Fluent Framework

No Half Pixels Ltd

Here we aim to provide you with concise, up to date documentation for the Fluent Framework and its classes.

If there is anything missing, hard to understand or could be improved, please let us know, or make an edit request and send it to us. Any help we get from the community is very much appreciated.

01 Introduction

The Framework has been designed to be as simple as possible, lightweight and fast. So the documentation is pretty much the same.

To the Right you will see a menu, all documentation pages will be listed here, and/or linked to at appropriate places.

Below is the file list for the framework and what they include*

A full PHP Documentor based class list and definition can be found here: http://no-half-pixels.github.io/Fluent-Framework/docs

*Not a definitative list, but covers the main files

- readme.txt – normal plugin readme file
- readme.md – replication of the above
- package.json – this is an npm dependancy manager file which tells npm which assets to install during development
- languages/* – contains the translatable language string files (.po .mo) for the plugin
- Gruntfile.js – this file details all of the grunt tasks when developing
- fluent-framework.php – main "bootstrap" file. This file loads all the required classes, and also registers all the base field types into the options classes
- classes/class.fluent-base.php – base class, all other classes generally extend this as it provides a few helper methods
- classes/class.fluent-demo.php – this file is loaded if `define('FLUENT_DEMO', true);` is defined and will create an instance of each type with all field types to demo the framework
- classes/class.fluent-options-meta.php – this file loads the metabox class which interprets the options for use in post type meta boxes
- classes/class.fluent-options-page.php – this file loads the options page class which uses the options for a custom options pages
- classes/class.fluent-options-taxonomy.php – this file loads the options class which uses the options for displaying fields on taxonomy add/edit pages
- classes/class.fluent-options-user.php – as above but designed for showing fields on the user profile pages
- classes/class.fluent-options.php – this is the meat of the framework and contains all the base logic for using the options/fields in any context
- classes/class.fluent-page.php – this is a helper class which can create options pages, its is used by the class.fluent-options-page.php class
- fields/* – contains all of the field classes which are registered and used.
- assets/* – contains css/scss/js developed for the framework
- assets/vendor/* – contains any vendor assets, currently delta-ui theme and selectize.js for the select boxes (licenses included where applicable)

There may from release to release be files includes, but not loaded, these may be new features being added incrementally.

We would also like to explain the Gruntfile.js, package.json, and scss folders aren't required for the plugin to function. These are development files and should be ignored by most users. We have included them by default to allow the brave to edit the framework if they feel they can add to it. We don't want to make things harder for people to contribute so rather than a end user and developer download we have 1 simple download that has everything. The files are so small and aren't used during normal operation so performance is not effected.

01.1 Requirements

The Fluent Framework has been tested with WordPress 3.8 and up.

The styling of the framework relies heavily on 3.8 and up css, so it is unlikely it will look right on older version. It may possibly work with the MP6 plugin installed, but this is not tested and not on our list of things to test.

We are providing support for 3.8 and up so if any issues are encountered with this requirement please let us know.

PHP 5.2 is required by WordPress and there is nothing in the framework which would effect this.

HTML5 is used and CSS3 were applicable, all with degradation so older browsers will still function.

We are providing browser support for IE9 up and the usual browsers, IE8 support is provided as functionality only and may be dropped in future.

Then all that is required is a user with access to uploading plugins (or themes if enclosed in a theme).

01.2 Support

All support queries should be logged via the issue tracking system here at Github. Our code is actually hosted on private git servers, but we feel github has the best issue tracking which can be made public for users.

PLEASE DO NOT POST LOGIN DETAILS OR SENSITIVE INFORMATION IN ISSUES!

If the issue requires further investigation we can offer alternative methods of communication when its appropriate.

02 Installation

You can install/use the Fluent Framework in two ways. As a standalone/required plugin, or by including the framework within a theme or plugin. Below are the steps for both methods.

Once installed/included you can use the framework classes as needed.

02.1 Install as a plugin

1. Upload the `fluent-framework` folder to the `/wp-content/plugins/` directory
2. Activate the plugin through the ‘Plugins’ menu in WordPress

02.2 Include within a theme/plugin

1. Copy the `fluent-framework` folder into the theme/plugin folder
2. Include the `fluent-framework.php` file, like so:

```
include_once(dirname(__FILE__).'/fluent-framework/fluent-framework.php');
```

1. Define the framework url like so (obviously providing the correct url to the framework folder)*:

```
Fluent_Base::$url = 'the url path to the framework';
```

**It is important to do this step after including the framework main file to register the right location.*

03 Configuration

Configuration of the framework is by design provided in the same manner across all instance types.

