PPI Documentation

Release 2.0.0

PPI Team

May 07, 2015

Contents

PPI is an open source PHP meta-framework.

We have taken the good bits from Symfony2, ZendFramework2 & Doctrine2 and combined them together to create a solid and very easy web application framework. PPI can be considered the boilerplate of PHP frameworks.

CHAPTER 1

The Book

In 7 short chapters (or less!) learn how to use PPI2 for your web projects.

1.1 The Book

1.1.1 Getting Started

Downloading PPI

You can download the ppi skeletonaapp from the Homepage. If you just want everything in one folder ready to go, you should choose the **"ppi skeletonapp with vendors"** option.

If you are comfortable with using **git** then you can download the **"skeleton app without vendors"** option and run the following commands:

```
curl -s http://getcomposer.org/installer | php
php composer.phar install
mkdir app/cache && chmod -R 777 app/cache
```

Production Apache Configuration

We take **security** very seriously, so all your app code and configuration is kept hidden away outside of /public/ and is inaccesible via the browser, because of that we need to create a virtual host in order to route all web requests to the /public/ folder and from there your public assets (css/js/images) are loaded normally and the .htaccess rule kicks in to route all non-asset files to /public/index.php.

```
<VirtualHost *:80>
    DocumentRoot /var/www/ppiapplication/public
    ServerName www.myppiwebsite.com
    RewriteEngine On
    ErrorLog /var/log/apache2/error.log
    <Directory "/var/www/ppiapplication/public">
        AllowOverride All
        Options +Indexes +FollowSymLinks
    </Directory>
</VirtualHost>
```

System requirements

A web server with its rewrite module enabled. (mod_rewrite for apache)

PPI needs **5.3.23** or above as required by zend framework 2.

1.1.2 Skeleton Application

First, lets review the file structure of the PPI skeleton application that we have pre-built for you to get up and running as quickly as possible.:

```
www/ <- your web root directory

skeleton/ <- the unpacked archive
    app/
        app.config.php
        cache/
        views/
        ...

    public/
        index.php
        css/
        js/
        images/
        ...

    modules/
        Application/
            Module.php
            Controller/
            resources/
                config/
                views/
                ...
```

Lets break it down into parts:

The public folder

The structure above shows you the `/public/` folder. Anything outside of `/public/` i.e: all your business code will be secure from direct URL access. In your development environment you don't need a virtualhost file, you can directly access your application like so: http://localhost/skeleton/public/. In your production environment this will be http://www.mysite.com/. All your publicly available asset files should be here, CSS, JS, Images.

The public index.php file

The /public/index.php is also known are your bootstrap file, or front controller is explained in-depth below

```php
<?php

// All relative paths start from the main directory, not from /public/
chdir(dirname(__DIR__));

// Lets include PPI
```

```php
include('app/init.php');

// Initialise our PPI App
$app = new PPI\App();
$app->moduleConfig = include 'app/modules.config.php';
$app->config = include 'app/app.config.php';

// If you are using the DataSource component, enable this
//$app->useDataSource = true;

$app->boot()->dispatch();
```

If you uncomment the `useDataSource` line, it is going to look for your `/app/datasource.config.php` and load that into the DataSource component for you. Databases are not a requirement in PPI so if you dont need one then you wont need to bother about this. More in-depth documentation about this in the DataSource chapter.

The app folder

This is where all your apps global items go such as app config, datasource config and modules config and global templates (views). You wont need to touch these out-of-the-box but it allows for greater flexibility in the future if you need it.

The app.config.php file

Looking at the example config file below, you can control things here such as the environment, templating engine and datasource connection.

```php
<?php
$config = array(
    'environment' => 'development', // <-- Change this depending on your environment
    'templating.engine' => 'php', // <-- The default templating engine
    'datasource.connections' => include (__DIR__ . '/datasource.config.php')
);

// Are we in debug mode ?
if($config['environment'] !== 'development') { // <-- You can also check the env from your controlle
    $this->getEnv()
    $config['debug'] = $config['environment'] === 'development';
    $config['cache_dir'] = __DIR__ . '/cache';
}

return $config; // Very important
```

The `return $config` line gets pulled into your `index.php`'s `$app->config` variable.

