Poplar ProductivityWare

Creating a Compound Field Module for CCK in Drupal 6.x

By: Jennifer Hodgdon

When setting up a site using the *Drupal (http://drupal.org)* content management system, you'll often find that you need to define content types that have fields attached to them, beyond the default Title and Body. In Drupal version 7.x and later, this field functionality is planned to be in the core distribution of Drupal, but in 6.x and previous versions of Drupal, it is provided by the contributed *Content Construction Kit (CCK) module (http://drupal.org/project/cck)* and the many associated modules that provide additional field types for CCK.

For some sites, you may find that you need to define fields that have multiple values; for example, you might want to place several images along the right side of a page (each one being an Image field). That is no problem, in itself -- CCK allows you to set any field to accept multiple values, and in version 2 of CCK, you can easily re-order, delete, and add new items to the list using a very nice AJAX interface.

But what if you need to associate, for instance, a caption and a taxonomy term with each of those images? Or in other words, what if you want to **add fields to your content type as a group**? With the current CCK version, you have **several choices of how to do that**:

1. **Several separate fields:** Create a multiple-valued field for the image, another for the caption, and another for the taxonomy term, and tell the people editing content on the site to try to keep them synchronized. This is not really workable in general -- users would need to scroll up and down the content entry screen to do their data entry, and inevitably someone will make a mistake and you'll have a caption or term associated with the wrong image. Also, for this specific case, Content Taxonomy as a multiple-valued field just gives you one multi-select list, so there is no way to choose a given taxonomy term more than once, or indicate the order (term 1 goes with image 1, etc.).
2. **Content sub-type**: Create a second content type "Image Caption Term" to hold an image and its associated data. Then add a multiple "Node Reference" field to your original content type, which will associate your page with its images. This can be made to work, but in practice the editing is at best clumsy. Also, you'll end up with a lot of these little "Image Caption Term" items cluttering up your content management screens, which will confuse novice users.
3. **Experimental module:** Try out an *experimental "Content Multigroup" module (http://drupal.org/node/119102)* that is supposed to let you group multiple CCK fields together. As of this writing (February 2009), this module is not stable (not even released as an "alpha" version), and last time I tried it (October 2008), it didn't work at all with image fields.
4. **Custom CCK field:** Create your own custom CCK field module that contains the desired grouping of information. Doing this is not very difficult, works very well, and is the **subject of this article**.

(http://poplarware.com/sites/poplarware.com/files/page_images/cckfield.jpg)

With that motivation, in this article I describe how to create a module that implements a custom "compound" or "grouped" CCK field for Drupal 6.x and CCK 2.x. Within the compound field defined here, a user will be able to upload an image, provide "alt" and "title" text for the image, enter a caption (arbitrary HTML text), and choose one or more taxonomy terms . This field can be added to a CCK content type as a multiple-valued field, which will allow a content editor to add, edit, delete, and re-order all this information as a group (i.e. keeping each image with its caption and taxonomy). You can click the image to the left to see what it looks like in action.

If you need this exact functionality (image, caption, taxonomy field), you can just use the module I created (download it below) -- it should work. If you need a different set of fields, you should be able to follow the steps in this article to create your own compound CCK field. If you want to have an image or other uploaded file in your compound field, the examples here should be directly applicable.

If you don't have a file upload as a component of your compound field, you will still want to follow the outline of this module (i.e. implement the same hooks for the same reasons), and there are some notes on what you'd need to differently. I've also included some examples from another field module I created, which defines a field for an "office location", with several text components. I'd also recommend checking out the *Link module (http://drupal.org/project/link)* as an example.

If creating your own Drupal module is beyond you, you can also hire Poplar ProductivityWare to create a module for your Drupal site -- click here to *contact Poplar ProductivityWare (/contact)*.

Here's what this article contains:

To understand this article, you will need a basic understanding of PHP, Drupal and its terminology, Drupal module development (hooks, Forms API, and basic module file structure), and the CCK and Views modules (i.e. how to define content types and views -- you don't need to know all about the inner workings of the modules). Check the ***Downloads, Background, and References section*** *(#further)* below for more information, if you encounter terminology or concepts that you are unfamiliar with. The completed module is also provided as a download in that section of the article -- it is GPL licensed, so you can use it and modify it as you wish under the terms of the license. You might want to download it now so you can refer to it as you read this article. The actual module code has more comments, and a couple of functions that were omitted from the article.

And before going any further, I'd like to thank Josh Kopel of Kolaboration Studio, who sponsored part of the development of this article and module.

Getting Started

The first step in creating our module is to choose some names. The module we are creating implements a compound CCK field for an image, caption, and taxonomy term. So, I have chosen to call the CCK field "Image Caption Taxonomy", with machine-readable name "img_cap_tax"; below, when you see "(field)", it refers to the machine-readable name of the field. To distinguish the name of the CCK field from the name of the module, I have chosen to call the module "Image Caption Taxonomy Field", and to use the machine-readable name "img_cap_tax_fld" for the module; I'll refer to this machine-readable name as "(module)".

