PPI Framework 2 Documentation

Release 1

PPI Framework Team

February 17, 2016

Contents

PPI is a framework delivery engine. Using the concept of microservices, it lets you choose which parts of frameworks you wish to use on a per-feature basis. As such each feature makes its own independent decisions, allowing you to pick the best tools from the best PHP frameworks

PPI bootstraps framework components for you from the top frameworks such as ZendFramework2, Symfony2, Laravel4, Doctrine2

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

CHAPTER 1

Installation

1.1 Download from composer

Download the latest version of 2.1, in the current directory

```
composer create-project -sdev --no-interaction ppi/skeleton-app /var/www/skeleton "^2.1"
```

1.2 Downloading from the website

http://www.ppi.io/downloads

1.3 Automatic Vagrant Installation

The recommended install procedure is to use the pre-built vagrant image that ships with the skeleton app in the `ansible` directory.

1.4 Installing vagrant and ansible

Before you can run vagrant you'll need to install a few system dependencies.

Install vagrant https://docs.vagrantup.com/v2/installation/

Install ansible: http://docs.ansible.com/ansible/intro_installation.html#latest-releases-via-pip

1.5 Running vagrant

Running the vagrant image - it's that easy!

```
vagrant up
```

1.6 Accessing the application

If you wish to use the skeletonapp as a hostname, run this command and browse to *http://skeletonapp.ppi*

```
sudo sh -c 'echo "192.168.33.99 skeletonapp.ppi" >> /etc/hosts'
```

Otherwise you can browse straight to the ip address of: *http://192.168.33.99*

1.7 Manual Web Server Configuration

Security is crucial to consider. As a result all your app code and configuration is kept hidden away outside of `/public/` and is inaccessible via the browser. Therefore 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. The `.htaccess` or web server's rewrite rules kick in which route all non-asset files to `/public/index.php`.

1.8 Apache Configuration

We are now creating an Apache virtual host for the application to make http://skeletonapp.ppi serve `index.php` from the `skeletonapp/public` directory.

```
<VirtualHost *:80>
    ServerName    skeletonapp.ppi
    DocumentRoot  "/var/www/skeleton/public"
    SetEnv        PPI_ENV dev
    SetEnv        PPI_DEBUG true

    <Directory "/var/www/skeleton/public">
        AllowOverride All
        Allow from all
        DirectoryIndex index.php
        Options Indexes FollowSymLinks

        RewriteEngine On
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteRule ^ index.php [L]

    </Directory>
</VirtualHost>
```

1.9 Nginx Virtual Host

```
server {
    listen 80;
    server_name skeletonapp.ppi;
    root /var/www/skeleton/public;
    index index.php;

    location / {
        try_files $uri /index.php$is_args$args;
    }
```

```
    location ~ \.php$ {
        fastcgi_pass 127.0.0.1:9000;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param HTTPS off;
    }
}
```

Restart your web server. The skeletonapp website can now be accessed using http://skeletonapp.ppi

1.10 Requirements

To easily check if your system passes all requirements, PPI provides two ways and we recommend you do both.

Why do we have both scripts? Because your CLI environment can have a separate **php.ini** file from your web environment so this will ensure you're good to go from both sides.

1.11 Requirements checking on the command-line

```
$ ./app/check
                        _
       ______    ______ |_|
      / __   | / __ | / |
      | |__| || |__| || |
      |  ___/ |  ___/ | |
      | |     | |     |_|
      |/      |/

      Framework Version 2

    -- Requirements Check --

* Configuration file used by PHP: /etc/php/cli-php5.4/php.ini
* Mandatory requirements **

OK        PHP version must be at least 5.3.3 (5.4.13--pl0-gentoo installed)
OK        PHP version must not be 5.3.16 as PPI won't work properly with it
OK        Vendor libraries must be installed
OK        app/cache/ directory must be writable
OK        app/logs/ directory must be writable
OK        date.timezone setting must be set
OK        Configured default timezone "Europe/Lisbon" must be supported by your installation of PHP
OK        json_encode() must be available
OK        session_start() must be available
OK        ctype_alpha() must be available
OK        token_get_all() must be available
OK        simplexml_import_dom() must be available
OK        detect_unicode must be disabled in php.ini
OK        xdebug.show_exception_trace must be disabled in php.ini
OK        xdebug.scream must be disabled in php.ini
OK        PCRE extension must be available
```

Watch out for the green `OK` markers. If they all light up, congratulations, you're good to go!

Below is the list of required and optional requirements.