Each instance type may have more or less options depending on what is required to get it working, any differences are details on the instance specific pages.

In the Plugin you will find the file `example.php` which contains an example of how to create instances.

03.1 The Arguments Array

Below is the base configuration arguments array which is the same across all instances:

```
$args = array(
    'option_name' => 'option_name',//the name of the option/meta_key to save against
    'dev_mode' => false,//boolean to turn n dev mode or not
    'default_layout' => 'options-normal',//the default layout to use*
);
```

**this is saved against a user just like normal screen options, so once loaded or changed by the user this has no effect*

In all instances the args array above is set by default, If you add any of these arguments in your instance they will overwrite the above defaults.

03.2 The Sections Array

The sections array is as the arguments above required by every instance and this is where you define your sections, which contain the fields/options you desire. Details of each field configuration can be found on the Field Page of this wiki.

Depending on the instance type this will be rendered as either metaboxes, and can use the metabox priority and context options, or on other instance types they will be displayed one after another.

Below is an example section array:

```
$sections = array(
    'first_section' => array(//the unique id of the section
        'dash_icon' => 'list-view', //if set this will add a dash icon which can be
sourced here: https://github.com/melchoyce/dashicons
        'title' => __('Section Title', 'domain'),//the title of the section
        'description' => __('Section Description', 'domain' ),//the description of
the section
        'context' => 'normal',//when sections are displayed as a metabox where on
the page the metabox should display (normal,side,advanced)
        'priority' => 'high',//the priority of the section (if its displayed as a
metabox)
        'fields' => array(),//an array of the fields this section contains
        'roles' => array('administrator', 'editor'),//default = false. If set the
current user must be in one of these roles to access the setting
        'caps' => array('edit_posts'),//default = false. If set the current user
must have all these capabilities
    ),
);
```

As of v1.6.0 we have added another element to the sections array called `tabs`. Tabs allow you to convert the sections fields table into a tabbed interface when used on metabox style pages (options pages and metaboxes).

When using tabs you move your `fields` array into the corresponding tabs array like so:

```
$sections = array(
    'first_section' => array(//the unique id of the section
        ....../other attributes
        'tabs' => array(
            'tab_id' => array(
                'title' => 'the tab title',
                'description' => 'the tabs description',
                'fields' => array(),//an array of the fields this tab contains
            ),
        ),
    ),
);
```

You can define as many tabs as you like, however there is limited space in metaboxes and the width of the tab links is dependant of the titles of each tab, so always check how the tabs display.

Even though this feature is only designed for metabox type instances (options pages and metaboxes) you can define your sections array like this for the other field types, they will just order the fields in the order they are added to the tabs, so there is no problem with cross compatibility.

03.3 Creating Instances

Every class is created by calling the class with the `$args` and `$sections` arrays like so:

```
$instance = new Fluent_Options_Page($args, $sections);
$instance = new Fluent_Options_Meta($args, $sections);
$instance = new Fluent_Options_User($args, $sections);
$instance = new Fluent_Options_Taxonomy($args, $sections);
```

03.4 Meta Boxes

The Meta Box Class allows you to create an options instance which will be used on post type add/edit screens. This class supports built in post types like posts and pages, and also supports custom post types. It renders each section as a metabox in the location set for each section by the `context` and `priority` keys.

In addition to the arguments passed for any instance type the meta box class requires you to define an extra key in the array of `post-types`:

```
$args = array(
    //default keys...
    'post-types' => array(
        'post' //this is default
    )
);
```

As you can see by default this is set to add the metabox to the post type of post, this can be overridden and any number of post types can be defined.

03.5 Options Page

This Options Page Class allows you to utilise the same layout/design as the metabox class, but on its own custom options page.

For an Options Page instance you need to define a few options which will be passed to the page creation helper class `Fluent_Page`.

These arguments need to passed into the `$args` array via the key `page_args`, below are the defaults sent, any of which can be overwritten:

```
$args = array(
    //default keys...
    'page_args' => array(
        'slug' => 'options',//the unique slug of the page
        'menu_title' => __( 'Options', 'domain' ),//the title in the sidebar menu of
the page
        'page_title' => __( 'Options', 'domain' ),//the page title when rendered
        'parent' => '',//a string referencing the parent menu item if any, e.g:
'admin.php?page=somepage'
        'cap' => 'manage_options',//the capability of users who can access this page
        'priority' => null,//the menu item priority
        'menu_icon' => '',//the dash icon or url to image icon for use in the menu
(only for top level pages)
        'page_icon' => 'icon-themes',//the dash icon if any to use on the page when
rendered
        'callback' => false,//the page render callback
        'network' => true//enable the page on the network admin
    )
);
```

Generally speaking the only ones you will need to access are the `slug`,`menu_title`,`page_title` and maybe the `icon` keys.