Next, we need to create files "(module).info" and "(module).module", in order for this to be a Drupal module. A couple of notes:

- Because this module implements a CCK field, we can rely on the CCK module to take care of all the database tables for us. For that reason, we don't need a .install file for this module.
- We do need to define some module dependencies in the .info file, as we will see below. Check the completed .info file in the module download for a list.
- It is possible to define multiple CCK fields in a single module. However, having done it once, I really wouldn't recommend it -- it makes the module file confusing, and less "modular" (i.e. someone might only need one of your fields, but have to load the entire module with several fields in order to get that functionality on their site).

Back to table of contents *(#contents)*

Defining the Field

Now that we have our files created, the next step is to implement some core and CCK-specific hooks that tell CCK about our field. These hook implementations go into the .module file.

CCK hook_field_info()

First, we'll give CCK the field's machine-readable name, human-readable name, and a longer description, by implementing the CCK hook_field_info():

```
function img_cap_tax_fld_field_info() {
  return array(
    'img_cap_tax' => array(
      'label' => t('Image Caption Taxonomy'),
      'description' => t('Stores an image file, text for alt and title tags, a caption, and a taxonomy term'),
    )
  );
}
```

As you can see, the return value for hook_field_info() is an associative array, keyed on the machine-readable name of the CCK field we are defining. Each element of the returned array is an associative array, with components 'label' giving the human-readable name of our field, and 'description' giving a more verbose description. And since all human-readable text in Drupal modules should be internationalization-ready, we enclose it in the t() function.

Core hook_install(), hook_uninstall(), hook_enable(), hook_disable()

Next, we will implement the core Drupal hook_install(), hook_uninstall(), hook_enable(), and hook_disable() so that our field will be properly added and removed from Drupal and CCK. These all work the same way: we just tell Drupal to let CCK handle the process through its content_notify() function, which takes care of all of the details. For example, here is the function for the core Drupal hook_install():

```
function img_cap_tax_fld_install() {
  content_notify('install', 'img_cap_tax');
}
```

The other three hooks are exactly the same -- just substitute "uninstall", "enable", or "disable" for "install". Note that it is possible to put these functions into a module's (module).install file instead of (module).module, but since they are so small in this case (and we have nothing else to go in the (module).install file), I've chosen to leave them in (module).module.

One other detail: these functions are calling the CCK module function content_notify() directly. So, we need to make sure that function is loaded when we call it:

- Make our module dependent on the CCK module (whose machine-readable name is "content") -- that dependency goes into the .info file.
- Verify which file the content_notify() function is defined in -- if it is in an include file rather than the main .module file, put an include directive in the hook_init() implementation in our module.

We'll need to do this for other module functions we'll be calling directly later in this article, so I'll just suggest that you look at the completed module's .info file (or the ***reference section below*** *(#further)*) to see the module dependencies, and the img_cap_tax_fld_init() function (implementation of hook_init()) to see what include files we needed to load explicitly.

CCK hook_field_settings()

Now, we need to tell CCK about the "settings" for this field, using CCK hook_field_settings(). This is a sort of catch-all hook for CCK, with several "operations", each one letting our field module give a different piece of information to CCK. Our CCK field is based on an image, and needs to behave very similarly to the Image field from the existing ImageField module. That field, in turn (in the current implementation of ImageField) is actually just a FileField file field, with a special "widget" and "formatter" (see sections below) that makes it show as an image rather than a file. So, for our field, we will borrow heavily from the FileField module in our field settings function. For the operations that need to do something different from FileField, we'll call into other functions, which are described in the next section. Here's the function:

```
function img_cap_tax_fld_field_settings( $op, $field ) {
  switch( $op ) {
    case 'form':
      return img_cap_tax_fld_field_settings_form( $field );

    case 'save':
      return img_cap_tax_fld_field_settings_save( $field );

    default:
      return filefield_field_settings( $op, $field );
  }
}
```

(Of course, we'll need to make sure that FileField is a dependency of this module, as described above.)

Note: if your CCK field does not contain an uploaded file, you will need to do something a little different here. The same operations apply (see section below), but you'll need to handle the operations yourself. If you don't need to do any validation beyond what CCK does, and if you don't need a special settings form, which is often the case, you might have a function like this example from a field that defines an "office location" (perhaps with some additional operations more similar to the example above):

```
function office_field_field_settings($op, $field) {
  switch ($op) {
    case 'database columns':
      $columns['loc_name'] = array('type' => 'varchar', 'length' => 255, 'not null' => FALSE, 'sortable' => TRUE, 'default' =>
 '');
      $columns['phone'] = array('type' => 'varchar', 'length' => 255, 'not null' => FALSE, 'sortable' => FALSE, 'default' => '
');
      $columns['fax'] = array('type' => 'varchar', 'length' => 255, 'not null' => FALSE, 'sortable' => FALSE, 'default' => '')
;
      $columns['address'] = array('type' => 'text', 'not null' => FALSE, 'sortable' => FALSE, 'default' => '');
      return $columns;
  }
}
```

