Pop PHP Framework

Release 3.6.1

Sep 22, 2017

Contents

1 Overview **3**
- 1.1 Introduction . . . 3
- 1.2 About the Framework . . . 3
- 1.3 Community & Support . . . 4

2 Installation **5**
- 2.1 Using Composer . . . 5
- 2.2 Stand-alone Installation . . . 5
- 2.3 Requirements . . . 6
- 2.4 Recommendations . . . 6

3 Getting Started **9**
- 3.1 Applications . . . 9
- 3.2 Routing . . . 12
- 3.3 Controllers . . . 15
- 3.4 Services . . . 17
- 3.5 Events . . . 19
- 3.6 Modules . . . 20

4 User Guide **23**
- 4.1 MVC . . . 23
- 4.2 Views . . . 24
- 4.3 HTTP and the Web . . . 33
- 4.4 CLI and the Console . . . 38
- 4.5 Databases . . . 41
- 4.6 Forms . . . 52
- 4.7 Images . . . 64
- 4.8 PDFs . . . 70
- 4.9 Popcorn . . . 82

5 Tutorial Application **87**
- 5.1 Get the Tutorial Application . . . 87
- 5.2 Application Structure . . . 88
- 5.3 Working with the Web App . . . 90
- 5.4 Working with the Console App . . . 93

6 Reference **95**

6.1 pop-acl . . . 95
6.2 pop-application . . . 98
6.3 pop-auth . . . 100
6.4 pop-cache . . . 102
6.5 pop-code . . . 104
6.6 pop-config . . . 106
6.7 pop-console . . . 108
6.8 pop-controller . . . 109
6.9 pop-cookie . . . 110
6.10 pop-csv . . . 111
6.11 pop-db . . . 112
6.12 pop-debug . . . 124
6.13 pop-dir . . . 126
6.14 pop-dom . . . 128
6.15 pop-event . . . 130
6.16 pop-form . . . 131
6.17 pop-ftp . . . 144
6.18 pop-http . . . 145
6.19 pop-i18n . . . 149
6.20 pop-image . . . 152
6.21 pop-loader . . . 159
6.22 pop-log . . . 161
6.23 pop-mail . . . 163
6.24 pop-module . . . 167
6.25 pop-nav . . . 168
6.26 pop-paginator . . . 171
6.27 pop-pdf . . . 173
6.28 pop-router . . . 185
6.29 pop-service . . . 189
6.30 pop-session . . . 192
6.31 pop-validator . . . 193
6.32 pop-view . . . 195
6.33 popcorn . . . 204

7 Changelog **209**
7.1 3.6.1 . . . 209
7.2 3.6.0 . . . 209
7.3 3.5.2 . . . 209
7.4 3.5.1 . . . 210
7.5 3.5.0 . . . 210
7.6 3.0.1 . . . 210
7.7 3.0.0 . . . 211

Welcome to the Pop PHP Framework documentation. Here, you should find what you need to get familiar with the framework and start building applications with it. Follow the links below to get started.

CHAPTER 1

Overview

Introduction

The Pop PHP Framework is a lightweight, yet robust PHP framework that can be used for rapid application development. The framework itself has a base set of core components as well as numerous other additional components to facilitate many of the common features needed for a PHP application.

The two main concepts behind the Pop PHP Framework have always been:

1. To be easy to use and lightweight
2. To promote standards in development while maintaining a manageable learning curve

The goal is so that anyone from an advanced programmer to a novice developer can install and start using the Pop PHP Framework quickly and effectively.

About the Framework

Pop PHP is an open source, object-oriented PHP framework. At a minimum, it requires PHP 5.6, but has been tested for PHP 7 as well. It is available on Github, through Composer or as a stand-alone installation. The framework is tested using PHPUnit and continuous integration is maintained using Travis CI.

A Brief History of Pop PHP

The humble beginnings of the framework started back in 2009 as a small library of components. From that, the Pop PHP Framework began to take shape and the very first version of it was released in early 2012. That version made it to 1.7 in 2014, where support for version 1 ended and work on version 2 began. Pop PHP 2 ushered in a significant refactor, taking advantage of PHP 5.4, eliminating dependencies among components, separating the components into self-contained repositories, and incorporating Composer.

On September 14, 2017, version 3.6.1 was released and it continues the development and growth that the previous versions set forth.

Community & Support

Being an open source project, Pop PHP welcomes input and collaboration from the community and maintains an open dialogue with the community. Issues can be submitted to the appropriate repository on Github. Additional support and communication is maintained in the Gitter chat room and Twitter. The API documentation is located on the website.

CHAPTER 2

Installation

There are a couple of different options to install the Pop PHP Framework. You can use Composer or you can download the stand-alone version from the website http://www.popphp.org/.

Using Composer

If you want to use the full framework and all of its components, you can install the `popphp/popphp-framework` repository in the following ways:

Create a new project

```
composer create-project popphp/popphp-framework project-folder
```

Add it to the composer.json file

```
"require": {
    "popphp/popphp-framework": ">=3.6.1"
}
```

Add it to an existing project

```
composer require popphp/popphp-framework
```

If you only want to use the core components, you can use the `popphp/popphp` repository instead of the full `popphp/popphp-framework` repository.

Stand-alone Installation

If you do not wish to use Composer and want to install a stand-alone version of Pop, you can download a full version of the framework from the website http://www.popphp.org/. It is set up for web applications by default, but can be used to handle CLI applications as well. The following files and folders are included:

- **/public/**
    - index.php
    - .htaccess
- /vendor/

The /vendor/ folder contains the autoloader, the framework and all of the necessary components.

Requirements

The only main requirement for the Pop PHP Framework is that you have at least **PHP 5.6.0** installed in your environment. It has been tested and works with **PHP 7.0** as well.

Recommendations

Web Server

When writing web applications, a web server that supports URL rewrites is recommended, such as:

- Apache
- Nginx
- Lighttpd
- IIS

Extensions

Various components of the Pop PHP Framework require different PHP extensions to function correctly. If you wish to take advantage of the many components of Pop PHP, the following extensions are recommended:

- **pop-db**
    - mysqli
    - pdo_mysql
    - pdo_pgsql
    - pdo_sqlite
    - pgsql
    - sqlite3
    - sqlsrv
- **pop-image**
    - gd
    - imagick*
    - gmagick*
- **pop-cache**
    - apc
    - memcache

    - memcached
    - redis
    - sqlite3 or pdo_sqlite
- **pop-debug**
    - redis
    - sqlite3 or pdo_sqlite
- **other**
    - curl
    - ftp
    - ldap

* - The **imagick** and **gmagick** extensions cannot be used simultaneously.

CHAPTER 3

Getting Started

This part of the documentation is intended to give you an introduction to the core components of the Pop PHP Framework and a basic understanding of how they work and how to use them. More in-depth examples and concepts will be explored later in the User Guide and Reference sections.

Applications

The application object of the Pop PHP Framework is the main object that helps control and provide access to the application's elements, configuration and current state. Within the application object are ways to create, store and manipulate common elements that you may need during an application's life-cycle, such as the router, service locator, event manager and module manager. Additionally, you can also have access to the config object and the autoloader, if needed.

Configuring an Application

The application object's constructor is flexible in what it can accept when setting up your application. You can pass it individual instances of the objects your application will need:

```
$router  = new Pop\Router\Router();
$service = new Pop\Service\Locator();
$events  = new Pop\Event\Manager();

$app = new Pop\Application(
    $router, $service, $events
);
```

Or, you can pass it a configuration array and let the application object create and set up the objects for you:

```
$config = [
    'routes' => [
        '/' => [
            'controller' => 'MyApp\Controller\IndexController',
```

```
            'action'     => 'index'
        ]
    ],
    'events' => [
        [
            'name'     => 'app.init',
            'action'   => 'MyApp\Event::doSomething',
            'priority' => 1000
        ]
    ],
    'services' => [
        'session' => 'Pop\Session\Session::getInstance'
    ]
];

$app = new Pop\Application($config);
```

The Autoloader

You can also pass in the autoloader and access it through the application object if it is needed to register other components of the application. However, it is required that the autoloader object's API mirrors that of Composer's `Composer\Autoload\ClassLoader` class or Pop PHP's `Pop\Loader\ClassLoader` class.

```
$autoloader = include __DIR__ . '/vendor/autoload.php';

$app = new Pop\Application($autoloader);

$app->autoloader->add('Test', __DIR__ . '/src/Test');   // PSR-0
$app->autoloader->addPsr4('MyApp\\', __DIR__ . '/src'); // PSR-4
```

If needed, you can autoload your application's source through the application constructor by setting a `prefix` and `src` keys in the configuration array:

```
$autoloader = include __DIR__ . '/vendor/autoload.php';

$app = new Pop\Application($autoloader, [
    'prefix' => 'MyApp\\',
    'src'    => __DIR__ . '/src'
]);
```

If you need to autoload the above example as PSR-0, then set the `psr-0` config key to `true`. And then you can always continue autoloading other code sources by accessing the autoloader through the application object, as in the first example.

Basic API

Once the application object and its dependencies are wired up, you'll be able to interact with the application object through the appropriate API calls.

- `$app->bootstrap($autoloader = null)` - Bootstrap the application
- `$app->init()` - Initialize the application
- `$app->registerConfig($config)` - Register a new configuration object
- `$app->registerRouter($router)` - Register a new router object
- `$app->registerServices($services)` - Register a new service locator

- `$app->registerEvents($events)` - Register a new event manager
- `$app->registerModules($modules)` - Register a new module manager
- `$app->registerAutoloader($autoloader)` - Register an autoloader with the application
- `$app->mergeConfig($config, $preserve = false)` - Merge config values into the application
- `$app->run()` - Run the application

You can access the main elements of the application object through the following methods:

- `$app->autoloader()` - Access the autoloader
- `$app->config()` - Access the configuration object
- `$app->router()` - Access the router
- `$app->services()` - Access the service locator
- `$app->events()` - Access the event manager
- `$app->modules()` - Access the module manager

Also, magic methods expose them as direct properties as well:

- `$app->autoloader` - Access the autoloader
- `$app->config` - Access the configuration object
- `$app->router` - Access the router
- `$app->services` - Access the service locator
- `$app->events` - Access the event manager
- `$app->modules` - Access the module manager

Shorthand Methods

The application object has some shorthand methods to help tidy up common calls to elements within the application object:

- `$app->register($name, $module);` - Register a module
- `$app->unregister($name);` - Unregister a module
- `$app->isRegistered($name);` - Check if a module is registered
- `$app->module($module)` - Get a module object
- `$app->addRoute($route, $controller);` - Add a route
- `$app->addRoutes($routes);` - Add routes
- `$app->setService($name, $service);` - Set a service
- `$app->getService($name);` - Get a service
- `$app->removeService($name);` - Remove a service
- `$app->on($name, $action, $priority = 0);` - Attach an event
- `$app->off($name, $action);` - Detach an event
- `$app->trigger($name, array $args = []);` - Trigger an event

Running an Application

Of course, once you've configured your application object, you can run the application by simply executing the `run` method:

```
$app->run();
```

Routing

The router object facilitates the configuration and matching of the routes to access your application. It supports both HTTP and CLI routing. With it, you can establish valid routes along with any parameters that may be required with them.

HTTP Route Example

```
$router->addRoute('/hello', function() {
    echo 'Hello World';
});
```

In the above example, a web request of `http://localhost/hello` will execute the closure as the controller and echo `Hello World` out to the browser.

CLI Route Example

```
$router->addRoute('hello', function($name) {
    echo 'Hello World';
});
```

In the above example, a CLI command of `./app hello` will execute the closure as the controller and echo `Hello World` out to the console.

A controller object can be, and usually is, an instance of a class instead of a closure for more control over what happens for each route:

```
class MyApp\Controller\IndexController extends \Pop\Controller\AbstractController
{
    public function index()
    {
        echo 'Hello World!';
    }
}

$router->addRoute('/', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'index'
]);
```

In the above example, the request `/` is routed to the `index()` method in the defined controller class.

Controller Parameters

It's common to require access to various elements and values of your application while within an instance of your controller class. To provide this, the router object allows you to inject parameters into the controller upon instantiation. Let's assume your controller's constructor looks like this:

```
class MyApp\Controller\IndexController extends \Pop\Controller\AbstractController
{
    protected $foo;
    protected $bar;

    public function __construct($foo, $bar)
    {
        $this->foo = $foo;
        $this->bar = $bar;
    }
}
```

You could then inject parameters into the controller's constructor like this:

```
$router->addControllerParams(
    'MyApp\Controller\IndexController', [
        'foo' => $foo,
        'bar' => $bar
    ]
);
```

If you require parameters to be injected globally to all of your controller classes, then you can replace the controller name `'MyApp\Controller\IndexController` with `*` and they will be injected into all controllers. You can also define controller parameters within the route configuration as well.

```
$config = [
    'routes' => [
        '/products' => [
            'controller'       => 'MyApp\Controller\ProductsController',
            'action'           => 'index',
            'controllerParams' => [
                'baz'    => 789
            ]
        ]
    ]
];

$app = new Pop\Application($config);
```

Dispatch Parameters

Defining route dispatch parameters, you can define required (or optional) parameters that are needed for a particular route:

```
$router->addRoute('/hello/:name', function($name) {
    echo 'Hello ' . ucfirst($name);
});
```

```
$router->addRoute('hello <name>', function($name) {
    echo 'Hello ' . ucfirst($name);
});
```

The HTTP request of `http://localhost/hello/pop` and the CLI command of `./app hello pop` will each echo out `Hello Pop` to the browser and console, respectively.

Optional Dispatch Parameters

Consider the following controller class and method:

```
class MyApp\Controller\IndexController extends \Pop\Controller\AbstractController
{
    public function hello($name = null)
    {
        if (null === $name) {
            echo 'Hello World!';
        } else {
            echo 'Hello ' . ucfirst($name);
        }
    }
}
```

Then add the following routes for HTTP and CLI:

HTTP:

```
$router->addRoute('/hello[/:name]', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'hello'
]);
```

CLI:

```
$router->addRoute('hello [<name>]', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'hello'
]);
```

In the above example, the parameter `$name` is an optional dispatch parameter and the `hello()` method performs differently depending on whether or not the parameter value it present.

Dynamic Routing

Dynamic routing is also supported. You can define routes as outlined in the examples below and they will be dynamically mapped and routed to the correct controller and method. Let's assume your application has the following controller class:

```
class MyApp\Controller\UsersController extends \Pop\Controller\AbstractController
{

    public function index()
    {
        // Show a list of users
    }

    public function edit($id = null)
    {
        // Edit the user with the ID# of $id
    }
}
```

You could define a dynamic route for HTTP like this:

```
$router->addRoute('/:controller/:action[/:param]', [
    'prefix' => 'MyApp\Controller\\'
]);
```

and routes such as these would be valid:

- `http://localhost/users`
- `http://localhost/users/edit/1001`

For CLI, you can define a dynamic route like this:

```
$router->addRoute('<controller> <action> [<param>]', [
    'prefix' => 'MyApp\Controller\\'
]);
```

and routes such as these would be valid:

- `./app users`
- `./app users edit 1001`

Routing Syntax

The tables below outline the accepted routing syntax for the route matching:

HTTP

Web Route	What's Expected
/foo/:bar/:baz	The 2 params are required
/foo/:bar[/:baz]	First param required, last one is optional
/foo/:bar/:baz*	One required param, one required param that is a collection (array)
/foo/:bar[/:baz*]	One required param, one optional param that is a collection (array)

CLI

CLI Route	What's Expected
foo bar	Two commands are required
foo bar|baz	Two commands are required, the 2nd can accept 2 values
foo [bar|baz]	The second command is optional and can accept 2 values
foo -o1 [-o2]	First option required, 2nd option is optional
foo –option1|-o1 [–option2|-o2]	1st option required, 2nd optional; long & short supported for both
foo <name> [<email>]	First param required, 2nd param optional
foo –name= [–email=]	First value param required, 2nd value param optional

Controllers

The Pop PHP Framework comes with a base abstract controller class that can be extended to create the controller classes needed for your application. If you choose not to use the provided abstract controller class, you can write your own, but it needs to implement the main controller interface that is provided with Pop.

When building your controller classes by extending the abstract controller class, you can define the methods that represent the actions that will be executed on the matched route. Here's an example of what a basic controller might look like for a web application:

```
namespace MyApp\Controller;

use Pop\Controller\AbstractController;
use Pop\Http\Request;
use Pop\Http\Response;

class IndexController extends AbstractController
{
    protected $request;
    protected $response;

    public function __construct(Request $request, Response $response)
    {
        $this->request  = $request;
        $this->response = $response;
    }

    public function index()
    {
        // Show the index page
    }

    public function products()
    {
        // Show the products page
    }

}
```

The above example uses the `popphp/pop-http` component and injects a request and a response object into the controller's constructor. For more on how to inject controller parameters into the controller's constructor, refer the the section on controller parameters under Routing.

For a console application, your controller class might look like this, utilizing the `popphp/pop-console` component:

```
namespace MyApp\Controller;

use Pop\Controller\AbstractController;
use Pop\Console\Console;

class IndexController extends AbstractController
{
    protected $console;

    public function __construct(Console $console)
    {
        $this->console = $console;
    }

    public function home()
    {
        // Show the home screen
    }
```

```
    public function users()
    {
        // Show the users screen
    }

}
```

Services

If you need access to various services throughout the life-cycle of the application, you can register them with the service locator and recall them later. You can pass an array of services into the constructor, or you can set them individually as needed.

```
$services = new Pop\Service\Locator([
    'foo' => 'MyApp\SomeService'
]);

$services->set('bar', 'MyApp\SomeService->bar');
$services['baz'] = 'MyApp\SomeService->baz';
```

Then, you can retrieve a service in a number of ways:

```
$foo = $services['foo'];
$bar = $services->get('bar');
```

You can use the `isAvailable` method if you'd like to determine if a service is available, but not loaded yet:

```
if ($services->isAvailable('foo')) {
    $foo = $services['foo'];
} else {
    $services->set('foo', 'MyApp\SomeService');
}
```

The `isLoaded` method determines if the service has been set and previously called:

```
if ($services->isLoaded('foo')) {
    $foo = $services['foo'];
}
```

The service locator uses "lazy-loading" to store the service names and their attributes, and doesn't load or create the services until they are actually needed and called from the service locator.

You can also remove a service from the service locator if needed:

```
$services->remove('foo');
unset($services['bar']);
```

Syntax & Parameters

You have a couple of different options when setting services. You can pass callable strings or already instantiated instances of objects, although the latter could be potentially less efficient. Also, if needed, you can define parameters that will be passed into the service being called.

Syntax

Valid callable service strings are as follows:

1. 'SomeClass'
2. 'SomeClass->foo'
3. 'SomeClass::bar'

The first callable string example creates a new instance of `SomeClass` and returns it. The second callable string example creates a new instance of `SomeClass`, calls the method `foo()` and returns the value from it. The third callable string example calls the static method `bar()` in the class `SomeClass` and returns the value from it.

Parameters

Additionally, if you need to inject parameters into your service upon calling your service, you can set a service using an array with a `call` key and a `params` key like this:

```
$services = new Pop\Service\Locator([
    'foo' => [
        'call'   => 'MyApp\SomeService->foo',
        'params' => [
            'bar' => 123,
            'baz' => 456
        ]
    ]
]);
```

In the example above, the service `foo` is defined by the callable string `MyApp\SomeService->foo`. When the service `foo` is retrieved, the locator will create a new instance of `MyApp\SomeService`, call the method `foo` while passing the params `bar` and `baz` into the method and returning that value from that method.

Service Container

A service container class is available if you prefer to track and access your services through it. The first call to create a new service locator object will automatically register it as the 'default' service locator.

```
$services = new Pop\Service\Locator([
    'foo' => 'MyApp\SomeService'
]);
```

At some later point in your application:

```
$services = Pop\Service\Container::get('default');
```

If you would like register additional custom service locator objects, you can do that like so:

```
Pop\Service\Container::set('customServices', $myCustomServiceLocator);
```

And then later in your application:

```
if (Pop\Service\Container::has('customServices')) {
    $myCustomServiceLocator = Pop\Service\Container::get('customServices');
}
```

Events

The event manager provides a way to hook specific event listeners and functionality into certain points in an application's life cycle. You can create an event manager object and attach, detach or trigger event listeners. You can pass callable strings or already instantiated instances of objects, although the latter could be potentially less efficient.

```
$events = new Pop\Event\Manager();

$events->on('foo', 'MyApp\Event->bootstrap');
$events->on('bar', 'MyApp\Event::log');

$events->trigger('foo');
```

Similar to services, the valid callable strings for events are as follows:

1. 'SomeClass'
2. 'SomeClass->foo'
3. 'SomeClass::bar'

With events, you can also inject parameters into them as they are called, so that they may have access to any required elements or values of your application. For example, perhaps you need the events to have access to configuration values from your main application object:

```
$events->trigger('foo', ['application' => $application]);
```

In the above example, any event listeners triggered by `foo` will get the application object injected into them so that the event called can utilize that object and retrieve configuration values from it.

To detach an event listener, you call the `off` method:

```
$events->off('foo', 'MyApp\Event->bootstrap');
```

Event Priority

Event listeners attached to the same event handler can be assigned a priority value to determine the order in which they fire. The higher the priority value, the earlier the event listener will fire.

```
$events->on('foo', 'MyApp\Event->bootstrap', 100);
$events->on('foo', 'MyApp\Event::log', 10);
```

In the example above, the `bootstrap` event listener has the higher priority, so therefore it will fire before the `log` event listener.

Events in a Pop Application

Within the context of a Pop application object, an event manager object is created by default or one can be injected. The default hook points within a Pop application object are:

- app.init
- app.route.pre
- app.dispatch.pre
- app.dispatch.post

- app.error

This conveniently wires together various common points in the application's life cycle where one may need to fire off an event of some kind. You can build upon these event hook points, creating your own that are specific to your application. For example, perhaps you require an event hook point right before a controller in your application sends a response. You could create an event hook point in your application like this:

```
$application->on('app.send.pre', 'MyApp\Event::logResponse');
```

And then in your controller method, right before you send then response, you would trigger that event:

```
class MyApp\Controller\IndexController extends \Pop\Controller\AbstractController
{
    public function index()
    {
        $this->application->trigger->('app.send.pre', ['controller' => $this]);
        echo 'Home Page';
    }
}
```

The above example assumes that the application object is injected into the controller object and stored as a property. Also, it injects the controller object into the event listener in case the event called requires interaction with the controller or any of its properties. By default, the application object is injected into the events that are triggered from a Pop application object, but as demonstrated above, you can inject your own required parameters into an event call as well.

Modules

Modules can be thought of as "mini-application objects" that allow you to extend the functionality of your application. Module objects accept similar configuration parameters as an application object, such as `routes`, `services` and `events`. Additionally, it accepts a `prefix` configuration value as well to allow the module to register itself with the application autoloader. Here's an example of what a module might look like and how you'd register it with an application:

```
$application = new Pop\Application();

$moduleConfig = [
    'routes' => [
        '/' => [
            'controller' => 'MyModule\Controller\IndexController',
            'action'     => 'index'
        ]
    ],
    'prefix' => 'MyModule\\'
];

$application->register('myModule', $moduleConfig);
```

In the above example, the module configuration is passed into the application object. From there, an instance of the base module object is created and the configuration is passed into it. The newly created module object is then registered with the module manager within the application object.

Custom Modules

You can pass your own custom module objects into the application as well, as long as they implement the module interface provided. As the example below shows, you can create a new instance of your custom module and pass

that into the application, instead of just the configuration. The benefit of doing this is to allow you to extend the base module class and methods and provide any additional functionality that may be needed. In doing it this way, however, you will have to register your module's namespace prefix with the application's autoloader prior to registering the module with the application so that the application can properly detect and load the module's source files.

```
$application->autoloader->addPsr4('MyModule\\', __DIR__ . '/modules/mymodule/src');

$myModule = new MyModule\Module([
    'routes' => [
        '/' => [
            'controller' => 'MyModule\Controller\IndexController',
            'action'     => 'index'
        ]
    ]
]);

$application->register('myModule', $myModule);
```

The Module Manager

The module manager serves as the collection of module objects within the application. This facilitates accessing the modules you've added to the application during its life-cycle. In the examples above, the modules are not only being configured and created themselves, but they are also being registered with the application object. This means that at anytime, you can retrieve a module object or its properties in a number of ways:

```
$fooModule = $application->module('fooModule');

$barModule = $application->modules['barModule'];
```

You can also check to see if a module has been registered with the application object:

```
if ($application->isRegistered('fooModule')) {
    // Do something with the 'fooModule'
}
```

CHAPTER 4

User Guide

This section of the documentation will dive a little deeper and explore common concepts and ways to build PHP applications using the Pop PHP Framework. The topics covered here are some of the more common programming topics that utilize a set of the more common components from the framework.

MVC

Pop PHP Framework is an MVC framework. It is assumed that you have some familiarity with the MVC design pattern. An overly simple description of it is that the "controller" (C) serves as the bridge between the "models" (M) and "view" (V). It calls the proper models to handle the business logic of the request, returning the results of what was requested back to the user in a view. The basic idea is separation of concerns in that each component of the MVC pattern is only concerned with the one area it is assigned to handle, and that there is very little, if any, cross-cutting concerns among them.

Controllers

There is a controller interface `Pop\Controller\ControllerInterface` and an abstract controller class `Pop\Controller\AbstractController` that are provided with the core components of the Pop PHP Framework. The main application object and router object are wired to interact with controller objects that extend the abstract controller class, or at least implement the controller interface. The functionality is basic, as the API manages a default action and the dispatch method:

- **`$controller->setDefaultAction($default)`**
  - The "setDefaultAction" method sets the default action to handle a request that hasn't been assigned an action. Typically, this would be an "error" method or something along those lines. This method may not be used at all, as you can set the protected `$defaultAction` property within your child controller class directly.
- **`$controller->getDefaultAction()`**
  - This method retrieves the name of the current default action.

- **$controller->dispatch($action = null, $params = null)**
  - This is the main dispatch method, which will look for the "$action" method within the controller class and attempt to execute it, passing the "$params" into it if they are present. If the "$action" method is not found, the controller will fall back on the defined default action.

Views

The popphp/pop-view component provides the functionality for creating and rendering views. The topic of views will be covered more in-depth in the next section of the user guide, Views. But for now, know that the view component supports both file-based templates and string or stream-based templates. Data can be pushed into and retrieved from a view object and a template can be set in which the data will be rendered. A basic example would be:

```
$data = [
    'title'   => 'Home Page',
    'content' => '<p>Some page content.</p>'
];

$view = new Pop\View\View('index.phtml', $data);

echo $view;
```

Assuming the index.phtml template file is written containing the variables $title and $content, that data will be parsed and displayed within that template.

Again, the main ideas and concepts of the view component will be explored more the Views section of the user guide.

Models

There is no official "model component" with Pop PHP. This is because the model component of your application should be specific to your application. It is the thing that knows all about your application's business logic and data and knows how to handle it. So a component with that responsibility within you application should be written specifically for your application.

Model objects could be simple repositories that handle data transactions and provide data to views. They could integrate with your data source, such as a database, to save and retrieve data for your application. There are many different ways you can go with how you build and utilize model objects within your application.

More in-depth examples connecting all of these concepts will be covered later in the user guide.

Views

The popphp/pop-view component provides the functionality for creating and rendering views within your application. Data can be passed into a view, filtered and pushed out to the UI of the application to be rendering within the view template. As mentioned in the MVC section of the user guide, the popphp/pop-view component supports both file-based and stream-based templates.

Files

With file-based view templates, the view object utilizes traditional PHP script files with the extension .php or .phtml. The benefit of this type of template is that you can fully leverage PHP in manipulating and displaying your data in your view.

Let's revisit and expand upon the basic example given in the previous MVC section. First let's take a look at the view template, `index.phtml`:

```
<!DOCTYPE html>
<html>

<head>
    <title><?=$title; ?></title>
</head>
<body>
    <h1><?=$title; ?></h1>
<?=$content; ?>
    <ul>
<?php foreach ($links as $url => $link): ?>
        <li><a href="<?=$url; ?>"><?=$link; ?></a></li>
<?php endforeach; ?>
    </ul>
</body>

</html>
```

Then, we can set up the view and its data like below. Notice in the script above, we've set it up to loop through an array of links with the `$links` variable.

```
$data = [
    'title'   => 'View Example',
    'content' => '    <p>Some page content.</p>',
    'links'   => [
        'http://www.popphp.org/'     => 'Pop PHP Framework',
        'http://popcorn.popphp.org/' => 'Popcorn Micro Framework',
        'http://www.phirecms.org/'   => 'Phire CMS'
    ]
];

$view = new Pop\View\View('index.phtml', $data);

echo $view;
```

The result of the above example is:

```
<!DOCTYPE html>
<html>

<head>
    <title>View Example</title>
</head>
<body>
    <h1>View Example</h1>
    <p>Some page content.</p>
    <ul>
        <li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
        <li><a href="http://popcorn.popphp.org/">Popcorn Micro Framework</a></li>
        <li><a href="http://www.phirecms.org/">Phire CMS</a></li>
    </ul>
</body>

</html>
```

As mentioned before, the benefit of using file-based templates is you can fully leverage PHP within the script file. One

common thing that can be utilized when using file-based templates is file includes. This helps tidy up your template code and makes script files easier to manage by re-using template code. Here's an example that would work for the above script:

header.phtml

```
<!DOCTYPE html>
<html>

<head>
    <title><?=$title; ?></title>
</head>
<body>
```

footer.phtml

```
</body>

</html>
```

index.phtml

```
<?php include __DIR__ . '/header.phtml'; ?>
    <h1><?=$title; ?></h1>
<?=$content; ?>
    <ul>
<?php foreach ($links as $url => $link): ?>
        <li><a href="<?=$url; ?>"><?=$link; ?></a></li>
<?php endforeach; ?>
    </ul>
<?php include __DIR__ . '/footer.phtml'; ?>
```

Streams

With stream-based view templates, the view object uses a string template to render the data within the view. While using this method doesn't allow the use of PHP directly in the template like the file-based templates do, it does support basic logic and iteration to manipulate your data for display. The benefit of this is that it provides some security in locking down a template and not allowing PHP to be directly processed within it. Additionally, the template strings can be easily stored and managed within the application and remove the need to have to edit and transfer template files to and from the server. This is a common tactic used by content management systems that have template functionality built into them.

Let's look at the same example from above, but with a stream template:

```
$tmpl = <<<TMPL
<!DOCTYPE html>
<html>

<head>
    <title>[{title}]</title>
</head>
<body>
    <h1>[{title}]</h1>
[{content}]
    <ul>
[{links}]
        <li><a href="[{key}]">[{value}]</a></li>
```

```
[{/links}]
    </ul>
</body>

</html>
TMPL;
```

The above code snippet is a template stored as string. The stream-based templates use a system of **placeholders** to mark where you want the value to go within the template string. This is common with most string-based templating engines. In the case of `popphp/pop-view`, the placeholder uses the square bracket/curly bracket combination to wrap the variable name, such as `[{title}]`. In the special case of arrays, where iteration is allowed, the placeholders are marked the same way, but have an end mark like you see in the above template: `[{links}]` to `[{/links}]`. The iteration you need can happen in between those placeholder marks.

Let's use the exact same examples from above, except passing the string template, `$tmpl`, into the view constructor:

```
$data = [
    'title'   => 'View Example',
    'content' => '    <p>Some page content.</p>',
    'links'   => [
        'http://www.popphp.org/'     => 'Pop PHP Framework',
        'http://popcorn.popphp.org/' => 'Popcorn Micro Framework',
        'http://www.phirecms.org/'   => 'Phire CMS'
    ]
];

$view = new Pop\View\View($tmpl, $data);

echo $view;
```

We can achieve exact same results as above:

```
<!DOCTYPE html>
<html>

<head>
    <title>View Example</title>
</head>
<body>
    <h1>View Example</h1>
    <p>Some page content.</p>
    <ul>
        <li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
        <li><a href="http://popcorn.popphp.org/">Popcorn Micro Framework</a></li>
        <li><a href="http://www.phirecms.org/">Phire CMS</a></li>
    </ul>
</body>

</html>
```

As mentioned before, the benefit of using stream-based templates is you can limit the use of PHP within the template for security, as well as store the template strings within the application for easier access and management for the application users. And, streams can be stored in a number of ways. The most common is as a string in the application's database that gets passed in to the view's constructor. But, you can store them in a text-based file, such as `index.html` or `template.txt`, and the view constructor will detect that and grab the string contents from that template file. This will be applicable when we cover **includes** and **inheritance**, as you will need to be able to reference other string-based templates outside of the main one currently being used by the view object.

Stream Syntax

Scalars

Examples of using scalar values were shown above. You wrap the name of the variable in the placeholder bracket notation, `[{title}]`, in which the variable `$title` will render.

Arrays

As mentioned in the example above, iterating over arrays use a similar bracket notation, but with a start key `[{links}]` and an end key with a slash `[{/links}]`. In between those markers, you can write a line of code in the template to define what to display for each iteration:

```
$data = [
    'links'   => [
        'http://www.popphp.org/'     => 'Pop PHP Framework',
        'http://popcorn.popphp.org/' => 'Popcorn Micro Framework',
        'http://www.phirecms.org/'   => 'Phire CMS'
    ]
];
```

```
[{links}]
        <li><a href="[{key}]">[{value}]</a></li>
[{/links}]
```

Additionally, when you are iterating over an array in a stream template, you have access to a counter in the form of the placeholder, `[{i}]`. That way, if you need to, you can mark each iteration uniquely:

```
[{links}]
        <li id="li-item-[{i}]"><a href="[{key}]">[{value}]</a></li>
[{/links}]
```

The above template would render like this:

```
<li id="li-item-1"><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
<li id="li-item-2"><a href="http://popcorn.popphp.org/">Popcorn Micro Framework</a></
↪li>
<li id="li-item-3"><a href="http://www.phirecms.org/">Phire CMS</a></li>
```

You can also access nested associated arrays and their values by key name, to give you an additional level of control over your data, like so:

```
$data = [
    'links' => [
        [
            'title' => 'Pop PHP Framework',
            'url'   => 'http://www.popphp.org/'
        ],
        [
            'title' => 'Popcorn Micro Framework',
            'url'   => 'http://popcorn.popphp.org/'
        ]
    ]
];
```

```
[{links}]
        <li><a href="[{url}]">[{title}]</a></li>
[{/links}]
```

The above template and data would render like this:

```
<li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
<li><a href="http://popcorn.popphp.org/">Popcorn Micro Framework</a></li>
```

Conditionals

Stream-based templates support basic conditional logic as well to test if a variable is set. Here's an "if" statement:

```
[{if(foo)}]
    <p>The variable 'foo' is set to [{foo}].</p>
[{/if}]
```

And here's an "if/else" statement:

```
[{if(foo)}]
    <p>The variable 'foo' is set to [{foo}].</p>
[{else}]
    <p>The variable 'foo' is not set.</p>
[{/if}]
```

You can also use conditionals to check if a value is set in an array:

```
[{if(foo[bar])}]
    <p>The value of '$foo[$bar]' is set to [{foo[bar]}].</p>
[{/if}]
```

Furthermore, you can test if a value is set within a loop of an array, like this:

```
$data = [
    'links' => [
        [
            'title' => 'Pop PHP Framework',
            'url'   => 'http://www.popphp.org/'
        ],
        [
            'title' => 'Popcorn Micro Framework'
        ]
    ]
];
```

```
[{links}]
[{if(url)}]
        <li><a href="[{url}]">[{title}]</a></li>
[{/if}]
[{/links}]
```

The above template and data would only render one item because the *url* key is not set in the second value:

```
<li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
```

An "if/else" statement also works within an array loop as well:

```
[{links}]
[{if(url)}]
        <li><a href="[{url}]">[{title}]</a></li>
[{else}]
        <li>No URL was set</li>
[{/if}]
[{/links}]
```

```
<li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
<li>No URL was set</li>
```

Includes

As referenced earlier, you can store stream-based templates as files on disk. This is useful if you want to utilize includes with them. Consider the following templates:

header.html

```
<!DOCTYPE html>
<html>

<head>
    <title>[{title}]</title>
</head>
<body>
```

footer.html

```
</body>

</html>
```

You could then reference the above templates in the main template like below:

index.html

```
{{@include header.html}}
    <h1>[{title}]</h1>
[{content}]
{{@include footer.html}}
```

Note the include token uses a double curly bracket and @ symbol.

Inheritance

Inheritance, or blocks, are also supported with stream-based templates. Consider the following templates:

parent.html

```
<!DOCTYPE html>
<html>

<head>
{{header}}
    <title>[{title}]</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
```

```
{{/header}}
</head>

<body>
    <h1>[{title}]</h1>
    [{content}]
</body>

</html>
```

child.html

```
{{@extends parent.html}}

{{header}}
{{parent}}
    <style>
        body { margin: 0; padding: 0; color: #bbb;}
    </style>
{{/header}}
```

Render using the parent:

```
$view = new Pop\View\View('parent.html');
$view->title   = 'Hello World!';
$view->content = 'This is a test!';

echo $view;
```

will produce the following HTML:

```
<!DOCTYPE html>
<html>

<head>

    <title>Hello World!</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

</head>

<body>
    <h1>Hello World!</h1>
    This is a test!
</body>

</html>
```

Render using the child:

```
$view = new Pop\View\View('child.html');
$view->title   = 'Hello World!';
$view->content = 'This is a test!';

echo $view;
```

will produce the following HTML:

```
<!DOCTYPE html>
<html>

<head>

    <title>Hello World!</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

    <style>
        body { margin: 0; padding: 0; color: #bbb;}
    </style>

</head>

<body>
    <h1>Hello World!</h1>
    This is a test!
</body>

</html>
```

As you can see, using the child template that extends the parent, the `{{header}}` section was extended, incorporating the additional **style** tags in the header of the HTML. Note that the placeholder tokens for the extending a template use double curly brackets.

Filtering Data

You can apply filters to the data in the view as well for security and tidying up content. You pass the `addFilter()` method a callable and any optional parameters and then call the `filter()` method to iterate through the data and apply the filters.

```
$view = new Pop\View\View('index.phtml', $data);
$view->addFilter('strip_tags');
$view->addFilter('htmlentities', [ENT_QUOTES, 'UTF-8'])
$view->filter();

echo $view;
```

You can also use the `addFilters()` to apply muliple filters at once:

```
$view = new Pop\View\View('index.phtml', $data);
$view->addFilters([
    [
        'call'   => 'strip_tags'
    ],
    [
        'call'   => 'htmlentities',
        'params' => [ENT_QUOTES, 'UTF-8']
    ]
]);

$view->filter();

echo $view;
```

And if need be, you can clear the filters out of the view object as well:

```
$view->clearFilters();
```

HTTP and the Web

In building a web application with the Pop PHP Framework, there are a few concepts and components with which you'll need to be familiar. Along with the core components, one would commonly leverage the `popphp/pop-http` and `popphp/pop-session` components to get started on building a web application with Pop PHP.

HTTP

The `popphp/pop-http` component contains a **request object** and a **response object** that can assist in capturing and managing the incoming requests to your application and handle assembling the appropriate response back to the user.

Requests

The main request class is `Pop\Http\Request`. It has a robust API to allow you to interact with the incoming request and extract data from it. If you pass nothing to the constructor a new request object, it will attempt to parse the value contained in `$_SERVER['REQUEST_URI']`. You can, however, pass it a `$uri` to force a specific request, and also a `$basePath` to let the request object know that the base of the application is contained in a sub-folder under the document root.

