Wordless Documentation

Release 4.0.0

weLaika

Jun 03, 2021

Installation


Wordless

CHAPTER 1

Introduction

Wordless is an opinionated WordPress plugin + starter theme that dramatically speeds up and enhances your custom theme creation. Some of its features are:

- A structured, organized and clean theme organization
- **Scaffold** a new theme directly within wp-cli
- Write PHP templates with **PUG** templating language
- Write CSS stylesheets using the awesome **SCSS** syntax
- Write Javascript logic in **ES2015**
- A growing set of handy and documented PHP **helper functions** ready to be used within your views
- Preconfigured support to **MailHog** mail-catcher.
- Development workflow backed by **WebPack**, BrowserSync (with live reload), WP-CLI, Yarn. All the standards you already know, all the customizations you may need.

Wordless is a micro-framework for custom themes development. Thus is a product intended for developers.

A compiled Wordless theme will run on any standard Wordpress installation.

Wordless does not alter any core functionality, thus it is compatible with reasonably any generic plugin.

CHAPTER 2

Table of Contents

2.1 Prerequisites

1. Node. Depending on the Wordless version you'll need a specific Node version. Using NVM is recommended and the theme will be preconfigured with a `.nvmrc` file.
2. WP-CLI `brew install wp-cli`
3. `Yarn`[1] globally installed: `npm install -g yarn`
4. Test-related requirements (skip if you won't use the test suite)
    1. Composer[2] `brew install composer`
    2. Selenium `brew install selenium-server-standalone`
    3. Chrome Driver `brew install chromedriver`
5. If you'd like to enable the mail-catcher while developing, install MailHog[3]. On MacOS this is as simple as `brew install mailhog`. Wordless will do the rest.

See also:

MailHog for documentation about how to use MailHog in Wordless

See also:

Nodejs for documentation about how to use nodejs in Wordless

2.2 Wordless gem

The quickest CLI tool to setup a new WordPress locally. Wordless ready.

[1] https://www.npmjs.com/package/yarn
[2] https://getcomposer.org/
[3] https://github.com/mailhog/MailHog

Navigate to https://github.com/welaika/wordless_gem to discover the tool and set up all you need for local development. In less than 2 minutes ;)

The quickstart, given the prerequisites, is:

```
gem install wordless
cd MY_DEV_FOLDER
wordless new THEME_NAME [--db-user=DB_USER --db-password=DB_PASSWORD]
```

When `--db-user` is omitted it will default to `admin`, when `db-password` is omitted it will default to a blank password.

If you already have a WordPress installation and just want to add Wordless to it, read the following paragraph.

2.3 Manual installation

At the end of the installation process you will have

- a plugin - almost invisible: no backend page, just custom `wp-cli` commands
- a theme - where you will do all of the work

2.3.1 Additional prerequisites

1. WordPress installed and configured as per official documentation

---

Note: We don't know if you have a local apache {M,L,W}AMPP instance or whatever in order to perform the official installation process. Keep in mind that Wordless' flow does not need any external web server, since it will use the wp server command to serve your wordpress.

See also:

Development environment

2.3.2 Steps

Note: We consider that you have WordPress already up and running and you are in the project's root directory in your terminal.

---

1. Install and activate the wordpress plugin

   ```
   wp plugin install --activate wordless
   ```

2. Scaffold a new theme

   ```
   wp wordless theme create mybrandnewtheme
   ```

See also:

CLI for info about wp-cli integration

1. Enter theme directory

```
cd wp-content/themes/mybrandnewtheme
```

2. Setup all the things

```
yarn setup
```

3. Start the server - and the magic

```
yarn run server
```

Webpack, php server and your browser will automatically come up and serve your needs :)

2.4 Theme anatomy

This is a typical Wordless theme directory structure (as per latest release):

```
vanilla_theme/
├── .vscode
│   ├── launch.json
│   └── settings.json
├── config
│   ├── initializers
│   └── locales
├── dist
│   ├── fonts
│   ├── images
│   ├── javascripts
│   ├── stylesheets
│   └── README.md
├── helpers
│   ├── ComponentPost.php
│   └── README.mdown
├── src
│   ├── images
│   ├── javascripts
│   ├── stylesheets
│   └── main.js
├── tests
│   ├── _data
│   ├── _output
│   ├── _support
│   ├── acceptance
│   ├── functional
│   ├── unit
│   ├── wpunit
│   ├── acceptance.suite.yml
│   ├── functional.suite.yml
│   ├── unit.suite.yml
│   └── wpunit.suite.yml
├── tmp
│   └── .gitkeep
├── views
│   ├── components
│   ├── layouts
│   ├── partials
```

(continues on next page)

(continued from previous page)

```
│   └── posts
├── .browserslistrc
├── .env
├── .env.ci
├── .env.testing
├── .eslintrc.json
├── .gitignore
├── .nvmrc
├── .stylelintignore
├── .stylelintrc.json
├── Procfile
├── Procfile.testing
├── codeception.ci.yml
├── codeception.dist.yml
├── composer.json
├── index.php
├── package.json
├── release.txt
├── screenshot.png
├── style.css
├── webpack.config.js
├── webpack.env.js
└── yarn.lock
```

Next chapters will deepen into each part of the structure, in reasoned order.

2.5 Routing

index.php file in theme's root serves as a router to all the theme views.

Listing 1: index.php

```
21 if (is_single()) {
22   render_template("templates/single");
23 } else if (is_front_page()){
24   render_static("templates/static");
25 }else if (is_archive()) {
26   render_template("templates/archive");
27 } else {
28   render_template("templates/404");
29 }
```

As you can see, you first determine the type of the page using WordPress conditional tags, and then delegate the rendering to an individual view.

While `index.php` is the entry point of any WordPress theme, as it is called/required by it, the `render_template()` function is where we connect WordPress core with Wordless' powerups.

The next chapter is all about rendering.

See also:

Using Page Template Wordpress' feature inside Wordless

2.6 Rendering PUG

In Wordless templates are written in PUG. The plugin incorporates and loads for you the excellent `pug-php/pug` library, that's a complete PHP rewrite of the original javascript PUG.

See also:

PHUG section @ *Code compilation*

Rendering a view is mainly achieved through calling `render_template()` method inside the `index.php`, as mentioned in *Routing*.

The vanilla theme is shipped with an example scaffolding, but you can scaffold as you wish, **as long as you retain the ''views/'' folder**.

This is the proposed scaffold for views:

```
vanilla_theme/views/
├── components
├── layouts
├── partials
└── templates
```

2.6.1 Layouts

`layouts/` is where to put the outer part of your templates; usually a layout represents the always repeated (non content) parts of your template such as `<head>`, the main header, the navigation, the footer.

Vanilla theme ships this `default.pug` layout:

Listing 2: views/layouts/default.pug

```
doctype html
html
  head
    include /layouts/head.pug
  body
    .page-wrapper
      header.site-header
        include /layouts/header.pug

      section.site-content
        block yield

      footer.site-footer
        include /layouts/footer.pug

    // jQuery and application.js is loaded by default with wp_footer() function. See
↪config/initializers/default_hooks.php for details
    - wp_footer()
```

Please, do ignore the function of the `include` keyword at the moment. Will exmplain it in the "Partial" paragraph of this same chapter.

