Forms API 3.0

Adrian Rossouw

Forms API 1.0

- Drupal 4.7.
- Nested arrays for form definition.
- Separation of validation, and submission code into callbacks.
- Introduce hook_form_alter.
- Introduced callback array format.

Forms API 2.0

- Drupal 5.0.
- Separation of form definition and page workflow.
- Form definitions are now ‘pulled’ instead of ‘pushed’.
- Programmatic submission of forms.
- Form rendering pipeline now used for links and node body.

Forms API 3.0 : Goals

- Choose a better name =)
- Separation of data model from display structure.
- Sensible defaults, consistently applied.
- Stricter and more consistent validation.
- Provide context for important considerations, such as filters and localization
- Improved performance, through better caching and less needless recursion.
- Cleaner, and simpler code. Hacks begone.

A rose by any other name.

- Forms API has always been a misnomer. It is far more general purpose than that.
- Structure definition : Defining field and view structure information.
- Execution Pipeline : Program structure through callbacks. Like hooks on steroids.
- Render Pipeline : drupal_render().
- Input Pipeline : What to do with $_POST. Handled primarily in form_builder.
- Validation Pipeline : Validating the input against the data structure.

Introduction to MVC Pattern.

- What we have in FAPI, is an emergent MVC pattern, albeit with a merged Model and View component.

- Though MVC comes in different flavors, control flow generally works as follows:

    - The user interacts with the user interface in some way (e.g., user presses a button)
    - A controller handles the input event from the user interface, often via a registered handler or callback.
    - The controller accesses the model, possibly updating it in a way appropriate to the user's action (e.g., controller updates user's shopping cart). Complex controllers are often structured using the command pattern to encapsulate actions and simplify extension.
    - A view uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view.
    - The user interface waits for further user interactions, which begins the cycle anew.

Model API : Defining Your Data Model.

- Multiple data models already exist in drupal.
    - Forms API.
    - CCK. CCK defines content types by adding fields, of certain types, with certain validation, to the content type.
    - User profiles. User profiles have a very similar system, except not quite as cleanly designed as CCK, without the proper separation of the data structure, and the display widgets.

Existing Data Models (Continued)

- Views: Provides a model of which fields are available in certain objects, as well as where their data is stored, how it should be represented in a view AND the relationships between fields.

- Import Export API: A complete data model for all data types in Drupal core (nodes, users, vocabularies, terms and many many more.). These are then used in combination with a query builder to extract and import data. Similar to views data model, in that table and relationship information is contained.

- Node Import: Data models for each individual node type, and re-implementation of the validation rules.

- and probably a couple more.

A Quick Note about the Following Examples.

The code examples are taken from running proof of concept code, built entirely outside of Drupal.

One of the most noticeable things is the complete lack of ‘#’ notation.

Instead of having the properties and children inhabit the same namespace within the tree, I have opted to use [p] notation, meaning that all properties are contained within an array element accessed with the constant ‘p’.

When you read [p], it actually means ‘__properties__’, or in my mind ‘__pretty__’.

This is actually also incredibly helpful in debugging, as all the properties are grouped together when you do a print_r.

Proposed Data Model in ‘FAPI 3.0’

```
/**
 * Very simplified model.
 * These are the fields you want in your data model.
 * These are the fields that will be validated against.
 */
function model_node() {
  $model[‘nid’] = drupal_field(‘id’);
  $model['title'] = drupal_field('title');
  $model['author'] = drupal_field('author');
  $model['content'] = drupal_field('content');

  return $model;
}
```

I Know What You're Thinking... drupal_field?!?

The system makes use of 'constructor functions', but is still inherently based on arrays. The constructor functions are responsible for providing the 'default values' for fields, based on their type.

```
/**
 * Field constructor.
 * returns a structured array, possibly also merging any defaults that might
 * have been defined for this specific field type.
 */
function drupal_field($type, $arguments = array()) {
  $field = array();
  $function = 'field_' . $type;
  if (function_exists($function)) {
    $field = $function();
  }
  $data[p] = $arguments + $field +
             array('type' => 'field', 'field_type' => $type,
             'display_widget' => 'text', 'edit_widget' => 'textfield');
  return $data;
}
```

Wait, But Where Did All the Properties Go?

