Writing Registry Plugins

Registry functionality can be extended with the use of Plugins.

Building a Registry Plugin requires knowledge of PHP, CakePHP, and COmanage.

If you just need to publish static documents, see Publishing Mostly Static Public Content instead.

1. Building a Registry Plugin

1.1. Background

1. Understand the Cake Framework. You should minimally have worked through the tutorials and examples.
2. Understand Cake Plugins. Registry Plugins are just Cake Plugins, with some extra conventions.
   a. ⚠ In general, when implementing a Cake callback in a plugin, call the parent as well. eg, in a plugin's `beforeValidate()`, call `parent::beforeValidate($options)`.
3. Understand the Registry Data Model.

1.2. Instantiated vs Non-Instantiated

There are two basic categories of Plugins supported by COmanage Registry.

- *Instantiated* plugins must be configured before they can be used. For example, the LDAP Provisioner must be given connection information that is specific to each CO. Creating that configuration is *instantiating* the plugin.
- *Non-Instantiated* plugins do not have CO-specific configuration. For example, normalizing whitespace is not CO specific.

This categorization is based on how Registry handles the plugin. An instantiated plugin follows a typical add/edit flow in order to use the plugin, with entry points described in the *Additional Requirements By Plugin Type* section, below.

It is possible for a non-instantiated plugin to maintain its own configuration, however Registry provides no direct support for configuring non-instantiated plugins.

⚠ It is possible that non-instantiated plugins will be converted to instantiated plugins in future releases as Registry capabilities evolve. Such a conversion may require changes to the plugin interfaces.

1.3. Plugin Directory

2. Additional Requirements By Plugin Type

Plugin Types not listed here have no additional requirements.

- Authenticator Plugins
- Cluster Plugins
- Dashboard Widget Plugins
- Data Filter Plugins
- Enrollment Flow Plugins
- Identifier Assignment Plugins
- Identifier Validation Plugins
- Invitation Confirmer Plugins
- Job Plugins
- LDAP Schema Plugins
- Normalization Plugins
- Organizational Identity Source Plugins
- Provisioner Plugins
- Vetter Plugins

Set up a new Plugin in the directory `local/Plugin/MyPlugin`. You might find it handy to use Cake's `bake` command.

```
$ cd app
$ ./Console/cake bake plugin MyPlugin
Welcome to CakePHP v2.7.1 Console
---------------------------------------------------------------
App : app
Path: /home/user/src/comanage/git/registry/app/
---------------------------------------------------------------
1. /home/user/src/comanage/git/registry/local/Plugin/
2. /home/user/src/comanage/git/registry/app/Plugin/
3. /home/user/src/comanage/git/registry/plugins/
Choose a plugin path from the paths above.
[1] > 1
```

⚠ Prior to v1.0.0, Plugins must be placed in `app/Plugin/MyPlugin`.

1.4. Plugin Model

1. Create a Model whose name matches the name of the Plugin. In this example, the Model is created at `app/Plugin/MyPlugin/Model/MyPlugin.php`.
2. Define `$cmPluginType` to indicate the type of the Plugin, from the following options:

$cmPluginType	Description	Available Since	Instantiated?
`authenticator`	Authenticator Plugin	v3.1.0	Yes
`cluster`	Cluster Plugin	v3.3.0	Yes
`confirmer`	Invitation Confirmer Plugin	v3.1.0	No
`dashboardwidget`	Dashboard Widget Plugin	v3.2.0	Yes
`datafilter`	Data Filter Plugin	v3.3.0	Yes
`enroller`	Enrollment Flow Plugin	v0.9.4	Yes, as of v4.0.0
`identifierassigner`	Identifier Assignment Plugin	v4.1.0	No
`identifiervalidator`	Identifier Validation Plugin	v2.0.0	Optional
`job`	Job Plugin	v3.3.0	Yes, when queued
`ldapschema`	LDAP Schema Plugin	v2.0.0	Yes, via LDAP schema configuration
`normalizer`	Normalization Plugin	v0.9.2	No
`orgidsource`	Organizational Identity Sources Plugin	v2.0.0	Yes
`provisioner`	Provisioning Plugin	v0.8	Yes
`vetter`	Vetting Plugin	v4.1.0	Yes
`other`	Any other type of Plugin	v0.8	No

As of v2.0.0, `$cmPluginType` may also be an array. A Plugin that implements more than one plugin type is referred to as a *Polymorphic Plugin*. Note that, due to naming conventions and other constraints, not all combinations of plugin types may currently be supported. These constraints will be removed over time as Plugin interfaces are updated.