We can note what it brings in:

- doctype declaretion
- `<html>` tag

- <head> tag
- site header
- site content
- site footer

The most important thing to focus on now is the `block yield`. This is where a **template** will fill the layout with content. We're going to cover how this happens in the next paragraph. But it's important to understand this concept: **a layout**, by convention, **is not meant to be directly rendered**, instead each **template** is in charge to declare what layout it want to use.

2.6.2 Templates

Templates are what you will directly render.

The helper function used to render a template is `render_template()` and it is intended to be used mainly into `index.php` file. Here is its signature:

Listing 3: render_helper.php

```
61      /**
62      * Renders a template and its contained plartials. Accepts
63      * a list of locals variables which will be available inside
64      * the code of the template
65      *
66      * @param  string $name   The template filenames
67      *
68      * @param  array  $locals An associative array. Keys will be variables'
69      *                        names and values will be variable values inside
70      *                        the template
71      *
72      * @param boolean $static If `true` static rendering of PUG templates will
73      *                        be activated.
74      *
75      */
76      function render_template($name, $locals = array(), $static = false) {
```

For example:

```
<?php
render_template('posts/single.pug')
```

will search for the PUG template `views/posts/single.pug` relative to the theme folder.

You can also pass an array of variables to your template, by setting the `$locals` parameter:

```
<?php
render_template('posts/single.pug', ['foo' => 'bar'])
```

The `$locals` array will be auto-`extract()`-ed inside the required view, so you can use them into the template. E.g: inside `views/posts/single.pug`

```
h1= $foo
```

Rendering a template for a webpage would involve to write a lot of boilerplate code (<html> tag, <head> and so on). This is where our **layouts** come handy. Let's see how a simple template is structured:

Listing 4: view/templates/single.pug

```
extends /layouts/default.pug

block yield
  h2 Post Details
  - the_post()
  include /partials/post.pug
```

- with `extends /layouts/default.pug` the template is declaring which layout it is going to extend
- `block yield` is the same declaration we found into `views/templates/default.pug` and it's telling to the chosen template: "Hey, template! You must inject all the below code into your *block* named `yield`

The result, just to easily imagine it out, would be:

```
doctype html
html
  head
    include /layouts/head.pug
  body
    .page-wrapper
      header.site-header
        include /layouts/header.pug

      section.site-content
        h2 Post Details // This is where the layout declared `block yield`
        - the_post()
        include /partials/post.pug

      footer.site-footer
        include /layouts/footer.pug

    // jQuery and application.js is loaded by default with wp_footer() function. See
↪config/initializers/default_hooks.php for details
    - wp_footer()
```

Obviously this composition is transparently handled by PUG.

So we have a structured template now; we're ready to undestand **partials** and how to use them with the `include` keyword.

Note: You will notice that `extend` and `include` argument always strarts with a trailing slash. This is the PUG convention to search for files into the `views/` folder, which is configured as the "root" search folder.

2.6.3 Partials

Partials are optional but powerful; they're a tool for split your bigger into smaller and more managable chunks.

They are included into parent files using the `include` keyword.

For example this template

Listing 5: view/templates/single.pug

```
extends /layouts/default.pug

block yield
  h2 Post Details
  - the_post()
  include /partials/post.pug
```

having this partial

Listing 6: view/templates/single.pug

```
post
  header
    h3!= link_to(get_the_title(), get_permalink())
  content!= get_the_filtered_content()
```

will produce

```
doctype html
html
  head
    include /layouts/head.pug
  body
    .page-wrapper
      header.site-header
        include /layouts/header.pug

      section.site-content
        h2 Post Details // This is where the layout declared `block yield`
        - the_post()
        post
          header
            h3!= link_to(get_the_title(), get_permalink())
          content!= get_the_filtered_content()

      footer.site-footer
        include /layouts/footer.pug

    // jQuery and application.js is loaded by default with wp_footer() function. See
↪config/initializers/default_hooks.php for details
    - wp_footer()
```

Straight. The included partial will share the scope with the including template.

There is a specific type of partial supported by Wordless by default: **components**. We'll see how to use them in the next paragraph.

2.6.4 Components

Components are a special flavour of partials that can receive scoped variables. **Components are functions**: given the same parameters they will always render the same piece of HTML. This way you can **re-use** a component in any place of your app, being sure to obtain the same output.

Let's see an example taken from the vanilla theme.

Listing 7: view/templates/archive.pug

```
extends /layouts/default.pug
include /components/post.pug

block yield
  h2 Archive

  ul.archive
    while (have_posts())
      - the_post()

      li
        - $component = new ComponentPost(['post' => get_post()])
        +post($component)
```

Listing 8: view/components/post.pug

```
mixin post($component)
  post
    header
      h3!= link_to($component->post->post_title, get_permalink($component->post))
    content!= get_the_excerpt($component->post)
```

First news: in `view/templates/archive.pug` we have a top `include`; this will not directly produce anything, because `views/components/post.pug` contains only a `mixin` declaration.

The `mixin` is named `post` and takes 1 argument `$component`. A mixin won't produce anything when `include`-ed, but only when invoked. You have really to think of it like a regular PHP function.

By using `include /components/post.pug` we're now able to invoke the mixin using the syntax `+ + mixinName`, thus `+post($arg1)` which is our second and last news about components.

Note: Wordless supports the `component` keyword as an alias to the default `mixin` keyword in PUG templates. This is much more expressive. The counter-effect is that your syntax highligter won't appreciate it that much :)

See also:

PHUG `mixin` documentation @ https://www.phug-lang.com/#mixins

Arguments validation

In the previous example you've seen that the `$component` argument is an instance of `ComponentPost` class. Let's explain what and why it is.

Note: When you write your mixins, you decide what and how many arguments they will require. Validators aren't mandatory, but a useful and poweful tool you're free to use or not.

Visual components, given they accept arguments, are strictly dependent on data passed to them through arguments. This is true in any front-end development stack/scenario/framework.

Since in WordPress you have not *models* and since you'll often rely on custom fields to gather and pass data from the front-end to the DB and vice versa, you have not a "core" way to ensure that you're passing valid objects (or data-structures) to your components to be rendered.

See this example:

```
mixin post($title)
  post
    header
      h3!= "My title is {$title}"

+post('')
```

Will render an `<h3>` tag with `My title is`. This is a trivial example, but receiving wrong data in specific situations could entirely broke your component and thus your view. Not speaking about types.

