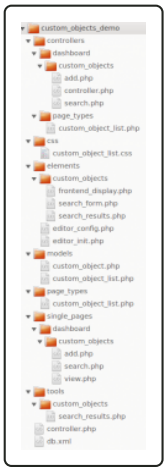
Using custom objects and lists to create concrete5 dashboard pages and front end search pages

I mentioned over the last few posts that I had learned a lot about how custom item lists work in concrete5, here's how to make a dashboard search / add / edit / delete interface for your own custom database rows. Since not everything in concrete5 needs to be a page, I'm also showing you how to make a custom front end filter page for these database items that keeps unique pretty URLs for each item. Sample code is included.

In order to really understand this, I'm assuming that you have knowledge of how the concrete5, including how the package system works, and a fairly decent knowledge of object oriented programming in php. I'm probably not going to be able to make it simple enough for anyone and everyone to understand, but I will try.

Overall Package Structure

It's not the easiest thing to do to create this application, but there are a lot of built in functions and classes in concrete5 that will help make things easier. I guess the first thing to look at is the overall structure of the package. It's a lot of files, but each one is necessary, and each one makes things work in the overall application.



Pages

We have controllers and single pages for a "custom_objects" section in the dashboard. Then we have a controller and a view for a "custom_object_list" page type. A css file for that page type.

Elements

Elements are basically include files that you can pass parameters to. For this, we have editor_config and editor_init which control the display of the WYSIWYG editor for the content section of our custom objects. These are taken from the content block, and will check the home page to find out what theme it is using to apply the themes from typography.css to our dashboard edit form.

We also have a section for all of our custom objects related items. There's a frontend display, used on the custom_object_list page type. A search form, used in the dashboard search / list view. And a search results element which will show the results from that search form on the dashboard search page.

Models

Here we only need two objects. One is the model for saving / getting data for our custom object, the other is responsible for listing and searching through those objects. We'll get into them more later on.

Search

Category Tags

Blog Archive

2014
2013
2012
2011
2010

Featured Post

Getting precision fit for my bike (/blog/getting-precision-fit-for-my-bike/)

I've been waiting to get a frame custom welded for touring, gravel, and possibly some randonneuring. Today I rode out to Freewheel Bike in Eden Prairi…

Read full post » (/blog/getting-precision-fit-for-my-bike/)

Tools

We only have one object here, which is used to display the search results on the dashboard page. It basically gets the results from the request and then includes the search_results element to display them. It's called via ajax by the built in custom search results javascript.

Extra Files

The other two files that we have are a db.xml file to create our database schema, and a controller file to install everything for the application.

What Kind of Objects Are We Creating?

I tried to keep it really simple, just a title and a text area. Since I'm using the WYSIWYG editor that's built in, there are a lot of extra functions in our /models/custom_object.php file that are related to translating the content from edit mode to view mode. This is because concrete5 uses some shorthand notation and IDs to create the links to pages and file downloads and images instead of linking to the file name. I'm not going to go into the details of these here, but the model source code should be enough for you to go from. You'd have to replace all instances of "CustomObject" in the callback functions with the name of your actual object if you modify it.

This is the database schema that we are working with:

```
1. <?xml version="1.0"?>
2. <schema version="0.3">
3.      <table name="customObjects">
4.           <field name="coID" type="I">
5.                <key />
6.                <unsigned />
7.                <autoincrement />
8.           </field>
9.           <field name="title" type="C" size="255">
10.          </field>
11.          <field name="content" type="X2">
12.          </field>
13.     </table>
14. </schema>
```

Sometimes you want something much simpler than a page in a concrete5 website. Typically, the smallest unit that you have is a page, but often it's actually better to just make a form to update a simple database row and then search and filter off of that. The problem with this is that you have a lot of extra stuff when you add a page, sometimes you just want a simple database row but still want a custom url to view that particular database row. By creating the custom_object_list page type and controller, we can override the view function to take a parameter for what custom object we want to display, then use that to change the output of the page to display that particular object. To the end user, it looks like you have a lot of different pages, but really you are just pulling up the item out of the database.

The Custom Object Model

At the base of it, we are making "Custom Objects" - so we need a model to interact with the database for these. There are several methods that it will need. Some are static, some are instance. Here's a list of all the methods:

```
static function getByID($id)
static function getByTitle($title)
public function delete()
public function save($data)
public static function add($data)
public function getTitle()
function getContent()
function getContentEditMode()
public function getCustomObjectID()
```

a whole bunch of functions about translating content.