The modules.config.php file

The example below shows that you can control which modules are active and a list of directories module_paths that PPI will scan for your modules. Pay close attention to the order in which your modules are loaded. If one of your modules relies on resources loaded by another module. Make sure the module loading the resources is loaded before the others that depend upon it.

```php
<?php
return array(
    'activeModules'    => array('Application', 'User'),
```

```
    'listenerOptions' => array('module_paths' => array('./modules')),
);
```

Note that this file returns an array too, which is assigned against your `index.php`'s $app->moduleConfig variable

The app/views folder

This folder is your applications global views folder. A global view is one that a multitude of other module views extend from. A good example of this is your website's template file. The following is an example of `/app/views/base.html.php`:

```
<html>
    <body>
        <h1>My website</h1>
        <div class="content">
            <?php $view['slots']->output('_content'); ?>
        </div>
    </body>
</html>
```

You'll notice later on in the Templating section to reference and extend a global template file, you will use the following syntax in your modules template.

```
<?php $view->extend('::base.html.php'); ?>
```

Now everything from your module template will be applied into your base.html.php files _content section demonstrated above.

The modules folder

This is where we get stuck into the real details, we're going into the `/modules/` folder. Click the next section to proceed

1.1.3 Modules

By default, one module is provided with the SkeletonApp, named **Application**. It provides a simple route pointing to the homepage. A simple controller to handle the "home" page of the application. This demonstrates using routes, controllers and views within your module.

Module Structure

Your module starts with Module.php. You can have configuration on your module. Your can have routes which result in controllers getting dispatched. Your controllers can render view templates.

```
modules/

    Application/

        Module.php

        Controller/
            Index.php

        resources/
```

```
            views/
                index/index.html.php
                index/list.html.php

            config/
                config.php
                routes.yml
```

The Module.php class

Every PPI module looks for a `Module.php` class file, this is the starting point for your module.

```
<?php
namespace Application;

use PPI\Module\Module as BaseModule
use PPI\Autoload;

class Module extends BaseModule {

    protected $_moduleName = 'Application';

    function init($e) {
        Autoload::add(__NAMESPACE__, dirname(__DIR__));
    }

}
```

Init

The above code shows you the Module class, and the all important `init()` method. Why is it important? If you remember from The Skeleton Application section previously, we have defined in our `modules.config.php` config file an activeModules option, when PPI is booting up the modules defined activeModules it looks for each module's init() method and calls it.

The `init()` method is run for every page request, and should not perform anything heavy. It is considered bad practice to utilize these methods for setting up or configuring instances of application resources such as a database connection, application logger, or mailer.

Your modules resources

`/Application/resources/` is where non-PHP-class files live such as config files (`resources/config`) and views (`resources/views`). We encourage you to put your own custom config files in `/resources/config/` too.

Configuration

Expanding on from the previous code example, we're now adding a `getConfig()` method. This must return a raw php array. All the modules with getConfig() defined on them will be merged together to create 'modules config' and this is merged with your global app's configuration file at `/app/app.config.php`. Now from any controller you can get access to this config by doing `$this->getConfig()`. More examples on this later in the Controllers section.

```php
<?php
class Module extends BaseModule {

protected $_moduleName = 'Application';

    public function init($e) {
        Autoload::add(__NAMESPACE__, dirname(__DIR__));
    }

    public function getConfig() {
        return include(__DIR__ . '/resources/config/config.php');
    }

}
```

Routing

The getRoutes() method currently is re-using the Symfony2 routing component. It needs to return a Symfony RouteCollection instance. This means you can setup your routes using PHP, YAML or XML.

```php
class Module extends BaseModule {

    protected $_moduleName = 'Application';

    public function init($e) {
        Autoload::add(__NAMESPACE__, dirname(__DIR__));
    }

    /**
    * Get the configuration for this module
    *
    * @return array
    */
    public function getConfig() {
        return include(__DIR__ . '/resources/config/config.php');
    }

    /**
    * Get the routes for this module, in YAML format.
    *
    * @return \Symfony\Component\Routing\RouteCollection
    */
    public function getRoutes() {
        return $this->loadYamlRoutes(__DIR__ . '/resources/config/routes.yml');
    }

}
```

Conclusion

So, what have we learnt in this section so far? We learnt how to initialize our module, and how to obtain configuration options and routes from it.

