WP DNA Documentation

WP DNA is a plugin that makes it easier for developers to write new plugins for Wordpress.

Foundation

WP DNA provides:

- **Plugin inheritance**
  Plugins based on the WP DNA boilerplate inherit many functions and properties handled by the WP DNA plugin automatically. Boring and repetitive development tasks, such as registering a new custom post types, building their related admin. forms, saving and loading custom data, and displaying that data in the front-end are all made much easier by using just a minimal set of function calls, as most of the code is already written in the parent plugin.
- **Advanced plugin boilerplate**
  By using Tom McFarlin's original boilerplate, we created an advanced boilerplate that not only sets up the perfect file structure and lays out a template for fundamental Wordpress hooks needed by most plugins, but is truly built to make Wordpress plugin development as fast as possible.
- **Object-oriented wrappers**
  WP DNA provides an object-oriented structure for your new software. Instead of registering custom post types, writing procedural code to manage them, and having to deal with the Wordpress ecosystem and its numerous requirements, simply write a new Class that inherits our wrapper. Declare your Class attributes, and they will automatically be converted to WP_Post fields and magically appear in the admin. forms.

How to a create a new plugin?

Two ways:

Option 1:

Use the "Generate child plugin" feature in WP DNA

Option 2:

Manually download the plugin boilerplate and:

- Rename the two following files to your chosen plugin name:
  - wp-my-plugin.php
  - includes/class.wp-my-plugin.php
- Do a case-sensitive search on the entire plugin, and replace the following strings with your chosen plugin name. For example, if your chosen plugin name is "Example App", you would operate the three following replacements:
  - WP MY PLUGIN → EXAMPLE APP
  - WP_MY_PLUGIN → EXAMPLE_APP
  - wp-my-plugin → example-app

Understanding the boilerplate's structure

/admin → All files dealing with the Wordpress back-end

/public → All files dealing with the Wordpress front-end

/includes → All plugin core files
The files below contain the code needed to plug into the Wordpress ecosystem. Most of the file names are self-explanatory, but if you need extra information, refer to Tom McFarlin's writings on the original Boilerplate's website at: WordPress Plugin Boilerplate Generator

- class-activator.php
- class-custom-fields.php
- class-deactivator.php
- class-display.php
- class-i18n.php
- class-loader.php
- class-post-types.php
- class-shortcodes.php
- class-wp-my-plugin.php

/includes/app → Most of your custom code

- helpers.php → general utilities

/includes/app/objects → This is where you'd place your custom Classes, and Classes representing a custom post type have to inherit our WP Wrapper class.

- {my-custom-class.php}

How and where to write my custom Classes?

Custom classes should be written to **/includes/app/objects**

Our idea is for developers to be able to create object-oriented code that is independent of the Worpdress ecosystem. We've developed a framework in which only two things need to happen in order for this to be possible:

1. Inherit from our Wrapper:
   Classes that seek to handle your custom post types will need to inherit from our WPwrapper_post Class.
   Classes that seek to extend Wordpress users will need to inherit from our WPwrapper_user Class.

   `MyCustomClass extends \WP_DNA\wpwrapper_post`
2. Map your Class attributes with Wordpress fields:
   The only code that needs to be written in order to connect your Custom Class with Wordpress is what we call the $map.

The $map is an array that maps WP Post fields with Class attributes.

Here's an example taken from the "API" Class from the WP API Central plugin:

```
public static function loadMap() {

 $scope = 'wac-api';

 self::addField($scope, 'name', [
  "key"  => "post_title",
  "title"     => "Name",
  "type"      => "text"
 ]);

 self::addField($scope, 'description', [
  "title"     => "Description",
  "type"      => "textarea",
 ]);

 self::addField($scope, 'environment', [
        "title"     => "Environment",
  "type"      => "select",
 ]);
```

The $scope is the slug of the custom post type associated with the Class.

When creating a new API object, it will automatically be loaded with data pulled from the wp_post and wp_post_meta tables.

`$my_api = new API($post_id);`

You will then be able to access the object's variables with a simple:

`$my_api->get('description');`

The $map will also automatically generate the associated admin. form for you to add, edit and delete your objects. In this example, by creating the custom Class "API", you would automatically see a new CRUD form appear in the Wordpress backend, enabling you to add and edit "APIs".

The $map in details

The $map of loading in a static function inside your Class.

The function signature is:

```
public static function loadMap()
```

Only two functions are available to build the map:

- addField($scope, $field, $params)
- addGroup($scope, $field, $params)

$scope is the custom post type' slug.

$field is the slug of the field

$params can contain the following elements:

- title < Free string >
- description < Free string >
- type < Field type > (see section below)
- default < Default value >
- parent < Parent field >
- controller < Controller info > (see section below)

Field types:

- **text**
  Short string
- **textarea**
  Long string
- **wysiwig**
  Markup string (uses the Wordpress Tiny MCE Editor)
- **checkbox**
  Checkbox
- **select**
  Dropdown
- **radiobutton**
  Radio buttons
- **group**
  This isn't really a field, but its purpose it to enable you to group fields into sections, which is very useful if you want to be able to conditionally show/hide a group of fields

Controller details

When setting up a field, you can specify a controller. The controller will enable you to conditionally show/hide a field (or group) based on a particular value.

For example, if you had a custom post type named "pet", and wanted the "Dog breed" dropdown to show only if the value of the "`Species`" dropdown was set to "Mammal", then you would write this:

```
self::addField('pet', 'species', [
 "title"        => "Species",
 "type"         => "select",
]);

self::addGroup('pet', 'dog_details', [
 "title"        => "Dog details",
 "controller"    =>  [
  'field' => 'species',
  'value' => 'dog'
 ]
]);

self::addField('pet', 'dog_breed', [
   "title"        => "Dog breed",
   "type"         => "select",
   "parent"       => "dog_details",
]);
```

Loading data into your select and radiobuttons fields:

In the class-custom-fields.php file, you'll find a function named loadData(). The boilerplate offers an example of how to successfully load data.

The process is very simple, you are simply loading an associative array into the 'options' cell of your field. For example:

```
$dog_breeds['options'][] = array('option_name' => 'bulldog', 'option_value' =>
'Bulldog');
$dog_breeds['options'][] = array('option_name' => 'labrador', 'option_value' =>
'Labrador');
```