Registry PE Coding Guide

Background

This document is intended to provide guidance to anyone writing code against COmanage Registry PE (v5.0.0 or later), including plugin developers. It describes various facilities developed for COmanage specifically, to help make it easier to write DRY code.

Prerequisites

Before continuing, be sure to be familiar with the following:

1. PHP, and in particular newer PHP features such as
   a. Traits
   b. Named Arguments
2. CakePHP v4 and the MVC design pattern
3. The Registry Data Model (and the Technical Manual in general)

PHP 8

Registry PE is targeting PHP 8+. In particular:

1. Function calls should be typed, including the return type.

PSR Coding Style

Registry PE generally adopts PSR-12, with some variations. Where discrepancies exist between this section and the rest of this document, this section controls and the conflict should be assumed to be a legacy artifact that has not been updated.

In general, CakePHP Coding Conventions should also be followed, except where they conflict with guidance in this document.

Variations

1. **Indent 2 spaces rather than 4.** This is a change to PSR-12 §2.4. Indenting 4 spaces makes code sprawl horizontally, making it harder to read and more difficult to keep to 80 characters per line (PSR-12 §2.3).
2. **Empty lines should have white space aligned to the start of the preceding line.** This makes it easier to visually align blocks of code without have the cursor jump all over the place. This is technically a change to PSR-12 §2.3.
3. **Ignore CakePHP guidance on Line Length.** PSR-12 §2.3 controls (line length target of 80 characters, soft limit of 120 characters, no hard limit).
4. **Ignore CakePHP guidance on nested ternaries.** Nested Ternary Operations are permitted with concise, and can be written more clearly than the equivalent if/then/if/then/else clause.

Differences From The COmanage Coding Style

For developers already familiar with the v4 and earlier coding style, this section highlights the changes:

1. **Method names MUST be declared in camelCase** (PSR-1 §4.3). While this was already generally the case, there are some places that used under_score style names.
2. **Use short form of type keywords** (PSR-12 §2.5). There was not previously a standard, so (eg) `int` and `integer` were used interchangeable.
3. **Opening braces for function definitions start on a new line** (PSR-12 §4.4).
4. **Multi-line function arguments are each placed on their own line** (PSR-12 §4.5).
5. **Spacing for if/then/else** (PSR-12 §5.1), **while** (§5.3), **for** (§5.4), **foreach** (§5.5), **and try/catch** (§5.6). A space is now always inserted after a keyword, and alignment for expressions split across multiple lines has changed.

Automating PSR-12 Compliance

XXX PHP_CodeSniffer

Adding New Models

Data Model and Functional Design

Before writing any code, start by proposing changes to the existing data model (which may include defining new tables that will map to the new models added to the application). Build a proposed functional design around the data model changes. For code that will be contributed to the project, review these designs with the development group before beginning any meaningful coding, since code that does not align with the project's direction will not be accepted.

schema.json

Schema management is handled by Doctrine DBAL, using a Registry specific JSON schema file processed by `DatabaseCommand`. The format of the schema file is fairly self documenting, but note the following:

- The `columnLibrary` provides default definitions for commonly used attributes. These defaults will be used when the table defines a column name with the same name as the library definition. All library values are inherited by default, it is then only necessary to explicitly define the ones that should be changed.
- Index names must be explicitly specified (as opposed to autogenerated) so that if the index definition changes DBAL can recompute the index. (This is much more efficient than the previous ADOdb based system, which rebuilt all indexes on the table any time the table was changed in any way.)
    - In general, do not define unique constraints on indexes, as they will conflict with Changelog Behavior. Uniqueness can be enforced using Application
      Rules.
- Cake's Timestamp fields will be automatically inserted, unless `timestamps` is set to false in the table definition.
- Fields for Changelog Behavior will be automatically inserted, unless `changelog` is set to false in the table definition.
- For attributes that can be created via Registry Pipelines, setting `sourced` to true will insert the necessary foreign key and index.
- For Multi-Valued Entity Attributes (MVEAs), setting `mvea` to a list of parent tables will insert the necessary foreign keys and indexes.

Note that since JSON files inexplicably can't have comments, the key `comment` is reserved in all contexts except the list of column definitions to be used for comments.

Application Rules

"Implicit logic" must be documented in the form of Application Rules.

Application Rules are typically enforced using Cake's Application Rules, which are applied to a table using the table's `buildRules()` function. By convention, COmanage rules are named `ruleSomethingOrOther()`, and are defined in the table they apply to. Rules common to multiple tables should be implemented in `RulesTrait`. Global rules that apply to all tables are implemented in the `RuleBuilderEventListener`.