PPI will boot up all the modules and call the `getRoutes()` method on them all. It will merge the results together and match them against a request URI such as `/blog/my-blog-title`. When a matching route is found it dispatches the controller specified in that route.

Lets move onto the Routing section to check out what happens next.

1.1.4 Routing

Routes are the rules that tell the framework what URLs map to what area of your application. The routing here is simple and expressive. We are using the Symfony2 routing component here, this means if you're a Symfony2 developer you already know what you're doing. If you don't know Symfony2 already, then learning the routes here will allow you to read routes from existing Symfony2 bundles out there in the wild. It's really a win/win situation.

Routes are an integral part of web application development. They make way for nice clean urls such as `/blog/view/5543` instead of something like `/blog.php?Action=view&article=5543`.

By reading this routing section you'll be able to:

- Create beautiful clean routes
- Create routes that take in different parameters
- Specify complex requirements for your parameters
- Generate URLs within your controllers
- Redirect to routes within your controllers

The Details

Lets talk about the structure of a route, you have a route name, pattern, defaults and requirements.

Name

This is a symbolic name to easily refer to this actual from different contexts in your application. Examples of route names are `Homepage`, `Blog_View`, `Profile_Edit`. These are extremely useful if you want to just redirect a user to a page like the login page, you can redirect them to User_Login. If you are in a template file and want to generate a link you can refer to the route name and it will be created for you. The good part about this is you can maintain the routes via your `routes.yml` file and your entire system updates.

Pattern

This is the URI pattern that if present will activate your route. In this example we're targeting the homepage. This is where you can specify params like `{id}` or `{username}`. You can make URLs like `/article/{id}` or `/profile/{username}`.

Defaults

This is the important part, The syntax is `Module:Controller:action`. So if you specify `Application:Blog:show` then this will execute the following class path: `/modules/Application/Controller/Blog->showAction()`. Notice how the method has a suffix of Action, this is so you can have lots of methods on your controller but only the ones ending in `Action()` will be executable from a route.

Requirements

This is where you can specify things like the request method being POST or PUT. You can also specify rules for the parameters you created above in the pattern section. Such as `{id}` being numeric, or `{lang}` being in a whitelist of values such as `en|de|pt`.

With all this knowledge in mind, take a look at all the different examples of routes below and come back up here for reference.

Basic Routes

```
Homepage:
pattern: /
defaults: { _controller: "Application:Index:index"}

Blog_Index:
pattern: /blog
defaults: { _controller: "Application:Blog:index"}
```

Routes with parameters

The following example is basically `/blog/*` where the wildcard is the value given to title. If the URL was `/blog/using-ppi2` then the title variable gets the value `using-ppi2`, which you can see being used in the Example Controller section below.

```
Blog_Show:
pattern: /blog/{title}
defaults: { _controller: "Application:Blog:show"}
```

This example optionally looks for the `{pageNum}` parameter, if it's not found it defaults to `1`.

```
Blog_Show:
pattern: /blog/{pageNum}
defaults: { _controller: "Application:Blog:index", pageNum: 1}
```

Routes with requirements

Only form submits using `POST` will trigger this route. This means you dont have to check this kind of stuff in your controller.

```
Blog_EditSave:
pattern: /blog/edit/{id}
defaults: { _controller: "Application:Blog:edit"}
requirements:
    _method: POST
```

Checking if the `{pageNum}` parameter is numerical. Checking if the `{lang}` parameter is `en` or `de`.

```
Blog_Show:
pattern: /blog/{lang}/{pageNum}
defaults: { _controller: "Application:Blog:index", pageNum: 1, lang: en}
requirements:
    id: \d+
    lang: en|de
```

Checking if the page is a `POST` request, and that `{id}` is numerical.

```
Blog_EditSave:
pattern: /blog/edit/{id}
defaults: { _controller: "Application:Blog:edit"}
requirements:
    _method: POST
    id: \d+
```

1.1.5 Controllers

So what is a controller? A controller is just a PHP class, like any other that you've created before, but the intention of it, is to have a bunch of methods on it called actions. The idea is: each route in your system will execute an action method. Examples of action methods would be your homepage or blog post page. The job of a controller is to perform a bunch of code and respond with some HTTP content to be sent back to the browser. The response could be a HTML page, a JSON array, XML document or to redirect somewhere. Controllers in PPI are ideal for making anything from web services, to web applications, to just simple html-driven websites.

