The WordPress way, the modern way: developing as if it were 2016

You are not doing it wrong.

But you could do it better.

Slides at http://goo.gl/woxuvT

About me

- Freelance WordPress Developer, about 3 years of experience with WordPress and PHP
- @lucatume on Twitter and GitHub
- http://theaveragedev.com

I don't care about PHP, JavaScript is the way to go!

- do not run away from PHP: the REST API is awesome but there is still much to say about PHP
- bad habits and practices will follow you across languages and frameworks
- the title can be used for anything: "I don't care about Star Wars, Star Trek is the way to go!"

Not even having this discussion.

"WordPress as a PHP framework" vs "WordPress is a CMS"

- provides a wealth of functions I do not have to develop myself
- to me it's a PHP framework **and** a CMS
- the "WordPress way" is **one way** to develop meant to be accessible and back compatible

But it's not the **only** way.

We can have a polite discussion later

Disclaimer

- there will be code
- the purpose of the presentation is to give you ideas, not truths
- I'm expressing opinions, not axioms

This said...

The PHP 5.2 disclaimer

- PHP 5.2 is unsupported since Jan 6, 2011
- **but** I will use PHP 5.2 compatible code in the examples (give me a second)
- you should **use a supported version of PHP for your projects!**

PHP 5.2 limits my amazing development skills

- not my case 95% of the times
- it's **your** PHP code, it's not WordPress' code; take responsibility
- language version and techniques are **different** matters
- sounds like saying **"not having a Millenium Falcon limits my driving skills"**

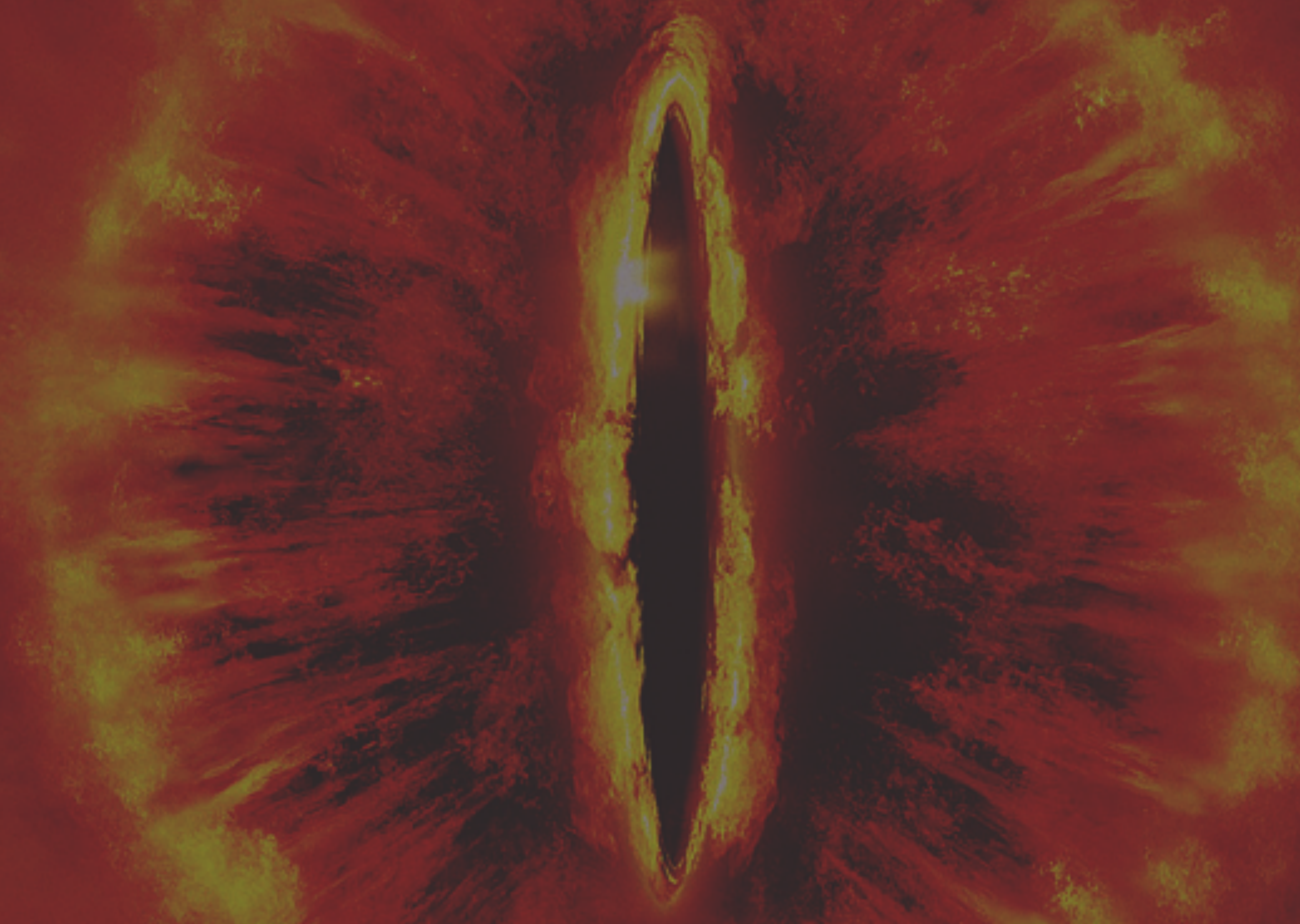
This is not the excuse you are looking for.