Here comes into play `\Wordless\Component` class. You can see it in action in our vanilla theme:

Listing 9: view/templates/archive.pug

```
extends /layouts/default.pug
include /components/post.pug

block yield
  h2 Archive

  ul.archive
    while (have_posts())
      - the_post()

      li
        - $component = new ComponentPost(['post' => get_post()])
        +post($component)
```

Where `ComponentPost` is a custom class extending `\Wordless\Component`:

Listing 10: This is a simplified version of helpers/ComponentPost.php in vanilla theme

```
<?php

use Symfony\Component\Validator\Constraints as Assert;

class ComponentPost extends \Wordless\Component {
    public $post;

    public static function loadValidatorMetadata($metadata)
    {
        $metadata->addPropertyConstraint('post', new Assert\Type(WP_Post::class));
    }
}
```

We are using Symphony's Validator; it is already loaded and ready to use, so you can write your component's classes implementing all the validations as per the detailed documentation https://symfony.com/doc/current/reference/constraints.html.

This is how's intended to be used inside Wordless:

- define a class extending `\Wordless\Component`
- declare as many public attributes as your component needs
- instance the object passing arguments as an associative array `$component = new ComponentPost(['post' => get_post()])`

- each key will be automatically cheked to be declared as an attribute into the component and the corresponding attribute will be set to the corresponding value. You can pass arguments only if they are declared into the component class.
- into the component is mandatory to implement a `loadValidatorMetadata` public static function. Inside of it you will write your actual validations. This name was chosen in order to stick with official documentation's naming.
- `$component` will be validated at instantiation time, so you will have an error or a valid object. No doubts.
- passing `$component` as your mixin's argument, inside the mixin you will be able to get its properties as expected: `$component->attribute`.

Revisiting our previous exaple:

```
<?php

use Symfony\Component\Validator\Constraints as Assert;

class ComponentPost extends \Wordless\Component {
    public $title;

    public static function loadValidatorMetadata($metadata)
    {
        $metadata->addPropertyConstraint('post', new Assert\Type('string'));
        $metadata->addPropertyConstraint('post', new Assert\NotBlank());
    }
}
```

```
mixin post($component)
  post
    header
      h3!= "My title is {$component->title}"

- $component = new ComponentPost(['title' => ['My title']]) // Error: not a string
- $component = new ComponentPost(['title' => '']) // Error: is empty
- $component = new ComponentPost(['title' => '']) // Error: is empty
- $component = new ComponentPost(['turtle' => 'My title']) // Error: "turtle"␣
↪undeclared property
+post($component)
```

When a validation error is thrown, an error will be rendered instead of the template. This is true if the `ENVIRONMENT` constant is not set to `production`. If you've declared the environment as production, nothing will happen by default. You can implement your custom action for production using the `wordless_component_validation_exception` action. For more info head to *Filters*

2.7 SCSS and JS

2.7.1 The Fast Way

- write your SCSS in `src/stylesheets/screen.scss`
- write your JS in `src/javascripts/application.js`

and all will automagically work! :)

2.7.2 A real explanation

Wordless has 2 different places where you want to put your assets (javascript, css, images):

- Place all your custom, project related assets into `src/*`
- Since you are backed by Webpack, you can use NPM (`node_modules`) to import new dependencies following a completely standard approach

Custom assets

They must be placed inside `src/javascript/` and `src/stylesheets/` and `src/images/`.

They will be compiled and resulting compilation files will be moved in the corresponding `dist/xxx` folder.

Compilation, naming and other logic is fully handled by webpack.

Images will be optimized by image-minimizer-webpack-plugin. The default setup already translates `url` s inside css/scss files in order to point to images in the right folder.

Take a look to the default `screen.scss` and `application.js` to see usage examples.

See also:

Code compilation

See also:

- Official SCSS guide

node_modules

You can use node modules just as any SO answer teaches you :)

Add any vendor library through YARN with

```
yarn add slick-carousel
```

Then in your Javascript you can do

```
require('slick-carousel');
```

or if the library exports ES6 modules you can do

```
import { export1 } from "module-name";
```

and go on as usual.

2.7.3 Linters

Wordless ships with preconfigured linting of SCSS using Stylelint.

It is configured in `.stylelintrc.json`, you can add exclusion in `.stylelintignore`; all is really standard.

The script `yarn lint` is preconfigured to run lint tasks.

Tip: Code linting could be chained in a build script, e.g.:

Tip: Code linting could be integrated inside a Wordmove hook

Tip: You can force linting on a pre-commit basis integrating Husky in your workflow.

2.8 PHP Helpers

`helpers/*.php` files

Helpers are basically small functions that can be called in your views to help keep your code stay DRY. Create as many helper files and functions as you want and put them in this directory: they will all be required within your views, together with the default Wordless helpers. These are just a small subset of all the 40+ tested and documented helpers Wordless gives you for free:

- `lorem()` - A "lorem ipsum" text and HTML generator
- `pluralize()` - Attempts to pluralize words
- `truncate()`- Truncates a given text after a given length
- `new_post_type()` and `new_taxonomy()` - Help you create custom posts and taxonomy
- `distance_of_time_in_words()` - Reports the approximate distance in time between two dates

Our favourite convention for writing custom helpers is to write 1 file per function and naming both the same way. It will be easier to find with `cmd+p`

Where is my `functions.php` ?

In a Wordless theme the isn't a `functions.php` file. It was too ugly to us to support it. You have simply to consider `helpers/*.php` files as the explosion of your old messy `functions.php` into smaller chunks. And since all the helpers you'll write will be autorequired, defined functions will work exactly the same way you are used to.

2.9 Initializers

`config/initializers/*.php` files

Remember the freaky `functions.php` file, the one where you would drop every bit of code external to the theme views (custom post types, taxonomies, wordpress filters, hooks, you name it?) That was just terrible, right? Well, forget it.

Wordless lets you split your code into many modular initializer files, each one with a specific target:

```
config/initializers
├──── backend.php
├──── custom_gutenberg_acf_blocks.php
├──── custom_post_types.php
├──── default_hooks.php
├──── hooks.php
├──── login_template.php
├──── menus.php
├──── shortcodes.php
├──── thumbnail_sizes.php
```

- **backend**: remove backend components such as widgets, update messages, etc

- **custom_gutenbers_acf_blocks**: Wordless has built-in support to ACF/Gutenberg blocks. Read more at *ACF Gutenberg Blocks*
- **custom_post_types**: well... if you need to manage taxonomies, this is the place to be
- **default_hooks**: these are used by wordless's default behaviours; tweak them only if you know what are you doing
- **login template**: utilities to customize the default WP login screen
- **hooks**: this is intended to be your custom hooks collector
- **menus**: register new WP nav_menus from here
- **shortcodes**: as it says
- **thumbnail_sizes**: if you need custom thumbnail sizes

These are just some file name examples: you can organize them the way you prefer. Each file in this directory will be automatically required by Wordless.

Moreover: each of these files comes already packed with interesting, often used functions and configurations. They are ready to be uncommented. Take youself a tour directly in the code @ https://github.com/welaika/wordless/tree/master/wordless/theme_builder/vanilla_theme/config/initializers

2.10 Locale files

`config/locales` directory

Just drop all of your theme's locale files in this directory. Wordless will take care of calling load_theme_textdomain() for you.

Note: Due to the WordPress localization framework, you need to append our `"wl"` domain when using internationalization. For example, calling `__("News")` without specifying the domain *will not work*.

