Recite CMS Development Guide

Recite CMS 2.1.3

Recite CMS Development Guide

Table of Contents

List of Examples

Chapter 1. Introduction

This document is a guide to custom development in Recite CMS. Custom development may involve creating custom modules; custom control panel widgets; custom drivers for existing modules; or extending Recite CMS in some other way.

Regardless of how you extend Recite CMS, this document contains the fundamental knowledge required in order to complete such development. For specific details on implementing item such as modules or drivers, please refer to relevant documentation.

The following documents are also available:

- Recite CMS Module Development Guide
- Recite CMS Control Panel Widget Development Guide
- Development guides for various driver types

Chapter 2. Getting Started

Recite CMS is written in PHP. It will only work in PHP 5.1.2 and newer. At time of writing, it runs on Unix-based systems (such as Linux, FreeBSD or Mac OS X). Microsoft Windows support is experimental only. Additionally, it currently supports Apache HTTP Server and PostgreSQL or MySQL database server.

For all development you will require a text editor or Integrated Development Environment (IDE). One editor you can use is jEdit (http://jedit.org). This editor works on all major platforms, and also has a PHP-editing mode.

Loading PHP Classes

Recite CMS uses an automatic class-loading system, meaning you don't have make calls to PHP functions such as `require_once()` to load PHP scripts. This relies on all classes (including classes you develop for your custom modules) being named appropriately and stored on the filesystem correctly. Classes are automatically loaded when you try to use them.

For example, the `Text_Lipsum` class (used to generate dummy "Lorem Ipsum" text) needs to be stored in the application include directory with a path of `Text/Lipsum.php`. In other words, the class filename should end in `.php` and underscores in the class name correspond to slashes in the path.

Below are two identical ways in Recite CMS to use this class. Firstly, manually including the class:

Example 2.1. Manually including the PHP class

```
<?php
    require_once('Text/Lipsum.php');
    echo Text_Lipsum::sentence();
?>
```

Example 2.2. Auto-loading the `Text_Lipsum` class simply by using it

```
<?php
    echo Text_Lipsum::sentence();
?>
```

It is also important to note that each module in Recite CMS has its own class namespace [1] and include directory. As you will see in the "Developing Custom Modules" document, each module will have the file path `./name/include`. For example, if you have a module called **Timesheets** and a class called `Report`, its full path (relative to the modules directory) would be `./timesheets/include/Report.php`. You must also name the class accordingly, prefixing it with `Module_Timesheets_`.

For example, to define the class (called `Module_Timesheets_Report`), you might use the following code.

Example 2.3. Creating the `Module_Timesheets_Report` class (`./timesheets/include/Report.php`)

```
<?php
    class Module_Timesheets_Report
    {
        public function __construct()
        {
        }
    }
?>
```

[1] Not technically a namespace, but for the purposes of this document we'll assume that it is.

Once you have created the class, it would be somewhat cumbersome to include the full path to the class in order to use it. Thanks to the class auto-loader, you don't need to. You can simply use the following code.

Example 2.4. Auto-loading a module class simply by using it

```
<?php
    $report = new Module_Timesheets_Report();
?>
```

Zend Framework

The Zend Framework is an open-source PHP 5 library that is used extensively within Recite. It provides a number of useful modules that with building an application that easily scales.

Recite makes use of the Zend Framework wherever possible. You can find documentation on the Zend Framework at http://framework.zend.com/manual/en.

Chapter 3. Application Environment

This section contains information about the Recite CMS application environment. Certain items and objects (such as current client and user information) are available from all custom code that you implement with the Recite CMS framework.

Current Client

You will frequently need access to the current client, since all data for a site is tied to that particular client. You can retrieve the active client object (an instance of the `Application_Client` class) using the following code.

Example 3.1. Retrieving the active client object

```
<?php
    $client = Application::GetClient();

    // output the client ID
    echo $client->getId();
?>
```

You can use this method in both the control panel and on client sites (such as in custom request handlers).

