PHP Framework Framy Documentation

Version	Framy Version	Date	Description	Author
v1.0	v0.1.4	27.02.18	Initial version	Marco Bier

Table of Contents

1. Getting started with Framy

This is the official documentation from probably the badest PHP Framework ever. Framy! The Ugly little brother from Laravel, Symfony and other good frameworks.

Have fun or what ever ¯_(ツ)_/¯

1.1 Setup

You can easily setup Framy by copying its data to your Apache folder. There is no further installation or setup by Framy needed. This still requires an Apache and MySQL server.

1.2 Requirements

The Framy framework has only a few system requirements.

- PHP >= 7.0.0
- PDO PHP Extension
- MySQL

1.2 Configuration

1.2.1 Public Directory

After installing Framy you should configure your web server's document / web root to be the ***public*** directory. The ***index.php*** in this directory serves as the front controller for all HTTP requests entering your application.

1.2.2 Configuration files

All the configuration files are stored in the config directory. Each option is documented, so feel free to look through the files and get familiar with the options available to you.

1.2.3 Directory permissions

After configuring Framy you may need to set up some permissions. Framy needs in the Storage directory writeable otherwise your web server will not run.

1.2.4 CrypKey

You should change the CrypKey in ***config/app.php*** an random string. Typically, this string should be 32 characters long.

If the application key is not set, your user sessions and other encrypted data will not be secure!

1.2.4 Database connection

If you want a database connection you will need to configure it in the ***config/database.php*** file. It comes with an example where you just need to copy&paste you Information, also Framy can Handle multiple connections, if you want that you can simply copy&paste the hole block and just need to change the name to make the connection work.

1.2.5 Additional configuration

Framy needs almost no other configuration out of the box. You are free to get started developing! However, you may wish to review the ***config/app.php*** file and its documentation. It contains several options such as ***timezone*** and locale that you may wish to change according to your application.

1.3 Directory Structure

1.3.1 Introduction

The default Framy application structure is intended to provide a great starting point for both large and small applications. Of course, you are free to organize your application however you like. Framy imposes almost no restrictions on where any given class is located.

1.3.2 The Root Directory

In the root directory is every file of the project and Framy.

1.3.2.1 The app Directory

Inside the ***app*** directory, as you might expect, are all the core Class of your code and Framys. However especially for you its required to know that specially folder ***custom*** is for you. Because you are Special. Happy Birthday.

1.3.2.2 The config Directory

The *config* directory, as the name implies, contains all the config files. It's a great idea toread through all of these files and familiarize yourself with all of the options available to you.

1.3.2.3 The public Directory

The ***public*** directory contains the ***index.php*** file, which is the entry point for all requests entering your application. This directory also houses your assets such as images, JavaScript, and CSS.

1.3.2.4 The routes Directory

The ***routes*** directory contains the files ***web.php*** and ***api.php***. Where you register the web an api routes.

1.3.2.5 The storage Directory

The *storage* directory contains your views as well as your raw, un-compiled assets such as LESS, SASS, or JavaScript. This directory also houses all of your language files and everything else you want to store.

1.3.3 The App Directory

The ***app*** Directory which contains the core code of Framy. And your code, in the ***custom*** directory. For further information about the core code see ***4. Framy API Reference***.

1.4 Components
1.4.3 EventManager
1.4.3.1 Introduction

EventManager Component allows you to easily manage events throughout your application.

1.4.3.2 Usage

Accessing **EventManager** can be done in two ways. The preferable way using **EventManagerTrait**, but you can also access it directly, using ***EventManager::getInstance()***. Let's see a simple example of subscribing to an event called ***some.evet*** with an instance of ***YourHandler***:

```
class YourClass
{
	use EventManagerTrait;

	public function index() {
		$this->eventManager()->listen('some.event')->handler(new YourHandler());
	}
}
```

You're done! You have just subscribed to an event and the moment ***some.event*** is fired, your handler will process it.