You'll **have** to add the domain *"wl"* to make it work: `__("News", "wl")`

2.11 Filters

The plugin exposes WordPress filters to let the developer alter specific data.

2.11.1 wordless_pug_configuration

Listing 11: wordless/helpers/pug/wordless_pug_options.php

```php
<?php

class WordlessPugOptions {
    public static function get_options() {
        $wp_debug = defined('WP_DEBUG') ? WP_DEBUG : false;
        return apply_filters( 'wordless_pug_configuration', [
            'expressionLanguage' => 'php',
            'extension' => '.pug',
            'cache' => Wordless::theme_temp_path(),
```

(continues on next page)

(continued from previous page)

```
            'strict' => true,
            'debug' => $wp_debug,
            'enable_profiler' => false,
            'error_reporting' => E_ERROR | E_USER_ERROR,
            'keep_base_name' => true,
            'paths' => [Wordless::theme_views_path()],
            'mixin_keyword' => ['mixin','component'],
        ]);
    }
}
```

Usage example

```
<?php
add_filter('wordless_pug_configuration', 'custom_pug_options', 10, 1);

function custom_pug_options(array $options): array {
    $options['expressionLanguage'] = 'js';

    return $options;
}
```

2.11.2 wordless_acf_gutenberg_blocks_views_path

Listing 12: wordless/helpers/acf_gutenberg_block_helper.php

```
48  function _acf_block_render_callback( $block ) {
49    $slug = str_replace('acf/', '', $block['name']);
50
51    // The filter must return a string, representing a folder relative to `views/`
52    $blocks_folder = apply_filters('wordless_acf_gutenberg_blocks_views_path',
    ↪'blocks/');
53
54    $admin_partial_filename = Wordless::theme_views_path() . "/{$blocks_folder}/admin/
    ↪_{$slug}";
55
56    if (
57      file_exists( "{$admin_partial_filename}.html.pug" ) ||
58      file_exists( "{$admin_partial_filename}.pug" ) ||
59      file_exists( "{$admin_partial_filename}.html.php" ) ||
60      file_exists( "{$admin_partial_filename}.php" )
61    ) {
62        $admin_partial = "{$blocks_folder}/admin/{$slug}";
63    } else {
64        $admin_partial = "{$blocks_folder}/{$slug}";
65    }
```

Usage example

```
<?php
add_filter('wordless_acf_gutenberg_blocks_views_path', 'custom_blocks_path', 10, 1);

function custom_blocks_path(string $path): string {
    return 'custom_path';
}
```

This way Wordless will search for blocks' partials in `views/custom_path/block_name.html.pug` so you can use `render_partial('custom_path/block_name')` to render them in your template.

The default path is `blocks/`.

Note: The path will be always relative to `views/` folder

2.12 Actions

2.12.1 wordless_component_validation_exception

Listing 13: wordless/helpers/component_helper.php

```
52          try {
53              $this->setProperties();
54              $this->validate();
55          } catch (ComponentValidationException $e) {
56              if ( 'production' === ENVIRONMENT ) {
57                  do_action('wordless_component_validation_exception', $e);
58                  // Would be nice to have an exception collector in your callback, e.g.
    ↪ Sentry:
59                  //
60                  // function yourhandler(\Wordless\ComponentValidationException $e) {
61                  //      if ( function_exists( 'wp_sentry_safe' ) ) {
62                  //          wp_sentry_safe( function ( \Sentry\State\HubInterface
    ↪$client ) use ( $e ) {
63                  //                          $client->captureException( $e );
64                  //                      } );
65                  //      }
66                  // }
67                  // add_action('wordless_component_validation_exception', 'yourhandler
    ↪', 10, 1)
68              } else {
69                  render_error('Component validation error', $e->getMessage());
70              }
```

When an object of class `Wordless\Component` fails its validation, it will throw an exception only if `ENVIRONMENT` **is not** `production`. When in `production` nothing will happen, in order to be unobstrusive and not breaking the site to your users. The developer will still see specific excpetion happening.

You can customize the behaviour by adding your action as documented in the code.

What we like to do is to add here a notification to our Sentry account (thanks to https://github.com/stayallive/wp-sentry/ plugin)

2.13 CLI

When a Wordless theme is activated and you are inside project's path, you automatically get an *ad-hoc* WP-CLI plugin.

Typing `wp help` you'll notice a `wordless` subcommand.

All subcommands are self-documented, so you can simply use, e.g.:

```
wp help wordless theme upgrade
```

to get the documentation.

2.14 Test Suite

The default Wordless theme is shipped with preconfigured test suite.

The test suite is implemented using the awesome WPBrowser and thus Codeception.

Note: By default Wordless is configured to run **acceptance** (aka **integration** or **e2e** or **browser**) test suite alone. If you'd like to run *functional* or *unit* suites, you'll simply have to update the `yarn test` script accordingly in `package.json` file.

2.14.1 Quick start

Add tests to the `tests/acceptance/WPFirstCest.php` file or write your own file in the same folder.

To run acceptance test suite you have to start the test server in one terminal

```
yarn test:server
```

and in another terminal let's actually run tests:

```
yarn test
```

While `test` will simply run **acceptance** test suite, `test:server` is a variant of the default `server` task which load different `Procfile` and `.env` files.

2.14.2 Where are test configurations?

- `test/` folder. This is where your test suites lay.
- PHP dependencies declared in `composer.json` file shipped within the theme. This will create a `/vendor` folder inside the theme whilist `yarn setup` task
- custom `wp-config.php`. This will be helpful to autodymagically (automatically, dynamically, magically; just in case you were wondering) switch from development to test database whilist test suite execution
- 2 test related node scripts: `yarn test:server` and `yarn test`. Obviously declared inside `package.json`
- a test database on your local machine called `$THEME_NAME_test` (where `$THEME_NAME` is the chosen name during Wordless' installation process) is created whilist `yarn setup` task
- *ad hoc* `Procfile.testing`, `.env.testing` and `.env.ci`
- ready-to-go `.gitlab-ci.yml` file into the project root

Note: `vendor/` folders are ignored in `.gitignore` by default

2.14.3 How should I write tests?

This documentation is not intended to giude you thourgh testing concepts nor on Codeception's syntax. You can already find great documentation and I advice you to start from

- https://wpbrowser.wptestkit.dev/modules/wpwebdriver
- https://wpbrowser.wptestkit.dev/modules/wpbrowser
- https://wpbrowser.wptestkit.dev/modules/wpdb

where you will find Wordpress specific methods and links to base Codeception's methods all in one place.

Factory template

The only thing Wordless actually adds to the default WPBrowser's setup is a `FactoryHelper` class, which is intended to create factory methods and which already integrates Faker.

Take a look at its `haveOnePost()` method to understand the simple concept behind the factory.

2.14.4 CI

We ship default configuration for GitLab by putting a `.gitlab-ci.yml` file in you project's root folder.