03.6 User Meta

The User Meta Class allows you to add custom sections and fields to your profile, and any other user profile admin pages.

Each section will be displayed as a heading, description text and a table of the form fields in keeping with the look and feel of the page. This means you don't need to worry about the `priority` and `context` options for each section.

The User Meta Class doesn't require any special arguments and works with just the default set as detailed in the main configuration wiki page.

03.7 Taxonomy Meta

The Taxonomy Options Class allows you to create groups of fields and options for taxonomies. This class supports built in taxonomies like categories and tags, and also supports custom taxonomies. Each section will be displayed as a heading, description text and a table of the form fields in keeping with the look and feel of the page. This means you don't need to worry about the `priority` and `context` options for each section.

In addition to the arguments passed for any instance type the taxonomy class requires you to define an extra key in the array of `taxonomies`:

```
$args = array(
    //default keys...
    'taxonomies' => array(
        'tags' //this is empty by default
    )
);
```

You can add as many taxonomies to the array and the instance will be rendered on each one.

03.8 Fields

As each field type is so different they generally all have a few configuration options that are specific, below is an example of what is required for every field type, regardless of what extra the field needs. Field specific configuration details can be found below underneath all the general configuration data.

```
$sections = array(
    //...
    'fields' => array(
        //this is the field definition
        'field_unique_id' => array(
            'type' => 'text',//the field type
            'title' => __('The Field Title', 'domain'),//field title
            'sub_title' => __('', 'domain'),//field sub title - optional
            'required' => true|false,//false by default, will prevent form
submission and show error if not supplied when set to true - optional
            'required_message' => __('', 'domain'),//custom error message if desired
- optional
            'description' => __('', 'domain'),//description - optional
            'default' => string|array|int,//a default value for the field depending
on the field type* - optional
            'separate' => false|true,//default = false. If set to true this field
will be saved in the main array and as its own item under the key
$option_name_$field_id
            'roles' => array('administrator', 'editor'),//default = false. If set
the current user must be in one of these roles to access the setting
            'caps' => array('edit_posts'),//default = false. If set the current user
must have all these capabilities
            'conditions' => array(//default = false. If set this field will only
show if any of the following condition arrays are matched
                array(//this is a condition array, all conditions within it must
equal true for the condition to be met
                    array(//this is a condition
                        'id' => 'blocked_field_text',//the field id to match against
                        'value' => 'match',//the matched value
                        'type' => 'contains'//the type of condition. default '=='
                    ),
                    array(...)//multiple conditions per condition array can be made,
this acts like an "if and" statement
                ),
                array(...)//another conditions array, this acts like an "if/or"
statement
            )
        )
    )
);
```

*When providing a default value for group multiple values you must make sure you provide a uuid key for each array index, this can be achieved by using the `Fluent_Base::guid();` function.

03.9 Field Conditions

The conditions array can be used to hide/show fields based on other values. This is great when you want to hide features, if a checkbox is unchecked, or options require further options to be used.

Above is the syntax for creating conditions, and below are the possible condition types:

```
'==' = equals, this is default
'!=' = doesnt equal
'required' = must not be empty
'contains' = contains the value
'starts_with' = starts with the value
'ends_with' = ends with the value
'!contains' = doesnt contain the value
'>' = more than the value
'>=' = more than or equal to the value
'<' = less than the value
'<=' = less than or equal to the value
'between' = between the values "supply the value with a pipe '|' between the two
numbers" e.g. '10|20'
'!between' = not between the values "separated by pipe as above"
```

04 Fields

Each field type has the configuration options detailed above, but some field types have additional configuration values, those fields are detailed below.

04.1 Text

The text field has the following additional attributes:

```
array(
    //...
    'type' => 'text',
    'classes' => array(
        'large-text'//default
    ),
    'placeholder' => ''//placeholder attribute text
)
```

04.2 Background

The Background field has he following additional attributes:

```
array(
    //...
    'type' => 'background',
    'upload_title' => __('Select Media', $self->domain),
    'media_title' => __('Media Library', $self->domain),
    'media_select' => __('Select Media', $self->domain),
    'value' => array(
        'color' => '',
        'transparent' => false,
        'background-image' => '',
        'background-attachment' => '',
        'background-position' => '',
        'background-size' => '',
        'background-repeat' => 'repeat'
    )
)
```

04.3 URL

The url field has the following additional attributes:

```
array(
    //...
    'type' => 'url',
    'classes' => array(
        'large-text'//default
    ),
    'placeholder' => ''//placeholder attribute text
)
```