1.12 Requirements checking in the browser

The check.php script is accessible in your browser at: http://skeletonapp.ppi/check.php

1.13 Must have requirements

- PHP needs to be a minimum version of PHP 5.3.3
- JSON needs to be enabled
- ctype needs to be enabled
- Your PHP.ini needs to have the date.timezone setting

1.14 Optional requirements

- You need to have the PHP-XML module installed
- You need to have at least version 2.6.21 of libxml
- PHP tokenizer needs to be enabled
- mbstring functions need to be enabled
- iconv needs to be enabled
- POSIX needs to be enabled (only on *nix)
- Intl needs to be installed with ICU 4+
- APC 3.0.17+ (or another opcode cache needs to be installed)
- PHP.ini recommended settings
  - `short_open_tag = On`
  - `magic_quotes_gpc = Off`
  - `register_globals = Off`
  - `session.autostart = Off`

CHAPTER 2

Skeleton Application

The skeleton application is an app for you to get up and running as quickly as possible. Inside you'll find the PHP libraries (`vendor` dir), a selection of useful modules, our recommended directory structure and some default configuration.

First, lets review the file structure of the PPI skeleton application:

```
www/                                    # your web root directory
|
-- skeleton/                           # the unpacked archive
    |
    -- app/
    |   -- console                      # CLI script to help debug the application
    |   -- init.php
    |   -- config/                      # application configuration files
    |   |   -- base/                    # base configuration to be extended by other environments
    |   |   |   -- app.yml
    |   |   -- dev/                     # configuration for the development environment (``dev``)
    |   |   |   -- app.yml
    |   |   -- prod/                    # configuration for the production environment (``prod``)
    |   |   |   -- app.yml
    |   -- cache/                       # application cache (must be writable by the web server)
    |   -- logs/                        # application logs  (must be writable by the web server)
    |   -- views/                       # global template (view) files
    |       -- base.html.php
    |
    -- modules/                         # application modules
    |   -- Application/
    |   -- Framework/
    |   -- UserModule/
    |
    -- public/
    |   -- index.php                    # front controller
    |   -- css/
    |   -- images/
    |   -- js/
    |
    -- vendor/                          # libraries installed by Composer
```

Lets break it down into parts:

2.1 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.

2.2 The public index.php file

The `/public/index.php` is also known as your bootstrap file, or front controller and is presented below:

```
<?php

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

// Setup autoloading and include PPI
require_once 'app/init.php';

// Set the environment
$env     = getenv('PPI_ENV') ?: 'dev';
$debug   = getenv('PPI_DEBUG') !== '0'  && $env !== 'prod';

// Create our PPI App instance
$app = new PPI\App(array(
    'environment'   => $env,
    'debug'         => $debug
));

// Configure the application
$app->loadConfig($app->getEnvironment().'/app.php');

// Load the application, match the URL and send an HTTP response
$app->boot()->dispatch()->send();
```

2.3 Environments

PPI supports the notion of "environments" to make the application behave differently from when you are coding and testing the application in your laptop to when you deploy it to a production server. While in *production* debug messages won't be logged, your application won't stop due to non-fatal PHP errors and we'll use caching wherever possible. In *development* you'll get everything!

2.3.1 Auto-set the environment using web server variables

Editing `index.php` whenever you want to test the application in another environment can be tedious. An alternative is to set environment variables in your web server on a per vhost basis.

If you're using Apache, environment variables can be set using the SetEnv directive.

Production VirtualHost configuration:

```
<VirtualHost *:80>
       ServerName    prod.skeletonapp.ppi.localhost
       DocumentRoot  "/var/www/skeleton/public"
       SetEnv        PPI_ENV prod
       SetEnv        PPI_DEBUG false
       ...
```

And a **development** VirtualHost configuration:

```
<VirtualHost *:80>
       ServerName    dev.skeletonapp.ppi.localhost
       DocumentRoot  "/var/www/skeleton/public"
       SetEnv        PPI_ENV dev
       SetEnv        PPI_DEBUG true
       ...
```

The front controller (`index.php`) needs to be slightly edited to load these environment variables:

```
// file: public/index.php

// Set the environment
$env     = getenv('PPI_ENV') ?: 'dev';
$debug   = getenv('PPI_DEBUG') !== '0'  && $env !== 'prod';

// Create our PPI App instance
$app = new PPI\App(array(
    'environment'   => $env,
    'debug'         => $debug
));
```

After this change `http://prod.skeletonapp.ppi.localhost/` will use production settings while `http://dev.skeletonapp.ppi.localhost/` is configured to work in development mode.