Creating a new request object with a base path

In the following example, let's assume our application is in a sub-folder under the main document root:

- `/httpdocs`
- `/httpdocs/system`
- `/httpdocs/system/index.php`

We create a request object and pass it the base path `/system` so that application knows to parse incoming request after the `/system` base path.

```
$request = new Pop\Http\Request(null, '/system');
```

For example, if a request of `/system/users` came in, the application would know to use `/users` as the request and route it accordingly. If you need to reference the request URI, there are a couple of different methods to do so:

- `$request->getBasePath();` - returns only the base path ('/system')
- `$request->getRequestUri();` - returns only the request URI ('/users')
- `$request->getFullRequestUri();` - returns the full request URI string ('/system/users')

Getting path segments

If you need to break apart a URI into its segments access them for your application, you can do it with the `getSegment()` method. Consider the URI `/users/edit/1001`:

- `$request->getSegment(0);` - returns 'users'
- `$request->getSegment(1);` - returns 'edit'
- `$request->getSegment(2);` - returns '1001'

- `$request->getSegments();` - returns an array containing all of the path segments

Check the HTTP Method

- `$request->isGet();`
- `$request->isHead();`
- `$request->isPost();`
- `$request->isPut();`
- `$request->isPatch();`
- `$request->isDelete();`
- `$request->isTrace();`
- `$request->isHead();`
- `$request->isOptions();`
- `$request->isConnect();`

Retrieve Data from the Request

- `$request->getQuery($key = null);`
- `$request->getPost($key = null);`
- `$request->getFiles($key = null);`
- `$request->getPut($key = null);`
- `$request->getPatch($key = null);`
- `$request->getDelete($key = null);`
- `$request->getServer($key = null);`
- `$request->getEnv($key = null);`

If you do not pass the `$key` parameter in the above methods, the full array of values will be returned. The results from the `getQuery()`, `getPost()` and `getFiles()` methods mirror what is contained in the `$_GET`, `$_POST` and `$_FILES` global arrays, respectively. The `getServer()` and `getEnv()` methods mirror the `$_SERVER` and `$_ENV` global arrays, respectively.

If the request method passed is **PUT**, **PATCH** or **DELETE**, the request object will attempt to parse the raw request data to provide the data from that. The request object will also attempt to be content-aware and parse JSON or XML from the data if it successfully detects a content type from the request.

If you need to access the raw request data or the parsed request data, you can do so with these methods:

- `$request->getRawData();`
- `$request->getParsedData();`

Retrieve Request Headers

- `$request->getHeader($key);` - return a single request header value
- `$request->getHeaders();` - return all header values in an array

Responses

The `Pop\Http\Response` class has a full-featured API that allows you to create a outbound response to send back to the user or parse an inbound response from a request. The main constructor of the response object accepts a configuration array with the basic data to get the response object started:

```
$response = new Pop\Http\Response([
    'code'    => 200,
    'message' => 'OK',
    'version' => '1.1',
    'body'    => 'Some body content',
    'headers' => [
        'Content-Type' => 'text/plain'
    ]
]);
```

All of that basic response data can also be set as needed through the API:

- `$response->setCode($code);` - set the response code
- `$response->setMessage($message);` - set the response message
- `$response->setVersion($version);` - set the response version
- `$response->setBody($body);` - set the response body
- `$response->setHeader($name, $value);` - set a response header
- `$response->setHeaders($headers);` - set response headers from an array

And retrieved as well:

- `$response->getCode();` - get the response code
- `$response->getMessage();` - get the response message
- `$response->getVersion();` - get the response version
- `$response->getBody();` - get the response body
- `$response->getHeader($name);` - get a response header
- `$response->getHeaders($headers);` - get response headers as an array
- `$response->getHeadersAsString();` - get response headers as a string

Check the Response

- `$response->isSuccess();` - 100, 200 or 300 level response code
- `$response->isRedirect();` - 300 level response code
- `$response->isError();` - 400 or 500 level response code
- `$response->isClientError();` - 400 level response code
- `$response->isServerError();` - 500 level response code

And you can get the appropriate response message from the code like this:

```
use Pop\Http\Response;

$response = new Response();
$response->setCode(403);
$response->setMessage(Response::getMessageFromCode(403)); // Sets 'Forbidden'
```

Sending the Response

```
$response = new Pop\Http\Response([
    'code'    => 200,
    'message' => 'OK',
    'version' => '1.1',
    'body'    => 'Some body content',
    'headers' => [
        'Content-Type'   => 'text/plain'
    ]
]);

$response->setHeader('Content-Length', strlen($response->getBody()));
$response->send();
```

The above example would produce something like:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 19

Some body content
```

Redirecting a Response

```
Pop\Http\Response::redirect('http://www.domain.com/some-new-page');
exit();
```

Parsing a Response

In parsing a response from a request, you pass either the URL or a response string that already exists. A new response object with all of its data parsed from that response will be created:

```
$response = Pop\Http\Response::parse('http://www.domain.com/some-page');

if ($response->getCode() == 200) {
    // Do something with the response
} else if ($response->isError()) {
    // Uh oh. Something went wrong
}
```

Sessions

The session component gives you multiple ways to interact with the `$_SESSION` variable and store and retrieve data to it. The following are supported:

- Managing basic sessions and session values
- Creating namespaced sessions
- Setting session value expirations
- Setting request-based session values

Basic Sessions

```
$sess = Pop\Session\Session::getInstance();
$sess->user_id    = 1001;
$sess['username'] = 'admin';
```

The above snippet saves values to the user's session. To recall it later, you can access the session like this:

```
$sess = Pop\Session\Session::getInstance();
echo $sess->user_id;     // echos out 1001
echo $sess['username']; // echos out 'admin'
```

And to destroy the session and its values, you can call the `kill()` method:

```
$sess = Pop\Session\Session::getInstance();
$sess->kill();
```

Namespaced Sessions

Namespaced sessions allow you to store session under a namespace to protect and preserve that data away from the normal session data.

```
$sessFoo = new Pop\Session\SessionNamespace('foo');
$sessFoo->bar = 'baz'
```

What's happening "under the hood" is that an array is being created with the key `foo` in the main `$_SESSION` variable and any data that is saved or recalled by the `foo` namespaced session object will be stored in that array.

```
$sessFoo = new Pop\Session\SessionNamespace('foo');
echo $sessFoo->bar; // echos out 'baz'

$sess = Pop\Session\Session::getInstance();
echo $sess->bar; // echos out null, because it was only stored in the namespaced␣
↪session
```

And you can unset a value under a session namespace like this:

```
$sessFoo = new Pop\Session\SessionNamespace('foo');
unset($sessFoo->bar);
```

Session Value Expirations

Both basic sessions and namespaced sessions support timed values used to "expire" a value stored in session.

```
$sess = Pop\Session\Session::getInstance();
$sess->setTimedValue('foo', 'bar', 60);
```

The above example will set the value for `foo` with an expiration of 60 seconds. That means that if another request is made after 60 seconds, `foo` will no longer be available in session.

Request-Based Session Values

Request-based session values can be stored as well, which sets a number of time, or "hops", that a value is available in session. This is useful for **flash messaging**. Both basic sessions and namespaced sessions support request-based session values.

```
$sess = Pop\Session\Session::getInstance();
$sess->setRequestValue('foo', 'bar', 3);
```

The above example will allow the value for `foo` to be available to the user for 3 requests. After the 3rd request, `foo` will no longer be available in session. The default value of "hops" is 1.

CLI and the Console

In writing an application tailored for the CLI, you can leverage the *popphp/pop-console* component to help you configure and build your application for the console. Please note, While there is support for CLI-based applications to run on both Windows and Linux/Unix based systems, the *popphp/pop-console* component's colorize feature is not supported in Windows CLI-based applications.

CLI

Before getting into utilizing the *popphp/pop-console* component with CLI-based applications, let's take a look at some simple CLI scripts to call and execute your PHP scripts. For these examples, let's assume you have a small PHP script like this:

```
<?php
echo 'Hello ' . $argv[1];
```

If we would like to access and run that script via a simple command, there a few ways to do it.

Natively on Linux/Unix

Add *#!/usr/bin/php* as the first line to the PHP script above (or wherever your PHP binary is located):

```
#!/usr/bin/php
<?php
echo 'Hello, ' . $argv[1];
```

You can then name the script without a PHP extension, for example *foo*, make it executable and run it.

```
$ ./foo pop
```

Bash

If you want to use a BASH script as a wrapper to access and run your PHP script, named *foo.php* in this case, then you can create a BASH script file like this:

```
#!/bin/bash
php ./foo.php $@
```

Let's name the BASH script *app* and make it executable. Then you can run it like this:

```
$ ./app pop
```

Windows Batch

Similarly on Windows, you can create a batch file to do the same. Let's create a batch file called *app.bat*:

```
@echo off
php foo.php %*
```

Then on the Windows command line, you can run:

```
C:\> app pop
```

Arguments

In the above examples, the initial example PHP script is accessing an argument from the *$argv* array, which is common. As you can see, all the examples pushed the argument value of 'pop' into the script, as to echo *Hello, pop* on the screen.

While PHP can access that value via the *$argv* array, BASH scripts and batch files can pass them into the PHP scripts via:

```
#!/bin/bash
php ./foo.php $1 $2 $3
```

```
@echo off
php foo.php %1 %2 %3
```

Of course, those examples only allow for up to 3 arguments to be passed. So, as you can see, the examples above for BASH and batch files use the catch-alls *$@* and *%** respectively, to allow all possible parameters to be passed into the PHP script.

```
#!/bin/bash
php ./foo.php $@
```

```
@echo off
php foo.php %*
```

Console

Using the *popphp/pop-console* component when building a CLI-based application with Pop gives you access to a set of features that facilitate the routing and display of your application.

```
$console = new Pop\Console\Console();
```

Here's a look at the basic API:

- `$console->setWidth(80);` - sets the character width of the console
- `$console->setIndent(4);` - sets the indentation in spaces at the start of a line
- `$console->colorize($string, $fg, $bg);` - colorize the string and return the value
- `$console->prompt($prompt, $options, $caseSensitive, $length);` - call a prompt and return the answer
- `$console->append($text = null, $newline = true);` - appends text to the current console response body
- `$console->write($text = null, $newline = true);` - appends text to the current console response body and sends the response
- `$console->send();` - sends the response
- `$console->clear();` - clears the console screen (Linux/Unix only)

Commands

When using the *popphp/pop-console* component, you can create command objects and add them to the console object. This is useful for storing and calling help screens on a per-command basis

Using a Command

```
use Pop\Console\Console;
use Pop\Console\Command;

```

```
$edit = new Command('edit');
$edit->setHelp('This is the help screen for the edit command.');

$console = new Console();
$console->addCommand($edit);

$console->append($console->help('edit'));
$console->send();
```

And if you wire up your controller correctly, the following example would be output like below:

```
$ ./pop edit help
  This is the help screen for the edit command.
```

Prompts

With the *popphp/pop-console* component, you can call a prompt to read in user input:

```
$input = $console->prompt('Are you sure? [Y/N]', ['Y', 'N']);
```

What the above line of code does is echo the prompt to the user and once the user enters an answer, that answer gets returned back and stored in the variable *$input*. The *$options* array allows you to enforce a certain set of options. Failure to input one of those options will result in the prompt being printed to the console screen again.

Colors

As mentioned before, on terminals that support basic ANSI color, such as on a Linux or Unix machine, you can colorize your text:

```
use Pop\Console\Console;

$coloredText = $console->colorize('Hello World!', Console::BOLD_CYAN);
$console->append($coloredText);
```

The list of available color constants are:

- NORMAL
- BLACK
- RED
- GREEN
- YELLOW
- BLUE
- MAGENTA
- CYAN
- WHITE
- GRAY
- BOLD_RED
- BOLD_GREEN

- BOLD_YELLOW
- BOLD_BLUE
- BOLD_MAGENTA
- BOLD_CYAN
- BOLD_WHITE

Databases

Databases are commonly a core piece of an application's functionality. The *popphp/pop-db* component provides a layer of abstraction and control over databases within your application. Natively, there are adapters that support for the following database drivers:

- MySQL
- PostgreSQL
- SQLServer
- SQLite
- PDO

One can use the above adapters, or extend the base `Pop\\Db\\Adapter\\AbstractAdapter` class and write your own. Additionally, access to individual database tables can be leveraged via the `Pop\Db\Record` class.

Connecting to a Database

You can use the database factory to create the appropriate adapter instance and connect to a database:

```
$mysql = Pop\Db\Db::connect('mysql', [
    'database' => 'my_database',
    'username' => 'my_db_user',
    'password' => 'my_db_password',
    'host'     => 'mydb.server.com'
]);
```

And for other database connections:

```
$pgsql  = Pop\Db\Db::connect('pgsql', $options);
$sqlsrv = Pop\Db\Db::connect('sqlsrv', $options);
$sqlite = Pop\Db\Db::connect('sqlite', $options);
```

If you'd like to use the PDO adapter, it requires the *type* option to be defined so it can set up the proper DSN:

```
$pdo = Pop\Db\Db::connect('pdo', [
    'database' => 'my_database',
    'username' => 'my_db_user',
    'password' => 'my_db_password',
    'host'     => 'mydb.server.com',
    'type'     => 'mysql'
]);
```

And there are shorthand methods as well:

```
$mysql  = Pop\Db\Db::mysqlConnect($options);
$pgsql  = Pop\Db\Db::pgsqlConnect($options);
$sqlsrv = Pop\Db\Db::sqlsrvConnect($options);
$sqlite = Pop\Db\Db::sqliteConnect($options);
$pdo    = Pop\Db\Db::pdoConnect($options);
```

The database factory outlined above is simply creating new instances of the database adapter objects. The code below would produce the same results:

```
$mysql  = new Pop\Db\Adapter\Mysql($options);
$pgsql  = new Pop\Db\Adapter\Pgsql($options);
$sqlsrv = new Pop\Db\Adapter\Sqlsrv($options);
$sqlite = new Pop\Db\Adapter\Sqlite($options);
$pdo    = new Pop\Db\Adapter\Pdo($options);
```

The above adapter objects are all instances of `Pop\\Db\\Adapter\\AbstractAdapter`, which implements the `Pop\\Db\\Adapter\\AdapterInterface` interface. If necessary, you can use that underlying foundation to build your own database adapter to facilitate your database needs for your application.

Querying a Database

Once you've created a database adapter object, you can then use the API to interact with and query the database. Let's assume the database has a table *users* in it with the column *username* in the table.

```
$db = Pop\Db\Db::connect('mysql', $options);

$db->query('SELECT * FROM `users`');

while ($row = $db->fetch()) {
    echo $row['username'];
}
```

Using Prepared Statements

You can also query the database using prepared statements as well. Let's assume the *users* table from above also has and *id* column.

```
$db = Pop\Db\Db::connect('mysql', $options);

$db->prepare('SELECT * FROM `users` WHERE `id` > ?');
$db->bindParams(['id' => 1000]);
$db->execute();

$rows = $db->fetchResult();

foreach ($rows as $row) {
    echo $row['username'];
}
```

The Query Builder

The query builder is a part of the component that provides an interface that will produce syntactically correct SQL for whichever type of database you have elected to use. One of the main goals of this is portability across different systems

and environments. In order for it to function correctly, you need to pass it the database adapter your application is currently using so that it can properly build the SQL.

```
$db = Pop\Db\Db::connect('mysql', $options);

$sql = $db->createSql();
$sql->select(['id', 'username'])
    ->from('users')
    ->where('id > :id');

echo $sql;
```

The above example will produce:

```
SELECT `id`, `username` FROM `users` WHERE `id` > ?
```

If the database adapter changed to PostgreSQL, then the output would be:

```
SELECT "id", "username" FROM "users" WHERE "id" > $1
```

And SQLite would look like:

```
SELECT "id", "username" FROM "users" WHERE "id" > :id
```

The SQL Builder component has an extensive API to assist you in constructing complex SQL statements. Here's an example using JOIN and ORDER BY:

```
$db = Pop\Db\Db::connect('mysql', $options);

$sql = $db->createSql();
$sql->select([
    'user_id'    => 'id',
    'user_email' => 'email'
])->from('users')
  ->leftJoin('user_data', ['users.id' => 'user_data.user_id'])
  ->orderBy('id', 'ASC');
  ->where('id > :id');

echo $sql;
```

The above example would produce the following SQL statement for MySQL:

```
SELECT `id` AS `user_id`, `email` AS `user_email` FROM `users`
    LEFT JOIN `user_data` ON `users`.`id` = `user_data`.`user_id`
    WHERE `id` > ?
    ORDER BY `id` ASC;
```

The Schema Builder

In addition to the query builder, there is also a schema builder to assist with database table structures and their management. In a similar fashion to the query builder, the schema builder has an API that mirrors the SQL that would be used to create, alter and drop tables in a database.

```
$db = Pop\Db\Db::connect('mysql', $options);

$schema = $db->createSchema();
```

```
$schema->create('users')
    ->int('id', 16)
    ->varchar('username', 255)
    ->varchar('password', 255);

echo $schema;
```

The above code would produced the following SQL:

```
CREATE TABLE `users` (
  `id` INT(16),
  `username` VARCHAR(255),
  `password` VARCHAR(255)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Active Record

The `Pop\Db\Record` class uses the Active Record pattern as a base to allow you to work with and query tables in a database directly. To set this up, you create a table class that extends the `Pop\Db\Record` class:

```
class Users extends Pop\Db\Record { }
```

By default, the table name will be parsed from the class name and it will have a primary key called *id*. Those settings are configurable as well for when you need to override them. The "class-name-to-table-name" parsing works by converting the CamelCase class name into a lower case underscore name (without the namespace prefix):

- Users -> users
- MyUsers -> my_users
- MyApp\Table\SomeMetaData -> some_meta_data

If you need to override these default settings, you can do so in the child table class you create:

```
class Users extends Pop\Db\Record
{
    protected $table  = 'my_custom_users_table';

    protected $prefix = 'pop_';

    protected $primaryKeys = ['id', 'some_other_id'];
}
```

In the above example, the table is set to a custom value, a table prefix is defined and the primary keys are set to a value of two columns. The custom table prefix means that the full table name that will be used in the class will be *pop_my_custom_users_table*.

Once you've created and configured your table classes, you can then use the API to interface with them. At some point in the beginning stages of your application's life cycle, you will need to set the database adapter for the table classes to use. You can do that like this:

```
$db = Pop\Db\Db::connect('mysql', $options);
Pop\Db\Record::setDb($db);
```

That database adapter will be used for all table classes in your application that extend `Pop\Db\Record`. If you want a specific database adapter for a particular table class, you can specify that on the table class level:

```
$userDb = Pop\Db\Db::connect('mysql', $options)
Users::setDb($userDb);
```

From there, the API to query the table in the database directly like in the following examples:

Fetch a single row by ID, update data

```
$user = Users::findById(1001);

if (isset($user->id)) {
    $user->username = 'admin2';
    $user->save();
}
```

Fetch a single row by another column

```
$user = Users::findOne(['username' => 'admin2']);

if (isset($user->id)) {
    $user->username = 'admin3';
    $user->save();
}
```

Fetch multiple rows

```
$users = Users::findAll([
    'order' => 'id ASC',
    'limit' => 25
]);

foreach ($users as $user) {
    echo $user->username;
}

$users = Users::findBy(['logins' => 0]);

foreach ($users as $user) {
    echo $user->username . ' has never logged in.';
}
```

Fetch and return only certain columns

```
$users = Users::findAll(['select' => ['id', 'username']]);

foreach ($users as $user) {
    echo $user->id . ': ' . $user->username;
}

$users = Users::findBy(['logins' => 0], ['select' => ['id', 'username']]);

foreach ($users as $user) {
    echo $user->id . ': ' . $user->username . ' has never logged in.';
}
```

Create a new record

```
$user = new Users([
    'username' => 'editor',
```

```
    'email'    => 'editor@mysite.com'
]);

$user->save();
```

You can execute custom SQL to run custom queries on the table. One way to do this is by using the SQL Builder:

```
$sql = Users::db()->createSql();

$sql->select()
    ->from(Users::table())
    ->where('id > :id');

$users = Users::execute($sql, ['id' => 1000]);

foreach ($users as $user) {
    echo $user->username;
}
```

The basic overview of the record class static API is as follows, using the child class `Users` as an example:

- `Users::setDb(Adapter\AbstractAdapter $db, $prefix = null, $isDefault = false)` - Set the DB adapter
- `Users::hasDb()` - Check if the class has a DB adapter set
- `Users::db()` - Get the DB adapter object
- `Users::sql()` - Get the SQL object
- `Users::findById($id)` - Find a single record by ID
- `Users::findOne(array $columns = null, array $options = null)` - Find a single record
- `Users::findBy(array $columns = null, array $options = null, $resultAs = Record::AS_RECORD)` - Find a record or records by certain column values
- `Users::findAll(array $options = null, $resultAs = Record::AS_RECORD)` - Find all records in the table
- `Users::execute($sql, $params, $resultAs = Record::AS_RECORD)` - Execute a custom prepared SQL statement
- `Users::query($sql, $resultAs = Record::AS_RECORD)` - Execute a simple SQL query

In the `findOne`, `findBy` and `findAll` methods, the `$options` parameter is an associative array that can contain values such as:

```
$options = [
    'select' => ['id', 'username'],
    'order'  => 'username ASC',
    'limit'  => 25,
    'offset' => 5
];
```

The *select* key value can be an array of only the columns you would like to select. Otherwise it will select all columns *. The *order*, *limit* and *offset* key values all relate to those values to control the order, limit and offset of the SQL query.

The `$resultAs` parameter allows you to set what the row set is returned as:

- `AS_ARRAY` - As arrays
- `AS_OBJECT` - As array objects
- `AS_RECORD` - As instances of the `Pop\Db\Record`

The benefit of `AS_RECORD` is that you can operate on that row in real time, but if there are many rows returned in the result set, performance could be hindered. Therefore, you can use something like `AS_ARRAY` as an alternative to keep the row data footprint smaller and lightweight.

Accessing records non-statically

If you're interested in an alternative to the active record pattern, there is a non-static API within the `Pop\Db\Record` class:

```
$user = new Users();
$user->getById(5);
echo $user->username;
```

The basic overview of the result class API is as follows:

- `$user->getById($id)` - Find a single record by ID
- `$user->getOneBy(array $columns = null, array $options = null)` - Find a single record by ID
- `$user->getBy(array $columns = null, array $options = null, $resultAs = Record::AS_RECORD)` - Find a record or records by certain column values
- `$user->getAll(array $options = null, $resultAs = Record::AS_RECORD)` - Find all records in the table

Relationships & Associations

Relationships and associations are supported to allow for a simple way to select related data within the database. Building on the example above with the *Users* table, let's add an *Info* and an *Orders* table. The user will have a 1:1 relationship with a row in the *Info* table, and the user will have a 1:many relationship with the *Orders* table:

```
class Users extends Pop\Db\Record
{

    // Define a 1:1 relationship
    public function info()
    {
        return $this->hasOne('Info', 'user_id')
    }

    // Define a 1:many relationship
    public function orders()
    {
        return $this->hasMany('Orders', 'user_id');
    }

}

// Foreign key to the related user is `user_id`
class Info extends Pop\Db\Record
{

}
```

```
// Foreign key to the related user is `user_id`
class Orders extends Pop\Db\Record
{

    // Define the parent relationship up to the user that owns this order record
    public function user()
    {
        return $this->belongsTo('User', 'user_id');
    }

}
```

So with those table classes wired up, there now exists a useful network of relationships among the database entities that can be accessed like this:

```
$user = Users::findById(1);

// Loop through all of the user's orders
foreach ($user->orders as $order) {
    echo $order->id;
}

// Display the user's title stored in the `info` table
echo $user->info->title;
```

Or, in this case, if you have selected an order already and want to access the parent user that owns it:

```
$order = Orders::findById(2);
echo $order->user->username;
```

Eager-Loading

In the 1:many example given above, the orders are "lazy-loaded," meaning that they aren't called from of the database until you call the `orders()` method. However, you can access a 1:many relationship with what is called "eager-loading." However, to take full advantage of this, you would have alter the method in the *Users* table:

```
class Users extends Pop\Db\Record
{

    // Define a 1:many relationship
    public function orders($options = null, $eager = false)
    {
        return $this->hasMany('Orders', 'user_id', $options, $eager);
    }

}
```

The `$options` parameter is a way to pass additional select criteria to the selection of the order rows, such as *order* and *limit*. The `$eager` parameter is what triggers the eager-loading, however, with this set up, you'll actually access it using the static `with()` method, like this:

```
$user = Users::with('orders')->getById(10592005);

// Loop through all of the user's orders
foreach ($user->orders as $order) {
    echo $order->id;
}
```

A note about the access in the example given above. Even though a method was defined to access the different relationships, you can use a magic property to access them as well, and it will route to that method. Also, object and array notation is supported throughout any record object. The following example all produce the same result:

```
$user = Users::findById(1);

echo $user->info()->title;
echo $user->info()['title'];
echo $user->info->title;
echo $user->info['title'];
```

Shorthand SQL Syntax

To help with making custom queries more quickly and without having to utilize the Sql Builder, there is shorthand SQL syntax that is supported by the `Pop\Db\Record` class. Here's a list of what is supported and what it translates into:

Basic operators

```
$users = Users::findBy(['id' => 1]);   => WHERE id = 1
$users = Users::findBy(['id!=' => 1]); => WHERE id != 1
$users = Users::findBy(['id>' => 1]);  => WHERE id > 1
$users = Users::findBy(['id>=' => 1]); => WHERE id >= 1
$users = Users::findBy(['id<' => 1]);  => WHERE id < 1
$users = Users::findBy(['id<=' => 1]); => WHERE id <= 1
```

LIKE and NOT LIKE

```
$users = Users::findBy(['%username%'   => 'test']); => WHERE username LIKE '%test%'
$users = Users::findBy(['username%'    => 'test']); => WHERE username LIKE 'test%'
$users = Users::findBy(['%username'    => 'test']); => WHERE username LIKE '%test'
$users = Users::findBy(['-%username'   => 'test']); => WHERE username NOT LIKE '%test'
$users = Users::findBy(['username%-'   => 'test']); => WHERE username NOT LIKE 'test%'
$users = Users::findBy(['-%username%-' => 'test']); => WHERE username NOT LIKE '%test%
↪'
```

NULL and NOT NULL

```
$users = Users::findBy(['username' => null]);  => WHERE username IS NULL
$users = Users::findBy(['username-' => null]); => WHERE username IS NOT NULL
```

IN and NOT IN

```
$users = Users::findBy(['id' => [2, 3]]);  => WHERE id IN (2, 3)
$users = Users::findBy(['id-' => [2, 3]]); => WHERE id NOT IN (2, 3)
```

BETWEEN and NOT BETWEEN

```
$users = Users::findBy(['id' => '(1, 5)']);  => WHERE id BETWEEN (1, 5)
$users = Users::findBy(['id-' => '(1, 5)']); => WHERE id NOT BETWEEN (1, 5)
```

Additionally, if you need use multiple conditions for your query, you can and they will be stitched together with AND:

```
$users = Users::findBy([
    'id>'       => 1,
    '%username' => 'user1'
]);
```

which will be translated into:

```
WHERE (id > 1) AND (username LIKE '%test')
```

If you need to use OR instead, you can specify it like this:

```
$users = Users::findBy([
    'id>'       => 1,
    '%username' => 'user1 OR'
]);
```

Notice the ' OR' added as a suffix to the second condition's value. That will apply the OR to that part of the predicate like this:

```
WHERE (id > 1) OR (username LIKE '%test')
```

Database Migrations

Database migrations are scripts that assist in implementing new changes to the database, as well rolling back any changes to a previous state. It works by storing a directory of migration class files and keeping track of the current state, or the last one that was processed. From that, you can write scripts to run the next migration state or rollback to the previous one.

You can create a blank template migration class like this:

```
use Pop\Db\Sql\Migrator;

Migrator::create('MyNewMigration', 'migrations');
```

The code above will create a file that look like `migrations/20170225100742_my_new_migration.php` and it will contain a blank class template:

```
<?php

use Pop\Db\Sql\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{

    public function up()
    {

    }

    public function down()
    {

    }

}
```

From there, you can write your forward migration steps in the `up()` method, or your rollback steps in the `down()` method. Here's an example that creates a table when stepped forward, and drops that table when rolled back:

```
<?php

use Pop\Db\Sql\Migration\AbstractMigration;
```

```
class MyNewMigration extends AbstractMigration
{

    public function up()
    {
        $schema = $this->db->createSchema();
        $schema->create('users')
            ->int('id', 16)->increment()
            ->varchar('username', 255)
            ->varchar('password', 255)
            ->primary('id');

        $this->db->query($schema);
    }

    public function down()
    {
        $schema = $this->db->createSchema();
        $schema->drop('users');
        $this->db->query($schema);
    }

}
```

To step forward, you would call the migrator like this:

```
use Pop\Db\Db;
use Pop\Db\Sql\Migrator;

$db = Pop\Db\Db::connect('mysql', [
    'database' => 'my_database',
    'username' => 'my_db_user',
    'password' => 'my_db_password',
    'host'     => 'mydb.server.com'
]);

$migrator = new Migrator($db, 'migrations');
$migrator->run();
```

The above code would have created the table `users` with the defined columns. To roll back the migration, you would call the migrator like this:

```
use Pop\Db\Db;
use Pop\Db\Sql\Migrator;

$db = Pop\Db\Db::connect('mysql', [
    'database' => 'my_database',
    'username' => 'my_db_user',
    'password' => 'my_db_password',
    'host'     => 'mydb.server.com'
]);

$migrator = new Migrator($db, 'migrations');
$migrator->rollback();
```

And the above code here would have dropped the table `users` from the database.

Forms

HTML Forms are common to web applications and present a unique set of challenges in building, rendering and validating a form and its elements. The *popphp/pop-form* component helps to manage those aspects of web forms and streamline the process of utilizing forms in your web application.

Form Elements

Most of the standard HTML5 form elements are supported within the *popphp/pop-form* component. If you require a different element of any kind, you can extend the *Pop\Form\Element\AbstractElement* class to build your own. With each element instance, you can set attributes, values and validation parameters.

The generic input class is:

- *Pop\Form\Element\Input*

The standard available input element classes extend the above class are:

- *Pop\Form\Element\Input\Button*
- *Pop\Form\Element\Input\Checkbox*
- *Pop\Form\Element\Input\Datalist*
- *Pop\Form\Element\Input\Email*
- *Pop\Form\Element\Input\File*
- *Pop\Form\Element\Input\Hidden*
- *Pop\Form\Element\Input\Number*
- *Pop\Form\Element\Input\Password*
- *Pop\Form\Element\Input\Radio*
- *Pop\Form\Element\Input\Range*
- *Pop\Form\Element\Input\Reset*
- *Pop\Form\Element\Input\Submit*
- *Pop\Form\Element\Input\Text*
- *Pop\Form\Element\Input\Url*

Special case input element classes include:

- *Pop\Form\Element\Input\Captcha*
- *Pop\Form\Element\Input\Csrf*

Other available form element classes are:

- *Pop\Form\Element\Button*
- *Pop\Form\Element\Select*
- *Pop\Form\Element\SelectMultiple*
- *Pop\Form\Element\Textarea*

Special form element collection classes include:

- *Pop\Form\Element\CheckboxSet*

- *Pop\Form\Element\RadioSet*

In the case of the CAPTCHA and CSRF input element classes, they have special parameters that are required for them to perform their functions. In the case of the form element collection classes, they provide a grouping of elements within a fieldset for easier management.

Here's an example that creates and renders a simple input text element:

```
$text = new Pop\Form\Element\Input\Text('first_name');
$text->setRequired(true);
$text->setAttribute('size', 40);
echo $text;
```

The above code will produce:

```
<input name="first_name" id="first_name" type="text" required="required" size="40" />
```

Note the *required* attribute. Since the element was set to be required, this will assign that attribute to the element, which is only effective client-side, if the client interface hasn't bypassed HTML form validation. If the client interface has bypassed HTML form validation, then the form object will still account for the required setting when validating server-side with PHP. If the field is set to be required and it is empty, validation will fail.

Also, the *name* and *id* attributes of the element are set from the first *$name* parameter that is passed into the object. However, if you wish to override these, you can by doing this:

```
$text = new Pop\Form\Element\Input\Text('first_name');
$text->setAttribute('size', 40);
$text->setAttribute('id', 'my-custom-id');
echo $text;
```

The above code will produce:

```
<input name="first_name" id="my-custom-id" type="text" size="40" />
```

Here's an example of a select element:

```
$select = new Pop\Form\Element\Select('colors', [
    'Red'   => 'Red',
    'Green' => 'Green',
    'Blue'  => 'Blue'
]);
$select->setAttribute('class', 'drop-down');
echo $select;
```

The above code will produce:

```
<select name="colors" id="colors" class="drop-down">
    <option value="Red">Red</option>
    <option value="Green">Green</option>
    <option value="Blue">Blue</option>
</select>
```

Here's an example of a checkbox set:

```
$checkbox = new Pop\Form\Element\CheckboxSet('colors', [
    'Red'   => 'Red',
    'Green' => 'Green',
    'Blue'  => 'Blue'
```

```
]);
echo $checkbox;
```

The above code will produce:

```
<fieldset class="checkbox-fieldset">
    <input class="checkbox" type="checkbox" name="colors[]" id="colors" value="Red" />
    <span class="checkbox-span">Red</span>
    <input class="checkbox" type="checkbox" name="colors[]" id="colors1" value="Green
↪" />
    <span class="checkbox-span">Green</span>
    <input class="checkbox" type="checkbox" name="colors[]" id="colors2" value="Blue"␣
↪/>
    <span class="checkbox-span">Blue</span>
</fieldset>
```

In the special case of a field collection set, the object manages the creation and assignment of values and other elements, such as the <*span*> elements that hold the field values. Each element has a class attribute that can be utilized for styling.

Labels

When you create instances of form elements, you can set the label to uses in conjunction with the element. This is typically used when rendering the main form object.

```
$text = new Pop\Form\Element\Input\Text('first_name');
$text->setLabel('First Name:');
```

When rendered with the form, the label will render like this:

```
<label for="first_name">First Name:</label>
```

Validators

When if comes to attaching validators to a form element, there are a few options. The default option is to use the *popphp/pop-validator* component. You can use the standard set of validator classes included in that component, or you can write your own by extending the main *Pop\Validator\AbstractValidator* class. Alternatively, if you'd like to create your own, independent validators, you can do that as well. You just need to pass it something that is callable.

Here's an example using the *popphp/pop-validator* component:

```
$text = new Pop\Form\Element\Input\Text('first_name');
$text->addValidator(new Pop\Validator\AlphaNumeric());
```

If the field's valid was set to something that wasn't alphanumeric, then it would fail validation:

```
$text->setValue('abcd#$%');
if (!$text->validate()) {
    print_r($text->getErrors());
}
```

If using a custom validator that is callable, the main guideline you would have to follow is that upon failure, your validator should return a failure message, otherwise, simply return null. Those messages are what is collected in the elements *$errors* array property for error message display. Here's an example:

```
$myValidator = function($value) {
    if (preg_match('/^\w+$/', $value) == 0) {
        return 'The value is not alphanumeric.';
    } else {
        return null;
    }
};

$text = new Pop\Form\Element\Input\Text('first_name');
$text->addValidator($myValidator);

$text->setValue('abcd#$%');
if (!$text->validate()) {
    print_r($text->getErrors());
}
```

Form Objects

The form object serves as the center of the functionality. You can create a form object and inject form elements into it. The form object then manages those elements, their values and processes the validation, if any, attached to the form elements. Consider the following code:

```
use Pop\Form\Form;
use Pop\Form\Element\Input;
use Pop\Validator;

$form = new Form();
$form->setAttribute('id', 'my-form');

$username = new Input\Text('username');
$username->setLabel('Username:')
         ->setRequired(true)
         ->setAttribute('size', 40)
         ->addValidator(new Validator\AlphaNumeric());

$email = new Input\Email('email');
$email->setLabel('Email:')
      ->setRequired(true)
      ->setAttribute('size', 40);

$submit = new Input\Submit('submit', 'SUBMIT');

$form->addFields([$username, $email, $submit]);

if ($_POST) {
    $form->setFieldValues($_POST);
    if (!$form->isValid()) {
        echo $form; // Re-render, form has errors
    } else {
        echo 'Valid!';
        print_r($form->toArray());
    }
} else {
    echo $form;
}
```

The form's action is pulled from the current *REQUEST_URI* of the current page, unless otherwise directly specified. Also, the form's method defaults to *POST* unless otherwise specified. The above code will produce the following HTML as the initial render by default:

```
<form action="/" method="post" id="my-form">
    <fieldset id="my-form-fieldset-1" class="my-form-fieldset">
        <dl>
            <dt>
                <label for="username" class="required">Username:</label>
            </dt>
            <dd>
                <input type="text" name="username" id="username" value="" required=
↪"required" size="40" />
            </dd>
            <dt>
                <label for="email" class="required">Email:</label>
            </dt>
            <dd>
                <input type="email" name="email" id="email" value="" required=
↪"required" size="40" />
            </dd>
            <dd>
                <input type="submit" name="submit" id="submit" value="SUBMIT" />
            </dd>
        </dl>
    </fieldset>
</form>
```

If the user were to input non-valid data into on of the fields, and then submit the form, then the script would be processed again, this time, it would trigger the form validation and render with the error messages, like this:

```
<form action="/" method="post" id="my-form">
    <fieldset id="my-form-fieldset-1" class="my-form-fieldset">
        <dl>
            <dt>
                <label for="username" class="required">Username:</label>
            </dt>
            <dd>
                <input type="text" name="username" id="username" value="dfvdfv##$dfv"␣
↪required="required" size="40" />
                <div class="error">The value must only contain alphanumeric␣
↪characters.</div>
            </dd>
            <dt>
                <label for="email" class="required">Email:</label>
            </dt>
            <dd>
                <input type="email" name="email" id="email" value="" required=
↪"required" size="40" />
            </dd>
            <dd>
                <input type="submit" name="submit" id="submit" value="SUBMIT" />
            </dd>
        </dl>
    </fieldset>
</form>
```

As you can see above, the values entered by the user are retained so that they may correct any errors and re-submit the

form. Once the form is corrected and re-submitted, it will pass validation and then move on to the portion of the script that will handle what to do with the form data.

Using Filters

When dealing with the data that is being passed through a form object, besides validation, you'll want to consider adding filters to further protect against bad or malicious data. We can modify the above example to add filters to be used to process the form data before it is validated or re-rendered to the screen. A filter can be anything that is callable, like this:

```
if ($_POST) {
    $form->addFilter('strip_tags');
    $form->addFilter('htmlentities', [ENT_QUOTES, 'UTF-8']);
    $form->setFieldValues($_POST);
    if (!$form->isValid()) {
        echo $form; // Has errors
    } else {
        echo 'Valid!';
        print_r($form->getFields());
    }
} else {
    echo $form;
}
```

In the above code, the *addFilter* methods are called before the data is set into the form for validation or re-rendering. The example passes the *strip_tags* and *htmlentities* functions and those functions are applied to the each value of form data. So, if a user tries to submit the data *<script>alert("Bad Code");</script>* into one of the fields, it would get filtered and re-rendered like this:

```
<input type="text" name="username" id="username" value="alert(&amp;quot;Bad Code&amp;quot;);"␣
↪required="required" size="40" />
```

As you can see, the *<script>* tags were stripped and the quotes were converted to HTML entities.