Developing is like going to war

- you won't be home by Christmas
- it will escalate
- you will lose the public support when it goes wrong

Prepare for it and put your ducks in a row: start writing better code today.

The mandatory joke about project managers.



SOLID and DRY principles vs "it works"

- Or "Why should I care?"
- Or "The moment you realize you should have attendend the other speech."

SOLID principles for solid code - 1

- **Single responsibility** - one thing well
- **Open/closed** - extending in place of modifying
- **Liskov substitution** - it should work as its parent

SOLID principles for solid code - 2

- **Interface segregation** - if you call it "Manager" you are doing it wrong
- **Dependency inversion** - type hint interfaces

This is the only way I can remember them.

Single responsibility - before

```
class CustomPostTypes {

    public function __construct(){
        add_action('init', array($this, 'register_custom_post_types'));
        add_action('save_post', array($this, 'on_save_post'));
        add_filter('the_title', array($this, 'the_title'), 10 ,2);
    }

    public function register_custom_post_types() { //... }

    public function on_save_post($post_id){ //... }

    public function the_title($title, $post_id) { //... }
}
```

The class is doing a lot of different things.

Single responsibility - after

Let's add an Hooks class and move different functions into different smaller classes.

```
class Hooks {
    public function __construct(){
        $post_types = new Custom_Post_Types_Register();
        $save_operations = new Save_Operations();
        $template_tags = new Template_Tags();

        add_action('init', array( $post_types, 'register_custom_post_types') );
        add_action('save_post', array( $save_operations, 'on_save_post') );
        add_filter('the_title', array( $template_tags, 'the_title'), 10 ,2);
    }
}
```

Each class does **one** thing.

Open/closed - before 1

```
class Template_Tags {
    public function the_title( $title ) {
        return '.:: ' . $title . ' ::.';
    }
}
```

"We might need to append a smiley face to the title when dealing with the bad_news post type..."

Open/closed - before 2

Well: that's an easy modification...

```
class Template_Tags {
    public function the_title( $title, $post_id = null ) {
        $post = get_post( $post_id );
        if ($post->post_type === 'bad_news') {
            return $title . ' ;)';
        } else {
            return '.:: ' . $title . ' ::.';
        }
    }
}
```

Many more modifications will come...

Open/closed - after 1

```
class Template_Tags {
    public function the_title( $title, $post_id ) {
        return $this->prefix( $post_id )
        . $title
        . $this->postfix( $post_id );
    }

    protected function prefix(){
        return '.:';
    }

    protected function postfix() {
       return ':.';
    }
}
```

Open/closed - after 2

```
class Smarter_Template_Tags extends Template_Tags {

    protected function prefix( $post_id ){
        $is_bad_news = get_post( $post_id )->post_type === 'bad_news';
        return $is_bad_news ? '' : '.: ';
    }

    protected function postfix( $post_id ) {
        $is_bad_news = get_post( $post_id )->post_type === 'bad_news';
        return $is_bad_news ? ' ;)' : ' :.';
    }

}
```

Leveraging the decorator pattern can extend flexibility even further.

Liskov substitution - before 1

```
class Save_Operations {

    protected $published_counter;

    public function __construct( Published_Counter $published_counter ) {
        $this->published_counter = $published_counter;
    }

    public function on_save_post( $post_id, WP_Post $post, $update ) {
        if( $update ) {
            return;
        }

        if ($post->post_status === 'published') {
            $this->published_counter->increase_count(1);
        }
    }

}
```

Liskov substitution - before 2

```
class Filtering_Save_Operations extends Save_Operations {

    protected $post_counter

    public function __construct( Post_Counter $post_counter ) {
        $this->post_counter = $post_counter;
    }

    public function on_save_post( $post_id, WP_Post $post, $update ) {
        if ($post->post_type === 'post') {
            $this->post_counter->increase_count(1);
        }
    }
}
```

This is **not** doing what the parent class does and more: it's just doing different stuff.

