Internal Developer Advocate

We'd like to think that developers stay at Automattic for many years, not because of the perks or free meals at the office, but because of the technical challenges to keep the open web strong, because of the inclusive environment, and because Automattic's creed starts with "I will never stop learning."

Our Developer Experience team makes all this easier through hiring, learning, onboarding, tools, and technical direction. Right now it's four of us and our immediate tasks vary from scaling parts of our hiring process, through coordinating a cross-team framework effort, to leading a user-facing project. We'd love you to join our merry band, too, to help with learning, onboarding, and keeping developers at Automattic happy all around.

Here is what you will achieve in your first 6–12 months:

- You will lead creating content for our developer onboarding process. The current onboarding process has a lot covered – assigning an onboarding buddy, regular check-ins, a checklist of topics to study. But what it's missing is more engaging and structured learning materials that can guide a newcomer step by step through our codebases. Together with more experienced developers, you will coordinate and participate hands-on in producing codelabs, presentations, screencasts, and any other materials you think will be useful.
- You will have an impact on technical communication and learning patterns for developers via initiatives of your own. For example you might be working on how to document design and architecture decisions, how to share knowledge between teams, and how to grow developers according to their interests.

You might be a good fit if you:

- Are great at communicating complex technical topics effectively.
- Can empathize with other developers through their joys and hardships.
- Have some prior developer advocate or learning or onboarding experience.
- Have a strong software development background – hands-on full-stack skills are very useful, especially in JavaScript and/or PHP.
- Have influenced developers who do not report to you, via words, visuals, or code.
- Value simplicity, in both code and in process.
- Understand the complexity of a wide community of developers (i.e. open source) and how short-term decisions have long-term ripple effects on groups.

You don't need to be an expert in all those areas, we're looking for somebody who will excel in some areas and have some interest and skills in others.

Speaking of interests and skills, here some areas in which you can grow and have further impact in the future:

- Engineering effectiveness – having a close relationship with developers may put you in a great position to find ways to make developers more effective through tools, practices, cross-team collaborations, and process improvements.
- Engineering hiring – bringing in diverse developers to support Automattic's mission to make the web a better place requires a lot of thought, planning and hard work.
- Managing a team – while you will be the first internal developer advocate, if the experiment is successful we may grow the practice.

We're serious about growing diversity in the tech industry. We want to build Automattic as an environment where people love their work and show respect and empathy to those with whom we interact. Diversity typically includes, but is not limited to, differences in race, gender, sexual orientation, gender identity or expression, political and religious affiliation, socioeconomic background, cultural background, geographic location, disabilities and abilities, relationship status, veteran status, and age. To work on diversity means that we welcome these differences, and strive to increase the visibility of traditionally underrepresented groups. Read more about our dedication to diversity and inclusion.

How to Apply

Does this sound interesting? If yes, please send a short email to **jobs @ this domain** telling us about yourself and **attach a résumé**. Let us know what you can contribute to the team. **Include the title** of the position you're applying for and your name in the subject.
Proofread! Make sure you spell and capitalize WordPress and Automattic correctly. We are lucky to receive hundreds of applications for every position, so try to make your application stand out. If you apply for multiple positions or send multiple emails there will be one reply.
If you're reading this on a site other than automattic.com please ensure you visit automattic.com/work-with-us for the latest details on applying.

Please answer the following questions in your cover letter. Applications without these questions answered will not be considered:

- Describe in a paragraph or two a great developer onboarding process, step by step, as you imagine it.
- Describe a developer community (internal or external) that could have been more effective?
- What questions do you have for us?

TLDR;

I’m glad you have a Developer Advocate role. You need it internally but you might consider to have public facing advocacy, too.

I’m a designer, developer and writer working with WordPress since 2005. I have the INFJ-T / Advocate personality type. No wonder I’m doing advocacy on my own.

What about working together to make WordPress better for all, inside and outside Automattic?

Introduction

I’m a designer and developer in tech terms; in non-tech terms I’m a writer and maybe a philosopher. Just check my blog Gust [1] about the control of culture, or, an article written for A List Apart — *Devices as designers* [2] — rejected, “too murky”.