Field Configurations

Most of the functionality outlined above can be administered and managed by passing field configuration arrays into the form object. This helps facilitate and streamline the form creation process. Consider the following example:

```
use Pop\Form\Form;
use Pop\Validator;

$fields = [
    'username' => [
        'type'       => 'text',
        'label'      => 'Username',
        'required'   => true,
        'validators' => new Validator\AlphaNumeric(),
        'attributes' => [
            'class' => 'username-field',
            'size'  => 40
        ]
    ],
    'password' => [
        'type'       => 'password',
```

```
        'label'      => 'Password',
        'required'   => true,
        'validators' => new Validator\GreaterThanEqual(6),
        'attributes' => [
            'class' => 'password-field',
            'size'  => 40
        ]
    ],
    'submit' => [
        'type'       => 'submit',
        'value'      => 'SUBMIT',
        'attributes' => [
            'class' => 'submit-btn'
        ]
    ]
];

$form = Form::createFromConfig($fields);
$form->setAttribute('id', 'login-form');

echo $form;
```

which will produce the following HTML code:

```
<form action="/" method="post" id="login-form">
    <fieldset id="login-form-fieldset-1" class="login-form-fieldset">
        <dl>
            <dt>
                <label for="username" class="required">Username</label>
            </dt>
            <dd>
                <input type="text" name="username" id="username" value="" required=
↪"required" class="username-field" size="40" />
            </dd>
            <dt>
                <label for="password" class="required">Password</label>
            </dt>
            <dd>
                <input type="password" name="password" id="password" value="" 
↪required="required" class="password-field" size="40" />
            </dd>
            <dd>
                <input type="submit" name="submit" id="submit" value="SUBMIT" class=
↪"submit-btn" />
            </dd>
        </dl>
    </fieldset>
</form>
```

In the above example, the *$fields* is an associative array where the keys are the names of the fields and the array values contain the field configuration values. Some of the accepted field configuration values are:

- `'type'` - field type, i.e. 'button', 'select', 'text', 'textarea', 'checkbox', 'radio', 'input-button'
- `'label'` - field label
- `'required'` - boolean to set whether the field is required or not. Defaults to false.
- `'attributes'` - an array of attributes to apply to the field.

- `'validators'` - an array of validators to apply to the field. Can be a single callable validator as well.
- `'value'` - the value to be set for the field
- `'values'` - the option values to be set for the field (for selects, checkboxes and radios)
- `'selected'` - the field value or values that are to be marked as 'selected' within the field's values.
- `'checked'` - the field value or values that are to be marked as 'checked' within the field's values.

Here is an example using fields with multiple values:

```
use Pop\Form\Form;
use Pop\Validator;

$fields = [
    'colors' => [
        'type'   => 'checkbox',
        'label'  => 'Colors',
        'values' => [
            'Red'   => 'Red',
            'Green' => 'Green',
            'Blue'  => 'Blue'
        ],
        'checked' => [
            'Red', 'Green'
        ]
    ],
    'country' => [
        'type'   => 'select',
        'label'  => 'Country',
        'values' => [
            'United States' => 'United States',
            'Canada'        => 'Canada',
            'Mexico'        => 'Mexico'
        ],
        'selected' => 'United States'
    ]
];

$form = Form::createFromConfig($fields);

echo $form;
```

which will produce:

```
<form action="/" method="post">
    <fieldset id="pop-form-fieldset-1" class="pop-form-fieldset">
        <dl>
            <dt>
                <label for="colors1">Colors</label>
            </dt>
            <dd>
                <fieldset class="checkbox-fieldset">
                    <input type="checkbox" name="colors[]" id="colors" value="Red"␣
↪class="checkbox" checked="checked" />
                    <span class="checkbox-span">Red</span>
                    <input type="checkbox" name="colors[]" id="colors1" value="Green"␣
↪class="checkbox" checked="checked" />
                    <span class="checkbox-span">Green</span>
```

```
                    <input type="checkbox" name="colors[]" id="colors2" value="Blue"␣
↪class="checkbox" />
                    <span class="checkbox-span">Blue</span>
                </fieldset>
            </dd>
            <dt>
                <label for="country">Country</label>
            </dt>
            <dd>
                <select name="country" id="country">
                    <option value="United States" selected="selected">United States</
↪option>
                    <option value="Canada">Canada</option>
                    <option value="Mexico">Mexico</option>
                </select>
            </dd>
        </dl>
    </fieldset>
</form>
```

Fieldsets

As you've seen in the above examples, the fields that are added to the form object are enclosed in a fieldset group. This can be leveraged to create other fieldset groups as well as give them legends to better define the fieldsets.

```
use Pop\Form\Form;
use Pop\Validator;

$fields1 = [
    'username' => [
        'type'       => 'text',
        'label'      => 'Username',
        'required'   => true,
        'validators' => new Validator\AlphaNumeric(),
        'attributes' => [
            'class' => 'username-field',
            'size'  => 40
        ]
    ],
    'password' => [
        'type'       => 'password',
        'label'      => 'Password',
        'required'   => true,
        'validators' => new Validator\GreaterThanEqual(6),
        'attributes' => [
            'class' => 'password-field',
            'size'  => 40
        ]
    ]
];
$fields2 = [
    'submit' => [
        'type'       => 'submit',
        'value'      => 'SUBMIT',
        'attributes' => [
            'class' => 'submit-btn'
```

```
        ]
    ]
];

$form = Form::createFromConfig($fields1);
$form->getFieldset()->setLegend('First Fieldset');
$form->createFieldset('Second Fieldset');
$form->addFieldsFromConfig($fields2);

echo $form;
```

In the above code, the first set of fields are added to an initial fieldset that's automatically created. After that, if you want to add more fieldsets, you call the `createFieldset` method like above. Then the current fieldset is changed to the newly created one and the next fields are added to that one. You can always change to any other fieldset by using the `setCurrent($i)` method. The above code would render like this:

```
<form action="/" method="post">
    <fieldset id="pop-form-fieldset-1" class="pop-form-fieldset">
        <legend>First Fieldset</legend>
        <dl>
            <dt>
                <label for="username" class="required">Username:</label>
            </dt>
            <dd>
                <input type="text" name="username" id="username" value="" required=
↪"required" size="40" />
            </dd>
            <dt>
                <label for="email" class="required">Email:</label>
            </dt>
            <dd>
                <input type="email" name="email" id="email" value="" required=
↪"required" size="40" />
            </dd>
        </dl>
    </fieldset>
    <fieldset id="pop-form-fieldset-2" class="pop-form-fieldset">
        <legend>Second Fieldset</legend>
        <dl>
            <dd>
                <input type="submit" name="submit" id="submit" value="SUBMIT" />
            </dd>
        </dl>
    </fieldset>
</form>
```

The container elements within the fieldset can be controlled by passing a value to the `$container` parameter. The default is *dl*, but *table*, *div* and *p* are supported as well.

```
$form->createFieldset('Second Fieldset', 'table');
```

Alternately, you can inject an entire fieldset configuration array. The code below is a more simple way to inject the fieldset configurations and their legends. And, it will generate the same HTML as above.

```
use Pop\Form\Form;
use Pop\Validator;
```

```
$fieldsets = [
    'First Fieldset' => [
        'username' => [
            'type'       => 'text',
            'label'      => 'Username',
            'required'   => true,
            'validators' => new Validator\AlphaNumeric(),
            'attributes' => [
                'class' => 'username-field',
                'size'  => 40
            ]
        ],
        'password' => [
            'type'       => 'password',
            'label'      => 'Password',
            'required'   => true,
            'validators' => new Validator\GreaterThanEqual(6),
            'attributes' => [
                'class' => 'password-field',
                'size'  => 40
            ]
        ]
    ],
    'Second Fieldset' => [
        'submit' => [
            'type'       => 'submit',
            'value'      => 'SUBMIT',
            'attributes' => [
                'class' => 'submit-btn'
            ]
        ]
    ]
];

$form = Form::createFromFieldsetConfig($fieldsets);

echo $form;
```

Using Views

You can still use the form object for managing and validating your form fields and still send the individual components to a view for you to control how they render as needed. You can do that like this:

```
use Pop\Form\Form;
use Pop\Validator;

$fields = [
    'username' => [
        'type'       => 'text',
        'label'      => 'Username',
        'required'   => true,
        'validators' => new Validator\AlphaNumeric(),
        'attributes' => [
            'class' => 'username-field',
            'size'  => 40
        ]
```

```
    ],
    'password' => [
        'type'       => 'password',
        'label'      => 'Password',
        'required'   => true,
        'validators' => new Validator\GreaterThanEqual(6),
        'attributes' => [
            'class' => 'password-field',
            'size'  => 40
        ]
    ],
    'submit' => [
        'type'       => 'submit',
        'value'      => 'SUBMIT',
        'attributes' => [
            'class' => 'submit-btn'
        ]
    ]
];

$form = Form::createFromConfig($fields);
$formData = $form->prepareForView();
```

You can then pass the array `$formData` off to your view object to be rendered as you need it to be. That array will contain the following `key => value` entries:

```
$formData = [
    'username'        => '<input type="text" name="username"...',
    'username_label'  => '<label for="username" ...',
    'username_errors' => [],
    'password'        => '<input type="text" name="username"...',
    'password_label'  => '<label for="username" ...',
    'password_errors' => [],
    'submit'          => '<input type="submit" name="submit"...',
    'submit_label'    => '',
]
```

Or, if you want even more control, you can send the form object itself into your view object and access the components like this:

```
<form action="/" method="post" id="login-form">
    <fieldset id="login-form-fieldset-1" class="login-form-fieldset">
        <dl>
            <dt>
                <label for="username" class="required"><?=$form->getField('username')-
↪>getLabel(); ?></label>
            </dt>
            <dd>
                <?=$form->getField('username'); ?>
<?php if ($form->getField('username')->hasErrors(): ?>
<?php foreach ($form->getField('username')->getErrors() as $error): ?>
                <div class="error"><?=$error; ?></div>
<?php endforeach; ?>
<?php endif; ?>
            </dd>
            <dt>
                <label for="password" class="required"><?=$form->getField('password')-
↪>getLabel(); ?></label>
```

```
            </dt>
            <dd>
                <?=$form->getField('password'); ?>
<?php if ($form->getField('password')->hasErrors(): ?>
<?php foreach ($form->getField('password')->getErrors() as $error): ?>
                <div class="error"><?=$error; ?></div>
<?php endforeach; ?>
<?php endif; ?>
            </dd>
            <dd>
                <?=$form->getField('submit'); ?>
            </dd>
        </dl>
    </fieldset>
</form>
```

Images

Image manipulation and processing is another set of features that is often needed for a web application. It is common to have to process images in some way for the web application to perform its required functionality. The *popphp/pop-image* component provides that functionality with a robust set of image processing and manipulation features. Within the component are adapters written to support the `Gd`, `Imagick` and `Gmagick` extensions.

By using either the `Imagick` or `Gmagick` adapters*[0], you will open up a larger set of features and functionality for your application, such as the ability to handle more image formats and perform more complex image processing functions.

Choose an Adapter

Before you choose which image adapter to use, you may have to determine which PHP image extensions are available for your application within its environment. There is an API to assist you with that. The following example tests for each individual adapter to see if one is available, and if not, then moves on to the next:

```
if (Pop\Image\Gmagick::isAvailable()) {
    $image = Pop\Image\Gmagick::load('image.jpg');
} else if (Pop\Image\Imagick::isAvailable()) {
    $image = Pop\Image\Imagick::load('image.jpg');
} else if (Pop\Image\Gd::isAvailable()) {
    $image = Pop\Image\Gd::load('image.jpg');
}
```

Similarly, you can check their availability like this as well:

```
// This will work with any of the 3 adapters
$adapters = Pop\Image\Image::getAvailableAdapters();

if ($adapters['gmagick']) {
    $image = Pop\Image\Gmagick::load('image.jpg');
} else if ($adapters['imagick']) {
    $image = Pop\Image\Imagick::load('image.jpg');
} else if ($adapters['gd']) {
```

[0] It must be noted that the `imagick` and `gmagick` extensions cannot be used at the same time as they have conflicts with shared libraries and components that are used by both extensions.

```
    $image = Pop\Image\Gd::load('image.jpg');
}
```

As far as which adapter or extension is the "best" for your application, that will really depend on your application's needs and what's available in the environment on which your application is running. If you require advanced image processing that can work with a large number of image formats, then you'll need to utilize either the `Imagick` or `Gmagick` adapters. If you only require simple image processing with a limited number of image formats, then the `Gd` adapter should work well.

The point of the API of the *popphp/pop-image* component is to help make applications more portable and mitigate any issues that may arise should an application need to be installed on a variety of different environments. The goal is to achieve a certain amount of "graceful degradation," should one of the more feature-rich image extensions not be available on a new environment.

Basic Use

Loading an Image

You can load an existing image from disk like this:

```
// Returns an instance of Pop\Image\Adapter\Gd with the image resource loaded
$image = Pop\Image\Gd::load('path/to/image.jpg');
```

Or you can load an image from a data source like this:

```
// Returns an instance of Pop\Image\Adapter\Gd with the image resource loaded
$image = Pop\Image\Gd::loadFromString($imageData);
```

Or create an instance of an image object with a new image resource via:

```
// Returns an instance of Pop\Image\Gd with a new image resource loaded
$image =  Pop\Image\Gd::create(640, 480, 'new.jpg');
```

All three of the above adapters have the same core API below:

- `$img->resizeToWidth($w);` - resize the image to a specified width
- `$img->resizeToHeight($h);` - resize the image to a specified height
- `$img->resize($px);` - resize image to largest dimension
- `$img->scale($scale);` - scale image by percentage, 0.0 - 1.0
- `$img->crop($w, $h, $x = 0, $y = 0);` - crop image to specified width and height
- `$img->cropThumb($px, $offset = null);` - crop image to squared image of specified size
- `$img->rotate($degrees, Color\ColorInterface $bgColor = null, $alpha = null);` - rotate image by specified degrees
- `$img->flip();` - flip the image over the x-axis
- `$img->flop();` - flip the image over the y-axis
- `$img->convert($to);` - convert image to specified image type
- `$img->writeToFile($to = null, $quality = 100);` - save image, either to itself or a new location
- `$img->outputToHttp($quality = 100, $to = null, $download = false, $sendHeaders = true);` - output image via HTTP

Advanced Use

The *popphp/pop-image* component comes with set of image manipulation objects that provide a more advanced feature set when processing images. You can think of these classes and their object instances as the menus at the top of your favorite image editing software.

Adjust

The adjust object allows you to perform the following methods:

- `$img->adjust->brightness($amount);`
- `$img->adjust->contrast($amount);`
- `$img->adjust->desaturate();`

And with the `Imagick` or `Gmagick` adapter, you can perform these advanced methods:

- `$img->adjust->hue($amount);`
- `$img->adjust->saturation($amount);`
- `$img->adjust->hsb($h, $s, $b);`
- `$img->adjust->level($black, $gamma, $white);`

Here's an example making some adjustments to the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->adjust->brightness(50)
    ->contrast(20)
    ->level(0.7, 1.0, 0.5);
```

Draw

The draw object allows you to perform the following methods:

- `$img->draw->line($x1, $y1, $x2, $y2);`
- `$img->draw->rectangle($x, $y, $w, $h = null);`
- `$img->draw->square($x, $y, $w);`
- `$img->draw->ellipse($x, $y, $w, $h = null);`
- `$img->draw->circle($x, $y, $w);`
- `$img->draw->arc($x, $y, $start, $end, $w, $h = null);`
- `$img->draw->chord($x, $y, $start, $end, $w, $h = null);`
- `$img->draw->pie($x, $y, $start, $end, $w, $h = null);`
- `$img->draw->polygon($points);`

And with the `Imagick` or `Gmagick` adapter, you can perform these advanced methods:

- `$img->draw->roundedRectangle($x, $y, $w, $h = null, $rx = 10, $ry = null);`
- `$img->draw->roundedSquare($x, $y, $w, $rx = 10, $ry = null);`

Here's an example drawing some different shapes with different styles on the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->draw->setFillColor(255, 0, 0);
    ->draw->setStrokeColor(0, 0, 0);
    ->draw->setStrokeWidth(5);
    ->draw->rectangle(100, 100, 320, 240);
    ->draw->circle(400, 300, 50);
```

Effect

The effect object allows you to perform the following methods:

- `$img->effect->border(array $color, $w, $h = null);`
- `$img->effect->fill($r, $g, $b);`
- `$img->effect->radialGradient(array $color1, array $color2);`
- `$img->effect->verticalGradient(array $color1, array $color2);`
- `$img->effect->horizontalGradient(array $color1, array $color2);`
- `$img->effect->linearGradient(array $color1, array $color2, $vertical = true);`

Here's an example applying some different effects to the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->effect->verticalGradient([255, 0, 0], [0, 0, 255]);
```

Filter

Each filter object is more specific for each image adapter. While a number of the available filter methods are available in all 3 of the image adapters, some of their signatures vary due the requirements of the underlying image extension.

The `Gd` filter object allows you to perform the following methods:

- `$img->filter->blur($amount, $type = IMG_FILTER_GAUSSIAN_BLUR);`
- `$img->filter->sharpen($amount);`
- `$img->filter->negate();`
- `$img->filter->colorize($r, $g, $b);`
- `$img->filter->pixelate($px);`
- `$img->filter->pencil();`

The `Imagick` filter object allows you to perform the following methods:

- `$img->filter->blur($radius = 0, $sigma = 0, $channel = \Imagick::CHANNEL_ALL);`
- `$img->filter->adaptiveBlur($radius = 0, $sigma = 0, $channel = \Imagick::CHANNEL_DEFAULT);`
- `$img->filter->gaussianBlur($radius = 0, $sigma = 0, $channel = \Imagick::CHANNEL_ALL);`
- `$img->filter->motionBlur($radius = 0, $sigma = 0, $angle = 0, $channel = \Imagick::CHANNEL_DEFAULT);`

- `$img->filter->radialBlur($angle = 0, $channel = \Imagick::CHANNEL_ALL);`
- `$img->filter->sharpen($radius = 0, $sigma = 0, $channel = \Imagick::CHANNEL_ALL);`
- `$img->filter->negate();`
- `$img->filter->paint($radius);`
- `$img->filter->posterize($levels, $dither = false);`
- `$img->filter->noise($type = \Imagick::NOISE_MULTIPLICATIVEGAUSSIAN, $channel = \Imagick::CHANNEL_DEFAULT);`
- `$img->filter->diffuse($radius);`
- `$img->filter->skew($x, $y, $color = 'rgb(255, 255, 255)');`
- `$img->filter->swirl($degrees);`
- `$img->filter->wave($amp, $length);`
- `$img->filter->pixelate($w, $h = null);`
- `$img->filter->pencil($radius, $sigma, $angle);`

The `Gmagick` filter object allows you to perform the following methods:

- `$img->filter->blur($radius = 0, $sigma = 0, $channel = \Gmagick::CHANNEL_ALL);`
- `$img->filter->motionBlur($radius = 0, $sigma = 0, $angle = 0);`
- `$img->filter->radialBlur($angle = 0, $channel = \Gmagick::CHANNEL_ALL);`
- `$img->filter->sharpen($radius = 0, $sigma = 0, $channel = \Gmagick::CHANNEL_ALL);`
- `$img->filter->negate();`
- `$img->filter->paint($radius);`
- `$img->filter->noise($type = \Gmagick::NOISE_MULTIPLICATIVEGAUSSIAN);`
- `$img->filter->diffuse($radius);`
- `$img->filter->skew($x, $y, $color = 'rgb(255, 255, 255)');`
- `$img->filter->solarize($threshold);`
- `$img->filter->swirl($degrees);`
- `$img->filter->pixelate($w, $h = null);`

Here's an example applying some different filters to the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->filter->gaussianBlur(10)
    ->swirl(45)
    ->negate();
```

Layer

The layer object allows you to perform the following methods:

- `$img->layer->overlay($image, $x = 0, $y = 0);`

And with the `Imagick` or `Gmagick` adapter, you can perform this advanced method:

- `$img->layer->flatten();`

Here's an example working with layers over the image resource:

```
$img = new Pop\Image\Imagick('image.psd');
$img->layer->flatten()
    ->overlay('watermark.png', 50, 50);
```

Type

The type object allows you to perform the following methods:

- `$img->type->font($font);` - set the font
- `$img->type->size($size);` - set the font size
- `$img->type->x($x);` - set the x-position of the text string
- `$img->type->y($y);` - set the y-position of the text string
- `$img->type->xy($x, $y);` - set both the x- and y-position together
- `$img->type->rotate($degrees);` - set the amount of degrees in which to rotate the text string
- `$img->type->text($string);` - place the string on the image, using the defined parameters

Here's an example working with text over the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->type->setFillColor(128, 128, 128)
    ->size(12)
    ->font('fonts/Arial.ttf')
    ->xy(40, 120)
    ->text('Hello World!');
```

Extending the Component

The *popphp/pop-image* component was built in a way to facilitate extending it and injecting your own custom image processing features. Knowing that the image processing landscape is vast, the component only scratches the surface and provides the core feature set that was outlined above across the different adapters.

If you are interested in creating and injecting your own, more robust set of features into the component within your application, you can do that by extending the available manipulation classes.

For example, if you wanted to add a couple of methods to the adjust class for the `Gd` adapter, you can do so like this:

```
namespace MyApp\Image;

class CustomAdjust extends \Pop\Image\Adjust\Gd
{
    public function customAction1() {}

    public function customAction2() {}

    public function customAction3() {}
}
```

Then, later in your application, when you call up the `Gd` adapter, you can inject your custom adjust adapter like this:

```
namespace MyApp;

$image = new \Pop\Image\Gd\('image.jpg');
$image->setAdjust(new MyApp\Image\CustomAdjust());
```

So when you go you use the image adapter, your custom features will be available along will the original set of features:

```
$image->adjust->brightness(50)
    ->customAction1()
    ->customAction2()
    ->customAction3();
```

This way, you can create and call whatever custom features are needed for your application on top of the basic features that are already available.

PDFs

PDF generation in an application is typically a required feature for any application that does any type of in-depth reporting or data exporting. Many applications may require this exported data to be in a concise, well-formatted and portable document and PDF provides this.

The PDF specification, as well as its shared assets' specifications, such as fonts and images, are an extremely large and complex set of rules and syntax. This component attempts to harness the power and features defined by those specifications and present an intuitive API that puts the power of PDF at your fingertips.

Building a PDF

At the center of the *popphp/pop-pdf* component is the main `Pop\Pdf\Pdf` class. It serves as a manager or controller of sorts for all of the various PDF assets that will pass through during the process of PDF generation. The different assets are each outlined with their own section below.

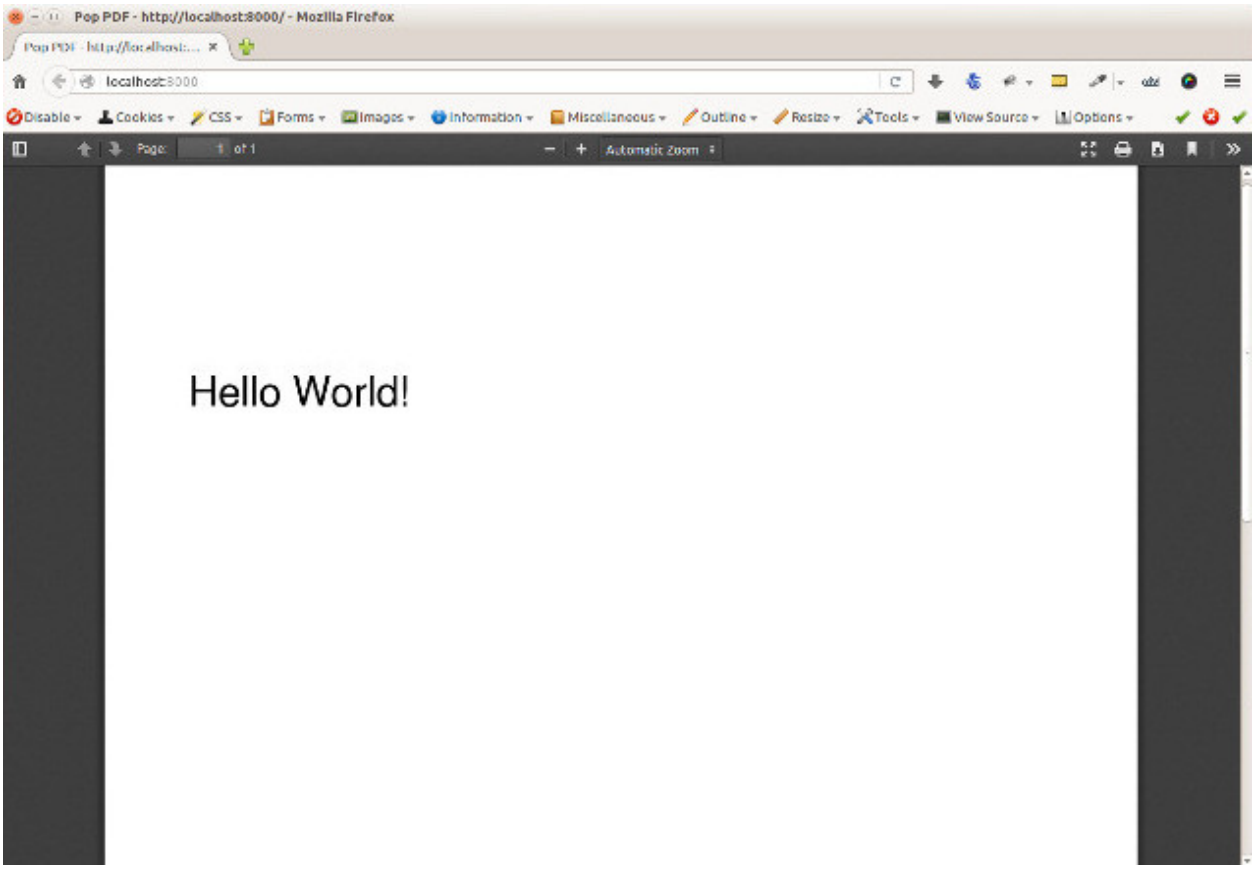
Here's a simple example building and generating a PDF document with some text. The finer points of what's happening will be explained more in depth in the later sections.

```
use Pop\Pdf\Pdf;
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;
use Pop\Pdf\Document\Page;

// Create a page and add the text to it
$page = new Page(Page::LETTER);
$page->addText(new Page\Text('Hello World!', 24), Font::ARIAL, 50, 650);

// Create a document, add the font to it and then the page
$document = new Document();
$document->addFont(new Font(Font::ARIAL));
$document->addPage($page);

// Pass the document to the Pdf object to build it and output it to HTTP
$pdf = new Pdf();
$pdf->outputToHttp($document);
```



Importing a PDF

Importing an existing PDF or pages from one may be required in your application. Using the main PDF object, you can specify the pdf to import as well as the select pages you may wish to import. From there you can either select pages to modify or add new pages to the document. When you do import an existing PDF, the method will return a parsed and working document object. In the example below, we will import pages 2 and 5 from the existing PDF, add a new page in between them and then save the new document:

```
use Pop\Pdf\Pdf;
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;
use Pop\Pdf\Document\Page;

$pdf = new Pdf();
$document = $pdf->importFromFile('doc.pdf', [2, 5])

// Create a page and add the text to it
$page = new Page(Page::LETTER);
$page->addText(new Page\Text('Hello World!', 24), Font::ARIAL, 50, 650);

// Create a document, add the font to it and then the page
$document = new Document();
$document->addFont(new Font(Font::ARIAL));
$document->addPage($page);
$document->orderPages([1, 3, 2]); // 3 being our new page.

// Pass the document to the Pdf object to build it and write it to a new file
```

```
$pdf = new Pdf();
$pdf->writeToFile('new-doc.pdf');
```

When the 2 pages are imported in, they default to page 1 and 2, respectively. Then we can add any pages we need from there and control the final order of the pages with the `orderPages` method like in the above example.

If you wish to import the whole PDF and all of its pages, simply leave the `$pages` parameter blank.

Coordinates

It should be noted that the PDF coordinate system has its origin (0, 0) at the bottom left. In the example above, the text was placed at the (x, y) coordinate of (50, 650). When placed on a page that is set to the size of a letter, which is 612 points x 792 points, that will make the text appear in the top left. It the coordinates of the text were set to (50, 50) instead, the text would have appeared in the bottom left.

As this coordinate system may or may not suit a developer's personal preference or the requirements of the application, the origin point of the document can be set using the following method:

```
use Pop\Pdf\Document;

$document = new Document();
$document->setOrigin(Document::ORIGIN_TOP_LEFT);
```

Now, with the document's origin set to the top left, when you place assets into the document, you can base it off of the new origin point. So for the text in the above example to be placed in the same place, the new (x, y) coordinates would be (50, 142).

Alternatively, the full list of constants in the `Pop\Pdf\Document` class that represent the different origins are:

- ORIGIN_TOP_LEFT
- ORIGIN_TOP_RIGHT
- ORIGIN_BOTTOM_LEFT
- ORIGIN_BOTTOM_RIGHT
- ORIGIN_CENTER

Documents

A document object represents the top-level "container" object of the the PDF document. As you create the various assets that are to be placed in the PDF document, you will inject them into the document object. At the document level, the main assets that can be added are **fonts**, **forms**, **metadata** and **pages**. The font and form objects are added at the document level as they can be re-used on the page level by other assets. The metadata object contains informational data about the document, such as title and author. And the page objects contain all of the page-level assets, as detailed below.

Fonts

Font objects are the global document objects that contain information about the fonts that can be used by the text objects within the pages of the document. A font can either be one of the standard fonts supported by PDF natively, or an embedded font from a font file.

Standard Fonts

The set of standard, native PDF fonts include:

- Arial
- Arial,Italic
- Arial,Bold
- Arial,BoldItalic
- Courier
- Courier-Oblique
- Courier-Bold
- Courier-BoldOblique
- CourierNew
- CourierNew,Italic
- CourierNew,Bold
- CourierNew,BoldItalic
- Helvetica
- Helvetica-Oblique
- Helvetica-Bold
- Helvetica-BoldOblique
- Symbol
- Times-Roman
- Times-Bold
- Times-Italic
- Times-BoldItalic
- TimesNewRoman
- TimesNewRoman,Italic
- TimesNewRoman,Bold
- TimesNewRoman,BoldItalic
- ZapfDingbats

When adding a standard font to the document, you can add it and then reference it by name throughout the building of the PDF. For reference, there are constants available in the `Pop\Pdf\Document\Font` class that have the correct standard font names stored in them as strings.

```
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;

$font = new Font(Font::TIMES_NEW_ROMAN_BOLDITALIC);

$document = new Document();
$document->addFont($font);
```

Now, the font defined as "TimesNewRoman,BoldItalic" is available to the document and for any text for which you need it.

Embedded Fonts

The embedded font types that are supported are:

- TrueType
- OpenType
- Type1

When embedding an external font, you will need access to its name to correctly reference it by string much in the same way you do for a standard font. That name becomes accessible once you create a font object with an embedded font and it is successfully parsed.

Notice about embedded fonts

There may be issues embedding a font if certain font data or font files are missing, incomplete or corrupted. Furthermore, there may be issues embedding a font if the correct permissions or licensing are not provided.

```
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;
use Pop\Pdf\Document\Page;

$customFont = new Font('custom-font.ttf');

$document = new Document();
$document->embedFont($customFont);

$text = new Page\Text('Hello World!', 24);

$page = new Page(Page::LETTER);
$page->addText($text, $customFont->getName(), 50, 650);
```

The above example will attach the name and reference of the embedded custom font to that text object. Additionally, when a font is added or embedded into a document, its name becomes the current font, which is a property you can access like this:

```
$page->addText($text, $document->getCurrentFont(), 50, 650);
```

If you'd like to override or switch the current document font back to another font that's available, you can do so like this:

```
$document->setCurrentFont('Arial');
```

Forms

Form objects are the global document objects that contain information about fields that are to be used within a Form object on a page in the document. By themselves they are fairly simple to use and inject into a document object. From there, you would add fields to a their respective pages, while attaching them to a form object.

The example below demonstrates how to add a form object to a document:

```
use Pop\Pdf\Document;
use Pop\Pdf\Document\Form;

$form = new Form('contact_form');

$document = new Document();
$document->addForm($form);
```

Then, when you add a field to a page, you can reference the form to attach it to:

```
use Pop\Pdf\Document\Page;

$name = new Page\Field\Text('name');
$name->setWidth(200)
     ->setHeight(40);

$page = new Page(Page::LETTER);
$page->addField($name, 'contact_form', 50, 650);
```

The above example creates a name field for the contact form, giving it a width and height and placing it at the (50, 650) coordinated. *Fields* will be covered more in depth below.

Metadata

The metadata object contains the document identifier data such as title, author and date. This is the data that is commonly displayed in the the document title bar and info boxes of a PDF reader. If you'd like to set the metadata of the document, you can with the following API:

```
use Pop\Pdf\Document;

$metadata = new Document\Metadata();
$metadata->setTitle('My Document')
    ->setAuthor('Some Author')
    ->setSubject('Some Subject')
    ->setCreator('Some Creator')
    ->setProducer('Some Producer')
    ->setCreationDate('August 19, 2015')
    ->setModDate('August 22, 2015')

$document = new Document();
$document->setMetadata($metadata);
```

And there are getter methods that follow the same naming convention to retrieve the data from the metadata object.

Pages

Page objects contain the majority of the assets that you would expect to be on a page within a PDF document. A page's size can be either custom-defined or one of the predefined sizes. There are constants that define those predefine sizes for reference:

- ENVELOPE_10 (297 x 684)
- ENVELOPE_C5 (461 x 648)
- ENVELOPE_DL (312 x 624)
- FOLIO (595 x 935)
- EXECUTIVE (522 x 756)
- LETTER (612 x 792)
- LEGAL (612 x 1008)
- LEDGER (1224 x 792)
- TABLOID (792 x 1224)
- A0 (2384 x 3370)

- A1 (1684 x 2384)
- A2 (1191 x 1684)
- A3 (842 x 1191)
- A4 (595 x 842)
- A5 (420 x 595)
- A6 (297 x 420)
- A7 (210 x 297)
- A8 (148 x 210)
- A9 (105 x 148)
- B0 (2920 x 4127)
- B1 (2064 x 2920)
- B2 (1460 x 2064)
- B3 (1032 x 1460)
- B4 (729 x 1032)
- B5 (516 x 729)
- B6 (363 x 516)
- B7 (258 x 363)
- B8 (181 x 258)
- B9 (127 x 181)
- B10 (91 x 127)

```
use Pop\Pdf\Document\Page;

$legal  = new Page(Page::LEGAL);
$custom = new Page(640, 480);
```

The above example creates two pages - one legal size and one a custom size of 640 x 480.

Images

An image object allows you to place an image onto a page in the PDF document, as well as control certain aspects of that image, such as size and resolution. The image types that are supported are:

- JPG (RGB, CMYK or Grayscale)
- PNG (8-Bit Index)
- PNG (8-Bit Index w/ Transparency)
- PNG (24-Bit RGB or Grayscale)
- GIF (8-Bit Index)
- GIF (8-Bit Index w/ Transparency)

Here is an example of embedding a large image and resizing it down before placing on the page:

```
use Pop\Pdf\Document\Page;

$image = new Page\Image('large-image.jpg');
$image->resizeToWidth(320);

$page = new Page(Page::LETTER);
$page->addImage($image, 50, 650);
```

In the above example, the large image is processed (down-sampled) and resized to a width of 320 pixels and placed into the page at the coordinates of (50, 650).

If you wanted to preserve the image's high resolution, but fit it into the smaller dimensions, you can do that by setting the `$preserveResolution` flag in the resize method.