2.3.2 Creating a new environment

You don't need to be restricted to the `dev` and `prod` environments. To create a new environment with a special configuration, let's call it `staging`, just copy the folder contents of an existing environment to the new one and edit the `app.yml` file inside the `staging` dir.

```
$ cd /path/to/skeletonapp/app/config
$ cp -r prod staging
$ vim staging/app.yml
```

Now make sure `public/index.php` is picking up your new environment:

```
<?php
// ...

// Staging
$app = new PPI\App(array(
    'environment'   => 'staging',
    'debug'         => true
));

$app->loadConfig($app->getEnvironment().'/app.yml');

// ...
```

Note: PPI creates cache and log files associated with each environment. For this new `staging` environment cache files will be available under `app/cache/staging/` and the log file is available at `app/logs/staging.log`.

2.4 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.

2.5 The app/config folder

Starting with version 2.1 all the application configuration lives inside `app/config/<env>/` folders. Each `<env>` folder holds configuration for a specific environment: `dev`, `prod`.

2.5.1 Supported configuration formats

PPI supports both PHP and YAML formats. PHP is more powerful whereas YAML is more clean and readable. It is up to you to pick the format of your liking.

Note: In 2.1 we changed the default configuration file format from PHP to YAML because (we think) it is less verbose and faster to type but don't worry because PHP configuration files are and will always be supported.

2.5.2 YAML imports/include

The YAML language doesn't natively provide the capability to include other YAML files like a PHP `include` or `require` statement. To overcome this limitation PPI supports two special syntaxes: `imports` and `@include`.

Note: One of the goals of the PPI Framework is to provide an environment familiar to users coming from or going to the Symfony and Zend frameworks (among others). We support these two variants so these users do not need to worry about learning new syntaxes.

imports:

Available in the Symfony framework. Works like a PHP include statement providing base configuration to be tweaked in the current file. It is usually added at the top of the file.

```
    imports:
        - { resource: ../base/app.yml }
```

@include:

Available in the Zend framework. Similar to the `imports` syntax but can be used also in a subelement of a value.

```
    framework:
        @include: ../base/datasource.yml
```

2.6 The app.yml file

Looking at the example config file below, you can control things here such as the enabled templating engines, the datasource connection and the logger (`monolog`).

- *YAML*

```
imports:
    - { resource: datasource.yml }
    - { resource: modules.yml }

framework:
    templating:
        engines: ["php", "smarty", "twig"]
    skeleton_module:
        path: "./utils/skeleton_module"

monolog:
    handlers:
        main:
            type:  stream
            path:  %app.logs_dir%/%app.environment%.log
            level: debug
```

- *PHP*

```
<?php
$config = array();

$config['framework'] = array(
    'templating' => array(
        'engines'     => array('php', 'smarty', 'twig'),
    ),
    'skeleton_module'   => array(
        'path'  => './utils/skeleton_module'
    )
);

$config['datasource'] => array(
    'connections' = require __DIR__ . '/datasource.php'
);

$config['modules'] = require __DIR__ . 'modules.php';

return $config;
```

Tip: The configuration shown above is not exhaustive. For a complete listing of the available configuration options please check the sections in the Configuration Reference chapter.

2.7 The datasource.yml file

The `datasource.yml` is where you setup your database connection information.

Warning: Because this file may hold sensitive information consider not adding it to your source control system.

- *YAML*

```yaml
datasource:
    connections:
        main:
            type:   'pdo_mysql'
            host:   'localhost'
            dbname: 'ppi2_skeleton'
            user:   'root'
            pass:   'secret'
```

- *PHP*

```php
<?php
return array(
    'main' => array(
        'type'   => 'pdo_mysql',    // This can be any pdo driver. i.e: pdo_sqlite
        'host'   => 'localhost',
        'dbname' => 'ppi2_skeleton',
        'user'   => 'root',
        'pass'   => 'secret'
    )
);
```

2.8 The modules.yml 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.

- *YAML*

```yaml
modules:
    active_modules:
        - Framework
        - Application
        - UserModule

    module_listener_options:
        module_paths: ['./modules', './vendor']
```

- *PHP*

```php
<?php
return array(
    'active_modules' => array(
        'Framework',
        'Application',
        'UserModule',
    ),
    'module_listener_options' => array(
        'module_paths' => array('./modules', './vendor')
    ),
);
```

2.9 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.

2.10 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.

CHAPTER 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.