Any properties you want to override, can be specified in the second parameter for the constructor functions.

```
/**
 * Short hand notation, using just the field type and all field, defaults.
 */

$model['title'] = drupal_field('title');

/**
 * Which is functionally the same as :
 */
$model['title'] = drupal_field('title', array('title' => t('Title'), 'value' => $node['title']));

/**
 * Or if you want to add more properties you can do :
 */
$model['title'] = drupal_field('title', array('description' => t('My description goes here'),
'some_other_property' => 'blah'));
```

Field Definition Functions.

Fields are defined in their own functions, and provide sensible defaults that make sense for that field. Field types are not just ‘checkbox’ or ‘radio’. Field types are more related to data types, such as integer , string and real.

```
/**
* Field definition functions.
 * Defaults are applied to the fields in the data model.
 * The different widgets specify how to display the field types on either display or edit (forms).
 */
function field_title() {
 return array('edit_widget' => 'textfield', 'view_widget' => ‘title', 'filter' => drupal_callback
(‘filter_xss’));
}
function field_content() {
  return array('edit_widget' => 'textarea', 'view_widget' => 'markup');
}
function field_email() {
  return array('edit_widget' => 'textfield', 'view_widget' => 'email', 'validate' => drupal_callback
('valid_email'));
}
```

drupal_callback?

Just like there is a constructor for field types, there are constructors for callbacks. Callbacks have become one of FAPI's most powerful features, what drupal_callback does, is it formalizes them, as well as allow ANY property to be a callback instead of a value.

```
/**
* This function simply returns an array structure, which is executed
* when found.
*/

function callback($function) {
  $args = func_get_args();
  $callback = array_shift($args);
  $data[p] = array('type' => 'callback'));
  $data[$function . '-' . sizeof($args)] => array('callback' => $callback, 'arguments'
=> $args);
  return $data;
}
```

A Quick Note About Callbacks.

Callbacks are much more important in this system than previously, because of one of their most important distinctions : cache-ability.

For example, in the case of the ‘value’ property, if you set the value property to a $value, you will not be able to re-use the model for other objects. You can cache the promise of a value, when the callback is triggered, but you can’t reliably cache the return value for all fields that have the field.

Since the data model could end up being fairly complex, there is a real need to be able to cache the created data model, after all the hook_model_alter malarky has taken place, because rebuilding the model every time you want to validate a node (for instance), would be a lot of wasted processing.

Advanced callbacks.

```
/**
 * Example of callback unions.
 * Because callbacks are arrays, you can create a union of them (+
operator).
 */
$model['field'] = drupal_field('type', array(
  'access' => drupal_callback('user_access', 'some permission')
                + drupal_callback('user_access', 'another permission')
));
```

Data models and CRUD (create, read, update and delete)

- One of the things that is generally part and parcel of implementing a data model, is developing CRUD functionality around it.
- Each data model, can have function which handle the creation, reading (loading) , updating (editing) and deletion of the specified object.
- Simply by having any of the CRUD functions defined for the specific object type, you will be able to generate both a display, and a form for it, due to the nature of the data model API.
- Rails calls this ‘scaffolding’.

Crud functions (simplified).

```
/**
 * Load function.
 */
function load_node($nid) {
  return db_fetch_array(db_query("select * from {node} where nid='%s'", $nid));
}

/**
 * Delete function.
 */
function delete_node($nid) {
  return db_query("delete from {node} where nid='%s'", $nid);
}
```

Hooks, callbacks and their relationship to CRUD.

- Instead of having just nodeapi, or userapi, the data model provides the capability of adding in any callbacks to each of the actions you want.

- Hooks, in turn, are also automatically registered in the callback structure, and can be shifted around using hook_model_alter.

- Additionally, instead of calling load_node directly, one would call a function called drupal_load('node', $nid), which would then automatically trigger all the callbacks for that model, and provide you with your loaded node.

- This allows you the freedom to (for instance), and an image field to the term admin screen, and have it automatically load whenever you call drupal_load ('term', $termid).

But What About Performance, With All These Hooks!