```
$image->resizeToWidth(320, true);
```

This way, the high resolution image is not processed or down-sampled and keeps its high quality. It is only placed into scaled down dimensions.

Color

With path and text objects, you will need to set colors to render them correctly. The main 3 colorspaces that are supported are RGB, CMYK and Grayscale. Each color space object is created by instantiating it and passing the color values:

```
use Pop\Pdf\Document\Page\Color;

$red  = new Color\Rgb(255, 0, 0);      // $r, $g, $b (0 - 255)
$cyan = new Color\Cmyk(100, 0, 0, 0); // $c, $m, $y, $k (0 - 100)
$gray = new Color\Gray(50);            // $gray (0 - 100)
```

These objects are then passed into the methods that consume them, like `setFillColor` and `setStrokeColor` within the path and text objects.

Paths

Since vector graphics are at the core of PDF, the path class contains a robust API that allows you to no only draw various paths and shapes, but also set their colors and styles. On instantiation, you can set the style of the path object:

```
use Pop\Pdf\Document\Page\Path;
use Pop\Pdf\Document\Page\Color\Rgb;

$path = new Path(Path::FILL_STROKE);
$path->setFillColor(new Rgb(255, 0, 0))
     ->setStrokeColor(new Rgb(0, 0, 0))
     ->setStroke(2);
```

The above example created a path object with the default style of fill and stroke, and set the fill color to red, the stroke color to black and the stroke width to 2 points. That means that any paths that are drawn from here on out will have those styles until they are changed. You can create and draw more than one path or shape with in path object. The path class has constants that reference the different style types you can set:

- STROKE
- STROKE_CLOSE

- FILL
- FILL_EVEN_ODD
- FILL_STROKE
- FILL_STROKE_EVEN_ODD
- FILL_STROKE_CLOSE
- FILL_STROKE_CLOSE_EVEN_ODD
- CLIPPING
- CLIPPING_FILL
- CLIPPING_NO_STYLE
- CLIPPING_EVEN_ODD
- CLIPPING_EVEN_ODD_FILL
- CLIPPING_EVEN_ODD_NO_STYLE
- NO_STYLE

From there, this is the core API that is available:

- `$path->setStyle($style);`
- `$path->setFillColor(Color\ColorInterface $color);`
- `$path->setStrokeColor(Color\ColorInterface $color);`
- `$path->setStroke($width, $dashLength = null, $dashGap = null);`
- `$path->openLayer();`
- `$path->closeLayer();`
- `$path->drawLine($x1, $y1, $x2, $y2);`
- `$path->drawRectangle($x, $y, $w, $h = null);`
- `$path->drawRoundedRectangle($x, $y, $w, $h = null, $rx = 10, $ry = null);`
- `$path->drawSquare($x, $y, $w);`
- `$path->drawRoundedSquare($x, $y, $w, $rx = 10, $ry = null);`
- `$path->drawPolygon($points);`
- `$path->drawEllipse($x, $y, $w, $h = null);`
- `$path->drawCircle($x, $y, $w);`
- `$path->drawArc($x, $y, $start, $end, $w, $h = null);`
- `$path->drawChord($x, $y, $start, $end, $w, $h = null);`
- `$path->drawPie($x, $y, $start, $end, $w, $h = null);`
- `$path->drawOpenCubicBezierCurve($x1, $y1, $x2, $y2, $bezierX1, $bezierY1, $bezierX2, $bezierY2);`
- `$path->drawClosedCubicBezierCurve($x1, $y1, $x2, $y2, $bezierX1, $bezierY1, $bezierX2, $bezierY2);`
- `$path->drawOpenQuadraticBezierCurve($x1, $y1, $x2, $y2, $bezierX, $bezierY, $first = true);`

- `$path->drawClosedQuadraticBezierCurve($x1, $y1, $x2, $y2, $bezierX, $bezierY, $first = true);`

Extending the original code example above, here is an example of drawing a rectangle and placing it on a page:

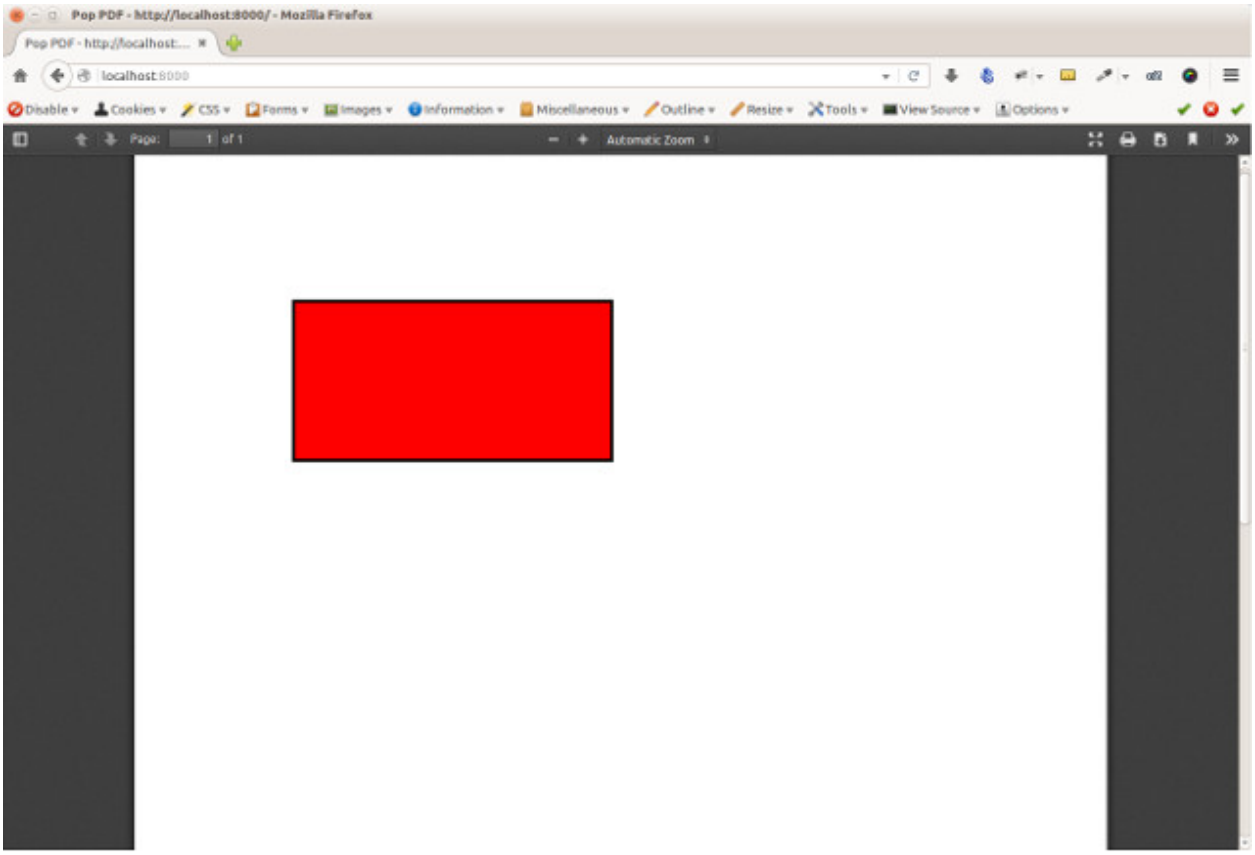
```
use Pop\Pdf\Pdf;
use Pop\Pdf\Document;
use Pop\Pdf\Document\Page;
use Pop\Pdf\Document\Page\Path;
use Pop\Pdf\Document\Page\Color\Rgb;

// Create a path object, set the styles and draw a rectangle
$path = new Path(Path::FILL_STROKE);
$path->setFillColor(new Rgb(255, 0, 0))
     ->setStrokeColor(new Rgb(0, 0, 0))
     ->setStroke(2)
     ->drawRectangle(100, 600, 200, 100);

// Create a page and add the path to it
$page = new Page(Page::LETTER);
$page->addPath($path);

// Create a document and add the page
$document = new Document();
$document->addPage($page);

// Pass the document to the Pdf object to build it and output it to HTTP
$pdf = new Pdf();
$pdf->outputToHttp($document);
```



Layers

As the API shows, you can also layer paths using the `openLayer()` and `closeLayer()` methods which open and close an independent graphics state. Any paths added while in this state will render onto that "layer." Any paths rendered after the state is closed will render above that layer.

Clipping Paths

The path object also supports clipping paths via setting the path style to a clipping style. In doing so, the path will render as a clipping path or "mask" over any paths before it.

Text

With text objects, you can control a number of parameters that affect how the text is displayed beyond which font is used and the size. As with path objects, you can set color and style, as well as a few other parameters. As one of the above examples demonstrated, you can create a text object like this:

```
use Pop\Pdf\Document\Page;

$text = new Page\Text('Hello World!', 24);

// Create a page and add the text to it
$page = new Page(Page::LETTER);
$page->addText($text, 'Arial', 50, 650);
```

The above code create a text object with the font size of 24 points and added it to a page using the Arial font. From there, you can do more with the text object API. Here is what the API looks like for a text object:

- `$text->setFillColor(Color\ColorInterface $color);`
- `$text->setStrokeColor(Color\ColorInterface $color);`
- `$text->setStroke($width, $dashLength = null, $dashGap = null);`
- `$test->setWrap($wrap, $lineHeight = null);`
- `$test->setLineHeight($lineHeight);`
- `$test->setRotation($rotation);`
- `$test->setTextParams($c = 0, $w = 0, $h = 100, $v = 100, $rot = 0, $rend = 0);`

With the `setTextParams()` method, you can set the following render parameters:

- `$c` - character spacing
- `$w` - word spacing
- `$h` - horizontal stretch
- `$v` - vertical stretch
- `$rot` - rotation in degrees
- `$rend` - render mode 0 - 7;
  - 0 - Fill
  - 1 - Stroke
  - 2 - Fill and stroke
  - 3 - Invisible

- 4 - Fill then use for clipping
- 5 - Stroke the use for clipping
- 6 - Fill and stroke and use for clipping
- 7 - Use for clipping

Extending the example above, we can render red text to the page like this:

```
use Pop\Pdf\Pdf;
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;
use Pop\Pdf\Document\Page;

// Create the text object and set the fill color
$text = new Page\Text('Hello World!', 24);
$text->setFillColor(new Rgb(255, 0, 0));

// Create a page and add the text to it
$page = new Page(Page::LETTER);
$page->addText($text, Font::ARIAL, 50, 650);

// Create a document, add the font to it and then the page
$document = new Document();
$document->addFont(new Font(Font::ARIAL));
$document->addPage($page);

// Pass the document to the Pdf object to build it and output it to HTTP
$pdf = new Pdf();
$pdf->outputToHttp($document);
```

Wrap and Line-height

The `setWrap` and `setLineHeight()` methods help facilitate larger blocks of text that you might add to the PDF page. By setting values with these two methods, you give the PDF page the parameters needed to calculate wrapping the large body of text with the proper line-height for you, instead of you having to break the text up and place it manually.

Annotations

Annotation objects give you the functionality to add internal document links and external web links to the page. At the base of an annotation object, you would set the width and height of the annotation's click area or "hot spot." For an internal annotation, you would pass in a set of target coordinates as well:

```
use Pop\Pdf\Document\Page\Annotation;

$link = new Annotation\Link(200, 25, 50, 650); // $width, $height, $xTarget, $yTarget
```

In the above example, an internal annotation object that is 200 x 25 in width and height has been created and is linked to the coordinates of (50, 650) on the current page. If you'd like to target coordinates on a different page, you can set that as well:

```
$link->setPageTarget(3);
```

And if you would like to zoom in on the target, you can set the Z target as well:

```
$link->setZTarget(2);
```

For external URL annotations, instead of an internal set of coordinates, you would pass the URL into the constructor:

```
use Pop\Pdf\Document\Page\Annotation;

$link = new Annotation\Url(200, 25, 'http://www.mywebsite.com/');
```

The above example will create an external annotation link that, when clicked, will link out to the URL given.

Fields

As mentioned earlier, field objects are the entities that collect user input and attach that data to form objects. The benefit of this is the ability to save user input within the document. The field types that are supported are:

- Text (single and multi-line)
- Choice
- Button

Here is an example creating a simple set of fields and attaching them to a form object:

```
use Pop\Pdf\Document;
use Pop\Pdf\Document\Form;
use Pop\Pdf\Document\Page;

// Create the form object and inject it into the document object
$form = new Form('contact_form');

$document = new Document();
$document->addForm($form);

$name = new Page\Field\Text('name');
$name->setWidth(200)
     ->setHeight(40);

$colors = new Page\Field\Choice('colors');
$colors->addOption('Red')
    ->addOption('Green')
    ->addOption('Blue')

$comments = new Page\Field\Text('comments');
$comments->setWidth(200)
     ->setHeight(100)
     ->setMultiline();

$page = new Page(Page::LETTER);
$page->addField($name, 'contact_form', 50, 650)
     ->addField($colors, 'contact_form', 50, 600)
     ->addField($comments, 'contact_form', 50, 550);
```

In the above example, the fields are created, attached to the form object and added to the page object.

Popcorn

The Popcorn PHP Micro-Framework is a lightweight REST-based micro-framework that's built on top of the Pop PHP Framework core components. With it, you can rapidly wire together the routes and configuration needed for your REST-based web application, while leveraging the pre-existing features and functionality of the Pop PHP Framework.

It provides a simple layer on top of the main `Pop\Application` object that allows you to wire up routes and enforce their access based on the request method. By default, it ships with `popphp/popphp`, `popphp/pop-http`, `popphp/pop-session` and `popphp/pop-view` components.

Basic Use

In a simple `index.php` file, you can define the routes you want to allow in your application. In this example, we'll use simple closures as our controllers. The wildcard route '*' can serve as a "catch-all" to handle routes that are not found or not allowed.

```
use Popcorn\Pop;

$app = new Pop();

// Home page: http://localhost:8000/
$app->get('/', function() {
    echo 'Hello World!';
});

// Say hello page: http://localhost:8000/hello/john
$app->get('/hello/:name', function($name) {
    echo 'Hello ' . ucfirst($name) . '!';
});

// Wildcard route to handle errors
$app->get('*', function() {
    header('HTTP/1.1 404 Not Found');
    echo 'Page Not Found.';
});

// Post route to process an auth request
$app->post('/auth', function() {
    if ($_SERVER['HTTP_AUTHORIZATION'] == 'my-token') {
        echo 'Auth successful';
    } else {
        echo 'Auth failed';
    }
});

$app->run();
```

In the above POST example, if you attempted access that URL via GET (or any method that wasn't POST), it would fail. If you access that URL via POST, but with the wrong application token, it will return the 'Auth failed' message as enforced by the application. Access the URL via POST with the correct application token, and it will be successful:

```
curl -X POST --header "Authorization: bad-token" http://localhost:8000/auth
Auth failed

curl -X POST --header "Authorization: my-token" http://localhost:8000/auth
Auth successful
```

Advanced Usage

In a more advanced example, we can take advantage of more of an MVC-style of wiring up an application using the core components of Pop PHP with Popcorn. Keeping it simple, let's look at a controller class

`MyApp\Controller\IndexController` like this:

```
namespace MyApp\Controller;

use Pop\Controller\AbstractController;
use Pop\Http\Request;
use Pop\Http\Response;
use Pop\View\View;

class IndexController extends AbstractController
{

    protected $response;
    protected $viewPath;

    public function __construct()
    {
        $this->request = new Request();
        $this->response = new Response();
        $this->viewPath = __DIR__ . '/../view/';
    }

    public function index()
    {
        $view = new View($this->viewPath . '/index.phtml');
        $view->title = 'Welcome';

        $this->response->setBody($view->render());
        $this->response->send();
    }

    public function error()
    {
        $view = new View($this->viewPath . '/error.phtml');
        $view->title =  'Error';

        $this->response->setBody($view->render());
        $this->response->send(404);
    }

}
```

and two view scripts, `index.phtml` and `error.phtml`, respectively:

```
<!DOCTYPE html>
<!-- index.phtml //-->
<html>

<head>
    <title><?=$title; ?></title>
</head>

<body>
    <h1><?=$title; ?></h1>
    <p>Hello World.</p>
</body>

</html>
```

```
<!DOCTYPE html>
<!-- error.phtml //-->
<html>

<head>
    <title><?=$title; ?></title>
</head>

<body>
    <h1 style="color: #f00;"><?=$title; ?></h1>
    <p>Sorry, that page was not found.</p>
</body>

</html>
```

Then we can set the app like this:

```
use Popcorn\Pop;

$app = new Pop();

$app->get('/', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'index',
    'default'    => true
]);

$app->run();
```

The 'default' parameter sets the controller as the default controller to handle routes that aren't found. Typically, there is a default action such as an 'error' method to handle this.

API Overview

Here is an overview of the available API within the module `Popcorn\Pop` class:

- `get($route, $controller)` - Set a GET route
- `head($route, $controller)` - Set a HEAD route
- `post($route, $controller)` - Set a POST route
- `put($route, $controller)` - Set a PUT route
- `delete($route, $controller)` - Set a DELETE route
- `trace($route, $controller)` - Set a TRACE route
- `options($route, $controller)` - Set an OPTIONS route
- `connect($route, $controller)` - Set a CONNECT route
- `patch($route, $controller)` - Set a PATCH route
- `setRoute($method, $route, $controller)` - Set a specific route
- `setRoutes($methods, $route, $controller)` - Set a specific route and apply to multiple methods at once

The `setRoutes()` method allows you to set a specific route and apply it to multiple methods all at once, like this:

```
use Popcorn\Pop;

$app = new Pop();

$app->setRoutes('get,post', '/login', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'login'
]);

$app->run();
```

In the above example, the route `/login` would display the login form on GET, and then submit the form on POST, processing and validating it.

CHAPTER 5

Tutorial Application

This section of the documentation will step you through the tutorial application to demonstrate a few of the various concepts explained the previous sections. The tutorial application will step you through a basic MVC application with some simple CRUD functionality using a small SQLite database. Contained within the application is a demonstration of both a web-based and a console-based application.

There is also a more stripped down and traditional skeleton application that you can install via composer or from GitHub if you are looking for more of a basic foundation on which to begin directly coding your application using Pop.

Get the Tutorial Application

You can get the Pop PHP Framework tutorial application over via composer or directly from Github.

Install via composer:

```
composer create-project popphp/popphp-tutorial project-folder
cd project-folder
sudo php -S localhost:8000 -t public
```

If you run the commands above, you should be able to see Pop welcome screen in your browser at the `http://localhost:8000` address.

Install via git:

You can clone the repository directly and install it and play around with it there:

```
git clone https://github.com/popphp/popphp-tutorial.git
cd popphp-tutorial
composer install
sudo php -S localhost:8000 -t public
```

Again, running the above commands, you should be able to visit the Pop welcome screen in your browser.

Application Structure

After installation, if you take a look at the files and folders in the tutorial application, you'll see the following basic application structure:

- app/
  - config/
  - database/
  - src/
  - view/
- public/
- script/
- vendor/

The `app/` folder contains the main set of folders and files that make the application work. The `database/` folder contains the SQLite database file. The `public/` folder is the web document root that contains the main index file. And the `script/` folder contains the main script to execute the CLI side of the application.

A more expanded version of an Pop application structure may look like this:

- app/
  - config/
    - forms/ *(Form and form field configurations)*
    - resources/ *(ACL resources and permissions)*
    - routes/ *(Route configurations)*
      - web.php
      - cli.php
    - app.web.php
    - app.cli.php
  - database/ *(Database structure and seed files; database migrations)*
    - migrations/
    - app.sql
  - src/ *(Main application source files)*
    - Controller/
    - Form/
    - Model/
    - Table/
    - Module.php
  - view/ *(Application view scripts)*
- data/ *(Application data store for logs, files, etc.)*
  - logs/

    - tmp/
- logs/ *(Web server log files)*
- public/ *(Web server document root)*
- script/ *(CLI-based application scripts)*
- tests/ *(Application tests)*
- vendor/ *(3rd-party vendor packages)*

This structure isn't necessarily set in stone, but it follows a typical structure that one might expect to see within a PHP application.

Application Module

Application development with Pop PHP promotes modular development, meaning that it aides creating smaller "mini-application" modules that can be registered with the main application object. If you look inside the `Tutorial\Module` class, you see the lines:

```
$this->application->on('app.init', function($application){
    Record::setDb($application->services['database']);
});
```

Once those lines of code are executed upon the `app.init` event trigger, the database becomes available to the rest of the application. Furthermore, you'll see a CLI specific header and footer that is only triggered if the environment is on the console.

Front Controllers

You can see the main front controllers in the `public/` and `scripts/` folders: `public/index.php` and `script/app` respectively. Looking into each of those, you can see that the main `Pop\Application` object is created and wired up, with the `Tutorial\Module` object being registered with the main application object so that it will be wired up and function correctly. We will look at these more in-depth in the next sections.

Application Classes

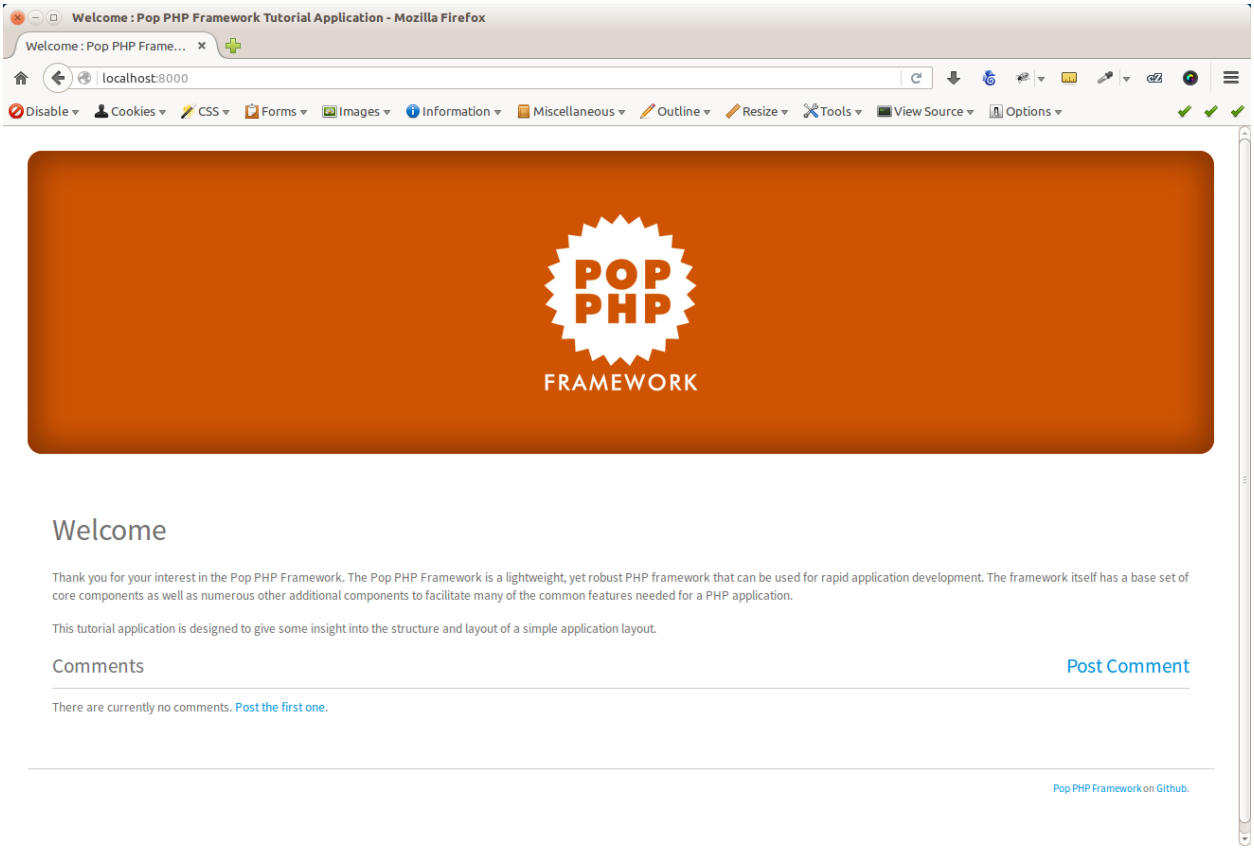
Beyond the front controllers and the main module class, there are classes for the following, each in its own folder:

- controllers - `app/src/Controller`
- forms - `app/src/Form`
- models - `app/src/Model`
- tables - `app/src/Table`

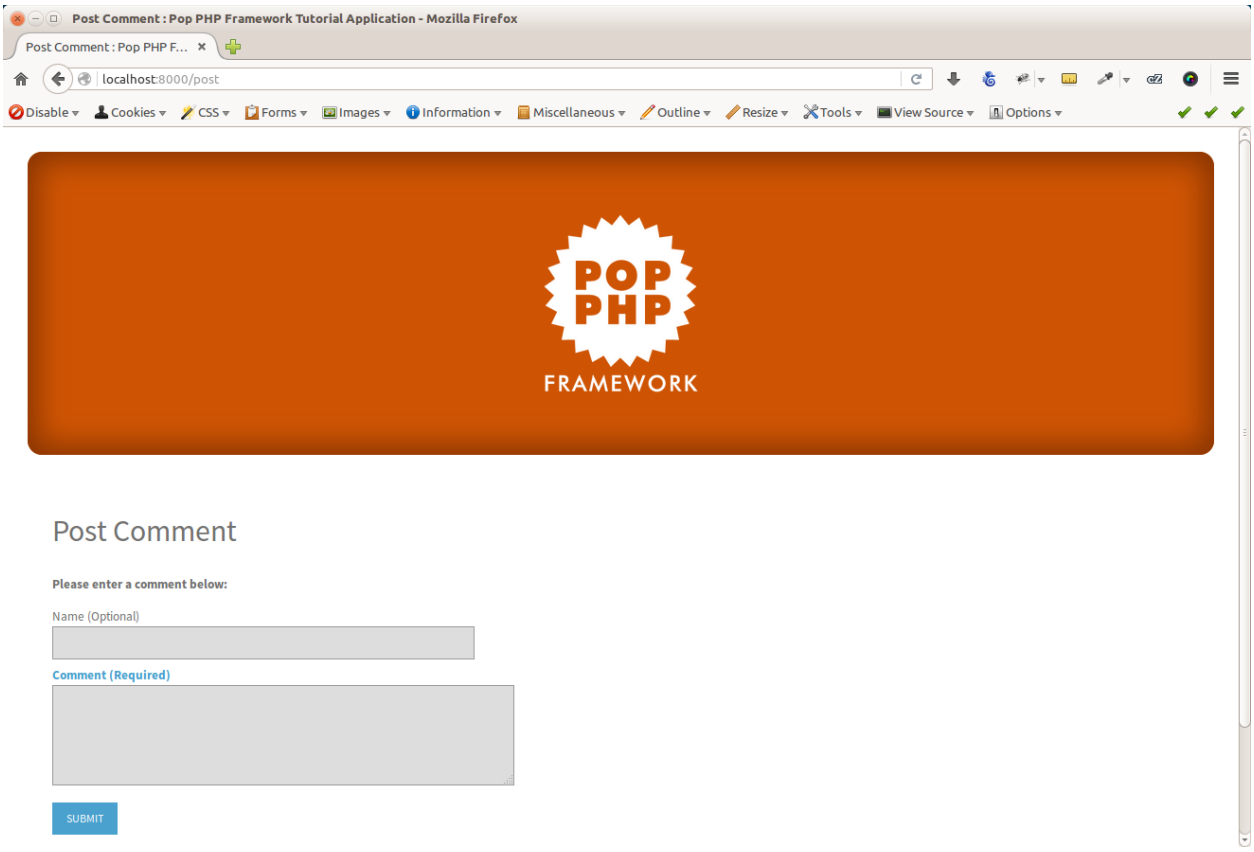
There is a specific controller for each of the two environments, web and console. There is a single form for collecting, validating and submitting a post comment. That form class is only used in the web environment as it is a web form. And the model and table classes are used in both environments, as they serve as the gateway between the application and its data.

Working with the Web App

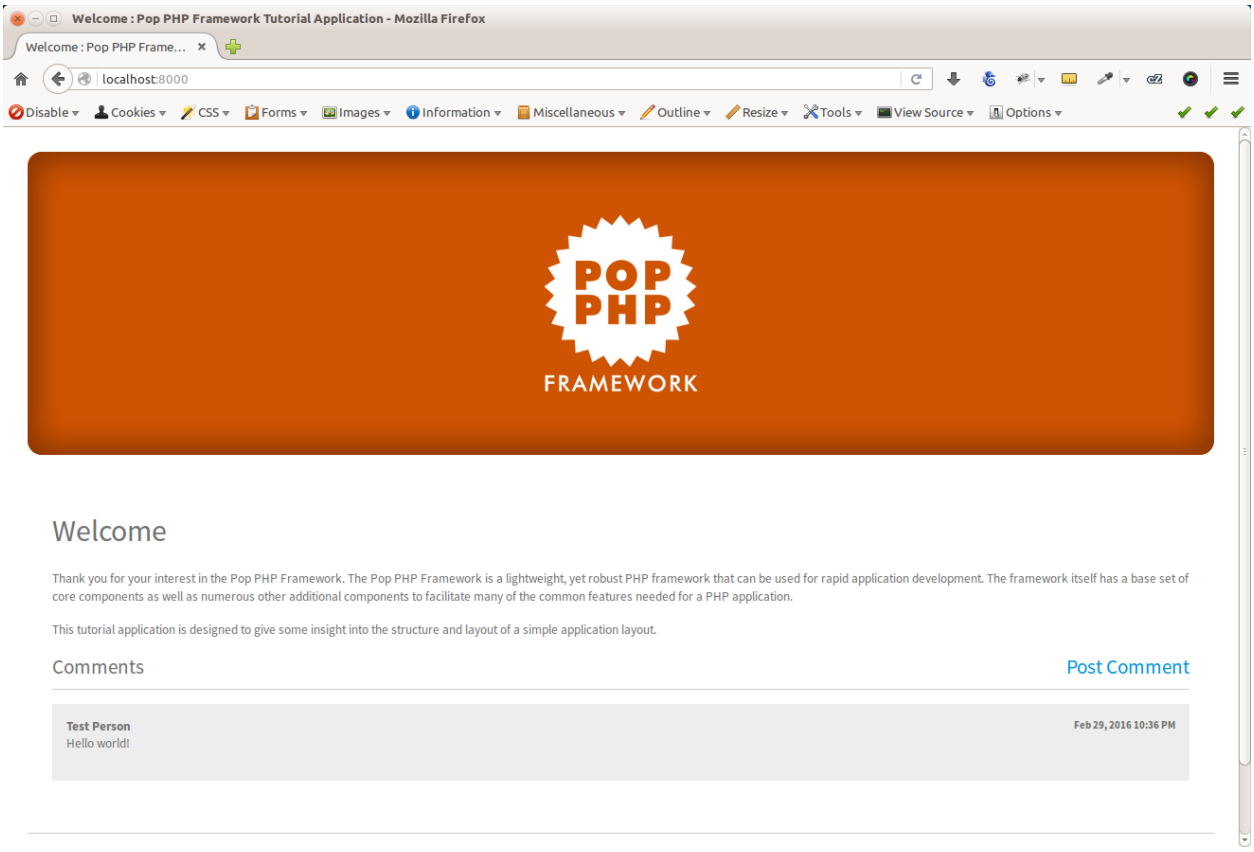
Once you have the web application up and running, you should be able to see the Pop welcome screen in your browser:



From the user perspective, you can click on the *post* link to view the post comment form:



Once you successfully fill out the post comment form, you are redirected back to the home page, where you can see the new post is now displayed in the feed of posts at the bottom of the page:



The Index File

A closer look at the application code in the main `public/index.php` file and you'll see:

```
<?php

// Require autoloader
$autoloader = include __DIR__ . '/../vendor/autoload.php';

// Create main app object, register the app module and run the app
$app = new Pop\Application($autoloader, include __DIR__ . '/../app/config/app.web.php
↪');
$app->register(new Tutorial\Module());
$app->run();
```

In the above file, the autoloader is loaded up, and the new application object is created. Then, the module object is created, passing the web application configuration file into the module object. From there, the `run()` method is called and the web application is routed and on its way.

If you take a look at the `app/config/app.web.php` file, you'll see the web routes, as well as the database service, are defined in the file. The routes are automatically passed and wired up to a router object and the main application sets the database object that is to be used from the service.

IndexController

Looking at the main `IndexController` class in the `app/src/Controller/` folder, you will see the various method actions that serve as the route end points. Within the constructor of the controller, a few object properties are wired up that will be needed, such at the request and response objects and the view path. Within the `index` method, you can see a basic call the model, and the setting of data in the view to be displayed in the browser. The `post` method handles a more complex transaction, testing for POST, and validating the form and passing the form data to the model object upon a successful validation.

Working with the Console App

The CLI side of the application is set up to demonstrate how one would program and interact with an application via the console. From a user perspective, running the following commands will trigger certain actions.

To show the help screen:

```
script/pop help
```

```
nick@wolverine:~/Projects/pop/popphp-tutorial/script$ ./pop help

    Pop Tutorial CLI
    ----------------

    ./pop show          Show current posts
    ./pop delete        Delete a post
    ./pop help          This help screen

    ----------------
    Complete!
```

To show the current posts:

```
script/pop show
```

```
nick@wolverine:~/Projects/pop/popphp-tutorial/script$ ./pop show

    Pop Tutorial CLI
    ----------------

    1. Test Person [Feb 29, 2016 10:36 PM]

    ----------------
    Complete!
```

To delete a post:

```
script/pop delete
```

```
nick@wolverine:~/Projects/pop/popphp-tutorial/script$ ./pop delete

    Pop Tutorial CLI
    ----------------

    1:  Test Person

    Select Post ID: 1

    Post Removed!

    ----------------
    Complete!
```

Script File

A closer look at the application code in the main `script/pop` file and you'll see:

```
#!/usr/bin/php
<?php

// Require autoloader
$autoloader = include __DIR__ . '/../vendor/autoload.php';

// Create main app object, register the app module and run the app
$app = new Pop\Application($autoloader, include __DIR__ . '/../app/config/app.cli.php
↪');
$app->register(new Tutorial\Module());
$app->run();
```

In the above file, the shell environment is set to PHP. Like the web index file, this script file loads the autoloader, and the new application object is created, passing the console application configuration file into the application object. From there, the `run()` method is called and the console application is routed and on its way.

If you take a look at the `app/config/app.cli.php` file, you'll see the console routes, as well as the database service, are defined in the file. The routes are automatically passed and wired up to a router object and the main application object sets the database object that is to be used from the service.

ConsoleController

Looking at the main `ConsoleController` class in the `app/src/Controller/` folder, you will see the various method actions that serve as the route end points. Within the constructor of the controller, a few object properties are wired up that will be needed, such at the console object and the help command. Within the `help` method, you can see a basic call to display the help text in the console. The `show` method accesses the model to retrieve the data to display the posts in the console. The `delete` method handles a more complex transaction, triggering a prompt for the user to enter the selected post ID to delete.

CHAPTER 6

Reference

pop-acl

The *popphp/pop-acl* component is an authorization and access control component the serves as a hybrid between standard ACL and RBAC user access concepts. Beyond allowing or denying basic user access, it provides support for roles, resources, inherited permissions and also assertions for fine-grain access control.

It is not to be confused with the authentication component, as that deals with whether or not a user is whom they claim they are (identity) and not about the resources to which they may or may not have access.

Installation

Install it directly into your project:

```
composer require popphp/pop-acl
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-acl": "3.0.*",
    }
}
```

Basic Use

With an ACL object, you can create user roles, resources and set the permissions for which user has access to which resource, and to what degree.

```
use Pop\Acl\Acl;
use Pop\Acl\AclRole as Role;
use Pop\Acl\AclResource as Resource;
```

```
$acl = new Acl();

$admin  = new Role('admin');
$editor = new Role('editor');
$reader = new Role('reader');

$page = new Resource('page');

$acl->addRoles([$admin, $editor, $reader]);
$acl->addResource($page);

$acl->allow('admin', 'page')           // Admin can do anything to a page
    ->allow('editor', 'page', 'edit')  // Editor can only edit a page
    ->allow('reader', 'page', 'read'); // Editor can only edit a page

if ($acl->isAllowed('admin', 'page', 'add'))    { } // Returns true
if ($acl->isAllowed('editor', 'page', 'edit')) { } // Returns true
if ($acl->isAllowed('editor', 'page', 'add'))  { } // Returns false
if ($acl->isAllowed('reader', 'page', 'edit')) { } // Returns false
if ($acl->isAllowed('reader', 'page', 'read')) { } // Returns true
```

You can fine-tune the permissions as well, setting which user is denied or allowed.

```
$acl->allow('admin', 'page')         // Admin can do anything to a page
    ->allow('editor', 'page')        // Editor can do anything to a page
    ->deny('editor', 'page', 'add'); // except add a page
```

Role Inheritance

You can have roles inherit access rules as well.

```
use Pop\Acl\Acl;
use Pop\Acl\AclRole as Role;
use Pop\Acl\AclResource as Resource;

$acl = new Acl();

$editor = new Role('editor');
$reader = new Role('reader');

// Add the $reader role as a child role of $editor.
// The role $reader will now inherit the access rules
// of the role $editor, unless explicitly overridden.
$editor->addChild($reader);

$page = new Resource('page');

$acl->addRoles([$editor, $reader]);
$acl->addResource($page);

// Neither the editor or reader can add a page
$acl->deny('editor', 'page', 'add');

// The editor can edit a page
$acl->allow('editor', 'page', 'edit');
```

```
// Both the editor or reader can read a page
$acl->allow('editor', 'page', 'read');

// Over-riding deny rule so that a reader cannot edit a page
$acl->deny('reader', 'page', 'edit');

if ($acl->isAllowed('editor', 'page', 'add'))  { } // Returns false
if ($acl->isAllowed('reader', 'page', 'add'))  { } // Returns false
if ($acl->isAllowed('editor', 'page', 'edit')) { } // Returns true
if ($acl->isAllowed('reader', 'page', 'edit')) { } // Returns false
if ($acl->isAllowed('editor', 'page', 'read')) { } // Returns true
if ($acl->isAllowed('reader', 'page', 'read')) { } // Returns true
```

Assertions

If you want even more of a fine-grain control over permissions and who is allowed to do what, you can use assertions. First, define the assertion class, which implements the AssertionInterface. In this example, we want to check that the user "owns" the resource via a matching user ID.

```
use Pop\Acl\Acl;
use Pop\Acl\AclRole;
use Pop\Acl\AclResource;
use Pop\Acl\Assertion\AssertionInterface;

class UserCanEditPage implements AssertionInterface
{

    public function assert(
        Acl $acl, AclRole $role,
        AclResource $resource = null,
        $permission = null
    )
    {
        return ((null !== $resource) && ($role->id == $resource->user_id));
    }

}
```

Then, within the application, you can use the assertions like this:

```
use Pop\Acl\Acl;
use Pop\Acl\AclRole as Role;
use Pop\Acl\AclResource as Resource;

$acl = new Acl();

$admin  = new Role('admin');
$editor = new Role('editor');

$page = new Resource('page');

$admin->id     = 1001;
$editor->id    = 1002;
$page->user_id = 1001;
```

```
$acl->addRoles([$admin, $editor]);
$acl->addResource($page);

$acl->allow('admin', 'page', 'add')
    ->allow('admin', 'page', 'edit', new UserCanEditPage())
    ->allow('editor', 'page', 'edit', new UserCanEditPage())

// Returns true because the assertion passes,
// the admin's ID matches the page's user ID
if ($acl->isAllowed('admin', 'page', 'edit')) { }

// Although editors can edit pages, this returns false
// because the assertion fails, as this editor's ID
// does not match the page's user ID
if ($acl->isAllowed('editor', 'page', 'edit')) { }
```

pop-application

The `Pop\Application` class is the main application class of the Pop PHP Framework. It comes with the core *popphp/popphp* component and serves as the main application container within an application written using Pop. With it, you can wire your application's necessary configuration and manage all of the aspects of your application.

Installation

Install it directly into your project:

```
composer require popphp/popphp
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/popphp": "3.0.*",
    }
}
```

Basic Use

The application object of the Pop PHP Framework is the main object that helps control and provide access to the application's elements, configuration and current state. Within the application object are ways to create, store and manipulate common elements that you may need during an application's life-cycle, such as the router, service locator, event manager and module manager. Additionally, you can also have access to the config object and the autoloader, if needed.

The application object's constructor is flexible in what it can accept when setting up your application. You can pass it individual instances of the objects your application will need:

```
$autoloader = include __DIR__ . '/vendor/autoload.php';
$router     = new Pop\Router\Router();
$service    = new Pop\Service\Locator();
$events     = new Pop\Event\Manager();
```

```
$app = new Pop\Application(
    $autoloader, $router, $service, $events
);
```

In the above example, the autoloader, a router, a service locator and an event manager all get passed into the application so that can be utilized at any given point with the application's life cycle. Additionally, you can pass it a configuration array and let the application object create and set up the objects for you:

```
$config = [
    'routes' => [
        '/' => [
            'controller' => 'MyApp\Controller\IndexController',
            'action'     => 'index'
        ]
    ],
    'events' => [
        [
            'name'     => 'app.init',
            'action'   => 'MyApp\Event::doSomething',
            'priority' => 1000
        ]
    ],
    'services' => [
        'session' => 'Pop\Web\Session::getInstance'
    ]
];

$app = new Pop\Application($config);
```

Once the application object and its dependencies are wired up, you'll be able to interact with the application object through the appropriate API calls.

- `$app->bootstrap($autoloader = null)` - Bootstrap the application
- `$app->init()` - Initialize the application
- `$app->loadConfig($config)` - Load a new configuration
- `$app->loadRouter($router)` - Load a new router object
- `$app->loadServices($services)` - Load a new service locator
- `$app->loadEvents($events)` - Load a new event manager
- `$app->loadModules($modules)` - Load a new module manager
- `$app->registerAutoloader($autoloader)` - Register an autoloader with the application
- `$app->mergeConfig($config, $replace = false)` - Merge config values into the application
- `$app->run()` - Run the application

You can access the main elements of the application object through the following methods:

- `$app->autoloader()` - Access the autoloader
- `$app->config()` - Access the configuration object
- `$app->router()` - Access the router
- `$app->services()` - Access the service locator
- `$app->events()` - Access the event manager

- `$app->modules()` - Access the module manager

Also, magic methods expose them as direct properties as well:

- `$app->autoloader` - Access the autoloader
- `$app->config` - Access the configuration object
- `$app->router` - Access the router
- `$app->services` - Access the service locator
- `$app->events` - Access the event manager
- `$app->modules` - Access the module manager

The application object has some shorthand methods to help tidy up common calls to elements within the application object:

- `$app->register($name, $module);` - Register a module
- `$app->unregister($name);` - Unregister a module
- `$app->isRegistered($name);` - Check if a module is registered
- `$app->module($module)` - Get a module object
- `$app->addRoute($route, $controller);` - Add a route
- `$app->addRoutes($routes);` - Add routes
- `$app->setService($name, $service);` - Set a service
- `$app->getService($name);` - Get a service
- `$app->removeService($name);` - Remove a service
- `$app->on($name, $action, $priority = 0);` - Attach an event
- `$app->off($name, $action);` - Detach an event
- `$app->trigger($name, array $args = []);` - Trigger an event

Of course, once you've configured your application object, you can run the application by simply executing the `run` method:

```
$app->run();
```

pop-auth

The *popphp/pop-auth* component is an authentication component that provides different adapters to authenticate a user's identity. It is not to be confused with the ACL component, as that deals with user roles and access to certain resources and not authenticating user identity.

Installation

Install it directly into your project:

```
composer require popphp/pop-auth
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-auth": "3.0.*",
    }
}
```

Basic Use

You can authenticate using a file, a database, over HTTP or over LDAP.

File

For this example, we use a file called `.htmyauth` containing a colon-delimited list of usernames and encrypted passwords:

```
admin:$...some hash...
editor:$...some hash...
reader:$...some hash...
```

```
use Pop\Auth;

$auth = new Auth\File('/path/to/.htmyauth');
$auth->authenticate('admin', '12admin34');

if ($auth->isValid()) { } // Returns true
```

Database

For this example, there is a table in a database called 'users' and a correlating table class called `MyApp\\Users` that extends `Pop\\Db\\Record`.

For simplicity, the table has a column called *username* and a column called *password*. By default, the table adapter will look for a *username* column and a *password* column unless otherwise specified.