Liskov substitution - after

```
class Filtering_Save_Operations extends Save_Operations {

    protected $post_counter;

    public function __construct( Published_Counter $published_conter,
        Post_Counter $post_counter = null) {
        parent::__construct( $published_counter );
        $this->post_counter = $post_counter;
    }

    public function on_save_post( $post_id, WP_Post $post, $update ) {
        parent::on_save_post($post_id, $post, $update);

        if ($this->post_counter && $post->post_type === 'post') {
            $this->post_counter->increase_count( 1 );
        }
    }
```

Interface segregation - before

```
interface Post_Repository_Interface {
    public function get_post( $post_id );
    public function insert_post( array $data );
    public function update_post( $post_id, array $data );
    public function delete_post( $post_id );
    public function regenerate_post_cache( $post_id );
    public function get_cached_post( $post_id );
    public function on_post_insert ( $post_id );
    public function on_post_update( $post_id );
    public function on_post_delete( $post_id );
}
```

A legitimate post repository.

Interface segregation - after

```
interface Post_Repository_Interface {
    public function get_post( $post_id );
    public function insert_post( array $data );
    public function update_post( $post_id, array $data );
    public function delete_post( $post_id );
}

interface Post_Cache_Interface {
    public function regenerate_post_cache( $post_id );
    public function get_cached_post( $post_id );
}

interface Post_Events_Listener_Interface {
    public function on_post_insert ( $post_id );
    public function on_post_update( $post_id );
    public function on_post_delete( $post_id );
}
```

Interface segregation - Why?

- client classes can now depend on smaller abstractions
- implementations can now be changed as the code evolves
- the class can now be split with no repercussion on client classes

This is about abstractions, the "S" in "SOLID" is about implementations.

Dependency inversion - before

```
class Like_Handler {

    protected $post_repository;

    public function __construct( Post_Repository $post_repository ) {
        $this->post_repository = $post_repository;
    }

    public function handle_request( $post_id ) {
        if( !$this->post_repository->get_post( $post_id ) ) {
            return;
        }

        $data = array(
            'post_type' => 'like',
            'post_parent' => $post_id,
        );

        $like_id = $this->post_repository->insert_post( $data );

        if (!empty ( $like_id ) ) {
            $this->post_repository->regenerate_post_cache( $like_id );
        }

        return $like_id;
    }

}
```

Dependency inversion - after

```
class Like_Handler {

    protected $post_repository;
    protected $post_cache;

    public function __construct( Post_Repository_Interface $post_repository, Post_Cache_Interface $post_cache ) {
        $this->post_repository = $post_repository;
        $this->post_cache = $post_cache;
    }

    public function handle_request( $post_id ) {
        if( !$this->post_repository->get_post( $post_id ) ) {
            return;
        }

        $data = array(
            'post_type' => 'like',
            'post_parent' => $post_id,
        );

        $like_id = $this->post_repository->insert_post( $data );

        if (!empty ( $like_id ) ) {
            $this->post_cache->regenerate_post_cache( $like_id );
        }

        return $like_id;
    }

}
```

Dependency inversion - Why?

- build **layers** of functionalities in place of **silos**
- put information about what a class does in the interface, not the implementation
- again this is about abstraction; the "D" opens the stage for the "S" in "SOLID" again

I will not repeat myself.
I will not repeat myself.
I will not repeat myself.
I will not repeat myself.
I will not repeat myself.

TXT2PIC.COM

Don't Repeat Yourself

- stick to the principles above and it should not happen
- if it happens you are `doing_it_wrong`
- classes that do too much, methods that do too much, too much dependency knowledge
- copying and pasting **is not** reusing code

Autoloading VS require/include

- the biggest cost of development is developer time
- change a dependency, change `require` or `include` statements in all the code base (think of the principles above)
- performance may vary, autoloading will pay off on bigger projects

Composer to autoload PHP 5.2 compatible projects

- Composer is a dependency manager for PHP written in PHP
- it also packs an amazing autoloader generation possibility
- using the `xrstf/composer-php52` package you can use autoload in PHP 5.2 projects

Composer - an example plugin configuration file

```
{
    "name": "lucatume/my-plugin",
    "description": "A plugin of mine",
    "type": "wordpress-plugin",
    "require": {
        "xrstf/composer-php52": "^1.0",
    },
    "autoload": {
        "psr-0": {
            "MyPlugin_": "src/"
        }
    },
    "scripts": {
        "post-install-cmd": [
            "xrstf\\Composer52\\Generator::onPostInstallCmd"
        ],
        "post-update-cmd": [
            "xrstf\\Composer52\\Generator::onPostInstallCmd"
        ],
        "post-autoload-dump": [
            "xrstf\\Composer52\\Generator::onPostInstallCmd"
        ]
    }
}
```