Here's an example Model:

```
class LdapProvisioner extends AppModel {
  // Required by COmanage Plugins
  public $cmPluginType = "provisioner";
}
```

1.5. Database Schema

Plugins have full access to the Registry database. You can create your own additional tables by creating a schema file and placing it in `Plugin/MyPlugin/Config/Schema/schema.xml`. The file is specified in ADOdb AXMLS format.

Tables should follow the Cake standard conventions, including `id`, `created`, and `modified`.

Once the file is created, the database schema will automatically be updated by the normal mechanism:

```
$ cd app
$ ./Console/cake database
```

Database Prefixing

For the most part, the database prefix as specified in the database configuration file will just work. The exception is that foreign keys must have the prefix explicitly hardcoded (CO-174). For now, using the prefix `cm_` is recommended.

1.6. Foreign Key Dependencies

If you define tables for your plugin, you will almost certainly want foreign keys into the core database schema. For example, your tables may have `co_person_id` or `co_id` to refer to CO People or COs, respectively.

In order for deletes to cascade successfully when the parent object is deleted, you must specify any dependencies your plugin has. (Failure to do so will result in foreign key violation errors when the parent object is deleted.) To do so, define an array `$cmPluginHasMany` in `Model/MyPlugin.php`, which consists of an array where the keys are the model the foreign key points to and the values are the name of the model they point from.

For example:

```
// Document foreign keys
public $cmPluginHasMany = array(
  "CoPerson" => array("CoChangelogProvisionerExport")
);
```

`CoPerson` *hasMany* `CoChangelogProvisionerExports`. Put another way, `co_changelog_provisioner_exports` has a column `co_person_id` that is a foreign key to `co_people`.

As of Registry v3.2.0, `$cmPluginHasMany` also accepts a standard Cake model dependency array. eg:

```
// Document foreign keys
public $cmPluginHasMany = array(
  "CoPerson" => array(
    "CoAnnouncementPosterCoPerson" => array(
      'className'  => 'CoAnnouncement',
      'foreignKey' => 'poster_co_person_id'
    )
  )
);
```

Unfreezing Foreign Keys

As of Registry v3.3.3, foreign keys are by default frozen when a record is initially saved. For example, if your record has a foreign key to `co_person_id`, once the record is created it can't be reassigned to a new CO Person.

For more information, see Unfreezing Foreign Keys.

1.7. Language Texts

Do not hardcode display texts, but instead create lookup files that translate keys into language specific texts. For now, Registry uses a custom mechanism for I18N/L10N (CO-351). Use `_txt(key)` or `_txt(key, array(param1, param2))` in your code to generate language-specific text, and then define those keys in the file `Plugin/MyPlugin/Lib/lang.php`:

```
// When localizing, the number in format specifications (eg: %1$s)
indicates the argument
// position as passed to _txt.  This can be used to process the arguments
in
// a different order than they were passed.

$cm_my_plugin_texts['en_US'] = array(
  // Titles, per-controller
  'ct.co_my_plugin_model.1'  => 'My Plugin Model',
  'ct.co_my_plugin_model.pl' => 'My Plugin Models',

  // Plugin texts
  'pl.myplugin.someparam'    => 'Some Parameter',
  'pl.myplugin.another'      => 'Another Parameter'
);
```

1.8. Enumerations

Plugins may define enumerations in `Plugin/MyPlugin/Lib/enum.php`:

```
class MyPluginFruitEnum
{
  const Apple = 'A';
  const Orange = 'O';
}
```

1.9. Use of Standard Views

You may use Registry's "standard" views to easily render your pages in the Registry look. Simply create links from `Plugin/MyPlugin/View/Model` to `../../../../View/Standard`. See core Registry views for examples. Note that you will need to define the language texts `ct.co_X_model.1` and `ct.co_X_model.pl` to use the standard views, replacing `X` with the name of your model.