04.4 Password

The password field has the following additional attributes:

```
array(
    //...
    'type' => 'password',
    'classes' => array(
        'large-text'//default
    ),
)
```

04.5 Email

The email field has the following additional attributes:

```
array(
    //...
    'type' => 'email',
    'classes' => array(
        'large-text'//default
    ),
    'placeholder' => ''//placeholder attribute text
)
```

04.6 Number

The number field has the following additional attributes:

```
array(
    //...
    'type' => 'number',
    'classes' => array(
        'small-text'//default
    ),
    'step' => 1,//increment size
    'min' => 0,//min value
    'max' => 999999999//max value
)
```

04.7 Textarea

The textarea field has the following additional attributes:

```
array(
    //...
    'type' => 'textarea',
    'classes' => array(
        'large-text'//default
    ),
    'placeholder' => ''//placeholder attribute text
    'cols' => 60,//default
    'rows' => 6//default
)
```

04.8 Date

The date field has the following additional attributes:

```
array(
    //...
    'type' => 'date',
    'classes' => array(
        'regular-text'//default
    ),
    'placeholder' => ''//placeholder attribute text
)
```

04.9 Color

The colour field has the following additional attributes:

```
array(
    //...
    'type' => 'color',
    'classes' => array(),
    'placeholder' => ''//placeholder attribute text
)
```

04.10 Switch

The switch field has the following additional attributes:

```
array(
    //...
    'type' => 'switch',
    'labels' => array(
        'on' => __('On', 'domain'),
        'off' => __('Off', 'domain')
    ),
)
```

04.11 WP Editor

The editor field has the following additional attributes:

```
array(
    //...
    'type' => 'editor',
    'args' => array(
        'textarea_rows' => 6
    ),
)
```

The args array can contain any values which are sent through to the `wp_editor` function found here: http://codex.wordpress.org/Function_Reference/wp_editor

04.12 Checkbox

The checkbox field has the following additional attributes:

```
array(
    //...
    'type' => 'checkbox',
    'options' => array(//checkbox value = checkbox label
        'value' => 'label',
        'value2' => 'label2'
    ),
    'inline' => true|false//display checkboxes inline or block, default true
)
```

As you can see checkboxes allow for single or multiple checkboxes and allows you to change the display from inline or block. IF you just want the one checkbox only supply one key/vaue pair in the options array.

04.13 Radio

The radio field has the following additional attributes:

```
array(
    //...
    'type' => 'radio',
    'options' => array(//radio value = radio label
        'value' => 'label',
        'value2' => 'label2'
    ),
    'inline' => true|false//display radio buttons inline or block, default true
)
```

As you can see radio field allow for single or multiple radio buttons and allows you to change the display from inline or block. IF you just want the one radio button only supply one key/vaue pair in the options array.

04.14 Radio Image Select

The radio image select field has the following additional attributes:

```
array(
    //...
    'type' => 'radio-img',
    'options' => array(//radio value = radio label
        'value' => 'path to image',
        'value2' => 'path to image'
    ),
    'inline' => true|false//display radio buttons inline or block, default true
    'width' => '100px'//default. this is the width of the images
)
```

As you can see radio-img field allow for single or multiple radio buttons and allows you to change the display from inline or block. IF you just want the one radio button only supply one key/vaue pair in the options array.

Any time one of the images is clicked the corresponding radio input will be checked.

04.15 Select

The select field has the following additional attributes:

```
array(
    //...
    'type' => 'select',
    'options' => array(//option value = option label
        'value' => 'label',
        'value2' => 'label2'
    ),
    'multiple' => true|false,//allow multiple values to be selected - default false
    'placeholder' => ''//placeholder text for the element
)
```

The select field utilises the Selectize.js select plugin for better displaying on the UI.

04.16 Gallery

The gallery field has the following additional attributes:

```
array(
    //...
    'type' => 'gallery',
    'edit_title' => __('Add/Edit Gallery', 'domain'),//set the button text
    'remove_title' => __('Delete Gallery', 'domain')//set the button text
)
```

04.17 Media

The media field has the following additional attributes:

```
array(
    //...
    'type' => 'media',
    'upload_title' => __('Select Media', 'domain'),//set the button text
    'media_title' => __('Media Library', 'domain')//set the form text
    'media_select' => __('Select Media', 'domain')//set select element title
)
```

04.18 Info

The info field has the following additional attributes:

```
array(
    //...
    'type' => 'info',
    'icon' => ,//if set this must be a dash icon class name
    'info_type' => null|'notice'|'warning'|'success'|'error',//the type of alert to
display - default null
    'show_title' => true|false//display the title - default true
)
```

