Wordless Documentation

Release 2.3.0

weLaika

Dec 01, 2020

Contents

1 Introduction **3**

2 Contents **5**

2.1 Installation . . . 5
2.1.1 Wordless GEM (favourite) . . . 5
2.1.2 (Not so) Manual . . . 5
2.1.2.1 Prerequisites . . . 5
2.1.2.2 Steps . . . 6
2.2 Usage . . . 6
2.2.1 Theme anatomy . . . 6
2.2.1.1 Routing . . . 7
2.2.1.2 Rendering . . . 8
2.2.1.2.1 render_view() . . . 8
2.2.1.2.2 render_partial() . . . 9
2.2.1.2.3 Layouts . . . 9
2.2.1.2.4 Views . . . 10
2.2.1.3 Helpers . . . 10
2.2.1.4 Initializers . . . 11
2.2.1.5 Locale files . . . 11
2.2.1.6 Assets . . . 12
2.2.1.6.1 The Fast Way . . . 12
2.2.1.6.2 I need to really understand . . . 12
2.2.2 Build and distribution . . . 13
2.2.2.1 Build for development . . . 13
2.2.2.2 Build for production . . . 13
2.2.2.2.1 Release signature . . . 14
2.2.2.3 Code linting . . . 14
2.2.2.4 PHUG optimizer . . . 14
2.2.2.5 Deploy . . . 15
2.2.3 Filters . . . 15
2.2.3.1 wordless_pug_configuration . . . 15
2.3 Development stack . . . 16
2.3.1 Nodejs . . . 16
2.3.1.1 Version . . . 16
2.3.1.2 NVM . . . 16
2.3.2 Development environment . . . 17

2.3.2.1 YARN 17
2.3.2.2 Foreman 17
2.3.2.3 wp server 17
2.3.2.4 BrowserSync 18
2.3.2.5 MailHog 18
2.3.3 Code compilation 19
2.3.3.1 PHUG 20
2.3.3.1.1 Who compiles PUG? 20
2.3.3.2 CoffeeScript and Sass 20
2.3.3.2.1 Paths 20
2.3.3.2.2 Inclusion of compiled files 21
2.3.3.2.3 Multiple “entries” 22
2.3.4 Using plain PHP templates 23
2.3.4.1 Conclusions 24

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 the Pug templating system
- Write CSS stylesheets using the awesome Sass syntax
- Write Javascript logic in Coffeescript
- A growing set of handy and documented PHP helper functions ready to be used within your views
- Development workflow backed by WebPack, BrowserSync (with live reload), WP-CLI, Yarn. All the standards you already know, all the customizations you may need.

CHAPTER 2

Contents

2.1 Installation

2.1.1 Wordless GEM (favourite)

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

No prerequisites. Just joy.

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 ;)

2.1.2 (Not so) Manual

At the end of the installation process you will have

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

2.1.2.1 Prerequisites

1. Install WP-CLI http://wp-cli.org/#installing
2. Install global packages from NPM: `npm install -g foreman yarn`[1][2] (you already have node on your development machine, haven't you?)
3. WordPress installed and configured as per official documentation
4. Install MailHog. On MacOS this is as simple as `brew install mailhog`. Wordless will do the rest.

[1] https://www.npmjs.com/package/yarn
[2] https://www.npmjs.com/package/foreman

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's flow does not need any external web server, since it will use the wp server command to serve your wordpress.

See also:

Development environment

2.1.2.2 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
   ```

3. Enter theme directory

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

4. Bundle NPM packages

   ```
   yarn install
   ```

5. Start the server - and the magic

   ```
   yarn run server
   ```

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

See also:

Development environment

Note: It is possible that your OS asks you to allow connections on server ports (3000 and/or 8080). It's just ok to do it.

2.2 Usage

2.2.1 Theme anatomy

This is a typical Wordless theme directory structure:

```
your_theme_dir
├── config/
│   ├── initializers/
│   └── locales/
├── dist/
│   ├── fonts/
│   ├── images/
│   ├── javascripts/
│   ├── stylesheets/
│   └── README.md
├── helpers/
│   └── README.mdown
├── node_modules/
├── src/
│   ├── images/
│   ├── javascripts/
│   ├── stylesheets/
│   └── main.js
├── tmp
│   └── .gitkeep
├── views
│   ├── layouts
│   └── posts
├── .browserslistrc
├── .env
├── .gitignore
├── .nvmrc
├── .stylelintignore
├── .stylelintrc.json
├── Procfile
├── index.php
├── package.json
├── release.txt
├── screenshot.png
├── style.css
├── webpack.config.coffee
├── webpack.env.coffee
└── yarn.lock
```

Now let's see in detail what is the purpose of all those directories.

2.2.1.1 Routing

The *index.php* serves as a router to all the theme views.

