kiwi42 Documentation

Release latest

Thomas Payer, Bernhard Efler, Lucas Seinfeld

Aug 22, 2017

Introduction

A Zend Framework based Content Management System for Developers.

The documentation is organized into following main sections:

- *Introduction* Get a basic understanding of what the idea behind kiwi42 is.
- *Getting Started* How to setup a kiwi42 project.
- *Components* Information on usage of core components.
- *Frontend Development* How to build and integrate assets and output site contents.
- *Extend & Customize* Whitelabeling.

CHAPTER 1

Core Concepts

kiwi42 is a Zend Framework (ZF) based Content Management System (CMS) targeted at developers. It's meant for small to large scaled projects while maintaining full flexibility for individual requirements at all times.

The core feature set includes but is not bound to:

- Users & Permissions by Roles
- Multilingual Contents
- Content Management
- Event Management
- Media Management
- Translation

Content Data Hierarchy

The data structure for site contents within kiwi42 may be reduced to this simple hierarchy:

- **Sitemap taking care of depth, language and versioning**
  - **Pages with a defined Page Type**
    - **Sections/Areas within Pages**
      - **Blocks dynamically added to Sections/Areas**

        Fields holding content data or referencing to Media, Links, etc.

Next to the structured content data above kiwi42 allows to easily maintain arbitrary resources that may be administrated through basic Create, Read, Update and Delete (CRUD) actions. kiwi42 provides the basic functionality and views for these but may be extended for more sophisticated and complex use cases.

Directory Structure

kiwi42 lends its directory structure from the ZF Skeleton application for zend-mvc projects. Only a few structural requirements have to be fulfilled within custom modules to work with kiwi42 which will be mentioned and highlighted wherever applicable within this documentation. Apart from these kiwi42 does not dictate how developers should set up their application to maintain an optimal workflow - *especially* when it comes to the front end. Have it your way.

To be as flexible as possible when it comes to different deployment methods and server setups kiwi42 tries to introduce some best practices.

- The *data* folder in the project root is ignored by default. This allows it to be moved to a shared location much easier when e.g. using Capistrano's releases and causes no permission conflicts by being included in the repository.
- The same applies to assets in general as they might not be served from the same *public* folder but a "static" CDN host instead. Assets' folders within modules are linked to the *resources* root folder from where a link is added to the *public* folder but may be moved somewhere else easily.

User Interface

The administration area's User Interface (UI) is enhanced by AngularJS (version 1 on principle as of this writing) for a fluid and dynamic user experience (UX) while keeping the interface itself to a bare minimum. It is based on Flatfull's Angulr Theme which maintains optimal usability on mobile devices.

The main AngularJS module is developed within the admin42 module and gets extended from within other modules as well.

Architecture

kiwi42 itself may be considered being a slightly adapted and opinionated version of the ZF Skeleton application for zend-mvc projects. raum42's additional custom modules and extensions follow ZF's architectural patterns to a very high degree whenever possible and useful.

However specific parts and tasks have been replaced by more specialized libraries such as:

- Stash Cache for caching
- Swiftmailer for sending mails
- Monolog for logging

Take a look at the latest composer.json file to get a sense of how the stack is composed.

raum42 provides its own set of modules as an application foundation and are declared as dependencies of kiwi42. These modules take care of abstracting and enhancing ZF to fit kiwi42's needs from "end to end".

core42

Provides the Application Framework Layer. It includes the fruit console and base classes for abstracting and extending the Zend Framework.

admin42

Includes the main components of the administration area.

frontend42

Provides content related features for admin42, houses the core sitemap features and makes them accessible for view templates. It manages and processes page types and blocks and their configuration.

CHAPTER 2

Installation

Requirements

To install kiwi42, you will need:

- PHP 5.6/7.0+
- php-intl extension
- php-fileinfo extension
- php-mbstring extension
- php-pdo extension with mysql support
- php-imagick extension (suggested)
- MySQL >=5.6 or equivalent
- Composer

Moreover php-cli should be available.

Install

Use Composer's `create-project` in your command line (CLI):

```
$ composer create-project raum42/kiwi42 kiwi42-project
$ cd kiwi42-project
```

Now run the setup command provided by kiwi42's own *Fruit Console* which will be explained in detail later on:

```
$ bin/fruit setup
```

This will among other things turn on development mode, run database migrations and symlink/copy the assets folders given by the credentials you provided. The resulting configuration can be found in the `config/autoload` directory.

Permissions

Set write permissions/ownership to the `data` folder recursively for the user that will run the application.