Alternatively, if you only need the ID of the current client you can use the `getClientId()` function.

Example 3.2. Retrieving the ID of the active client

```
<?php
    $clientId = getClientId();
?>
```

Current User

The methods for retrieving the current user in client sites is slightly different from the Control Panel. Each method is as follows.

Control Panel

To retrieve the current user, use the following code.

Example 3.3. Retrieving the active user object

```
<?php
    $user = Application::GetUser();
?>
```

This object will always correspond to a valid user with Control Panel access.

Client Site

On the client site, you're not always guaranteed to have a logged-in user. If the user is logged in, they will always belong to a custom user directory that belongs to the active client. If you try to retrieve the user and a user is not logged-in, an exception is thrown.

Example 3.4. Retrieving the active user

```
<?php
   try {
       $user = Application::GetUser();
       // user is logged-in
   }
   catch (Exception $ex) {
       // user is not logged-in
   }
?>
```

If the user is not-logged in, you can retrieve a unique hash that corresponds to the user (based on their IP address and web browser). This is an MD5 hash (32 characters long made up of hexadecimal characters). You can retrieve this hash using the following code.

Example 3.5. Retrieving the unique user hash.

```
<?php
    $hash = Application::GetUserHash();
?>
```

You can access this hash even if the user is logged-in but you typically won't need it. The following listing shows how you can retrieve the user value you need.

Example 3.6. Retrieving the user or the hash

```
<?php
    try {
        $user = Application::GetUser();

        // user is logged-in
        // do something with the user object
    }
    catch (Exception $ex) {
        $hash = Application::GetUserHash();
        // do something with the user hash
    }
?>
```

Database Abstraction

In order to query the Recite CMS database you will need the database abstraction object. You can retrieve it with the following code. Note that this call will never fail - a valid connection is guaranteed to be returned.

Example 3.7. Retrieving the active database connection

```
<?php
    $db = Application::GetDb();
?>
```

The database object that is returned from this call inherits from the Zend_Db_Adapter_Abstract class, meaning you can use the documentation at http://framework.zend.com/manual/en/zend.db.adapter.html as a reference.

Maintaining Cross-Platform Compatability

Recite CMS provides helper objects to maintain cross-database compatibility. For instance, to retrieve a formatted timestamp in MySQL the `date_format()` function is used, whereas PostgreSQL uses `to_char()`.

To access the helper object for the current database connection, call `$db->getStatementHelper()`.

The following methods are available from the returned object:

- `getRandomExpr()`. Get the expression used for selecting or ordering by a random field. Returns `Zend_Db_Expr`.
- `getTimestamp($ts)`. Accepts a unix-timestamp (such as from PHP's `time()` function) and returns a value in a format that can be inserted into a database table.
- `unixTimestamp($val)`. Accepts a value from the database and returns a unix timestamp. Returns null
- `getOlderThanExpression($field, $qty, $unit, $orEqual = false)`. Get a database expression that returns true if the given database value is older than the passed interval. The `$unit` variable can be one of `Zend_Date::SECOND`, `Zend_Date::MINUTE`, `Zend_Date::HOUR`, `Zend_Date::DAY`, `Zend_Date::MONTH` or `Zend_Date::YEAR`. Returns an instanceof `Zend_Db_Expr`
- `getYoungerThanExpression($field, $qty, $unit, $orEqual = false)`. This is the same as the previous function except it returns an expression that returns true if the database value is younger than the given value.
- `getDateString($field)`. Returns a `Zend_Db_Expr` expression to fetch the given field in `YYYY-MM-DD` format.
- `getDateTimeString($field)`. Returns a `Zend_Db_Expr` expression to fetch the given field in `YYYY-MM-DD HH:MM:SS` format.

Chapter 4. Audit Log

The Recite CMS audit log allows you as a developer to record exactly what is occurring in an installation of Recite CMS. Typically this will be to record exactly what users are doing, however the log can also be used to record other activity such as automated scripts or communication with third-party services.