So every time that we want to add or update or delete one of our database rows, we'll be calling up this object and using one of these methods to do whatever we want to with it. Here's what the whole model looks like. I won't go into exactly what each function does, but they should be pretty self explanatory.

```php
<?php
defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://

class CustomObject extends Object {

	static function getByID($id) {
		$db = Loader::db();
		$data = $db->getRow('SELECT * FROM customObjects WHERE c
		if (!empty (http://www.php.net/empty)($data)) {
			$customObject = new CustomObject();
			$customObject->setPropertiesFromArray($data);
		}
		return (is_a (http://www.php.net/is_a)($customObject, "C
	}

	static function getByTitle($title) {
		$db = Loader::db();
		$data = $db->getRow('SELECT * FROM customObjects WHERE t
		if (!empty (http://www.php.net/empty)($data)) {
			$customObject = new CustomObject();
			$customObject->setPropertiesFromArray($data);
		}
		return (is_a (http://www.php.net/is_a)($customObject, "C
	}

	public function delete() {
		$db = Loader::db();
		$db->execute("DELETE FROM customObjects where coID = ?",
	}

	public function save($data) {
		$db = Loader::db();
		$content = CustomObject::translateTo($data['content']);
		$title = $data['title'];
		$vals = array (http://www.php.net/array)($title, $conten
		$db->query("UPDATE customObjects SET title = ?, content
		$customObject = CustomObject::getByID($this->getCustomOb
		return (is_a (http://www.php.net/is_a)($customObject, "C
	}

	public static function add($data) {
		$db = Loader::db();
		$content = CustomObject::translateTo($data['content']);
		$title = $data['title'];
		$vals = array (http://www.php.net/array)($title, $conten
		$db->query("INSERT INTO customObjects (title, content) V
		$id = $db->_insertID();
		if (intval (http://www.php.net/intval)($id) > 0) {
			return CustomObject::getByID($id);
		} else {
			return false;
		}
	}

	public function getTitle() {
		return $this->title;
	}

	function getContent() {
		$content = $this->translateFrom($this->content);
		return $content;
	}

	public function getCustomObjectID() {
		return intval (http://www.php.net/intval)($this->coID);
	}

	function br2nl($str) {
		$str = str_replace (http://www.php.net/str_replace)("\r\
		$str = str_replace (http://www.php.net/str_replace)("<br
		return $str;
	}

	function getContentEditMode() {
		$content = $this->translateFromEditMode($this->content);
```

```
        return $content;
    }

    public static function replacePagePlaceHolderOnImport($match)
        $cPath = $match[1];
        if ($cPath) {
            $pc = Page::getByPath($cPath);
            return '{CCM:CID_' . $pc->getCollectionID() . '}';
        } else {
            return '{CCM:CID_1}';
        }
    }

    public static function replaceDefineOnImport($match) {
        $define = $match[1];
        if (defined (http://www.php.net/defined)($define)) {
            $r = get_defined_constants (http://www.php.net/get_
            return $r[$define];
        }
    }

    public static function replaceImagePlaceHolderOnImport($match
        $filename = $match[1];
        $db = Loader::db();
        $fID = $db->GetOne('select fintval(ID from FileVersions
        return '{CCM:FID_' . $fID . '}';
    }

    public static function replaceFilePlaceHolderOnImport($match)
        $filename = $match[1];
        $db = Loader::db();
        $fID = $db->GetOne('select fID from FileVersions where f
        return '{CCM:FID_DL_' . $fID . '}';
    }

    function translateFromEditMode($text) {
        // now we add in support for the links

        $text = preg_replace (http://www.php.net/preg_replace)(
                '/{CCM:CID_([0-9]+)}/i', BASE_URL . DIR_REL . '/

        // now we add in support for the files

        $text = preg_replace_callback (http://www.php.net/preg_r
                '/{CCM:FID_([0-9]+)}/i', array (http://www.php.n


        $text = preg_replace_callback (http://www.php.net/preg_r
                '/{CCM:FID_DL_([0-9]+)}/i', array (http://www.ph


        return $text;
    }

    function translateFrom($text) {
        // old stuff. Can remove in a later version.
        $text = str_replace (http://www.php.net/str_replace)('hr
        $text = str_replace (http://www.php.net/str_replace)('sr

        // we have the second one below with the backslash due t
        // 5.1 release. Can remove in a later version.

        $text = preg_replace (http://www.php.net/preg_replace)(
                array (http://www.php.net/array)(
            '/{\[CCM:BASE_URL\]}/i',
            '/{CCM:BASE_URL}/i'), array (http://www.php.net/arra
            BASE_URL . DIR_REL,
            BASE_URL . DIR_REL)
                , $text);

        // now we add in support for the links

        $text = preg_replace_callback (http://www.php.net/preg_r
                '/{CCM:CID_([0-9]+)}/i', array (http://www.php.n

        $text = preg_replace_callback (http://www.php.net/preg_r
                '/<img [^>]*src\s*=\s*"{CCM:FID_([0-9]+)}"[^>]*>
```

```
        // now we add in support for the files that we view inli
        $text = preg_replace_callback (http://www.php.net/preg_r
                '/{CCM:FID_([0-9]+)}/i', array (http://www.php.n

        // now files we download

        $text = preg_replace_callback (http://www.php.net/preg_r
                '/{CCM:FID_DL_([0-9]+)}/i', array (http://www.ph

        return $text;
    }

    private function replaceFileID($match) {
        $fID = $match[1];
        if ($fID > 0) {
            $path = File (http://www.php.net/file)::getRelative
            return $path;
        }
    }

    private function replaceImageID($match) {
        $fID = $match[1];
        if ($fID > 0) {
            preg_match (http://www.php.net/preg_match)('/width\
            preg_match (http://www.php.net/preg_match)('/height
            $file = File (http://www.php.net/file)::getByID($fI
            if (is_object (http://www.php.net/is_object)($file)
                $imgHelper = Loader::helper('image');
                $maxWidth = ($matchWidth[1]) ? $matchWidth[1]
                $maxHeight = ($matchHeight[1]) ? $matchHeight[
                if ($file->getAttribute('width') > $maxWidth |
                    $thumb = $imgHelper->getThumbnail($file,
                    return preg_replace (http://www.php.net/p
                }
            }
            return $match[0];
        }
    }

    private function replaceDownloadFileID($match) {
        $fID = $match[1];
        if ($fID > 0) {
            $c = Page::getCurrentPage();
            if (is_object (http://www.php.net/is_object)($c)) {
                return View::url('/download_file', 'view', $fI
            } else {
                return View::url('/download_file', 'view', $fI
            }
        }
    }

    private function replaceDownloadFileIDInEditMode($match) {
        $fID = $match[1];
        if ($fID > 0) {
            return View::url('/download_file', 'view', $fID);
        }
    }

    private function replaceFileIDInEditMode($match) {
        $fID = $match[1];
        return View::url('/download_file', 'view_inline', $fID);
    }

    private function replaceCollectionID($match) {
        $cID = $match[1];
        if ($cID > 0) {
            $c = Page::getByID($cID, 'APPROVED');
            return Loader::helper("navigation")->getLinkToColle
        }
    }

    public static function translateTo($text) {
        // keep links valid
        $url1 = str_replace (http://www.php.net/str_replace)('/'
        $url2 = str_replace (http://www.php.net/str_replace)('/'
        $url3 = View::url('/download_file', 'view_inline');
```

```
31.          $url3 = str_replace (http://www.php.net/str_replace)('/'
32.          $url3 = str_replace (http://www.php.net/str_replace)('-'
33.          $url4 = View::url('/download_file', 'view');
34.          $url4 = str_replace (http://www.php.net/str_replace)('/'
35.          $url4 = str_replace (http://www.php.net/str_replace)('-'
36.          $text = preg_replace (http://www.php.net/preg_replace)(
37.                  array (http://www.php.net/array)(
38.              '/' . $url1 . '\?cID=([0-9]+)/i',
39.              '/' . $url3 . '([0-9]+)\//i',
40.              '/' . $url4 . '([0-9]+)\//i',
41.              '/' . $url2 . '/i'), array (http://www.php.net/array
42.              '{CCM:CID_\\1}',
43.              '{CCM:FID_\\1}',
44.              '{CCM:FID_DL_\\1}',
45.              '{CCM:BASE_URL}')
46.                  , $text);
47.          return $text;
48.      }
```

The Custom Object List

The next thing that forms the core of our application is the Custom Object List. This extends the concrete5 built in class DatabaseItemList, which gives you a lot of really great features. It's kind of hard to explain everything that you can do with this class, but I'm going to try. Basically it's a system for querying the database, and presenting it back as a paginated array. There are methods for sorting and filtering the list, complete with methods to make links that will sort and call for the sort selected class to style your sortable search column headers. You can make the database queries as complex as you need them to be, in this case I'm keeping it really simple to show the very basics.