```
use Pop\Auth;

$auth = new Auth\Table('MyApp\Users');

// Attempt #1
$auth->authenticate('admin', 'bad-password');

// Returns false because the value of the hashed attempted
// password does not match the hash in the database
if ($auth->isValid()) { }

// Attempt #2
$auth->authenticate('admin', '12admin34');

// Returns true because the value of the hashed attempted
// password matches the hash in the database
if ($auth->isValid()) { }
```

HTTP

In this example, the user can simply authenticate using a remote server over HTTP. Based on the headers received from the initial request, the Http adapter will auto-detect most things, like the the auth type (Basic or Digest), content encoding, etc.

```
use Pop\Auth;

$auth = new Auth\Http('https://www.domain.com/auth', 'post');
$auth->authenticate('admin', '12admin34');

if ($auth->isValid()) { } // Returns true
```

LDAP

Again, in this example, the user can simply authenticate using a remote server, but this time, using LDAP. The user can set the port and other various options that may be necessary to communicate with the LDAP server.

```
use Pop\Auth;

$auth = new Auth\Ldap('ldap.domain', 389, [LDAP_OPT_PROTOCOL_VERSION => 3]);
$auth->authenticate('admin', '12admin34');

if ($auth->isValid()) { } // Returns true
```

pop-cache

The *popphp/pop-cache* component is a caching component that provides different adapters to cache data and have it persist for a certain length of time.

Installation

Install it directly into your project:

```
composer require popphp/pop-cache
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-cache": "3.2.*",
    }
}
```

Basic Use

Each adapter object can be created and passed configuration parameters specific to that adapter:

APC

```
use Pop\Cache\Adapter;

// Create an APC cache adapter object, with a 5 minute lifetime
$apcCache = new Adapter\Apc(300);
```

File

```
use Pop\Cache\Adapter;

// Create a file cache adapter object, with a 5 minute lifetime
$cacheAdapter = new Adapter\File('/path/to/my/cache/dir', 300);
```

Memcache

```
use Pop\Cache\Adapter;

// Create a Memcache cache adapter object, with a 5 minute lifetime
$cacheAdapter = new Adapter\Memcache(300);
```

Memcached

```
use Pop\Cache\Adapter;

// Create a Memcached cache adapter object, with a 5 minute lifetime
$cacheAdapter = new Adapter\Memcached(300);
```

Redis

```
use Pop\Cache\Adapter;

// Create a Redis cache adapter object, with a 5 minute lifetime
$cacheAdapter = new Adapter\Redis(300);
```

Session

```
use Pop\Cache\Adapter;

// Create a session cache adapter object, with a 5 minute lifetime
$cacheAdapter = new Adapter\Session(300);
```

SQLite

```
use Pop\Cache\Adapter;

// Create a database cache adapter object, with a 5 minute lifetime
$cacheAdapter = new Adapter\Sqlite('/path/to/my/.htcachedb.sqlite', 300);
```

You can then pass any of the above cache adapter objects into the main cache object to begin storing and recalling data.

```
use Pop\Cache\Cache;

$cache = new Cache($cacheAdapter);

// Save some data to the cache
$cache->saveItem('foo', $myData);

// Recall that data later in the app.
// Returns false is the data does not exist or has expired.
$foo = $cache->getItem('foo');
```

To remove data from cache, you call the `deleteItem` method:

```
$cache->deleteItem('foo');
```

And to clear all data from cache, you call the `clear` method:

```
$cache->clear();
```

pop-code

The *popphp/pop-code* component is a code generation and reflection component that provides an API to generate and save PHP code, as well as utilize reflection to parse existing code and modify and build on it.

Installation

Install it directly into your project:

```
composer require popphp/pop-code
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-code": "3.0.*",
    }
}
```

Basic Use

Here's an example to create a class with a property and a method:

```
use Pop\Code\Generator;

// Create the class object and give it a namespace
$class = new Generator('MyClass.php', Generator::CREATE_CLASS);
$class->setNamespace(new Generator\NamespaceGenerator('MyApp'));

// Create a new protected property with a default value
$prop = new Generator\PropertyGenerator('foo', 'string', 'bar', 'protected');

// Create a method and give it an argument, body and docblock description
$method = new Generator\MethodGenerator('setFoo', 'public');
$method->addArgument('foo')
       ->setBody('$this->foo = $foo;')
       ->setDesc('This is the method to set foo.');

// Add the property and the method to the class code object
$class->code()->addProperty($prop);
$class->code()->addMethod($method);

// Save the class file
$class->save();

// Or, you can echo out the contents of the code directly
echo $class;
```

The newly created class will look like:

```
<?php
/**
 * @namespace
 */
namespace MyApp;

class MyClass
{

    /**
     * @var   string
     */
    protected $foo = 'bar';


    /**
     * This is the method to set foo.
     */
    public function setFoo($foo)
    {
        $this->foo = $foo;

    }

}
```

And here's an example using the existing code from above and adding a method to it. The reflection object provides you with a code generator object like the one above so that you can add or remove things from the parsed code:

```
use Pop\Code\Generator;
use Pop\Code\Reflection;
```

```php
$class = new Reflection('MyApp\MyClass');

// Create the new method that you want to add to the existing class
$method = new Generator\MethodGenerator('hasFoo', 'public');
$method->addArgument('foo')
       ->setBody('return (null !== $this->foo);')
       ->setDesc('This is the method to see if foo is set.');

// Access the generator and it's code object to add the new method to it
$reflect->generator()->code()->addMethod($method);

// Echo out the code
echo $reflect->generator();
```

The modified class, with the new method, will look like:

```php
<?php
/**
 * @namespace
 */
namespace MyApp;

class MyClass implements
{

    /**
     * @var   string
     */
    protected $foo = 'bar';

    /**
     * This is the method to set foo.
     */
    public function setFoo($foo)
    {
        $this->foo = $foo;
    }

    /**
     * This is the method to see if foo is set.
     */
    public function hasFoo($foo)
    {
        return (null !== $this->foo);

    }

}
```

pop-config

The *popphp/pop-config* component is a configuration component that allows you to store configuration data for the life cycle of your application. It also has the ability to parse existing and common configuration formats, such as INI, JSON and XML files.

Installation

Install it directly into your project:

```
composer require popphp/pop-config
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-config": "3.0.*",
    }
}
```

Basic Use

The values of a config object can be access either via object arrow notation or as an array:

```
$config = new Pop\Config\Config(['foo' => 'bar']);

$foo = $config->foo;
// OR
$foo = $config['foo'];
```

By default, the config object is set to not direct allow changes to its values, unless the `$allowChanges` property is set to `true`. The following example isn't possible unless the `$allowChanges` property is set to `true`.

```
$config = new Pop\Config\Config(['foo' => 'bar'], true);
$config->foo    = 'baz';
// OR
$config['foo'] = 'new';
```

However, if the `$allowChanges` property is set to false, you can append new values to the config object with the `merge()` method.

```
$config = new Pop\Config\Config($configData);
$config->merge($newData);
```

And, if you need to convert the configuration object down to a simple array, you can do so:

```
$config = new Pop\Config\Config($configData);
$data   = $config->toArray();
```

Parsing a Config File

Let's look at the following example `ini` configuration file:

```
; This is a sample configuration file config.ini
[foo]
bar = 1
baz = 2
```

You would just pass that file into the constructor on object instantiation, and then access the configuration values like so:

```
$config = new Pop\Config\Config('/path/to/config.ini');

$bar = $config->foo->bar; // equals 1
$baz = $config->foo->baz; // equals 2
```

pop-console

The *popphp/pop-console* component is a component for integration and managing a console user interface with your application. It supports the various aspects of the CLI user experience, colorization and prompt input.

Installation

Install it directly into your project:

```
composer require popphp/pop-console
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-console": "3.0.*",
    }
}
```

Basic Use

In this simple example, we create a script called *pop* and wire it up. First, we'll create some commands and an option and add them to the console object:

```
use Pop\Console\Console;
use Pop\Console\Command;

$edit = new Command('edit', Input\Command::VALUE_REQUIRED);
$edit->setHelp('This is the help screen for the edit command.');

$console = new Console();
$console->addCommand($help);
$console->addCommand($edit);
```

Once the commands are registered with the main *$console* object, we access them like so:

```
$console->write($console->help('edit'), '    ');
$console->send();
```

Using a Prompt

You can also trigger a prompt to get information from the user. You can enforce a certain set of options as well as whether or not they are case-sensitive:

```
$console = new Pop\Console\Console();
$letter  = $console->prompt(
    'Which is your favorite letter: A, B, C, or D? ',
    ['A', 'B', 'C', 'D'],
    true
);
echo 'Your favorite letter is ' . $letter . '.';
```

```
./pop
Which is your favorite letter: A, B, C, or D? B   // <- User types 'B'
Your favorite letter is B.
```

pop-controller

The `Pop\Controller` sub-component is part of the core *popphp/popphp* component. It serves as the blueprint controller class on which you can build your application's controller classes.

Installation

Install it directly into your project:

```
composer require popphp/popphp
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/popphp": "3.0.*",
    }
}
```

Basic Use

The main controller class that is provided is actually an abstract class on which you can build the controller classes for your application. In it, is the main `dispatch()` method, as well as methods to set and get the default action. The default action is set to `error` and that would be the method the controller class expect to find and default to if no other method satisfies the incoming route. You can change that to whatever method name you prefer with the `setDefaultAction()` method.

Take a look at an example controller class:

```
namespace MyApp\Controller;

use Pop\Controller\AbstractController;

class IndexController extends AbstractController
{

    public function index()
    {
        // Do something for the index page
    }
```

```
    public function users()
    {
        // Do something for the users page
    }

    public function edit($id)
    {
        // Edit user with $id
    }

    public function error()
    {
        // Handle a non-match route request
    }

}
```

As each incoming route's action is matched to a method, it will execute the corresponding method in the controller object. If no route match is found, then it will default to the default action, which in this case, is the `error` method. So depending on your type of application and how it is configured, an example of a successful route could be:

```
http://localhost/users/edit/1001
```

which would route to and execute:

```
MyApp\Controller\UsersController->edit($id)
```

pop-cookie

The *popphp/pop-cookie* component provides the basic functionality to manage cookies.

Installation

Install it directly into your project:

```
composer require popphp/pop-cookie
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-cookie": "3.1.*",
    }
}
```

Basic Use

The cookie component allows you to interact with and manage cookies within the user's session. When you create a new instance of a cookie object, you can pass it some optional parameters for more control:

```
$cookie = Pop\Web\Cookie::getInstance([
    'expire'   => 300,
    'path'     => '/system',
    'domain'   => 'www.domain.com',
    'secure'   => true,
    'httponly' => true
]);
```

These are all options that give you further control over when a cookie value expires and where and how it is available to the user. From there, you can store and retrieve cookie values like this:

```
$cookie->foo   = 'bar';
$cookie['baz'] = 123;

echo $cookie->foo;   // echos 'bar'
echo $cookie['baz']; // echos 123
```

And then you can delete a cookie value like this:

```
$cookie->delete('foo');
unset($cookie['baz']);
```

pop-csv

The *popphp/pop-csv* component is a component for managing CSV data and files.

Installation

Install it directly into your project:

```
composer require popphp/pop-csv
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-csv": "3.0.*",
    }
}
```

Basic Use

The *popphp/pop-csv* component provides a streamlined way to work with PHP data and the CSV format.

Serialize Data

To serialize the data into one of the data types, you can create a data object and call the `serialize()` method:

```
$phpData = [
    [
        'first_name' => 'Bob',
```

```
        'last_name'  => 'Smith'
    ],
    [
        'first_name' => 'Jane',
        'last_name'  => 'Smith'
    ]
];

$data = new Pop\Data\Data($phpData);

$csvString   = $data->serialize();
```

The `$csvString` variable now contains:

```
first_name,last_name
Bob,Smith
Jane,Smith
```

Unserialize Data

You can either pass the data object a direct string of serialized data or a file containing a string of serialized data. It will detect which one it is and parse it accordingly.

```
$csv = new Pop\Data\Data($csvString);
// OR
$csv = new Pop\Data\Data('/path/to/file.csv');

$phpData = $csv->unserialize();
```

Write to File

```
$data = new Pop\Data\Data($phpData);
$data->serialize();
$data->writeToFile('/path/to/file.csv');
```

Output to HTTP

```
$data = new Pop\Data\Data($phpData);
$data->serialize();
$data->outputToHttp();
```

If you want to force a download, you can set that parameter:

```
$data->outputToHttp('my-file.csv', true);
```

pop-db

The *popphp/pop-db* component is a database component for interfacing with databases. By default, it provides adapters for MySQL, PDO, PostgreSQL, SQLServer and SQLite. Other adapters can be built by extending the core abstract adapter. The component provides a SQL builder to assist with writing portable standard SQL queries that can be used

across the different database platforms. And, it also provides a record class that services as an active record/table gateway hybrid. The record sub-component provides easy set up of database tables, along with an easy API to access the database tables and the data in them.

Installation

Install it directly into your project:

```
composer require popphp/pop-db
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-db": "4.0.*",
    }
}
```

Basic Use

You can use the database factory to create the appropriate adapter instance and connect to a database:

```
$mysql = Pop\Db\Db::connect('mysql', [
    'database' => 'my_database',
    'username' => 'my_db_user',
    'password' => 'my_db_password',
    'host'     => 'mydb.server.com'
]);
```

And for other database connections:

```
$pgsql  = Pop\Db\Db::connect('pgsql', $options);
$sqlsrv = Pop\Db\Db::connect('sqlsrv', $options);
$sqlite = Pop\Db\Db::connect('sqlite', $options);
```

If you'd like to use the PDO adapter, it requires the *type* option to be defined so it can set up the proper DSN:

```
$pdo = Pop\Db\Db::connect('pdo', [
    'database' => 'my_database',
    'username' => 'my_db_user',
    'password' => 'my_db_password',
    'host'     => 'mydb.server.com',
    'type'     => 'mysql'
]);
```

And there are shorthand methods as well:

```
$mysql  = Pop\Db\Db::mysqlConnect($options);
$pgsql  = Pop\Db\Db::pgsqlConnect($options);
$sqlsrv = Pop\Db\Db::sqlsrvConnect($options);
$sqlite = Pop\Db\Db::sqliteConnect($options);
$pdo    = Pop\Db\Db::pdoConnect($options);
```

The database factory outlined above is simply creating new instances of the database adapter objects. The code below would produce the same results:

```
$mysql  = new Pop\Db\Adapter\Mysql($options);
$pgsql  = new Pop\Db\Adapter\Pgsql($options);
$sqlsrv = new Pop\Db\Adapter\Sqlsrv($options);
$sqlite = new Pop\Db\Adapter\Sqlite($options);
$pdo    = new Pop\Db\Adapter\Pdo($options);
```

The above adapter objects are all instances of `Pop\\Db\\Adapter\\AbstractAdapter`, which implements the `Pop\\Db\\Adapter\\AdapterInterface` interface. If necessary, you can use that underlying foundation to build your own database adapter to facilitate your database needs for your application.

Querying a Database

Once you've created a database adapter object, you can then use the API to interact with and query the database. Let's assume the database has a table *users* in it with the column *username* in the table.

```
$db = Pop\Db\Db::connect('mysql', $options);

$db->query('SELECT * FROM `users`');

while ($row = $db->fetch()) {
    echo $row['username'];
}
```

Using Prepared Statements

You can also query the database using prepared statements as well. Let's assume the *users* table from above also has and *id* column.

```
$db = Pop\Db\Db::connect('mysql', $options);

$db->prepare('SELECT * FROM `users` WHERE `id` > ?');
$db->bindParams(['id' => 1000]);
$db->execute();

$rows = $db->fetchResult();

foreach ($rows as $row) {
    echo $row['username'];
}
```

The Query Builder

The query builder is a part of the component that provides an interface that will produce syntactically correct SQL for whichever type of database you have elected to use. One of the main goals of this is portability across different systems and environments. In order for it to function correctly, you need to pass it the database adapter your application is currently using so that it can properly build the SQL.

```
$db = Pop\Db\Db::connect('mysql', $options);

$sql = $db->createSql();
$sql->select(['id', 'username'])
    ->from('users')
    ->where('id > :id');
```

```
echo $sql;
```

The above example will produce:

```
SELECT `id`, `username` FROM `users` WHERE `id` > ?
```

If the database adapter changed to PostgreSQL, then the output would be:

```
SELECT "id", "username" FROM "users" WHERE "id" > $1
```

And SQLite would look like:

```
SELECT "id", "username" FROM "users" WHERE "id" > :id
```

The SQL Builder component has an extensive API to assist you in constructing complex SQL statements. Here's an example using JOIN and ORDER BY:

```
$db = Pop\Db\Db::connect('mysql', $options);

$sql = $db->createSql();
$sql->select([
    'user_id'    => 'id',
    'user_email' => 'email'
])->from('users')
  ->leftJoin('user_data', ['users.id' => 'user_data.user_id'])
  ->orderBy('id', 'ASC');
  ->where('id > :id');

echo $sql;
```

The above example would produce the following SQL statement for MySQL:

```
SELECT `id` AS `user_id`, `email` AS `user_email` FROM `users`
    LEFT JOIN `user_data` ON `users`.`id` = `user_data`.`user_id`
    WHERE `id` > ?
    ORDER BY `id` ASC;
```

The Schema Builder

In addition to the query builder, there is also a schema builder to assist with database table structures and their management. In a similar fashion to the query builder, the schema builder has an API that mirrors the SQL that would be used to create, alter and drop tables in a database.

```
$db = Pop\Db\Db::connect('mysql', $options);

$schema = $db->createSchema();
$schema->create('users')
    ->int('id', 16)
    ->varchar('username', 255)
    ->varchar('password', 255);

echo $schema;
```

The above code would produced the following SQL:

```
CREATE TABLE `users` (
  `id` INT(16),
  `username` VARCHAR(255),
  `password` VARCHAR(255)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Active Record

The `Pop\Db\Record` class uses the Active Record pattern as a base to allow you to work with and query tables in a database directly. To set this up, you create a table class that extends the `Pop\Db\Record` class:

```
class Users extends Pop\Db\Record { }
```

By default, the table name will be parsed from the class name and it will have a primary key called *id*. Those settings are configurable as well for when you need to override them. The "class-name-to-table-name" parsing works by converting the CamelCase class name into a lower case underscore name (without the namespace prefix):

- Users -> users
- MyUsers -> my_users
- MyApp\Table\SomeMetaData -> some_meta_data

If you need to override these default settings, you can do so in the child table class you create:

```
class Users extends Pop\Db\Record
{
    protected $table  = 'my_custom_users_table';

    protected $prefix = 'pop_';

    protected $primaryKeys = ['id', 'some_other_id'];
}
```

In the above example, the table is set to a custom value, a table prefix is defined and the primary keys are set to a value of two columns. The custom table prefix means that the full table name that will be used in the class will be *pop_my_custom_users_table*.

Once you've created and configured your table classes, you can then use the API to interface with them. At some point in the beginning stages of your application's life cycle, you will need to set the database adapter for the table classes to use. You can do that like this:

```
$db = Pop\Db\Db::connect('mysql', $options);
Pop\Db\Record::setDb($db);
```

That database adapter will be used for all table classes in your application that extend `Pop\Db\Record`. If you want a specific database adapter for a particular table class, you can specify that on the table class level:

```
$userDb = Pop\Db\Db::connect('mysql', $options)
Users::setDb($userDb);
```

From there, the API to query the table in the database directly like in the following examples:

Fetch a single row by ID, update data

```
$user = Users::findById(1001);
```

```
if (isset($user->id)) {
    $user->username = 'admin2';
    $user->save();
}
```

Fetch a single row by another column

```
$user = Users::findOne(['username' => 'admin2']);

if (isset($user->id)) {
    $user->username = 'admin3';
    $user->save();
}
```

Fetch multiple rows

```
$users = Users::findAll([
    'order' => 'id ASC',
    'limit' => 25
]);

foreach ($users as $user) {
    echo $user->username;
}

$users = Users::findBy(['logins' => 0]);

foreach ($users as $user) {
    echo $user->username . ' has never logged in.';
}
```

Fetch and return only certain columns

```
$users = Users::findAll(['select' => ['id', 'username']]);

foreach ($users as $user) {
    echo $user->id . ': ' . $user->username;
}

$users = Users::findBy(['logins' => 0], ['select' => ['id', 'username']]);

foreach ($users as $user) {
    echo $user->id . ': ' . $user->username . ' has never logged in.';
}
```

Create a new record

```
$user = new Users([
    'username' => 'editor',
    'email'    => 'editor@mysite.com'
]);

$user->save();
```

You can execute custom SQL to run custom queries on the table. One way to do this is by using the SQL Builder:

```
$sql = Users::db()->createSql();
```

```
$sql->select()
    ->from(Users::table())
    ->where('id > :id');

$users = Users::execute($sql, ['id' => 1000]);

foreach ($users as $user) {
    echo $user->username;
}
```

The basic overview of the record class static API is as follows, using the child class `Users` as an example:

- `Users::setDb(Adapter\AbstractAdapter $db, $prefix = null, $isDefault = false)` - Set the DB adapter
- `Users::hasDb()` - Check if the class has a DB adapter set
- `Users::db()` - Get the DB adapter object
- `Users::sql()` - Get the SQL object
- `Users::findById($id)` - Find a single record by ID
- `Users::findOne(array $columns = null, array $options = null)` - Find a single record
- `Users::findBy(array $columns = null, array $options = null, $resultAs = Record::AS_RECORD)` - Find a record or records by certain column values
- `Users::findAll(array $options = null, $resultAs = Record::AS_RECORD)` - Find all records in the table
- `Users::execute($sql, $params, $resultAs = Record::AS_RECORD)` - Execute a custom prepared SQL statement
- `Users::query($sql, $resultAs = Record::AS_RECORD)` - Execute a simple SQL query

In the `findOne`, `findBy` and `findAll` methods, the `$options` parameter is an associative array that can contain values such as:

```
$options = [
    'select' => ['id', 'username'],
    'order'  => 'username ASC',
    'limit'  => 25,
    'offset' => 5
];
```

The *select* key value can be an array of only the columns you would like to select. Otherwise it will select all columns *. The *order*, *limit* and *offset* key values all relate to those values to control the order, limit and offset of the SQL query.

The `$resultAs` parameter allows you to set what the row set is returned as:

- `AS_ARRAY` - As arrays
- `AS_OBJECT` - As array objects
- `AS_RECORD` - As instances of the `Pop\Db\Record`

The benefit of `AS_RECORD` is that you can operate on that row in real time, but if there are many rows returned in the result set, performance could be hindered. Therefore, you can use something like `AS_ARRAY` as an alternative to keep the row data footprint smaller and lightweight.

Accessing records non-statically

If you're interested in an alternative to the active record pattern, there is a non-static API within the `Pop\Db\Record` class:

```
$user = new Users();
$user->getById(5);
echo $user->username;
```

The basic overview of the result class API is as follows:

- `$user->getById($id)` - Find a single record by ID
- `$user->getOneBy(array $columns = null, array $options = null)` - Find a single record by ID
- `$user->getBy(array $columns = null, array $options = null, $resultAs = Record::AS_RECORD)` - Find a record or records by certain column values
- `$user->getAll(array $options = null, $resultAs = Record::AS_RECORD)` - Find all records in the table

Relationships & Associations

Relationships and associations are supported to allow for a simple way to select related data within the database. Building on the example above with the *Users* table, let's add an *Info* and an *Orders* table. The user will have a 1:1 relationship with a row in the *Info* table, and the user will have a 1:many relationship with the *Orders* table:

```
class Users extends Pop\Db\Record
{

    // Define a 1:1 relationship
    public function info()
    {
        return $this->hasOne('Info', 'user_id')
    }

    // Define a 1:many relationship
    public function orders()
    {
        return $this->hasMany('Orders', 'user_id');
    }

}

// Foreign key to the related user is `user_id`
class Info extends Pop\Db\Record
{

}

// Foreign key to the related user is `user_id`
class Orders extends Pop\Db\Record
{

    // Define the parent relationship up to the user that owns this order record
    public function user()
    {
        return $this->belongsTo('User', 'user_id');
```

```
    }

}
```

So with those table classes wired up, there now exists a useful network of relationships among the database entities that can be accessed like this:

```
$user = Users::findById(1);

// Loop through all of the user's orders
foreach ($user->orders as $order) {
    echo $order->id;
}

// Display the user's title stored in the `info` table
echo $user->info->title;
```

Or, in this case, if you have selected an order already and want to access the parent user that owns it:

```
$order = Orders::findById(2);
echo $order->user->username;
```

Eager-Loading

In the 1:many example given above, the orders are "lazy-loaded," meaning that they aren't called from of the database until you call the `orders()` method. However, you can access a 1:many relationship with what is called "eager-loading." However, to take full advantage of this, you would have alter the method in the *Users* table:

```
class Users extends Pop\Db\Record
{

    // Define a 1:many relationship
    public function orders($options = null, $eager = false)
    {
        return $this->hasMany('Orders', 'user_id', $options, $eager);
    }

}
```

The `$options` parameter is a way to pass additional select criteria to the selection of the order rows, such as *order* and *limit*. The `$eager` parameter is what triggers the eager-loading, however, with this set up, you'll actually access it using the static `with()` method, like this:

```
$user = Users::with('orders')->getById(10592005);

// Loop through all of the user's orders
foreach ($user->orders as $order) {
    echo $order->id;
}
```

A note about the access in the example given above. Even though a method was defined to access the different relationships, you can use a magic property to access them as well, and it will route to that method. Also, object and array notation is supported throughout any record object. The following example all produce the same result:

```
$user = Users::findById(1);

echo $user->info()->title;
echo $user->info()['title'];
```

```
echo $user->info->title;
echo $user->info['title'];
```

Shorthand SQL Syntax

To help with making custom queries more quickly and without having to utilize the Sql Builder, there is shorthand SQL syntax that is supported by the `Pop\Db\Record` class. Here's a list of what is supported and what it translates into:

Basic operators

```
$users = Users::findBy(['id' => 1]);   => WHERE id = 1
$users = Users::findBy(['id!=' => 1]); => WHERE id != 1
$users = Users::findBy(['id>' => 1]);  => WHERE id > 1
$users = Users::findBy(['id>=' => 1]); => WHERE id >= 1
$users = Users::findBy(['id<' => 1]);  => WHERE id < 1
$users = Users::findBy(['id<=' => 1]); => WHERE id <= 1
```

LIKE and NOT LIKE

```
$users = Users::findBy(['%username%'   => 'test']); => WHERE username LIKE '%test%'
$users = Users::findBy(['username%'    => 'test']); => WHERE username LIKE 'test%'
$users = Users::findBy(['%username'    => 'test']); => WHERE username LIKE '%test'
$users = Users::findBy(['-%username'   => 'test']); => WHERE username NOT LIKE '%test'
$users = Users::findBy(['username%-'   => 'test']); => WHERE username NOT LIKE 'test%'
$users = Users::findBy(['-%username%-' => 'test']); => WHERE username NOT LIKE '%test%
↪'
```

NULL and NOT NULL

```
$users = Users::findBy(['username' => null]);  => WHERE username IS NULL
$users = Users::findBy(['username-' => null]); => WHERE username IS NOT NULL
```

IN and NOT IN

```
$users = Users::findBy(['id' => [2, 3]]);  => WHERE id IN (2, 3)
$users = Users::findBy(['id-' => [2, 3]]); => WHERE id NOT IN (2, 3)
```

BETWEEN and NOT BETWEEN

```
$users = Users::findBy(['id' => '(1, 5)']);  => WHERE id BETWEEN (1, 5)
$users = Users::findBy(['id-' => '(1, 5)']); => WHERE id NOT BETWEEN (1, 5)
```

Additionally, if you need use multiple conditions for your query, you can and they will be stitched together with AND:

```
$users = Users::findBy([
    'id>'       => 1,
    '%username' => 'user1'
]);
```

which will be translated into:

```
WHERE (id > 1) AND (username LIKE '%test')
```

If you need to use OR instead, you can specify it like this:

```
$users = Users::findBy([
    'id>'       => 1,
    '%username' => 'user1 OR'
]);
```

Notice the ' OR' added as a suffix to the second condition's value. That will apply the OR to that part of the predicate like this:

```
WHERE (id > 1) OR (username LIKE '%test')
```

Database Migrations

Database migrations are scripts that assist in implementing new changes to the database, as well rolling back any changes to a previous state. It works by storing a directory of migration class files and keeping track of the current state, or the last one that was processed. From that, you can write scripts to run the next migration state or rollback to the previous one.

You can create a blank template migration class like this:

```
use Pop\Db\Sql\Migrator;

Migrator::create('MyNewMigration', 'migrations');
```

The code above will create a file that look like `migrations/20170225100742_my_new_migration.php` and it will contain a blank class template:

```
<?php

use Pop\Db\Sql\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{

    public function up()
    {

    }

    public function down()
    {

    }

}
```

From there, you can write your forward migration steps in the `up()` method, or your rollback steps in the `down()` method. Here's an example that creates a table when stepped forward, and drops that table when rolled back:

```
<?php

use Pop\Db\Sql\Migration\AbstractMigration;

class MyNewMigration extends AbstractMigration
{

    public function up()
```

```
    {
        $schema = $this->db->createSchema();
        $schema->create('users')
            ->int('id', 16)->increment()
            ->varchar('username', 255)
            ->varchar('password', 255)
            ->primary('id');

        $this->db->query($schema);
    }

    public function down()
    {
        $schema = $this->db->createSchema();
        $schema->drop('users');
        $this->db->query($schema);
    }

}
```

To step forward, you would call the migrator like this:

```
use Pop\Db\Db;
use Pop\Db\Sql\Migrator;

$db = Pop\Db\Db::connect('mysql', [
    'database' => 'my_database',
    'username' => 'my_db_user',
    'password' => 'my_db_password',
    'host'     => 'mydb.server.com'
]);

$migrator = new Migrator($db, 'migrations');
$migrator->run();
```

The above code would have created the table `users` with the defined columns. To roll back the migration, you would call the migrator like this:

```
use Pop\Db\Db;
use Pop\Db\Sql\Migrator;

$db = Pop\Db\Db::connect('mysql', [
    'database' => 'my_database',
    'username' => 'my_db_user',
    'password' => 'my_db_password',
    'host'     => 'mydb.server.com'
]);

$migrator = new Migrator($db, 'migrations');
$migrator->rollback();
```

And the above code here would have dropped the table `users` from the database.

pop-debug

The *popphp/pop-debug* is a simple debugging component that can be used to hooked into an application to track certain aspects of the application's lifecycle.

Installation

Install it directly into your project:

```
composer require popphp/pop-debug
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-debug": "1.0.*",
    }
}
```

Basic Use

The debugger supports a number of handlers that can record various events during an application's lifecycle. The provided handlers are:

- **ExceptionHandler**
  - Capture exceptions thrown by the application
- **MemoryHandler**
  - Capture memory usage and peak memory usage
- **MessageHandler**
  - Capture messages at various points in the application's lifecycle
- **QueryHandler**
  - Capture database queries and their parameters and information
- **RequestHandler**
  - Capture information about the current request
- **TimeHandler**
  - Trigger a timer to time the current request or a part of the request.

Also, the debugger supports a few storage methods to storage the debug data after the request is complete:

- File
- SQLite Database
- Redis

Setting up the debugger

```
use Pop\Debug;

$debugger = new Debug\Debugger();
$debugger->addHandler(new Debug\Handler\MessageHandler());
$debugger->setStorage(new Debug\Storage\File('log'));

$debugger['message']->addMessage('Hey! Something happened!');

$debugger->save();
```

The above code will save the following output to the *log* folder in a plain text file:

```
1504213206.00000    Hey! Something happened!
```

Setting up multiple handlers

You can configure multiple handlers to capture different points of data within the application:

```
use Pop\Debug;

$debugger = new Debug\Debugger();
$debugger->addHandler(new Debug\Handler\MessageHandler())
    ->addHandler(new Debug\Handler\ExceptionHandler())
    ->addHandler(new Debug\Handler\RequestHandler())
    ->addHandler(new Debug\Handler\MemoryHandler())
    ->addHandler(new Debug\Handler\TimeHandler());
$debugger->setStorage(new Debug\Storage\File('log'));

$debugger['message']->addMessage('Hey! Something happened!');
$debugger['exception']->addException(new \Exception('Whoops!'));
$debugger['memory']->updateMemoryUsage();
$debugger['memory']->updatePeakMemoryUsage();

$debugger->save();
```

In the above example, if the debugger is exposed as a service throughout the application, then you can access it and call those methods above for the individual handlers to capture the things you need to examine.

Storage formats

The storage object allows you to store the debug data in the following formats:

- Plain text
- JSON
- Serialized PHP

```
$debugger = new Debug\Debugger();
$debugger->addHandler(new Debug\Handler\MessageHandler());
$debugger->setStorage(new Debug\Storage\File('log', 'json'));
```

Query handler

The query handler is a special handler that ties into the *pop-db* component and the profiler available with that component. It allows you to capture any database queries and any information associated with them.

You can set up the query handler like this:

```
use Pop\Debug;
use Pop\Db;

$db = Db\Db::mysqlConnect([
    'database' => 'popdb',
    'username' => 'popuser',
    'password' => '12pop34'
]);

$queryHandler = $db->listen('Pop\Debug\Handler\QueryHandler');

$debugger = new Debug\Debugger();
$debugger->addHandler($queryHandler);
$debugger->setStorage(new Debug\Storage\File('log'));

// Run DB queries...

$debugger->save();
```

So with the query handler attached to the database adapter object, any and all queries that are executed will be recorded by the debugger's query handler.

pop-dir

The *popphp/pop-dir* component provides an API to easily manage traversal of directories.

Installation

Install it directly into your project:

```
composer require popphp/pop-dir
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-dir": "3.0.*",
    }
}
```

Basic Use

The directory object provides the ability to perform directory traversals, recursively or non, while setting configuration parameters to tailor the results to how you would like them.

Simple flat directory traversal

```
use Pop\Dir\Dir;

$dir = new Dir('my-dir');

foreach ($dir as $file) {
    echo $file;
}
```

The above example will just echo out the base name of each file and directory in the first level of the directory:

```
some-dir1
some-dir2
file1.txt
file2.txt
file3.txt
```

If you want to have only files in your results, then you can set the *filesOnly* option:

```
use Pop\Dir\Dir;

$dir = new Dir('my-dir', ['filesOnly' => true]);
```

Recursive traveral

In the following example, we'll set it to traverse the directory recursively, get only the files and store the absolute path of the files:

```
use Pop\Dir\Dir;

$dir = new Dir('my-dir', [
    'recursive' => true,
    'filesOnly' => true,
    'absolute'  => true
]);

foreach ($dir->getFiles() as $file) {
    echo $file;
}
```

The result would look like:

```
/path/to/my-dir/file1.txt
/path/to/my-dir/file2.txt
/path/to/my-dir/file3.txt
/path/to/my-dir/some-dir1/file1.txt
/path/to/my-dir/some-dir1/file2.txt
/path/to/my-dir/some-dir2/file1.txt
/path/to/my-dir/some-dir2/file2.txt
```

If you wanted the relative paths instead, you could set the *relative* option:

```
use Pop\Dir\Dir;

$dir = new Dir('my-dir', [
    'recursive' => true,
    'filesOnly' => true,
    'relative'  => true
]);
```

```
foreach ($dir->getFiles() as $file) {
    echo $file;
}
```

In which the result would look like:

```
./file1.txt
./file2.txt
./file3.txt
./some-dir1/file1.txt
./some-dir1/file2.txt
./some-dir2/file1.txt
./some-dir2/file2.txt
```

Emptying a directory

You can empty a directory as well:

```
use Pop\Dir\Dir;

$dir = new Dir('my-dir');
$dir->emptyDir(true);
```

The *true* flag sets it to delete the actual directory as well.

pop-dom

The *popphp/pop-dom* component is a component for managing and building DOM objects. It provides an easy-to-use API to assist in creating XML and HTML documents, while creating and managing the documents' elements and attributes.

Installation

Install it directly into your project:

```
composer require popphp/pop-dom
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-dom": "3.0.*",
    }
}
```

Basic Use

The *popphp/pop-dom* component is for generating and rendering DOM documents and elements. With it, you can easily create document nodes and their children and have control over node content and attributes.

Creating a Simple Node

```php
use Pop\Dom\Child;

$div = new Child('div');
$h1  = new Child('h1', 'This is a header');
$p   = new Child('p');
$p->setNodeValue('This is a paragraph.');

$div->addChildren([$h1, $p]);

echo $div;
```

The above code produces the following HTML:

```
<div>
    <h1>This is a header</h1>
    <p class="paragraph">This is a paragraph.</p>
</div>
```

Build a DOM Document

Putting all of it together, you can build a full DOM document like this:

```php
// Title element
$title = new Child('title', 'This is the title');

// Meta tag
$meta = new Child('meta');
$meta->setAttributes([
    'http-equiv' => 'Content-Type',
    'content'    => 'text/html; charset=utf-8'
]);

// Head element
$head = new Child('head');
$head->addChildren([$title, $meta]);

// Some body elements
$h1 = new Child('h1', 'This is a header');
$p  = new Child('p', 'This is a paragraph.');

$div = new Child('div');
$div->setAttribute('id', 'content');
$div->addChildren([$h1, $p]);

// Body element
$body = new Child('body');
$body->addChild($div);

// Html element
$html = new Child('html');
$html->addChildren([$head, $body]);

// Create and render the DOM document with HTTP headers
$doc = new Document(Document::HTML, $html);
echo $doc;
```

Which produces the following HTML:

```
<!DOCTYPE html>
<html>
    <head>
        <title>This is the title</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    </head>
    <body>
        <div id="content">
            <h1>This is a header</h1>
            <p>This is a paragraph.</p>
        </div>
    </body>
</html>
```

pop-event

The `Pop\Event` sub-component is part of the core *popphp/popphp* component. It serves as the event manager and listener of the event-driven portion of an application written with Pop.