3.1 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.

```
-- Module.php
-- resources
|   -- config
|   |   -- config.yml
|   -- routes
|   |   -- laravel.php
|   |   -- symfony.yml
|   -- views
|       -- index
|           -- index.html.php
-- src
|   -- Controller
|   |   -- Index.php
|   |   -- Shared.php
```

3.2 The Module.php class

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

```php
<?php
namespace Application;
use PPI\Framework\Module\AbstractModule;

class Module extends AbstractModule
{
}
```

3.3 Autoloading

Registering your namespace can be done using the Zend Framework approach below. You can also skip this and register your module's namespace to your `composer.json` file

```
<?php
namespace Application;
use PPI\Framework\Module\AbstractModule;

class Module extends AbstractModule
{
    public function getAutoloaderConfig()
    {
        return array(
            'Zend\Loader\StandardAutoloader' => array(
                'namespaces' => array(
                    __NAMESPACE__ => __DIR__ . '/src/',
                ),
            ),
        );
    }
}
```

3.4 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.

```
<?php
namespace Application;
use PPI\Framework\Module\AbstractModule;

class Module extends AbstractModule
{
    public function init()
    {
    }
}
```

3.5 Configuration

Expanding on from the previous code example, we're now adding a `getConfig()` method. This must return a raw PHP array. You may `require/include` a PHP file directly or use the `loadConfig()` helper that works for both PHP and YAML files. When using `loadConfig()` you don't need to tell the full path, just the filename.

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 AbstractModule
{
    /**
     * Returns configuration to merge with application configuration.
     *
     * @return array
     */
    public function getConfig()
    {
        return $this->loadConfig(__DIR__ . '/resources/config/config.yml');
    }
}
```

Tip: To help you troubleshoot the configuration loaded by the framework you may use the `app/console config:dump` command

3.6 Conclusion

Lets move onto Services and Routing for our modules on the next pages.

CHAPTER 4

Services

Each of your features (modules) wants to be self-contained, isolated and in control of its own destiny. To keep such separation is a good thing (Separation of Responsibility principal). Once you've got that nailed then you want to begin exposing information out of your module. A popular architectural pattern is Service Oriented Architecture (SOA).

Services in PPI have names that you define. This can be something simple like `user.search` or `cache.driver` or it's even somewhat popular to use the Fully Qualified Class Name (FQCN) as the name of the service like this: `MyService::class`. With that in mind it's just a string and it's up to you what convention you use just make sure it's consistent.

4.1 Defining the Service in our Module

This is of course optional for your module but if you want to begin doing services then the method is `getServiceConfig`. This will be called on all modules upon `boot()` of PPI. Boot should be almost instantaneous and non-blocking, so be sure not to do anything expensive here such as make network connections, as that'll slow down your boot process.

```
<?php
namespace MyModule;

use PPI\Framework\Module\AbstractModule;
use MyModule\Factory\UserSearchFactory;
use MyModule\Factory\UserCreateFactory;
use MyModule\Factory\UserImportService;

class Module extends AbstractModule
{

    public function getServiceConfig()
    {
        return ['factories' => [
            'user.search' => UserSearchFactory::class,
            'user.create' => UserCreateFactory::class,
            'user.import' => function ($sm) {
               return new UserImportService($sm->get('Doctrine\ORM\EntityManager'));
            }
        ]];
    }
}
```

Above you'll see two types of ways to create a service. One is a Factory class and one is an inline factory closure. It's recommended to use a Factory class but each to their own.

4.2 Creating a Service Factory

```php
<?php
namespace MyModule\Factory;

use Zend\ServiceManager\ServiceLocatorInterface;
use Zend\ServiceManager\FactoryInterface;
use MyModule\Service\UserSearchService;

class UserSearchFactory implements FactoryInterface
{
    public function createService(ServiceLocatorInterface $sm)
    {
        $config = $sm->get('config');
        if(!isset($config['usersearch']['search_key']) {
            throw new \RuntimeException('Missing user search configuration');
        }

        return new UserSearchService(
            $config['usersearch']['search_key'],
            $sm->get('Doctrine\ORM\EntityManager')
        );
    }

}
```

4.3 Using services in our Controllers

To use the services in our Controllers, we just need to call `$this->getService('service.name')`

```php
<?php
public function searchUsersAction(Request $request, $lat, $long)
{
    $userSearchService = $this->getService('user.search');
    $users = $userSearchService->getUsersFromLatLong($lat, $long);

    return $this->render('MyModule:search:searchUsers.twig', compact('users'));
}
```

CHAPTER 5

Routing

Routes are the rules that tell the framework what URLs map to what actions of your application.

When PPI is booting up it will take call `getRoutes()` on each module and register its entry within the main `ChainRouter`, which is a router stack. PPI will call `match` on each router in the order that your modules have been defined in your application config.

PPI provides bindings for popular PHP routers which you will see examples of below. Review the documentation for each router to learn more about using them.

5.1 Using Symfony Router

```php
<?php
// Module.php
class Module extends AbstractModule
{
    // ....
    public function getRoutes()
    {
        return $this->loadSymfonyRoutes(__DIR__ . '/routes/symfonyroutes.yml');
    }
}
```