Base Paramaters

We start out with the base values for 3 parameters.

```
1. private $queryCreated;
2. protected $autoSortColumns = array (http://www.php.net/array)("tit
3. protected $itemsPerPage = 10;
```

The thing that really matters here is the $autoSortColumns - if you don't define your sort columns here then sorting on the front end won't work at all. If you are using an alias in your base query, then this will be the name of the alias. I think there might be ways of setting this but I'm not sure what they are. If you look at the way that search results work in Page Search or User Search in the dashboard, you can add extra search fields and extra results columns with a few clicks. I'm not getting into how to do that, here I'm sticking with a hard coded results list and search form. But if you want to do something more complicated like those pages, look at the files in the core that correspond with the ones in this sample package and hopefully it makes sense to you how to expand things.

Setting the Base Query

The next thing we want to look at is setting our base query. This is pretty key to what you get back in your search results. For this application, it's pretty simple:

```
1. protected function setBaseQuery() {
2.         $this->setQuery('SELECT * FROM customObjects');
3. }
4.
5. public function createQuery() {
6.         if (!$this->queryCreated) {
7.                 $this->setBaseQuery();
8.                 $this->queryCreated = 1;
9.         }
10. }
```

This can get a lot more complicated. Here's an example from another package I've been working on:

```
1.  protected function setBaseQuery($extra = "") {
2.          $this->setQuery("SELECT
3.                  cmCommunities.cmID,
4.                  csia.ak_cm_community_name AS community_name,
5.                  u.uID AS uID,
6.                  u.uName AS user_name,
7.                  csia.ak_cm_community_state AS state,
8.                  csia.ak_cm_community_zip_code AS zip_code,
9.                  csia.ak_cm_amenity_titles,
10.                 csia.ak_cm_living_option_titles"
11.                 . $extra .
12.                 " FROM cmCommunities
13.                 LEFT JOIN Collections c ON cmCommunities.cID = c.c
14.                 LEFT JOIN CollectionVersions cv ON cv.cID = c.cID
15.                 LEFT JOIN CollectionSearchIndexAttributes csia ON
16.                 LEFT JOIN Users AS u ON cmCommunities.uID = u.uID
17.                 LEFT JOIN UserSearchIndexAttributes AS usia ON u.u
18.                 LEFT JOIN cmCommunitiesLivingOptionsPages AS clop
```

In the second one, I have a form for creating my objects that's a lot more complicated. That form has options for what page type to add, what parent page to add the page under, and what user to associate with the page. This is all tied through the secondary table. Instead of having to pull up a whole page list with all the complicated functions, we can store the cID of the page in our custom object table and then join with that to the CollectionSearchIndexAttributes table and use that to filter by attributes even though we don't have attributes associated with our custom object. You can also join to the UserSearchIndexAttributes table to get attribues from there, or the FileSearchIndexAttributes table.

The point here is that you can set up your base query to be as complicated as you actually need it to be. Creating your filters becomes a little more compicated. The main difference when you are joining across many tables is that you need to preface your filter calls with the alias of the table you want to filter on. Just using the aliased name of the column won't work. In the autosort list and front-end search functions, the aliased name is what you want to use. But in filter, you must use the full table alias and column name. I'm not sure why that is. It makes some stuff not really work, I tried to set up some base queries that used CONCAT() in the SQL with an alias, then wanted to filter on that alias and it didnt't work.

Getting Results

In order to really make our list useful, we want to override the get method of the parent. If we didn't, we would still get an array of database rows that match our criteria, but we wouldn't have a list of actual objects that we want to interact with. So we will be calling the parent get function to get our database row / array, then looping over that to create our actual Custom Objects and return that instead. You could decide to do this differently - maybe you want to return just an array and then use ids in that array to create your objects. I think that creating objects and returning them is simpler and cleaner. If you need to get page or user objects from your custom object, it's a simple matter to create a getPageObject() or a getUserObject() method to pull in the actual object you need. I guess that's one of the nice things about concrete5 - you could do it however you want to.

Here's my get method:

```
1. public function get($itemsToGet = 0, $offset = 0) {
2.          Loader::model("custom_object", "custom_objects_demo");
3.          $customObjectsList = array (http://www.php.net/array)();
4.          $this->createQuery();
5.          $r = parent::get($itemsToGet, $offset);
6.          foreach ($r as $row) {
7.                  $customObject = CustomObject::getByID($row['coID']
8.                  $customObjectsList[] = $customObject;
9.          }
10.         return $customObjectsList;
11. }
12.
13. public function getTotal() {
14.         $this->createQuery();
15.         return parent::getTotal();
16. }
```

Getting All The Objects

For the application we're building, we have one point where we need to get all the objects for the front end list. Rather than just resetting any query and calling get with no parameters, we're making a static method to get it a bit cleaner and easier.

```
1. public static function getAllCustomObjects() {
2.         Loader::model("custom_object", "custom_objects_demo");
3.         $customObjectList = array (http://www.php.net/array)();
4.         $customObjectList = new CustomObjectList();
5.         $customObjectList->createQuery();
6.         $customObjects = $customObjectList->get();
7.         return $customObjects;
8. }
```

Filtering Results

There's a whole lot that you can do with the filter function, actually. Basically, it allows you to add to the WHERE statements in your sql query. In the item list, this is what the input looks like public function filter($column, $value, $comparison = '=')

If you pass in false as the first parameter, then it's just anything want for the $value of the filter. Leaving the comparison empty means that it won't be used. So unless you are actually using the $column value, you can just pass in false, then your filter and be good. Here are a coupe of examples of filtering:

```
1. public function filterByTitle($title){
2.         $db = Loader::db();
3.         $title = $db->quote("%" . $title . "%");
4.         $this->filter(false, '(title LIKE ' . $title . ')');
5. }
6.
7. public function filterByCommunityName($name) {
8.         $db = Loader::db();
9.         $name = $db->quote("%" . $name . "%");
10.        $this->filter(false, '(csia.ak_cm_community_name LIKE ' .
11. }
```

There's probably more that could be said about the DatabaseItemList - it's really the core of a whole lot of how concrete5 does things. Objects, and lists of objects.

Anyway, here's what our whole custom object list looks like:

```
<?php
defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://

/**
 *
 * A filtered list of custom objects
 * @package Item List Demo
 *
 */
class CustomObjectList extends DatabaseItemList {

        private $queryCreated;
        protected $autoSortColumns = array (http://www.php.net/arr
        protected $itemsPerPage = 10;

        protected function setBaseQuery() {
                $this->setQuery('SELECT * FROM customObjects');
```

Dashboard Pages

The dashboard I've set up I think is pretty solid, but might be better set up. What I do is make a search / results page, and then an add / edit page. It may be better to have a separate page for each. I didn't like some of how it makes the URLs in the dashboard semantically. I guess that's really a kind of stupid thing to think about, but I do. I also like keeping code all in one place, so it's kind of OK.