Installation

Install it directly into your project:

```
composer require popphp/popphp
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/popphp": "3.0.*",
    }
}
```

Basic Use

The event manager provides a way to hook specific event listeners and functionality into certain points in an application's life cycle. You can create an event manager object and attach, detach or trigger event listeners. You can pass callable strings or already instantiated instances of objects, although the latter could be potentially less efficient.

```
$events = new Pop\Event\Manager();

$events->on('foo', 'MyApp\Event->bootstrap');
$events->on('bar', 'MyApp\Event::log');

$events->trigger('foo');
```

The valid callable strings for events are as follows:

1. 'SomeClass'
2. 'SomeClass->foo'

3. 'SomeClass::bar'

With events, you can also inject parameters into them as they are called, so that they may have access to any required elements or values of your application. For example, perhaps you need the events to have access to configuration values from your main application object:

```
$events->trigger('foo', ['application' => $application]);
```

In the above example, any event listeners triggered by `foo` will get the application object injected into them so that the event called can utilize that object and retrieve configuration values from it.

To detach an event listener, you call the `off` method:

```
$events->off('foo', 'MyApp\Event->bootstrap');
```

Event Priority

Event listeners attached to the same event handler can be assigned a priority value to determine the order in which they fire. The higher the priority value, the earlier the event listener will fire.

```
$events->on('foo', 'MyApp\Event->bootstrap', 100);
$events->on('foo', 'MyApp\Event::log', 10);
```

In the example above, the `bootstrap` event listener has the higher priority, so therefore it will fire before the `log` event listener.

pop-form

The *popphp/pop-form* component provides a robust API for the creation and management of HTML web forms, their elements and validation.

Installation

Install it directly into your project:

```
composer require popphp/pop-form
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-form": "3.0.*",
    }
}
```

Basic Use

HTML Forms are common to web applications and present a unique set of challenges in building, rendering and validating a form and it's elements. The *popphp/pop-form* component helps to manage those aspects of web forms and streamline the process of utilizing forms in your web application.

Most of the standard HTML5 form elements are supported within the *popphp/pop-form* component. If you require a different element of any kind, you can extend the *Pop\Form\Element\AbstractElement* class to build your own. With each element instance, you can set attributes, values and validation parameters.

The generic input class is:

- *Pop\Form\Element\Input*

The standard available input element classes extend the above class are:

- *Pop\Form\Element\Input\Button*
- *Pop\Form\Element\Input\Checkbox*
- *Pop\Form\Element\Input\Datalist*
- *Pop\Form\Element\Input\Email*
- *Pop\Form\Element\Input\File*
- *Pop\Form\Element\Input\Hidden*
- *Pop\Form\Element\Input\Number*
- *Pop\Form\Element\Input\Password*
- *Pop\Form\Element\Input\Radio*
- *Pop\Form\Element\Input\Range*
- *Pop\Form\Element\Input\Reset*
- *Pop\Form\Element\Input\Submit*
- *Pop\Form\Element\Input\Text*
- *Pop\Form\Element\Input\Url*

Special case input element classes include:

- *Pop\Form\Element\Input\Captcha*
- *Pop\Form\Element\Input\Csrf*

Other available form element classes are:

- *Pop\Form\Element\Button*
- *Pop\Form\Element\Select*
- *Pop\Form\Element\SelectMultiple*
- *Pop\Form\Element\Textarea*

Special form element collection classes include:

- *Pop\Form\Element\CheckboxSet*
- *Pop\Form\Element\RadioSet*

In the case of the CAPTCHA and CSRF input element classes, they have special parameters that are required for them to perform their functions. In the case of the form element collection classes, they provide a grouping of elements within a fieldset for easier management.

Here's an example that creates and renders a simple input text element:

```
$text = new Pop\Form\Element\Input\Text('first_name');
$text->setRequired(true);
$text->setAttribute('size', 40);
echo $text;
```

The above code will produce:

```
<input name="first_name" id="first_name" type="text" required="required" size="40" />
```

Note the *required* attribute. Since the element was set to be required, this will assign that attribute to the element, which is only effective client-side, if the client interface hasn't bypassed HTML form validation. If the client interface has bypassed HTML form validation, then the form object will still account for the required setting when validating server-side with PHP. If the field is set to be required and it is empty, validation will fail.

Also, the *name* and *id* attributes of the element are set from the first *$name* parameter that is passed into the object. However, if you wish to override these, you can by doing this:

```
$text = new Pop\Form\Element\Input\Text('first_name');
$text->setAttribute('size', 40);
$text->setAttribute('id', 'my-custom-id');
echo $text;
```

The above code will produce:

```
<input name="first_name" id="my-custom-id" type="text" size="40" />
```

Here's an example of a select element:

```
$select = new Pop\Form\Element\Select('colors', [
    'Red'   => 'Red',
    'Green' => 'Green',
    'Blue'  => 'Blue'
]);
$select->setAttribute('class', 'drop-down');
echo $select;
```

The above code will produce:

```
<select name="colors" id="colors" class="drop-down">
    <option value="Red">Red</option>
    <option value="Green">Green</option>
    <option value="Blue">Blue</option>
</select>
```

Here's an example of a checkbox set:

```
$checkbox = new Pop\Form\Element\CheckboxSet('colors', [
    'Red'   => 'Red',
    'Green' => 'Green',
    'Blue'  => 'Blue'
]);
echo $checkbox;
```

The above code will produce:

```
<fieldset class="checkbox-fieldset">
    <input class="checkbox" type="checkbox" name="colors[]" id="colors" value="Red" />
    <span class="checkbox-span">Red</span>
```

```
    <input class="checkbox" type="checkbox" name="colors[]" id="colors1" value="Green
↪" />
    <span class="checkbox-span">Green</span>
    <input class="checkbox" type="checkbox" name="colors[]" id="colors2" value="Blue"␣
↪/>
    <span class="checkbox-span">Blue</span>
</fieldset>
```

In the special case of a field collection set, the object manages the creation and assignment of values and other elements, such as the ** elements that hold the field values. Each element has a class attribute that can be utilized for styling.

Labels

When you create instances of form elements, you can set the label to uses in conjunction with the element. This is typically used when rendering the main form object.

```
$text = new Pop\Form\Element\Input\Text('first_name');
$text->setLabel('First Name:');
```

When rendered with the form, the label will render like this:

```
<label for="first_name">First Name:</label>
```

Validators

When if comes to attaching validators to a form element, there are a few options. The default option is to use the *popphp/pop-validator* component. You can use the standard set of validator classes included in that component, or you can write your own by extending the main *Pop\Validator\AbstractValidator* class. Alternatively, if you'd like to create your own, independent validators, you can do that as well. You just need to pass it something that is callable.

Here's an example using the *popphp/pop-validator* component:

```
$text = new Pop\Form\Element\Input\Text('first_name');
$text->addValidator(new Pop\Validator\AlphaNumeric());
```

If the field's valid was set to something that wasn't alphanumeric, then it would fail validation:

```
$text->setValue('abcd#$%');
if (!$text->validate()) {
    print_r($text->getErrors());
}
```

If using a custom validator that is callable, the main guideline you would have to follow is that upon failure, your validator should return a failure message, otherwise, simply return null. Those messages are what is collected in the elements *$errors* array property for error message display. Here's an example:

```
$myValidator = function($value) {
    if (preg_match('/^\w+$/', $value) == 0) {
        return 'The value is not alphanumeric.';
    } else {
        return null;
    }
};
```

```
$text = new Pop\Form\Element\Input\Text('first_name');
$text->addValidator($myValidator);

$text->setValue('abcd#$%');
if (!$text->validate()) {
    print_r($text->getErrors());
}
```

Form Objects

The form object serves as the center of the functionality. You can create a form object and inject form elements into it. The form object then manages those elements, their values and processes the validation, if any, attached to the form elements. Consider the following code:

```
use Pop\Form\Form;
use Pop\Form\Element\Input;
use Pop\Validator;

$form = new Form();
$form->setAttribute('id', 'my-form');

$username = new Input\Text('username');
$username->setLabel('Username:')
         ->setRequired(true)
         ->setAttribute('size', 40)
         ->addValidator(new Validator\AlphaNumeric());

$email = new Input\Email('email');
$email->setLabel('Email:')
      ->setRequired(true)
      ->setAttribute('size', 40);

$submit = new Input\Submit('submit', 'SUBMIT');

$form->addFields([$username, $email, $submit]);

if ($_POST) {
    $form->setFieldValues($_POST);
    if (!$form->isValid()) {
        echo $form; // Re-render, form has errors
    } else {
        echo 'Valid!';
        print_r($form->toArray());
    }
} else {
    echo $form;
}
```

The form's action is pulled from the current *REQUEST_URI* of the current page, unless otherwise directly specified. Also, the form's method defaults to *POST* unless otherwise specified. The above code will produce the following HTML as the initial render by default:

```
<form action="/" method="post" id="my-form">
    <fieldset id="my-form-fieldset-1" class="my-form-fieldset">
        <dl>
```

```
            <dt>
                <label for="username" class="required">Username:</label>
            </dt>
            <dd>
                <input type="text" name="username" id="username" value="" required=
↪"required" size="40" />
            </dd>
            <dt>
                <label for="email" class="required">Email:</label>
            </dt>
            <dd>
                <input type="email" name="email" id="email" value="" required=
↪"required" size="40" />
            </dd>
            <dd>
                <input type="submit" name="submit" id="submit" value="SUBMIT" />
            </dd>
        </dl>
    </fieldset>
</form>
```

If the user were to input non-valid data into on of the fields, and then submit the form, then the script would be processed again, this time, it would trigger the form validation and render with the error messages, like this:

```
<form action="/" method="post" id="my-form">
    <fieldset id="my-form-fieldset-1" class="my-form-fieldset">
        <dl>
            <dt>
                <label for="username" class="required">Username:</label>
            </dt>
            <dd>
                <input type="text" name="username" id="username" value="dfvdfv##$dfv"␣
↪required="required" size="40" />
                <div class="error">The value must only contain alphanumeric␣
↪characters.</div>
            </dd>
            <dt>
                <label for="email" class="required">Email:</label>
            </dt>
            <dd>
                <input type="email" name="email" id="email" value="" required=
↪"required" size="40" />
            </dd>
            <dd>
                <input type="submit" name="submit" id="submit" value="SUBMIT" />
            </dd>
        </dl>
    </fieldset>
</form>
```

As you can see above, the values entered by the user are retained so that they may correct any errors and re-submit the form. Once the form is corrected and re-submitted, it will pass validation and then move on to the portion of the script that will handle what to do with the form data.

Using Filters

When dealing with the data that is being passed through a form object, besides validation, you'll want to consider adding filters to further protect against bad or malicious data. We can modify the above example to add filters to be used to process the form data before it is validated or re-rendered to the screen. A filter can be anything that is callable, like this:

```
if ($_POST) {
    $form->addFilter('strip_tags');
    $form->addFilter('htmlentities', [ENT_QUOTES, 'UTF-8']);
    $form->setFieldValues($_POST);
    if (!$form->isValid()) {
        echo $form; // Has errors
    } else {
        echo 'Valid!';
        print_r($form->getFields());
    }
} else {
    echo $form;
}
```

In the above code, the *addFilter* methods are called before the data is set into the form for validation or re-rendering. The example passes the *strip_tags* and *htmlentities* functions and those functions are applied to the each value of form data. So, if a user tries to submit the data *<script>alert("Bad Code");</script>* into one of the fields, it would get filtered and re-rendered like this:

```
<input type="text" name="username" id="username" value="alert(&amp;quot;Bad Code&amp;quot;);"␣
↪required="required" size="40" />
```

As you can see, the *<script>* tags were stripped and the quotes were converted to HTML entities.

Field Configurations

Most of the functionality outlined above can be administered and managed by passing field configuration arrays into the form object. This helps facilitate and streamline the form creation process. Consider the following example:

```
use Pop\Form\Form;
use Pop\Validator;

$fields = [
    'username' => [
        'type'       => 'text',
        'label'      => 'Username',
        'required'   => true,
        'validators' => new Validator\AlphaNumeric(),
        'attributes' => [
            'class' => 'username-field',
            'size'  => 40
        ]
    ],
    'password' => [
        'type'       => 'password',
        'label'      => 'Password',
        'required'   => true,
        'validators' => new Validator\GreaterThanEqual(6),
        'attributes' => [
```

```
            'class' => 'password-field',
            'size'  => 40
        ]
    ],
    'submit' => [
        'type'       => 'submit',
        'value'      => 'SUBMIT',
        'attributes' => [
            'class' => 'submit-btn'
        ]
    ]
];

$form = Form::createFromConfig($fields);
$form->setAttribute('id', 'login-form');

echo $form;
```

which will produce the following HTML code:

```
<form action="/" method="post" id="login-form">
    <fieldset id="login-form-fieldset-1" class="login-form-fieldset">
        <dl>
            <dt>
                <label for="username" class="required">Username</label>
            </dt>
            <dd>
                <input type="text" name="username" id="username" value="" required=
↪"required" class="username-field" size="40" />
            </dd>
            <dt>
                <label for="password" class="required">Password</label>
            </dt>
            <dd>
                <input type="password" name="password" id="password" value="" ␣
↪required="required" class="password-field" size="40" />
            </dd>
            <dd>
                <input type="submit" name="submit" id="submit" value="SUBMIT" class=
↪"submit-btn" />
            </dd>
        </dl>
    </fieldset>
</form>
```

In the above example, the *$fields* is an associative array where the keys are the names of the fields and the array values contain the field configuration values. Some of the accepted field configuration values are:

- `'type'` - field type, i.e. 'button', 'select', 'text', 'textarea', 'checkbox', 'radio', 'input-button'
- `'label'` - field label
- `'required'` - boolean to set whether the field is required or not. Defaults to false.
- `'attributes'` - an array of attributes to apply to the field.
- `'validators'` - an array of validators to apply to the field. Can be a single callable validator as well.
- `'value'` - the value to be set for the field

- `'values'` - the option values to be set for the field (for selects, checkboxes and radios)
- `'selected'` - the field value or values that are to be marked as 'selected' within the field's values.
- `'checked'` - the field value or values that are to be marked as 'checked' within the field's values.

Here is an example using fields with multiple values:

```
use Pop\Form\Form;
use Pop\Validator;

$fields = [
    'colors' => [
        'type'   => 'checkbox',
        'label'  => 'Colors',
        'values' => [
            'Red'   => 'Red',
            'Green' => 'Green',
            'Blue'  => 'Blue'
        ],
        'checked' => [
            'Red', 'Green'
        ]
    ],
    'country' => [
        'type'   => 'select',
        'label'  => 'Country',
        'values' => [
            'United States' => 'United States',
            'Canada'        => 'Canada',
            'Mexico'        => 'Mexico'
        ],
        'selected' => 'United States'
    ]
];

$form = Form::createFromConfig($fields);

echo $form;
```

which will produce:

```
<form action="/" method="post">
    <fieldset id="pop-form-fieldset-1" class="pop-form-fieldset">
        <dl>
            <dt>
                <label for="colors1">Colors</label>
            </dt>
            <dd>
                <fieldset class="checkbox-fieldset">
                    <input type="checkbox" name="colors[]" id="colors" value="Red"␣
↪class="checkbox" checked="checked" />
                    <span class="checkbox-span">Red</span>
                    <input type="checkbox" name="colors[]" id="colors1" value="Green"␣
↪class="checkbox" checked="checked" />
                    <span class="checkbox-span">Green</span>
                    <input type="checkbox" name="colors[]" id="colors2" value="Blue"␣
↪class="checkbox" />
                    <span class="checkbox-span">Blue</span>
                </fieldset>
```

```
            </dd>
            <dt>
                <label for="country">Country</label>
            </dt>
            <dd>
                <select name="country" id="country">
                    <option value="United States" selected="selected">United States</
↪option>
                    <option value="Canada">Canada</option>
                    <option value="Mexico">Mexico</option>
                </select>
            </dd>
        </dl>
    </fieldset>
</form>
```

Fieldsets

As you've seen in the above examples, the fields that are added to the form object are enclosed in a fieldset group. This can be leveraged to create other fieldset groups as well as give them legends to better define the fieldsets.

```
use Pop\Form\Form;
use Pop\Validator;

$fields1 = [
    'username' => [
        'type'       => 'text',
        'label'      => 'Username',
        'required'   => true,
        'validators' => new Validator\AlphaNumeric(),
        'attributes' => [
            'class' => 'username-field',
            'size'  => 40
        ]
    ],
    'password' => [
        'type'       => 'password',
        'label'      => 'Password',
        'required'   => true,
        'validators' => new Validator\GreaterThanEqual(6),
        'attributes' => [
            'class' => 'password-field',
            'size'  => 40
        ]
    ]
];
$fields2 = [
    'submit' => [
        'type'       => 'submit',
        'value'      => 'SUBMIT',
        'attributes' => [
            'class' => 'submit-btn'
        ]
    ]
];
```

```
$form = Form::createFromConfig($fields1);
$form->getFieldset()->setLegend('First Fieldset');
$form->createFieldset('Second Fieldset');
$form->addFieldsFromConfig($fields2);

echo $form;
```

In the above code, the first set of fields are added to an initial fieldset that's automatically created. After that, if you want to add more fieldsets, you call the `createFieldset` method like above. Then the current fieldset is changed to the newly created one and the next fields are added to that one. You can always change to any other fieldset by using the `setCurrent($i)` method. The above code would render like this:

```
<form action="/" method="post">
    <fieldset id="pop-form-fieldset-1" class="pop-form-fieldset">
        <legend>First Fieldset</legend>
        <dl>
            <dt>
                <label for="username" class="required">Username:</label>
            </dt>
            <dd>
                <input type="text" name="username" id="username" value="" required=
↪"required" size="40" />
            </dd>
            <dt>
                <label for="email" class="required">Email:</label>
            </dt>
            <dd>
                <input type="email" name="email" id="email" value="" required=
↪"required" size="40" />
            </dd>
        </dl>
    </fieldset>
    <fieldset id="pop-form-fieldset-2" class="pop-form-fieldset">
        <legend>Second Fieldset</legend>
        <dl>
            <dd>
                <input type="submit" name="submit" id="submit" value="SUBMIT" />
            </dd>
        </dl>
    </fieldset>
</form>
```

The container elements within the fieldset can be controlled by passing a value to the `$container` parameter. The default is *dl*, but *table*, *div* and *p* are supported as well.

```
$form->createFieldset('Second Fieldset', 'table');
```

Alternately, you can inject an entire fieldset configuration array. The code below is a more simple way to inject the fieldset configurations and their legends. And, it will generate the same HTML as above.

```
use Pop\Form\Form;
use Pop\Validator;

$fieldsets = [
    'First Fieldset' => [
        'username' => [
            'type'       => 'text',
```

```
            'label'      => 'Username',
            'required'   => true,
            'validators' => new Validator\AlphaNumeric(),
            'attributes' => [
                'class' => 'username-field',
                'size'  => 40
            ]
        ],
        'password' => [
            'type'       => 'password',
            'label'      => 'Password',
            'required'   => true,
            'validators' => new Validator\GreaterThanEqual(6),
            'attributes' => [
                'class' => 'password-field',
                'size'  => 40
            ]
        ]
    ],
    'Second Fieldset' => [
        'submit' => [
            'type'       => 'submit',
            'value'      => 'SUBMIT',
            'attributes' => [
                'class' => 'submit-btn'
            ]
        ]
    ]
];

$form = Form::createFromFieldsetConfig($fieldsets);

echo $form;
```

Using Views

You can still use the form object for managing and validating your form fields and still send the individual components to a view for you to control how they render as needed. You can do that like this:

```
use Pop\Form\Form;
use Pop\Validator;

$fields = [
    'username' => [
        'type'       => 'text',
        'label'      => 'Username',
        'required'   => true,
        'validators' => new Validator\AlphaNumeric(),
        'attributes' => [
            'class' => 'username-field',
            'size'  => 40
        ]
    ],
    'password' => [
        'type'       => 'password',
        'label'      => 'Password',
```

```
        'required'   => true,
        'validators' => new Validator\GreaterThanEqual(6),
        'attributes' => [
            'class' => 'password-field',
            'size'  => 40
        ]
    ],
    'submit' => [
        'type'       => 'submit',
        'value'      => 'SUBMIT',
        'attributes' => [
            'class' => 'submit-btn'
        ]
    ]
];

$form = Form::createFromConfig($fields);
$formData = $form->prepareForView();
```

You can then pass the array `$formData` off to your view object to be rendered as you need it to be. That array will contain the following `key => value` entries:

```
$formData = [
    'username'        => '<input type="text" name="username"...',
    'username_label'  => '<label for="username" ...',
    'username_errors' => [],
    'password'        => '<input type="text" name="username"...',
    'password_label'  => '<label for="username" ...',
    'password_errors' => [],
    'submit'          => '<input type="submit" name="submit"...',
    'submit_label'    => '',
]
```

Or, if you want even more control, you can send the form object itself into your view object and access the components like this:

```
<form action="/" method="post" id="login-form">
    <fieldset id="login-form-fieldset-1" class="login-form-fieldset">
        <dl>
            <dt>
                <label for="username" class="required"><?=$form->getField('username')-
↪>getLabel(); ?></label>
            </dt>
            <dd>
                <?=$form->getField('username'); ?>
<?php if ($form->getField('username')->hasErrors(): ?>
<?php foreach ($form->getField('username')->getErrors() as $error): ?>
                <div class="error"><?=$error; ?></div>
<?php endforeach; ?>
<?php endif; ?>
            </dd>
            <dt>
                <label for="password" class="required"><?=$form->getField('password')-
↪>getLabel(); ?></label>
            </dt>
            <dd>
                <?=$form->getField('password'); ?>
<?php if ($form->getField('password')->hasErrors(): ?>
```

```
<?php foreach ($form->getField('password')->getErrors() as $error): ?>
                <div class="error"><?=$error; ?></div>
<?php endforeach; ?>
<?php endif; ?>
            </dd>
            <dd>
                <?=$form->getField('submit'); ?>
            </dd>
        </dl>
    </fieldset>
</form>
```

pop-ftp

The *popphp/pop-ftp* component provides a simple API for the managing FTP connections and transferring files over FTP.

Installation

Install it directly into your project:

```
composer require popphp/pop-ftp
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-ftp": "3.0.*",
    }
}
```

Basic Use

Create a new directory, change into it and upload a file:

```
use Pop\Ftp\Ftp;

$ftp = new Ftp('ftp.myserver.com', 'username', 'password');

$ftp->mkdir('somedir');
$ftp->chdir('somedir');

$ftp->put('file_on_server.txt', 'my_local_file.txt');
```

Download a file from a directory:

```
use Pop\Ftp\Ftp;

$ftp = new Ftp('ftp.myserver.com', 'username', 'password');

$ftp->chdir('somedir');
```

```
$ftp->get('my_local_file.txt', 'file_on_server.txt');
```

pop-http

The *popphp/pop-http* component provides a robust API to handle HTTP requests and responses. Also, it provides HTTP client adapters via cURL and streams, as well as file uploads via HTTP.

Installation

Install it directly into your project:

```
composer require popphp/pop-http
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-http": "3.0.*",
    }
}
```

Basic Use

The `popphp/pop-http` component contains a **request object** and a **response object** that can assist in capturing and managing the incoming requests to your application and handle assembling the appropriate response back to the user.

Requests

The main request class is `Pop\Http\Request`. It has a robust API to allow you to interact with the incoming request and extract data from it. If you pass nothing to the constructor a new request object, it will attempt to parse the value contained in `$_SERVER['REQUEST_URI']`. You can, however, pass it a `$uri` to force a specific request, and also a `$basePath` to let the request object know that the base of the application is contained in a sub-folder under the document root.

Creating a new request object with a base path

In the following example, let's assume our application is in a sub-folder under the main document root:

- `/httpdocs`
- `/httpdocs/system`
- `/httpdocs/system/index.php`

We create a request object and pass it the base path `/system` so that application knows to parse incoming request after the `/system` base path.

```
$request = new Pop\Http\Request(null, '/system');
```

For example, if a request of `/system/users` came in, the application would know to use `/users` as the request and route it accordingly. If you need to reference the request URI, there are a couple of different methods to do so:

- `$request->getBasePath();` - returns only the base path ('/system')
- `$request->getRequestUri();` - returns only the request URI ('/users')
- `$request->getFullRequestUri();` - returns the full request URI string ('/system/users')

Getting path segments

If you need to break apart a URI into its segments access them for your application, you can do it with the `getSegment()` method. Consider the URI `/users/edit/1001`:

- `$request->getSegment(0);` - returns 'users'
- `$request->getSegment(1);` - returns 'edit'
- `$request->getSegment(2);` - returns '1001'
- `$request->getSegments();` - returns an array containing all of the path segments

Check the HTTP Method

- `$request->isGet();`
- `$request->isHead();`
- `$request->isPost();`
- `$request->isPut();`
- `$request->isPatch();`
- `$request->isDelete();`
- `$request->isTrace();`
- `$request->isHead();`
- `$request->isOptions();`
- `$request->isConnect();`

Retrieve Data from the Request

- `$request->getQuery($key = null);`
- `$request->getPost($key = null);`
- `$request->getFiles($key = null);`
- `$request->getPut($key = null);`
- `$request->getPatch($key = null);`
- `$request->getDelete($key = null);`
- `$request->getServer($key = null);`
- `$request->getEnv($key = null);`

If you do not pass the `$key` parameter in the above methods, the full array of values will be returned. The results from the `getQuery()`, `getPost()` and `getFiles()` methods mirror what is contained in the `$_GET`, `$_POST` and `$_FILES` global arrays, respectively. The `getServer()` and `getEnv()` methods mirror the `$_SERVER` and `$_ENV` global arrays, respectively.

If the request method passed is **PUT**, **PATCH** or **DELETE**, the request object will attempt to parse the raw request data to provide the data from that. The request object will also attempt to be content-aware and parse JSON or XML from the data if it successfully detects a content type from the request.

If you need to access the raw request data or the parsed request data, you can do so with these methods:

- `$request->getRawData();`
- `$request->getParsedData();`

Retrieve Request Headers

- `$request->getHeader($key);` - return a single request header value
- `$request->getHeaders();` - return all header values in an array

Responses

The `Pop\Http\Response` class has a full-featured API that allows you to create a outbound response to send back to the user or parse an inbound response from a request. The main constructor of the response object accepts a configuration array with the basic data to get the response object started:

```
$response = new Pop\Http\Response([
    'code'    => 200,
    'message' => 'OK',
    'version' => '1.1',
    'body'    => 'Some body content',
    'headers' => [
        'Content-Type' => 'text/plain'
    ]
]);
```

All of that basic response data can also be set as needed through the API:

- `$response->setCode($code);` - set the response code
- `$response->setMessage($message);` - set the response message
- `$response->setVersion($version);` - set the response version
- `$response->setBody($body);` - set the response body
- `$response->setHeader($name, $value);` - set a response header
- `$response->setHeaders($headers);` - set response headers from an array

And retrieved as well:

- `$response->getCode();` - get the response code
- `$response->getMessage();` - get the response message
- `$response->getVersion();` - get the response version
- `$response->getBody();` - get the response body
- `$response->getHeader($name);` - get a response header
- `$response->getHeaders($headers);` - get response headers as an array
- `$response->getHeadersAsString();` - get response headers as a string

Check the Response

- `$response->isSuccess();` - 100, 200 or 300 level response code
- `$response->isRedirect();` - 300 level response code
- `$response->isError();` - 400 or 500 level response code
- `$response->isClientError();` - 400 level response code

- `$response->isServerError();` - 500 level response code

And you can get the appropriate response message from the code like this:

```
use Pop\Http\Response;

$response = new Response();
$response->setCode(403);
$response->setMessage(Response::getMessageFromCode(403)); // Sets 'Forbidden'
```

Sending the Response

```
$response = new Pop\Http\Response([
    'code'    => 200,
    'message' => 'OK',
    'version' => '1.1',
    'body'    => 'Some body content',
    'headers' => [
        'Content-Type'   => 'text/plain'
    ]
]);

$response->setHeader('Content-Length', strlen($response->getBody()));
$response->send();
```

The above example would produce something like:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 19

Some body content
```

Redirecting a Response

```
Pop\Http\Response::redirect('http://www.domain.com/some-new-page');
exit();
```

Parsing a Response

In parsing a response from a request, you pass either the URL or a response string that already exists. A new response object with all of its data parsed from that response will be created:

```
$response = Pop\Http\Response::parse('http://www.domain.com/some-page');

if ($response->getCode() == 200) {
    // Do something with the response
} else if ($response->isError()) {
    // Uh oh. Something went wrong
}
```

File Uploads

With the file upload class, you can not only control basic file uploads, but also enforce a set of rules and conditions to control what type of files are uploaded. Please note, the `upload()` method expects to have an element from the `$_FILES` array passed into it.

```
use Pop\Http\Upload;

$upload = new Upload('/path/to/uploads');
$upload->useDefaults();

$upload->upload($_FILES['file_upload']);

// Do something with the newly uploaded file
if ($upload->isSuccess()) {
    $file = $upload->getUploadedFile();
} else {
    echo $upload->getErrorMessage();
}
```

The `setDefaults()` method sets a standard group of rules and conditions for basic web file uploads. The max filesize is set to 10 MBs and a set of media and document file types (jpg, pdf, doc, etc.) are set as *allowed* and a set of web and script file types (js, php, html, etc.) are set as *disallowed*.

If you'd like to set your own custom rules, you can do so like this:

```
use Pop\Http\Upload;

$upload = new Upload('/path/to/uploads');
$upload->setMaxSize(25000000)
       ->setAllowedTypes('pdf')
       ->setDisallowedTypes('php');
```

The example above sets the max filesize to 25 MBs and allows only PDF files and disallows PHP files.

Checking file names

The upload object is set to NOT overwrite existing files on upload. It will perform a check and rename the uploaded file accordingly with a underscore and a number ('filename_1.doc', 'filename_2.doc', etc.) If you may want to test the filename on your own you can like this:

```
use Pop\Http\Upload;

$upload   = new Upload('/path/to/uploads');
$fileName = $upload->checkFilename($_FILES['file_upload']['name']);
```

If the name of the file being uploaded is found on disk in the upload directory, the returned value of the newly renamed file will be something like 'filename_1.doc'. You can then pass that value (or any other custom filename value) into the `upload()` method:

```
$upload->upload($_FILES['file_upload'], $fileName);
```

If you want to override this behavior and overwrite any existing files, you can set the overwrite property before you upload the file:

```
$upload->overwrite(true);
```

pop-i18n

The *popphp/pop-i18n* component is the internationalization and localization component. It handles the translation of strings into the correct and necessary language for the country and region.

Installation

Install it directly into your project:

```
composer require popphp/pop-i18n
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-i18n": "3.0.*",
    }
}
```

Basic Use

The *popphp/pop-i18n* component handles internationalization and localization. It provides the features for translating and managing different languages and locales that may be required for an application. It also provides for parameters to be injected into the text for personalization. To use the component, you'll have to create you language and locale files. The accepted formats are either XML or JSON:

fr.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE language [
        <!ELEMENT language ANY>
        <!ELEMENT locale ANY>
        <!ELEMENT text (source,output)>
        <!ELEMENT source ANY>
        <!ELEMENT output ANY>
        <!ATTLIST language
                src       CDATA    #REQUIRED
                output    CDATA    #REQUIRED
                name      CDATA    #REQUIRED
                native    CDATA    #REQUIRED
                >
        <!ATTLIST locale
                region    CDATA    #REQUIRED
                name      CDATA    #REQUIRED
                native    CDATA    #REQUIRED
                >
        ]>
<language src="en" output="fr" name="French" native="Français">
    <locale region="FR" name="France" native="France">
        <text>
            <source>Hello, my name is %1. I love to program %2.</source>
            <output>Bonjour, mon nom est %1. Je aime programmer %2.</output>
        </text>
    </locale>
</language>
```

fr.json

```
{
    "language"  : {
        "src"    : "en",
        "output" : "fr",
```

```
        "name"   : "French",
        "native" : "Français",
        "locale" : [{
            "region" : "FR",
            "name"   : "France",
            "native" : "France",
            "text"   : [
                {
                    "source" : "Hello, my name is %1. I love to program %2.",
                    "output" : "Bonjour, mon nom est %1. Je aime programmer %2."
                }
            ]
        }]
    }
}
```

From there, you can create your I18n object and give it the folder with the language files in it. It will auto-detect which file to load based on the language passed.

```
use Pop\I18n\I18n;

$lang = new I18n('fr_FR', '/path/to/language/files');

$string = $lang->__('Hello, my name is %1. I love to program %2.', ['Nick', 'PHP']);
echo $string;
```

```
Bonjour, mon nom est Nick. Je aime programmer PHP.
```

Alternatively, you can directly echo the string out like this:

```
$lang->_e('Hello, my name is %1. I love to program %2.', ['Nick', 'PHP']);
```

The I18n Constant

You can set the language and locale when you instantiate the I18n object like above, or if you prefer, you can set it in your application as a constant `POP_LANG` and the I18n object will look for that as well. The default is `en_US`.

Advanced Use

The *popphp/pop-i18n* component provides the functionality to assist you in generating your required language files. Knowing the time and possibly money required to translate your application's text into multiple languages, the component can help with assembling the language files once you have the content.

You can give it arrays of data to generate complete files:

```
use Pop\I18n\Format;

$lang = [
    'src'    => 'en',
    'output' => 'de',
    'name'   => 'German',
    'native' => 'Deutsch'
];

$locales = [
    [
```

```
        'region' => 'DE',
        'name'   => 'Germany',
        'native' => 'Deutschland',
        'text' => [
            [
                'source' => 'This field is required.',
                'output' => 'Dieses Feld ist erforderlich.'
            ],
            [
                'source' => 'Please enter your name.',
                'output' => 'Bitte geben Sie Ihren Namen ein.'
            ]
        ]
    ]
];

// Create the XML format
Format\Xml::createFile($lang, $locale, '/path/to/language/files/de.xml');

// Create in JSON format
Format\Json::createFile($lang, $locale, '/path/to/language/files/de.json');
```

Also, if you have a a source text file and an output text file with a 1:1 line-by-line ratio, then you can create the language files in fragment set and merge them as needed. An example of a 1:1 ratio source-to-output text files:

source/en.txt

```
This field is required.
Please enter your name.
```

source/de.txt

```
Dieses Feld ist erforderlich.
Bitte geben Sie Ihren Namen ein.
```

So then, you can do this:

```
use Pop\I18n\Format;

// Create the XML format fragment
Format\Xml::createFragment('source/en.txt', 'output/de.txt', '/path/to/files/');

// Create the JSON format fragment
Format\Json::createFragment('source/en.txt', 'output/de.txt', '/path/to/files/');
```

And merge the fragments into a main language file.

pop-image

The *popphp/pop-image* component provides a robust API for image creation and manipulation. Adapters are provided to utilize either the GD, Imagick or Gmagick extensions. Also, the SVG format is supported with its own adapter as well.

Installation

Install it directly into your project:

```
composer require popphp/pop-image
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-image": "3.0.*",
    }
}
```

Basic Use

Image manipulation and processing is another set of features that is often needed for a web application. It is common to have to process images in some way for the web application to perform its required functionality. The *popphp/pop-image* component provides that functionality with a robust set of image processing and manipulation features. Within the component are adapters written to support the `Gd`, `Imagick` and `Gmagick` extensions.

By using either the `Imagick` or `Gmagick` adapters*[0], you will open up a larger set of features and functionality for your application, such as the ability to handle more image formats and perform more complex image processing functions.

Choose an Adapter

Before you choose which image adapter to use, you may have to determine which PHP image extensions are available for your application within its environment. There is an API to assist you with that. The following example tests for each individual adapter to see if one is available, and if not, then moves on to the next:

```
if (Pop\Image\Gmagick::isAvailable()) {
    $image = Pop\Image\Gmagick::load('image.jpg');
} else if (Pop\Image\Imagick::isAvailable()) {
    $image = Pop\Image\Imagick::load('image.jpg');
} else if (Pop\Image\Gd::isAvailable()) {
    $image = Pop\Image\Gd::load('image.jpg');
}
```

Similarly, you can check their availability like this as well:

```
// This will work with any of the 3 adapters
$adapters = Pop\Image\Image::getAvailableAdapters();

if ($adapters['gmagick']) {
    $image = Pop\Image\Gmagick::load('image.jpg');
} else if ($adapters['imagick']) {
    $image = Pop\Image\Imagick::load('image.jpg');
} else if ($adapters['gd']) {
    $image = Pop\Image\Gd::load('image.jpg');
}
```

[0] It must be noted that the `imagick` and `gmagick` extensions cannot be used at the same time as they have conflicts with shared libraries and components that are used by both extensions.

As far as which adapter or extension is the "best" for your application, that will really depend on your application's needs and what's available in the environment on which your application is running. If you require advanced image processing that can work with a large number of image formats, then you'll need to utilize either the `Imagick` or `Gmagick` adapters. If you only require simple image processing with a limited number of image formats, then the `Gd` adapter should work well.

The point of the API of the *popphp/pop-image* component is to help make applications more portable and mitigate any issues that may arise should an application need to be installed on a variety of different environments. The goal is to achieve a certain amount of "graceful degradation," should one of the more feature-rich image extensions not be available on a new environment.

Basic Use

Loading an Image

You can load an existing image from disk like this:

```
// Returns an instance of Pop\Image\Adapter\Gd with the image resource loaded
$image = Pop\Image\Gd::load('path/to/image.jpg');
```

Or you can load an image from a data source like this:

```
// Returns an instance of Pop\Image\Adapter\Gd with the image resource loaded
$image = Pop\Image\Gd::loadFromString($imageData);
```

Or create an instance of an image object with a new image resource via:

```
// Returns an instance of Pop\Image\Gd with a new image resource loaded
$image =  Pop\Image\Gd::create(640, 480, 'new.jpg');
```

All three of the above adapters have the same core API below:

- `$img->resizeToWidth($w);` - resize the image to a specified width
- `$img->resizeToHeight($h);` - resize the image to a specified height
- `$img->resize($px);` - resize image to largest dimension
- `$img->scale($scale);` - scale image by percentage, 0.0 - 1.0
- `$img->crop($w, $h, $x = 0, $y = 0);` - crop image to specified width and height
- `$img->cropThumb($px, $offset = null);` - crop image to squared image of specified size
- `$img->rotate($degrees, Color\ColorInterface $bgColor = null, $alpha = null);` - rotate image by specified degrees
- `$img->flip();` - flip the image over the x-axis
- `$img->flop();` - flip the image over the y-axis
- `$img->convert($to);` - convert image to specified image type
- `$img->writeToFile($to = null, $quality = 100);` - save image, either to itself or a new location
- `$img->outputToHttp($quality = 100, $to = null, $download = false, $sendHeaders = true);` - output image via HTTP

Advanced Use

The *popphp/pop-image* component comes with set of image manipulation objects that provide a more advanced feature set when processing images. You can think of these classes and their object instances as the menus at the top of your favorite image editing software.

