getPrice()
    {
        return $this->price;
    }
}
```

Now, we want our product(s) be retrieved from the database, but we cannot implement this in core layer, since it doesn't know anything about the database we use. So, let's create `ProductRepositoryInterface` and leave its implementation for later in infrastructure layer.

```php
namespace SimpleStore\Core\Product\Interfaces;

interface ProductRepositoryInterface
{
    public function loadAllProducts();
    public function loadProductById($id);
}
```

Let's do a bit more and create `ProductService` to let our application layer consume "Product" module.

```php
namespace SimpleStore\Core\Product\Service;
use SimpleStore\Core\Product\Interfaces\ProductRepositoryInter

class ProductService
```

```
5  {
6      private $repository;
7
8      public  function __construct(ProductRepositoryInterface $r
9      {
10         $this->repository = $repository;
11     }
12
13     public function getAllProducts()
14     {
15         return $this->repository->loadAllProducts();
16     }
17
18     public function getProductById($id)
19     {
20         return $this->repository->loadProductById($id);
21     }
22 }
```

Worth to note here that `ProductService` depends on the repository interface which will be implemented in infrastructure layer and injected in di layer later.

Next, let's create `ProductRepository` in infrastructure layer which will implement `ProductRepositoryInterface`.

```
1  namespace SimpleStore\Infrastructure\Product;
2  use SimpleStore\Core\Product\Interfaces\ProductRepositoryInter
3
4  class ProductRepository implements ProductRepositoryInterface
5  {
6      public function loadAllProducts()
7      {
8          // TODO: Implement loadAllProducts() method.
9      }
10
11     public function loadProductById()
12     {
13         // TODO: Implement loadProductById() method.
14     }
15 }
```

It's time to put the things together.

Let's create `ProductServiceFactory` in di layer which will create `ProductService` and inject `ProductRepository` into it.

```
1  namespace SimpleStore\Di\Shared;
2  use Zend\ServiceManager\ServiceLocatorInterface;
3  use SimpleStore\Core\Product\Service\ProductService;
4  use SimpleStore\Infrastructure\Product\ProductRepository;
5
6  class ProductServiceFactory implements FactoryInterface
7  {
8      public function createService(ServiceLocatorInterface $ser
9      {
10         return new ProductService(new ProductRepository());
```

```
11          }
12      }
```

Finally, we can now consume "ProductService" in application layer

```
1   namespace SimpleStore\Web\Product\Controller;
2   use Zend\Mvc\Controller\AbstractActionController;
3
4   class ProductController extends AbstractActionController
5   {
6       public function indexAction()
7       {
8           return [
9               'products' => $this->getServiceLocator()->get('Pro
10          ];
11      }
12  }
```

I hope you like this post. If you have any question related to this topic feel free to ask them in the comments area below. I will be also happy to hear any feedback regarding this post or other posts in my blog.

Happy coding!

Let me know if you like it...

3

You may also like

- Setting a custom layout for error pages in Zend Framework 2
- Creating a custom template injector to deal with sub-namespaces in Zend Framework 2
- Controlling access to web pages with ChkAccess in Zend Framework 2
- Setting up multiple entry points for your Zend Framework 2 project
- Advanced "Hydrator" usage in Zend Framework 2

Igor Vorobiov Posted in Zend Framework 2 0 Comments

← Speech recognition web service for iOS apps built with Google Speech API v2 and PHP

Setting up multiple entry points for your Zend Framework 2 project →

ALSO ON **IGOR VOROBIOV | BLOG**

Setting a custom layout for error pages in Zend Framework 2

2 comments • 2 years ago

ivorobioff — This is a standard view manager configuration. Have you seen this thishttp://framework.zend.com/... ? I

Setting up multiple entry points for your Zend Framework 2 project

2 comments • a year ago

manuakasam — Multiple entry points like that may not be needed if you simply set the environment variable on your host

Speech recognition web service for iOS apps built with Google Speech API v2

1 comment • a year ago

S1m0n3 — really good trick (about Chromium-dev group) Thank u so much! =)

Advanced “Hydrator” usage in Zend Framework 2

2 comments • a year ago

ivorobioff — Hello Dhruv, Thanks for visiting my blog :) yeah, I will be glad to help you out. So, you have to create