filefield_field_settings() Operations Functions

The next step is to define the individual operations functions that are called from the field settings function in the previous section. For historical reasons, in this module the functions are located in an include file, (module)_field.inc. Let's start with the **"form" and "save" operations**, which define the field settings form the user can use to customize the field (apart from the sections already provided by the FileField module), and which fields in the settings form should be saved to the database. There is also a **"validate"** operation, which lets you run checks on the submitted data, but we don't need anything beyond what FileField does for this, so I won't describe that here.

In our field settings form, we want to do start with what the FileField module does for its field, add in most of what the Content Taxonomy module does for its field, and also allow the user to choose between plain text and having an Input Format (filtered) for the caption field.

We'll also make a couple of changes to what Content Taxonomy does. First, Content Taxonomy has a setting that allows you to take the taxonomy choices from its field and apply them to the node as a whole. We won't allow for that option on our field, as it doesn't make sense in a compound field. Second, we also want to let the user choose whether the taxonomy term is optional, and whether it should be multiple-valued. We will also add some fieldsets to the form for grouping.

So, with all of that motivation, here are our "form" and "save" operations functions:

```
function img_cap_tax_fld_field_settings_form( $field ) {
  $form1 = filefield_field_settings( 'form', $field );

  $form2 = content_taxonomy_field_settings( 'form', $field );
  $form2['save_term_node']['#type'] = 'hidden';
  $form2['taxonomy_group'] = array(
    '#type' => 'fieldset',
    '#title' => 'Taxonomy',
    '#collapsible' => 0,
  );
  $form2['taxonomy_group']['vid'] = $form2['vid'];
  unset( $form2['vid'] );
  $form2['taxonomy_group']['allow_multiple'] = array(
    '#type' => 'checkbox',
    '#title' => t('Allow multiple taxonomy terms'),
    '#default_value' => is_numeric($field['allow_multiple']) ? $field['allow_multiple'] : 0,
    '#description' => t('If this option is checked, the user can select multiple taxonomy terms for each image; otherwise, at
most one.'),
  );
  $form2['taxonomy_group']['required_term'] = array(
    '#type' => 'checkbox',
    '#title' => t('Taxonomy required'),
    '#default_value' => is_numeric($field['required_term']) ? $field['required_term'] : 0,
    '#description' => t('If this option is checked, the user must select at least one taxonomy term for each image; otherwise,
 it is optional.'),
  );
  $form2['hierarchical_vocabulary']['#weight'] = 100;

  $form3 = array( 'text_processing' => array(
    '#type' => 'radios',
    '#title' => t('Text processing for Caption'),
    '#default_value' => is_numeric($field['text_processing']) ? $field['text_processing'] : 0,
    '#options' => array( 0 => t('Plain text'), 1 => t('Filtered text (user selects input format)')),
  ));

  return $form1 + $form3 + $form2;
}

function img_cap_tax_fld_field_settings_save( $field ) {
  $flds1 = filefield_field_settings( 'save', $field );
  $flds2 = content_taxonomy_field_settings( 'save', $field );
  $flds2[] = 'allow_multiple';
  $flds2[] = 'required_term';
  $flds3 = array( 'text_processing' );

  return array_merge( $flds1, $flds2, $flds3 );
}
```

There is also a hook_field_settings() operation for **"database columns"**, where you can define the database columns that CCK will use to store your field's data. However, in this case the FileField module creates a serialized "data" field in the database for us to use. We'll put all of the alt, title, caption, and taxonomy information into the "data" array, so we don't have to add any additional database columns. (The example above for an "office location" field shows how to create database columns, if you need to in your module.)

Finally, there is a hook_field_settings() operation for **"views data"**, which allows a CCK field to return information about how it can be used for sorting and filtering in the Views module. (Any CCK field can automatically be included as a Field in Views, so we don't have to worry about that.) Keeping in mind that our field is mainly useful if it is added to a content type as a multiple-valued field (otherwise, you wouldn't really need a compound field), probably *sorting* the nodes in a view based on some aspect of this field doesn't really make sense. However, I can envision *filtering* a view using this field in a couple of ways:

- Whether or not the content has at least one image attached to it. This should come from the FileField module.
- Whether a particular taxonomy term is present or not.

As of this writing, this hasn't been implemented. Sorry!

