COmanage Coding Style

> ⚠ **Work In Progress**
>
> This Coding Guide is a work in progress. Furthermore, existing code may not meet these guidelines. However, all new code must, including any refactoring of existing code.

Background

A common coding style ensures consistency of the look and feel of the code, regardless of who wrote a given portion. This, in turn, increases readability, makes the code easier to understand, and makes the project look more polished and professional.

Many of the style decisions here are arbitrary. The purpose of this document is not to answer the general question "What is the proper number of spaces to indent code?" but the specific question "What number of spaces is code indented in the COmanage codebase?" (The answer is 2, described below.)

For examples of other coding style guides, see The Laminas Project (PHP), PHP-FIG's PSR-12 (PHP), Apache (C), and Google (various).

External Packages

Information regarding the use of external packages can be found in Managing External Packages.

General

- Use the most compact notation that is still easy to understand.
- Follow CakePHP guidelines, such as for file names.
- If there's a reason to have an exception to these guidelines in the code, then there's a reason to update this document.

Git Commit and JIRA Logs

- Git commit logs must include the JIRA issue they are addressing. In addition, a useful description must be included, so that `git log` will show useful messages. By convention, commit messages should list the description first with JIRA issue numbers appended in parentheses. If no ticket is relevant to a commit, use "NOJIRA". For example:
  - *Fix normalization of certain attributes during enrollment (CO-155)*
  - *Fix for PHP 7.2 compatibility (NOJIRA)*

- JIRA comments must include a brief explanation of what was fixed, to facilitate historical searches. (Simple fixes, such as typos, need no explanation.) JIRA issues must be updated with corresponding SVN commit revision numbers.
- When submitting a Pull Request, add a link the the PR in the JIRA issue (in the comments) to make it easier to track the status of the issue.

Error Reporting

There are a number of mechanisms for reporting errors. Which to use depends on context.

setFlash

For most interactive transactions, use `$this->Session->setFlash` to set a message that will be rendered with the next view. This includes standard pages ("default" layout) as well as page sections retrieved via AJAX calls ("ajax" layout).

REST HTTP Response Code

For API transactions (ie: XML or JSON over HTTP), a suitable message and HTTP Response Code should be generated, as per the API definition.

Exceptions

Exceptions should be used rarely, and only in situations where the application cannot recover gracefully enough to generate an error using a preferred method.

CakeError

CakeErrors are deprecated as of 2.0 and should not be used.

Logging

Starting with Registry v5 (and for all versions of Match), detailed logging should be added when it would be useful to help trace business logic execution, especially for documented Application Rules. Registry v5 defines `LabeledLogTrait` that can be used to automatically prefix metadata to the log entry. LabeledLogTrait builds on top of Cake's Logging services.

```
class Bar {
  use \App\Lib\Traits\LabeledLogTrait;

  public function foo() {
    $this->llog('debug', "Something happened");
  }
}
```

will place an entry like this into the configured Debug log:

```
2020-02-01 17:33:10 Debug Bar::foo Something happened
```

Debugging

Debugging should be done with `debug()` rather than `print`. Debugging should not be committed, and this will make it easier to find stray debug statements before committing. Furthermore, `debug()` statements will become no-ops in a production setup if they do accidentally end up in the codebase.

For versions of Registry before v5, the use of `$this->log()` is also acceptable.

Making Breaking Changes

Breaking changes cause version number changes as defined by Semantic Versioning.

As of v1.0, for changes that require updates to existing data, such changes must be scripted and run as part of the upgrade shell.

Naming Correct Case

Methods

Camel case with lower initial:

```
$this->bindModel();
$this->initializeParentCou();
```

Methods should be in alphabetical order within the file that contains them. eg: `add` should be before `delete` in `FooController.php`.

Variables

Camel case with lower initial:

```
$cou = 'whatever';
$couAllowed = array();
$isInCou = false;
```

Avoid very short names except in very compact contexts, such as `for` loops.

View Variables

View Variables are those set in a Controller and passed to a View. View Variables must be prefixed `vv_`.