Serve

To run your kiwi42 project set up your local development server accordingly (virtual hosts) and point them to the `public` directory.

Tip: The Zend Framework Skeleton Application ships with both Vagrantfile and Dockerfile. Feel free to use these if you want to virtualize your development environment. Refer to the respective documentations for installation and setup.

Now configure the application to your needs which will be explained in general in the following *Configuration* section and in detail wherever relevant.

CHAPTER 3

Configuration

Configurations can be shipped inside a module and/or can be set/overwritten inside the `config/autoload/` directory by adding files with an `config.php` file ending. Static or default configurations should be shipped inside the module directory, environment based configurations (for production server, staging, development etc.) should be inside the `config/autoload/` directory.

Occasionally you need to set configurations only for the administration panel of kiwi42 (for example extending the navigation inside the administration panel). Therefor you can add an admin directory in you module config or in your `config/autoload/` directory. All configuration files inside this admin directory will be only loaded, when the administration panel is requested.

Environment based configurations should be located in `config/autoload/` and prefixed with `local.` (like `local.mail.config.php`). They will be merged on top of all other available configurations. `local*` files will be git ignored by default.

After a basic setup with fruit console there should be some separated configuration files (assets, database, ...) but they may be combined into a single `local.config.php` as well.

Current configuration values can be inspected by using fruit console. For example to see which assets folders are registered within modules and will be symlinked/copied run the following on your command line:

```
$ bin/fruit config assets
```

Configuration values and options will be explained in detail where they are relevant within this documentation. Some basic configurations below.

Database Config

```
[
    'db' => [
        'adapters' => [
            'Db\Master' => [
                'database' => 'your_database',
                'username' => 'root',
```

```
                'password' => '',
                'hostname' => '127.0.0.1',
            ]
        ]
    ]
]
```

Mail Config

Null Mailer (default):

```
[
    'mail' => [
         'transport' => [
            'type' => 'null',
            'options' => [],
        ],
    ],
]
```

Over SMTP

```
[
    'mail' => [
        'transport' => [
            'type' => 'smtp',
            'options' => [
                'host'              => '', //optional, default: localhost
                'port'              => 25, //optional, default: 25
                'encryption'        => '', //optional (ssl|tls), default: tls
                'username'          => '', //optional
                'password'          => '', //optional

            ],
        ],
    ],
]
```

Over Sendmail

```
[
    'mail' => [
        'transport' => [
            'type' => 'sendmail',
            'options' => [
                'command'           => '', //optional, default: /usr/sbin/sendmail -bs
            ],
        ],
    ],
]
```

Over PHP Mail

```
[
    'mail' => [
```

```
        'transport' => [
            'type' => 'mail',
            'options' => [
                'extra'             => '', //optional, default: -f%s
            ],
        ],
    ],
]
```

CSP Protection Config

You can enable/disable the content security policy headers.

```
[
   'security' => [
        'csp' => [
            'enable'         => false,
            'nonce'          => false,
            'connect_src'    => false,
            'font_src'       => false,
            'img_src'        => false,
            'media_src'      => false,
            'object_src'     => false,
            'script_src'     => false,
            'style_src'      => false,
            'default_src'    => false,
            'form_action'    => false,
            'form_ancestors'=> false,
            'plugin_types'   => false,
            'child_src'      => false,
        ],
    ],
]
```

I18n Config

Definition which locales/languages are available in your frontend

```
[
   'i18n' => [
        'type' => 'language' // can be language or region
        'locales' => [
            'de-AT' => [
                'default' => true
            ],
            'en-US' => [
                'default' => false
            ],
        ]
    ]
]
```

Session Config

```
[
   'session_config' => [
        'name' => 'kiwi42',
        'use_trans_sid' => false,
        'use_cookies' => true,
        'use_only_cookies' => true,
        'cookie_httponly' => true,
    ]
]
```

Admin Config

```
[
   'admin' => [
        'timezone' => 'Europe/Vienna',
        'locale' => 'en-US',
        'assets' => [],
        'login_captcha' => false,
        'login_captcha_options' => [ //your google reCaptcha credentials
            'sitekey' => '',
            'secret' => '',
        ]
    ]
]
```

Project Config

Some basic information about your application (mainly used for email sending

```
[
   'project' => [
        'name' => 'kiwi42',
        'email_subject_prefix' => '[kiwi42]: ',
        'email_from' => 'noreply@kiwi42.com',
        'project_base_url' => 'http://kiwi42.com',
    ]
]
```

CHAPTER 4

Update & Upgrade

Through the power of Composer it's basically down to:

```
$ composer update
```

Note: **Upgrade Guides** for future releases will be made available within this documentation.