The audit log can be accessed via the Recite CMS Administration Site.

Creating a New Audit Entry

To create a new audit entry, the `Application_Auditor_Message` class is used.

Typically an audit entry will contain one or more messages. These are intended to be human-readable strings of information.

You can write a message to the entry either by passing it as the first argument when instantiating `Application_Auditor_Message`, or by calling the `add()` method on the returned instance.

To write the entry to the Recite audit log, call the `record()` method.

Example 4.1. Creating and recording an audit message

```
<?php
    $message = new Application_Auditor_Message('Some action occurred');
    $message->add('This is a secondary message')
            ->record();
?>
```

> **Note**
>
> As demonstrated in this example, you can chain method calls together in this class.

Associating Audit Entries With a Module

Typically your audit messages will be associated with a particular module. You can set which module your audit entry relates to by passing the module as either the second argument to the `Application_Auditor_Message` constructor, or by calling the `setModule()` method.

You can pass either the name of a module, or an instance of `Application_Module_Interface`.

Setting the module on your audit messages allows administrators to better understand your audit messages, and also allows them to easily filter messages. This can also help you in development of your module.

The following shows an example of setting the module on your audit message.

Example 4.2. Setting the module for an audit message

```
<?php
    // set module in constructor
    $message = new Application_Auditor_Message('Some action occurred', 'mymodule');
    $message->record();

    // set module with setModule() method
    $message = new Application_Auditor_Message('Some action occurred');
    $message->setModule('mymodule')
            ->record();

    // pass an instance of Application_Module
    $module = Application_Module_Manager::Factory('mymodule');
    $message = new Application_Auditor_Message('Some action occurred', $module);
    $message->record();
?>
```

Including Custom Data With Audit Message

In addition to passing one or more strings with your audit messages you may want to including some other arbitrary data.

To do so, call the `addMemo()` method. The first argument is the name of the memo value while the second argument is the value. It is recommended you pass only simple values such as strings or ints as other values may produce inconsistent results.

You can add multiple memos with the same name to a single audit entry.

Example 4.3. Including memo data with an audit message

```
<?php
    $message = new Application_Auditor_Message('Item created', 'mymodule');
    $message->addMemo('title', 'XYZ Product')
            ->addMemo('country', 'Australia')
            ->record();
?>
```

Shorthand Method of Creating Audit Entries

You can write audit entries that consist of only a string information string (and no memo data) by calling the `Application_Auditor::Record()` method. This method accepts as its first argument either a string or an instance of `Application_Auditor_Message`.

Example 4.4. Shorthand Method of Writing to Audit Log

```
<?php
    // pass a string
    Application_Auditor::Record('Some action occurred');

    // pass an object
    Application_Auditor::Record(new Application_Auditor_Message('Some action occurred'));
?>
```

Chapter 5. Maintenance Queue

Frequently you will want a task to be performed immediately but it may take some time to execute. For example, if you want to process a credit card transaction it may take several seconds or minutes to process, so therefore this should be done in the background.

In order to achieve this, you can schedule a maintenance task. If Recite CMS has been correctly configured the maintenance queue will be processed every few minutes, so if you queue a task it will be performed in the near future.

There are two steps involved in doing this:

1. Create a maintenance script by extending the `Application_Maintenance_Abstract` class.
2. Schedule its execution by calling the `queue()` method on the queue item object.

Creating a Maintenance Script

To create a new maintenance you must either implement the `Application_Maintenance_Interface` interface, or extend the `Application_Maintenance_Abstract` class.

Note

It is recommended you extend the `Application_Maintenance_Abstract` class so you pick up any new functionality if it is added to the interface, and also so your existing scripts won't break if new methods are added to the interface.

The methods you must implement in your maintenance script are as follows:

- `public function __construct()`

  The class constructor accepts no arguments. If you extend the `Application_Maintenance_Abstract` class you do not need to create this method. If you choose to implement this method be aware that this class may be instantiated even when not being run, so ensure the constructor does not cause any unexpected side-effects or have a high execution cost.