Controller:

```
$this->set('vv_my_variable', $this->Model->find('first'));
```

View:

```
print $vv_my_variable
```

Quotes

Single Quotes

Single quotes are used when using a string as an index.

```
$foo['this']['that'];
```

Double Quotes

Double quotes are used when quoting a string in other contexts.

```
$txt = "This is some text.";
```

Whitespace

Indentation

Indentation is in increments of two (2) spaces. Tabs are not used.

Note this differs from the PSR-2/PSR-12 standard of 4 spaces. The earliest COmanage code pre-dates PSR-2, and there are various arguments (none particularly important) as to why 2 spaces are better than 4. More importantly, it'd be a lot of noise and churn to change at this point.

```
function foo($i) {
  $j = $i + 1;

  if($j > 1)
    bar($j);
}
```

Separation of Characters

In general, whitespace is omitted where it is not necessary. For example:

```
function foo($a);

$a = $b['foo'];
```

And not:

```
function foo ( $a );

$a = $b[ 'foo' ];
```

The exceptions are where extra whitespace dramatically increases readability:

```
// Argument lists, arrays, etc
passingArguments($first, $second, $third);

// Nested array references
$a = $b[ $c['foo'] ];

// If-then-else shorthand
return($a ? $a : $b);
```

Separation of Statements

In general, all statements, procedures, functions, etc are separated by a blank line. However, "like" statements (such as blocks of variable declarations or calculations performed as part of an operation) are grouped together without an intermediate blank line.

```
part of previous function;
  end of previous function;

  return foo;
}

function textFunction() {
  $var = 0;
  $othervar = 1;

  beginning of next function;
```

End of Line

There should be no trailing white space at the end of a line.

End of File

One or more extra newlines at the end of a file may cause errors (the page does not render) on some systems, including MAMP and Ubuntu, and so must be avoided.

See also #NoClosingTag.

Alignment

Curly Braces

Curly braces ({,}) start on the same line as the introductory statement, separated by a space. Continuation statements (such as else) begin on the same line as the previous block's closing brace, again separated by a space.

Curly braces may be omitted only when each contained clause is one line, however in general they should be used.

OK:

```
if($a > 0) {
  return true;
} elseif($b > 0) {
  return false;
}

if($a > 0)
  return true;
elseif($b > 0)
  return false;
```

Not OK:

```
if($a > 0) {
  $a++;
  $foo = bar;
} else
  return false;
```

Parentheses

Parentheses are used even when considered optional. The exception is return statements, since return is a language construct and not a function.

OK:

```
print ($a ? $a : $b);

return false;
```

Not:

```
print $a ? $a : $b;

return(false);
```

When creating a list within parentheses, such as when constructing an array or passing parameters to a function, if the entire list is not readable on one line, indent one level and begin on the next line:

```
$v = array('small');
$w = array(
  "the beginning of a longer list",
  "the continuation of a longer list",
  "the conclusion of a longer list"
);
```

Hashes and Arrays

Historically, PHP array shorthand notation using brackets `[]` is not used, since the project predates this capability. However, the project will likely switch to this style as part of the Framework Migration.

When constructing a hash, align on the => as much as possible:

```
$h = array(
  'A'  => "Option A",
  'B'  => "Option B",
  'YZ' => "Option YZ"
);
```

When constructing an array, align on the = as much as possible:

```
$args['foo']['bar']        = "Sushi";
$args['foo']['restaurant'] = "Hibachi";
```

Conditionals

`if` statements with complex conditionals should have each conditional on a new line, aligned with the previous conditional. Nested conditionals should be further indented. Exceptions are made for those that are easy to read and fit on one line.

```
if(($a == $b)
   || (($a == $c)
       && ($d == $e))
```

Comments

Comment Delimiters

Comments are exclusively delimited with double slashes (`//`). The exceptions are the docblock comments, and code that may be temporarily commented with C-style comments (`/* */`).