1.4.3.3 Event handlers

Now let's take a look at ***YourHandler***. An event handler can be any class. ***EventManager*** will call ***handle(Event $event)*** method on your handler object, by default. You can, however, specify a method you want ***EventManager*** to call:

```
class YourHandler
{
	public function customHandle(Event $event) {
		// Do something with the $event...
	}
}

// Using your custom method
$this->eventManager()->listen('some.event')->handler(new YourHandler())-
>method('customHandle');
```

Besides using classes, you can also respond to an event using a callable:

```
// Using callable as event handler

$handler = function(Event $event){
	// Do something with the $event...
};

$this->eventManager()->listen('some.event')->handler($handler);
```

1.4.3.4 Firing events

To fire a simple event use the following code:

```
$this->eventManager()->fire('some.event');
```

You can also pass some data when firing an event, which will be passed to every event listener:

```
$data = [
        'some' => 'data',
        'ip' => '192.168.1.10'
];

$this->eventManager()->fire('some.event', $data);
```

Any given data that is not an ***Event*** object, will be converted to generic ***Event*** object and your data will be accessible either by using array keys, or as object properties:

```
class YourHandler
{
        public function customHandle(Event $event) {
                // Access your data
                echo $event->some; // 'data'
                echo $event['some'] // 'data'
        }
}
```

If you want to use custom ***Event*** data types, refer to section custom event classes

1.4.3.5 Firing events using a wildcard

You can also use wildcard to fire multiple events at once. The following code will fire all events starting with ***event***. and pass ***$data*** to each one of them:

```
$this->eventManager()->fire('event.*', $data);
```

1.4.3.6 Execution priority

EventManager allows you to specify an execution priority using ***priority()*** method. Here's an example:

```
// Specify a priority of execution for your event listeners
$this->eventManager()->listen('some.event')->handler(new YourHandler())-
>method('customHandler')->priority(250);
$this->eventManager()->listen('some.event')->handler(new YourHandler())-
>method('secondCustomHandler')->priority(400);
$this->eventManager()->listen('some.event')->handler(new YourHandler())-
>method('thirdCustomHandler');

// Now let's fire an event
$this->eventManager()->fire('some.event');
```

After firing an event, the event listeners will be ordered by priority in descending order. The higher the priority, the sooner the listener will be executed. In this example, the order of execution will be as follows: ***secondCustomHandler***, ***customHandler***, ***thirdCustomHandler***. Default priority is 101, so ***thirdCustomHandler*** is executed last.

1.4.3.7 Custom event classes

When firing events, you can also pass your own event classes, that extend generic ***Event*** class. For example, you want to fire an event called ***cms.page_saved*** and pass the Page object. Of course, you could simply pass an array like *['page' => $pageObject]*, but for the sake of the example, let's pretend it's more complicated than that:

```
// Create your `PageEvent` class
class PageEvent extends Event
{
	private $_page;

	public function __construct(Page $page) {
		// Call constructor of parent Event class
		parent::__construct();

		// Set your page object
		$this->_page = $page;
	}

	public function getPage() {
		return $this->_page;
	}
}

// Fire an event
$pageEvent = new PageEvent($pageObject);

$this->eventManager()->fire('cms.page_saved', $pageEvent);

// In your handler, you can now access page object using $event->getPage()
class YourHandler
{
	public function customHandle(PageEvent $event) {
		$pageObject = $event->getPage();
	}
}
```

This is a simple example, but it shows the power of creating your own ***Event*** classes and add as much functionality to your events as you need.

1.4.3.8 Event subscriber

Another cool feature of the ***EventManager*** is the ability to subscribe to multiple events at once. You will need to create a subscriber class implementing ***EventSubscriberInterface***:

```
class PageEventSubscriber implements EventSubscriberInterface
{
	use EventManagerTrait;

	/**
	* Handle page creation event
	*/
	public function onPageCreated($event) {
		//
	}

	/**
	* Handle page update
	*/
	public function onPageUpdated($event) {
		//
	}

	/**
	* Register the listeners for the subscriber.
	*/
	public function subscribe() {
		$this->eventManager()->listen('cms.page_created')->handler($this)->method('onPageCreated');
		$this->eventManager()->listen('cms.page_updated')->handler($this)->method('onPageUpdated');
	}
}

// Subscriber to multiple events using your new subscriber class
$this->eventManager()->subscribe($subscriber);
```