- `public function __toString()`

  Returns: `string`

  This method should return a short descriptive string for the action the maintenance script performs. This is primarily used by the administrator when viewing the maintenance queue.

- `public function run(array $options)`

  Argument 1: `array`. This argument holds an arbitrary list of parameters used for executing the maintenance script.

  Throws: `Application_Maintenance_Exception`

  This is the method that is run by the maintenance queue. If required, you can throw the specified exception in this function to indicate that an error has occurred. Doing so does not cause any side-effects (such as requeueing)

- `public function accept(array $options)`

  Argument 1: `array`. This argument holds an arbitrary list of parameters used for executing the maintenance script.

Returns: `Boolean`

This method is used to decide whether or not the maintenance script can be queued. The arguments passed are identical to if the `run()` method was called. Return true if the script can be added to the queue or false if not.

Scheduling the Maintenance Script to Run

In order the schedule the maintenance script you've created you use the `Application_Maintenance_QueueItem` class. Create an instance of this class then call its `queue()` method.

When instantiating `Application_Maintenance_QueueItem`, pass the name of your maintenance script class as the only argument to the constructor.

Example 5.1. Queueing up a maintenance script

```
<?php
    $queueItem = new Application_Maintenance_QueueItem('Your_Maintenance_Class');
    $queueItem->queue();
?>
```

Storing Custom Data With Maintenance Queue Item

Often you will want to store data with your maintenance queue item. This is data that is passed to the `run()` method of your maintenance script.

To store data with the queue item use the `setParam()` method. The first argument to this method is the name of the parameter while the second is the corresponding value.

Example 5.2. Storing custom data with maintenance queue item

```
<?php
    $queueItem = new Application_Maintenance_QueueItem('Your_Maintenance_Class');
    $queueItem->setParam('foo', 'bar')
              ->queue();
?>
```

Scheduling the Queue Item

You can decide when the execute the queue item by calling the `setCreatedTimestamp()` method. This method accepts as its only argument a UNIX timestamp which indicates when the queue item should be run. You can generate this value with PHP's `mktime()`.

Example 5.3. Scheduling Queue Item Execution For January 1, 2011.

```
<?php
    $queueItem = new Application_Maintenance_QueueItem('Your_Maintenance_Class');
    $queueItem->setCreatedTimestamp(mktime(0, 0, 0, 1, 1, 2011))
              ->queue();
?>
```

Note

If you specify a time in the past, the queue item will be executed next time the queue is processed.

Retaining Client Context

When the maintenance queue runs it is running for the entire Recite CMS installation, not for a specific client. Since a queue will typically be the result of an action on a particular client's web site, you will need access to that client in your maintenance script.

Recite CMS automates this process of retaining client context for you. That is, you can still call `Application::GetClient()` and `getClientId()` and be sure that it will correspond to the correct client.

Viewing the Maintenance Queue

It is possible to view the current maintenance queue using the Recite CMS Administration Site. This view allows you to see queue items that are currently running and also future queue items. You can also cancel future queue items.

For full details on managing the maintenance queue, refer to the Recite Administration Guide.

Command-Line Tools

There are several command-line tools available to help you manipulate the maintenance queue. Each of these is available in the `./application/tools/maintenance` directory.

Queueing A Maintenance Task

To add an item to the maintenance queue manually, use the `queue.php` script.

The first argument is the class name of the maintenance script while any subsequent **`key=value`** pairs are used as options for the item.

For instance, **./queue.php** ***Your_Maintenance_Class param1=foo***.

You can set client context for the item by specifying a parameter called `client_id`. For instnace, For instance, **./queue.php** ***Your_Maintenance_Class*** **client_id=*id param1=foo***.

Running a Maintenance Task Immediately

Similar to queueing a maintenance task from the command-line, you can also run it in real-time using the `run.php` script.

This script accepts parameters in the same fashion (including setting client context).