Redirect Page

First off, we have to create our controller for the main 'view' for the custom objects page. In 5.5 this link is clickable, in 5.6 it's not. Either way, we want it to just redirect to the search page. This is what that looks like:

```
defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://
class DashboardCustomObjectsController extends Controller {
        public function __construct() {
                $this->redirect('/dashboard/custom_objects/search'
        }
}
```

Search Page

The search page shows lists of custom objects, then the results of saving / editing custom objects on the 'add' page. By setting up our views and wrapper IDs and class names you can get a fully functional ajax enabled search interface that's managed by the concrete5 core, with very little need for you to code functionality. You will still need to program some stuff, but a lot of the core functionality is basically built in.

As with most concrete5 single pages, it's just a controller and a view, calling on our Custom Object List and Custom Object to get it's data. The main thing here is looking at the view function. There's one line there that makes *everything* work:

```
1. <?php
2.
3. defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://
4.
5. class DashboardCustomObjectsSearchController extends Controller {
6.
7.      public function on_start() {
8.
9.           Loader::model('custom_object', 'custom_objects_demo');
10.          Loader::model('custom_object_list', 'custom_objects_demo
11.          $this->set('form', Loader::helper('form'));
12.          $this->set('valt', Loader::helper('validation/token'));
13.          $this->set('valc', Loader::helper('concrete/validation')
14.          $this->set('ih', Loader::helper('concrete/interface'));
15.          $this->set('av', Loader::helper('concrete/avatar'));
16.          $this->set('dtt', Loader::helper('form/date_time'));
17.
18.          $this->error = Loader::helper('validation/error');
```

The addHeaderItem with the ccm_setupAdvancedSearch call is what makes the whole search interface work. The value passed into the function is used to determine what html DOM elements are used in the search. On the front end, it's a div with a class of ccm-pane-options that either holds the buttons when displaying results from a Custom Object update, or the search form when you are in list view. Then an area that gets switched out below in search view is swapped out via ajax from the form fields in the search form with the contents of the "/elements/custom_objects/search_results.php" when you submit the form.

The other part of what makes this work is the function getRequestedSearchResults() on the controller. This can be called remotely, which means that you can get a list from the page based on whatever request parameters you send via ajax. You call it from inside of the tools file for the custom object, "/tools/custom_objects/search_results.php".

```
1. $cnt = Loader::controller('/dashboard/custom_objects/search');
2. $customObjectsList = $cnt->getRequestedSearchResults();
3.
4. $customObjects = $customObjectsList->getPage();
5.
6. Loader::packageElement('custom_objects/search_results', 'custom_ob
```

Search Results Element

The dashboard search results are pulled in from a package element, both in the initial view and then when loaded in from the built in search interface. So both need the same root element to allow the system to hook onto it and swap it out, and that root element has to match what you put in the addHeaderItem() call. So if it's 'custom-objects' there, then your ID would be ccm-custom-objects-search-results - if it was some other object that you wanted to initiate, it would be ccm-some-other-object-search-results for the ID that wraps your element.

Here's what the dashboard search results element looks like:

```
<?php
defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://
$ih = Loader::helper('concrete/interface');
$form = Loader::helper('form');
?>

<div id="ccm-custom-objects-search-results">

        <div class="ccm-pane-body">
                <div style="margin-bottom: 10px">
                        <?php print $ih->button(t('Add Custom Obje
                        <div class="clearfix"></div>
                </div>
                <div id="ccm-list-wrapper">
                        <a name="ccm-custom-object-list-wrapper-an
                        <?php
                        $txt = Loader::helper('text');
                        $title = $_REQUEST['title'];
```

First off we're making a link / button to add a new Custom Object, and displaying that above our list. The list is a simple table. We grab some of the request variables and get the link back to our search results tools file for highlighting the search results and making our title column sortable by clicking on it. The DatabaseItemList that we are extending has two custom methods in it, one is getSearchResultsClass('column') and the other is getSortByURL('column', 'order', "base_url") - these will only work if you've set the column as sortable in the CustomObjectsList controller. The text helper has a method for highlightSearch - pass in the value you want to highlight $customObject->getTitle() and then the value you want to check for which we're grabbing from the $_REQUEST array. The last part of our search results list are links to the Add page to edit or delete each individual object.

The Search Form

I'm keeping this pretty simple, but you could get as complicated as you want with this. The form submits to our tools file, so we use the urls helper to get the url we're going to submit to. I don't know if it makes a difference to use post or get, but all the samples I've seen use get so I kept with that. We have a title text field, and a drop down for the number of results. The image underneath our submit button has our custom-objects in the id - ccm-**custom-objects**-search-loading. This means that the built in search javascript will show it as the request is being submitted.

```
<?php  defined (http://www.php.net/defined)('C5_EXECUTE') or die (
<?php  $form = Loader::helper('form');
$url = Loader::helper('concrete/urls');
$urlSearchAction = $url->getToolsURL('custom_objects/search_result
?>

        <form method="get" id="ccm-custom-objects-advanced-search"
        <input type="hidden" name="search" value="1" />

        <div class="ccm-pane-options-permanent-search">

                <div style="width: 160px; margin-left: 20px; float
                <?php echo $form->label('title', t('Title'))?>
                        <?php echo $form->text('title', $_REQUEST[
                </div>
                <div  style="width: 100px; margin-left: 20px; floa
                <?php echo $form->label('numResults', t('# Per Pag
                        <?php echo $form->select('numResults', arr
```

Putting it all together into a page

The search page is also used when results need to displayed from saving a Custom Object. Those pages will redirect here, and it will then show the details of the saved Object and some links to edit it again, delete it, or go back to list view.

Here's how the overall view shows these different things:

```
<?php
defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://
$ih = Loader::helper('concrete/interface');
?>


<?php if ($_REQUEST['custom_object_created'] || $_REQUEST['custom_
<?php echo Loader::helper('concrete/dashboard')->getDashboardPaneH
        <div class="ccm-pane-options">
                <?php print $ih->button(t('Edit Again'), $this->ur
                <?php print $ih->button(t('Delete'), $this->url('/
                <?php print $ih->button(t('Back to List'), $this->
        </div>
        <div class="ccm-pane-body">
                <dl>
                        <dt><?php echo t("Title");?></dt>
                        <dd><?php echo $customObject->getTitle();?
                        <dt><?php echo t("Content");?></dt>
```

I could probably make this a bit cleaner, setting the custom_object_created and custom_object_updated in the controller for the view, if it was a front-end page I might do that. For a dashboard page I don't think it matters as much. At any rate, if either of those are set, then we should have a Custom Object to show the details of. We're showing this 'success' page because we want to show the success message above the dashboard panel. This message doesn't go away after a bit of time, so instead of having it there for a long time while you search and sort the objects we show the details of the object, with links to delete or edit it, and then a link back to the search list.