CCK hook_field()

The next set of things we need to define is included in CCK hook_field(), which basically tells CCK if it needs to do anything special when a the field is loaded from or saved to the database. For the image in our field, there is a special action: move the image file from its temporary location to a permanent location, and make note of that location in the database field (the FileField module has a function that does this). For the caption, we need to "sanitize" it, according to the input format (i.e. make sure it contains only the allowed HTML tags, or no text if the input format is plain text). For the Taxonomy Term, we don't need to do anything special. So, here's our hook_field() operation:

```
function img_cap_tax_fld_field($op, $node, $field, &$items, $teaser, $page) {
  if( $op == 'sanitize' ) {
    img_cap_tax_fld_field_sanitize( $node, $field, $items, $teaser, $page );
  }
  return filefield_field( $op, $node, $field, $items, $teaser, $page );
}
```

In our "sanitize" operation function, we will do pretty much what the Text field module in CCK does to sanitize text fields: figure out which input format was chosen, filter using that input format, and save the result with a "safe" prefix so it can be used for theming. Here is the function (I've put it into the (module)_field.inc file):

```
function img_cap_tax_fld_field_sanitize($node, $field, &$items, $teaser, $page) {
  $isplain = empty( $field['text_processing'] );
  $check_access = is_null( $node ) ||
    ( isset($node->build_mode) && $node->build_mode == NODE_BUILD_PREVIEW );

  foreach( $items as $delta => $item ) {
    $dat = $item['data'];
    if( !is_array( $dat )) {
      $dat = unserialize( $dat );
    }
    $text = isset( $dat['caption'] ) ? $dat['caption'] : '';
    if( $isplain ) {
      $text = check_plain( $text );
    } else {
      $text = check_markup( $text, $item['format'], $check_access );
    }
    $items[$delta]['safe_caption'] = $text;
  }
}
```

One detail to note here is that I found that the "data" array from FileField is sometimes coming into this function in a serialized format (I'm not sure exactly why). So this function checks to see whether it needs to be unserialized before making use of it.

If you're defining a non-file CCK field, typical operations you might need to do in hook_field() would be to serialize or unserialize data, and "sanitize" data for viewing. Most modules will just need to sanitize; here is an example from the "office location" field, which assumes that all of its fields should be plain text:

```
function office_field_field($op, &$node, $field, &$items, $teaser, $page) {
  switch ($op) {
    case 'sanitize':
      foreach ($items as $delta => $item) {
        foreach ( $item as $col => $dat ) {
          $items[$delta]['safe_' . $col ] = check_plain($item[ $col ]);
        }
      }
      break;
  }
}
```

CCK hook_content_is_empty(), hook_default_value()

There are a few more pieces of information CCK needs about our field: how to tell if it is "empty" of information (CCK hook_content_is_empty()), and what the default value should be (CCK hook_default_value()). Since the primary component of our field is an image, we will say that the field is considered "empty" if it has no image file, and we'll let FileField handle that:

```
function img_cap_tax_fld_content_is_empty( $item, $field ) {
  return filefield_content_is_empty( $item, $field );
}
```

Here's another example from the "office location" field -- assuming the field is "empty" if both the location name and address are missing:

```
function office_field_content_is_empty($item, $field) {
  if (empty($item['loc_name']) && empty( $item['address'])) {
    return TRUE;
  }
  return FALSE;
}
```

In our image/caption module we don't have a default value, actually (it isn't too common), but just in case FileField implements a default value setting in the future, we'll let FileField define the default value:

```
function img_cap_tax_fld_default_value(&$form, &$form_state, $field, $delta) {
  return filefield_default_value($form, $form_state, $field, $delta);
}
```

If you want a default value for your field, you'll probably need to set something up in your settings form so that your users can define what the default value is, and then use the default value hook to set it up. I don't know of a good example for this.

Back to table of contents *(#contents)*

Defining the Widget

Now that we have the CCK field itself defined, our next task is to define a "widget", which is the CCK name for a form used to edit the field. Our widget will need to provide a way to upload the image file and preview/change the uploaded file; enter the alt, title, and caption text; choose an input format for the caption; and choose a taxonomy term (we'll use a drop-down select list for that). As before, we'll let FileField and Content Taxonomy handle their parts, so all we'll need to do is add the text fields and image format. We'll create a widget called "Image, Caption, Taxonomy Select", with machine-readable name "img_cap_tax_sel_widget", referred to below as "(widget)".

I will note here that it is possible for a CCK field module to define more than one widget for a single field. For instance, the Content Taxonomy module has several options: a drop-down select list, a type-ahead auto-complete text field, etc. There are notes below on what you'd need to change to add additional widgets.

CCK hook_widget_info()

The first step in providing a widget is to give CCK the basic widget information (machine-readable name, human-readable name, what field types it applies to, etc.), by implementing the CCK hook_widget_info():

```
function img_cap_tax_fld_widget_info() {
  return array(
    'img_cap_tax_sel_widget' => array(
      'label' => t('Image, Caption, Taxonomy Select'),
      'field types' => array('img_cap_tax'),
      'multiple values' => CONTENT_HANDLE_CORE,
      'callbacks' => array('default value' => CONTENT_CALLBACK_CUSTOM),
      'description' => t('An edit widget for Image Caption Taxonomy fields that allows upload/preview of the image, and choose
s taxonomy terms from a drop-down select list.' ),
    ),
  );
}
```

As is common in Drupal, the return value of this hook is an associative array of associative arrays, and if we wanted to provide multiple widgets in our module, we would just need to add additional elements to the outer array.