Lets quote something we said in the last chapter's introduction section

Defaults

This is the important part, The syntax is `Module:Controller:action`. So if you specify Application:Blog:show then this will execute the following class path: `/modules/Application/Controller/Blog->showAction()`. Notice how the method has a suffix of Action, this is so you can have lots of methods on your controller but only the ones ending in `Action()` will be executable from a route.

Example controller

Review the following route that we'll be matching.

```
Blog_Show:
    pattern: /blog/{id}
    defaults: { _controller: "Application:Blog:show"}
```

So lets presume the route is `/blog/show/{id}`, and look at what your controller would look like. Here is an example blog controller, based on some of the routes provided above.

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function showAction() {

        $blogID = $this->getRouteParam('id');

        $bs = $this->getBlogStorage();

        if(!$bs->existsByID($blogID)) {
            $this->setFlash('error', 'Invalid Blog ID');
            return $this->redirectToRoute('Blog_Index');
```

```
        }

        // Get the blog post for this ID
        $blogPost = $bs->getByID($blogID);

        // Render our main blog page, passing in our $blogPost article to be rendered
        $this->render('Application:blog:show.html.php', compact('blogPost'));
    }

}
```

Generating urls using routes

Here we are still executing the same route, but making up some urls using route names

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function showAction() {

        $blogID = $this->getRouteParam('id');

        // pattern: /about
        $aboutUrl = $this->generateUrl('About_Page');

        // pattern: /blog/show/{id}
        $blogPostUrl = $this->generateUrl('Blog_Post', array('id' => $blogID));

    }
}
```

Redirecting to routes

An extremely handy way to send your users around your application is redirect them to a specific route.

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function showAction() {

        // Send user to /login, if they are not logged in
        if(!$this->isLoggedIn()) {
            return $this->redirectToRoute('User_Login');
        }

        // go to /user/profile/{username}
        return $this->redirectToRoute('User_Profile', array('username' => 'ppi_user'));
```

```
    }
}
```

Working with POST values

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function postAction() {

        $this->getPost()->set('myKey', 'myValue');

        var_dump($this->getPost()->get('myKey')); // string('myValue')

        var_dump($this->getPost()->has('myKey')); // bool(true)

        var_dump($this->getPost()->remove('myKey'));
        var_dump($this->getPost()->has('myKey')); // bool(false)

        // To get all the post values
        $postValues = $this->post();

    }
}
```

Working with QueryString parameters

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {


    // The URL is /blog/?action=show&id=453221
    public function queryStringAction() {

        var_dump($this->getQueryString()->get('action')); // string('show')
        var_dump($this->getQueryString()->has('id')); // bool(true)

        // Get all the query string values
        $allValues = $this->queryString();

    }
}
```

Working with server variables

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function serverAction() {

        $this->getServer()->set('myKey', 'myValue');

        var_dump($this->getServer()->get('myKey')); // string('myValue')

        var_dump($this->getServer()->has('myKey')); // bool(true)

        var_dump($this->getServer()->remove('myKey'));
        var_dump($this->getServer()->has('myKey')); // bool(false)

        // Get all server values
        $allServerValues =  $this->server();

    }
}
```

Working with cookies

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function cookieAction() {

        $this->getCookie()->set('myKey', 'myValue');

        var_dump($this->getCookie()->get('myKey')); // string('myValue')

        var_dump($this->getCookie()->has('myKey')); // bool(true)

        var_dump($this->getCookie()->remove('myKey'));
        var_dump($this->getCookie()->has('myKey')); // bool(false)

        // Get all the cookies
        $cookies = $this->cookies();

    }
}
```

Working with session values

```php
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function sessionAction() {

        $this->getSession()->set('myKey', 'myValue');

        var_dump($this->getSession()->get('myKey')); // string('myValue')

        var_dump($this->getSession()->has('myKey')); // bool(true)

        var_dump($this->getSession()->remove('myKey'));
        var_dump($this->getSession()->has('myKey')); // bool(false)

        // Get all the session values
        $allSessionValues = $this->session();

    }
}
```

Working with the config

Using the `getConfig()` method we can obtain the config array. This config array is the result of ALL the configs returned from all the modules, merged with your application's global config.