If those aren't set, then we're displaying the search form and results elements. Because of our header javascript, the search form will be submitted via ajax to our tools file, and the results will replace the outer div. The naming is pretty important here. For the search form, it needs to be an id of "ccm-**your-object**-advanced-search" and for the results it's "ccm-**your-object**-search-results" where "your-object" is the parameter you passed into the header javascript. ccm_setupAdvancedSearch('**your-object**');

The Add / Edit Page

We're kind of combining both functions here, you might want to split things out a bit and have different dashboard pages for each. It's really up to you, I think. There are several methods on the controller that edit or delete objects. Adding is assumed if no other method is being called. If there are post variables for 'create' or 'update' in the view then we're validating the form, setting errors, or if it's good we create or update our CustomObject and redirect to the dashboard search page and display our success message.

It's worth noting the difference in how update and add work. In the create section, we're calling a static function CustomObject::add($data) which will return either the custom object or false. In the update section, we're again calling a static function, CustomObject::getByID($coID) which will return the same way. If we have a valid Custom Object, we then call the instance method $customObject->save($data). For the save method it also returns the same way, with the updated object. It probably should be verifying that it's a valid object returned, but it just redirects.

The delete function is set up in two different functions. The first one is 'confirm_delete' which will get our Custom Object and assign it to the view, which then links back to the actual 'delete' function on the controller. By using the validation/token helper we can assure that our requests are coming from the confirm_delete view, and keep people from just hitting /dashboard/custom_objects/add/delete/coID and deleting objects without validation or confirmation.

Here's the controller:

```php
<?php
defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://

class DashboardCustomObjectsAddController extends Controller {

     public function on_start() {
          Loader::model('custom_object', 'custom_objects_demo');
          Loader::model('custom_object_list', 'custom_objects_demo
          $this->set('form', Loader::helper('form'));
          $this->set('valt', Loader::helper('validation/token'));
          $this->set('valc', Loader::helper('concrete/validation')
          $this->set('ih', Loader::helper('concrete/interface'));

          $this->error = Loader::helper('validation/error');
     }

     public function view() {

          $vals = Loader::helper('validation/strings');
          $valt = Loader::helper('validation/token');
          $valc = Loader::helper('concrete/validation');

          if ($_POST['create']) {

               $title = $_POST['title'];
               if (!$vals->notempty($title)) {
                    $this->error->add(t('Please include a title.')
               }

               $content = $_POST['content'];
               if (!$vals->notempty($content)) {
                    $this->error->add(t('Please include content.')
               }

               if (!$valt->validate('create_custom_object')) {
                    $this->error->add($valt->getErrorMessage());
               }

               if (!$this->error->has()) {
                    $data = array (http://www.php.net/array)(
                        'title' => $title,
                        'content' => $content);
                    $customObject = CustomObject::add($data);
                    if (is_a (http://www.php.net/is_a)($customObje
                         $this->redirect('/dashboard/custom_object
                    } else {
                         $this->error->add(t('An error occurred wh
                         $this->set('error', $this->error);
                    }
               } else {
                    $this->set('error', $this->error);
               }
          }

          if ($_POST['update']) {

               $coID = $_POST['coID'];
               if (!intval (http://www.php.net/intval)($coID) > 0)
                    $this->error->add(t('Invalid Custom Object ID.
               }

               $title = $_POST['title'];
               if (!$vals->notempty($title)) {
                    $this->error->add(t('Please include a title.')
               }

               $content = $_POST['content'];
               if (!$vals->notempty($content)) {
                    $this->error->add(t('Please include content.')
               }

               if (!$valt->validate('update_custom_object')) {
                    $this->error->add($valt->getErrorMessage());
               }

```

```
                if (!$this->error->has()) {
                    $customObject = customObject::getByID($coID);
                    $data = array (http://www.php.net/array)(
                        'title' => $title,
                        'content' => $content);
                    if (is_a (http://www.php.net/is_a)($customObje
                         $customObject->save($data);
                         $this->redirect('/dashboard/custom_object
                    } else {
                         $this->error->add(t('An error occurred wh
                         $this->set('error', $this->error);
                    }
                } else {
                    $this->set('error', $this->error);
                }
            }
      }

```

The view here is a bit convoluted. Since it's being used for adding / editing and deleting, we have to change the output quite a bit based on the task being done. We also have to get our data from two different sources - if we are editing and it's our first time displaying the page, then we want to pull it in from the CustomObject that the controller passed in. But if we've submitted the form and there were errors, then we want to be grabbing from the $_POST array so that we have the most recent data and not the old data from the object we're editing. So we set a variable $useCustomObjectDetails when showing the edit form to change where we get the information from. There are a couple of hidden fields at the end of the edit / add form that help keep track of what we are doing, and the validation token changes according to it being an update or an addition.

I'm sure there are probably better ways to do this, but it seems pretty solid to me and I haven't had any real problems with it, so I'll keep doing it like this. For more complex forms this can get kind of overwhelming, but it does scale well.

One thing worth noting is that the form is using the full tinyMCE WYSIWYG (what you see is what you get) editor. This is pretty easy to include in concrete5. There's an element that you can include from the core to to create one. In our package we're using a package element instead of the built in one. The package element is based off the include files from the Content block instead. That one checks the current page for a theme and then uses the typography.css file from that theme to style the content in the editor. Since we are in the dashboard, we are just grabbing the home page and getting the theme page from it. Simply loading the package element editor_init and giving our text area the class ccm-advanced-editor will turn it into a rich text editor. I'm not going to include the elements in this page, but you can check out the files in the attached package if you're interested.

