Creating a Plugin

Plugins can do just about anything. Create custom cron tasks, listen for events, create pages, create widgets, extend the API, etc., etc. If it's not a Module or a Payment Gateway, it's a plugin. So let's dig in.

> **Use the Extension Generator**
>
> As of Blesta 4.12 we've included a useful tool to help developers get started and save time. Blesta's Extension Generator can be used to generate many the files necessary for a plugin and will create basic code with an option to include comments to help you understand each part of the code.

Getting Started

Plugins follow the same MVC design pattern that Blesta adheres to. The plugin system is provided as a feature of the minPHP framework, Blesta simply defines the naming convention. For the purpose of this manual, the plugin name we'll refer to is **my_plgn**. Your plugin name will, of course, differ.

> **Plugin names must be unique**
>
> A user will not be able to install two plugins with the same name, so try to use descriptive and non-generic terms for your plugin name.

File Structure

Plugins are fully contained within their named plugin directory and placed in the */plugins/* directory. Below is the minimum required file structure for a plugin:

- */plugins/*
  - *my_plgn/*
    - *controllers/*
    - *models/*
    - *views/*
    - *language/*
    - **my_plgn_controller.php**
    - **my_plgn_model.php**
    - **my_plgn_plugin.php**
    - **config.json** (for version 3.1+)

my_plgn_controller.php is the plugin's parent controller. This controller can be used to add supporting methods for the other controllers to inherit. Similarly, **my_plgn_model.php** is the plugin's parent model and can be used to add supporting methods for the other models to inherit.

Because plugins inherit from the application stack their views default to the view directory defined for application. For this reason, you should always declare the view directory for the views within your plugin. The example below requires that you place all views (.pdt files) into */plugins/my_plgn/views /default/*.

/plugins/my_plgn/my_plgn_controller.php

```
<?php
class MyPlgnController extends AppController {
    public function preAction() {
        // Set the view directory to the default core view directory so that AppController preAction code uses
the appropriate
        // directory for the structure file
        $this->structure->setDefaultView(APPDIR);
        parent::preAction();

        // Override default view directory with that of the plugin
        $this->view->view = "default";
    }
}
?>
```

Finally, **my_plgn_plugin.php** is a special class that must extend the **Plugin** class of **/components/plugins/lib/plugin.php**. This class is used for installing, upgrading, uninstalling, and branding the plugin in conjunction with the **config.json** file.

Install Logic

If your plugin requires any code to execute when installed, place that logic in your plugin's **install()** method.

/plugins/my_plgn/my_plgn_plugin.php

```
<?php
class MyPlgnPlugin extends Plugin {

    ...

    public function install($plugin_id) {
        #
        # TODO: Place installation logic here
        #
    }
}
?>
```

Need access to the database?

See how you can use the Record component to manipulate the database in the Database Access section of this manual.

Uninstall Logic

If your plugin required code to install, it likely requires code to uninstall. Place that logic in your plugin's **uninstall()** method.

There are two parameters passed to the uninstall() method. The first (**$plugin_id**) is the ID of the plugin being uninstalled. Because a plugin can be installed independently under different companies you may need to perform uninstall logic for a single plugin instance. The second parameter (**$last_instance**) identifies whether or not this is the last instance of this type of plugin in the system. If **true**, be sure to remove any remaining remnants of the plugin.

/plugins/my_plgn/my_plgn_plugin.php

```
<?php
class MyPlgnPlugin extends Plugin {

    ...

    public function uninstall($plugin_id, $last_instance) {
        #
        # TODO: Place uninstallation logic here
        #
    }
}
?>
```

Upgrade Logic

When the version of your code differs from that recorded within Blesta a user may initiate an upgrade, which will invoke your plugin's **upgrade()** method.

The **$current_version** is the version currently installed. That is, the version that will be upgraded from. The **$plugin_id** is the ID of the plugin being upgraded.

/plugins/my_plgn/my_plgn_plugin.php

```
<?php
class MyPlgnPlugin extends Plugin {

    ...

    public function upgrade($current_version, $plugin_id) {
        #
        # TODO: Place upgrade logic here
        #
    }

}
?>
```

Error Passing

Blesta facilitates error message passing through the use of the **Input** component. Simply load the Input component into your plugin and set any errors you encounter using **Input::setErrors()**. The setErrors() method accepts an array of errors. This can be a multi-dimensional array (in the case of multiple errors triggered for the same input or criteria) or a single dimensional array. The first dimension should be the name of the input that triggered the error. The second dimension, if necessary, is used to identify the type of error encountered.