Gust got featured on Brutalist Websites [3] — which I find valuable since the design skills I’ve acquired on my own. In school I’ve studied only Computer Science.

My career: it starts with WordPress back in 2005 and version 1.2.3 when one could found gems like *Slow down cowboy* in your code. I’ve built a local startup with WordPress using the shop & blog formula which went immense success. In 2013 I’ve sold my shares and ventured back into learning, experimenting, and freelancing.

Since then I’ve tried many other platforms and languages than WordPress and PHP — most notably Ruby and Rails, Yii, various static site generators, built my own static site generator on Webpack / Gulp together with a dynamic styleguide, Vue, Contentful, Craft, Shopify.

I couldn’t stick to WordPress as my main framework and PHP my main language because I’ve felt they are ungainly, incumbent and monolithic, far from best practices — just to name few. I needed to pursue other ways, follow some hype to realize others are not (much) better. Every approach has its own advantages and drawbacks, there is no silver bullet.

A very interesting thing happened all those years: clients kept coming back and asking for WordPress. In fact it was no year without a project completed on WordPress starting from small artist websites to e-commerce making 50k hits a day.

Then last year I’ve decided I’ll stick to WordPress and start making a living from it on long term. Why WordPress? Because it is the most resilient platform from the last 10 years I know, and, because it has to have some awesome magic in it to power every third website on the planet.

The news around the ecosystem were good too: Calypso, Gutenberg, HardyPress (static site generator), Multisite incorporated into the main codebase, the power to influence React licensing — finally, after long years of silence WordPress seemed to got some traction and limelight again and the direction was modern, trustful, and interesting.

Still one thing kept ticking in the back of my mind: why when almost every other website runs WordPress I can't recognize none of them? In fact I recognize only those sites which are still not responsive and have a blogroll and a meta in their sidebar.

If WordPress is such a success story why I don't have that *aha!* moment from time to time? Why the pride is not shared? Why there is no WordPress badge like *Fork me on Github*, or *Made with love and WP*?

Moreover after two years working exclusively with WordPress I still can't find a community for it like Hacker News, Designer News, Lobsters, Mixed Parts … nor leaders like Guido Van Rossum, Paul Graham, DHH, or at least David Lee of Squarespace. Not even a simple fully satisfying newsletter like WDRL by Anselm Hannemann or Val Head's UI Animation Newsletter.

The last time I've heard about Matt he was hiring a former US Army general. Sic! Where is WordPress ???

Meantime:

- I see forward-looking, developer friendly initiatives like the Indie Web [4] embracing and advocating WordPress but no visible support in return.
- I can't find real Multisite hosting. It's like being a second class citizen, or worse, even got reminded often times by hosting providers multisite accounts will be suspended.
- SAAS startups, MVP builders, Marketplace builders are all willing to go Multisite but no incentives can be found even around WordPress.org's own website.
- According to statistics WordPress is in decline as blogs and blogging platforms do globally, but declining faster than content and content management systems do. In the same climate others like Shopify, Wix, or even Squarespace prosper.[#1]

I think it can be done better. WordPress needs leadership, needs voice, needs more use cases underpinned with boilerplate examples, needs a visible, vibrant community site, more advocates, designers, developers, startup founders betting their ventures and lives on it.

WordPress needs new directions to gain professional mindshare like Mozilla does with Firefox, or recently Microsoft with open source.

We need to tell the world that beside Guttenberg and Calypso — which are very considerable results for a certain audience — WordPress stepped away from being that clumsy, slow, expensively caching incumbent and now it can be compiled into a static site with zero vulnerabilities and sub-second page loading speed — for example.

Or it can be the next platform for your MVP, SAAS, or Marketplace by easily wiring together functionalities from pre-made, vetted pieces in no time.

Skills

According to others [5] my personality type is INFJ-T / The Advocate. Certainly I’m not suited well to be a Code Wrangler but I definitively present good skills on advocating.