```
# resources/config/symfonyroutes.yml
BlogModule_Index:
    pattern:  /blog
    defaults: { _controller: "BlogModule:Blog:index"}
```

5.2 Using Aura Router

```php
<?php
// Module.php
class Module extends AbstractModule
{
    // ....
    public function getRoutes()
    {
        return $this->loadAuraRoutes(__DIR__ . '/resources/config/auraroutes.php');
    }
}
```

```php
<?php
// resources/config/auraroutes.php
$router
    ->add('BlogModule_Index', '/blog')
    ->addValues(array(
        'controller' => 'BlogModule\Controller\Index',
        'action' => 'indexAction'
    ));

// add a named route using symfony controller name syntax
$router->add('BlogModule_View', '/blog/view/{id}')
    ->addTokens(array(
        'id' => '\d+'
    ))
    ->addValues(array(
        'controller' => 'BlogModule:Index:view'
    ));

return $router;
```

5.3 Using FastRoute Router

```php
<?php
// Module.php
class Module extends AbstractModule
{
    public function getRoutes()
    {
        return $this->loadFastRouteRoutes(__DIR__ . '/resources/routes/fastroutes.php');
    }
}
```

```php
<?php
// resources/config/fastroutes.php
/**
 * @var \FastRoute\RouteCollector $r
 */
$r->addRoute('GET', '/blog', 'BlogModule\Controller\Index');
```

CHAPTER 6

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

6.1 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.

6.2 Example controller

Review the following route that we'll be matching.

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

So lets presume the route is `/blog/show/{blogId}`, 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(Request $request, $blogId)
    {
```

```
        $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'));
    }

}
```

6.3 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(Request $request, $blogId)
    {

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

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

    }
}
```

6.4 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(Request $request, $blogId)
    {

        // 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'));

    }
}
```

6.5 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();

    }
}
```

6.6 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();

    }
}
```

6.7 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();

    }
}
```

6.8 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();

    }
}
```

6.9 Working with session values

```
<?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();

    }
}
```

6.10 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
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;

        }
    }
}
```

6.11 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') {}

    }
}
```

6.12 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();
```

```
    }
}
```

6.13 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');
    }
}
```

6.14 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;

        }
    }
}
```

CHAPTER 7

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

7.1 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 repetitive 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.

7.2 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>
```

7.3 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>
```

7.4 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.

7.4.1 The route

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

7.4.2 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'));
    }
}
```

7.4.3 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>
```

7.5 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.

7.5.1 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>
```

7.5.2 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".

7.6 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 particular 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.

7.7 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>
```

CHAPTER 8

Databases

When you're developing an app, it is 100% sure you'll need to persist and read information to and from a database. Fortunately, PPI makes it simple to work with databases with our powerful DataSources component, which makes use of Doctrine DBAL layer, a library whose sole goal is to give you robust tools to make this easy. In this chapter, you'll learn the basic philosophy behind doctrine and see how easy is to use the DataSource component to work with databases.

Note: We suggest to use our DataSource component which is a wrapper around the Doctrine DBAL component. This provides you with a simple yet very powerful database layer to talk to any PDO supported database engine. If you prefer to work with another database component then you can simply create that as a service and inject that into your storage classes instead of the 'datasource' component.

8.1 A Simple Example: A User

The easiest way to understand how the DataSource component works is to see it in action. In this section, you'll configure your database, create a **Product** storage class, persist it to the database and fetch it back out.

8.2 Configuring the Database

Before you begin, you'll need to configure your database connection information. By convention, this information is usually configured in the app/datasource.config.php file:

```php
<?php

$connections = array();

$connections['main'] = array(
    'type'   => 'pdo_mysql', // This can be any pdo driver. i.e: pdo_sqlite
    'host'   => 'localhost',
    'dbname' => 'database',
    'user'   => 'database_user',
    'pass'   => 'database_password'
);

return $connections; // Very important you must return the connections variable from this script
```

Note: You can have multiple connections within your app. That means you may need to have multiple db engines, like MySQL, PGSQL, MSSQL, or any other PDO driver.

8.3 Creating the Storage Class

After configuring the database connection information, we need to have a storage class, which is the one that's going to be talking to the DataSource component when there's a need to persist information.

