Authenticator Plugins

See also: Writing Registry Plugins

Some additional conventions are required when writing an Authenticator Plugin.

1. The name of the Plugin must match the format `FooAuthenticator`.
2. The Plugin must implement a model `FooAuthenticator`, and a corresponding Controller. (These are in addition to the other models and controllers required for Plugins.)
    a. This Model should extend `AuthenticatorBackend`, which defines some standard interfaces and provides some behind the scenes common functionality.
    b. The Model should set a public variable `$multiple` that is true is the Authenticator supports multiple values, or false otherwise.
    c. The Controller should to extend `SAuthController` ("Standard Authenticator" Controller), which provides some common functionality.
    d. When a new Authenticator Backend is instantiated, a skeletal row in the corresponding `co_foo_authenticators` table will be created. There is no `add` operation or view required. The skeletal row will point to the parent Authenticator Model.
    e. When an Authenticator Backend is edited, the entry point to the Authenticator Plugin will be `foo_authenticator /foo_authenticators/edit/#`. This will be called immediately after the Authenticator Backend is first instantiated.
    f. Authenticator Plugins that support Authenticator Reset should set `$enableSSR = true` in the model.
3. Note `Authenticator` has a `hasOne` (ie: 1 to 1) relationship with `FooAuthenticator`.
4. The table `cm_foo_authenticators` should include a foreign key to `cm_authenticators:id`.
    a. Other tables used by the plugin should reference `cm_foo_authenticators:id`.
5. The Plugin must implement several functions, which are defined in `Model/AuthenticatorBackend.php`. More details are in the next section.
6. The Plugin must implement a model for the Authenticator itself, `Model/Foo.php`. For example, for the `PasswordAuthenticator` Plugin, this Model is `Password` and is defined in `Password.php`. The Plugin must also implement a corresponding Controller, `Controller /FoosController.php`. This Controller should extend `SAMController.php` ("Standard Authenticator Model" Controller).
    a. ⚠️ The Authenticator's model should *not* `actsAs Provisioner`. SAMController will handle triggering provisioning under appropriate circumstances.
    b. Data from Authenticator models following these conventions and with a relation to CoPerson (ie: there is a `co_person_id` foreign key) will automatically be made available to Provisioner plugins.

Implementing an Authenticator Plugin

There are two supported approaches for implementing an Authenticator Plugin. Plugins that only need to provide a form, storing the results in the database, can use the Simple method. Plugins with more complicated requirements, such as redirecting the user to an external service, must instead use the Complex method.

Simple Method

1. Define the database schema for the Authenticator model (`Foo` or `Password` in the above examples; `Foo` will be used going forward) in `schema.xml` as usual, and create the Model file (`Foo.ctp`) as usual.
2. If your Plugin supports a single Authenticator per instantiation
    a. Create two symlinks in your Plugin's `View/Foos` directory:
        i. `info.ctp -> ../../../../View/Authenticators/info.ctp`
        ii. `manage.ctp -> ../../../../View/Standard/edit.ctp`
    b. Create a `fields.inc` file in the same directory that contains the form elements you need for your Plugin. (See other Authenticator plugins in the `AvailablePlugins` directory for examples.) The plugin configuration will be available in `$vv_authenticator`.
    c. Implement `Model/FooAuthenticator.php` as described above. Specifically, you must
        i. Override `current()` so that it returns the current record(s) based on the parameters passed via the function signature. The results from this function will be passed to your `fields.inc` View via the `$vv_current` variable.
            1. ℹ️ As of Registry v4.0.0, there is now a default implementation of this function implemented in `AuthenticatorBacke`nd that should work for most simple models. This function will also be used to provide data to Provisioner Plugins.
        ii. Override `manage()` so that it implements whatever backend logic your plugin requires, including data validation and the actual saving to the database. You should also create a history record indicating that the Authenticator was updated, as part of this call.
        iii. As of Registry v4.0.0, plugins may override `lock()` and `unlock()` if plugin-specific actions are required when the Authenticator is locked or unlocked. (Ordinarily, locking or unlocking disables the COmanage management interface and removes the Authenticator from provisioning data.) When doing so, the plugin must manually manage `AuthenticatorStatus` records, or else locked data may be provided to Provisio
    d. Implement the `reset()` call.
3. If your Plugin supports multiple Authenticators per instantiation
    a. Implement an `index` view. If you follow the typical add/delete/edit pattern used by other controllers, the parent SAMController will take care of much of the busy work for you, and you can simply provide a `fields.inc` with the typical contents (and symlinks to the Standard views).
    b. `reset()` is not automatically supported. You can either use the standard `delete` action to remove an Authenticator, or add your own custom action to the index view (with the usual supporting MVC requirements to implement it).
4. Implement the `status()` call. If your Plugin supports multiple Authenticators per instantiation, the result should aggregate status across all Authenticators attached to the Backend.
5. `FoosController.php` should use `$this->calculateParentPermissions()` to calculate permissions for the required actions in `isAuthorized()`.

Complex Method

In the Complex method, the plugin is expected to override the controller's `manage()` action (and `reset()` if appropriate) instead of overriding the model's functions as described above. In other words, if you override `SAMController::manage()`, you do not need to worry about overriding `AuthenticatorBackend::manage()` or `current()`. This gives your plugin the ability to perform whatever logic and process flow it needs.

As of Registry v4.0.0 it may be necessary to override `current()` to provide information to Provisioner Plugins, unless the default implementation in `AuthenticatorBackend` is sufficient.

Your controller's `manage` and (if appropriate) `reset` actions must manually call provisioning at the appropriate point in your plugin's logic. ⚠️ As of Registry v3.3.0, Authenticators may be established during enrollment. Authenticator plugins should *not* trigger provisioning during enrollment.

As of Registry v4.0.0, the plugin must also trigger notification when the Authenticator is updated. The easiest way to do this is with `AuthenticatorBackend::notify()`, which will handle all the necessary steps.

When you are finished, your controller should return control to Registry by calling `$this->performRedirect()`.

```
...
// Finished updating our Authenticator's state in the database
if(!isset($this->request->params['named']['copetitionid'])) {
  // Don't provision or notify if we're in an enrollment
  $this->Authenticator->provision($coPersonId);
  $this->Authenticator->notify($coPersonId);
}

// All done
$this->performRedirect();
```

Use `$this->calculateParentPermissions()` to calculate permissions for the required actions in `isAuthorized()`.

Implementing REST APIs

As of Registry v3.3.0, Authenticator plugins may expose APIs.