Top of File

The top of each file must include the following header in docblock format. Do NOT include copyright information, as this is provided in the master NOTICE file.

```
/**
 * COmanage Directory People Controller
 *
 * Portions licensed to the University Corporation for Advanced Internet
 * Development, Inc. ("UCAID") under one or more contributor license agreements.
 * See the NOTICE file distributed with this work for additional information
 * regarding copyright ownership.
 *
 * UCAID licenses this file to you under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with the
 * License. You may obtain a copy of the License at:
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * @link          http://www.internet2.edu/comanage COmanage Project
 * @package       registry
 * @since         COmanage Registry v2.0.0
 * @license       Apache License, Version 2.0 (http://www.apache.org/licenses/LICENSE-2.0)
 */
```

Use "Registry" or "Directory" as appropriate.

`@package` is either `directory` or `registry`.

~~Note that `$Id` is an SVN keyword that must be set on a per-file basis. (It will not be updated automatically otherwise.) This can be done with~~

```
svn propset svn:keywords "Id" foo.php
```

~~Don't add a new file with the Id string copy and pasted from a different file.~~

Function Descriptions

Functions must have a docblock of the following form *preceding* the function definition:

```
/**
   * Update the amount of foo
   * - precondition: $this->foo set
   * - postcondition: Flash message set on error
   *
   * @since  COmanage Directory 0.1
   * @todo   Handle nested foo
   * @param  integer $i New amount of foo
   * @param  string  $s New name for foo
   * @return void
   * @throws RuntimeException
   */

  public function foobar($i,
                         $s) {
```

Comments and descriptions are written in normal English, except periods are omitted if there is only one sentence.

precondition, *postcondition*, *todo*, *param*, and *throws* are optional, and should be omitted if no relevant information applies.

Inline Comments

Comments should be included where it isn't patently obvious what is going on. Comments should be written in normal English using normal grammar and syntax.

```
// We need to figure out what the person's name is as asserted by
// the home organization, so pull it from the Org Identity.
```

When a code block relates to a detailed JIRA issue (eg: a bug or a specific change), link to the issue in the comment. Do not link to JIRA for big feature requests (those that encompass more than a small block of code).

```
// This change is for CO-90210. Don't undo it!
// https://bugs.internet2.edu/jira/browse/CO-90210
```

Data Filtering

Input and output sanitization should be achieved using standard PHP filters. Cake's native Sanitize:: filter has been deprecated as of Cake 3 and should be avoided. Guidelines for converting existing Cake Sanitize:: filters to PHP filters is documented in CO-667.

For input validation, see the PHP validate filters reference.

For views producing html output to a browser, all user supplied content must be escaped. In most cases filter_var with FILTER_SANITIZE_SPECIAL_CHARS is appropriate:

filter_var for output escaping

```
<?php print filter_var($var, FILTER_SANITIZE_SPECIAL_CHARS); ?>
```

If output stripping is required or desired, use FILTER_SANITIZE_STRING with or without extra stripping flags, e.g. the following will strip tags as well as new lines (and any other character < 32):

filter_var for output stripping

```
<?php print filter_var($var, FILTER_SANITIZE_STRING, FILTER_FLAG_STRIP_LOW); ?>
```

See PHP filters and the PHP sanitize filters reference for more information.

Note that Cake's link() function ($this->Html->link()) will escape title and attributes unless escaping is explicitly set to false. To avoid double-encoding strings, allow Cake to manage escaping when using link(). For more information about link(), visit Cake's HTML Helper documentation (version 2 / version 3).

Strings passed to the addCrumb() function should be escaped when no arguments accompany the string. When arguments are passed, Cake will generate a link() and escape the string, so do not filter a string passed in this case or you will double-escape it. When no arguments are passed, the string will be echoed to the output and must be filtered using filter_var.

PHP-isms

No Short Tags

The full PHP tag must be used, since short tags require server configuration.

```
<?php some stuff; ?>
```

Echo Tags

As of Registry v5, echo tags are used.

```
<?= __('registry.op.add'); ?>
```

No Closing Tag

To avoid errors related to whitespace at the end of file, do *not* close PHP files with ?> if they are primarily PHP code.

print, not echo

echo is a language construct, not a function, and so its use within COmanage is deprecated to avoid unexpected behavior. (`print` is also a language construct, but behaves like a function.)