```php
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function configAction() {

        $config = $this->getConfig();

        switch($config['mailer']) {

            case 'swift':
                break;

            case 'sendgrid':
                break;

            case 'mailchimp':
                break;

        }
    }
}
```

Working with the is() method

The `is()` method is a very expressive way of coding and has a variety of options you can send to it. The method always returns a boolean as you are saying "is this the case?"

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function isAction() {

        if($this->is('ajax')) {}

        if($this->is('post') {}
        if($this->is('patch') {}
        if($this->is('put') {}
        if($this->is('delete') {}

        // ssl, https, secure: are all the same thing
        if($this->is('ssl') {}
        if($this->is('https') {}
        if($this->is('secure') {}

    }
}
```

Getting the users IP or UserAgent

Getting the user's IP address or user agent is very trivial.

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function userAction() {

        $userIP = $this->getIP();
        $userAgent = $this->getUserAgent();
    }
}
```

Working with flash messages

A flash message is a notification that the user will see on the next page that is rendered. It's basically a setting stored in the session so when the user hits the next designated page it will display the message, and then disappear from the session. Flash messages in PPI have different types. These types can be `'error'`, `'warning'`, `'success'`, this will determine the color or styling applied to it. For a success message you'll see a positive green message and for an error you'll see a negative red message.

Review the following action, it is used to delete a blog item and you'll see a different flash message depending on the scenario.

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function deleteAction() {

        $blogID = $this->getPost()->get('blogID');

        if(empty($blogID)) {
            $this->setFlash('error', 'Invalid BlogID Specified');
            return $this->redirectToRoute('Blog_Index');
        }

        $bs = $this->getBlogStorage();

        if(!$bs->existsByID($blogID)) {
            $this->setFlash('error', 'This blog ID does not exist');
            return $this->redirectToRoute('Blog_Index');
        }

        $bs->deleteByID($blogID);
        $this->setFlash('success', 'Your blog post has been deleted');
        return $this->redirectToRoute('Blog_Index');
    }
}
```

Getting the current environment

You may want to perform different scenarios based on the site's environment. This is a configuration value defined in your global application config. The `getEnv()` method is how it's obtained.

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function envAction() {

        $env = $this->getEnv();
        switch($env) {
            case 'development':
                break;

            case 'staging':
                break;

            case 'production':
            default:
                break;
```

```
        }
    }
}
```

1.1.6 Templating

As discovered in the previous chapter, a controller's job is to process each HTTP request that hits your web application. Once your controller has finished its processing it usually wants to generate some output content. To achieve this it hands over responsibility to the templating engine. The templating engine will load up the template file you tell it to, and then generate the output you want, his can be in the form of a redirect, HTML webpage output, XML, CSV, JSON; you get the picture!

In this chapter you'll learn:

- How to create a base template
- How to load templates from your controller
- How to pass data into templates
- How to extend a parent template
- How to use template helpers

Base Templates

What are base templates?

Why do we need base templates? well you don't want to have to repeat HTML over and over again and perform repetative steps for every different type of page you have. There's usually some commonalities between the templates and this commonality is your base template. The part that's usually different is the content page of your webpage, such as a users profile or a blog post.

So lets see an example of what we call a base template, or somethings referred to as a master template. This is all the HTML structure of your webpage including headers and footers, and the part that'll change will be everything inside the page-content section.

Where are they stored?

Base templates are stored in the `./app/views/` directory. You can have as many base templates as you like in there.

This file is `./app/views/base.html.php`

Example base template:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <div id="header">...</div>
        <div id="page-content">
            <?php $view['slots']->output('_content'); ?>
        </div>
        <div id="footer">...</div>
    </body>
</html>
```

Lets recap a little, you see that slots helper outputting something called _content? Well this is actually injecting the resulting output of the CHILD template belonging to this base template. Yes that means we have child templates that extend parent/base templates. This is where things get interesting! Keep on reading.