04.19 Group

The group field has the following additional attributes:

```
array(
    //...
    'type' => 'group',
    'multiple' => false,//can the group of fields be duplicated/repeated with an add
new button?
    'layout' => 'horizontal',//horizontal default, can be set to 'vertical' for a
vertical table layout (great for groups containing lots of fields)
    'show_headers' => true,//show table headers (names of fields) - default true
    'add_row_text' => __('Add Row', 'domain'),//text used for add button
    'fields' => array()//nested array of fields attached to this group
)
```

To set default values for group fields which can be multiplied, you can use the `Fluent_Base::guid();` function like so:

```
'field-id' => array(
    'type' => 'group',
    ....
    'default' => array(
        Fluent_Base::guid() => array(
            'nested-field-id' => 'some value',
            'another-nested-id' => 'somevalue'
        ),
        Fluent_Base::guid() => array(
            'nested-field-id' => 'some value',
            'another-nested-id' => 'somevalue'
        ),
    )
),
```

04.20 Font

The font field has no additional attributes

```
array(
    //...
    'type' => 'font',
)
```

04.21 Import

The import field has the following additional attributes

```
array(
    //...
    'type' => 'import',
    'upload_title' => __('Select Import File', 'domain'),
    'media_title' => __('Import', 'domain'),
    'media_select' => __('Import This File', 'domain')
)
```

04.22 Export

The export field has no additional attributes

```
array(
    //...
    'type' => 'export',
)
```

05 Usage

We have designed the framework to be consistent as a result you can be sure all of your user supplied options will remain the same across any instance type.

05.1 Introduction

The key thing to remember is every field id you define at the top level within a section will be a top level key in the returned array of values, so:

```
$sections = array(
    'section1' => array(
        //..
        'fields' => array(
            'field1' => //...
            'field2' => //...
        )
     ),
     'section2' => array(
        //..
        'fields' => array(
            'field3' => //...
            'field4' => //...
        )
     )
);
```

Will be saved and returned like so:

```
array(
    'field1' => '',
    'field2' => '',
    'field3' => '',
    'field4' => ''
);
```

Each field returned will be a string unless:

- The field is a group
- It is a multiple selection input like multi select and multi checkbox

For group fields the returned value will look like this.

For none multiples:

```
'field1' => array(
    'nested_field1' => ''
    'nested_field2' => ''
)
```

For multiples:

```
'field1' => array(
    '1-uy89-34y7-u43y-476r' => array(
        'nested_field1' => ''
        'nested_field2' => ''
    ),
    '1-u89g-3egg-uegy-ew6r' => array(
        'nested_field1' => ''
        'nested_field2' => ''
    ),
)
```

The array keys for multiples are just `uuids` used when saving the forms to ensure the correct order and nestability of the fields. As groups can be nested indefinitely the same applies as you dig deeper into the returned value.

If its a multiple selection type field it will return an array of the values selected from the options provided to the field when created.

05.2 Metaboxes

To retrieve the options from a meta box instance you need to call the `get_post_meta` wordpress function with the `post_id` and `option_name` provided to the instance args array upon creation.

```
$options = get_post_meta($post_id, 'option_name', true);
```

The `$options` var will now be an array as described on the main usage page.

05.3 Options Page

To retrieve the options from a options page instance you need to call the `get_option` wordpress function with the `option_name` provided to the instance args array upon creation.

```
$options = get_option('option_name');
```

The `$options` var will now be an array as described on the main usage page.

As a special note, options pages created for network admins with the instance arg `network = true` you would replace the `get_option` function with the multisite `get_site_option` function.

05.4 User Meta

To retrieve the options from a user meta instance you need to call the `get_user_meta` wordpress function with the `user_id` and `option_name` provided to the instance args array upon creation.

```
$options = get_user_meta($user_id, 'option_name', true);
```

The `$options` var will now be an array as described on the main usage page.

05.5 Taxonomy Meta

To retrieve the options from a taxonomy instance you need to call the `get_option` wordpress function with the `taxonomy_id` and `option_name` provided to the instance args array upon creation.

This is slightly different to most as taxonomies don’t have a meta table, we have utilised the “catch-all” options tables to store these values.

```
$options = get_option('taxonomy_' . $id . '_' . $option_name);
```

The `$options` var will now be an array as described on the main usage page.

As a side note you may ask the question about filling up the options table with taxonomies and what happens when they get deleted.

Firstly the impact on the options table saving these details isn’t worth considering as it doesn’t effect performance at all.

Secondly we have added a method to the class that will delete taxonomy data from this table when the taxonomy is deleted.