That is configured to run out-of-the-box. And if you use other CI's products you can use it as a starting point for your own configuration and then delete it without any regard :)

2.14.5 Troubleshooting

- **yarn setup -> Error: Error establishing a database connection.**

  Check your db's username & password in the `wp-config.php`

- **yarn test -> Db: SQLSTATE[HY000] [2054] The server requested authentication method unknown to the client while creating PDO connection**

  Check your db's username & password in `.env.testing`, inside the theme's folder

- **yarn test -> Could not find, or could not parse, the original site URL; you can set the "originalUrl" parameter in the module configuration to skip this step and fix this error.**

  The command `yarn test:db:snapshot` can be useful.

- **yarn test -> [ConnectionException] Can't connect to Webdriver at http://localhost:4444/wd/hub. Please make sure that Selenium Server or PhantomJS is running.**

  Check if you are running `yarn test:server` in another terminal .

2.15 Intro

Since Wordless uses Webpack, we have to manage build and distribution strategies for dev and staging/production.

The source asset code is placed in `src/{javascripts|stylesheets|images}`, while built/optimized code is placed - automatically by Webpack - in `dist/{javascripts|stylesheets|images}`

See also:

JS and SCSS

We offer standard approaches for both environments. They are handled - as expected - through `package.json` 's scripts[1]:

Listing 14: package.json

```
  "scripts": {
    "server": "npx nf start",
    "build:dev": "webpack --env NODE_ENV=development",
    "build:prod": "yarn sign-release && webpack --env NODE_ENV=production",
    "clean:js": "rimraf dist/javascripts/*",
    "clean:css": "rimraf dist/stylesheets/*",
    "clean:images": "rimraf dist/images/*",
    "clean:dist": "yarn clean:js && yarn clean:css && yarn clean:images",
    "sign-release": "git rev-parse HEAD | cut -c 1-8 > release.txt",
    "lint": "yarn lint:sass",
    "lint:sass": "npx stylelint 'src/stylesheets/**/*.scss'",
    "test:db:create": "WP_ENV=test wp db create",
```

It is expected - but it's still up to you - that before every build you will clean the compiled files. `yarn clean:dist` will do the cleanup.

2.16 Development build

```
yarn clean:dist && yarn build:dev
```

Note: Most of the time you'll be working using the built-in development server through `yarn server`, but invoking a build arbitrarily is often useful.

2.17 Production build

```
yarn clean:dist && yarn build:prod
```

Production build will essentially:

- enable Webpack's production mode
- do not produce source maps for CSS
- do minimize assets

Note: By default the production build won't produce source-maps for JS.

You can easily change this behaviour updating `const needSourceMap = (env.DEBUG == 'true');` to `const needSourceMap = true;` in `webpack.env.js`

[1] https://docs.npmjs.com/files/package.json#scripts

2.17.1 Release signature

You notice that `build:prod` script will invoke `sign-release` too. The latter will write the SHA of the current GiT commit into the `release.txt` file in the root of the theme.

You can easily disable this behaviour if you'd like to.

`release.txt` is implemented to have a reference of the code version deployed in production and to integrate external services that should requires release versioning (for us in Sentry).

2.17.2 PHUG optimizer

When performance is a must, PHUG ships a built-in *Optimizer*. You can read about it in the phug documentation:

> The Optimizer is a tool that avoids loading the Phug engine if a file is available in the cache. On the other hand, it does not allow to change the adapter or user post-render events.

Wordless supports enabling this important optimization by setting an environment variable (in any way your system supports it) or a global constant to be defined in `wp-config.php`. Let's see this Wordless internal code snippet:

Listing 15: render_helper.php

```
                    } elseif ( defined('BYPASS_STATIC') ) {
                        $bypass_static = var_export(BYPASS_STATIC, true); // constant␣
↪could be a boolean so we uniform to a string representation
                    } else {
                        $bypass_static = 'false'; // default value
                    }

                    if ( in_array( $env, array('staging', 'production') ) ) {
                        if (true === $static && 'false' == strtolower($bypass_
↪static)) {
```

where we search for `ENVIRONMENT` and thus we'll activate PHUG's `Optimizer` if the value is either `production` or `staging`.

Note: Arbitrary values are not supported.

The simplest approach is to to define a constant inside `wp-config.php`.

Listing 16: wp-config.php

```
<?php
// [...]
define('ENVIRONMENT', 'production');
// [...]
```

2.18 Deploy

Wordless is agnostic about the deploy strategy. Our favourite product for deploying WordPress is Wordmove.

By the way keep in mind that once you have built you theme with `yarn build:prod`, the theme is ready to be simply copied to the production server. The only requirement is to have Wordless plugin activated.

2.19 Intro

Here are the stack components of Wordless' development workflow:

- WordPress plugin
- A theme with a convenient default scaffold
- Webpack
- WP-CLI

2.20 Nodejs

Nodejs is used for all the front-end build chain.

You need to have Node installed on your machine. The setup is not covered by this documentation.

2.20.1 Version

Each release of Wordless is bound to a node version. It is declared inside `package.json`.

Listing 17: pakage.json

```
10  "engines": {
11    "node": "14.15.3"
12  },
```

Wordless is tested with the enforced nodejs version and the shipped `yarn.lock` file. You're free to change version as you wish, but you'll be on your own managing all the dependancies.

2.20.2 NVM

Using NVM is **strongly recommended**.

In a Wordless theme you'll find an `.nvmrc` file; you can use NVM node version manager to easily switch to the right node version.

Installing NVM is as simple as

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.36.0/install.sh |␣
↪bash
```

but you can read more at https://github.com/nvm-sh/nvm#install–update-script.

Note: v0.36.0 is the most recent version at time of writing

Once set up, you can install the required node version

```
nvm install x.x.x
```

where `x.x.x` is the version reported in previous paragraph. Then, once you'll be ready to work, use it with within your theme

```
nvm use
```

2.21 Development environment

Starting by saying that with a

```
yarn run server
```

you should be up and running, let's see in depth what happens behind the scenes.

2.21.1 YARN

`yarn run` (or simply `yarn scriptName`) will search for a `scripts` section inside your `package.json` file and will execute the matched script.

Listing 18: package.json

```
  "scripts": {
    "server": "npx nf start",
    "build:dev": "webpack --env NODE_ENV=development",
    "build:prod": "yarn sign-release && webpack --env NODE_ENV=production",
    "clean:js": "rimraf dist/javascripts/*",
    "clean:css": "rimraf dist/stylesheets/*",
    "clean:images": "rimraf dist/images/*",
    "clean:dist": "yarn clean:js && yarn clean:css && yarn clean:images",
```

`yarn server` will run `nf start`, where `nf` is the Node Foreman executable.

2.21.2 Foreman

Node Foreman (`nf`) could do complex things, but Wordless uses it only to be able to launch multiple processes when `server` is fired.

Listing 19: Procfile