```php
<?php
namespace UserModule\Storage;

use UserModule\Storage\Base as BaseStorage;
use UserModule\Entity\User as UserEntity;

// Note here, we extend from
// a BaseStorage class
class User extends BaseStorage
{

    protected $_meta = array(
        'conn'    => 'main', // the connection.
        'table'   => 'user',
        'primary' => 'id',
        'fetchMode' => \PDO::FETCH_ASSOC
    );

    /**
     * Create a user record
     *
     * @param  array $userData
     * @return mixed
     */
    public function create(array $userData)
    {
        return $this->insert($userData);
    }

    /**
     * Get a user entity by its ID
     *
     * @param $userID
     * @return mixed
     * @throws \Exception
     */
    public function getByID($userID)
    {
        $row = $this->find($userID);
        if ($row === false) {
            throw new \Exception('Unable to obtain user row for id: ' . $userID);
        }

        return new UserEntity($row);
    }
```

```
    /**
     * Delete a user by their ID
     *
     * @param  integer $userID
     * @return mixed
     */
    public function deleteByID($userID)
    {
        return $this->delete(array($this->getPrimaryKey() => $userID));
    }

    /**
     * Count all the records
     *
     * @return mixed
     */
    public function countAll()
    {
        $row = $this->_conn->createQueryBuilder()
            ->select('count(id) as total')
            ->from($this->getTableName(), 'u')
            ->execute()
            ->fetch($this->getFetchMode());

        return $row['total'];
    }

    /**
     * Get entity objects from all users rows
     *
     * @return array
     */
    public function getAll()
    {
        $entities = array();
        $rows = $this->fetchAll();
        foreach ($rows as $row) {
            $entities[] = new UserEntity($row);
        }

        return $entities;
    }

}
```

First of all, we can see the class extends a BaseController class, which is a Shared Storage class, where we can place reusable code for all of our storage classes.

```
<?php

namespace UserModule\Storage;
use PPI\DataSource\ActiveQuery;
class Base extends ActiveQuery
{
    public function sharedFunction()
    {
        // code here...
    }
```

```
}
```

As you can see, the storage class is pretty explanatory by itself, you have a set of functions that perform specific tasks on the database; please note the use of the Doctrine DBAL Query Builder. Let's see how it works:

```
public function getByUsername($username)
{

    $row = $this->createQueryBuilder()
        ->select('u.*')
        ->from($this->getTableName(), 'u')
        ->andWhere('u.username = :username')
        ->setParameter(':username', $username)
        ->execute()
        ->fetch($this->getFetchMode());

    if ($row === false) {
        throw new \Exception('Unable to find user record by username: ' . $username);
    }

    return new UserEntity($row);

}
```

Note: Doctrine 2.1 ships with a powerful query builder for the SQL language. This QueryBuilder object has methods to add parts to an SQL statement. If you built the complete state you can execute it using the connection it was generated from. The API is roughly the same as that of the DQL Query Builder. For more information please refer to http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/query-builder.html

8.4 Entities

The previous function returns an object called UserEntity, you may be wondering, what is that, right? well, an Entity is just an object representing a record in a table. Now, let's see how does an Entity class looks like:

```
<?php

namespace UserModule\Entity;

class User
{

    protected $_id = null;
    protected $_username = null;
    protected $_firstname = null;
    protected $_lastname = null;
    protected $_email = null;

    public function __construct(array $data)
    {
        foreach ($data as $key => $value) {
            if (property_exists($this, '_' . $key)) {
                $this->{'_' . $key} = $value;
            }
        }
```

```
    }

    public function getID()
    {
        return $this->_id;
    }

    public function getFirstName()
    {
        return $this->_firstname;
    }

    public function getLastName()
    {
        return $this->_lastname;
    }

    public function getFullName()
    {
        return $this->getFirstName() . ' ' . $this->getLastName();
    }

    public function getEmail()
    {
        return $this->_email;
    }

    public function setUsername($username)
    {
        $this->_username = $username;
    }

    public function getUsername()
    {
        return $this->_username;
    }

}
```

8.4.1 Fetching Data

We have covered so far the Storage and Entities classes, now let's see how it actually works, for that, let's put a sample code:

```
    <?php
    namespace UserModule\Controller;

    use UserModule\Controller\Shared as SharedController;

    class Profile extends SharedController
    {

        public function viewAction()
        {

            // Get the username from the route params
```

```
            $username = $this->getRouteParam('username');

            // Instantiate the storage service
            $storage  = $this->getService('user.storage');

            // Fetch the user by username
            // This returns a UserEntity Object
            $user     = $storage->getByUsername($username);


            // Using the UserEntity Object is that simple:
            echo $user->getFullName(); // Returns the user's full name.
        }
    }
```

8.4.2 Inserting Data

In the previous section we saw how to fetch information from the database, now, let's see how to insert it.