```php
<?php

if (is_front_page()) {
  render_view("static/homepage)");
} else if (is_post_type_archive("portfolio_work")) {
  render_view("portfolio/index");
} else if (is_post_type("portfolio_work")) {
  render_view("portfolio/show");
}
```

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.

See also:

render_view() helper documentation

See also:

Using Page Template Wordpress' feature inside Wordless

2.2.1.2 Rendering

2.2.1.2.1 render_view()

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

```php
<?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())     {
      /* [...] */
    }
```

Thanks to this helper, Wordless will always intercept **PUG** files and automatically translate them to HTML.

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

See also:

PHUG section @ *Code compilation*

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

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

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

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

```pug
h1= $title
```

2.2.1.2.2 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
<?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()) {
    $parts = preg_split("/\//", $name);
    if (!preg_match("/^_/", $parts[sizeof($parts)-1])) {
        $parts[sizeof($parts)-1] = "_" . $parts[sizeof($parts)-1];
    }
    render_template(implode($parts, "/"), $locals);
}
```

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**.

2.2.1.2.3 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.html.phug` 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")
```

(continues on next page)

(continued from previous page)

```
      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.

2.2.1.2.4 Views

`views/**/*.pug` or `views/**/*.php`

This is the directory where you'll find yourself coding most of the time. Here you can create a view for each main page of your theme, using Pug syntax or plain HTML.

Feel free to create subdirectories to group together the files. Here's what could be an example for the typical WordPress loop in an archive page:

```
// views/posts/archive.html.pug
h2 Blog archive
ul.blog_archive
  while have_posts()
    - the_post()
    li.post= render_partial("posts/single")
```

```
// views/posts/_single.html.pug
h3!= link_to(get_the_title(), get_permalink())
.content= get_the_filtered_content()
```

Wordless uses Pug.php - formerly called Jade.php - for your Pug views, a great PHP port of the PugJS templating language. In this little snippet, please note the following:

- The view is delegating some rendering work to a partial called `_single.html.pug`
- There's no layout here, just content: the layout of the page is stored in a secondary file, placed in the `views/layouts` directory, as mentioned in the paragraph above
- We are already using two of the 40+ Wordless helper functions, `link_to()` and `get_the_filtered_content()`, to DRY up this view
- Because the `link_to` helper will return html code, we used unescaped buffered code to print PUG's function: `!=`. Otherwise we'd have obtained escaped html tags.

It looks awesome, right?

2.2.1.3 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``

2.2.1.4 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_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_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
- **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.

2.2.1.5 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")`

2.2.1.6 Assets

2.2.1.6.1 The Fast Way

- write your SASS in `src/stylesheets/screen.sass`
- write your CoffeeScript in `src/javascripts/application.js.coffee`

and all will automagically work! :)

2.2.1.6.2 I need to really understand

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 `assets/xxx` folder.

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

Images will be optimized by ImageminPlugin. The default setup already translates `url` s inside css/sass files in order to point to images in the right folder.

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

See also:

Code compilation

See also:

- Official Sass guide
- Official CoffeeScritp 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 CoffeeScritp/Javascript you can do

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

and go on as usual.

2.2.2 Build and distribution

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:

CoffeeScript and Sass

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

Listing 1: package.json

```
  "scripts": {
    "server": "npx nf start",
    "build:dev": "webpack --debug --env.NODE_ENV=development",
    "build:prod": "yarn sign-release && webpack -p --bail --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 --report-needless-disables 'src/stylesheets/**/*.sass
↪' && npx stylelint 'src/stylesheets/**/*.sass'"
  },
```

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.

2.2.2.1 Build for development

```
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.

2.2.2.2 Build for production

```
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

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

Note: By default the production build **will** produce source-maps for JS; this is done to lower the debugging effort, to respect the readability of the source code in users' browser and to simplify the shipping of source-maps to error monitoring softwares such as Sentry.

You can easily disable this behaviour setting `devtool: false` in `webpack.env.coffee` inside the `prodOptions` object.

2.2.2.2.1 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).