Processing Maintenance Queue

You can process the maintenance queue on-demand by using the `process.php` script. This script accepts no argument. It is the same script that administrators set up as a cron job / scheduled task when installing Recite CMS.

Chapter 6. Event Handling

Recite CMS allows you to write event-driven scripts. These are scripts that are executed only when certain events occur.

For example, if you want to run a custom script after a user is deleted you can easily achieve this by "listening" for the `/module/users/user/postdelete` event.

There are many different events that occur that you can listen for. Additionally, you can trigger your own events that other scripts can listen for. All of this is covered in this section.

In Recite CMS, a script that listens to an event is called an *observer*.

Event Naming

Events are named similar to a filesystem path. That is, it is made of several segments, each separated by a slash. This mechanism is used to easily map observers to event names.

For example, the event that is triggered when a user is created is called `/module/users/users/postinsert`. It doesn't matter whether or not there is a slash at the start or event of the event name.

If you attempt to trigger an event with a name that doesn't follow this style, the event will not be triggered.

For the purposes of this chapter we'll use a sample event name called `/some/sample/event`.

Triggering Events

An event is triggered in Recite CMS using the `Application_Event` class. You can associate any data as required with the event. Additionally, you can receive data back from observers.

To create a new event, instantiate the `Application_Event` class. This class accepts as its sole argument to the constructor the name of the event. You can then call the `trigger()` method to trigger the event.

Example 6.1. Creating and triggering an event

```
<?php
    $event = new Application_Event('/some/sample/event');
    $event->trigger();
?>
```

At this point, all observers for this event will be triggered. There is no defined order of execution for these observers.

Including Data With An Event

You can include data with an event using the `set()` method. This method accepts the name of the value as its first argument and the value as its second argument. Observers can then access this data.

Example 6.2. Including data with an event

```
<?php
    $event = new Application_Event('/some/sample/event');
    $event->set('email', 'foo@example.com')
          ->trigger();
?>
```

Handling Data Returned by Observers

It is possible to trigger events that expect data to be returned. When triggering an event, you can specify that you expect return data. This results in the event manager returning to you any data returned from all observers.

To do so, call the `responds()` method. This will result in an instance of `Application_Event_ResponseCollection` being returned from the call to `trigger()`.

Example 6.3. Receiving a response from an event

```
<?php
    $event = new Application_Event('/some/sample/event');
    $event->responds();

    $responseCollection = $event->trigger();
?>
```

You can then make use of the response data as required. To do so, call the `getResponses()` method. This returns an array of `Application_Event_Response`. Each instance corresponds to a single observer.

> **Note**
>
> Not every observer is guaranteed to send back a response.

Each response can have any amount of data associated with it (each value indexed by a unique name). You can retrieve any value by calling the `get()` method, or you can retrieve all of the data using `getData()`

If the given value doesn't exist in the response, null is returned.

Example 6.4. Using an event response

```
<?php
    $event = new Application_Event('/some/sample/event');
    $event->responds();

    $responseCollection = $event->trigger();

    foreach ($responseCollection->getResponses() as $response) {
        $response->get('foo'); // get the foo value from a response
    }
?>
```

Listening for Events

You can write your a custom event handler by creating a PHP class that extends from `Application_Event_Observer_Abstract`.

Observer Name and Location

The name of the event you're observing determine the path, filename and class name of the class. The name of the event is also its path (with `.php` appended) on the filesystem. The class name is the name of the event with slashes replaced by underscores.

For example, a handler for `/some/sample/event` would have the filename `./basepath/some/sample/event.php`. Its PHP class name would be *`classprefix`*`_some_sample_event`.

The base path and base class name depend on whether your observer lives with a module or if it lives with a driver.

If the observer is for a module, it must live inside the `Observers` directory in the module include path. For example, `./lib/modules/ModuleName/include/Observers/some/sample/event.php`. The class name would be `ModuleName_Observers_some_sample_event`.