FAPI hook_elements()

Next, we need to define the widget's data entry form and how to process it. This is done using the core Forms API hook_elements(), which returns an associative array with one element (or more, if you have multiple widgets) whose key is the machine-readable name of our widget, and whose value is an associative array that either defines the form completely (as is done in the Content Taxonomy module), or gives a "process" callback function and some other information (as is done in ImageField; this is also the method recommended, in general, for the Forms API). We'll use the process callback method, and let ImageField and FileField set up most of the array:

```
function img_cap_tax_fld_elements() {
  $imgel = imagefield_elements();
  $elements = array( 'img_cap_tax_sel_widget' => $imgel[ 'imagefield_widget' ]);
  $elements['img_cap_tax_sel_widget']['#process'][] = 'img_cap_tax_fld_widget_process';
  $elements['img_cap_tax_sel_widget']['#element_validate']= array('img_cap_tax_fld_widget_validate');

  return $elements;
}
```

We will need to define the "process" callback function, which is what actually defines the form elements -- see section below. Also, we have a custom validate callback, because the one used by FileField (and ImageField) does not work for our module -- see section below. One other detail is that we'll also need to register a themeable element for this widget form; we'll cover that in the "Formatter and Theming" section below with the other theme information, and show the theming functions below.

If your CCK field doesn't include a file, you'll need to define all of the elements array yourself, but it's pretty simple for most compound fields, because they won't have any special processing to do. Here's an example from the "office location" field:

```
function office_field_elements() {
  $elements = array( 'loc_entry' =>
    array(
      '#input' => TRUE,
      '#process' => array( 'office_field_loc_entry_process' ),
    ),
  );

  return $elements;
}
```

Process and Validate Callbacks for Widget Form

The "process" callback from the form array of the previous section is what actually defines and returns the form elements (a Forms API array). We've left ImageField to take care of its processing, and added our processing function to the end; we'll put the function in file (module)_widget.inc. So, our function needs to add a text area and input format selector for the "caption" field and a drop-down list for the Taxonomy term. All of this information will get stored in FileField's "data" array, and we'll have to be careful to choose the same array keys as the Content Taxonomy module for the taxonomy fields. For the caption's input format, we want to choose "format" as the array key, because this is the standard field name for input formats. (Some other modules depend on that convention; for example, the WYSIWYG editor module chooses which fields to attach editors to by looking for textarea fields that are followed by "format" fields.)

```
function img_cap_tax_sel_widget_process($element, $edit, &$form_state, $form) {
  $defaults = $element['#value']['data'];
  if( !is_array( $defaults )) {
    $defaults = unserialize( $defaults );
  }

  $field = content_fields($element['#field_name'], $element['#type_name']);

  $element['data']['caption'] = array(
    '#title' => t( 'Caption' ),
    '#type' => 'textarea',
    '#rows' => $field['widget']['rows'],
    '#cols' => $field['widget']['cols'],
    '#default_value' => $defaults['caption'],
    '#weight' => 4,
  );

  if (!empty($field['text_processing'])) {
    $filt = isset( $defaults['format'] ) ? $defaults['format'] : FILTER_FORMAT_DEFAULT;
    $par = $element['#parents'];
    $par[] = 'data';
    $par[] = 'format';
    $element['data']['format'] = filter_form( $filt, 1, $par );
    $element['data']['format']['#weight'] = 5;
  }

  $mult = $field['allow_multiple'];
  $req = $field['required_term'];
  $opts = content_taxonomy_allowed_values( $field );
  if( !$req && !$mult ) {
    $none = theme( 'content_taxonomy_options_widgets_none', $field );
    $opts = array( '' => $none ) + $opts;
  }
  $element['data']['value'] = array(
    '#title' => t( 'Taxonomy Terms' ),
    '#type' => 'select',
    '#default_value' => $defaults['value'],
    '#options' => $opts,
    '#weight' => 6,
  );
  if( $mult ) {
    $element['data']['value']['#multiple'] = TRUE;
  }

  return $element;
}
```

Note that the rows and columns settings for the caption, as well the settings for whether the taxonomy term is a single or multiple select and required or not, come from the widget settings form -- see CCK hook_widget_settings() below.

