

CMFFormController:

Everything You Ever Wanted To Know But Were Afraid
to Ask

Geoff Davis

Plone Conference, 2004

geoff@geoffdavis.net



What is CMFFormController?

- / *NOT* a way to autogenerate forms!
 - (You're thinking of Formulator – that's a different talk)

What is CMFFormController?

- | Framework used throughout Plone
 - Glue that binds forms to scripts and vice versa
 - Simplifies coding of forms / form handling scripts
 - Manages transitions between forms and scripts

What Problems Does It Solve?

- | Helps make products customizable in a way that is less likely to break when you upgrade
- | Without FormController
 - The script invoked by submitting a form is hardcoded into the form
 - The page displayed after invoking a script is hardcoded into the script
 - If you want to change what happens after a form is submitted, you have to customize the form / script.
 - Customizations can break when you upgrade!

What Problems Does It Solve?

I With FormController

- **Things that are likely to change on upgrade (contents of scripts, forms) are separated from things that are less likely to change**
- The script invoked by submitting a form can be specified in a .metadata file
- The page displayed after invoking a script can be specified in a .metadata file
- Metadata can be overridden in the ZMI

What Problems Does it Solve?

- | Provides better implementation of the Model / View / Controller (MVC) paradigm:
 - Model = Zope objects
 - View = page templates, DTML
 - Controller = CMFFormController + python scripts
- | FormController separates controller logic from views
- | Makes it unnecessary for your DreamWeaver person to know what scripts a form should call

Conceptualizing

| Old:

- my_form: invoke my_validation_script
- my_validation_script: check some stuff. on success, call my_script
- my_script: do some stuff. show my_page

| New:

- my_form: generic form
 - | Metadata:
 - validate using my_validation_script
 - If “success”, call my_script
- my_verification_script: Check some stuff. Return “success”
- my_script: Do some stuff. Return “success”
 - | Metadata:
 - If “success”, show my_page

Conceptualizing

- | Atomic units are:
 - [form + chain of validation scripts]
 - [script]
- | Return values of these units are *states*, not explicit directives to show a page / call a script
- | FormController takes care of the transitions
- | FormController implements the Controller/State design pattern (More buzzwords!)

Why All This Is Useful

I Example #1:

- By default, after you edit a Plone document, you are shown a view of the document
- Suppose, instead, you want to be taken to /index_html
- With FormController, you can make the change by modifying a single item in the ZMI
- If document_edit.py is changed in a Plone upgrade, the changes won't break your site

Why All This Is Useful

I Example #2:

- Suppose you want to add a spell checking script and a correction page before the `document_edit` script
- With `FormController` you can programmatically insert the new page + script without changing either `document_edit_form` or `document_edit`
- Forms + scripts are now like a linked list – you can chain new forms and scripts together and insert and remove things programmatically

Questions?

- | Any questions before we start coding an example?

How It's Done

- | FormController uses specially modified page templates and python scripts
 - Page Templates (.pt files) are replaced by Controller Page Templates (.cpt files)
 - Python Scripts (.py files) are replaced by Controller Python Scripts (.cpy files) and Validator Python Scripts (.vpy files)
- | If you don't use .cpt / .cpy / .vpy files, everything works as before. FormController is strictly optional!

My First ControllerPageTemplate

- | We're going to create a web application that
 - 1) Prompts a user for two integers
 - 2) Verifies that the user enters only integers
 - 3) Displays the sum of the entered integers
- | Follow along! Grab code snippets from <http://plone.org/Members/geoff/>

add_numbers form

I Step 1: Create the form

- In the ZMI, add a new Controller Page Template
- Give it ID “add_numbers” and click Add and Edit

```
<html>
  <body>
    <form action="" method="post">
      <p>Enter two integers.</p>
      First: <input type="text" name="n1" value="" /><br />
      Second: <input type="text" name="n2" value="" /><br />
      <input type="submit" name="submit" value="Submit" />
    </form>
  </body>
</html>
```

add_numbers form

Now make changes to the form for CMFFormController:

- 2) Make the form submit to itself
- 3) Add the hidden value form.submitted. This is a flag that tells the form that it needs to process the values in the request.

```
<form
  tal:attributes="action python:here.absolute_url()+ '/' +template.id"
  method="post">
  <input type="hidden" name="form.submitted" value="1" />
  <p>Enter two integers.</p>
  First: <input type="text" name="n1" value="" /><br />
  Second: <input type="text" name="n2" value="" /><br />
  <input type="submit" name="submit" value="Submit" />
</form>
```

add_numbers_validate

- | Step 2: Create a validator for the form
 - In the ZMI, add a new Controller Validator
 - Give it ID “add_numbers_validate” and click Add and Edit
 - Enter title “Validate add_numbers form”
 - Enter parameters “n1, n2”

add_numbers_validate code

Make sure that a value was entered for the first integer:

```
if not n1:  
    state.setError('n1', 'Please enter a value')
```

`state` = built-in object that carries the state for the current action

- Holds the status of the validation (e.g., success or failure)
- Holds error messages
- Holds status messages that should be displayed after validation

```
else:  
    try:  
        n1 = int(n1)  
    except (ValueError, TypeError):  
        state.setError('n1', 'Please enter an integer')
```

`state.setError` method:

- First parameter = id of variable associated with error
- Second parameter = error message