```
wp: wp server --host=0.0.0.0
webpack: npx webpack --watch --progress --color --env NODE_ENV=development
mailhog: mailhog
```

As you can see, each line has a simple named command. Each command will be launched and *foreman* will:

- run all the listed processes
- collect all STDOUTs from processes and print theme as one - with fancyness
- when stopped (CTRL-C) it will stop all of the processes

2.21.3 wp server

Launched by `nf`. Is a default *WP-CLI* command.

We are invoking it within a theme directory, but it will climb up directories until it finds a `wp-config.php` file, then it will start a PHP server on its default port (8080) and on the `127.0.0.1` address as per our config.

Note: You can directly reach `http://127.0.0.1:8080` in you browser in order to reach wordpress, bypassing all the webpack *things* we're going to show below.

2.21.4 BrowserSync

The only relevant **Webpack** part in this section is BrowserSync. It will start a web server at address `127.0.0.1` on port 3000. This is where your browser will automatically go once launched.

Listing 20: webpack.config.js

```
  plugins: [
    new BrowserSyncPlugin({
      host: "127.0.0.1",
      port: 3000,
      proxy: {
        target: "http://127.0.0.1:8080"
      },
      watchOptions: {
        ignoreInitial: true
      },
      files: [
        './views/**/*.pug',
        './views/**/*.php',
        './helpers/**/*.php'
      ]
```

As you can see from the configuration, web requests will be proxy-ed to the underlying `wp server`.

Since *BrowserSync* is invoked through a Webpack plugin (browser-sync-webpack-plugin) we will benefit from automatic **browser autoreloading** when assets are recompiled by Webpack itself.

The `files` option is there because `.pug` files are not compiled by webpack, so we force watching those files too, thus calling autoreload on template changes too.

See also:

Code compilation for other Webpack default configurations

Note: *BrowserSync*'s UI will be reachable at `http://127.0.0.1:3001` as per default configuration.

Warning: If you will develop with the WordPress backend in a tab, *BrowserSync* will ignorantly reload that tab as well (all tabs opened on port 3000 actually). This could slow down your server. We advise to use the WordPress backend using port 8080 and thus bypassing *BrowserSync*.

2.21.5 MailHog

MailHog is an email testing tool for developers:

- Configure your application to use MailHog for SMTP delivery
- View messages in the web UI, or retrieve them with the JSON API
- Optionally release messages to real SMTP servers for delivery

Wordless is configured to use it by default, so you can test mailouts from your site, from WordPress and from your forms.

The UI will be at http://localhost:8025 as per default configuration.

When you spawn `yarn server`, you'll have an environment variable exported thanks to the `.env` file:

Listing 21: .env

```
MAILHOG=true
```

This will trigger the `smtp.php` initializer:

Listing 22: config/initializers/smtp.php

```
<?php

add_action( 'phpmailer_init', 'wl_phpmailer_init' );
function wl_phpmailer_init( PHPMailer $phpmailer ) {
    $mailhog = getenv('MAILHOG');

    if ($mailhog !== "true")
      return false;

    $phpmailer->IsSMTP();
    $phpmailer->Host = 'localhost';
    $phpmailer->Port = 1025;
    // $phpmailer->SMTPAuth = true;
    // $phpmailer->Username = 'user';
    // $phpmailer->Password = 'password';
    // $phpmailer->SMTPSecure = 'ssl'; // enable if required, 'tls' is another␣
↪possible value
}
```

2.21.6 Debug in VSCode

We ship a `.vscode/launch.json` in theme's root which is preconfigured to launch debugger for XDebug and for JS (both Chrome and FireFox). In order to use these configuration you'll need to install some plugins in the editor:

- Debugger for Chrome
- Debugger for Firefox
- PHP Debug

Note: You may need to move `.vscode/launch.json` in another location if you are not opening the theme's folder as workspace in VSCode (maybe you prefere to open all the WordPress installation? Don't know...). It's up to you to use it as you need it.

2.22 Code compilation

First things first: **using "alternative" languages is not a constraint**. Wordless's scaffolded theme uses the following languages by default:

- **PHUG** for views as an alternative to PHP+HTML
- **ES2015** transpiled to JS using Babel
- **SCSS** for CSS

You could decide to use *plain* languages, just by renaming (and rewriting) your files.

Wordless functions which require filenames as arguments, such as

```
<?php

render_partial("posts/post")

// or

javascript_url("application")
```

will always require extension-less names and they will find your files whatever extension they have.

See also:

PHUG paragraph @ *Using plain PHP templates*

Anyway we think that the default languages are **powerful, more productive, more pleasant to read and to write**.

Add the fact that wordless will take care of all compilation tasks, giving you focus on writing: we think this is a win-win scenario.

2.22.1 PHUG

Pug is a robust, elegant, feature-rich template engine for Node.js. Here we use a terrific PHP port of the language: Phug. You can find huge documentation on the official site https://www.phug-lang.com/, where you can also find a neat live playground (click on the "Try Phug" menu item).

It comes from the JS world, so most front-end programmers should be familiar with it, but it is also very similar to other template languages such as SLIM and HAML (old!)

We love it because it is concise, clear, tidy and clean.

Listing 23: A snippet of a minimal WP template

```
h2 Post Details
- the_post()
.post
  header
    h3!= link_to(get_the_title(), get_permalink())
  content!= get_the_content()
```

Certainly, becoming fluent in PUG usage could have a not-so-flat learning curve, but starting from the basics shuold be affordable and the reward is high.

Who compiles PUG?

When a `.pug` template is loaded, the wordless plugin will automatically compile (and cache) it. As far as you have the plugin activated you are ok.

Important: By default, you have nothing to do to deploy in production, but if performance is crucial in your project, then you can optimize. See *PHUG optimizer* for more informations.

2.22.2 JS and SCSS

Here we are in the **Webpack** domain; from the compilation point of view there is nothing Wordless-specific but the file path configuration.

Configuration is pretty standard, so it's up to you to read Webpack's documentation. Let's see how paths are configured in `webpack.config.js`.

Paths

Paths are based on the Wordless scaffold. Variables are defined at the top:

Listing 24: webpack.config.js

```
const srcDir = path.resolve(__dirname, 'src');
const dstDir = path.resolve(__dirname, 'dist');
const javascriptsDstPath = path.join(dstDir, '/javascripts');
const _stylesheetsDstPath = path.join(dstDir, '/stylesheets');
```

and are used by the `entry` and `output` configurations:

Listing 25: webpack.config.js

```
  return {
    mode: envOptions.mode,

    bail: envOptions.mode == 'production' ? true : false,

    entry: entries.reduce((object, current) => {
      object[current] = path.join(srcDir, `${current}.js`);
      return object;
    }, {}),
```

CSS will be extracted from the bundle by the standard mini-css-extract-plugin

Listing 26: webpack.config.js

```
                    },
                ],
            },
```

Inclusion of compiled files

Wrapping up: the resulting files will be

- `dist/javascripts/application.js`
- `dist/stylesheets/screen.css`

As far as those files remain *as-is*, the theme will automatically load them.

If you want to edit names, you have to edit the WordPress asset enqueue configurations:

Listing 27: config/initializers/default_hooks.php

```php
<?php

// This function include main.css in wp_head() function

function enqueue_stylesheets() {
  wp_register_style("main", stylesheet_url("main"), [], false, 'all');
  wp_enqueue_style("main");
}

add_action('wp_enqueue_scripts', 'enqueue_stylesheets');