If you are defining a compound CCK field that does not involve file uploads, here is the process function for the "office location" field as an example:

```
function office_field_loc_entry_process($element, $edit, &$form_state, $form) {

  $defaults = $element['#value'];
  $field = content_fields($element['#field_name'], $element['#type_name']);

  $element['loc_name'] = array(
    '#title' => t( 'Location name' ),
    '#type' => 'textfield',
    '#default_value' => $defaults['loc_name'],
    '#weight' => 2,
  );

  $element['phone'] = array(
    '#title' => t( 'Phone' ),
    '#type' => 'textfield',
    '#default_value' => $defaults['phone'],
    '#weight' => 3,
  );

  $element['fax'] = array(
    '#title' => t( 'Fax' ),
    '#type' => 'textfield',
    '#default_value' => $defaults['fax'],
    '#weight' => 4,
  );

  $element['address'] = array(
    '#title' => t( 'Address' ),
    '#type' => 'textarea',
    '#default_value' => $defaults['address'],
    '#weight' => 5,
  );

  return $element;
}
```

Our module also needs a validation callback, which will verify information when the node edit form is submitted. In FileField, the validation checks to make sure the file exists, and that if the field can only reference a file if it has been uploaded via this field module (so that the field will not be deleted erroneously if the node is deleted). Unfortunately, the existing FileField validate function (filefield_widget_validate() in filefield_widget.inc) will not work as-is for our module, because it tacitly assumes that the field type name is 'filefield', which is not the case for our field. So, we have to create our own function to replicate it. Assuming that the person using this field is uploading their own images and not doing anything crazy, all we really need to do is make sure the file exists. Here's the function:

```
function img_cap_tax_fld_widget_validate(&$element, &$form_state) {

  if (empty($element['fid']['#value'])) {
    return;
  }

  $field = content_fields($element['#field_name'], $element['#type_name']);
  $ftitle = $field['widget']['label'];

  if ( !( $file = field_file_load($element['fid']['#value']))) {
    form_error($element, t('The file referenced by the %field field does not exist.', array('%field' => $ftitle )));
  }
}
```

Widget theme function

In order to display the editing widget, we need to define a theme function, theme_(widget), which is called as an envelope for each item in the multiple-valued list of image/caption/taxonomy data for a node. We simply tell it to render the form; ImageField and FileField will take care of the rest, such as showing a thumbnail of the image. Here is the function:

```
function theme_img_cap_tax_sel_widget(&$element) {
  return theme( 'form_element', $element, $element['#children'] );
}
```

Note that there appears to be a rather annoying choice in the FileField CSS file that makes the editing widget get very narrow. You may want to add this to your theme's CSS file:

```
.filefield-element .widget-edit, .filefield-element .widget-preview {
float: none;
}
```

CCK hook_widget()

The next hook we need to implement in order to define the editing widget is CCK hook_widget(). This hook will be called each time one of our fields is added to the form, with the $delta parameter set to the multiple-value index (0 for the first item, 1 for the second, etc.). The return value is a Forms API array that should set up default values for the form and define callbacks. As usual, we'll let FileField and ImageField handle most of the details (Content Taxonomy doesn't need to do anything special), by calling the filefield_widget() function. That function assumes that:

- Our module contains a file called (module)_widget.inc (the filefield_widget() function will load it).
- The $items['delta'] array has been set up with an array of the default values for the text fields in our compound field, before filefield_widget() is called.

So, our hook_widget() implementation is:

```
function img_cap_tax_fld_widget(&$form, &$form_state, $field, $items, $delta = 0) {
  if (empty($items[$delta])) {
    $items[$delta] = array('alt' => '', 'title' => '', 'caption' => '', 'value' => 0);
  }

  $element = filefield_widget($form, $form_state, $field, $items, $delta);
  $element['#upload_validators'] += imagefield_widget_upload_validators($field);

  return $element;
}
```

If your module contains more than one widget, you'll want to do something like this in your hook_widget() implementation:

```
switch( $field['widget']['type'] ) {
  case 'first_widget_machine_name':
    (code for this widget)
    break;

  case 'second_widget_machine_name':
    (code for this widget)
    break;
}
```

And here is what the "office location" field does:

```
function office_field_widget(&$form, &$form_state, $field, $items, $delta = 0) {
  $element = array(
    '#type' => $field['widget']['type'],
    '#default_value' => isset($items[$delta]) ? $items[$delta] : '',
  );
  return $element;
}
```

CCK hook_widget_settings()

The final piece in defining the widget is to create a widget settings form, which is done by implementing CCK hook_widget_settings(). Like many of the other CCK hooks, hook_widget_settings() has several operations: one to create a settings form, one to validate the form, and one to save the form. We'll let ImageField take care of validation, and build our own functions for the form and save operations:

```
function img_cap_tax_fld_widget_settings( $op, $widget ) {
  switch ($op) {
    case 'form':
      return img_cap_tax_fld_widget_settings_form($widget);
    case 'validate':
      return imagefield_widget_settings_validate($widget);
    case 'save':
      return img_cap_tax_fld_widget_settings_save($widget);
  }
}
```

Our "office location" form, like many CCK fields, doesn't have any widget settings -- there are no choices to be made to customize the widget.