1 != 0

Data validation is an important part of logic, and user friendly application design. Be sure to check out the Error Checking section of this manual for a full explanation on error handling.

/plugins/my_plgn/my_plgn_plugin.php

```
<?php
class MyPlgnPlugin extends Plugin {

    public function __construct() {
        Loader::loadComponents($this, array("Input"));
    }

    ...

    public function upgrade($current_version, $plugin_id) {
        // Ensure new version is greater than installed version
        if (version_compare($this->version, $current_version) < 0) {
            $this->Input->setErrors([
                'version' => [
                    'invalid' => "Downgrades are not allowed."
                ]
            ]);
            return;
        }
    }
}
?>
```

Branding The Plugin

For version 3.1+ simply create a config.json file and include it in the constructor of your plugin.

/plugins/my_plgn/config.json

```
{
    "version": "1.0.0",
    "name": "My Plugin Name",
    "description": "A plugin like no other!",
    "authors": [
        {
            "name": "Phillips Data, Inc.",
            "url": "http://www.blesta.com"
        }
    ]
}
```

/plugins/my_plgn/my_plgn_plugin.php

```
<?php
class MyPlgnPlugin extends Plugin {
    public function __construct() {
        $this->loadConfig(dirname(__FILE__) . DS . "config.json");
    }
}
?>
```

For version 3.0, you must define all of the branding details through methods.

All that any plugin truly requires is branding. These options come from three methods: **getName()**, **getVersion()**, **getAuthors()**.

/plugins/my_plgn/my_plgn_plugin.php

```
<?php
class MyPlgnPlugin extends Plugin {

    ...

    public function getName() {
                return "MyPlugin";
    }

        public function getVersion() {
                return "1.0.0";
        }

        public function getAuthors() {
                return [
                        [
                                'name' => "MyCompany",
                                'url' => "http://www.mycompanyplugindevelopment.com"
                        ]
                ];
        }
}
?>
```

The **getAuthors()** method requires a multi-dimensional array, so you can specify multiple authors if needed.

Lastly, each plugin needs a logo. By default these are loaded from */plugins/my_plgn/views/default/images/logo.png*. You can override the location of the logo file by implementing the **getLogo()** method in your plugin.

/plugins/my_plgn/my_plgn_plugin.php

```
<?php
class MyPlgnPlugin extends Plugin {
    ...
    public function getLogo() {
        return "views" . DS . "default" . DS . "images" . DS . "some_other_logo.png";
    }
}
?>
```

Internationalize it!

Use the Language library to help internationalize your plugin. Just create a *language* directory in your plugin, then a directory for each language (i.e. */plugins/my_plgn/language/en_us/*) and place your language files in there. See Translating Blesta for how to load, format, and use language definitions.

Managing a Plugin

If your plugin requires certain configurable options, you can create a management link viewable to staff members that have access to installed plugins. To do so, all you need to do is create an **AdminManagePlugin** controller. This is a special controller that is initialized by Blesta internally, so all views must be returned (as strings) from the requested action methods.

/plugins/my_plgn/controllers/admin_manage_plugin.php

```
<?php
class AdminManagePlugin extends AppController {

    /**
     * Performs necessary initialization
     */
    private function init()
    {
        // Set the view to render for all actions under this controller
        $this->view->setView(null, 'MyPlgn.default');
    }

    public function index() {
        $this->init();

                $var1 = "hello";
                $var2 = "world";
                return $this->partial("admin_manage_plugin", compact("var1", "var2"));
    }

        public function foo() {
        $this->init();

                return $this->partial("admin_manage_plugin_foo");
        }
}
?>
```

By default the **index()** method is called. You can link to other controller and actions from your views using the following URL format:

/plugins/my_plgn/views/default/admin_manage_plugin.pdt

```
<a href="<?php echo $this->Html->safe($this->base_uri . "settings/company/plugins/manage/" . $this->Html->_
($plugin_id, true) . "/?controller=admin_manage_plugin&action=foo");?>">Link to other action</a>
```

The above link would render the **AdminManagePlugin::foo()** method.

Making It Interactive

So far all we've see is how to build a plugin that can be installed, uninstalled, upgraded, and managed, but that's the just tip of the iceberg. As we discussed earlier, plugins can do so much more.

Actions

By register actions, a plugin makes itself available in the interface in certain areas. See Plugin Actions for how to register actions.

Events

Some plugins need to be executed when certain events occur outside of the plugin's control. This is accomplished by registering events. See Plugin Events for how to register events.

Cron Jobs

Plugins that need to perform certain tasks automatically at either set intervals are certain times of the day can register cron tasks. See Plugin Cron Tasks for how to register cron tasks.