There are situations when you need to temporarily disable EventManager. For example, deleting a huge portion of files that are not directly related to the application (local cache files) does not require firing all of related events. In this case use the following methods:

```
// Disabling EventManager
$this->eventManager()->disable();

// Do some work that would fire loads of unnecessary events...

// Enabling EventManager
$this->eventManager()->enable();
```

1.4.4 Standard Library

1.4.4.1 Introduction

The Standard Library Component contains some useful functions It's more or less an overpowered helper Component

1.4.4.2 StdLibTrait & ValidorTrait

Both contain basic functions like serialize(), unserialize() and basic validators like isEmpty() and many more the usage of them is quite easy and self explaining so no further explanation needed.

1.4.4.3 StdObject

The standard object library is a set of classes that wraps the process of working with some common objects like arrays, strings, URLs and dates into a objective style (Currently at version 0.8 just Arrays).

The benefit of working with StdObject, instead of native objects, is that you can chain several methods inside one call which improves the speed how you write your code. The code is fully objective and we have also "improved" and "fixed" some of the PHP core functions.

Most of the internal classes inside Framy are written using standard objects.

1.4.4.4 ArrayObject

```
$array = new ArrayObject(['one', 'two', 'three']);
$array->first(); // StringObject 'one'
$array->append('four')->prepend('zero'); // ['zero', 'one', 'two', 'three', 'four']
```

1.4.4.5 DateTimeObject

```
$dt = new DateTimeObject('3 months ago');
echo $dt; // 2013-02-12 17:00:36
$dt->add('10 days')->sub('5 hours'); // 2013-02-22 12:00:36
```

1.4.4.6 StringObject

```
$string = new StringObject('Some test string.');
$string->caseUpper()->trimRight('.')->replace(' test'); // SOME STRING
```

1.4.4.7 UrlObject

```
$url = new UrlObject('http://www.framy.com/');
$url->setPath('search')->setQuery(['q'=>'some string']); // http://www.framy.com/search?
q=some+string
```

The easies and most optimal way to use the ***StdObject*** library is by implementing the ***StdObjectTrait***.

```
class MyClass
{
        use StdObjectTrait;

        public function test() {
                // create StringObject
                $this->str('This is a string');

                // create ArrayObject
                $this->arr(['one', 'two']);

                // create UrlObject
                $this->url('http://www.framy.com/');

                // create DateTimeObject
                $this->dateTime('now');
        }
}
```

1.4.5 Stopwatch

1.4.5.1 Introduction

The Stopwatch component provides a way to profile code.

1.4.5.2 Usage

The Stopwatch component provides an easy and consistent way to measure execution time of certain parts of code so that you don't constantly have to parse micro time by yourself. Instead, use the simple Stopwatch class:

```
use app\framework\Component\Stopwatch\Stopwatch;

$stopwatch = new Stopwatch();
// Start event named 'eventName'
$stopwatch->start('eventName');
// ... some code goes here
$event = $stopwatch->stop('eventName');
```

The ***StopwatchEvent*** object can be retrieved from the start(), stop(), lap() and getEvent() methods. The latter should be used when you need to retrieve the duration of an event while it is still running.

You can also provide a category name to an event:

```
$stopwatch->start('eventName', 'categoryName');
```

You can consider categories as a way of tagging events. For example, the Framy Profiler tool uses categories to nicely color-code different events.

1.4.5.3 Periods

As you know from the real world, all stopwatches come with two buttons: one to start and stop the stopwatch, and another to measure the lap time. This is exactly what the lap() method does:

```
$stopwatch = new Stopwatch();
// Start event named 'foo'
$stopwatch->start('foo');
// ... some code goes here
$stopwatch->lap('foo');
```

```
// ... some code goes here
$stopwatch->lap('foo');
// ... some other code goes here
$event = $stopwatch->stop('foo');
```

Lap information is stored as "periods" within the event. To get lap information call:

```
$event->getPeriods();
```