Example 6.5. Creating a module observer (`event.php`)

```
<?php
    class ModuleName_Observers_some_sample_event extends Application_Event_Observer_Abstract
    {
        // observer code here
    }
?>
```

If the observer is for a driver, it must live inside the `observers` directory in the the driver path. For example, `./lib/drivers/path/to/driver/observers/some/sample/event.php`. The class name would be `Driver_path_to_driver_observers_some_sample_event`.

Example 6.6. Creating a driver observer (`event.php`)

```
<?php
    class Driver_path_to_driver_observers_some_sample_event extends Application_Event_Observer_Abstrac
    {
        // observer code here
    }
?>
```

Implementing the Observer

The only method you must implement when creating an observer is the notify method.

- `public function notify(Application_Event $event)`

This method accepts an instance of `Application_Event` as its only argument. This is the same class we used when triggering an event.

To retrieve the data stored with the event, call the `get()` method. This method accepts the name of the data the retrieve.

Example 6.7. Observing an event

```
<?php
    class Driver_path_to_driver_observers_some_sample_event extends Application_Event_Observer_Abstrac
    {
        public function notify(Application_Event $event)
        {
            $foo = $event->get('foo');

            // do something with the foo value
        }
    }
?>
```

Sending a Response

An observer can send a response, which the event triggerer can decide to use or not. You can check if a response is requested by calling the `getResponds()` method on the passed event.

To respond to an event, return an instance of `Application_Event_Response`. You can set any data to be passed along with this object, which will then made available to the requestor.

Example 6.8. Observing and responding to an event

```
<?php
    class Driver_path_to_driver_observers_some_sample_event extends Application_Event_Observer_Abstrac
    {
        public function notify(Application_Event $event)
        {
            if (!$event->getResponds()) {
                return;
            }

            $response = new Application_Event_Response();
            $response->set('success', true);

            return $response;
        }
    }
?>
```

Registering An Observer

In order to speed up response time, Recite CMS caches a list of the available observers for any given event. Once you've created a new observer, you can either clear the system-wide caching using the Recite CMS Administration Site, or you can clear the observer cache using the `clear-cache.php` script in the `./application/tools/events` directory.

To ensure Recite CMS can see your observer, you can run the `get-observers.php` script in the same directory. This script accepts the name of the event as its only argument.

For instance run **./get-observers** */some/sample/event* to see if your observer is being detected.

List of Triggered Events

This section is pending completion.

Chapter 7. Checking Permissions in the Control Panel

In Recite CMS it is possible to restrict which actions each user is allowed to perform. Permissions are defined on a per-role basis. That is, each user belongs to a certain role, and each role has certain things they are and aren't allowed to do.

Users and roles can be managed either from within the Control Panel or from within the Recite administration section. This chapter does not deal with how to manage users and roles, but rather, shows you to create and check permissions.

How The Control Panel Handles Permissions

When you perform a (failed) permissions check in your PHP code, an `Application_User_Permissions_Exception` exception is thrown. Recite will automatically catch this exception and send an appropriate message to the Control Panel so the user knows a permissions error occurred. As a developer all you need to do is ensure the exception is thrown when you want permissions enforced.

Sometimes you simply want to check if a user has permission to do something (rather than telling them they don't have permission to do something). The Recite permissions API allows you to easily do this. This is covered later in this chapter.

Defining Permissions

Permissions are defined on a per-module basis using the module's `permissions.xml` XML file. This is a file stored in `./lib/modules/modulename/settings/permissions.xml`.

You can find more information on creating this file in the Recite CMS Module Development Guide.

Checking Permissions

Checking permissions is achieved using the Recite ACL manager. In order to perform a permissions query the `Application_User_Permissions_Query` and `Application_User_Permissions_Query_Item` classes.

Checking a Single Permission

To check a single permission use the `Application_User_Permissions_Query_Item` class. The name of the permission to check is passed as the only argument to the constructor.

You can then call the `check()` to check the permission. This method returns true if the current user has the permission or false if not.