06 Custom Post Types

Using the `Fluent_Post_Type` class you can create custom post types in one line:

```
new Fluent_Post_Type('Post Type Name');
```

06.1 Configuration

And as with the rest of the classes provided by the framework there are tons of options, detailed below.

```
$name = 'Fluent Post Type';//name will be converted to lowercase with underscores
for spaces and dashes
new Fluent_Post_Type($name, array(
    //normal register_post_type() args, defaults used unless otherwise stated
    'labels'               => array(
        'name'               => sprintf(__('%ss', 'domain'), $name),
        'singular_name'      => $name,
        'menu_name'          => sprintf(__( '%ss', 'domain' ), $name),
        'name_admin_bar'     => sprintf(__( '%s', 'domain' ), $name),
        'add_new'            => __( 'Add New', 'domain' ),
        'add_new_item'       => sprintf(__( 'Add New %s', 'domain' ), $name),
        'new_item'           => sprintf(__( 'New %s', 'domain' ), $name),
        'edit_item'          => sprintf(__( 'Edit %s', 'domain' ), $name),
        'view_item'          => sprintf(__( 'View %s', 'domain' ), $name),
        'all_items'          => sprintf(__( 'All %ss', 'domain' ), $name),
        'search_items'       => sprintf(__( 'Search %ss', 'domain' ), $name),
        'parent_item_colon'  => sprintf(__( 'Parent %ss:', 'domain' ), $name),
        'not_found'          => sprintf(__( 'No %ss found.', 'domain' ),
strtolower($name)),
        'not_found_in_trash' => sprintf(__( 'No %ss found in Trash.', 'domain' ),
strtolower($name))
    ),
    'description'          => '',
    'public'               => true,
    'hierarchical'         => false,
    'exclude_from_search'  => null,
    'publicly_queryable'   => null,
    'show_ui'              => null,
    'show_in_menu'         => null,
    'show_in_nav_menus'    => null,
    'show_in_admin_bar'    => null,
    'menu_position'        => null,
    'menu_icon'            => null,
    'capability_type'      => 'post',
    'capabilities'         => array(),
    'map_meta_cap'         => null,
    'supports'             => array(),
    'register_meta_box_cb' => null,
    'taxonomies'           => array('post_tag'),//default is empty array, we have
assigned the tag taxonomy for demostration purposes. Any taxonomies added here must
be builtin or registered before the init priority 10. If you want to assign
Fluent_Taxonomies you need to declare this post type when registering the taxonomy.
    'has_archive'          => true,//default = false but we have set to true so you
can see the archive template overwrite
    'rewrite'              => true,
    'query_var'            => true,
    'can_export'           => true,
    'delete_with_user'     => null,
    '_edit_link'           => 'post.php?post=%d',
    //our custom messages array, defaults detailed below. this allows you to change
the notices when posts are changed in some way without adding yet more filters
    'messages' => array(
        //these strings are passed through sprintf with the post type name, so use
%s if you want that functionality
        0  => '', // Unused. Messages start at index 1.
        1  => __( '%s updated.', 'domain' ),
        2  => __( 'Custom field updated.', 'domain' ),
        3  => __( 'Custom field deleted.', 'domain' ),
        4  => __( '%s updated.', 'domain' ),
        5  => __( '%s restored to revision from %s', 'domain' ),
        6  => __( '%s published.', 'domain' ),
        7  => __( '%s saved.', 'domain' ),
        8  => __( '%s submitted.', 'domain' ),
        9  => __( '%s scheduled for: %s</strong>.', 'domain' ),
        10 => __( '%s draft updated.', 'domain' )
    ),
    //want to disable people adding new posts? this will remove the add new menu
item and the add new button in the and list page, it will also redirect post-
new.php?post_type=*** back to the list page.
    'disable_add_new' => false,
    //here you can set the template paths if creating the post type in a plugin, you
can also set it to override the theme version with override = true. default behavior
is to provide a fallback not to override.
    'templates' => array(
        'single' => array(//the single post page
            'override' => true,//default = false. set to true to force this to be
used before the theme version
            'path' => dirname(FLUENT_FILE) . '/example-singular-template.php'//full
path to file, default = false
        ),
        'archive' => array(//the archive/list page (if post type supports archives)
            'override' => true,//default = false. set to true to force this to be
used before the theme version
            'path' => dirname(FLUENT_FILE) . '/example-archive-template.php'//full
path to file, default = false
        )
    ),
));
```

Not only do you get all the default post type args, you can also customise the update notices, disable adding new posts, and automatically provide overrides or fallback templates (useful for post types created by plugins).