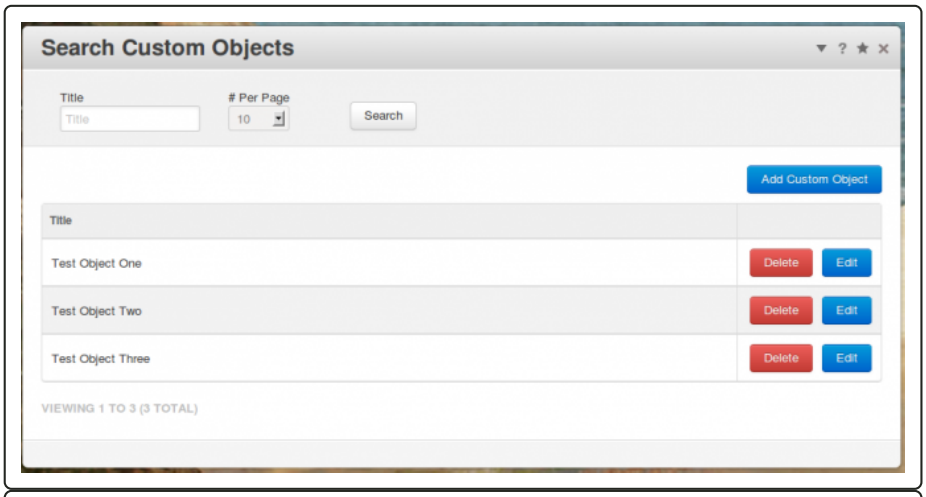
Here's the view:

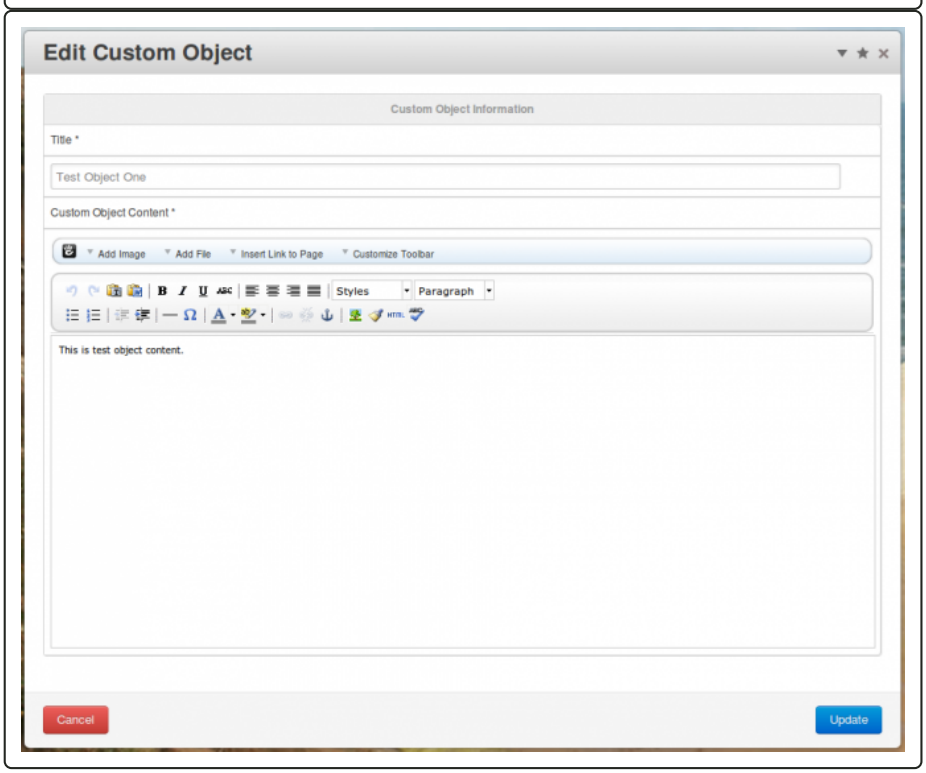
```
<?php
defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://

$th = Loader::helper('text');

if (is_a (http://www.php.net/is_a)($customObject, "CustomObject")
     $useCustomObjectDetails = 1;
     $isUpdate = 1;
}
if (intval (http://www.php.net/intval)($_POST['isUpdate']) > 0) {
     $isUpdate = 1;
}
if (is_a (http://www.php.net/is_a)($customObject, "CustomObject")
     $confirmDelete = 1;
}
?>
<?php
if ($confirmDelete) {
     ?>
     <?php echo Loader::helper('concrete/dashboard')->getDashboard
     <form method="post" enctype="multipart/form-data" id="cm-dele
          <?php echo $valt->output('delete_custom_object'); ?>
          <div class="ccm-pane-body">
               <h2><?php echo t("Do you really want to delete this
               <dl>
                    <dt><?php echo t("Title"); ?></dt>
                    <dd><?php echo $customObject->getTitle(); ?></
                    <dt><?php echo t("Content"); ?></dt>
                    <dd><?php echo $customObject->getContent(); ?>
               </dl>
          </div>
          <div class="ccm-pane-footer">
               <div class="ccm-buttons">
                    <input type="hidden" name="do_delete" value="1
                    <input type="hidden" name="coID" value="<?php
                    <?php print $ih->button(t('Cancel'), $this->ur
                    <?php print $ih->submit(t('Delete Permanently'
               </div>
          </div>
     <?php } else {
          ?>
          <?php if ($isUpdate) { ?>
               <?php echo Loader::helper('concrete/dashboard')->ge
          <?php } else { ?>
               <?php echo Loader::helper('concrete/dashboard')->ge
          <?php } ?>
          <form method="post" enctype="multipart/form-data" id="cm
               <div class="ccm-pane-body">
                    <?php
                    if ($useCustomObjectsDetails || $isUpdate) {
                         echo $valt->output('update_custom_object'
                         ?>
                    <?php } else {
                         echo $valt->output('create_custom_object'
                         ?>
                    <?php } ?>

                    <table border="0" cellspacing="0" cellpadding=
                         <thead>
                              <tr>
                                   <th><?php echo t('Custom Object
                              </tr>
                         </thead>
                         <tbody>
                              <tr>
                                   <td><?php echo t('Title') ?> <s
                              </tr>
                              <tr>
                                   <?php if ($useCustomObjectDetai
                                        <td><input type="text" id=
                                   <?php } else { ?>
                                        <td><input type="text" id=
                                   <?php } ?>
                              </tr>
                              <tr>
```

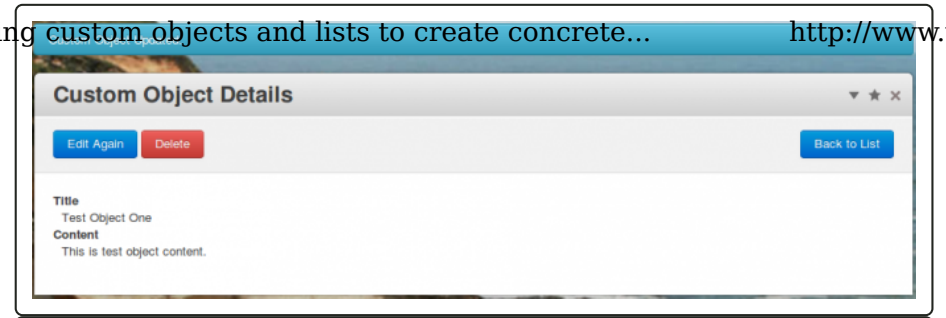
```
                    <td><?php echo t('Custom Object
                </tr>
                <tr>
                    <?php if ($useCustomObject
                    <td>
                    <?php Loader::packageEleme
                        <textarea
                            id="ccm-content-
                            class="advancedE
                            name="content"
                            style="width: 58
                    <?php echo ($customObject-
                        </textarea>
                    </td>
                    <?php } else { ?>
                    <td>
                    <?php Loader::packageEleme
                        <textarea
```