Alternatively, you can use the `assert()` method. If the current user does not have access the `Application_User_Permissions_Exception` exception is thrown.

Example 7.1. Checking a single permission with both check() and assert()

```
<?php
    $item = new Application_User_Permissions_Query_Item('some:permission:to:check');

    if ($item->check()) {
        // current user has permission
    }
    else {
        // current user does not have permission
    }

    try {
        $item->assert();

        // current user has permission
    }
    catch (Exception $ex) {
        // current user does not have permission
    }
?>
```

Typically you won't have to explicitly catch the exception since Recite will handle a permissions exception automatically.

You can short-cut this code using the static `BuildAndCheck()` or `BuildAndAssert()` methods. You can use these when you don't need access to the permissions query item.

Example 7.2. Using Shorter Notation for Permissions Check

```
<?php
    if (Application_User_Permissions_Query_Item::BuildAndCheck('some:permission:to:check')) {
        // current user has permission
    }
    else {
        // current user does not have permission
    }

    try {
        Application_User_Permissions_Query_Item::BuildAndAssert('some:permission:to:check');

        // current user has permission
    }
    catch (Exception $ex) {
        // current user does not have permission
    }
?>
```

Checking Multiple Permissions

In the previous section I showed you how to check a single permission. Sometimes you will want to check a series of permissions at once. This will typically involve either wanting *all* of the permissions to pass, or *any* of the permissions to pass.

Requiring All Permissions to Pass

If you want a permissions query in which all permissions must succeed, you want an *AND* query.

To create a new AND query you must create a new instance of the `Application_User_Permissions_Query` class by calling its static `NewAnd()`. This will return an empty query which you can add permissions to.

To add a permission to query call the `add()` method. You can pass either a string (the name of the permission) or an instance of `Application_User_Permissions_Query_Item`.

Once you've built the query you can call either `check()` (to return a `boolean`) or `assert()` (to thrown an `Application_User_Permissions_Exception` exception).

Example 7.3. Checking multiple permissions at once with an "AND" permissions query

```
<?php
    $query = Application_User_Permissions_Query::NewAnd();
    $query->add('first:permission:to:check')
          ->add(new Application_User_Permissions_Query_Item('second:permission:to:check'))
          ->assert();
?>
```

In the above example both permission checks must succeed to avoid the exception being thrown.

> **Important**
>
> Permissions are processed lazily and in-order. This means that in an "AND" query, if a single permission check fails then no more permissions are checked. The resultant exception will only contain details about the permissions check that failed and not the remaineder of the permissions that might have failed if it had proceeded.

Requiring Any Permission to Pass

If you have a two or more permissions to check but only one of them needs to pass then you want an "OR" query.

The code used to build and check the query is the same as an "AND" query, except that you use the `NewOr()` static method to create an empty query object.

Example 7.4. Checking multiple permissions at once with an "AND" permissions query

```
<?php
    $query = Application_User_Permissions_Query::NewOr();
    $query->add('first:permission:to:check')
          ->add(new Application_User_Permissions_Query_Item('second:permission:to:check'))
          ->assert();
?>
```

In the above example either permission check must succeed to avoid the exception being thrown.

> **Important**
>
> Permissions are processed lazily and in-order. This means that in an "OR" query, as soon as a permissions check succeeds the query is complete. If the query fails then the resultant exception will contain details about every single permission in the query.

Using Permissions in Templates

Checking permissions in templates is an important aspect of implementing a permissions system. For example, if a user is not allowed to create a page, then there's no point in showing them a **Create Page** button.

Note

Alternatively, you could show them the button then if they click on it give them a message as to why they can't use it. Even in this example you would likely need to check the permissions in the template so you know to display the alternative message.

To check permissions in templates, use the `is_allowed` modifier on the name of the permission. This modifier allows you to check a single permission at a time, and returns either true or false.

Tip

If you need more complex permissions checks, it is recommended you process these in your controller script rather than in the template.