07 Custom Taxonomies

As with the custom post type class, creating a custom taxonomy can be as simple as:

```
Fluent_Taxonomy('Taxonomy Name');
```

07.1 Configuration

And as with the post type class you can do a whole lot more. all arguments are optional.

```
$name = 'Fluent Taxonomy';
new Fluent_Taxonomy($name, array(
    //normal register_taxonomy() args. all optional unless otherwise stated
    'labels'                => array(
        'name' => sprintf(__('%ss', 'domain'), $name),
        'singular_name' => $name,
        'search_items' => sprintf(__('Search %ss', 'domain'), $name),
        'popular_items' => sprintf(__('Popular %ss', 'domain'), $name),
        'all_items' => sprintf(__('All %ss', 'domain'), $name),
        'parent_item' => sprintf(__('Parent %ss', 'domain'), $name),
        'parent_item_colon' => sprintf(__('Parent %s:', 'domain'), $name),
        'edit_item' => sprintf(__('Edit %s', 'domain'), $name),
        'view_item' => sprintf(__('View %s', 'domain'), $name),
        'update_item' => sprintf(__('Update %s', 'domain'), $name),
        'add_new_item' => sprintf(__('Add New %s', 'domain'), $name),
        'new_item_name' => sprintf(__('Add New %s Name', 'domain'), $name),
        'separate_items_with_commas' => sprintf(__('Seperate %ss with commas',
'domain'), strtolower($name)),
        'add_or_remove_items' => sprintf(__('Add or remove %ss', 'domain'),
strtolower($name)),
        'choose_from_most_used' => sprintf(__('Choose from the most used %ss',
'domain'), strtolower($name)),
        'not_found' => sprintf(__('No %ss found', 'domain'), strtolower($name)),
    ),
    'description'           => '',
    'public'                => true,
    'hierarchical'          => true,//default = false
    'show_ui'               => null,
    'show_in_menu'          => null,
    'show_in_nav_menus'     => null,
    'show_tagcloud'         => null,
    'meta_box_cb'           => null,
    'capabilities'          => array(),
    'rewrite'               => true,
    'query_var'             => true,
    'update_count_callback' => '',
    //here you can assign to post types right from the taxonomy register function.
    'post_types' => array('fluent_post_type'),//this is optional, but the taxonomy
is useless without assigning it to a post type
    //here you can set the template paths if creating the taxonomy in a plugin, you
can also set it to override the theme version with override = true. default behavior
is to provide a fallback not to override.
    'templates' => array(
        'archive' => array(
            'override' => false,//default = false. set to true to force this to be
used before the theme version
            'path' => false//dirname(FLUENT_FILE) . '/example-archive-
template.php'//full path to file, default = false
        )
    ),
));
```

Just like the post type class you can edit the messages, and provide override or fallback templates for the taxonomy archive pages.

08 Fluent_Support Class

Included in the Fluent Framework is a `Fluent_Support` class which allows you to define a form which can be used anywhere within the admin.

This form allows the site user to send a support request through ajax to the configured email, and allows them to submit additional data like WordPress version, plugins, and themes. Server and PHP details.

08.1 Configuration

The Fluent_Support class has a few configuration options which you can define:

```
$support = new Fluent_Support(array(
    'id' => 'fluent-framework',//must be unique, best to be the plugin/theme name
from which you are using it. This is required.
    'email' => 'your@emailaddress.com',//where to send the messages. This is
required
    'subject' => __('Support Request', 'domain'),//the email subject sent to you,
default shown
    'data' => array(//which data sets do you want checked on load - defaults are
below
        'wordpress' => true,//version-multisite
        'plugins' => true,//active plugins and version
        'theme' => true,//active theme and version
        'php' => false,//phpinfo
        'server' => false//$_SERVER variable
    ),
    'success_message' => __('Your message has been sent successfully.',
'domain')//optionally provide a custom success message
));
//we need to store this so we can use it later on, make sure you use unique keys
here too.
Fluent_Store::set('support', $support);
```

08.2 Usage

The Fluent_Support class provides the methods:

`$object->get_form();`
`$object->the_form();`//echos the output from `$object->get_form();`

These two methods will display or return the form and enqueue the required css and javascript for the form to function.

This form can be displayed anywhere within the admin area. In the `example-usage.php` file we have included examples of adding this form to a custom field type within the options classes, and also how to add it to the help tab, or a dashboard widget.

Its important to note in the `example-usage.php` file we have used the `Fluent_Store` class to store the instance created so we can display the form at any point during the page load.

We advise creating the instance outside of an action, or at most within the `init` action, then store it and access it at any point during execution.