Job	**Requirement**	**My skills**	**Notes**
Wrangler	Advanced proficiency with at least one scripting language like PHP or Python.	No	I have 6 kyu in PHP on Codewarrior
	Experience working on a large-scale system	No	Max 50k users/day
	The ability to iterate and ship ideas quickly	Yes	
	A deep understanding of the Web including HTTP, HTML, CSS, and JavaScript	Yes	
	Customer-facing roles including support and documentation.	Yes	
Advocate	Great at communicating complex technical topics effectively.	Yes	
	Can empathize with other developers through their joys and hardships.	Yes	
	Have some prior developer advocate or learning or onboarding experience	No	
	Have a strong software development background	Yes	In this context
	Have influenced developers who do not report to you, via words, visuals, or code.	No	
	Value simplicity, in both code and in process.	Yes	
	Understand the complexity of a wide community of developers (i.e. open source)	No	

. In your job description there are a few items which truly resonate with me:

> *Coordinate and participate hands-on in producing codelabs, presentations, screencasts, and any other materials you think will be useful.*
>
> *Find ways to make developers more effective through tools, practices, cross-team collaborations, and process improvements.*
>
> *Bringing in diverse developers to support Automattic's mission to make the web a better place.*

They all are in the same way important both internally and externally. A process can start advocacy inside then distill the results for other people worldwide.

Sample project

Recently I've been involved in a similar project [6] about advocating best practices. Started internally from real needs then open sourced for public access.

After publishing my first theme [7] and finishing its premium version I've started to collect the best practices on WordPress theme and plugin development to create something like the Twenty Seventeen boilerplate theme but instead of focusing on just a theme I've extended it with a plugin and a child theme.

The idea is to teach people how to create themes for WordPress.org which also offer a premium version through plugins and child themes.

Many people do this, and *an official, vetted help from WordPress* like the Twenty XX themes would yes, give another 1,000 hours head start. Instead of reinventing the wheel, to write the same code all over again, people would focus exclusively on design. Good looking themes are still a big need in every theme store, true for WordPress.org.

Next project would be to create a Multisite boilerplate project to teach people how to create their own SAAS business atop of WordPress.

Related works

A showcase of skills in global oversight together with small details.

- My current blog on WordPress, served as a static site by HardyPress
- My previous blog on a custom made static site generator
- Another high level blog / static site on yearly global tech trends

On technical, public facing documentation please check the Log Lolla Pro theme’s homepage.

Footnotes

1. Gust — a blog about the control of culture
2. Devices as designers — An article about the future of web design, written for A List Apart
3. Gust featured on Brutalist Websites
4. Indie web — An alternative for corporate blogging
5. The INFJ-T personality type
6. The More Themes Baby best practices for WordPress on Github
7. The Log Lolla theme in the WordPress.org store

WordPress best practices

Boilerplate code for WordPress plugins and themes based on best practices.

What's inside

1. A WordPress.org compatible theme: Mo Theme.
2. A WordPress.org compatible plugin: Mo Plugin.
3. A child theme using the plugin: Mo Pro Theme.

How it works

- Mo Theme simply displays posts and comments as required by the WordPress.org theme store.
- Mo Plugin adds a custom post type together with a shortcode.
- Mo Pro Theme extends Mo Theme and uses Mo Plugin:
    - Displays the shortcode added by the Mo Plugin.
    - Extends the homepage from Mo Theme by adding a sidebar.
    - Displays a widget in the sidebar with custom post types added by the Mo Plugin.

Best practices

- Default WordPress files organization
- Class based namespaces
- Components
- Loose coupling
- Single responsibility principle
- Optimized database operations
- Documentation

Mo Theme

A WordPress.org compatible boilerplate theme based on best practices.

Best practices

- Default WordPress theme files organization
- Class based namespaces for WordPress.org / PHP version <5.3 compatibility

- Loose coupling
- Single responsibility principle
- Components
- Extendability
- Semantic and outlined HTML structure
- Documentation for the PHP API, the templates and template parts, and for SCSS

PHP Best Practices

Class based namespacing

WordPress supports three techniques to avoid naming collisions:

- Prefixing
- OOP classes
- PHP namespaces

WordPress.org / PHP < 5.3 supports only the first two.
According to best practices OOP classes are the easier way to tackle this problem.

Loose coupling

Loose coupling makes sure code is open, easily modifiable without breaking the site.
For different types of code different techniques are used.