In addition to periods, you can get other useful information from the event object. For example:

```
$event->getCategory();      // Returns the category the event was started in
$event->getOrigin();        // Returns the event start time in milliseconds
$event->ensureStopped(); // Stops all periods not already stopped
$event->getStartTime();     // Returns the start time of the very first period
$event->getEndTime();       // Returns the end time of the very last period
$event->getDuration();      // Returns the event duration, including all periods
$event->getMemory();        // Returns the max memory usage of all periods
```

1.4.5.4 Sections

Sections are a way to logically split the timeline into groups. You can see how Framy uses sections to nicely visualize the framework life cycle in the Framy Profiler tool. Here is a basic usage example using sections:

```
$stopwatch = new Stopwatch();

$stopwatch->openSection();
$stopwatch->start('parsing_config_file', 'filesystem_operations');
$stopwatch->stopSection('routing');
$events = $stopwatch->getSectionEvents('routing');
```

You can reopen a closed section by calling the openSection() method and specifying the id of the section to be reopened:

```
$stopwatch->openSection('routing');
$stopwatch->start('building_config_tree');
$stopwatch->stopSection('routing');
```

1.4.6 Storage

1.4.6.1 Introduction

Storage Component is a storage abstraction layer that simplifies the way you work with files and directories.

1.4.6.2 Usage

You will need to use storage drivers to access different storage providers like local disk, Amazon, Rackspace, etc.

Framy Framework currently provides ***LocalStorageDriver*** but using a set of built-in interfaces will help you to develop a new driver in no time.

The following driver interfaces are available:

- ***DriverInterface*** - main storage driver interface
- ***TouchableInterface*** - for drivers that support touch functionality (change of time modified)
- ***SizeAwareInterface*** - for drivers that can access file size
- ***DirectoryAwareInterface*** - for drivers that can work with directories
- ***AbsolutePathInterface*** - for drivers that can provide absolute file path (ex: /var/www/app/storage/myFile.txt)

1.4.6.3 Configuring a storage service

The recommended way of using a storage is by defining a storage service. Here is an example of defining a service using ***LocalStorageDriver*** and ***S3StorageDriver***(this one only later on): NOTE: you can use **DIR** to have your file paths built dynamically. **DIR** will be replaced with the directory path containing current config file.

```
// inside config/filesystem.php
'disks' => [
    'public' => [
        'driver' => 'local',
        'root' => realpath(__DIR__."/../public"),
        'visibility' => 'public',
    ],

    'templates' => [
        'driver' => 'local',
        'root' => realpath(__DIR__.'/../storage/templates/')
    ],

    'language' => [
        'driver' => 'local',
        'root' => realpath(__DIR__.'/../storage/lang/')
    ]
]
```

1.4.6.4 Using your new storage

To make use of local storage easier and more flexible, there are 2 classes: ***\app\framework\Component\Storage\File\File*** and ***\app\framework\Component\Storage\Directory\Directory***. These 2 classes serve as wrappers so you never need to make calls to storage directly. Besides, they contain common methods you will need to perform actions on files and directories.

Let's take a look at how you would store a new file:

```
// use StorageTrait
// Get your storage service. Storage name is part of the service name after 'Storage.'
$storage = $this->storage('LocalStorage');

// Create a file object with a key (file name) and a $storage instance
$file = new File('file.txt', $storage);
```

```
$contents = file_get_contents(
                        'http://www.w3schools.com/images/w3schoolslogoNEW310113.gif'
                );
$file->setContents($contents);
```

After calling ***setContents($contents)*** the contents is written to the storage immediately and a bool is returned.

1.4.6.5 Working with directories

Sometimes you need to read the whole directory, filter files by name, extension, etc. There is a special interface for this type of manipulation, ***\app\framework\Component\Storage\Directory\DirectoryInterface***.

Since not all storage engines support directories, there is no generic implementation of this interface. There is, however, an implementation in form of Directory which works nicely with local file storage. There are 2 modes of reading a directory: recursive and non-recursive.

- Recursive will read the whole directory structure recursively and build a one-dimensional array of files (directory objects are not created but their children files are returned). This is very useful when you need to read all files at once or filter them by name, extension, etc.
- Non-recursive will only read current directory and return both child Directory and Fileobjects. You can then loop through child Directory object to go in-depth.