This method is preferred as it allows the assets to be enqueued at the correct point in time.

09 Fluent_Store Class

The `Fluent_Store` class provides a simple method for storing class instances without having to use global variables.

09.1 Usage

The `Fluent_Store` class is a very simple static class which has some very simple methods for setting and accessing objects.

This is especially useful for creating an instance of the `Fluent_Support` class and then using methods from that class during later execution.

To store an item in the Fluent_Store class you can use the method:

`Fluent_Store::set('unique_key', $objecttostore);`

It is important to use unique keys as setting a unique key twice will result in an overwrite in the store, so prefixing with your plugin name would be ideal.

To fetch an instance at any point use the method:

`$object = Fluent_Store::get('unique_key');`

This will be the full object set during earlier execution.

You can also “forget” an item from the store like so:

`Fluent_Store::forget('unique_key');`

This will remove the instance from the store.

The store class has been designed with future features in mind and it will become invaluable to the framework.

It is already used to store `Fluent_Support` instances for later usage, and you can use it to store anything you want, its not just objects, if you need to store any data type for the whole request you can store them.

Anything stored will be kept in memory for the current request, and emptied when the page finishes loading.

This isn’t a `$_SESSION` type store (although we are working on one).

10 Changelog

10.1 1.0.1

- Enabled Network Options Pages
- Fixed nested enqueue bug
- Removed Unused Files
- Updated git url in `package.json` file
- Added role lock on sections
- Added capability lock on sections
- Added `created_fields` javascript event
- Moved sortable group javascript to be run on `created_fields` event
- Moved localize variables to base options class
- Restructured javascript required functions making it easier to extend in the future
- Fixed default value for select element
- Added role lock on fields
- Added capability lock on fields
- Updated `example.php` and demo class to include new field and section options
- Fixed required fields showing errors on restore action
- Added conditional display to fields based on field value(s), multiple conditions allowed – not for group fields
- Added export field
- Added `fluent/options/save` filter to allow alteration of values before save
- Added import field
- Added ability to save fields as on there own in the database for querying
- Added javascript change event on color field selection
- Added grunt curl task for fetching google fonts data (only done through grunt and data saved in file, which is accessed by font field if used). we will update this file with every release of the framework. Developers can update this if they want by running the grunt task `grunt googlefonts`
- Added basic font selector with `family/color/line-height/font-size` properties, loaded google fonts with preview (google fonts are loaded on hover of selection to reduce page load times)
- Renamed `example.php` to `example-usage.php` for a better demo
- Removed demo class in favour of using the `example-usage.php` file to save duplication of code

10.2 1.0.2

- Added options page arg to disable restore
- Added filter `fluent/options/page/$option_name/restore/message` to options page
- Added filter `fluent/options/page/$option_name/saved/message` to options page
- Added action `fluent/options/page/$option_name/restore/messages` to options page
- Added action `fluent/options/page/$option_name/saved/messages` to options page
- Added messages array to options page args to allow custom message on save notice, restore notice, save button and save box area

- Added filter `fluent/options/page/$option_name/save/button` to options page
- Added options page args to hide the last updated text
- Added action `fluent/options/page/$option_name/save` on save
- Added `Fluent_Post_Type` class for post type creation
- Added `Fluent_Taxonomy` class for taxonomy creation
- Added image select field type

10.3 1.0.3

- Fixed typo in example usage for `Fluent_Options_Meta` `'post-types'` > `'post_types'`
- Fixed messages array notice undefined index in `Fluent_Post_Type`

10.4 1.0.4

- Network options pages now use `get_site_option` for saving values instead of primary site options table.

10.5 1.0.5

- Added underscore enqueue on color field
- Added default font size and line height to font field
- Added font preview background color switcher button
- Added `site_option_{$option_name}` filter to return default schema if not set (identical to `option_{$option_name}` but for multisite)
- Added `Fluent_Store` class for storing instances for later access (more features for this in later versions)
- Moved Fluent Taxonomy meta example to the created `Fluent_Taxonomy` instead of categories
- Added Ace field type
- Added `Fluent_Support` Class
- Added example usage of `Fluent_Support` Class to a custom field in the options panel
- Added example usage of `Fluent_Support` Class as a dashboard widget
- Added example usage of `Fluent_Support` Class as a help tab

10.6 1.0.6

- Added vertical group layout
- Added sections tabs (only displays tabs in metabox type instances (page,post), will display as normal in other instances)
- Added group styling for taxonomy pages (be carefull with this, nested groups still wont look right)
- Added group type to the example usage for txonomy and user (to demonstrate they function properly, but also to examine how they look)
- Removed some unused files to reduce framework size