Class variables

Class variables are dynamically set and get through overloading / magic methods.

This is not recommended.

```
/**
* Class arguments.
```

```
*
* Used to setup the class.
*
* @since 1.0.0
*
*/
public $block    = string;
public $element  = string;
public $modifier = string;
```

If an argument is removed / renamed the old version of the code might broke. For example if $modifier is removed an $object->modifier call breaks the code.

This is better:

```
/**
* Class arguments.
*
* Used to setup the class.
*
* @since 1.0.0
*
* @var array An array of arguments.
*/
public $arguments = array(
        'block'         => '',
        'element'       => '',
        'modifier'      => '',
        ....
);

public function __set( $variable, $value ) {}
...

public function __get( $variable, $value ) {}
```

Arguments are a dynamic array with any number of items. They are get and set programmatically and managed when an argument item is not found .

If $modifier is removed the _get method can manage what happens on an '$object->modifier` call.

Function arguments

Function arguments are passed as arrays instead of lists of arguments.

This is not recommended:

```
function display( $title, $description, $author ) { ... }
```

When one argument is removed, the name or order is changed the old versions of the code might break.

This is recommended:

```
$default_arguments = array(
        'title'       => '',
        'description => '',
        'author'      => '',
);
function display( $arguments = array() ) {
        $arguments = array_merge( $default_arguments, $arguments );
}
```

Arguments are a dynamic array with any number of items in any order. They are easily replaceable and modifiable.

If we need a new kind of author we can do something like:

```
$default_arguments = array(
        'title'       => '',
        'description => '',
        'author'      => '',
        'author2'     => array(),
);
```

Template variables

Templates are communicating with each other through the get_template_part and set_query_var / get_query_var.

Passing arguments between template parts is done with an array instead of a list of arguments.

This is wrong:

```
set_query_var( 'post-list-title', theme_get_archive_label( 'Posts' ) );
set_query_var( 'post-list-klass', 'for-archive' );
set_query_var( 'post-list-content', theme_get_archive_content() );
get_template_part( 'template-parts/post-list/post-list', '' );
```

This is recommended:

```
$post_list_query_vars = array(
	'title'   => theme_get_archive_label( 'Posts' ),
	'klass'   => 'for-archive',
	'content' => theme_get_archive_content(),
);
set_query_var( 'post-list-query-vars', $post_list_query_vars );
get_template_part( 'template-parts/post-list/post-list', '' );
```

The logic is the same: pass an array of arguments which can be handled dynamic instead of a list of arguments which is static.

Single responsibility principle

Every folder, file, class, function, mixin - you name it - is meant to do one thing and do it well.

Single source of truth

Make sure everything has a single origin.
For example wp_get_theme() gives us the theme version number. Instead of define('THEME_VERSION', '0.1.0'); use wp_get_theme()->get('version').

No hardwired data inside functions

This is not recommended:

```
$file_name = 'js/' . $text_domain . '.js';
```

Instead js/, .js should be moved into variables.

```
/**
* Theme arguments.
*
* @since 1.0.0
*
* @var array $arguments An Array of arguments.
*/
public $arguments = array(
	'javascript_folder'     => 'js/',
	'javascript_extension'  => '.js',
);

$file_name = $arguments['javascript_folder'] . $text_domain .
$arguments['javascript_extension']
```

No HTML in PHP code

HTML code belongs to templates and template tags.

When a PHP function needs to return HTML the output buffering method with a get_template_part call should be used.

This is not recommended:

```
	function theme_get_arrow_html( $direction ) {
		return
			'<span class="arrow-with-triangle arrow-with-triangle--' . $direction . '">
			<span class="arrow-with-triangle__line"></span>
			<span class="triangle triangle-- arrow-with-triangle__triangle"></span>
			</span>';
	 }
```

This is better:

```
	function theme_get_arrow_html( $query_vars ) {
		$arguments = array(
			'query_var_name'     => 'arrow-with-triangle-query-vars',
			'query_var_value'    => $query_vars,
			'template_part_slug' =>
	'template-parts/html-component/arrow-with-triangle/arrow-with-triangle',
			'template_part_name' => '',
		);

		return get_template_part( $arguments );
	}
```

Command-query separation

Every function either executes a *command* or performs a *query*. No functions do both at the same time.