1.4.6.6 Reading a directory (non-recursive mode)

```
// Get your storage service
$storage = $this->storage('LocalStorage');

// Read recursively
$dir = new Directory('2013', $storage, true);

// Get only PDF files
$pdfFiles = $dir->filter('*.pdf');

// Count ZIP files
$zipFiles = $dir->filter('*.zip')->count();

// Get files starting with 'log_'
$logFiles = $dir->filter('log_*');

// You can also pass the result of filter directly to loops as `filter()` returns a new Directory object
foreach($dir->filter('*.txt') as $file){
   // Do something with your file
}
```

NOTE: calling ***filter()*** does not change the original Directory object, but creates a new Directory object with filtered result, so once you've read the root directory you can filter it using any condition as many times as you need:

```
// Get your storage service
```

```
$storage = $this->storage('LocalStorage');

// Read recursively and don't filter
$dir = new Directory('2013', $storage, true);

// Now you can manipulate the whole directory

// Get number of all ZIP files in the directory tree
$zipFiles = $dir->filter('*.zip')->count();

// Get number of all RAR files in the directory tree
$zipFiles = $dir->filter('*.rar')->count();

// Get number of all LOG files in the directory tree
$zipFiles = $dir->filter('*.log')->count();

// Now output all files in the directory without filtering them
foreach($dir as $file){
    echo $file->getKey();
}
```

1.4.6.7 Filtering files (recursive mode)

To make use of local storage easier and more flexible, there are 2 classes: ***\app\framework\Component\Storage\File\File*** and ***\app\framework\Component\Storage\Directory\Directory***. These 2 classes serve as wrappers so you never need to make calls to storage directly. Besides, they contain common methods you will need to perform actions on files and directories.

Let's take a look at how you would store a new file:

```
// use StorageTrait
// Get your storage service. Storage name is part of the service name after 'Storage.'
$storage = $this->storage('LocalStorage');

// Create a file object with a key (file name) and a $storage instance
$file = new File('file.txt', $storage);

$contents = file_get_contents(
                        'http://www.w3schools.com/images/w3schoolslogoNEW310113.gif'
                );
$file->setContents($contents);
```

After calling ***setContents($contents)*** the contents is written to the storage immediately and a bool is returned.

1.4.6.8 Deleting a directory

Deleting a directory (this is done recursively) is as simple as:

```
// Get your storage service
$storage = $this->storage('LocalStorage');

// Get directory
$dir = new Directory('2013', $storage);
```

```
// This will delete the whole directory structure and fire FILE_DELETED event for each file
along the way
$dir->delete();

// If you don't want the events to be fired, pass as second parameter `false`:
$dir->delete(false);
```

1.4.6.9 Storage events

There are 3 types of events fired when certain actions are performed on a File:

- ***ff.storage.file_saved*** (StorageEvent::FILE_SAVED) - fired after a file was saved successfully.
- ***ff.storage.file_renamed*** StorageEvent::FILE_RENAMED) - fired after a file is renamed.
- ***ff.storage.file_deleted*** (StorageEvent::FILE_DELETED) - fired after a file is deleted.

All 3 events pass an instance of ***\app\framework\Component\Storage\StorageEvent*** to their event handlers. Once your handler gets executed, you can access the file objects using ***$event->getFile()*** method.

```
class Test
{
    use EventManagerTrait;

    public function index() {

        // Listen for StorageEvent::FILE_SAVED
        $this->eventManager()->listen(StorageEvent::FILE_SAVED)-
>handler(function(StorageEvent $event) {
            // Get the file object
            $file = $event->getFile();
        });
    }
}
```

StorageEvent::FILE_RENAMED event also assigns a special property called oldKey to the event object, which holds the value of file key before renaming (this will help you update your database records, etc.):

1.4.7 Validation

1.4.7.1 introduction

Validation of data means checking that the data provided by the user is in accordance with the expected format. For example, if you pass the ID of an item in your online shop as a GET parameter, you should check that this value is actually a number. Good data validation can significantly increase your protection against SQL injections and cross-site scripting.