Extending Base Templates

On our first line we extend the base template we want. You can extend literally any template you like by specifying its `Module:folder:file.format.engine` naming syntax. If you miss out the Module and folder sections, such as `::base.html.php` then it's going to take the global route of `./app/views/`.

```
<?php $view->extend('::base.html.php'); ?>
<div class="user-registration-page">
    <h1>Register for our site</h1>
    <form>...</form>
</div>
```

The resulting output

If you remember that the extend call is really just populating a slots section named _content then the injected content into the parent template looks like this.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <div id="header">...</div>
        <div id="page-content">

            <div class="user-registration-page">
                <h1>Register for our site</h1>
                <form>...</form>
            </div>

        </div>
        <div id="footer">...</div>
    </body>
</html>
```

Example scenario

Consider the following scenario. We have the route `Blog_Show` which executes the action `Application:Blog:show`. We then load up a template named `Application:blog:show.html.php` which is designed to show the user their blog post.

The route

```
Blog_Show:
    pattern: /blog/{id}
    defaults: { _controller: "Application:Blog:show"}
```

The controller

```
<?php
namespace Application\Controller;

use Application\Controller\Shared as BaseController;

class Blog extends BaseController {

    public function showAction() {

        $blogID = $this->getRouteParam('id');
        $bs     = $this->getBlogStorage();

        if(!$bs->existsByID($blogID)) {
            $this->setFlash('error', 'Invalid Blog ID');
            return $this->redirectToRoute('Blog_Index');
        }

        // Get the blog post for this ID
        $blogPost = $bs->getByID($blogID);

        // Render our blog post page, passing in our $blogPost article to be rendered
        $this->render('Application:blog:show.html.php', compact('blogPost'));
    }
}
```

The template

So the name of the template loaded is Application:blog:show.html.php then this is going to translate to `./modules/Application/blog/show.html.php`. We also passed in a `$blogPost` variable which can be used locally within the template that you'll see below.

```
<?php $view->extend('::base.html.php'); ?>

<div class="blog-post-page">
    <h1><?=$blogPost->getTitle();?></h1>
    <p class="created-by"><?=$blogPost->getCreatedBy();?></p>
    <p class="content"><?=$blogPost->getContent();?></p>
</div>
```

Using the slots helper

We have a bunch of template helpers available to you, the helpers are stored in the $view variable, such as `$view['slots']` or `$view['assets']`. So what is the purpose of using slots? Well they're really for segmenting the templates up into named sections and this allows the child templates to specify content that the parent is going to inject for them.

Review this example it shows a few examples of using the slots helper for various different reasons.

The base template

```
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <title><?php $view['slots']->output('title', 'PPI Skeleton Application') ?></title>
    </head>
    <body>
        <div id="page-content">
            <?php $view['slots']->output('_content') ?>
        </div>
    </body>
</html>
```

The child template

```
<?php $view->extend('::base.html.php'); ?>

<div class="blog-post-page">
    <h1><?=$blogPost->getTitle();?></h1>
    <p class="created-by"><?=$blogPost->getCreatedBy();?></p>
    <p class="content"><?=$blogPost->getContent();?></p>
</div>

<?php $view['slots']->start('title'); ?>
Welcome to the blog page
<?php $view['slots']->stop(); ?>
```

What's going on?

The slots key we specified first was title and we gave the output method a second parameter, this means when the child template does not specify a slot section named title then it will default to "PPI Skeleton Application".

Using the assets helper

So why do we need an assets helper? Well one main purpose for it is to include asset files from your project's `./public/` folder such as images, css files, javascript files. This is useful because we're never hard-coding any baseurl's anywhere so it will work on any environment you host it on.

Review this example it shows a few examples of using the slots helper for various different reasons such as including CSS and JS files.

```
<?php $view->extend('::base.html.php'); ?>

<div class="blog-post-page">

    <h1><?=$blogPost->getTitle();?></h1>

    <img src="<?=$view['assets']->getUrl('images/blog.png');?>" alt="The Blog Image">

    <p class="created-by"><?=$blogPost->getCreatedBy();?></p>
    <p class="content"><?=$blogPost->getContent();?></p>

    <?php $view['slots']->start('include_js'); ?>
    <script type="text/javascript" src="<?=$view['assets']->getUrl('js/blog.js');?>"></script>
    <?php $view['slots']->stop(); ?>
```