2.2.2.3 Code linting

Wordless ships with preconfigured linting of SASS (indented syntax) 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 the the 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.

2.2.2.4 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 2: render_helper.php

```
                if ($this->ensure_dir($tmp_dir)) {
                    if ( getenv('ENVIRONMENT') ) {
                        $env = getenv('ENVIRONMENT');
                    } elseif ( defined('ENVIRONMENT') ) {
```

(continues on next page)

(continued from previous page)

```
                    $env = ENVIRONMENT;
                } else {
                    $env = 'development';
                }

                if ( in_array( $env, array('staging', 'production') ) ) {
                    \Pug\Optimizer::call(
                        'displayFile', [$template_path, $locals],␣
↪WordlessPugOptions::get_options()
                    );
                } else {
                    $pug = new Pug(WordlessPugOptions::get_options());
                    $pug->displayFile($template_path, $locals);
                }
```

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`.

```
:caption: wp-config.php

<?php
define('ENVIRONMENT', 'production');
```

2.2.2.5 Deploy

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

2.2.3 Filters

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

2.2.3.1 wordless_pug_configuration

Listing 3: wordless/helpers/pug/wordless_pug_options.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(),
            'strict' => true,
            'debug' => $wp_debug,
            'enable_profiler' => false,
```

(continues on next page)

(continued from previous page)

```
            'error_reporting' => E_ERROR | E_USER_ERROR
        ]);
    }
}
```

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;
}
```

2.3 Development stack

Here are the stack components of Wordless' development workflow:

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

Contents

2.3.1 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.

2.3.1.1 Version

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

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.

2.3.1.2 NVM

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.

Setup of NVM is not covered in this documentation.

Once set up, you can use

```
nvm use
```

2.3.2 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.

2.3.2.1 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 4: package.json

```
  "scripts": {
    "server": "npx nf start",
    "build:dev": "webpack --debug --env.NODE_ENV=development",
    "build:prod": "yarn sign-release && webpack -p --bail --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.

2.3.2.2 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 5: Procfile

```
wp: wp server --host=0.0.0.0
webpack: npx webpack --debug --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

2.3.2.3 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.

2.3.2.4 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 6: webpack.config.coffee

```
    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*.

2.3.2.5 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 7: .env

```
MAILHOG=true
```

This will trigger the `smtp.php` initializer:

Listing 8: config/initializers/smtp.php

```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
}
```

2.3.3 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
- **CoffeeScript** 2 for JS (ES6 ready)
- **Sass** 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
<?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.

2.3.3.1 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 9: 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.

2.3.3.1.1 Who compiles PUG?

When a `.html.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.

2.3.3.2 CoffeeScript and Sass

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

The default webpack configuration file is written itself in Coffeescript, because it is natively supported by Webpack and because it makes the code easier to read.

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.coffee`.

2.3.3.2.1 Paths

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

Listing 10: webpack.config.coffee

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

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

Listing 11: webpack.config.coffee

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

    output: {
      filename: "[name].js"
      path: javascriptsDstPath
    }
```

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

Listing 12: webpack.config.coffee

```
      new MiniCssExtractPlugin {
        filename: '../stylesheets/[name].css'
      }
```

2.3.3.2.2 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 13: config/initializers/default_hooks.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() {
```

(continues on next page)

(continued from previous page)

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

add_action('wp_enqueue_scripts', 'enqueue_javascripts');
```

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

2.3.3.2.3 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 14: main.js

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

As we've already said having an *entry* which requires both JS and SASS, 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 SASS file or a piece of JS logic
- add the *entry* to webpack config

```
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");
```

(continues on next page)

(continued from previous page)

```
    wp_enqueue_script("backend");
}
```

or add it anywhere in your templates:

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

2.3.4 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 `render_partial` for the `_header` partial.

Listing 15: views/layouts/default.html.pug

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

Listing 16: views/layouts/_header.html.pug

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

Let's suppose we need to change `_header` 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.html.pug` in `_header.html.php`
2. Update its content, e.g.:

Listing 17: views/layouts/_header.html.php

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

3. Done

When `render_partial("layouts/header")` doesn't find `_header.html.pug` it will automatically search for `_header.html.php` and will use it *as is*, without passing through any compilation process.

2.3.4.1 Conclusions

As you can see, Wordless does not force you that much. Moreover, you will continue to have its goodies/helpers to break down views in little partials, simplifying code readability and organization.