The role of the function is described by a prefix. Either is a get_ for a query, or, another verb for a command like set_, add_, create_ and so on.

There should be no functions which have no prefix, except when the function name is a verb, and the function is a member of a class.

This is not recommended:

```
	function content_width() { ... }
```

This is better:

```
function get_content_width() { ... }
function set_content_width() { ... }
```

Or for classes:

```
$content = new ThemeContent();
$width   = $content->width->get();
```

HTML Best Practices

Default WordPress theme files organization

WordPress has a clear indication how to organize templates, template parts and template tags.

To keep the theme developer friendly stick to this standard instead of reinventing the wheel.

Components

Components are a way to organize code in such way developers can realize immediately where to look for a specific code part.

For example if a page has a <header class="site-header"> structural element (a.k.a component) developers should be able to easily realize where the PHP, HTML, and CSS and other code responsible for this element is located.

There should be a:

- template-parts/site-header/site-header.php template part for HTML code
- includes/template-tags/class-site-header.php template tag for PHP code, and

- assets/scss/parts/site-header.scss for (S)CSS code.
- assets/js/site-header.js for JS code.

It all starts with HTML which defines the structure of the page with class names: class="site-header". If class names are cleverly and intuitively set they can enable this kind of component architecture.

Auto-generated classnames

To achieve this consistency component class names and element identifiers are generated instead of being added manually. Manual work is a bug. Always be automating.

Manually we can make mistakes:

```
<aside class="non-consistent-naming">
```

With an algorithm is harder:

```
$attributes = array(
        'block'    => 'post',
        'element'  => 'excerpt',
    );

<aside <?php $component->attributes->display( $attributes ); ?>>
```

BEM naming conventions

BEM is a naming convention for components.

We use it with a small modification: instead of block__element--modifier we use block-element--modifier.

For usual WorPress projects this change enhances readability. There will be less nesting, more folders and better transparency.

Instead of:

```
block
|. __element
|.. --modifier-for-element
|. --modifier-for-block
```

We have:

```
block
|. --modifier-for-block
block-element
|. --modifier-for-element
```

If this is not the preferred way to work the original BEM naming syntax can be easily restored:

```
class MoThemeHTMLComponentAttributes extends MoThemeHTMLComponent {
/**
 * Class arguments.
 *
 * Used to setup the class.
 *
 * @since 1.0.0
 *
 * @var array An array of arguments.
*/
public $arguments = array(
	...
	'element_prefix'  => '-',
	'modifier_prefix' => '--',
	...
);
```

More examples can be found in CSS.md

Extendable

Use apply_filters() and get_template_part() extensively. They both can be overwritten in child themes.

Filters

Filters receive data, modify data, and return data. They must be used when a piece of *data* has to be made extendable / customizable.

Since it is easy to add a filter it is recommended to be used as often as possible.

Example: (in theme)

```
$attributes = apply_filters(
	'mo_theme_post_excerpt_attributes',
	array(
		'block'    => 'post',
		'element'  => 'excerpt',
	)
);

<aside <?php $component->attributes->display( $attributes ); ?>>
```

```
        <div>
            <?php the_excerpt(); ?>
        </div>
    </aside>
```

In child theme:

```
    add_filter( 'mo_theme_post_excerpt_attributes',
    'mo_theme_post_excerpt_attributes_filter' );

    function mo_theme_post_excerpt_attributes_filter() {
        return array(
            'block'    => 'post-new-classname',
            'element'  => 'excerpt-new-classname',
        );
    }
```

Template parts

They are like mixins or partials in other coding frameworks. They do one small thing and they do it well. They are combined and reused freely to compose the user interface.

Template parts can be easily overwritten in child themes. If you have a home.php in your child theme it will overwrite the same template from the main theme.

home.php in the parent theme:

```
    <section <?php $component->attributes->display( $attributes ); ?>>
        <?php
            $component->title->display( $title );
            get_template_part( 'template-parts/post-list/post-list', '' );
        ?>
    </section>

    <?php
    get_footer();
```

home.php in the child theme:

```
    <section <?php $component->attributes->display( $attributes ); ?>>
        <?php
            $component->title->display( $title );
            get_template_part( 'template-parts/post-list/post-list', '' );
        ?>
    </section>
```

```
<?php
get_sidebar( 'sidebar-1' );
get_footer();
```

In the child theme we've just simply added a sidebar which was not present in the parent theme.