Also note that if you have multiple widgets in your module, you can do a switch on $widget['type'] to handle the different widgets in your hook_widget_settings() implementation.

filefield_widget_settings() callbacks

There are two operation callbacks defined in our hook_widget_settings() implementation. The 'form' operation returns the widget settings form; the 'save' operation returns a list of which form data should be saved to the database.

First, let's work on the settings form. The existing FileField and ImageField modules have a widget settings form that lets the user set a file path, allowed file extensions, maximum file size, and other settings. We'll use that form, but modify it so that alt and title can always be customized rather than leaving those as options as they are in ImageField (we still want the settings to be there, so that other ImageField functions we call will have the right values set). The Content Taxonomy select list widget has settings for indentation and grouping terms, which we'll also want. Finally, most multi-line text fields let the user choose how many rows and/or columns to display, so we'll want that setting for the caption field. Putting this all together:

```
function img_cap_tax_fld_widget_settings_form( $widget ) {
  $form = imagefield_widget_settings_form( $widget );

  $form['custom_alt'] = $form['alt_settings']['custom_alt'];
  $form['custom_alt']['#type'] = 'hidden';
  $form['custom_alt']['#value'] = 1;
  $form['alt'] = $form['alt_settings']['alt'];
  $form['alt']['#type'] = 'hidden';
  $form['alt']['#value'] = '';
  unset( $form['alt']['#suffix'] );
  unset( $form['alt_settings'] );

  $form['custom_title'] = $form['title_settings']['custom_title'];
  $form['custom_title']['#type'] = 'hidden';
  $form['custom_title']['#value'] = 1;
  $form['title'] = $form['title_settings']['title'];
  $form['title']['#type'] = 'hidden';
  $form['title']['#value'] = '';
  unset( $form['title']['#suffix'] );
  unset( $form['title_settings'] );

  $rows = (isset($widget['rows']) && is_numeric($widget['rows'])) ? $widget['rows'] : 5;

  $form['rows'] = array(
    '#type' => 'textfield',
    '#title' => t('Number of rows in caption field'),
    '#default_value' => $rows,
    '#element_validate' => array('_text_widget_settings_row_validate'),
    '#required' => TRUE,
    '#weight' => 8,
  );

  $cols = (isset($widget['cols']) && is_numeric($widget['cols'])) ? $widget['cols'] : 40;

  $form['cols'] = array(
    '#type' => 'textfield',
    '#title' => t('Number of columns in caption field'),
    '#default_value' => $cols,
    '#element_validate' => array('_text_widget_settings_row_validate'),
    '#required' => TRUE,
    '#weight' => 9,
  );

  $form2 = content_taxonomy_options_widget_settings( 'form', $widget );
  $form2['settings']['#title'] = t( 'Settings for Taxonomy' );

  $form = $form + $form2;
  return $form;
}
```

We don't need to do anything special for the 'validate' operation, as neither ImageField nor Content Taxonomy needs us to do anything beyond what FileField does. So, we won't define this function. The 'save' operation returns a list of fields to save from the settings form:

```
function img_cap_tax_fld_widget_settings_save( $widget ) {
  $arr = imagefield_widget_settings_save( $widget );
  $arr[] = 'rows';
  $arr[] = 'cols';
  $arr2 = content_taxonomy_options_widget_settings( 'save', $widget );
  $arr2[] = 'allow_multiple';
  $arr2[] = 'required_term';
  return array_merge( $arr, $arr2 );
}
```

Back to table of contents *(#contents)*

Defining the Formatter and Theming

Having now defined our field and the editing widget, the final thing we need to do is to define how it will look to someone who is visiting the site (of course, the theme can override this). We do this by defining a "formatter" for the field; like widgets, there can be more than one formatter defined for a field, but we'll only create one in this example. We'll give our formatter the machine name 'default', which we'll refer to as "(formatter)" below, and human-readable name 'Image with Caption and Taxonomy Terms'. Note that machine-readable names for formatters do not need to be unique, except within your module.

CCK hook_field_formatter_info()

To tell CCK about our formatter, we implement CCK hook_field_formatter_info():

```
function img_cap_tax_fld_field_formatter_info() {
  return array(
    'default' => array(
       'label' => t( 'Image with Caption and Taxonomy Terms' ),
       'field types' => array( 'img_cap_tax' ),
    ),
  );
}
```

Core hook_theme()

Once the formatter has been defined, CCK will assume there is a corresponding themeable element called (module)_formatter_(formatter), which we need to register in the core hook_theme(). As noted above, we also need to register the themeable element for the widget form, so our hook_theme() implementation is:

```
function img_cap_tax_fld_theme() {
  return array(
    'img_cap_tax_sel_widget' => array(
       'arguments' => array('element' => NULL),
    ),

    'img_cap_tax_fld_formatter_default' => array(
      'arguments' => array('element' => NULL),
    ),
  );
}
```