1.10. Server Definitions

Registry v3.2.0 introduces Server objects, which are intended to reduce duplicate configuration, and allow multiple plugins (or other components) to share the same server information and state. Plugins should use Server objects for configuration whenever possible. To do so, the CO Administrator is expected to define the Server outside of the Plugin. Then, during configuration, the Plugin should present a list of available Servers of the desired type and store the foreign key reference to that configuration.

To obtain the list of available Server, any model associated with a controller that extends StandardController can set

`public $cmServerType = ServerEnum::`*`ServerType`*

The corresponding view will be passed `$vv_servers`, with a list of available servers keyed by `server_id`.

1.11. Menu Links

Plugins can add items to Registry's menus. To do so, define in `Model/MyPlugin.php` a function `cmPluginMenus()` that returns an array. The key in this array is the menu location to append to (see table below), and the value is another array, which defines one or more labels and the corresponding controllers and actions to generate links to. The Plugin infrastructure will automatically append CO IDs and CO Person IDs as appropriate.

Whether or not a Plugin menu is rendered is determined by the default permission as listed below.

Menu Location Key	Menu Location*	Default Permission	Icon?	CO ID Inserted?	CO Person ID Inserted?	Available Since
canvas	CO Person Canvas sidebar	CO Person	✓		✓	v4.1.0
`cmp`	Platform Menu	CMP Administrator				v0.8
`cos`	Collaborations Menu ⚠ Removed v3.0.0	Member of Any CO				v0.8 - v3.0.0
`coconfig`	CO Configuration Menu	CO Administrator	✓	✓		v0.8
`comain`	CO Main Menu	Member of CO	✓	✓		v3.2.0
`copeople`	CO People Menu	Member of CO		✓		v0.8
`cogroups`	CO Groups Menu	Member of CO		✓		v1.0.0
`coperson`	My Identities Menu	Member of CO			✓	v0.8
`coservices`	CO Services Menu ⚠ Removed v3.0.0	Member of CO		✓		v2.0.0 - v3.0.0

1.11.1. Menu Link Icons

As of Registry v3.2.0, icons are required in two contexts: `coconfig` and `comain`. The icon is specified by a special `icon` key in the menu URL array, and references the name of a Material Design icon or jQuery icon, depending on context.

1.11.2. Example

```
/**
  * Expose menu items.
  *
  * @since COmanage Registry v0.9.2
  * @return Array with menu location type as key and array of labels,
controllers, actions as values.
  */

public function cmPluginMenus() {
  return array(
    "coperson" => array(_txt('pl.dirviewer.viewmenu') =>
      array('controller' => "dir_viewers",
            'action'     => "view")),
    "coconfig" => array(_txt('pl.dirviewer.viewcfg') =>
      array('icon'       => "visibility",
            'controller' => "dir_viewers",
            'action'     => "index"))
  );
}
```

1.12. Search

As of Registry v3.2.0, Plugins may implement searching as part of the main search box.

First, declare which plugin models support search in `Model/MyPlugin.php` by defining a function `cmPluginSearchModels()` that returns an array, keyed on plugin model, of displayField and permissions:

```
/**
 * Declare searchable models.
 *
 * @since  COmanage Registry v3.2.0
 * @return Array Array of searchable models
 */

public function cmPluginSearchModels() {
  return array(
    'MyPlugin.Dir' => array(
      // The model field to display in the results
      'displayField' => 'name',
      // Which types of users may search this model
      'permissions' => array('cmadmin', 'coadmin', 'couadmin', 'comember')
    )
  );
}
```