Application Rules must be labeled in a comment adjacent to the code that enforces them using the form `AR-Model-#`, that is the string "AR-", the camel cased singular model name, a dash, and the number of the rule for that model (for example: `AR-ApiUser-3`). Global rules are referred to as *General Model Rules*, and are labeled `AR-GMR-#`.

Wherever possible, log entries (to the `rules` level) should be generated when an Application Rule is applied.

In general, Application Rules are not configurable.

Transmogrification

For migration of Registry tables from v4 to v5, appropriate support for migrating existing data must be added to `TransmogrificationCommand`.

Each table must be added to `$tables` in the same order as `schema.json` (ie: to correctly sequence the population of primary keys). By default, fields are mapped 1-1 unless configured via `fieldMap`. (The `displayField` is used when Transmogrification is running.)

In some cases, a custom mapping function is required to calculate the target table value. In some of these cases, results from an earlier table should be cached so later mapping actions can quickly find earlier values. This is accomplished with the `cache` entry.

If a table was not previously Changelog enabled but is Changelog enabled in v5, the key `addChangelog` must be set to `true`.

Boolean fields must currently be explicitly identified in the `booleans` entry.

Logging

XXX

Testing

XXX

Documentation

- Application Rules must be documented as described above.
- Functional documentation should be added to the Technical Manual.
- The Data Model must be updated.
- The REST API documentation must be updated, if needed.

Leveraging DRY Patterns

Registry builds various utilities on top of the Cake framework.

API Controllers

Unlike in v4 and earlier, API transactions are handled entirely by dedicated controllers. The standard Registry model level API is implemented by `ApiV2Controller`, other APIs are implemented in plugins. As a result, controller specific logic (such as overriding `beforeFilter`) *will not automatically apply* to APIs. Model specific logic that needs to apply to both the UI and API should be defined in the model (table) through the use of Traits or other similar techniques, and then referenced generically from the calling controllers.

Authorization (RegistryAuth Component)

XXX

Behaviors

Changelog Behavior

In general, the previous documentation on Changelog Behavior applies, though not all features are implemented yet.

1. Because ChangelogBehavior intercepts `delete` requests and converts them to updates, `beforeDelete` and `afterDelete` callbacks should not be used.
2. Tables using ChangelogBehavior should use `ChangelogBehaviorTrait`.

Log Behavior

XXX

Callbacks

1. Because ChangelogBehavior intercepts `delete` requests and converts them to updates, `beforeDelete` and `afterDelete` callbacks should not be used.
2. Tables using ChangelogBehavior should use `ChangelogBehaviorTrait`, which implements `afterSave`. As such, Tables should not implement their own `afterSave`, but should implement `localAfterSave` (with the same function signature as `afterSave`). `localAfterSave` will not be called when archived records are written to the database.

Enumerations

XXX

Function Parameters and Signatures

When defining functions, use parameter type, return types, and default values. When the ID of the current Model is a parameter, it should be first in the list.

```
public myFunction(int $id, string $label, bool $colorful=false): string { ... }
```

When calling functions, use parameter names wherever possible.

```
$s = $this->myFunction(id: $entity->id, label: __d('information', 'my.label'), colorful: true);
```

Avoid the use of configuration arrays (though Cake still makes heavy use of these).

```
public badExample($id, $options=[]);
```

Localizations

When localizing text strings, use the table name and/or field name as is whenever possible.

Non-Standard Foreign Key Relations

Normally, a relation can simply be defined using something like

```
$this->belongsTo('Types')
```

which implies the current table has a columns `type_id`. However, sometimes it is necessary or desirable to use a different foreign key name, such as `default_type_id`. This can be accomplished with something like

```
$this->belongsTo('Types')->setForeignKey('default_type_id');
```

Registry's foreign key checks further require a property to be set so that `ruleValidateCO` can properly validate foreign keys at run time. This can be accomplished by setting a property with the name of the foreign key without the `_id`:

```
$this->belongsTo('Type')->setForeignKey('default_type_id')->setProperty('default_type');
```

Ordering

Many models can be ordered, eg Provisioning Targets. In order to leverage common utility code:

1. The relevant table definition should include the field `ordr`, spelled with out the "e". (This is because `order` is a reserved keyword in MySQL.)
2. The Table should use *OrderableBehavior*, which will automatically set an appropriate value for `ordr` if none is provided when a new entity is saved.

a. OrderableBehavior will constrain searches by primary key when assigning the next `ordr` value. So, for example, when a new Provisioning Target is added, `max(ordr)` is determined for all Provisioning Targets within the same CO, while for Enrollment Flow Steps `max(ordr)` is determined for all Enrollment Flow Steps within the same Enrollment Flow.