Example 7.5. Checking a Permission From Within a Template

```
{if 'some:permission'|is_allowed}
    <input type="submit" value="Create Something" />
{else}
    <input type="submit" value="Create Something" disabled="disabled" />
{/if}
```

Using the Recite Permissions Exception

When an `Application_User_Permissions_Exception` exception is thrown as a result of a failed permissions check, it is possible to retrieve a list of the failed permissions

Warning

As noted previously, if an "AND" permissions query fails then only the first failed permission will be available.

To retrieve a list of the failed permissions, call the `getPerms()` method on the exception object. An array of the failed permissions will be returned. Each element in the array will either be the string name of the permission or an instance of `Application_User_Permissions_QueryItem`.

Example 7.6. Retrieving the Failed Permissions

```
<?php
    try {
        Application_User_Permissions_Query_Item::BuildAndAssert('some:permission');
    }
    catch (Application_User_Permissions_Exception $ex) {
        foreach ($ex->getPerms() as $perm) {
            echo sprintf("Failed: %s\n", $perm);
        }
    }
?>
```

Chapter 8. Caching

Recite CMS uses an on-disk caching mechanism to achieve significant performance gains when rendering client web sites.

As a developer, you can control how the cache operates in many different situations. For example, when developing custom container rules, you specify options to control how caching works for the output of the container (as well as for the page on which the container resides).

This chapter tells you to manipulate the cache so you can leverage the speed of Recite CMS in your own code, as well as being able to ensure that up-to-date content is being displayed.

Several of the other Recite CMS developer guides (available at `Recite CMS Documentation Portal [http://docs.recite.com.au]`) refer back to this chapter.

Specifying Cache Options

In certain instances (such as when developing container rules), custom code will be required to return options that define how caching should work. These options are specified using the `Application_Cache_Options` class.

This class is primarily used to specify the maximum cache lifetime of the content in question, as well as any tags (keywords) that should be stored with the cache item.

Disabling the Cache

If you do not want the content in question to be cached at all, call the `cancel()` method.

Example 8.1. Preventing caching of the content in question

```
<?php
    $options = new Application_Cache_Options();
    $options->cancel();
?>
```

Setting the Maximum Lifetime

You can specify the maximum amount of time (in seconds) content can be cached for using the `setMaxLifetime()` method. Recite CMS may decide to cache the content in question only for a shorter amount of time, but never for longer.

You can set an unlimited maximum lifetime (that is, so it never expires) by passing `Application_Cache_Options::LIFETIME_UNLIMITED`

Example 8.2. Specifying the maximum lifetime of a cache entry

```
<?php
    $options = new Application_Cache_Options();
    $options->setMaxLifetime(86400); // 1 day (in seconds)
?>
```

Setting the Tags

You can associate any number of tags (keywords) with a cache entry by calling the `addTag()` method. This is an extremely useful feature since it allows you to later clear the cache by the given keyword, resulting in the content in question no longer being cached.

For example, if you have a list of items that has been cached (to improve user loading time), you might want to clear this cache if a new item is added to the list. You can do so by clearing the cache by the tags you add to the cache options.

Example 8.3. Adding tags to a cache entry

```
<?php
    $options = new Application_Cache_Options();
    $options->addTag('my custom tag')
            ->addTag('another tag');
?>
```

Clearing the Cache By Tag(s)

The caching on a client web site is controlled by the `Module_Pages_ClientRenderer_Cache` class. You can use this class to clear any cache items that have a particular tag associated with them.

This is achieved using the `ClearByTag()` method. This method accepts any number of arguments, each of which is either a tag or an array of tags.

Example 8.4. Clearing the client web site cache by tags

```
<?php
    Module_Pages_ClientRenderer_Cache::ClearByTag('my custom tag');

    Module_Pages_ClientRenderer_Cache::ClearByTag('my custom tag', 'another tag');

    Module_Pages_ClientRenderer_Cache::ClearByTag(array('my custom tag', 'another tag'));
?>
```