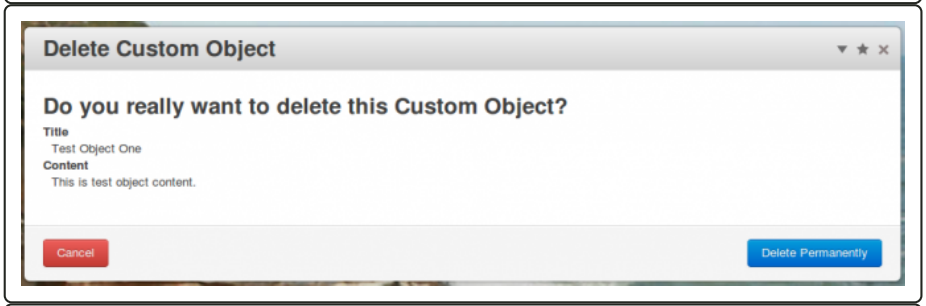
Screenshots of the Dashboard Pages

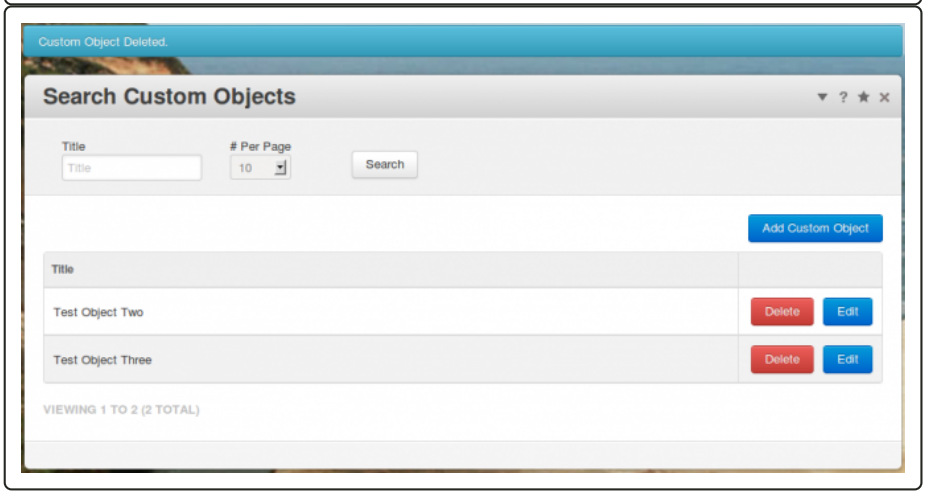
These are for 5.5, but the same code works on 5.6 it just looks a bit different.











Moving to the Front End

Now that we have our dashboard interface working to add / edit and update our Custom Objects, we need to move on to making the front end work. There are several ways that we could approach this - we could make a "Custom Objects List" block and output them. What I want to do for this one is actually make a page type which displays all of the object titles in a list on the sidebar, then clicking the title swaps out the content in the main area of the page. I also want this to update the url with the title of the Custom Object converted to lowercase-with-dashes so that people can bookmark links. Getting real pretty urls like this is pretty easy from a page type controller, but much harder to do with blocks.

One thing that's worth noting - because of the way that the View::url() function works, you will get index.php in your links, even if you have pretty urls turned on and your .htaccess file working. To avoid this, you can put define("URL_REWRITING_ALL", true); in your site.php file and it will go away.

The Custom Object List Page Type Controller

The first thing that we need to do is create a controller for our Page Type. It only needs two methods, one for on_start() that loads up our list and model, then adds a css file to the header to format the output. If you are making a custom page type within a theme, you probably don't need to add extra css.

The next part is really the key to getting it to look like these are different 'pages' in the URL. Normally the view() function of a Page Type doesn't take any arguments, but we're adding one here, which is the escaped title of the custom object we want to display. By giving it a blank default argument we don't have to have it in the URL to begin with. We first get the full list of

custom objects, then check to see if we have a $customObject argument or not. If there is no argument, then we set the 'selected' object to be the first item in the array. Otherwise, we unhandle the argument using the text helper, then call the static method CustomObject::getByTitle($customObjectTitle) and set that as the selected Custom Object.

We could have created a different method, perhaps view_object($object_handle) and then linked to that, and call the view() function after. I think it's a little cleaner in the URLs to just have it right off the parent page handle, which is why I'm doing it this way.

Here's the controller:

```
1. <?php
2.
3. defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://
4.
5. class CustomObjectListPageTypeController extends Controller {
6.
7.      public function on_start() {
8.           Loader::model('custom_object', 'custom_objects_demo');
9.           Loader::model('custom_object_list', 'custom_objects_demo
10.          $html = Loader::helper('html');
11.          $this->addHeaderItem($html->css('custom_object_list.css'
12.     }
13.
14.     public function view($customObject = "") {
15.          $txt = Loader::helper('text');
16.          $allCustomObjects = CustomObjectList::getAllCustomObject
17.          $this->set('allCustomObjects', $allCustomObjects);
18.
```

The Page Type View

Now that we have a controller, we need to create a view for it. This part is actually pretty simple. First off we're going to loop over the list of all the Custom Objects, outputting the title and a link for each one pointing back at the controller's view method. Then we use another package element to output the content of the selected custom object in the main area of the page. Technically you don't need to use an element here, you could just put your code directly in the page. Conversely, it might be better to do the sidebar as a package element as well. Really you could do it a lot of different ways.

Here's the view. Since this is a page type within a package, it will be displayed within the 'view.php' file in your theme. To make it a full page in your theme, you would want to make a file custom_object_list.php in your theme folder and style that however you want, copying the code from the package page type to where it should go in your theme.