```
<?php
namespace UserModule\Controller;

use UserModule\Controller\Shared as SharedController;

class Profile extends SharedController
{

    public function createAction()
    {

        // Assuming we're getting the info
        // from a submitted form through POST
        $post     = $this->post();

        // Instantiate the storage service
        $storage  = $this->getService('user.storage');

        // @todo You've got to add some codes here
        // To check for missing fields, or fields being empty.

        // Prepare user array for insertion
        $user     = array(
            'email'      => $post['userEmail'],
            'firstname'  => $post['userFirstName'],
            'lastname'   => $post['userLastName'],
            'username'   => $post['userName']
        );

        // Create the user
        $newUserID = $storage->create($user);

        // Successful registration. \o/
        $this->setFlash('success', 'User created');
        return $this->redirectToRoute('User_Thankyou_Page');

    }
```

```
}
```

CHAPTER 9

Configuration Reference

9.1 Configuration Reference

9.1.1 Framework Configuration ("framework")

This reference document is a work in progress. It should be accurate, but all options are not yet fully covered.

The core of the PPI Framework can be configured under the `framework` key in your application configuration. This includes settings related to sessions, translation, routing and more.

Configuration

session

cookie_lifetime **type**: `integer` **default**: `0`

This determines the lifetime of the session - in seconds. By default it will use `0`, which means the cookie is valid for the length of the browser session.

cookie_path **type**: `string` **default**: `/`

This determines the path to set in the session cookie. By default it will use `/`.

cookie_domain **type**: `string` **default**: `''`

This determines the domain to set in the session cookie. By default it's blank, meaning the host name of the server which generated the cookie according to the cookie specification.

cookie_secure **type**: `Boolean` **default**: `false`

This determines whether cookies should only be sent over secure connections.

cookie_httponly **type**: `Boolean` **default**: `false`

This determines whether cookies should only accessible through the HTTP protocol. This means that the cookie won't be accessible by scripting languages, such as JavaScript. This setting can effectively help to reduce identity theft through XSS attacks.

gc_probability New in version 2.1: The `gc_probability` option is new in version 2.1

type: `integer` **default**: `1`

This defines the probability that the garbage collector (GC) process is started on every session initialization. The probability is calculated by using `gc_probability` / `gc_divisor`, e.g. 1/100 means there is a 1% chance that the GC process will start on each request.

gc_divisor New in version 2.1: The `gc_divisor` option is new in version 2.1

type: `integer` **default**: `100`

See *gc_probability*.

gc_maxlifetime New in version 2.1: The `gc_maxlifetime` option is new in version 2.1

type: `integer` **default**: `14400`

This determines the number of seconds after which data will be seen as "garbage" and potentially cleaned up. Garbage collection may occur during session start and depends on *gc_divisor* and *gc_probability*.

save_path **type**: `string` **default**: `%app.cache.dir%/sessions`

This determines the argument to be passed to the save handler. If you choose the default file handler, this is the path where the files are created. You can also set this value to the `save_path` of your `php.ini` by setting the value to `null`:

- *YAML*

```
# app/config/app.yml
framework:
    session:
        save_path: null
```

- *PHP*

```
// app/config/app.php
return array(
'framework' => array(
    'session' => array(
        'save_path' => null,
    ),
));
```

templating

assets_base_urls **default**: `{ http: [], ssl: [] }`

This option allows you to define base URLs to be used for assets referenced from `http` and `ssl` (`https`) pages. A string value may be provided in lieu of a single-element array. If multiple base URLs are provided, PPI2 will select one from the collection each time it generates an asset's path.

For your convenience, `assets_base_urls` can be set directly with a string or array of strings, which will be automatically organized into collections of base URLs for `http` and `https` requests. If a URL starts with `https://` or is protocol-relative (i.e. starts with `//`) it will be added to both collections. URLs starting with `http://` will only be added to the `http` collection.

New in version 2.1: Unlike most configuration blocks, successive values for `assets_base_urls` will overwrite each other instead of being merged. This behavior was chosen because developers will typically define base URL's for each environment. Given that most projects tend to inherit configurations (e.g. `config_test.yml` imports `config_dev.yml`) and/or share a common base configuration (i.e. `app.yml`), merging could yield a set of base URL's for multiple environments.

assets_version **type**: `string`

This option is used to *bust* the cache on assets by globally adding a query parameter to all rendered asset paths (e.g. `/images/logo.png?v2`). This applies only to assets rendered via the Twig `asset` function (or PHP equivalent) as well as assets rendered with Assetic.

For example, suppose you have the following:

- *Twig*

```
<img src="{{ asset('images/logo.png') }}" alt="PPI!" />
```

- *PHP*

```
<img src="<?php echo $view['assets']->getUrl('images/logo.png') ?>" alt="PPI!" />
```

By default, this will render a path to your image such as `/images/logo.png`. Now, activate the `assets_version` option:

- *YAML*