In addition to the increase in security, a good validation of the input data also provides you with increased user experience, as incorrect entries are intercepted at an early stage.

In principle, you should check all entries of your users and point out wrong entries by means of suitable hints. A mistyping of the e-mail address happens quickly and can be annoying if it is not noticed.

1.4.7.2 Usage

Basic Example usage:

```
Validation::getInstance()->validate('example@web.de', 'email');  // returns true

Validation::getInstance()->validate('false.mail.de', 'email');  // throws Exception

Validation::getInstance()->validate('false.mail.de', 'email', false); // returns "Invalid email"
```

You can either use the example above or use the Validation Trait.

Like this:

```
use app\framework\Component\Validation\ValidationTrait;

class yourClass
{
    use ValidationTrait;

    function yourFunction()
    {
        self::validate()->validate('example@web.de', 'email');
    }
}
```

1.4.7.3 General Information

If the given Validators don't mach your requirements, you can add your own by using the **addValidator()** function.

2. The Basics

2.1 Creating Pages

Quite at first you should register the route. You can read how to exactly do this under Routing. But here an example:

```
$klein->respond("GET", "/", function(){
    view("welcome");
});
```

Inside the function can everything happen like calling an method from an class. In this example we will not do this.

We will just use the **view()** helper function and display the "welcome" template

To do this we need to add an Template. Lets name it ***welcome.tpl*** (.tpl ending -> syntax Smarty requires)

We add the file under *storage/templates*.

welcome.tpl shall look like this:

```
<head>
    <title>Framy</title>
</head>
<body>
<div class="flex-center position-ref full-height">
    <div class="content">
        <div class="title m-b-md">
            Framy
        </div>
    </div>
</div>
</body>
</html>
```

2.2 Routing

Klein Documentation.

2.3 Controller

2.3.1 Introduction

Instead of defining all of your request handling logic as Closures in route files, you may wish to organize this behaviour using Controller classes. Controllers can group related request handling logic into a single class. Controllers are stored in the ***app/custom/Http/Controller*** directory.

2.4 Views

2.4.1 Usage

With the awesome helper function view you can display your views extreamly simple:

```
view("welcome");
// or view("welcome.tpl") both works but the file itself must have .tpl as ending
```

Shows the *view.tpl* file inside the pre defined template disk. If you want to use sub directorys you can do that but you need to type them in like this: ***view("sub/dir/yourview");***

2.4.2 Extra features

You can assing data to Smarty values by simply adding it as an array parameter:

```
view("yourTemplate", ["valueName" => "someValuedataThatCouldBeAnythingYouWant"]);
```

You can add as many as you want.

You can even assign rendered templates to values:

```
view("yourTemplate", ["valueName" => "fetch:someTemplate"]);
```

3. Release Notes

3.1 Version 0

v0.1.3.2

- **Fix**: helper function app() default namespaces had wrong spelling
- **Fix**: little fetch issue in view
- **New**: helper function pathTo() gives the path to the project and adds the possibility to add some path

v0.1.3.1

- I am an stupid moron

v0.1.3

- **New**: Cookie comp.
- **Fix**: Issue #12, #13, #14

v0.1.2

- **New**: Fetch support for View comp.

v0.1.1

- **New**: Added EventManagerTrait for Model Comp.

v0.1.0

- **New**: Moved bootstrap file in to bootstrap folder and exported helper in to a separate file
- **New**: start page template
- **New**: helper function app, to call methods and store the class instance.
- **New**: helper function url, to get the URL completely by only entering shit like: /test to example.com/test
- **Fix**: Spelling correction

v0.1.0-alpha.3

- **New**: Changed the View Comp. you can now edit template path and compile template path in an new config file (view)
- **New**: version available in app config file
- **Fix**: resolved some issues: like an missing semicolon.
- **Fix(kind of)**: some shitty clean up work, like remove empty spaces.

v0.1.0-alpha.2

- **New**: Model component
- **New**: View comp.
- Changed Database comp.
- Made the Code a little more beautiful
- maybe more I forgot about

v0.1.0-alpha.1

- init version