Composer - using the autoloader in a plugin

```php
<?php

/**
 * Plugin Name: My Plugin
 * Plugin URI: http://theAverageDev.com
 * Description: A plugin of min.
 * Version: 1.0
 * Author: theAverageDev
 * Author URI: http://theAverageDev.com
 * License: GPL 2.0
 */

include 'vendor/autoload_52.php';

/**
 * Do what plugins do...
 */
```

Dependency Injection VS hardcoded dependencies

- the SOLID principles made it clear that dependencies should be injected
- a "dependency" is a variable; typically an object instance
- "injecting a dependency" means this variable is passed to the class using its `__construct` method or a "setter" method

Hardcoded dependency

```
public function on_save_post( $post_id, WP_Post $post, $update ) {
    if( $update ) {
        return;
    }

    if ( $post->post_status === 'published' ) {
        $published_counter = new Published_Post_Counter();
        $published_counter->increase_count( 1 );
    }
}
```

What if I need to change that Published_Post_Counter dependency?

And let's not talk about globals...

Injected dependency

What we have seen above (edited for brevity).

```
public function __construct( Published_Counter_Interface $published_counter ) {
    $this->published_counter = $published_counter;
}

public function on_save_post( $post_id, WP_Post $post, $update ) {
    if ( $post->post_status === 'published' ) {
        $this->published_counter->increase_count( 1 );
    }
}
```

I can now replace that dependency implementation and control it.

Dependency Injection Containers

So...

- any dependency should be injected
- any dependency should type hint an interface
- I should be able to change what concrete implementation (a `class`) is bound to an abstraction (an `interface`)

Bootstrap file - without a Dependency Injection Container

I will have **huge** initialization files

```
$published_counter = new Published_Counter();
$post_counter = new Post_Counter();
$post_repository = new Caching_Post_Repository();

$save_operations = new Filtering_Save_Operations( $published_counter, $post_counter );
$like_handler = new Like_Handler( $post_repository, $post_repository );

$hooks = new Hooks( $save_operations, $like_handler );
```

Think about having these injections scattered across code...

Bootstrap file - the problems

- imagine keeping this up to date
- every instance is created up front and it might not be used
- each and every class instance has to be specified

Bootstrap file - with a Dependency Injection Container 1

```
$container = new MyPlugin_Container();

$container->bind( 'Published_Counter_Interface', 'Published_Counter' );
$container->bind( 'Post_Counter_Interface', 'Post_Counter');
$container->singleton( 'Post_Repository_Interface', 'Caching_Post_Repository');
$container->singleton( 'Post_Cache_Interface', 'Caching_Post_Repository' );

$hooks = $container->make( 'Hooks' );
```

The example uses the DI52 package.

Bootstrap file - with a Dependency Injection Container 2

- provide bound implementations only when needed (lazy instantiation)
- no more singleton methods
- automatic class resolution
- always know what depends on what

ARE YOU
A CONTROL FREAK?

Closures VS ...nothing

There are no closures in PHP 5.2; no code like this.

```
function apply_parent_to_all( array $posts, $parent_id ) {

   array_map( function( WP_Post $post ) use( $parent_id ) {
        $post->post_parent = $parent_id;
        wp_update_post( $post );
   }, $posts );

}
```

Damn... I had my faith restored for a moment.

I know what you are thinking...

KEY FEATRURES

PHP 5.3 - 5.6

PHP 5.2 closures work-around

You can use foreach **or** something like this

```
public function apply_parent_to_all( array $posts, $parent_id ) {
    $this->_parent_id = $parent_id;
   array_map( array($this, 'update_post_parent' ), $posts );
}

protected function update_post_parent ( WP_Post $post ) {
    $post->post_parent = $this->_parent_id;
    wp_update_post( $post );
}
```

Test-driven development VS cowboy coding

- if the way you develop works for you I'm fine with it and you should be too
- TDD is a **developer tool** that will reflect on the final user in the long run
- TDD **will enforce** all of the above on your code

I use TDD as I'm a crappy "cowboy coder".

YOU ARE A CRAPPY COWBOY.

AND A CRAPPY CODER.

imgflip.com

TDD setup in WordPress for PHP 5.2 code

- see Composer and autoloading section above
- your test code can run on PHP 7: it's your production code that needs to be PHP 5.2 compatible
- I use wp-browser (and function-mocker when things get difficult)

PHP 5.3 is coming!

You know what? All of this in PHP 5.2!

Think of the amazing possibilities you will have when PHP 5.3 comes out!

Thanks

- Jonathan Brinley and Daniel Dvorkin for reviewing this speech
- Modern Tribe for supporting me and being generally cool people
- You for your time and patience

Questions?