CHAPTER 5

Fruit Console

kiwi42 has its own console which is powered by an extended Zend Console within the core42 module.

To see all currently registered commands run the following on your command line from your project root:

```
$ bin/fruit
```

Modules may register their own commands. kiwi42's default application module includes an example command to show how it works.

The command's CLI route is registered in `module/application/config/cli.config.php` and can be run like this:

```
$ bin/fruit example --foo=bar
```

To show this or another command's help and parameter descriptions run:

```
$ bin/fruit help example
```

CHAPTER 6

Project Module

After creating a new kiwi42 project a default module is present within the `modules` folder called `application`. If you rename it make sure to reflect this change within `config/modules.config`.

Tip: For convenience: the project's main module folder should be named after the project itself and represent the namespace used for PHP classes. This usually makes it easier to reason about the modules' responsibilities.

CHAPTER 7

Page Types

PageTypes are an important part of kiwi. On one hand they are responsible how the content manager can build the sitemap of the website, on the other hand PageTypes provides information like routing or which controller and action should be dispatched. Moreover they act as a container around FormElements and provides a content manager a form with information about a given page (like the actual page content or meta information like a publish date).

Per default PageType configurations are located under `module/application/config/page_types`

An Example configuration:

```
[
    'name'          => '', // internal name of the pageType
    'label'         => '', // label of the pageType (will be displayed in the admin␣
↪interface)
    'handle'        => '', // a pageType handle
    'class'         => '', // the pageType class
    'view'          => '', // the view which should be rendered
    'root'          => false, //true | false | null ... weather the pageType can be␣
↪used as Root PageType
    'allowedChildren' => [], // array or null ... a list of pageTypes which are␣
↪allowed as direct children of the pageType
    'allowedParents'=> [], // array or null ... a list of pageTypes which are allowed␣
↪as direct parents of the pageType
    'properties'    => [], // some pageTypes might need some extra configurations
    'controller'    => '', // controller which will be dispatched
    'action'        => '', // action which will be dispatched
    'terminal'      => true, // weather it is allowed to append child pages
    'sorting'       => true, // weather it is allowed to sort this page
    'sections'      => [
        [
            'label' => 'general',
            'elements' => [
                //See FormElements for more Information
            ],
        ],
    ],
```

```
    'defaults'      => [
        //See FormElements for more Information
    ],
    'layout'        => '', //the layout which sould be rendered. Defaul "layout/layout
↪"
```

]

CHAPTER 8

Blocks

A block is an instance of a certain type of content that can be displayed in the page. Every block is specified by an configuration and a html snippet. Inside the specification you describe what a content editor of the website can/must enter for a certain block. Inside the html snippet, you can access this information.

```
[
    'name'          => '', // internal name of the block
    'label'         => '', // label of the block (will be displayed in the admin␣
↪interface)
    'elements' => [
        //See FormElements for more Information
    ],
```

]

CHAPTER 9

Form Elements

Building forms are a very basic part of kiwi. Outside of the "admin" Environment is the standard Zend Framework Form available. Inside the "admin" Environment is a modified version of the Zend Form in use. It still strongly depends on the Zend Form, but it has a slightly different way of defining forms and has it own view helpers (do work better together with angular).

This documentation only covers form building inside the administration interface of kiwi. If you need a form outside of the interface you can choose the technology you want (eg. Zend Form, Zend Inputfilter without Form, any other techique).

Base Elements

FormElements are definied in an simple Array-Syntax. Here a basic syntax of the definition (some FormElements might have extra configuration parameters which is covered later):

```
[
    'name' => '', //required
    'type' => '', //required
    'label' => '', //optional - but should be not empty - will run through the␣
↪internal translator
    'value' => '', //optional
    'required' => true, // optional, default: false
    'description' => '', //optional
]
```

Checkbox

Renders a single `<input type="checkbox">` element

```
[
    'type' => 'checkbox', //required
    'checkedValue' => '', //optional, default string 'true'
```

```
    'uncheckedValue' => '', //optional, default string 'false'
    // ...
]
```

Csrf

The Csrf element helps to provide protection from CSRF attacks on forms, ensuring the data is submitted by the session that generated the form. This is achieved by adding a hash value to the form data. For security reason this element should be used in every form.

```
[
    'type' => 'csrf', //required
    'csrfOptions' => [
        'salt' => '', //optional
        'timeout' => 0, //optional, default 300 seconds
    ],
    // ...
]
```