```
# app/config/app.yml
framework:
    # ...
    templating: { engines: ['twig'], assets_version: v2 }
```

- *PHP*

```
// app/config/app.php
return array(
'framework' => array(
    ...,
    'templating'      => array(
        'engines'        => array('twig'),
        'assets_version' => 'v2',
    ),
));
```

Now, the same asset will be rendered as `/images/logo.png?v2` If you use this feature, you **must** manually increment the `assets_version` value before each deployment so that the query parameters change.

You can also control how the query string works via the *assets_version_format* option.

assets_version_format **type**: `string` **default**: `%%s?%%s`

This specifies a *sprintf* pattern that will be used with the *assets_version* option to construct an asset's path. By default, the pattern adds the asset's version as a query string. For example, if `assets_version_format` is set to `%%s?version=%%s` and `assets_version` is set to `5`, the asset's path would be `/images/logo.png?version=5`.

Note: All percentage signs (`%`) in the format string must be doubled to escape the character. Without escaping, values might inadvertently be interpreted as a service parameter.

Tip: Some CDN's do not support cache-busting via query strings, so injecting the version into the actual file path is necessary. Thankfully, `assets_version_format` is not limited to producing versioned query strings.

The pattern receives the asset's original path and version as its first and second parameters, respectively. Since the asset's path is one parameter, you cannot modify it in-place (e.g. `/images/logo-v5.png`); however, you can prefix the asset's path using a pattern of `version-%%2$s/%%1$s`, which would result in the path `version-5/images/logo.png`.

URL rewrite rules could then be used to disregard the version prefix before serving the asset. Alternatively, you could copy assets to the appropriate version path as part of your deployment process and forgo any URL rewriting. The latter option is useful if you would like older asset versions to remain accessible at their original URL.

Full Default Configuration

- *YAML*

```
framework:

    # router configuration
    router:
        resource:             ~ # Required
        type:                 ~
        http_port:            80
        https_port:           443

        # set to true to throw an exception when a parameter does not match the requirements
        # set to false to disable exceptions when a parameter does not match the requirements (a
        # set to null to disable parameter checks against requirements
```

```
            # 'true' is the preferred configuration in development mode, while 'false' or 'null' mig
            strict_requirements:  true

        # session configuration
        session:
            storage_id:           session.storage.native
            handler_id:           session.handler.native_file
            name:                 ~
            cookie_lifetime:      ~
            cookie_path:          ~
            cookie_domain:        ~
            cookie_secure:        ~
            cookie_httponly:      ~
            gc_divisor:           ~
            gc_probability:       ~
            gc_maxlifetime:       ~
            save_path:            %app.cache_dir%/sessions

        # templating configuration
        templating:
            assets_version:       ~
            assets_version_format:  %%s?%%s
            assets_base_urls:
                http:                 []
                ssl:                  []
            cache:                ~
            engines:              # Required

                # Example:
                - twig
            loaders:              []
            packages:

                # Prototype
                name:
                    version:              ~
                    version_format:       %%s?%%s
                    base_urls:
                        http:                 []
                        ssl:                  []

        # translator configuration
        translator:
            enabled:              false
            fallback:             en
```

9.1.2 Monolog Configuration Reference

Monolog is a logging library for PHP 5.3 used by PPI. It is inspired by the Python LogBook library.

- *YAML*

```
    monolog:
        handlers:

            # Examples:
            syslog:
```

```
            type:                stream
            path:                /var/log/symfony.log
            level:               ERROR
            bubble:              false
            formatter:           my_formatter
            processors:
                - some_callable
        main:
            type:                fingers_crossed
            action_level:        WARNING
            buffer_size:         30
            handler:             custom
        custom:
            type:                service
            id:                  my_handler

        # Default options and values for some "my_custom_handler"
        my_custom_handler:
            type:                 ~ # Required
            id:                   ~
            priority:             0
            level:                DEBUG
            bubble:               true
            path:                 "%app.logs_dir%/%app.environment%.log"
            ident:                false
            facility:             user
            max_files:            0
            action_level:         WARNING
            activation_strategy:  ~
            stop_buffering:       true
            buffer_size:          0
            handler:              ~
            members:              []
            channels:
                type:     ~
                elements: ~
            from_email:           ~
            to_email:             ~
            subject:              ~
            email_prototype:
                id:                   ~ # Required (when the email_prototype is used)
                factory-method:       ~
            channels:
                type:                 ~
                elements:             []
            formatter:            ~
```

- Framework
- Monolog
- Framework
- Monolog

C

D

M

R

S

T