Actions

Actions are more abstract. They deal with *code* instead of data. They are used to let others insert code into an existing codebase.

The problem with actions is that you don't known apriori where others would like to insert their code.

You can add a before and after action for every element on your webpage (logo, header, content, footer, post list, article title, ...) and still you can't make everybody happy.
In this theme actions are used only when there is a concrete need for them. We don't insert empty actions hoping they will be reused later.

Single responsibility principle

Don't replace HTML code with PHP code

There is a tendency to get rid of HTML code in template parts. And use more PHP code instead.

This is wrong:

```
Hybrid\\View\display( 'index' );
```

We don't have a clue what that index template is: a <section>, a list (<ul>) or something else maybe? What is its class name or element id? How I can locate in the browser's web inspector? And how I can locate the style (css) or functionality (js) associated?

This is better:

```
<section class="home">
	<h3 class="hidden">Homepage</h3>

	<?php get_template_part( 'template-parts/post-list/post-list', 'with-comments' ); ?>
</section>
```

Here we've replaced only the HTML which is not significant for the component.

Replace ugly HTML code with PHP code

This is ugly:

```
<a class="link" href="<?php echo esc_url( home_url( '/' ) ); ?>" title="<?php echo
bloginfo( 'name' ); ?>">
	<span class="text">
		<?php bloginfo( 'name' ); ?>
	</span>
</a>
```

This is better:

```
echo wp_kses_post(
	sprintf(
		'<a class="link" href="%1$s" title="%2$s"><span
class="text">%2$s</span></a>',
		esc_url( home_url( '/' ) ),
		get_bloginfo( 'name' )
	)
);
```

We haven't hide any attribute of the link but made it better understandable and modifiable.

Semantic and outlined

The HTML source and outline is validated with the W3C validator

Outlining is very important since it makes the code accessible. If your site outlines well in the W3C validator it will get a 100% accessibility score in Google Lighthouse: https://imgur.com/a/MYSgMKH

CSS Best practices

Standards

Instead of plain CSS the SCSS best practice is used. SCSS is CSS with superpowers.
To comply with the site-wide Component architecture SCSS class names follow the BEM syntax.

Moreover the SCSS folder structure follows an old BEM recommendation:

- framework is a set of SCSS mixins reused across many different projects.
- pages is a set of mixins related to the pages of the site.
- parts is a set of mixins related to the components of the site.

- themes is a set of configuration files defining colors, fonts, vertical rhythm — anything describing how the theme looks.

Coupling

All these *practices* above are meant to help developers easily locate and modify code. If there is a HTML class then it should be a same name SCSS mixin.

For example, if in the browser's web inspector we have a <body class="home"> then we should have an scss/pages/home.scss mixin. For a <header class="header"> we should have a scss/parts/header.scss mixin.

Beside this HTML / SCSS coupling we also have PHP coupling. For home we have a home.php template. For header we have a template-parts/header/header.php template part.

Mo Plugin

A WordPress.org boilerplate plugin based on best practices.
It is taylor made to the Mo Pro Theme which means it's functionality is defined by the feature requests coming from the theme. On its own, without the theme enabled, the plugin does nothing.

Best practices

- Optimized and cached queries
- Functionality defined by the theme
- Presentation done in the theme

PHP Principles

Optimized and cached queries

Implementing 10up best practices like:

- Use WP_Query instead of get_posts.
- Query only what is necessary. Like 'update_post_meta_cache' => false when we don't need post meta data.
- Don't do endless queries like posts_per_page => -1.
- Always use the cache.

Single responsibility principle

Return data instead of HTML

There are two types of plugin functionalities: do something on the admin interface, and/or, provide extra data and features to themes.

When providing for themes — like in this case — always return data instead of displaying data.

Themes have a proper built in mechanism to display data — template parts — when plugins have nothing like this.