Date

This generates a date picker element with an attached date validator

```
[
    'type' => 'date', //required
    // ...
]
```

DateTime

This generates a date and time picker element with an attached date time validator

```
[
    'type' => 'dateTime', //required
    // ...
]
```

Email

Renders a `<input type="text">` element with an attached email validator

```
[
    'type' => 'email', //required
    // ...
]
```

Fieldset

Hidden

Renders a <input type="hidden"> element

```
[
    'type' => 'hidden', //required
    // ...
]
```

Link

Media

Through the media element media files can be selected/uploaded.

```
[
    'type' => 'media', //required
    'categorySelection' => '', //optional - select only from given category, default:␣
↪*
    'typeSelection' => '' //optional - select only given type. possible values are *,␣
↪images or pdf. default: *
    // ...
]
```

MultiCheckbox

Renders a one or more <input type="checkbox"> elements with an attached inArray validator

```
[
    'type' => 'multiCheckbox', //required
    'values' => [
        // key => label - label will run through the internal translator
    ], //required
    // ...
]
```

Password

Renders a <input type="password"> element

```
[
    'type' => 'password', //required
    // ...
]
```

Radio

Renders a one or more <input type="radio"> elements with an attached inArray validator

```
[
    'type' => 'radio', //required
    'values' => [
        // key => label - label will run through the internal translator
    ], //required
    // ...
]
```

Select

Renders a `<select>` element with an attached inArray validator

```
[
    'type' => 'select', //required
    'values' => [
        // key => label - label will run through the internal translator
    ], //required
    'emptyValue' => [
        // key => label - label will run through the internal translator
    ], //required
    // ...
]
```

Stack

Renders a repeatable stack of form elements

```
[
    'type' => 'stack', //required
    'sets' => [
        [
                'name' => 'fieldset1',
                'label' => 'Fieldset1',
                'elements' => [
                    [
                        'name' => 'text',
                        'type' => 'text',
                    ],
                    [
                        'name' => 'image',
                        'type' => 'image',
                    ],
                ]
            ],
            [
                'name' => 'fieldset2',
                'label' => 'Fieldset2',
                'elements' => [
                    [
                        'name' => 'wysiwyg',
                        'type' => 'wysiwyg',
                    ],
                ]
            ]
    ], //required
```

```
    // ...
]
```

Switcher

Switcher is a visual component (an On/Off-Switch). Internally in works like the checkbox element.

```
[
    'type' => 'switcher', //required
    'checkedValue' => '', //optional, default string 'true'
    'uncheckedValue' => '', //optional, default string 'false'
    // ...
]
```

Text

Renders a `<input type="text">` element with an attached strlen validator

```
[
    'type' => 'text', //required
    'minLength' => 0, //optional, default 0
    'maxLength' => 0, //optional, default 524288
    // ...
]
```

Textarea

Renders a `<textarea>` element with an attached strlen validator

```
[
    'type' => 'textarea', //required
    'minLength' => 0, //optional, default 0
    'maxLength' => 0, //optional, default PHP_MAX_INT
    'rows' => 0, //optional, default 5
    // ...
]
```

Wysiwyg

Renders a `tinymce` wysiwyg editor

```
[
    'type' => 'wysiwyg', //required
    'editorOptions' => [] //optional => you can pass tinymce editor options as array
    // ...
]
```

YouTube

The YouTube Element provides the functionality to parse youtube links and storing the youtube id

```
[
    'type' => 'youtube', //required
    // ...
]
```

PreConfigured Elements

PreConfigured Elements are based on Base Elements but have already values and/or options pre configured

ActiveSwitcher

Short version of an online/offline switcher

```
[
    'type' => 'activeSwitcher', //required
    // ...
]
```

This element is an shortcut of:

```
[
    'type' => 'switcher',
    'checkedValue' => 'active',
    'uncheckedValue' => 'inactive',
]
```

Country

Based on a Select Element - a DropDown with all Countries

```
[
    'type' => 'country', //required
    // ...
]
```

This element is an shortcut of:

```
[
    'type' => 'select',
    'values' => [
        'AF' => 'Afghanistan',
        //...
        'AT' => 'Austria',
        //...
        'ZW' => 'Zimbabwe',

    ],
]
```

Image

Through the image element images can be selected/uploaded.

```
[
    'type' => 'image', //required
    'categorySelection' => '', //optional - select only from given category, default:␣
↪*
    // ...
]
```