// This function include jquery and main.js in wp_footer() function

function enqueue_javascripts() {
  wp_enqueue_script("jquery");
  wp_register_script("main", javascript_url("main"), [], false, true);
  wp_enqueue_script("main");
}

add_action('wp_enqueue_scripts', 'enqueue_javascripts');

// Load theme supports
// See http://developer.wordpress.org/reference/functions/add_theme_support/
// for more theme supports you'd like to add. `reponsive-embeds` is on by
// default.
function wordless_theme_supports() {
  add_theme_support('responsive-embeds');
}
add_action('after_setup_theme', 'wordless_theme_supports');
```

Note: The `stylesheet_url` and `javascript_url` Wordless' helpers will search for a file named as per the passed parameter inside the default paths, so if you use default paths and custom file naming, you'll be ok, but if you change the path you'll have to supply it using other WordPress functions.

See also:

stylesheet_url signature

javascript_url signature

Multiple "entries"

"Entries" in the WebPack world means JS files (please, let me say that!).

Wordless is configured to produce a new bundle for each entry and by default the only entry is `main`

Listing 28: main.js

```
require('./javascripts/application.js');
require('./stylesheets/screen.scss');
```

As we've already said having an *entry* which requires both JS and SCSS, will produce 2 separate files with the same name and different extension.

Add another *entry* and producing new bundles is as easy as

- create a new file
- write something in it, should it be a `require` for a SCSS file or a piece of JS logic
- add the *entry* to webpack config

```
const entries = ['main', 'backend']
```

- include somewhere in your theme. For example in the WP's asset queue in `default_hooks.php`

```
function enqueue_stylesheets() {
    wp_register_style("main", stylesheet_url("main"), [], false, 'all');
    wp_register_style("backend", stylesheet_url("backend"), [], false, 'all');
    wp_enqueue_style("main");
    wp_enqueue_style("backend");
}

function enqueue_javascripts() {
    wp_enqueue_script("jquery");
    wp_register_script("main", javascript_url("main"), [], false, true);
    wp_register_script("backend", javascript_url("backend"), [], false, true);
    wp_enqueue_script("main");
    wp_enqueue_script("backend");
}
```

or add it anywhere in your templates:

```
header
    = stylesheet_link_tag('backend')
footer
    = javascript_include_tag('backend')
```

Browserslist

At theme's root you'll find the .browserlistsrc file.

By default it's used by Babel and Core-js3 to understand how to polifill your ES2015 code. You can understand more about our default configuration reading Babel docs at https://babeljs.io/docs/en/babel-preset-env#browserslist-integration

Stylelint

We use Stylelint to lint SCSS and to enforce some practices. Nothing goes out of a standard setup. By the way some spotlights:

- configuration is in `.stylelintrc.json` file

- you have a blank `.stylelintignore` file if you may need
- `yarn lint` will launch the lint process
- if you use VS Code to write, we ship `.vscode/settings.json` in theme's root, which disables the built-in linters as per stylelint plugin instructions. You may need to move those configurations based on the folder from which you start the editor.

2.23 Static rendering

Static rendering is a built-in feature shipped since Wordless 5. It allows you to statically compile a template into HTML and serve it. Successive rendering requests will directly serve the static HTML if present.

You can compile any template into static HTML simply by using the `render_static()` function in place of any `render_template()` PHP function or any PUG's `include` into you views.

This way you have control on having a completely static template or just some partial contents; so you can isolate and make static a specific partial with heavy queries, or the whole page.

This is the definition of `render_static()` function:

Listing 29: render_helper.php

```
163     /**
164      * Wraps render_template() function activating the static rendering strategy
165      *
166      * @param string $name Template path relative to +views+ directory
167      * @param array $locals Associative array of variable that will be scoped into␣
    ↪the template
168      * @return void
169      */
170     function render_static($name, $locals = array()) {
```

Warning: Using static rendering could lead to undesired effects by design (not specifically with Wordless). Be sure to know what you're doing. It's not alwas just a matter to be faster.

2.23.1 Static template example

Given this into `index.php`

```
if (is_front_page()){
  render_static("templates/static");
}
```

and given this `views/templates/static.pug`

```
extends /layouts/default.pug

block yield
  h2 Archive (static example)

  ul.archive
    while (have_posts())
      - the_post()
```

(continues on next page)

(continued from previous page)

```
      li
        include /partials/post.pug
```

visiting your home page will produce a static HTML into theme's `tmp/` dir similar to `static.8739602554c7f3241958e3cc9b57fdecb474d508.html` (template name + sha + extension).

The first time the template will be evaluated and compiled. Reloading the page the HTML will be served without re-compiling.

2.23.2 Static partial example

Given this into `index.php`

```
if (is_front_page()){
  render_template("templates/archive");
}
```

and given this `views/templates/static.pug`

```
extends /layouts/default.pug

block yield
  h2 Archive (static example)

  ul.archive
    while (have_posts())
      - the_post()

      li
        - render_static('partials/post')
```

visiting your home page will produce a static HTML into theme's `tmp/` dir similar to `post.8739602554c7f3241958e3cc9b57fdecb474d508.html` (template name + sha + extension).

2.23.3 Invalidating the cache

You have 3 way to handle this:

- manually deleting one or more `.html` files from theme's `tmp/` folder
- blank `tmp/` folder with `wp wordless theme clear_tmp`
- from the "Cache management" menu within the admin panel

The "Cache management" menu needs to be activated decommenting this line

Listing 30: backend.php

```
85 /*
86  * Create Cache management menu & render cache management page
87  *
88  * A default page is rendered, but you can make your own function and replace it␣
    ↪instead of Wordless::render_static_cache_menu
89  * Enable cache management by uncommenting line below.
```

(continues on next page)

(continued from previous page)

```
 */
// add_action('admin_menu', 'cache_management');
```

2.23.4 Manually manage cache SHA

The cache policy of static generated views is based on the view's name + the **SHA1 of serialized $locals**. As it stands the best way to introduce business logic in the expiration logic is to pass ad hoc extra variables into the $locals array. For example having

```
render_static('pages/photos', $locals = [ 'cache_key' => customAlgorithm() ])
```

when `customAlgorithm()` will change its value, it will invalidate the static cache for this template

2.23.5 Known limitations

`render_static` creates hashes using the serialized *$locals* array; since it's possible to pass a `Closure` in `$locals` but **it's impossible to serialize closures in PHP** you cannot use Wordless's `render_static` if you're passing in closures.

2.24 ACF Gutenberg Blocks

Warning: If you're not using ACF plugin, this feature won't be available

Worldess has built-in support for registering new custom gutenberg blocks through Advanced Custom Fields.

To register a block go to the initializer `config/initializers/custom_gutenberg_acf_blocks.php` and uncomment the last line in the `custom_gutenberg_acf_blocks()` function.

The function is very well self documented:

Listing 31: config/initializers/custom_gutenberg_acf_blocks.php