Theme Functions

Finally, we need to create a theme_(element) function for the formatter (the widget theme functions were given in previous sections). The formatter theme function displays the image, caption, and taxonomy term:

```
function theme_img_cap_tax_fld_formatter_default( $element = NULL ) {

  if( empty( $element['#item'] )) {
    return '';
  }

  $img = theme( 'imagefield_formatter_image_plain', $element );

  $cap = $element['#item']['safe_caption'];

  $tax = '';
  $sep = '';
  $val = $element['#item']['data']['value'];
  if( !is_array( $val )) {
    $val = array( $val );
  }
  foreach( $val as $tid ) {
    $term = taxonomy_get_term( $tid );
    $tax .= $sep . check_plain( $term->name );
    $sep = ', ';
  }

  return '<div class="imgcaptax_outer">' .
    '<div class="imgcaptax_img">' . $img . '</div>' .
    '<div class="imgcaptax_cap">' . $cap . '</div>' .
    '<div class="imgcaptax_tax">' . $tax . '</div>' .
    '</div>';
}
```

Back to table of contents *(#contents)*

Downloads, Background, and References

Below you can download the completed module that this article describes. The file is 13.6 KB (13,982 bytes), and the current version of the module is 6.x-1.3.

To keep up with new releases (if any), subscribe to the RSS feed for this site. If you would like to hire Poplar ProductivityWare to create a module for your Drupal site, or have any comments about this article or the module, please *contact Poplar ProductivityWare (/contact)* .

This module was developed and tested using *Drupal (http://drupal.org)* 6.9 and the following module dependencies:

- *Content Construction Kit (CCK) (http://drupal.org/project/cck)* version 6.x-2.2 and its included Option Widgets and Text modules
- *ImageField (http://drupal.org/project/imagefield)* version 6.x-3.0. Note: A lot changed between the alpha versions of this module and the final release! The custom field module in the download above will NOT work with the alpha versions of ImageField and FileField.
- *FileField (http://drupal.org/project/filefield)* version 6.x-3.0 -- see note above about alpha version.
- ***ImageAPI*** *(http://drupal.org/project/imageapi)* version 6.x-1.6
- *Content Taxonomy (http://drupal.org/project/content_taxonomy)* version 6.x-1.0-beta6, and its included Content Taxonomy Options module

Background information and references:

- Background material on Drupal, CCK, and Views: my ***Drupal Cheat Sheet*** *(/articles/drupal_cheat_sheet)* article, and a *video of a talk I did on CCK and Views* (/articles/drupal_cck_talk) .

- Terminology note: a PHP programmer who is new to Drupal module development will need to understand the concept of a "hook" in Drupal. Basically, a **hook** is a structured way for a module to add custom functionality to Drupal. A module whose machine-readable name is "abc" can implement hook "foo", simply by defining a PHP function called "abc_foo()". When Drupal gets to the right place in its execution to do action "foo", it will automatically call all module functions that implement the hook. The core of Drupal has some hooks (see Drupal API Reference below); the CCK module also defines its own hooks (no good documentation I am aware of, which is one reason I wrote this article, to describe which hooks you need to implement).
- Useful references:
  - ***Drupal Module Developers Guide*** (http://drupal.org/node/508) , including the ***Writing .info files guide*** (http://drupal.org/node/231036) page
  - ***Drupal API Reference*** (http://api.drupal.org)
  - ***Drupal Forms API Quickstart Guide*** (http://api.drupal.org/api/file/developer/topics/forms_api.html/6) and ***Forms API Reference*** (http://api.drupal.org/api/file/developer/topics/forms_api_reference.html/6)
  - ***Guide to developing CCK field modules in Drupal 5*** (http://drupal.org/node/106716) and an unfortunately only ***partially-complete guide to upgrading CCK modules from Drupal 5 to Drupal 6*** (http://drupal.org/node/191796) . This article differs from the official references in that it pertains to Drupal 6; it presents a complete, working module as its end product rather than disjointed examples; and it specifically addresses how to incorporate existing CCK fields into a compound field, and in particular ImageField and FileField, rather than creating a stand-alone CCK field type from scratch.
  - ***Views module API documentation*** (http://views.doc.logrus.com/) , and ***help file on describing tables to Views*** (http://views-help.doc.logrus.com/help/views/api-tables)

Back to table of contents (#contents)

File Downloads:

Zip file containing the Image Caption and Taxonomy field (http://www.poplarware.com/sites/poplarware.com/files/downloads/img_cap_tax_fld.zip)

Comments

Post new comment

Your name: *

Anonymous

E-mail: *

The content of this field is kept private and will not be shown publicly.

Homepage:

Subject:

Comment: *

Web page addresses and e-mail addresses turn into links automatically.
Allowed HTML tags: <a>

Lines and paragraphs break automatically.

More information about formatting options *(/filter/tips)*

CAPTCHA

This question is for testing whether you are a human visitor and to prevent automated spam submissions.

Math question: *

+ 0 = 0

Solve this math question and enter the solution with digits. E.g. for "two plus four = ?" enter "6".

Preview