Next, in each supported model add a function `search($coId, $q)` that implements the backend search, based on the provided CO ID and unparsed search string. (Be sure to add databases indexes as needed.) Return an array of search results in the same format as `find()`.

```
/**
 * Perform a keyword search.
 *
 * @since  COmanage Registry v3.2.0
 * @param  integer $coId  CO ID to constrain search to
 * @param  string  $q     String to search for
 * @param  integer $limit Search limit, added in Registry v4.0.0
 * @return Array Array of search results, as from find('all)
 */

public function search($coId, $q, $limit) {
  // Tokenize $q on spaces
  $tokens = explode(" ", $q);

  $args = array();

  foreach($tokens as $t) {
    $args['conditions']['AND'][] = array(
      'OR' => array(
        'LOWER(Dir.name) LIKE' => '%' . strtolower($t) . '%',
      )
    );
  }

  $args['conditions']['Dir.co_id'] = $coId;
  $args['order'] = array('Dir.name');
  $args['limit'] = $limit;
  $args['contain'] = false;

  return $this->find('all', $args);
}
```

Models that may retrieve large numbers of records (such as Name) or that perform substring searches (such as Address) should use Linkable Behavior (which generates a single query using JOIN) instead of Containable Behavior (which generates at least one additional query for each result).

1.13. Duplicating COs

This feature is available since Registry v3.2.0.

When a CO is duplicated, Registry will attempt to copy configuration data, but not operational data (such as CO Person records).

For Instantiated plugins, duplication will attempt to copy plugin configuration. By default, this will involve copying data in the core model for the plugin, eg: `CoFooProvisionerTarget` for *provisioner* plugins or `FooSource` for *orgidsource* plugins. However, it is possible to override this default behavior in order to duplicate additional configuration records associated with the plugin. To do so, in the plugin's core model define $duplicatableModels, which is an array of relevant models and their configuration (parent model and foreign keys). For example:

```
// CoAnnouncementsWidget.php
public $duplicatableModels = array(
  "CoAnnouncementChannel" => array(
    "parent" => "Co",
    "fk"     => "co_id"
  ),
  // If you set $duplicatableModels, then you must also list the core
model as well
  "CoAnnouncementsWidget" => array(
    "parent" => "CoDashboardWidget",
    "fk"     => "co_dashboard_widget_id"
  )
);
```

Duplication does not currently support Non-Instantiated plugins.

1.14. Related Actions Links

Adding links to Related Actions is not currently supported. (CO-520)

1.15. REST API

Exposing Plugin functionality via the REST API is not currently officially supported (CO-521), however as of Registry v3.2.0 it is possible for plugins to expose a REST API. This feature is considered *experimental*, as future releases may impose structure to standardize or simplify the process.

To expose an API, the plugin will most likely need to defined routing in `$PLUGIN/Config/routes.php`. The plugin may use the standard Cake mapResources() call, and may leverage Registry's ApiComponent. For more details, see example plugins or core code. More detailed documentation will be available when REST APIs for plugins are fully supported.

Note that plugin API paths will be prefixed with the plugin name, as with web pages. ie: `/registry/some_plugin/controller/action/id`.

Controllers in `mapResources()` must include the plugin.

```
Router::mapResources(array(
  'PluginModel.plugin_model_widgets'
));
```

1.15.1. Filtering Index Views

Most REST API "View" calls can be filtered somehow, such as view all items by CO or CO Person. Plugins may wish to filter by a Plugin-specific data element. As of Registry v3.3.0, this is possible by declaring `$permittedApiFilters` in the appropriate Model file. Note this is the Model associated with the desired REST API (eg: `Model/MyOtherModel.php`), which is not necessarily the primary Plugin Model (`Model/MyPlugin.php`).

The content of `$permittedApiFilters` is an array, where each key corresponds to a column (which must be a foreign key) in the Model and the value is the related Model that will implement the desired filtering. By way of example, consider a plugin that implements two models: Boxes and Labels, where Box hasMany Label. To enable the REST API to retrieve all Labels associated with a given Box, add the following to `Model/Label.php`:

Model/Label.php

```
public $permittedApiFilters = array(
  'box_id' => 'MyPlugin.Box'
);
```

This will enable the REST call `GET /registry/my_plugin/labels.json?box_id=X`.