Adjust

The adjust object allows you to perform the following methods:

- `$img->adjust->brightness($amount);`
- `$img->adjust->contrast($amount);`
- `$img->adjust->desaturate();`

And with the `Imagick` or `Gmagick` adapter, you can perform these advanced methods:

- `$img->adjust->hue($amount);`
- `$img->adjust->saturation($amount);`
- `$img->adjust->hsb($h, $s, $b);`
- `$img->adjust->level($black, $gamma, $white);`

Here's an example making some adjustments to the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->adjust->brightness(50)
    ->contrast(20)
    ->level(0.7, 1.0, 0.5);
```

Draw

The draw object allows you to perform the following methods:

- `$img->draw->line($x1, $y1, $x2, $y2);`
- `$img->draw->rectangle($x, $y, $w, $h = null);`
- `$img->draw->square($x, $y, $w);`
- `$img->draw->ellipse($x, $y, $w, $h = null);`
- `$img->draw->circle($x, $y, $w);`
- `$img->draw->arc($x, $y, $start, $end, $w, $h = null);`
- `$img->draw->chord($x, $y, $start, $end, $w, $h = null);`
- `$img->draw->pie($x, $y, $start, $end, $w, $h = null);`
- `$img->draw->polygon($points);`

And with the `Imagick` or `Gmagick` adapter, you can perform these advanced methods:

- `$img->draw->roundedRectangle($x, $y, $w, $h = null, $rx = 10, $ry = null);`
- `$img->draw->roundedSquare($x, $y, $w, $rx = 10, $ry = null);`

Here's an example drawing some different shapes with different styles on the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->draw->setFillColor(255, 0, 0);
    ->draw->setStrokeColor(0, 0, 0);
    ->draw->setStrokeWidth(5);
    ->draw->rectangle(100, 100, 320, 240);
    ->draw->circle(400, 300, 50);
```

Effect

The effect object allows you to perform the following methods:

- `$img->effect->border(array $color, $w, $h = null);`
- `$img->effect->fill($r, $g, $b);`
- `$img->effect->radialGradient(array $color1, array $color2);`
- `$img->effect->verticalGradient(array $color1, array $color2);`
- `$img->effect->horizontalGradient(array $color1, array $color2);`
- `$img->effect->linearGradient(array $color1, array $color2, $vertical = true);`

Here's an example applying some different effects to the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->effect->verticalGradient([255, 0, 0], [0, 0, 255]);
```

Filter

Each filter object is more specific for each image adapter. While a number of the available filter methods are available in all 3 of the image adapters, some of their signatures vary due the requirements of the underlying image extension.

The `Gd` filter object allows you to perform the following methods:

- `$img->filter->blur($amount, $type = IMG_FILTER_GAUSSIAN_BLUR);`
- `$img->filter->sharpen($amount);`
- `$img->filter->negate();`
- `$img->filter->colorize($r, $g, $b);`
- `$img->filter->pixelate($px);`
- `$img->filter->pencil();`

The `Imagick` filter object allows you to perform the following methods:

- `$img->filter->blur($radius = 0, $sigma = 0, $channel = \Imagick::CHANNEL_ALL);`
- `$img->filter->adaptiveBlur($radius = 0, $sigma = 0, $channel = \Imagick::CHANNEL_DEFAULT);`
- `$img->filter->gaussianBlur($radius = 0, $sigma = 0, $channel = \Imagick::CHANNEL_ALL);`
- `$img->filter->motionBlur($radius = 0, $sigma = 0, $angle = 0, $channel = \Imagick::CHANNEL_DEFAULT);`

- `$img->filter->radialBlur($angle = 0, $channel = \Imagick::CHANNEL_ALL);`
- `$img->filter->sharpen($radius = 0, $sigma = 0, $channel = \Imagick::CHANNEL_ALL);`
- `$img->filter->negate();`
- `$img->filter->paint($radius);`
- `$img->filter->posterize($levels, $dither = false);`
- `$img->filter->noise($type = \Imagick::NOISE_MULTIPLICATIVEGAUSSIAN, $channel = \Imagick::CHANNEL_DEFAULT);`
- `$img->filter->diffuse($radius);`
- `$img->filter->skew($x, $y, $color = 'rgb(255, 255, 255)');`
- `$img->filter->swirl($degrees);`
- `$img->filter->wave($amp, $length);`
- `$img->filter->pixelate($w, $h = null);`
- `$img->filter->pencil($radius, $sigma, $angle);`

The `Gmagick` filter object allows you to perform the following methods:

- `$img->filter->blur($radius = 0, $sigma = 0, $channel = \Gmagick::CHANNEL_ALL);`
- `$img->filter->motionBlur($radius = 0, $sigma = 0, $angle = 0);`
- `$img->filter->radialBlur($angle = 0, $channel = \Gmagick::CHANNEL_ALL);`
- `$img->filter->sharpen($radius = 0, $sigma = 0, $channel = \Gmagick::CHANNEL_ALL);`
- `$img->filter->negate();`
- `$img->filter->paint($radius);`
- `$img->filter->noise($type = \Gmagick::NOISE_MULTIPLICATIVEGAUSSIAN);`
- `$img->filter->diffuse($radius);`
- `$img->filter->skew($x, $y, $color = 'rgb(255, 255, 255)');`
- `$img->filter->solarize($threshold);`
- `$img->filter->swirl($degrees);`
- `$img->filter->pixelate($w, $h = null);`

Here's an example applying some different filters to the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->filter->gaussianBlur(10)
    ->swirl(45)
    ->negate();
```

Layer

The layer object allows you to perform the following methods:

- `$img->layer->overlay($image, $x = 0, $y = 0);`

And with the `Imagick` or `Gmagick` adapter, you can perform this advanced method:

- `$img->layer->flatten();`

Here's an example working with layers over the image resource:

```
$img = new Pop\Image\Imagick('image.psd');
$img->layer->flatten()
    ->overlay('watermark.png', 50, 50);
```

Type

The type object allows you to perform the following methods:

- `$img->type->font($font);` - set the font
- `$img->type->size($size);` - set the font size
- `$img->type->x($x);` - set the x-position of the text string
- `$img->type->y($y);` - set the y-position of the text string
- `$img->type->xy($x, $y);` - set both the x- and y-position together
- `$img->type->rotate($degrees);` - set the amount of degrees in which to rotate the text string
- `$img->type->text($string);` - place the string on the image, using the defined parameters

Here's an example working with text over the image resource:

```
$img = new Pop\Image\Imagick('image.jpg');
$img->type->setFillColor(128, 128, 128)
    ->size(12)
    ->font('fonts/Arial.ttf')
    ->xy(40, 120)
    ->text('Hello World!');
```

Extending the Component

The *popphp/pop-image* component was built in a way to facilitate extending it and injecting your own custom image processing features. Knowing that the image processing landscape is vast, the component only scratches the surface and provides the core feature set that was outlined above across the different adapters.

If you are interested in creating and injecting your own, more robust set of features into the component within your application, you can do that by extending the available manipulation classes.

For example, if you wanted to add a couple of methods to the adjust class for the `Gd` adapter, you can do so like this:

```
namespace MyApp\Image;

class CustomAdjust extends \Pop\Image\Adjust\Gd
{
    public function customAction1() {}

    public function customAction2() {}

    public function customAction3() {}
}
```

Then, later in your application, when you call up the Gd adapter, you can inject your custom adjust adapter like this:

```
namespace MyApp;

$image = new \Pop\Image\Gd\('image.jpg');
$image->setAdjust(new MyApp\Image\CustomAdjust());
```

So when you go you use the image adapter, your custom features will be available along will the original set of features:

```
$image->adjust->brightness(50)
    ->customAction1()
    ->customAction2()
    ->customAction3();
```

This way, you can create and call whatever custom features are needed for your application on top of the basic features that are already available.

pop-loader

The *popphp/pop-loader* component provides an alternative autoloading option for those who may require a PHP application to function outside of the Composer eco-system. The API mirrors Composer's API so that the two autoloaders are interchangeable. Furthermore, the component provides a class mapper class that will parse a provided source folder and generate a static classmap for faster autoload times.

Installation

Stand-alone

If you are installing this component as a stand-alone autoloader and not using composer, you can get the component from the releases page on Github. Once you download it and unpack it, you can put the source files into your application's source directory.

Composer

If you want to install it via composer you can install it directly into your project:

```
composer require popphp/pop-loader
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-loader": "3.0.*",
    }
}
```

Basic Use

The *popphp/pop-loader* component manages the autoloading of an application. If, for some reason you do not or cannot use Composer, *popphp/pop-loader* provides an alternative with similar features and API. It supports both PSR-

4 and PSR-0 autoloading standards. Additionally, there is support for generating and loading class maps, if you are interested in boosting the speed and performance of your application's load times.

PSR-4

Let's assume the this class file exists here `app/src/Test.php`:

```
<?php
namespace MyApp;

class Test
{

}
```

You can then register that namespace prefix and source location with the autoloaded object like this:

```
// Require the main ClassLoader class file
require_once __DIR__ . '/../src/ClassLoader.php';

$autoloader = new Pop\Loader\ClassLoader();
$autoloader->addPsr4('MyApp\\', __DIR__ . '/../app/src');

// The class is now available
$test = new MyApp\Test();
```

PSR-0

There's also support for older the PSR-0 standard. If the class file existed here instead `app/MyApp/Test.php`, you could load it like so:

```
// Require the main ClassLoader class file
require_once __DIR__ . '/../src/ClassLoader.php';

$autoloader = new Pop\Loader\ClassLoader();
$autoloader->addPsr0('MyApp', __DIR__ . '/../app');

// The class is now available
$test = new MyApp_Test();
```

Classmaps

Loading

Let's use the following classmap file, `classmap.php`, as an example:

```
<?php

return [
    'MyApp\Foo\Bar' => '/path/to/myapp/src/Foo/Bar.php',
    'MyApp\Thing' => '/path/to/myapp/src/Thing.php',
    'MyApp\Test' => '/path/to/myapp/src/Test.php'
];
```

To load the above classmap, you can do the following:

```
$autoloader = new Pop\Loader\ClassLoader();
$autoloader->addClassMapFromFile('classmap.php');
```

Generating

If you'd like to generate a classmap based on your source folder, you can do that as well:

```
$mapper = new Pop\Loader\ClassMapper('path/to/myapp/src');
$mapper->generateClassMap();
$mapper->writeToFile('path/to/my-classmap.php');
```

From there, you can then set your autoloader to load that classmap for your application.

pop-log

The *popphp/pop-log* component provides basic logging functionality via a few different writers, including file, mail and database logs.

Installation

Install it directly into your project:

```
composer require popphp/pop-log
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-log": "3.0.*",
    }
}
```

Basic Use

The *popphp/pop-log* component is a logging component that provides a way of logging events following the standard BSD syslog protocol outlined in RFC-3164. Support is built-in for writing log messages to a file or database table or deploying them via email. The eight available log message severity values are:

- EMERG (0)
- ALERT (1)
- CRIT (2)
- ERR (3)
- WARN (4)
- NOTICE (5)
- INFO (6)
- DEBUG (7)

and are available via their respective methods:

- `$log->emergency($message);`
- `$log->alert($message);`
- `$log->critical($message);`
- `$log->error($message);`
- `$log->warning($message);`
- `$log->notice($message);`
- `$log->info($message);`
- `$log->debug($message);`

File

Setting up and using a log file is pretty simple. Plain text is the default, but there is also support for CSV, TSV and XML formats:

```
use Pop\Log\Logger;
use Pop\Log\Writer;

$log = new Logger(new Writer\File(__DIR__ . '/logs/app.log'));

$log->info('Just a info message.');
$log->alert('Look Out! Something serious happened!');
```

Then, your 'app.log' file will contain:

```
2015-07-11 12:32:32    6    INFO    Just a info message.
2015-07-11 12:32:33    1    ALERT   Look Out! Something serious happened!
```

Email

Here's an example using email, which requires you to install *popphp/pop-mail*:

```
use Pop\Log\Logger;
use Pop\Log\Writer;
use Pop\Mail;

$mailer = new Mail\Mailer(new Mail\Transport\Sendmail());
$log    = new Logger(new Writer\Mail($mailer, [
    'sysadmin@mydomain.com', 'logs@mydomain.com'
]));

$log->info('Just a info message.');
$log->alert('Look Out! Something serious happened!');
```

Then the emails listed above will receive a series of emails like this:

```
Subject: Log Entry: INFO (6)
2015-07-11 12:32:32    6    INFO    Just a info message.
```

and

```
Subject: Log Entry: ALERT (1)
2015-07-11 12:32:33    1    ALERT   Look Out! Something serious happened!
```

Database

Writing a log to a table in a database requires you to install *popphp/pop-db*:

```
use Pop\Db\Db;
use Pop\Log\Logger;
use Pop\Log\Writer;

$db  = Db::connent('sqlite', __DIR__ . '/logs/.htapplog.sqlite');
$log = new Logger(new Writer\Db($db, 'system_logs'));

$log->info('Just a info message.');
$log->alert('Look Out! Something serious happened!');
```

In this case, the logs are written to a database table that has the columns *id*, *timestamp*, *level*, *name* and *message*. So, after the example above, your database table would look like this:

Id	Timestamp	Level	Name	Message
1	2015-07-11 12:32:32	6	INFO	Just a info message.
2	2015-07-11 12:32:33	1	ALERT	Look Out! Something serious happened!

pop-mail

The *popphp/pop-mail* component provides an API to manage sending mail from your application. Support is built-in for multi-mimetype emails and attachments, as well as multiple recipients and queuing. It has a full feature set that supports:

- Send to email via sendmail, SMTP or any custom-written mail transport adapters
- Send emails to a queue of recipients, with individual message customization
- Save emails to be sent later
- Retrieve and manage emails from email mailboxes.

Installation

Install it directly into your project:

```
composer require popphp/pop-mail
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-mail": "3.0.*",
    }
}
```

A Note about SMTP

The SMTP transport component within *pop-mail* is forked from and built on top of the SMTP features and functionality of the Swift Mailer Library and the great work the Swift Mailer team has accomplished over the past years.

Basic Use

Here's an example sending a basic email using `sendmail`:

```
use Pop\Mail;

$transport = new Mail\Transport\Sendmail()

$mailer = new Mail\Mailer($transport);

$message = new Mail\Message('Hello World');
$message->setTo('you@domain.com');
$message->setFrom('me@domain.com');
$message->attachFile(__DIR__ . '/image.jpg');
$message->setBody('Hello World! This is a test!');

$mailer->send($message);
```

Here's an example sending a basic email using SMTP (MS Exchange):

```
use Pop\Mail;

$transport = new Mail\Transport\Smtp('mail.msdomain.com', 587);
$transport->setUsername('me')
    ->setPassword('password');

$mailer = new Mail\Mailer($transport);

$message = new Mail\Message('Hello World');
$message->setTo('you@domain.com');
$message->setFrom('me@domain.com');
$message->attachFile(__DIR__ . '/image.jpg');
$message->setBody('Hello World! This is a test!');

$mailer->send($message);
```

Here's an example sending a basic email using SMTP (Gmail Exchange):

```
use Pop\Mail;

$transport = new Mail\Transport\Smtp('smtp.gmail.com', 587, 'tls');
$transport->setUsername('me@mydomain.com')
    ->setPassword('password');

$mailer = new Mail\Mailer($transport);

$message = new Mail\Message('Hello World');
$message->setTo('you@domain.com');
$message->setFrom('me@domain.com');
$message->attachFile(__DIR__ . '/image.jpg');
$message->setBody('Hello World! This is a test!');

$mailer->send($message);
```

Attaching a File

```
use Pop\Mail;

$mailer = new Mail\Mailer(new Mail\Transport\Sendmail());

$message = new Mail\Message('Hello World');
$message->setTo('you@domain.com');
$message->setFrom('me@domain.com');

$fileData = file_get_contents($fileData);

$message->attachFile($fileData, 'image/jpeg', 'myimage.jpg');
$message->setBody('Hello World! This is a test!');

$mailer->send($message);
```

Sending an HTML/Text Email

```
use Pop\Mail;
$mailer = new Mail\Mailer(new Mail\Transport\Sendmail());

$message = new Mail\Message('Hello World');
$message->setTo('you@domain.com');
$message->setFrom('me@domain.com');

$message->addText('Hello World! This is a test!');
$message->addHtml('<html><body><h1>Hello World!</h1><p>This is a test!</p></body></
↪html>');

$mailer->send($message);
```

Sending Emails to a Queue

```
use Pop\Mail;

$queue = new Queue();
$queue->addRecipient([
    'email'   => 'me@domain.com',
    'name'    => 'My Name',
    'company' => 'My Company',
    'url'     => 'http://www.domain1.com/'
]);
$queue->addRecipient([
    'email'   => 'another@domain.com',
    'name'    => 'Another Name',
    'company' => 'Another Company',
    'url'     => 'http://www.domain2.com/'
]);

$message = new Mail\Message('Hello [{name}]!');
```

```
$message->setFrom('noreply@domain.com');
$message->setBody(
<<<TEXT
How are you doing? Your [{company}] is great!
I checked it out at [{url}]
TEXT
);

$queue->addMessage($message);

$mailer = new Mail\Mailer(new Mail\Transport\Sendmail());
$mailer->sendFromQueue($queue);
```

Saving an Email to Send Later

```
use Pop\Mail;

$message = new Mail\Message('Hello World');
$message->setTo('you@domain.com');
$message->setFrom('me@domain.com');

$message->addText('Hello World! This is a test!');
$message->addHtml('<html><body><h1>Hello World!</h1><p>This is a test!</p></body></
↪html>');

$message->save(__DIR__ . '/mailqueue/test.msg');

$mailer = new Mail\Mailer(new Mail\Transport\Sendmail());
$mailer->sendFromDir(__DIR__ . '/mailqueue');
```

Retrieving Emails from a Client

```
use Pop\Mail\Client;

$imap = new Client\Imap('imap.gmail.com', 993);
$imap->setUsername('me@domain.com')
     ->setPassword('password');

$imap->setFolder('INBOX');
$imap->open('/ssl');

// Sorted by date, reverse order (newest first)
$ids     = $imap->getMessageIdsBy(SORTDATE, true);
$headers = $imap->getMessageHeadersById($ids[0]);
$parts   = $imap->getMessageParts($ids[0]);

// Assuming the first part is an image attachement, display image
header('Content-Type: image/jpeg');
header('Content-Length: ' . strlen($parts[0]->content));
echo $parts[0]->content;
```

pop-module

The `Pop\Module` sub-component is part of the core *popphp/popphp* component. It serves as the module manager to the main application object. With it, you can inject module objects into the application which can extend the functionality and features of your main application.

Installation

Install it directly into your project:

```
composer require popphp/popphp
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/popphp": "3.0.*",
    }
}
```

Basic Use

Modules can be thought of as "mini-application objects" that allow you to extend the functionality of your application. Module objects accept similar configuration parameters as an application object, such as `routes`, `services` and `events`. Additionally, it accepts a `prefix` configuration value as well to allow the module to register itself with the application autoloader. Here's an example of what a module might look like and how you'd register it with an application:

```
$application = new Pop\Application();

$moduleConfig = [
    'name'   => 'myModule',
    'routes' => [
        '/' => [
            'controller' => 'MyModule\Controller\IndexController',
            'action'     => 'index'
        ]
    ],
    'prefix' => 'MyModule\\'
];

$application->register($moduleConfig);
```

In the above example, the module configuration is passed into the application object. From there, an instance of the base module object is created and the configuration is passed into it. The newly created module object is then registered with the module manager within the application object.

Custom Modules

You can pass your own custom module objects into the application as well, as long as they implement the module interface provided. As the example below shows, you can create a new instance of your custom module and pass that into the application, instead of just the configuration. The benefit of doing this is to allow you to extend the base

module class and methods and provide any additional functionality that may be needed. In doing it this way, however, you will have to register your module's namespace prefix with the application's autoloader prior to registering the module with the application so that the application can properly detect and load the module's source files.

```
$application->autoloader->addPsr4('MyModule\\', __DIR__ . '/modules/mymodule/src');

$myModule = new MyModule\Module([
    'name'   => 'myModule',
    'routes' => [
        '/' => [
            'controller' => 'MyModule\Controller\IndexController',
            'action'     => 'index'
        ]
    ]
]);

$application->register($myModule);
```

The Module Manager

The module manager serves as the collection of module objects within the application. This facilitates accessing the modules you've added to the application during its life-cycle. In the examples above, the modules are not only being configured and created themselves, but they are also being registered with the application object. This means that at anytime, you can retrieve a module object or its properties in a number of ways:

```
$fooModule = $application->module('fooModule');

$barModule = $application->modules['barModule'];
```

You can also check to see if a module has been registered with the application object:

```
if ($application->isRegistered('fooModule')) {
    // Do something with the 'fooModule'
}
```

pop-nav

The *popphp/pop-nav* component provides an API for managing the creation of web-based navigation objects. It also supports an ACL object from the *popphp/pop-acl* component to enforce access rights within the navigation object.

Installation

Install it directly into your project:

```
composer require popphp/pop-nav
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-nav": "3.0.*",
    }
}
```

Basic Use

First, you can define the navigation tree:

```
$tree = [
    [
        'name'     => 'First Nav Item',
        'href'     => '/first-page',
        'children' => [
            [
                'name' => 'First Child',
                'href' => 'first-child'
            ],
            [
                'name' => 'Second Child',
                'href' => 'second-child'
            ]
        ]
    ],
    [
        'name' => 'Second Nav Item',
        'href' => '/second-page'
    ]
];
```

Then, you have a significant amount of control over the branch nodes and attributes via a configuration array:

```
$config = [
    'top' => [
        'node'  => 'nav',
        'id'    => 'main-nav'
    ],
    'parent' => [
        'node'  => 'nav',
        'id'    => 'nav',
        'class' => 'level'
    ],
    'child' => [
        'node'  => 'nav',
        'id'    => 'menu',
        'class' => 'item'
    ],
    'on'  => 'link-on',
    'off' => 'link-off',
    'indent' => '    '
];
```

You can then create and render your nav object:

```
use Pop\Nav\Nav;

$nav = new Nav($tree, $config);
echo $nav;
```

```
<nav id="main-nav">
    <nav id="menu-1" class="item-1">
        <a href="/first-page" class="link-off">First Nav Item</a>
        <nav id="nav-2" class="level-2">
            <nav id="menu-2" class="item-2">
                <a href="/first-page/first-child" class="link-off">First Child</a>
            </nav>
            <nav id="menu-3" class="item-2">
                <a href="/first-page/second-child" class="link-off">Second Child</a>
            </nav>
        </nav>
    </nav>
    <nav id="menu-4" class="item-1">
        <a href="/second-page" class="link-off">Second Nav Item</a>
    </nav>
</nav>
```

Advanced Use

First, let's set up the ACL object with some roles and resources:

```
use Pop\Acl\Acl;
use Pop\Acl\AclRole as Role;
use Pop\Acl\AclResource as Resource;

$acl = new Acl();

$admin  = new Role('admin');
$editor = new Role('editor');

$acl->addRoles([$admin, $editor]);

$acl->addResource(new Resource('second-child'));
$acl->allow('admin');
$acl->deny('editor', 'second-child');
```

And then we add the ACL rules to the navigation tree:

```
$tree = [
    [
        'name'     => 'First Nav Item',
        'href'     => '/first-page',
        'children' => [
            [
                'name' => 'First Child',
                'href' => 'first-child'
            ],
            [
                'name' => 'Second Child',
                'href' => 'second-child',
                'acl'  => [
                    'resource' => 'second-child'
                ]
            ]
        ]
    ],
```

```
    [
        'name' => 'Second Nav Item',
        'href' => '/second-page'
    ]
];
```

We then inject the ACL object into the navigation object, set the current role and render the navigation:

```
$nav = new Nav($tree, $config);
$nav->setAcl($acl);
$nav->setRole($editor);
echo $nav;
```

```
<nav id="main-nav">
    <nav id="menu-1" class="item-1">
        <a href="/first-page" class="link-off">First Nav Item</a>
        <nav id="nav-2" class="level-2">
            <nav id="menu-2" class="item-2">
                <a href="/first-page/first-child" class="link-off">First Child</a>
            </nav>
        </nav>
    </nav>
    <nav id="menu-3" class="item-1">
        <a href="/second-page" class="link-off">Second Nav Item</a>
    </nav>
</nav>
```

Because the 'editor' role is denied access to the 'second-child' page, that nav branch is not rendered.

pop-paginator

The *popphp/pop-paginator* component is a simple component that provides pagination in a few different forms.

Installation

Install it directly into your project:

```
composer require popphp/pop-paginator
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-paginator": "3.0.*",
    }
}
```

Basic Use

Basic example of a list:

```
use Pop\Paginator\Range;

$paginator = new Range(42);
echo $paginator;
```

Which will produce this HTML:

```
<span>1</span>
<a href="/?page=2">2</a>
<a href="/?page=3">3</a>
<a href="/?page=4">4</a>
<a href="/?page=5">5</a>
```

And if you clicked on page 3, it would render:

```
<a href="/?page=1">1</a>
<a href="/?page=2">2</a>
<span>3</span>
<a href="/?page=4">4</a>
<a href="/?page=5">5</a>
```

Using Bookends for a Large Range

```
use Pop\Paginator\Range;

$paginator = new Range(4512);
echo $paginator;
```

If we go to page 12, it would render:

```
<a href="/?page=1">&laquo;</a>
<a href="/?page=10">&lsaquo;</a>
<a href="/?page=11">11</a>
<span>12</span>
<a href="/?page=13">13</a>
<a href="/?page=14">14</a>
<a href="/?page=15">15</a>
<a href="/?page=16">16</a>
<a href="/?page=17">17</a>
<a href="/?page=18">18</a>
<a href="/?page=19">19</a>
<a href="/?page=20">20</a>
<a href="/?page=21">&rsaquo;</a>
<a href="/?page=452">&raquo;</a>
```

As you can see, it renders the "bookends" to navigate to the next set of pages, the previous set, the beginning of the set or the end.

Using an Form Input Field

To have a cleaner way of displaying a large set of pages, you can use an input field within a form like this:

```
use Pop\Paginator\Form;
$paginator = new Form(558);
echo $paginator;
```

This will produce:

```
<a href="/?page=1">&laquo;</a>
<a href="/?page=13">&lsaquo;</a>
<form action="/" method="get">
    <div><input type="text" name="page" size="2" value="14" /> of 56</div>
</form>
<a href="/?page=15">&rsaquo;</a>
<a href="/?page=56">&raquo;</a>
```

So instead of a set a links in between the bookends, there is a form input field that will allow the user to input a specific page to jump to.

Other Options

You can set many options to tailor the paginator's look and functionality:

- Number of items per page
- Range of the page sets
- Separator between the page links
- Classes for the on/off page links
- Bookend characters (start, previous, next, end)

pop-pdf

The *popphp/pop-pdf* component provides a robust API for the creation and management of PDF documents. Many features of the PDF format are supported, including embedding images and fonts, as well as importing other PDFs.

Installation

Install it directly into your project:

```
composer require popphp/pop-pdf
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-pdf": "3.0.*",
    }
}
```

Basic Use

PDF generation in an application is typically a required feature for any application that does any type of in-depth reporting or data exporting. Many applications may require this exported data to be in a concise, well-formatted and portable document and PDF provides this.

The PDF specification, as well as its shared assets' specifications, such as fonts and images, are an extremely large and complex set of rules and syntax. This component attempts to harness the power and features defined by those specifications and present an intuitive API that puts the power of PDF at your fingertips.

Building a PDF

At the center of the *popphp/pop-pdf* component is the main `Pop\Pdf\Pdf` class. It serves as a manager or controller of sorts for all of the various PDF assets that will pass through during the process of PDF generation. The different assets are each outlined with their own section below.

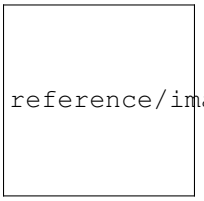
Here's a simple example building and generating a PDF document with some text. The finer points of what's happening will be explained more in depth in the later sections.

```
use Pop\Pdf\Pdf;
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;
use Pop\Pdf\Document\Page;

// Create a page and add the text to it
$page = new Page(Page::LETTER);
$page->addText(new Page\Text('Hello World!', 24), Font::ARIAL, 50, 650);

// Create a document, add the font to it and then the page
$document = new Document();
$document->addFont(new Font(Font::ARIAL));
$document->addPage($page);

// Pass the document to the Pdf object to build it and output it to HTTP
$pdf = new Pdf();
$pdf->outputToHttp($document);
```



Importing a PDF

Importing an existing PDF or pages from one may be required in your application. Using the main PDF object, you can specify the pdf to import as well as the select pages you may wish to import. From there you can either select pages to modify or add new pages to the document. When you do import an existing PDF, the method will return a parsed and working document object. In the example below, we will import pages 2 and 5 from the existing PDF, add a new page in between them and then save the new document:

```
use Pop\Pdf\Pdf;
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;
use Pop\Pdf\Document\Page;

$pdf = new Pdf();
$document = $pdf->importFromFile('doc.pdf', [2, 5])

// Create a page and add the text to it
```

```
$page = new Page(Page::LETTER);
$page->addText(new Page\Text('Hello World!', 24), Font::ARIAL, 50, 650);

// Create a document, add the font to it and then the page
$document = new Document();
$document->addFont(new Font(Font::ARIAL));
$document->addPage($page);
$document->orderPages([1, 3, 2]); // 3 being our new page.

// Pass the document to the Pdf object to build it and write it to a new file
$pdf = new Pdf();
$pdf->writeToFile('new-doc.pdf');
```

When the 2 pages are imported in, they default to page 1 and 2, respectively. Then we can add any pages we need from there and control the final order of the pages with the `orderPages` method like in the above example.

If you wish to import the whole PDF and all of its pages, simply leave the `$pages` parameter blank.

Coordinates

It should be noted that the PDF coordinate system has its origin (0, 0) at the bottom left. In the example above, the text was placed at the (x, y) coordinate of (50, 650). When placed on a page that is set to the size of a letter, which is 612 points x 792 points, that will make the text appear in the top left. It the coordinates of the text were set to (50, 50) instead, the text would have appeared in the bottom left.

As this coordinate system may or may not suit a developer's personal preference or the requirements of the application, the origin point of the document can be set using the following method:

```
use Pop\Pdf\Document;

$document = new Document();
$document->setOrigin(Document::ORIGIN_TOP_LEFT);
```

Now, with the document's origin set to the top left, when you place assets into the document, you can base it off of the new origin point. So for the text in the above example to be placed in the same place, the new (x, y) coordinates would be (50, 142).

Alternatively, the full list of constants in the `Pop\Pdf\Document` class that represent the different origins are:

- ORIGIN_TOP_LEFT
- ORIGIN_TOP_RIGHT
- ORIGIN_BOTTOM_LEFT
- ORIGIN_BOTTOM_RIGHT
- ORIGIN_CENTER

Documents

A document object represents the top-level "container" object of the the PDF document. As you create the various assets that are to be placed in the PDF document, you will inject them into the document object. At the document level, the main assets that can be added are **fonts**, **forms**, **metadata** and **pages**. The font and form objects are added at the document level as they can be re-used on the page level by other assets. The metadata object contains informational data about the document, such as title and author. And the page objects contain all of the page-level assets, as detailed below.

Fonts

Font objects are the global document objects that contain information about the fonts that can be used by the text objects within the pages of the document. A font can either be one of the standard fonts supported by PDF natively, or an embedded font from a font file.

Standard Fonts

The set of standard, native PDF fonts include:

- Arial
- Arial,Italic
- Arial,Bold
- Arial,BoldItalic
- Courier
- Courier-Oblique
- Courier-Bold
- Courier-BoldOblique
- CourierNew
- CourierNew,Italic
- CourierNew,Bold
- CourierNew,BoldItalic
- Helvetica
- Helvetica-Oblique
- Helvetica-Bold
- Helvetica-BoldOblique
- Symbol
- Times-Roman
- Times-Bold
- Times-Italic
- Times-BoldItalic
- TimesNewRoman
- TimesNewRoman,Italic
- TimesNewRoman,Bold
- TimesNewRoman,BoldItalic
- ZapfDingbats

When adding a standard font to the document, you can add it and then reference it by name throughout the building of the PDF. For reference, there are constants available in the `Pop\Pdf\Document\Font` class that have the correct standard font names stored in them as strings.

```
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;

$font = new Font(Font::TIMES_NEW_ROMAN_BOLDITALIC);

$document = new Document();
$document->addFont($font);
```

Now, the font defined as "TimesNewRoman,BoldItalic" is available to the document and for any text for which you need it.

Embedded Fonts

The embedded font types that are supported are:

- TrueType
- OpenType
- Type1

When embedding an external font, you will need access to its name to correctly reference it by string much in the same way you do for a standard font. That name becomes accessible once you create a font object with an embedded font and it is successfully parsed.

Notice about embedded fonts

There may be issues embedding a font if certain font data or font files are missing, incomplete or corrupted. Furthermore, there may be issues embedding a font if the correct permissions or licensing are not provided.

```
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;
use Pop\Pdf\Document\Page;

$customFont = new Font('custom-font.ttf');

$document = new Document();
$document->embedFont($customFont);

$text = new Page\Text('Hello World!', 24);

$page = new Page(Page::LETTER);
$page->addText($text, $customFont->getName(), 50, 650);
```

The above example will attach the name and reference of the embedded custom font to that text object. Additionally, when a font is added or embedded into a document, its name becomes the current font, which is a property you can access like this:

```
$page->addText($text, $document->getCurrentFont(), 50, 650);
```

If you'd like to override or switch the current document font back to another font that's available, you can do so like this:

```
$document->setCurrentFont('Arial');
```

Forms

Form objects are the global document objects that contain information about fields that are to be used within a Form object on a page in the document. By themselves they are fairly simple to use and inject into a document object. From

there, you would add fields to a their respective pages, while attaching them to a form object.

The example below demonstrates how to add a form object to a document:

```
use Pop\Pdf\Document;
use Pop\Pdf\Document\Form;

$form = new Form('contact_form');

$document = new Document();
$document->addForm($form);
```

Then, when you add a field to a page, you can reference the form to attach it to:

```
use Pop\Pdf\Document\Page;

$name = new Page\Field\Text('name');
$name->setWidth(200)
     ->setHeight(40);

$page = new Page(Page::LETTER);
$page->addField($name, 'contact_form', 50, 650);
```

The above example creates a name field for the contact form, giving it a width and height and placing it at the (50, 650) coordinated. *Fields* will be covered more in depth below.

Metadata

The metadata object contains the document identifier data such as title, author and date. This is the data that is commonly displayed in the the document title bar and info boxes of a PDF reader. If you'd like to set the metadata of the document, you can with the following API:

```
use Pop\Pdf\Document;

$metadata = new Document\Metadata();
$metadata->setTitle('My Document')
    ->setAuthor('Some Author')
    ->setSubject('Some Subject')
    ->setCreator('Some Creator')
    ->setProducer('Some Producer')
    ->setCreationDate('August 19, 2015')
    ->setModDate('August 22, 2015')

$document = new Document();
$document->setMetadata($metadata);
```

And there are getter methods that follow the same naming convention to retrieve the data from the metadata object.

Pages

Page objects contain the majority of the assets that you would expect to be on a page within a PDF document. A page's size can be either custom-defined or one of the predefined sizes. There are constants that define those predefine sizes for reference:

- ENVELOPE_10 (297 x 684)
- ENVELOPE_C5 (461 x 648)

- ENVELOPE_DL (312 x 624)
- FOLIO (595 x 935)
- EXECUTIVE (522 x 756)
- LETTER (612 x 792)
- LEGAL (612 x 1008)
- LEDGER (1224 x 792)
- TABLOID (792 x 1224)
- A0 (2384 x 3370)
- A1 (1684 x 2384)
- A2 (1191 x 1684)
- A3 (842 x 1191)
- A4 (595 x 842)
- A5 (420 x 595)
- A6 (297 x 420)
- A7 (210 x 297)
- A8 (148 x 210)
- A9 (105 x 148)
- B0 (2920 x 4127)
- B1 (2064 x 2920)
- B2 (1460 x 2064)
- B3 (1032 x 1460)
- B4 (729 x 1032)
- B5 (516 x 729)
- B6 (363 x 516)
- B7 (258 x 363)
- B8 (181 x 258)
- B9 (127 x 181)
- B10 (91 x 127)

```
use Pop\Pdf\Document\Page;

$legal  = new Page(Page::LEGAL);
$custom = new Page(640, 480);
```

The above example creates two pages - one legal size and one a custom size of 640 x 480.

Images

An image object allows you to place an image onto a page in the PDF document, as well as control certain aspects of that image, such as size and resolution. The image types that are supported are:

- JPG (RGB, CMYK or Grayscale)
- PNG (8-Bit Index)
- PNG (8-Bit Index w/ Transparency)
- PNG (24-Bit RGB or Grayscale)
- GIF (8-Bit Index)
- GIF (8-Bit Index w/ Transparency)

Here is an example of embedding a large image and resizing it down before placing on the page:

```
use Pop\Pdf\Document\Page;

$image = Page\Image::createImageFromFile('/path/to/some/image.jpg')
$image->resizeToWidth(320);

$page = new Page(Page::LETTER);
$page->addImage($image, 50, 650);
```

In the above example, the large image is processed (down-sampled) and resized to a width of 320 pixels and placed into the page at the coordinates of (50, 650).

If you wanted to preserve the image's high resolution, but fit it into the smaller dimensions, you can do that by setting the `$preserveResolution` flag in the resize method.

```
$image->resizeToWidth(320, true);
```

This way, the high resolution image is not processed or down-sampled and keeps its high quality. It is only placed into scaled down dimensions.

Color

With path and text objects, you will need to set colors to render them correctly. The main 3 colorspaces that are supported are RGB, CMYK and Grayscale. Each color space object is created by instantiating it and passing the color values:

```
use Pop\Pdf\Document\Page\Color;

$red  = new Color\Rgb(255, 0, 0);      // $r, $g, $b (0 - 255)
$cyan = new Color\Cmyk(100, 0, 0, 0); // $c, $m, $y, $k (0 - 100)
$gray = new Color\Gray(50);           // $gray (0 - 100)
```

These objects are then passed into the methods that consume them, like `setFillColor` and `setStrokeColor` within the path and text objects.