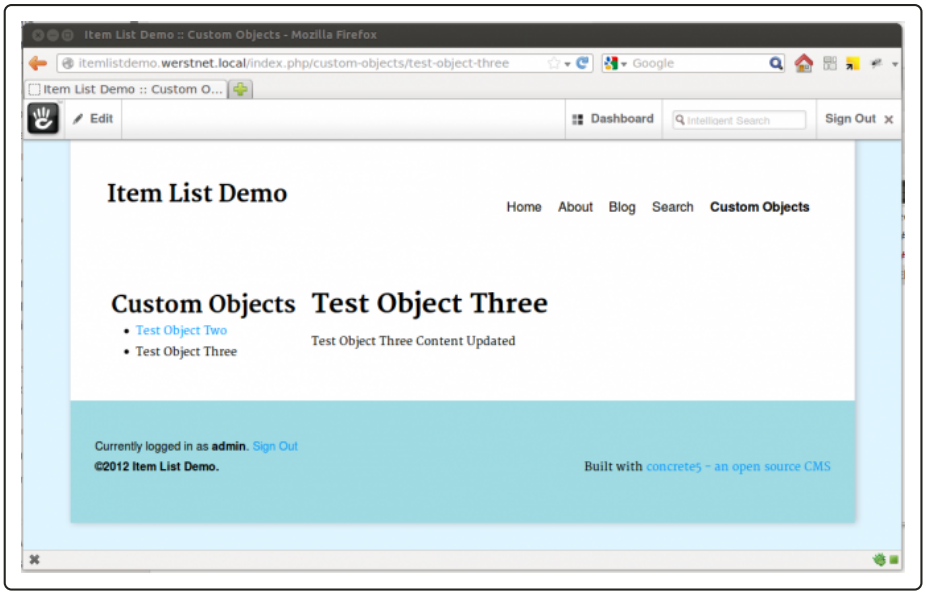
```
<?php
Loader::model("custom_object", "custom_objects_demo");
$txt = Loader::helper("text");
$c = Page::getCurrentPage();
$filter_custom_objects_url = BASE_URL . View::url($c->getCollectio
?>
<div class="left">
     <h2><?php echo t("Custom Objects"); ?></h2>
     <ul id="custom-objects-list">
          <?php
          $selectedCustomObjectID = intval (http://www.php.net/int
          foreach ($allCustomObjects as $customObject) {
               $title = $customObject->getTitle();
               $handle = $txt->sanitizeFileSystem($title);
               $coID = $customObject->getCustomObjectID();
               if ($selectedCustomObjectID == $coID) {
                    ?>
                    <li class="selected">
```

The Frontend Display Package Element

This is really pretty basic, but here it is anyway:

```
<?php
defined (http://www.php.net/defined)('C5_EXECUTE') or die (http://
$title = $customObject->getTitle();
$content = $customObject->getContent();
?>
<h1><?php echo $title;?></h1>
<div class="content">
        <?php echo $content;?>
</div>
```

Frontend Screenshot



Wrapping It Up

I hope this helps make things a bit clearer to people on how to use this built in functionality in concrete5. It seems like a lot of people know how to modify a page list a bit or create a custom block or even do dashboard pages. But creating powerful CRUD interfaces for custom objects isn't something you see very much. I think a lot of people default to doing everything with pages since that makes nice urls easy, and you can add in attributes pretty easily to extend their content. Sometimes, though a page is just too much overhead. If you just have a simple database item that you want to display, using a method like I've outlined here can make your pages work a lot faster.

One of the things I hear a fair amount about concrete5 is that it's kind of a 'toy' CMS - that it's so pretty and easy to use for the site owner that it ***can't*** have powerful tools for developers.

Hopefully this writeup helps dispel some of that myth. You can create nearly anything that you want with it.

Actually extending this method out a bit is possible too. One thing that I want to learn to do is to create attribute categories for the custom object models. That would let you do pretty much anything that you want. You can already get attribute filtering capability even with this though - by storing a cID or uID or fID in your custom object database, you can then join to the search index attributes table for that item type. Just put a file, user or page picker on your front end form and you can get the ID. Or you can create a page on save, then store the cID from that. Once you have one of these objects, you can start adding attributes to your edit forms and apply them to your object when you save. You can also use other lists and objects to create 'parent' objects that contain lists of secondary objects and apply the filter methods through your table joins.

But all of that will have to be the subject of another tutorial. This one is already long enough, I think. :)

If you want to try out the files on your server or get a closer look at the code, you can download it here (/download_file/view/230/386/) or browse it on github (https://github.com/herent/Custom-Objects-in-concrete5-Demo). It's in package format, so unzipping it in your packages directory should make it available to install from the dashboard. Checking out the github repo should be done into a folder called custom_object_demo - it doesn't have the parent folder included in the repo. It should work for 5.5 and 5.6, but not earlier versions of concrete5. Those used pretty much the same overall structure, but the dashboard styling was much different. I've also set up a demo site where you can try things out too, http://customobjects.werstnet.com (http://customobjects.werstnet.com) - the user name ic coAdmin and the pass is abc123.

Please leave any questions or comments below - I'll try to answer anything that I can, but I don't know everything.

If you find this useful, please consider sending me a tip through paypal (https://www.paypal.com/cgi-bin/webscr?cmd=_s-xclick&hosted_button_id=C4EJLXXKGP636) - I don't really make anything off of setting up demo files and tutorials. I'd really appreciate it, and would be more likely to do these more often if it was actually bringing in some income. ;)

g+1

26 Comments Werstnet Login

Sort by Newest Share Favorite ★

Join the discussion…

mm • 22 days ago
mmm

• Reply • Share ›

Nicolas Alberto • 2 months ago
Hi and thanks for this great tutorial. I love concrete5 and my level of understanding of its inner workings is by far below yours.

It would be great to have a generic object type rather than the page type.
That way a normal user could create its own custom list without having to have its sitemap filled up with potentially thousands of records.

Concrete5 is so intuitive and yet this is not intuitive at all.

As a dum user, used to the very intuitive concrete 5 interface, I would expect something like this.
Custom List Menu > Create new custom list (add a name)
then add your attributes, the same way you do it for pages
There could then be custom list viewer block to be used on pages.

This is my 5 cents worth contribution and will probably go nowhere but this may be the right place to suggest it. Just because after searching for a while for this I ended up here. :-)

What do you think?

• Reply • Share ›

hereNT Mod ➚ Nicolas Alberto • 2 months ago
Well, that's kind of what this gets at. There's low level support for it that you can code, but not really anything with the gui.

Mike Lay was working on a dashboard interface for making objects that could have attributes attached, not sure how far along things got but the git repo is here:
https://github.com/mkly/Data

I don't know how much of this type of architecture is actually around in 5.7, but it's entirely possible that all of this is going to be moot really shortly.

• Reply • Share ›

Guest • 4 months ago
Hey, is this still relevant or did they change everything in the latest Concrete5? Thanks!

• Reply • Share ›

akodde • a year ago
Ok, this is handy, thanks for sharing. Just wondering, how'd people use this without having to rename filenames, table names, object names, etc.? I took a quick look at the object in /models and figured you'd use a static variable for the table name. Also you'd gain flexibility by using $this and self:: more often

Site © Jeremy Werst 2010 - 2014

2219 31st Ave South #1, Minneapolis, MN 55406
jeremy@werstnet.com (mailto:jeremy@werstnet.com)
612-807-0102 (tel:6128070102)