Timezones

All timestamps are stored in the database in UTC (AR-GMR-4).

To automatically convert to UTC on save, the table should load `TimezoneBehavior`. `FieldHelper::control()` will convert from UTC on rendering, as will the Standard `index.php` when `columns.inc` sets the field type to `datetime`.

There is no timezone conversion for the REST API.

See also: Registry Timezones

Traits

Common code used to be placed in `AppModel`, which led to a large and complicated pile of code. In general, common functionality is now implemented using traits.

AutoViewVars Trait

XXX

History Trait

Models should record History at appropriate points to facilitate administrator review of actions affecting a Person record. `HistoryTrait` offers utility functions to simplify recording history. It is also possible to use `HistoryRecordsTable` directly, though it may be more complicated to do so.

PrimaryLink Trait

The *Primary Link* of a table is the foreign key to the most significant parent object, typically `co_id` or `person_id`. The Primary Link is used to automatically determine permissions, generate links, and other similar purposes.

Redirect Goals

After adding or editing an entity, different models may have different user experiences for where to go next. The selection of a redirect target can be controlled by setting a *Redirect Goal*. Currently supported Redirect Goals are

- `index`: Redirect to the index for the model, filtered by the Primary Link
- `primaryLink`: Redirect to the Primary Link entity
- `self`: Re-render the same form

Declaring Primary Links to Plugins

Primary Links can be declared to Plugin models (for example, the a Plugin defines secondary models to a Primary Plugin model) using the notation `Plugin.foreign_key_id`. Note that this is the *physical plugin name* and not the Entry Point Model. The foreign key ID will be inflected to get the Model name. For example, `$this->setPrimaryLink('CoreAssigner.format_assigner_id');` will declare the primary link to be to `CoreAssigner.FormatAssigners::id`.

Models / Tables

Accessing Models / Tables

Cake supports two main ways for referencing another model (table) from within a model (table) or controller. The first is via the model relation:

Model Relations

```
// eg, in GroupMembersTable.php:
$person = $this->People->get($entity->person_id);
```

The second is via the `TableLocator` class. The TableLocator is directly available in a controller, or can be accessed using the `LocatorAwareTrait` it a Model.

TableLocator

```
// In a controller
$People = $this->getTableLocator()->get('People');

// In a model
class MyTable extends Table {
  use \Cake\ORM\Locator\LocatorAwareTrait;

  public function doSomething() {
    $People = $this->getTableLocator()->get("People");
  }
}
```

In general, either approach is acceptable. The first approach is usually simpler and more compact, however when a long chain of relations is required to get to the desired table, the second approach may be preferable. When there is no directly relation, the second approach is required.

Determining the Current CO

In general, model code should obtain the current CO via parameters passed to the functions it implements, either directly (when there is no other parameter that implies a CO) or indirectly (when another parameter, such as `$personId`, can be used to calculate the CO). The function `findCoForRecord` (implemented in `PrimaryLinkTrait`) can be helpful.

In rare cases, it may be necessary to determine the CO by other means, for example in order to adjust validation rules based on the current CO. Models can `setAcceptsCoId` on their table (again via `PrimaryLinkTrait`), and `AppController` will then provide the CO to the table as part of `setCO`. Note this currently works only for the primary table of the request and its immediate relations.

Joins

In general, avoid using joins unless required for performance reasons. Joins make the code harder to read, require special annotations, may interact poorly with ChangelogBehavior, and require special handling when the database configuration has `quoteIdentifiers` enabled (which is required for MySQL).

Validation and Application Rules

XXX

Views

columns.inc

Appending a Custom String

It is possible to append a custom string (such as "Primary Link") to a field in the index view on a per-record basis using the `append` directive. The value is a function implemented on the entity. For example,

templates/Names/columns.inc

```
$indexColumns = [
  'type_id' => [
    'type' => 'fk',
    'append' => 'primaryLabel'
  ]
]
```

when rendering the field *type* for the index of *names*, `$nameprimaryLink(): string` will be called and the returned string will be appended with a comma to the string for the current value of the type foreign key. The result will be something like "`Official, Primary Name`".

fields.inc and FieldHelper

XXX

Custom Display Fields

By default, the display field used for a model is whatever is set using Cake's `setDisplayField`. However, tables can implement model-specific display logic by implementing `generateDisplayField($entity): string`.