Paths

Since vector graphics are at the core of PDF, the path class contains a robust API that allows you to no only draw various paths and shapes, but also set their colors and styles. On instantiation, you can set the style of the path object:

```
use Pop\Pdf\Document\Page\Path;
use Pop\Pdf\Document\Page\Color\Rgb;

$path = new Path(Path::FILL_STROKE);
$path->setFillColor(new Rgb(255, 0, 0))
     ->setStrokeColor(new Rgb(0, 0, 0))
     ->setStroke(2);
```

The above example created a path object with the default style of fill and stroke, and set the fill color to red, the stroke color to black and the stroke width to 2 points. That means that any paths that are drawn from here on out will have those styles until they are changed. You can create and draw more than one path or shape with in path object. The path class has constants that reference the different style types you can set:

- STROKE
- STROKE_CLOSE
- FILL
- FILL_EVEN_ODD
- FILL_STROKE
- FILL_STROKE_EVEN_ODD
- FILL_STROKE_CLOSE
- FILL_STROKE_CLOSE_EVEN_ODD
- CLIPPING
- CLIPPING_FILL
- CLIPPING_NO_STYLE
- CLIPPING_EVEN_ODD
- CLIPPING_EVEN_ODD_FILL
- CLIPPING_EVEN_ODD_NO_STYLE
- NO_STYLE

From there, this is the core API that is available:

- `$path->setStyle($style);`
- `$path->setFillColor(Color\ColorInterface $color);`
- `$path->setStrokeColor(Color\ColorInterface $color);`
- `$path->setStroke($width, $dashLength = null, $dashGap = null);`
- `$path->openLayer();`
- `$path->closeLayer();`
- `$path->drawLine($x1, $y1, $x2, $y2);`
- `$path->drawRectangle($x, $y, $w, $h = null);`
- `$path->drawRoundedRectangle($x, $y, $w, $h = null, $rx = 10, $ry = null);`
- `$path->drawSquare($x, $y, $w);`
- `$path->drawRoundedSquare($x, $y, $w, $rx = 10, $ry = null);`
- `$path->drawPolygon($points);`

- `$path->drawEllipse($x, $y, $w, $h = null);`
- `$path->drawCircle($x, $y, $w);`
- `$path->drawArc($x, $y, $start, $end, $w, $h = null);`
- `$path->drawChord($x, $y, $start, $end, $w, $h = null);`
- `$path->drawPie($x, $y, $start, $end, $w, $h = null);`
- `$path->drawOpenCubicBezierCurve($x1, $y1, $x2, $y2, $bezierX1, $bezierY1, $bezierX2, $bezierY2);`
- `$path->drawClosedCubicBezierCurve($x1, $y1, $x2, $y2, $bezierX1, $bezierY1, $bezierX2, $bezierY2);`
- `$path->drawOpenQuadraticBezierCurve($x1, $y1, $x2, $y2, $bezierX, $bezierY, $first = true);`
- `$path->drawClosedQuadraticBezierCurve($x1, $y1, $x2, $y2, $bezierX, $bezierY, $first = true);`

Extending the original code example above, here is an example of drawing a rectangle and placing it on a page:

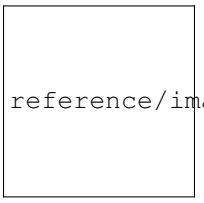
```
use Pop\Pdf\Pdf;
use Pop\Pdf\Document;
use Pop\Pdf\Document\Page;
use Pop\Pdf\Document\Page\Path;
use Pop\Pdf\Document\Page\Color\Rgb;

// Create a path object, set the styles and draw a rectangle
$path = new Path(Path::FILL_STROKE);
$path->setFillColor(new Rgb(255, 0, 0))
     ->setStrokeColor(new Rgb(0, 0, 0))
     ->setStroke(2)
     ->drawRectangle(100, 600, 200, 100);

// Create a page and add the path to it
$page = new Page(Page::LETTER);
$page->addPath($path);

// Create a document and add the page
$document = new Document();
$document->addPage($page);

// Pass the document to the Pdf object to build it and output it to HTTP
$pdf = new Pdf();
$pdf->outputToHttp($document);
```



Layers

As the API shows, you can also layer paths using the `openLayer()` and `closeLayer()` methods which open and close an independent graphics state. Any paths added while in this state will render onto that "layer." Any paths rendered after the state is closed will render above that layer.

Clipping Paths

The path object also supports clipping paths via setting the path style to a clipping style. In doing so, the path will render as a clipping path or "mask" over any paths before it.

Text

With text objects, you can control a number of parameters that affect how the text is displayed beyond which font is used and the size. As with path objects, you can set color and style, as well as a few other parameters. As one of the above examples demonstrated, you can create a text object like this:

```
use Pop\Pdf\Document\Page;

$text = new Page\Text('Hello World!', 24);

// Create a page and add the text to it
$page = new Page(Page::LETTER);
$page->addText($text, 'Arial', 50, 650);
```

The above code create a text object with the font size of 24 points and added it to a page using the Arial font. From there, you can do more with the text object API. Here is what the API looks like for a text object:

- `$text->setFillColor(Color\ColorInterface $color);`
- `$text->setStrokeColor(Color\ColorInterface $color);`
- `$text->setStroke($width, $dashLength = null, $dashGap = null);`
- `$test->setWrap($wrap, $lineHeight = null);`
- `$test->setLineHeight($lineHeight);`
- `$test->setRotation($rotation);`
- `$test->setTextParams($c = 0, $w = 0, $h = 100, $v = 100, $rot = 0, $rend = 0);`

With the `setTextParams()` method, you can set the following render parameters:

- `$c` - character spacing
- `$w` - word spacing
- `$h` - horizontal stretch
- `$v` - vertical stretch
- `$rot` - rotation in degrees
- `$rend` - render mode 0 - 7;
  - 0 - Fill
  - 1 - Stroke
  - 2 - Fill and stroke
  - 3 - Invisible
  - 4 - Fill then use for clipping
  - 5 - Stroke the use for clipping
  - 6 - Fill and stroke and use for clipping

- 7 - Use for clipping

Extending the example above, we can render red text to the page like this:

```
use Pop\Pdf\Pdf;
use Pop\Pdf\Document;
use Pop\Pdf\Document\Font;
use Pop\Pdf\Document\Page;

// Create the text object and set the fill color
$text = new Page\Text('Hello World!', 24);
$text->setFillColor(new Rgb(255, 0, 0));

// Create a page and add the text to it
$page = new Page(Page::LETTER);
$page->addText($text, Font::ARIAL, 50, 650);

// Create a document, add the font to it and then the page
$document = new Document();
$document->addFont(new Font(Font::ARIAL));
$document->addPage($page);

// Pass the document to the Pdf object to build it and output it to HTTP
$pdf = new Pdf();
$pdf->outputToHttp($document);
```

Wrap and Line-height

The `setWrap` and `setLineHeight()` methods help facilitate larger blocks of text that you might add to the PDF page. By setting values with these two methods, you give the PDF page the parameters needed to calculate wrapping the large body of text with the proper line-height for you, instead of you having to break the text up and place it manually.

Annotations

Annotation objects give you the functionality to add internal document links and external web links to the page. At the base of an annotation object, you would set the width and height of the annotation's click area or "hot spot." For an internal annotation, you would pass in a set of target coordinates as well:

```
use Pop\Pdf\Document\Page\Annotation;

$link = new Annotation\Link(200, 25, 50, 650); // $width, $height, $xTarget, $yTarget
```

In the above example, an internal annotation object that is 200 x 25 in width and height has been created and is linked to the coordinates of (50, 650) on the current page. If you'd like to target coordinates on a different page, you can set that as well:

```
$link->setPageTarget(3);
```

And if you would like to zoom in on the target, you can set the Z target as well:

```
$link->setZTarget(2);
```

For external URL annotations, instead of an internal set of coordinates, you would pass the URL into the constructor:

```
use Pop\Pdf\Document\Page\Annotation;

$link = new Annotation\Url(200, 25, 'http://www.mywebsite.com/');
```

The above example will create an external annotation link that, when clicked, will link out to the URL given.

Fields

As mentioned earlier, field objects are the entities that collect user input and attach that data to form objects. The benefit of this is the ability to save user input within the document. The field types that are supported are:

- Text (single and multi-line)
- Choice
- Button

Here is an example creating a simple set of fields and attaching them to a form object:

```
use Pop\Pdf\Document;
use Pop\Pdf\Document\Form;
use Pop\Pdf\Document\Page;

// Create the form object and inject it into the document object
$form = new Form('contact_form');

$document = new Document();
$document->addForm($form);

$name = new Page\Field\Text('name');
$name->setWidth(200)
     ->setHeight(40);

$colors = new Page\Field\Choice('colors');
$colors->addOption('Red')
    ->addOption('Green')
    ->addOption('Blue')

$comments = new Page\Field\Text('comments');
$comments->setWidth(200)
     ->setHeight(100)
     ->setMultiline();

$page = new Page(Page::LETTER);
$page->addField($name, 'contact_form', 50, 650)
     ->addField($colors, 'contact_form', 50, 600)
     ->addField($comments, 'contact_form', 50, 550);
```

In the above example, the fields are created, attached to the form object and added to the page object.

pop-router

The `Pop\Router` sub-component is part of the core *popphp/popphp* component. It serves as the primary router to the main application object. With it, you can define routes for both web-based and console-based applications.

Installation

Install it directly into your project:

```
composer require popphp/popphp
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/popphp": "3.0.*",
    }
}
```

Basic Use

The router object facilitates the configuration and matching of the routes to access your application. It supports both HTTP and CLI routing. With it, you can establish valid routes along with any parameters that may be required with them.

HTTP Route Example

```
$router->addRoute('/hello', function() {
    echo 'Hello World';
});
```

In the above example, a web request of `http://localhost/hello` will execute the closure as the controller and echo `Hello World` out to the browser.

CLI Route Example

```
$router->addRoute('hello', function($name) {
    echo 'Hello World';
});
```

In the above example, a CLI command of `./app hello` will execute the closure as the controller and echo `Hello World` out to the console.

A controller object can be, and usually is, an instance of a class instead of a closure for more control over what happens for each route:

```
class MyApp\Controller\IndexController extends \Pop\Controller\AbstractController
{
    public function index()
    {
        echo 'Hello World!';
    }
}

$router->addRoute('/', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'index'
]);
```

In the above example, the request `/` is routed to the `index()` method in the defined controller class.

Controller Parameters

It's common to require access to various elements and values of your application while within an instance of your controller class. To provide this, the router object allows you to inject parameters into the controller upon instantiation.

Let's assume your controller's constructor looks like this:

```
class MyApp\Controller\IndexController extends \Pop\Controller\AbstractController
{
    protected $foo;
    protected $bar;

    public function __construct($foo, $bar)
    {
        $this->foo = $foo;
        $this->bar = $bar;
    }
}
```

You could then inject parameters into the controller's constructor like this:

```
$router->addControllerParams(
    'MyApp\Controller\IndexController', [
        'foo' => $foo,
        'bar' => $bar
    ]
);
```

If you require parameters to be injected globally to all of your controller classes, then you can replace the controller name `'MyApp\Controller\IndexController` with `*` and they will be injected into all controllers. You can also define controller parameters within the route configuration as well.

```
$config = [
    'routes' => [
        '/products' => [
            'controller'       => 'MyApp\Controller\ProductsController',
            'action'           => 'index',
            'controllerParams' => [
                'baz'    => 789
            ]
        ]
    ]
];

$app = new Pop\Application($config);
```

Dispatch Parameters

Defining route dispatch parameters, you can define required (or optional) parameters that are needed for a particular route:

```
$router->addRoute('/hello/:name', function($name) {
    echo 'Hello ' . ucfirst($name);
});
```

```
$router->addRoute('hello <name>', function($name) {
    echo 'Hello ' . ucfirst($name);
});
```

The HTTP request of `http://localhost/hello/pop` and the CLI command of `./app hello pop` will each echo out `Hello Pop` to the browser and console, respectively.

Optional Dispatch Parameters

Consider the following controller class and method:

```
class MyApp\Controller\IndexController extends \Pop\Controller\AbstractController
{
    public function hello($name = null)
    {
        if (null === $name) {
            echo 'Hello World!';
        } else {
            echo 'Hello ' . ucfirst($name);
        }
    }
}
```

Then add the following routes for HTTP and CLI:

HTTP:

```
$router->addRoute('/hello[/:name]', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'hello'
]);
```

CLI:

```
$router->addRoute('hello [<name>]', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'hello'
]);
```

In the above example, the parameter `$name` is an optional dispatch parameter and the `hello()` method performs differently depending on whether or not the parameter value it present.

Dynamic Routing

Dynamic routing is also supported. You can define routes as outlined in the examples below and they will be dynamically mapped and routed to the correct controller and method. Let's assume your application has the following controller class:

```
class MyApp\Controller\UsersController extends \Pop\Controller\AbstractController
{

    public function index()
    {
        // Show a list of users
    }

    public function edit($id = null)
    {
        // Edit the user with the ID# of $id
    }
}
```

You could define a dynamic route for HTTP like this:

```
$router->addRoute('/:controller/:action[/:param]', [
    'prefix' => 'MyApp\Controller\\'
]);
```

and routes such as these would be valid:

- `http://localhost/users`
- `http://localhost/users/edit/1001`

For CLI, you can define a dynamic route like this:

```
$router->addRoute('<controller> <action> [<param>]', [
    'prefix' => 'MyApp\Controller\\'
]);
```

and routes such as these would be valid:

- `./app users`
- `./app users edit 1001`

Routing Syntax

The tables below outline the accepted routing syntax for the route matching:

HTTP

Web Route	What's Expected
/foo/:bar/:baz	The 2 params are required
/foo/:bar[/:baz]	First param required, last one is optional
/foo/:bar/:baz*	One required param, one required param that is a collection (array)
/foo/:bar[/:baz*]	One required param, one optional param that is a collection (array)

CLI

CLI Route	What's Expected
foo bar	Two commands are required
foo bar|baz	Two commands are required, the 2nd can accept 2 values
foo [bar|baz]	The second command is optional and can accept 2 values
foo -o1 [-o2]	First option required, 2nd option is optional
foo –option1|-o1 [–option2|-o2]	1st option required, 2nd optional; long & short supported for both
foo <name> [<email>]	First param required, 2nd param optional
foo –name= [–email=]	First value param required, 2nd value param optional

pop-service

The `Pop\Service` sub-component is part of the core *popphp/popphp* component. It serves as the service locator for the main application object. With it, you can wire up services that may be needed during the life-cycle of the application and call them when necessary.

Installation

Install it directly into your project:

```
composer require popphp/popphp
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/popphp": "3.0.*",
    }
}
```

Basic Use

If you need access to various services throughout the life-cycle of the application, you can register them with the service locator and recall them later. You can pass an array of services into the constructor, or you can set them individually as needed.

```
$services = new Pop\Service\Locator([
    'foo' => 'MyApp\SomeService'
]);

$services->set('bar', 'MyApp\SomeService->bar');
$services['baz'] = 'MyApp\SomeService->baz';
```

Then, you can retrieve a service in a number of ways:

```
$foo = $services['foo'];
$bar = $services->get('bar');
```

You can use the `isAvailable` method if you'd like to determine if a service is available, but not loaded yet:

```
if ($services->isAvailable('foo')) {
    $foo = $services['foo'];
} else {
    $services->set('foo', 'MyApp\SomeService');
}
```

The `isLoaded` method determines if the service has been set and previously called:

```
if ($services->isLoaded('foo')) {
    $foo = $services['foo'];
}
```

The service locator uses "lazy-loading" to store the service names and their attributes, and doesn't load or create the services until they are actually needed and called from the service locator.

You can also remove a service from the service locator if needed:

```
$services->remove('foo');
unset($services['bar']);
```

Syntax & Parameters

You have a couple of different options when setting services. You can pass callable strings or already instantiated instances of objects, although the latter could be potentially less efficient. Also, if needed, you can define parameters that will be passed into the service being called.

Syntax

Valid callable service strings are as follows:

1. 'SomeClass'
2. 'SomeClass->foo'
3. 'SomeClass::bar'

The first callable string example creates a new instance of `SomeClass` and returns it. The second callable string example creates a new instance of `SomeClass`, calls the method `foo()` and returns the value from it. The third callable string example calls the static method `bar()` in the class `SomeClass` and returns the value from it.

Parameters

Additionally, if you need to inject parameters into your service upon calling your service, you can set a service using an array with a `call` key and a `params` key like this:

```
$services = new Pop\Service\Locator([
    'foo' => [
        'call'   => 'MyApp\SomeService->foo',
        'params' => [
            'bar' => 123,
            'baz' => 456
        ]
    ]
]);
```

In the example above, the service `foo` is defined by the callable string `MyApp\SomeService->foo`. When the service `foo` is retrieved, the locator will create a new instance of `MyApp\SomeService`, call the method `foo` while passing the params `bar` and `baz` into the method and returning that value from that method.

Service Container

A service container class is available if you prefer to track and access your services through it. The first call to create a new service locator object will automatically register it as the 'default' service locator.

```
$services = new Pop\Service\Locator([
    'foo' => 'MyApp\SomeService'
]);
```

At some later point in your application:

```
$services = Pop\Service\Container::get('default');
```

If you would like register additional custom service locator objects, you can do that like so:

```
Pop\Service\Container::set('customServices', $myCustomServiceLocator);
```

And then later in your application:

```
if (Pop\Service\Container::has('customServices')) {
    $myCustomServiceLocator = Pop\Service\Container::get('customServices');
}
```

pop-session

The *popphp/pop-session* component provides the functionality to manage sessions.

Installation

Install it directly into your project:

```
composer require popphp/pop-session
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-session": "3.1.*",
    }
}
```

Basic Use

The session component gives you multiple ways to interact with the `$_SESSION` variable and store and retrieve data to it. The following are supported:

- Managing basic sessions and session values
- Creating namespaced sessions
- Setting session value expirations
- Setting request-based session values

Basic Sessions

```
$sess = Pop\Session\Session::getInstance();
$sess->user_id    = 1001;
$sess['username'] = 'admin';
```

The above snippet saves values to the user's session. To recall it later, you can access the session like this:

```
$sess = Pop\Session\Session::getInstance();
echo $sess->user_id;    // echos out 1001
echo $sess['username']; // echos out 'admin'
```

And to destroy the session and its values, you can call the `kill()` method:

```
$sess = Pop\Session\Session::getInstance();
$sess->kill();
```

Namespaced Sessions

Namespaced sessions allow you to store session under a namespace to protect and preserve that data away from the normal session data.

```
$sessFoo = new Pop\Session\SessionNamespace('foo');
$sessFoo->bar = 'baz'
```

What's happening "under the hood" is that an array is being created with the key `foo` in the main `$_SESSION` variable and any data that is saved or recalled by the `foo` namespaced session object will be stored in that array.

```
$sessFoo = new Pop\Session\SessionNamespace('foo');
echo $sessFoo->bar; // echos out 'baz'

$sess = Pop\Session\Session::getInstance();
echo $sess->bar; // echos out null, because it was only stored in the namespaced␣
↪session
```

And you can unset a value under a session namespace like this:

```
$sessFoo = new Pop\Session\SessionNamespace('foo');
unset($sessFoo->bar);
```

Session Value Expirations

Both basic sessions and namespaced sessions support timed values used to "expire" a value stored in session.

```
$sess = Pop\Session\Session::getInstance();
$sess->setTimedValue('foo', 'bar', 60);
```

The above example will set the value for `foo` with an expiration of 60 seconds. That means that if another request is made after 60 seconds, `foo` will no longer be available in session.

Request-Based Session Values

Request-based session values can be stored as well, which sets a number of time, or "hops", that a value is available in session. This is useful for **flash messaging**. Both basic sessions and namespaced sessions support request-based session values.

```
$sess = Pop\Session\Session::getInstance();
$sess->setRequestValue('foo', 'bar', 3);
```

The above example will allow the value for `foo` to be available to the user for 3 requests. After the 3rd request, `foo` will no longer be available in session. The default value of "hops" is 1.

pop-validator

The *popphp/pop-validator* component provides a basic API to process simple validation of values.

Installation

Install it directly into your project:

```
composer require popphp/pop-validator
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-validator": "3.0.*",
    }
}
```

Basic Use

Here's a list of the available built-in validators:

Built-in Validators		
AlphaNumeric	Ipv4	LessThanEqual
Alpha	Ipv6	LessThan
BetweenInclude	IsSubnetOf	NotContains
Between	LengthBetweenInclude	NotEmpty
Contains	LengthBetween	NotEqual
CreditCard	LengthGte	Numeric
Email	LengthGt	RegEx
Equal	LengthLte	Subnet
GreaterThanEqual	LengthLt	Url
GreaterThan	Length	

Here's an example testing an email value

```
$validator = new Pop\Validator\Email();

// Returns false
if ($validator->evaluate('bad-email-address')) {
    // Prints out the default message 'The value must be a valid email format.'
    echo $validator->getMessage();
}

// Returns true
if ($validator->evaluate('good@email.com')) {
    // Do something with a valid email address.
}
```

Validate Specific Values

```
$validator = new Pop\Validator\LessThan(10);

if ($validator->evaluate(8)) { } // Returns true
```

Set a Custom Message

```
$validator = new Pop\Validator\RegEx(
    '/^.*\.(jpg|jpeg|png|gif)$/i',
    'You must only submit JPG, PNG or GIF images.'
);

// Returns false
if ($validator->evaluate('image.bad')) {
```

```
    echo $validator->getMessage();
}
```

Alternatively:

```
$validator = new Pop\Validator\RegEx('/^.*\.(jpg|jpeg|png|gif)$/i');
$validator->setMessage('You must only submit JPG, PNG or GIF images.');

if ($validator->evaluate('image.jpg')) { } // Returns true
```

pop-view

The *popphp/pop-view* component provides an API to build and manage views to display to the user interface. Both file-based and stream-based templates are supported.

Installation

Install it directly into your project:

```
composer require popphp/pop-view
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/pop-view": "3.0.*",
    }
}
```

Basic Use

The `popphp/pop-view` component provides the functionality for creating and rendering views within your application. Data can be passed into a view, filtered and pushed out to the UI of the application to be rendering within the view template. As mentioned in the MVC section of the user guide, the `popphp/pop-view` component supports both file-based and stream-based templates.

Files

With file-based view templates, the view object utilizes traditional PHP script files with the extension `.php` or `.phtml`. The benefit of this type of template is that you can fully leverage PHP in manipulating and displaying your data in your view.

Let's revisit and expand upon the basic example given in the previous MVC section. First let's take a look at the view template, `index.phtml`:

```
<!DOCTYPE html>
<html>

<head>
    <title><?=$title; ?></title>
</head>
```

```
<body>
    <h1><?=$title; ?></h1>
<?=$content; ?>
    <ul>
<?php foreach ($links as $url => $link): ?>
        <li><a href="<?=$url; ?>"><?=$link; ?></a></li>
<?php endforeach; ?>
    </ul>
</body>

</html>
```

Then, we can set up the view and its data like below. Notice in the script above, we've set it up to loop through an array of links with the `$links` variable.

```
$data = [
    'title'   => 'View Example',
    'content' => '    <p>Some page content.</p>',
    'links'   => [
        'http://www.popphp.org/'     => 'Pop PHP Framework',
        'http://popcorn.popphp.org/' => 'Popcorn Micro Framework',
        'http://www.phirecms.org/'   => 'Phire CMS'
    ]
];

$view = new Pop\View\View('index.phtml', $data);

echo $view;
```

The result of the above example is:

```
<!DOCTYPE html>
<html>

<head>
    <title>View Example</title>
</head>
<body>
    <h1>View Example</h1>
    <p>Some page content.</p>
    <ul>
        <li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
        <li><a href="http://popcorn.popphp.org/">Popcorn Micro Framework</a></li>
        <li><a href="http://www.phirecms.org/">Phire CMS</a></li>
    </ul>
</body>

</html>
```

As mentioned before, the benefit of using file-based templates is you can fully leverage PHP within the script file. One common thing that can be utilized when using file-based templates is file includes. This helps tidy up your template code and makes script files easier to manage by re-using template code. Here's an example that would work for the above script:

header.phtml

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title><?=$title; ?></title>
</head>
<body>
```

footer.phtml

```
</body>

</html>
```

index.phtml

```
<?php include __DIR__ . '/header.phtml'; ?>
    <h1><?=$title; ?></h1>
<?=$content; ?>
    <ul>
<?php foreach ($links as $url => $link): ?>
        <li><a href="<?=$url; ?>"><?=$link; ?></a></li>
<?php endforeach; ?>
    </ul>
<?php include __DIR__ . '/footer.phtml'; ?>
```

Streams

With stream-based view templates, the view object uses a string template to render the data within the view. While using this method doesn't allow the use of PHP directly in the template like the file-based templates do, it does support basic logic and iteration to manipulate your data for display. The benefit of this is that it provides some security in locking down a template and not allowing PHP to be directly processed within it. Additionally, the template strings can be easily stored and managed within the application and remove the need to have to edit and transfer template files to and from the server. This is a common tactic used by content management systems that have template functionality built into them.

Let's look at the same example from above, but with a stream template:

```
$tmpl = <<<TMPL
<!DOCTYPE html>
<html>

<head>
    <title>[{title}]</title>
</head>
<body>
    <h1>[{title}]</h1>
[{content}]
    <ul>
[{links}]
        <li><a href="[{key}]">[{value}]</a></li>
[{/links}]
    </ul>
</body>

</html>
TMPL;
```

The above code snippet is a template stored as string. The stream-based templates use a system of **placeholders** to mark where you want the value to go within the template string. This is common with most string-based templating engines. In the case of `popphp/pop-view`, the placeholder uses the square bracket/curly bracket combination to wrap the variable name, such as `[{title}]`. In the special case of arrays, where iteration is allowed, the placeholders are marked the same way, but have an end mark like you see in the above template: `[{links}]` to `[{/links}]`. The iteration you need can happen in between those placeholder marks.

Let's use the exact same examples from above, except passing the string template, `$tmpl`, into the view constructor:

```
$data = [
    'title'   => 'View Example',
    'content' => '    <p>Some page content.</p>',
    'links'   => [
        'http://www.popphp.org/'     => 'Pop PHP Framework',
        'http://popcorn.popphp.org/' => 'Popcorn Micro Framework',
        'http://www.phirecms.org/'   => 'Phire CMS'
    ]
];

$view = new Pop\View\View($tmpl, $data);

echo $view;
```

We can achieve exact same results as above:

```
<!DOCTYPE html>
<html>

<head>
    <title>View Example</title>
</head>
<body>
    <h1>View Example</h1>
    <p>Some page content.</p>
    <ul>
        <li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
        <li><a href="http://popcorn.popphp.org/">Popcorn Micro Framework</a></li>
        <li><a href="http://www.phirecms.org/">Phire CMS</a></li>
    </ul>
</body>

</html>
```

As mentioned before, the benefit of using stream-based templates is you can limit the use of PHP within the template for security, as well as store the template strings within the application for easier access and management for the application users. And, streams can be stored in a number of ways. The most common is as a string in the application's database that gets passed in to the view's constructor. But, you can store them in a text-based file, such as `index.html` or `template.txt`, and the view constructor will detect that and grab the string contents from that template file. This will be applicable when we cover **includes** and **inheritance**, as you will need to be able to reference other string-based templates outside of the main one currently being used by the view object.

Stream Syntax

Scalars

Examples of using scalar values were shown above. You wrap the name of the variable in the placeholder bracket notation, `[{title}]`, in which the variable `$title` will render.

Arrays

As mentioned in the example above, iterating over arrays use a similar bracket notation, but with a start key `[{links}]` and an end key with a slash `[{/links}]`. In between those markers, you can write a line of code in the template to define what to display for each iteration:

```
$data = [
    'links'   => [
        'http://www.popphp.org/'     => 'Pop PHP Framework',
        'http://popcorn.popphp.org/' => 'Popcorn Micro Framework',
        'http://www.phirecms.org/'   => 'Phire CMS'
    ]
];
```

```
[{links}]
        <li><a href="[{key}]">[{value}]</a></li>
[{/links}]
```

Additionally, when you are iterating over an array in a stream template, you have access to a counter in the form of the placeholder, `[{i}]`. That way, if you need to, you can mark each iteration uniquely:

```
[{links}]
        <li id="li-item-[{i}]"><a href="[{key}]">[{value}]</a></li>
[{/links}]
```

The above template would render like this:

```
<li id="li-item-1"><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
<li id="li-item-2"><a href="http://popcorn.popphp.org/">Popcorn Micro Framework</a></
↪li>
<li id="li-item-3"><a href="http://www.phirecms.org/">Phire CMS</a></li>
```

You can also access nested associated arrays and their values by key name, to give you an additional level of control over your data, like so:

```
$data = [
    'links' => [
        [
            'title' => 'Pop PHP Framework',
            'url'   => 'http://www.popphp.org/'
        ],
        [
            'title' => 'Popcorn Micro Framework',
            'url'   => 'http://popcorn.popphp.org/'
        ]
    ]
];
```

```
[{links}]
        <li><a href="[{url}]">[{title}]</a></li>
[{/links}]
```

The above template and data would render like this:

```
<li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
<li><a href="http://popcorn.popphp.org/">Popcorn Micro Framework</a></li>
```

Conditionals

Stream-based templates support basic conditional logic as well to test if a variable is set. Here's an "if" statement:

```
[{if(foo)}]
    <p>The variable 'foo' is set to [{foo}].</p>
[{/if}]
```

And here's an "if/else" statement:

```
[{if(foo)}]
    <p>The variable 'foo' is set to [{foo}].</p>
[{else}]
    <p>The variable 'foo' is not set.</p>
[{/if}]
```

You can also use conditionals to check if a value is set in an array:

```
[{if(foo[bar])}]
    <p>The value of '$foo[$bar]' is set to [{foo[bar]}].</p>
[{/if}]
```

Furthermore, you can test if a value is set within a loop of an array, like this:

```
$data = [
    'links' => [
        [
            'title' => 'Pop PHP Framework',
            'url'   => 'http://www.popphp.org/'
        ],
        [
            'title' => 'Popcorn Micro Framework'
        ]
    ]
];
```

```
[{links}]
[{if(url)}]
        <li><a href="[{url}]">[{title}]</a></li>
[{/if}]
[{/links}]
```

The above template and data would only render one item because the *url* key is not set in the second value:

```
<li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
```

An "if/else" statement also works within an array loop as well:

```
[{links}]
[{if(url)}]
        <li><a href="[{url}]">[{title}]</a></li>
```

```
[{else}]
        <li>No URL was set</li>
[{/if}]
[{/links}]
```

```
<li><a href="http://www.popphp.org/">Pop PHP Framework</a></li>
<li>No URL was set</li>
```

Includes

As referenced earlier, you can store stream-based templates as files on disk. This is useful if you want to utilize includes with them. Consider the following templates:

header.html

```
<!DOCTYPE html>
<html>

<head>
    <title>[{title}]</title>
</head>
<body>
```

footer.html

```
</body>

</html>
```

You could then reference the above templates in the main template like below:

index.html

```
{{@include header.html}}
    <h1>[{title}]</h1>
[{content}]
{{@include footer.html}}
```

Note the include token uses a double curly bracket and @ symbol.

Inheritance

Inheritance, or blocks, are also supported with stream-based templates. Consider the following templates:

parent.html

```
<!DOCTYPE html>
<html>

<head>
{{header}}
    <title>[{title}]</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
{{/header}}
</head>
```

```
<body>
    <h1>[{title}]</h1>
    [{content}]
</body>

</html>
```

child.html

```
{{@extends parent.html}}

{{header}}
{{parent}}
    <style>
        body { margin: 0; padding: 0; color: #bbb;}
    </style>
{{/header}}
```

Render using the parent:

```
$view = new Pop\View\View('parent.html');
$view->title   = 'Hello World!';
$view->content = 'This is a test!';

echo $view;
```

will produce the following HTML:

```
<!DOCTYPE html>
<html>

<head>

    <title>Hello World!</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

</head>

<body>
    <h1>Hello World!</h1>
    This is a test!
</body>

</html>
```

Render using the child:

```
$view = new Pop\View\View('child.html');
$view->title   = 'Hello World!';
$view->content = 'This is a test!';

echo $view;
```

will produce the following HTML:

```
<!DOCTYPE html>
<html>
```

```
<head>

    <title>Hello World!</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

    <style>
        body { margin: 0; padding: 0; color: #bbb;}
    </style>

</head>

<body>
    <h1>Hello World!</h1>
    This is a test!
</body>

</html>
```

As you can see, using the child template that extends the parent, the `{{header}}` section was extended, incorporating the additional **style** tags in the header of the HTML. Note that the placeholder tokens for the extending a template use double curly brackets.

Filtering Data

You can apply filters to the data in the view as well for security and tidying up content. You pass the `addFilter()` method a callable and any optional parameters and then call the `filter()` method to iterate through the data and apply the filters.

```
$view = new Pop\View\View('index.phtml', $data);
$view->addFilter('strip_tags');
$view->addFilter('htmlentities', [ENT_QUOTES, 'UTF-8'])
$view->filter();

echo $view;
```

You can also use the `addFilters()` to apply muliple filters at once:

```
$view = new Pop\View\View('index.phtml', $data);
$view->addFilters([
    [
        'call'   => 'strip_tags'
    ],
    [
        'call'   => 'htmlentities',
        'params' => [ENT_QUOTES, 'UTF-8']
    ]
]);

$view->filter();

echo $view;
```

And if need be, you can clear the filters out of the view object as well:

```
$view->clearFilters();
```

popcorn

The Popcorn PHP Micro-Framework is a lightweight REST-based micro-framework that's built on top of the Pop PHP Framework core components. With it, you can rapidly wire together the routes and configuration needed for your REST-based web application, while leveraging the pre-existing features and functionality of the Pop PHP Framework.

Installation

Install it directly into your project:

```
composer require popphp/popcorn
```

Or, include it in your composer.json file:

```
{
    "require": {
        "popphp/popcorn": "3.1.*",
    }
}
```

Basic Use

In a simple `index.php` file, you can define the routes you want to allow in your application. In this example, we'll use simple closures as our controllers. The wildcard route '*' can serve as a "catch-all" to handle routes that are not found or not allowed.

```
use Popcorn\Pop;

$app = new Pop();

// Home page: http://localhost:8000/
$app->get('/', function() {
    echo 'Hello World!';
});

// Say hello page: http://localhost:8000/hello/john
$app->get('/hello/:name', function($name) {
    echo 'Hello ' . ucfirst($name) . '!';
});

// Wildcard route to handle errors
$app->get('*', function() {
    header('HTTP/1.1 404 Not Found');
    echo 'Page Not Found.';
});

// Post route to process an auth request
$app->post('/auth', function() {
    if ($_SERVER['HTTP_AUTHORIZATION'] == 'my-token') {
        echo 'Auth successful';
    } else {
        echo 'Auth failed';
    }
});
```

```
$app->run();
```

In the above POST example, if you attempted access that URL via GET (or any method that wasn't POST), it would fail. If you access that URL via POST, but with the wrong application token, it will return the 'Auth failed' message as enforced by the application. Access the URL via POST with the correct application token, and it will be successful:

```
curl -X POST --header "Authorization: bad-token" http://localhost:8000/auth
Auth failed

curl -X POST --header "Authorization: my-token" http://localhost:8000/auth
Auth successful
```

Advanced Usage

In a more advanced example, we can take advantage of more of an MVC-style of wiring up an application using the core components of Pop PHP with Popcorn. Keeping it simple, let's look at a controller class `MyApp\Controller\IndexController` like this:

```
namespace MyApp\Controller;

use Pop\Controller\AbstractController;
use Pop\Http\Request;
use Pop\Http\Response;
use Pop\View\View;

class IndexController extends AbstractController
{

    protected $response;
    protected $viewPath;

    public function __construct()
    {
        $this->request = new Request();
        $this->response = new Response();
        $this->viewPath = __DIR__ . '/../view/';
    }

    public function index()
    {
        $view = new View($this->viewPath . '/index.phtml');
        $view->title = 'Welcome';

        $this->response->setBody($view->render());
        $this->response->send();
    }

    public function error()
    {
        $view = new View($this->viewPath . '/error.phtml');
        $view->title =  'Error';

        $this->response->setBody($view->render());
        $this->response->send(404);
    }
```

```
}
```

and two view scripts, `index.phtml` and `error.phtml`, respectively:

```
<!DOCTYPE html>
<!-- index.phtml //-->
<html>

<head>
    <title><?=$title; ?></title>
</head>

<body>
    <h1><?=$title; ?></h1>
    <p>Hello World.</p>
</body>

</html>
```

```
<!DOCTYPE html>
<!-- error.phtml //-->
<html>

<head>
    <title><?=$title; ?></title>
</head>

<body>
    <h1 style="color: #f00;"><?=$title; ?></h1>
    <p>Sorry, that page was not found.</p>
</body>

</html>
```

Then we can set the app like this:

```
use Popcorn\Pop;

$app = new Pop();

$app->get('/', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'index',
    'default'    => true
]);

$app->run();
```

The 'default' parameter sets the controller as the default controller to handle routes that aren't found. Typically, there is a default action such as an 'error' method to handle this.

API Overview

Here is an overview of the available API within the module `Popcorn\Pop` class:

- `get($route, $controller)` - Set a GET route

- `head($route, $controller)` - Set a HEAD route
- `post($route, $controller)` - Set a POST route
- `put($route, $controller)` - Set a PUT route
- `delete($route, $controller)` - Set a DELETE route
- `trace($route, $controller)` - Set a TRACE route
- `options($route, $controller)` - Set an OPTIONS route
- `connect($route, $controller)` - Set a CONNECT route
- `patch($route, $controller)` - Set a PATCH route
- `setRoute($method, $route, $controller)` - Set a specific route
- `setRoutes($methods, $route, $controller)` - Set a specific route and apply to multiple methods at once

The `setRoutes()` method allows you to set a specific route and apply it to multiple methods all at once, like this:

```
use Popcorn\Pop;

$app = new Pop();

$app->setRoutes('get,post', '/login', [
    'controller' => 'MyApp\Controller\IndexController',
    'action'     => 'login'
]);

$app->run();
```

In the above example, the route `/login` would display the login form on GET, and then submit the form on POST, processing and validating it.

CHAPTER 7

Changelog

3.6.1

Reinstated Components

- pop-i18n

3.6.0

New Components

- pop-debug

Updated Components

- pop-cache
- pop-db

3.5.2

Updated Components

- pop-config
- pop-image
- pop-pdf
- pop-session
- popphp
- popcorn

3.5.1

Updated Components

- pop-auth
- popcorn
- pop-http
- pop-db

3.5.0

New or Changed Features

- The Database component has been significantly refactored for v4.
- The Data compoenent has been deprecated and the CSV functionality has been moved into its own component, *pop-csv*.
- The File Component has been deprecated and the upload functionality has been moved to the Http component and the directory functionality has been moved into its own component, *pop-dir*.

Removed Features

- The *pop-archive* component has been removed.
- The *pop-crypt* component has been removed.
- The *pop-data* component has been removed (see above.)
- The *pop-feed* component has been removed.
- The *pop-file* component has been removed (see above.)
- The *pop-filter* component has been removed.
- The *pop-geo* component has been removed.
- The *pop-i18n* component has been removed.
- The *pop-payment* component has been removed.
- The *pop-shipping* component has been removed.
- The *pop-version* component has been removed.
- The *pop-web* component has been removed (see above.)

3.0.1

Changed

- The mail component was updated to version 3.0.0.

3.0.0

New Features

- The Cache component now supports Redis and Session adapters.
- The Session and Cookie classes of the deprecated *pop-web* component have been broken out into their own individual components, *pop-session* and *pop-cookie*.
- The *pop-version* component now can pull its source from the Pop website or from GitHub.

Changed Features

- The Record sub-component of the Db component has been refactored. Functionality with this should remain largely the same, but there may be some backward compatibility breaks in older code.

Deprecated Features

- Due to the unavailability or instability of the **apc/apcu/apc_bc** extensions, the APC adapter in the *pop-cache* component may not function properly in PHP 7.
- Due to the unavailability or instability of the **memcache/memcached** extensions, the Memcache & Memcached adapters in the *pop-cache* component may not function properly in PHP 7

Removed Features

- The *pop-web* component has been removed. The cookie and session sub-components have been ported into their own individual components respectively.
- The *pop-filter* component has been removed.
- The *pop-geo* component has been removed.
- The Rar adapter in the *pop-archive* component has been removed.