- Caching caching caching.
- Because we now have the data model, and we know which fields the object can be identified with, we can implement both a session wide and a site-wide object cache (where either makes sense).
- In drupal_load(), we will simply try to load from the object cache when we have an ID, and the ‘cacheable’ property is true.
- In drupal_save, we will be able to invalidate the cache for any thing that has changed, in a more controlled way than we are currently able to.

Enter Relationship API.

- One of the other very important features of this hook, would be the ability to do relationships, between different objects. Due to us having enough context, we can implement a far cleaner relationship API than previously possible.
- The prototypical example of this would be the node reference and user reference fields of CCK.
- Relationships should be able to be created between any of the different data models.
- A lot of the previous implementations of data models we have already implement similar features.
- We have a great example of this already implemented in CakePHP.

Relationships, or how to fix file api.

```
function model_node() {
  $model[‘nid’] = drupal_field(‘id’);
  $model['title'] = drupal_field('title');
  $model['content'] = drupal_field('content');

 // File is an existing data model, with a full set of CRUD functions.
// The relationship is automatically keyed to ‘node:$nid’.
  $model[‘uploads’] = drupal_has_many(‘file’);

 // Image is also a model, but it actually uses file’s CRUD functions, and adds
 // an extra table for image data.
 $model[‘head_image’] = drupal_model(‘image’);
  return $model;
}
```

Views.

- No. Not those kinds of views. We're unfortunately in a situation where the more correct term for this is taken by the well awesome, and much beloved contributed module. Such is life, but I will be referring to views in the context of the MVC pattern in this presentation.

- Views are split into 2 (possibly more) types. Edit and Display views.

- Views extend the data model, in that additional properties (such as which widget to use to render the field), are added to the field data structure.

- Each view type, can automatically be created from an existing data model, but it is possible to make multiple derivative views of the same data model.

View example.

```
// File is an existing data model, with a full set of CRUD functions.
// The relationship is automatically keyed to ‘node:$nid’.
function view_node($model) {
  $view['title'] = drupal_widget($model['title']);
  $view['content'] = drupal_widget($model['content']);
  $view['group'] = drupal_widget('fieldset', array('title' => 'uploads',
'description' => 'my description'));
  $view['group']['uploads'] = drupal_widget($model['uploads']);

  return $view;
}
```

Derivative views.

```
// File is an existing data model, with a full set of CRUD functions.
// The relationship is automatically keyed to ‘node:$nid’.
function view_node_summary($model) {
  $view['title'] = drupal_widget($model['title']);
  $view['head_image'] = drupal_widget($model['head_image'], array(‘widget’ =>
‘image_thumb’);
  $view['content'] = drupal_widget($model['content']);

  return $view;
}
```

What about the theme layer?

- The drupal_render() function can be used to theme both fields and widgets.
- You could also completely disregard the view structure component, and do the tables and fieldsets yourself, which is actually incredibly similar to how Ruby on Rails does it.
- The same way that drupal_render automatically constructs the markup for you, the model can automatically be constructed into a view structure. This is the basis of the ‘scaffolding’ form generation.
- The function you use to do this, is the drupal_widgets() function, which can be used to place multiple elements.

Other notes about views.

- No sorting of fields by default. All fields will be displayed as they are added.
- The primary reason for this is performance, as the weight sorting is one of the slowest parts of the forms api, and with the additional use of drupal_render for other output, we stand to run into considerable performance issues.
- There would preferably be a set of API functions which allow you to manipulate the data model and turn them into more powerful views.
- Examples of such functions are drupal_sort() and drupal_table(). These functions should be as simple as possible, and help reduce the amount of code needed to do common operations, and things like needing to have a theme function just to add a table to a form.

A note about ‘inheritance’

```
/*
* Example of creating a settings form. This ‘inherits’ the edit and delete
* handlers of the standard settings forms. This is essentially similar to how
* system_settings_form currently works
*/
function model_mymodule_settings($model) {
  $model = drupal_model(‘settings’);
  $model[‘mymodule_title’] = drupal_field(‘title’);
  // the settings page ‘has a’ icon object. The relationship key
  // will automatically be set to ‘mymodule_settings’
  $model[‘mymodule_icon’] = drupal_model(‘icon’);
  return $model;
}
```