This element is an shortcut of:

```
[
    'type' => 'media',
    'typeSelection' => 'images',
]
```

OnlineSwitcher

Short version of an online/offline switcher

```
[
    'type' => 'onlineSwitcher', //required
    // ...
]
```

This element is an shortcut of:

```
[
    'type' => 'switcher',
    'checkedValue' => 'online',
    'uncheckedValue' => 'offline',
]
```

CHAPTER 10

Assets

For (nearly) every website or web application you will need some kind of assets (like css, javascripts or images). The kiwi asset system is helping to symlink/copy assets to the right place (inside a modular directory structure) and provides a helper to generate urls to selected assets.

Do symlink/copy your assets just run:

```
./bin/fruit assets
```

Note: **For Windows users:** the `assets` command tries to **symlink** folders from modules' `assets` folders. This is possible on Windows 7 and newer and requires to be run with administrative permissions. Otherwise use the command's additional `--copy` option to copy files instead:

```
$ bin/fruit assets --copy
```

Note: The asset system doesn't provide any functionality for compressing/compiling of assets but helps to locate asset folders and copy/symlink them. What kind of frontend setup (grunt, gulp, sass, less etc) you are using is completely up to you (and also your responsibility to setup correctly.

The default config of the asset system should look similar to this:

```
[asset_url] => null
[asset_path] => null
[prepend_commit] => false
[prepend_base_path] => true
[directories] =>
    [admin42] =>
        [target] => 'admin/admin42'
        [source] => 'vendor/fruit42/admin42/assets/dist/'
    [media42] =>
        [target] => 'admin/media42'
        [source] => 'vendor/fruit42/media42/assets/dist/'
```

```
    [frontend42] =>
        [target] => 'admin/frontend42'
        [source] => 'vendor/fruit42/frontend42/assets/dist/'
    [application] =>
        [target] => 'application'
        [source] => 'module/application/assets/dist/'
```

Tip: With `./bin/fruit config assets` you can check your local asset config.

Directories

Directories is a config list of target sources which will be symlinked/copied to the target folder through this command:

```
./bin/fruit config assets
```

Prepend Commit

To avoid browser caching issues you can enable `prepend_commit` which will prepend a `v-<commit hash>` to the asset url. Per default the short version of your git commit hash will be used. If git is unavailable a random hash will be used instead.

Note: The hash will be created when running `composer update` or `composer install`. Therefore it is not needed to have git installed on the production servers.

Note: You may have to adopt your rewrite rules on your server.

Examples

See *View Helpers* to see how to add URLs to assets in views.

The following asset URLs will be generated depending on the assets configuration.

Without `prepend_commit`:

```
/app/assets/admin/admin42/css/styles.min.css
```

With `prepend_commit` enabled:

```
/app/assets/v-54fah56/admin/admin42/css/styles.min.css
```

With `asset_url` set to e.g. `https://static.my-project.com/` and `prepend_commit` enabled:

```
https://static.your-project.com/app/assets/v-54fah56/admin/admin42/css/styles.min.css
```

CHAPTER 11

View Helpers

In view scripts there's usually need for functions aside from trivial control structures and displaying variables (like e.g. formatting, generating links, ...) which View Helpers can provide.

assetUrl

Generates relative or absolute URLs to assets depending on the assets configuration.

```
<?= $this->assetUrl('/css/style.min.css', 'application') ?>
```

Both parameters are optional. The first parameter determines the path to the file (relative from the asset url base or, when a second parameter is given, relative to the `target` parameter). The second parameter declares the key of your asset directory config. See *Assets* for further information.

CHAPTER 12

Whitelabel

kiwi42 allows to whitelabel specific parts of the administration interface to match with the project's identity.

Running `bin/fruit config whitelabel` from the command line will show the currently configured labeling configuration like the following:

```
[title] => 'kiwi42'
[show-topbar-title] => true
[topbar-title] => 'kiw<span class="text-r42">i</span>42'
[logo-icon] => '/admin/admin42/images/logo-icon.png'
[logo-lg] => '/admin/admin42/images/logo-lg.png'
[logo-xs] => '/admin/admin42/images/logo-icon.png'
[logo-xs-dark] => '/admin/admin42/images/logo-lg.png'
[sidebar-bottom-text] => 'kiw<span class="text-r42">i</span>42 &copy; 
↪raum42 OG'
[sidebar-bottom-link] => 'https://raum42.at'
[sidebar-bottom-link-title] => 'raum42 OG'
```