It is elegant and easy to return raw data to theme which displays it within it's own style guide than reinventing the template parts mechanism in the plugin.

Example

Adding a custom post type — people — cannot be done in a theme just in a plugin. A plugin has to be created for this feature.

To display a person in a post or a page the [person name="Bill"] shortcode can be used. To make the person look nice with avatar, role, email ... we need to use HTML.

The theme perhaps already has the template tags and parts displaying a person. Since plugins cannot use get_template_part they can't re-use that already written HTML code.

Solution

The plugin should return a global $person object and an action hook. The theme should use the global variable and the hook to display the person with the existing template parts.

Plugin:

```
function person_func( $atts ){
    global $person;
    $person = get_person();

    do_action( 'display_person_in_theme' );
}
add_shortcode( 'person', 'person_func' );
```

Theme:

```
function display_person() {
    global $person;

    get_template_part( 'template-parts/person' );
}
add_action( 'display_person_in_theme', 'display_person' );
```

Loose coupling

Decouple plugin and theme using add_theme_support

The implementation of a custom plugin should be decoupled from its use in a Theme.

Disabling the plugin should not result in any errors in the Theme code. Similarly switching the Theme should not result in any errors in the Plugin code.

from 10up Engineering Best Practices

How it works?

1. Features the theme needs from the plugin can be declared via an add_theme_support('custom-features'); call.
2. The plugin checks for these features via if (current_theme_supports('custom-features')) {}.
3. If features are requested the plugin enables and implements it.

The above checks must be wrapped into an after_setup_theme hook where the execution priority in the plugin must be higher than in the theme.

Theme:

```
add_action( 'after_setup_theme', 'define_theme_support', 10, 0 );
```

Plugin:

```
add_action( 'after_setup_theme', 'check_theme_support', 11, 0 );
```

The naming convention

The custom-features variable must be shared between the theme and the plugin. For that we have the default global $_wp_theme_features array. This makes it easy to add custom features into this array in the theme, then query them in the plugin.

To mimic the WordPress default post-formats feature we pass the custom features as an array:

Theme:

```
	add_theme_support(
		'custom-features',
		array(
			'feature-1',
			'feature-2'
			)
		)
	);
```

Plugin:

```
	if ( current_theme_supports( 'custom-features' ) ) {
		$features = get_theme_support( 'custom-features' );
		if ( $features['feature-1'] ) {
			....
		}
	}
```

The activation hook workaround

Every plugin has a register_activation_hook where the plugin features are set up. This hook is executed before any theme code is executed making the custom-features coming from the theme unavailable at the moment of the plugin activation.

A workaround is to re-call the plugin activation code after the theme sets up and pass the custom-features variable to the plugin.

Mo Pro Theme

A child theme of the Mo Theme using the Mo Plugin functionalities.

Best practices

- Decoupled from the plugin
- Easily extends the parent theme

PHP and HTML Best practices

By definition Mo Pro Theme extends Mo Theme and uses Mo Plugin by:

1. Displaying the shortcode added by the Mo Plugin.
2. Extending the homepage from Mo Theme by adding a sidebar.
3. Displaying a widget in the sidebar with custom post types added by the Mo Plugin.

These features, and only these features are present in this child theme:
1. Displaying the shortcode is done with CSS and a new template part for the book post type:

```
.
└── post-list
    └── post-list-with-external-query.php
```

1. Adding sidebar is done in homep.php
2. Displaying the widget is done is /includes:

```
.
├── class-moprotheme-setup.php
└── theme-functionalities
    ├── class-moprotheme-custom-widget.php
    └── class-moprotheme-functionalities.php
```

CSS Best practices

In the same way like in the case of PHP / HTML code the child theme overwrites / extends the parent theme only with what's necessary.

The structure of the scss folder is:

```
.
└── parts
    ├── content
    │   └── content.scss
    ├── post-content
    │   └── post-content.scss
    └── sidebar
        └── sidebar.scss
```

The child theme adds two additional functionalities to the parent theme: displays a shortcode inside a post and a widget in the sidebar.

- In content.scss we add the sidebar entry.
- In post-content.scss we style the shortcode.
- In sidebar.scss we style the widget and the sidebar.