```
    <?php $view['slots']->start('include_css'); ?>
    <link href="<?=$view['assets']->getUrl('css/blog.css');?>" rel="stylesheet">
    <?php $view['slots']->stop(); ?>

</div>
```

What's going on?

By asking for `images/blog.png` we're basically asking for `www.mysite.com/images/blog.png`, pretty straight forward right? Our `include_css` and `include_js` slots blocks are custom HTML that's loading up CSS/JS files just for this paritcular page load. This is great because you can split your application up onto smaller CSS/JS files and only load the required assets for your particular page, rather than having to bundle all your CSS into the one file.

Using the router helper

What is a router helper? The router help is a nice PHP class with routing related methods on it that you can use while you're building PHP templates for your application.

What's it useful for? The most common use for this is to perform a technique commonly known as reverse routing. Basically this is the process of taking a route key and turning that into a URL, rather than the standard process of having a URL and that translate into a route to become dispatched.

Why is reverse routing needed? Lets take the Blog_Show route we made earlier in the routing section. The syntax of that URI would be like: `/blog/show/{title}`, so rather than having numerous HTML links all manually referring to `/blog/show/my-title` we always refer to its route key instead, that way if we ever want to change the URI to something like `/blog/post/{title}` the templating layer of your application won't care because that change has been centrally maintained in your module's routes file.

Here are some examples of reverse routing using the routes helper

```
<a href="<?=$view['router']->generate('About_Page');?>">About Page</a>

<p>User List</p>
<ul>
<?php foreach($users as $user): ?>
    <li><a href="<?=$view['router']->generate('User_Profile', array('id' => $user->getID()));?>"><?=
<?php endforeach; ?>
</ul>
```

The output would be something like this

```
<a href="/about">About Page</a>

<p>User List</p>
<ul>
    <li><a href="/user/profile?id=23">PPI User</a></li>
    <li><a href="/user/profile?id=87675">Another PPI User</a></li>
</ul>
```

1.1.7 Services

What is a Service?

Let's put it simple, a Service is any PHP object that performs some sort of "global" task. It's a generic name used in computer science to describe an object that's created for a specific purpose (e.g. an API Handler). Each service is used

throughout your application whenever you need the specific functionality it provides. You don't have to do anything special to make a service; simply write a PHP class with some code that accomplishes a specific task.

Why using Services?

The advantage of thinking about "services" is that you begin to think about separating each piece of functionality in your application into a series of services. Since each service does just one job, you can easily access each service and use its functionality wherever you need it. Each service can also be more easily tested and configured since it's separated from the other functionality in your application. This idea is called service-oriented architecture and is not unique to PPI Framework or even PHP. Structuring your application around a set of independent service classes is a well-known and trusted object-oriented best-practice. These skills are key to being a good developer in almost any language.

Working with Services in PPI

Technically you can put your class file on any folder inside your module, this is because we use PSR0 for class autoloading, but for best practises we put all our file under the /Classes/ folder within a module.

Defining the Service in our Module.php file

To let our PPI app to know about the service, we need to declare it in our Module's Module.php file under the function getServiceConfig(), just as follows:

```
namespace ModuleName;

use PPI\Module\Module as BaseModule;
use PPI\Autoload;

class Module extends BaseModule
{

    // ....

    public function getServiceConfig()
    {
        return array('factories' => array(

            'foursquare.handler' => function($sm) {

                $handler = new \FourSquareModule\Classes\ApiHandler();
                $cache   = new \Doctrine\Common\Cache\ApcCache();
                $config  = $sm->get('config');

                $handler->setSecret($config['foursquare']['secret']);
                $handler->setKey($config['foursquare']['key']);
                $handler->setCache($cache);

                return $handler;
            }

        ));
    }

}
```

Using services in our Controllers

To use the services in our Controllers, we just need to call the function ->getService('service.name'), take the following code as a reference:

```
public function getVenuesAction()
{

    $lat      = $this->getRouteParam('lat');
    $lng      = $this->getRouteParam('lng');

    // Let's instantiate the service and then use it.
    $handler  = $this->getService('foursquare.handler');
    $venues   = $handler->getVenues($lat, $lng);

    return json_encode($venues);
}
```


- Getting Started
- Skeleton Application
- Modules
- Routing
- Controllers
- Templating
- Services

C

M

R

S

T