```
<?php

function custom_gutenberg_acf_blocks() {
    /*
     * Create Gutenberg Block with Advanced Custom Fields.
     * This function is a wrapper around the `acf_register_block` function. Read more␣
↪about it at
     * https://www.advancedcustomfields.com/blog/acf-5-8-introducing-acf-blocks-for-
↪gutenberg/
     *
     * Note: You can reapeat it for as many blocks as you have to create
     *
     * Params:
     *     string, mandatory:
     *         "block name"; if you use spaces in the name, they'll get converted to␣
↪`-`
                where needed. You'll need to name your partial the same as this param.
                E.g.: having "Home Page" Wordless will search for `views/blocks/home-
↪page.html.pug`
```

(continues on next page)

(continued from previous page)

```
                partial
     *     array, optional:
     *        title           => if blank use $block_name
     *        description     => if blank use $block_name
     *        category        => if blank use 'formatting'.
                                  Default categories are:
                                     'common',
                                     'formatting',
                                     'widgets',
                                     'layout',
                                     'embed'.
     *        icon            => if blank use 'smiley'; you can use any icon name from
     *                           https://developer.wordpress.org/resource/dashicons/
     *        render_callback => if blank use the default '_acf_block_render_callback
↪',
     *        keywords        => if blank use ['acf', 'block']
     *
     */

    /* Example:
    create_acf_block('slider', [
        'title' => 'Slider',
        'description' => 'Slider',
        'category' => 'widgets',
        'icon' => 'admin-comments',
        'render_callback' => '_acf_block_render_callback',
        'keywords' => [ 'image', 'slider' ]
    ]);
    */

    // create_acf_block('slider', array());
}

add_action('acf/init', 'custom_gutenberg_acf_blocks');
```

Having a block registered this way, you will found it selectable in the ACF field group's options.

Said you'll register a block like

```
<?php

create_acf_block('slider', [
    'title' => 'Slider',
    'description' => 'Slider',
    'category' => 'widgets',
    'icon' => 'admin-comments',
    'render_callback' => '_acf_block_render_callback',
    'keywords' => [ 'image', 'slider' ]
]);
```

Wordless will search for two partials to render:

- `views/blocks/admin/_slider.html.pug`
- `views/blocks/_slider.html.pug`

The first one is used to render the block in the backend Gutenberg's interface. If absent, then the second will be used.

You will be obviously free to render the block anywhere in your front-end template, since it's a simple partial:

```
render_partial('blocks/slider')
```

Note: You can change the path where the partial is searched for by using the *wordless_acf_gutenberg_blocks_views_path* filter

2.25 Using plain PHP templates

Let's take the unaltered default theme as an example. In `views/layouts` we have the `default` template which calls a `include` for the `header.pug` partial.

Listing 32: views/layouts/default.pug

```
doctype html
html
  head
    include /layouts/head.pug
  body
    .page-wrapper
      header.site-header
        include /layouts/header.pug

      section.site-content
        block yield

      footer.site-footer
        include /layouts/footer.pug

    // jQuery and application.js is loaded by default with wp_footer() function. See
↪config/initializers/default_hooks.php for details
    - wp_footer()
```

Listing 33: views/layouts/header.pug

```
h1!= link_to(get_bloginfo('name'), get_bloginfo('url'))
h2= get_bloginfo('description')
```

Let's suppose we need to change `header.pug` in a PHP template because we don't like PUG or we need to write complex code there.

Warning: If you have to write complex code in a view you are on the wrong path :)

1. Rename `header.pug` in `_header.php`
2. Update its content, e.g.:

Listing 34: views/layouts/_header.php

```
<h1> <?php echo link_to(get_bloginfo('name'), get_bloginfo('url')); ?> </
↪h1>
<h2> <?php echo htmlentities(get_bloginfo('description')) ?> </h2>
```

3. In `default.pug` substitute the line `include /layouts/header.pug` with the appropriate Wordless' PHP render helper

Listing 35: views/layouts/default.pug

```
= render_partial('layouts/header')
```

4. Done

When `render_partial("layouts/header")` doesn't find `_header.pug` it will automatically search for `_header.php` and will use it *as is*, directly including the PHP file.

Note: `render_partial` function expects that partials are named with a leading underscore (_). This is due to backward compatibility and we have to stuck and deal with it.

Note: PUG (and thus PHUG) has some builtin function to compose templates: `include`, `extends`, `mixin`. Such functions work only within and for PUG files. That's why in order to load a plain PHP template we need to use plain PHP functions.

2.25.1 PHP render helpers

render_view()

The main helper function used to render a view is `render_view()`. Here is its signature:

```
<?php
/**
    * Renders a view. Views are rendered based on the routing.
    *   They will show a template and a yielded content based
    *   on the page requested by the user.
    *
    * @param  string $name   Filename with path relative to theme/views
    * @param  string $layout The template to use to render the view
    * @param  array  $locals An associative array. Keys will be variable
    *                        names and values will be variable values inside
    *                        the view
    */
    function render_view($name, $layout = 'default', $locals = array())      {
      /* [...] */
    }
```

Note: Extension for `$name` can always be omitted.

Inside the `views` folder you can scaffold as you wish, but you'll have to always pass the relative path to the render function:

```
<?php
render_view('folder1/folder2/myview')
```

Note: By the way vanilla theme ships with a proposed scaffold.

The `$locals` array will be auto-`extract()`-ed inside the required view, so you can do

```
<?php
render_view('folder1/folder2/myview', 'default', array('title' => 'My title'))
```

and inside `views/folder1/folder2/myview.pug`

```
h1= $title
```

and `$title` variable will be set.

render_partial()

`render_partial()` is almost the same as its sister `render_view()`, but it does not accept a layout as argument. Here is its signature:

```
<?php
/**
* Renders a partial: those views followed by an underscore
*   by convention. Partials are inside theme/views.
*
* @param  string $name   The partial filenames (those starting
*                        with an underscore by convention)
*
* @param  array  $locals An associative array. Keys will be variables'
*                        names and values will be variable values inside
*                        the partial
*/
function render_partial($name, $locals = array()) {
```

Partial templates – usually just called **"partials"** – are another device for breaking the rendering process into more manageable chunks.

Note: Partials files are **named with a leading underscore** to distinguish them from regular views, even though they are **referred to without the underscore**.

Layouts

`views/layouts` directory

When Wordless renders a view, it does so by combining the view within a layout.

E.g. calling

```
render_view('folder1/folder2/myview')
```

will be the same as calling

```
render_view('folder1/folder2/myview', 'default', array())
```

so that the `default.pug` (or `.php` if you'll update it too) layout will be rendered. Within the layout, you have access to the `wl_yield()` helper, which will combine the required view inside the layout when it is called:

```
doctype html
html
  head= render_partial("layouts/head")
  body
    .page-wrapper
      header.site-header= render_partial("layouts/header")
      section.site-content= wl_yield()
      footer.site-footer= render_partial("layouts/footer")
    - wp_footer()
```

Note: For content that is shared among all pages in your application that use the same layout, you can use partials directly inside layouts.

CHAPTER 3

TODOs

A list of known bugs, wip and improvements this documentation needs, hoping it will be kept empty ;)