Plugin Development: Controllers

Controllers contain the logic of the plugin and is where most of your code will go. SupportPro is built on top of the [redacted], and thus most of the features available in the framework are also available to you to use in plugins, such as caching and the query builder.

Your plugin must have a PHP class and file with the same name in the Controllers folder. The namespace must be set correctly and the class should extend the Plugin class like in our example controller below. The class should have a constructor
where parent::__construct(); and setIdentifier('PluginName') is called and must also have
declared activate, deactivate and uninstall functions that are described in more detail below.

Controllers/HelloWorld.php

```
<?phpnamespace AddonsPluginsHelloWorldControllers;use App
ModulesCoreControllersPluginsPlugin;class HelloWorld extends Plugin{
    /**     * Initialise the plugin.     */    public function __construct()    {
        parent::__construct();        $this->setIdentifier('HelloWorld');    }
    /**     * Plugins can run an installation routine when they are activated. This
 will typically include adding default     * values, initialising database tables an
d so on.     *     * @return boolean     */    public function activate()    {
        return true;    }
    /**     * Deactivating serves as temporarily disabling the plugin, but the files
 still remain. This function should     * typically clear any caches and temporar
y directories.     *     * @return bo
olean     */    public function deactivate()    {        return true;    }
```

```
    /**     * When a plugin is uninstalled, it should be completely removed as if it never was there. This function should     * delete any created database tables, and any files created outside of the plugin directory.     *     * @return boolean     */    public function uninstall()    {        return true;    }}
```

The activate function is called when the plugin is activated in the operator panel, it must return a Boolean to determine if it was activated successfully. In this function you may wish to set up any settings and/or database tables used for this plugin. Our example below calls a function we've declared later in the class that adds a custom field.

```
public function activate(){    // Create related service custom field if it doesn't exist.    $this->createField();    return true;}
```

The activate function can be called multiple times so should handle deduplication carefully.

The deactivate function is called when the plugin is deactivated in the operator panel, it must return a Boolean to determine if it was deactivated successfully. In this function you may wish to revert any data saved or changed by the plugin when it was activated.

The uninstall function is called when the plugin is uninstalled in the operator panel, it must return a Boolean to determine if it was uninstalled successfully. In this function you should remove any data that was added by the plugin. The uninstall option will also remove the plugin folder from the file system. Our example below calls a function

we've declared later in the class that deletes the custom field it originally added and removes all settings related to the plugin (described later in this guide).

```
public function uninstall(){    $this->deleteField(false);
   // Remove settings    $this->removeSettings();    return true;}
```

Additional Controllers

Additional controllers can be stored in the Controllers folder and used alongside the main controller. These classes have no constraints.

Web Controller

Below is an example web controller. See [redacted] for defining routes.

```
<?phpnamespace AddonsPluginsHelloWorldControllers;use App
ModulesCoreControllersController;class WebController extends
 Controller{    public function foo()    {        return response()->make(
'Foo');    }}
```

To return a view or template file, see [redacted].

API Controller

API controllers must return a JSON response, below is an example API controller. See [redacted] for defining API routes.

```
<?phpnamespace AddonsPluginsHelloWorldControllers;use App
ModulesCoreControllersBaseApiController;class ApiController
 extends BaseApiController{    public function foo()    {        return
 response()->json([            'status' => 'success',            'data'   => 'foo',
```

```
    ]);    }}
```

To return an error message, there are several available exceptions:

```
// General resource error.throw new AppExceptionsApi
ResourceException('message');// Store failed error.throw new App
ExceptionsApiStoreResourceFailedException('message');
// Update failed error.throw new AppExceptionsApi
UpdateResourceFailedException('message');
// Delete failed error.throw new AppExceptionsApi
DeleteResourceFailedException('message');
// Not found error.throw new SymfonyComponentHttpKernel
ExceptionHttpException(404, 'message');
// Custom HTTP error, set
a HTTP error code (int).throw new SymfonyComponent
HttpKernelExceptionHttpException($code, 'message');
```

Authenticated Operator

The logged in operator's model can be fetched with the auth_user helper function.

```
$operator = auth_user();$name = $operator->
formatted_name;$settings = $operator->opsettings;
```

Activity Logging

You may wish to add an activity log entry when the plugin completes an action automatically so it's more visible that the action has happened. Below is a simple example of this using our activity log model.

```
AppModulesCoreModelsActivityLog::addEntry([    'type'       => App
ModulesCoreModelsActivityLog::SYSTEM,    'event_text' =>
 "Sent a ticket status updated Slack notification for #{$ticket->number}."]);
```

Online URL:
https://docs.supportpro.vn/article/plugin-development-controllers-205.html