Scoping

All object-oriented methods and variables must be appropriately and explicitly scoped (`private`, `public`, etc).

Controls Split Across Tags

When splitting control structures across multple `<?php  ?>` tags (ie: to intersperse HTML), use colon notation with comments in closing tags.

Preferred:

```
<?php if($foo): ?>
  stuff;
  <?php if($bar): ?>
    morestuff;
  <?php endif; // bar ?>
<?php endif; // foo ?>
```

Not Preferred:

```
<?php if($foo) { ?>
  stuff;
<?php } ?>
```

Logical Operators

`and` and `or` are **NOT** the same as `&&` and `||`. They have lower precedence, and are almost certainly not what you want to use.

CakePHP-isms (v2)

Arrays as Arguments

Where an array is required as an argument (used very commonly in CakePHP), define the array first and then pass it.

Preferred:

```
$args = array();
$args['fields'][] = "MAX(ordr)+1 as m";
$args['order'][] = "m";

$o = $this->CoEnrollmentAttribute->find('first', $args);
```

Not Preferred:

```
$o = $this->CoEnrollmentAttribute->find('first',
                                        array('fields' =>
                                              array("MAX(ordr)+1 as m")),
                                        array('order' =>
                                              array("m")));
```

SQL Query Optimization

In general, use Containable Behavior to constrain what Cake is pulling from the database. Cake will usually try to pull a bunch of associated data, which may or may not match what you actually need in a given context. You can use Containable to specify exactly which associated data you want returned.

Don't return anything but CoPerson:

```
$this->CoPerson->contain();
```

Obtain OrgIdentity and Name:

```
$this->OrgIdentity->contain('Name');
```

Obtain OrgIdentity, Name, CO, and CO Groups the Org Identity is a member of:

```
$args['contain'][] = 'Name';
$args['contain']['CoOrgIdentityLink']['CoPerson'][0] = 'Co';
$args['contain']['CoOrgIdentityLink']['CoPerson']['CoGroupMember'] = 'CoGroup';
$orgIdentities = $this->OrgIdentity->find('all', $args);
```

URLs (API)

URLs are specified in REST API v1 and REST API v2. The general form is

```
/controller/id.format
```

where *format* is the requested response format. The action is specified by the HTTP verb:

- GET: index (if no *id*), view

- POST: add
- PUT: edit

URLs (Browser)

GET operations follow the form

```
/controller/action/arguments
```

as in the example

```
/co_people/index/co_id:2
```

arguments should map to their database name (ie: underscored inflection) whenever possible. Note this standard wasn't particularly well followed prior to Registry v5.

POST operations follow the form

```
/controller/action
```

as in the example

```
/co_people/add
```

Arguments are embedded in the POST body. This facilitates the scenario where a form submission errors out (perhaps the user left out a required field) and Cake regenerates form with a URL having no args.

GET operations that are intended to convert to a POST operation (ie: an add or an edit that renders a form via GET and then POSTs the form) must have the URL arguments converted to hidden variables, usually by the form generation in the corresponding `fields.inc`.

Use Containable, not Recursive

`$this->recursive`, in addition to being too blunt an instrument to determine how much associated data is retrieved with a model (it generates way too many SQL queries in most circumstances), also doesn't currently work correctly with the Grouper datasource (CO-268).

Use of Linkable Behavior is also acceptable, though it should be considered somewhat experimental.

Magic Finds

Do not use the magic `findByX()` functions, as they do not appear to trigger Behaviors (especially ChangelogBehavior).

updateAll()

Do not use `updateAll()`, as it does not trigger Behaviors (especially ChangelogBehavior).

CakePHP-isms (v4)

Application Rules

Application Rules are applied on tables to enforce complex logic on save. By convention, COmanage application rules are named `ruleXXX()`.