Repeat tests for `n2`

add_numbers_validate

Validators return a status value via the state object

- | Typical status values are 'success' and 'failure'
- | Validators *must* return the state object
- | Default initial status is 'success'
- | Validators can be chained together. Status is passed along the chain via the state object.

```
if state.getErrors(): # an error has occurred
    state.setStatus('failure')
    return state.set(portal_status_message=\
        'Please correct the errors shown')

return state # no errors - always return the state
```

add_numbers_validate

```
if not n1:
    state.setError('n1', 'Please enter a value')
else:
    try:
        n1 = int(n1)
    except (ValueError, TypeError):
        state.setError('n1', 'Please enter an integer')

if not n2:
    state.setError('n2', 'Please enter a value')
else:
    try:
        n2 = int(n2)
    except (ValueError, TypeError):
        state.setError('n2', 'Please enter an integer')

if state.getErrors(): # an error has occurred
    state.setStatus('failure') # set status to failure
    return state.set(portal_status_message='Please correct the errors shown')

return state # no errors -- always return the state object
```

Wiring things together

- | Now we need to tell FormController that `add_numbers_validate` is a validator for the `add_numbers` form
- | In the ZMI, go to the `add_numbers` form
- | Click the Validators tab
- | Add a default validator:
 - Context_type: Any
 - Button: leave blank
 - Validators: `add_numbers_validate`

What is all this stuff?

I **Default Validator vs Validator Override**

- Default validators: Validators created by a Product creator / validators specified in a .metadata file
- Override: place for changes made by a 3rd party product

I **Context type:** Lets you specify different validators depending on the context object's type. Especially useful in Archetypes, since the same base_edit form is used for editing all context types.

I **Button:** Lets you specify different validators depending on the button pressed.

I **Validators:** List of scripts used to validate the form. Scripts are invoked in order.

Testing the Form

- | Go to the add_numbers form
- | Fill in 2 numbers, submit
 - Exception!
- | Fill in some non-integers, submit
 - We get the form back, but no error messages
- | We have a little more work to do

Actions

- | Need to tell FormController what to do after validation
- | In the ZMI, go to the add_numbers form
- | Click the Actions tab
- | Under default action, enter
 - Status: success
 - Context type: Any
 - Button: (leave blank)
 - Action type: traverse_to
 - Action argument: string:add_numbers_script

What is all this stuff?

I **Default Action vs Action Override**

- Default actions: Actions created by a Product creator / actions specified in a .metadata file
- Override: place for changes made by a 3rd party product

I **Status code**: Lets you specify different actions depending on the status code returned by a script / form validators

I **Context type**: Lets you specify different actions depending on the context object's type.

I **Button**: Lets you specify different actions depending on the button pressed.

I **Action type**: Should we traverse to the next form/script (and preserve the contents of the REQUEST) or redirect to it?

I **Action argument**: A TALES expression that specifies the next thing to do.

What we have done so far

- | We have told FormController:
 - when validation succeeds, call `add_numbers_script`
- | By default, 'failure' status results in traversal to the form submitted.
 - Can specify this explicitly if you want
 - Can override if you need to (e.g. errors send one to a special error page)

Showing Error Messages

- | Now we need to modify the form so that it displays any error messages generated by validation
- | The state object is passed to the form in options. Usually the only thing we need from the state object is the error messages
- | Get the messages as a dictionary using the following TALES expression:
 - options/state/getErrors

add_numbers form

```
<p tal:define="msg request/portal_status_message|nothing"
  tal:condition="msg"
  tal:content="msg" />
<form
  tal:define="errors options/state/getErrors"
  tal:attributes="action python:here.absolute_url()+ '/' +template.id"
  method="put">
  <input type="hidden" name="form.submitted" value="1" />
  <p>Enter two integers.</p>
  <p tal:define="err errors/n1|nothing" tal:condition="err"
    tal:content="err" />
  First: <input type="text" name="n1"
    tal:attributes="value request/n1|nothing" /><br />
  <p tal:define="err errors/n2|nothing" tal:condition="err"
    tal:content="err" />
  Second: <input type="text" name="n2" value="" /><br />
    tal:attributes="value request/n2|nothing" /><br />
  <input type="submit" name="submit" value="Submit" />
</form>
```

Testing, testing, 1, 2, 3...

- | Now try testing the `add_numbers` form
 - You should see error messages if you enter bad numbers
 - You should get an error if you enter integers (we haven't written `add_numbers_script` yet!)

add_numbers_script

- | Step 3: Create a script to process the values submitted
 - In the ZMI, add a new Controller Python Script
 - Give it ID “add_numbers_script” and click Add and Edit
 - Enter title “Process add_numbers form”
 - Enter parameters “n1, n2”

add_numbers_script

- | First convert the form values to integers

```
n1 = int(n1)
n2 = int(n2)
```

- | Next store the value in the state object

- Keyword arguments set in the state object get passed along,
 - | in the REQUEST if you do a traversal, or
 - | in the query string if you do a redirect
- ```
state.set(n=n1+n2)
```

- | Specify the next action.

- Action can be specified in the action tab
- `state.setNextAction` provides a shortcut

```
state.setNextAction('traverse_to:string:add_numbers_results')
```

- | Return the state (always return the state!)

```
return state
```

# Showing the Results

- Step 4: Create a page to show the results
  - In the ZMI, add a new Page Template
  - Give it ID “add\_numbers\_results” and click Add and Edit

```
<html>
 <head>
 <title tal:content="template/title">The title</title>
 </head>
 <body>
 <p tal:content="request/n|nothing" />
 </body>
</html>
```

# Testing

- | In the ZMI, click on the add\_numbers form
- | Click the test tab
- | Enter some non-integers
  - Should get nice error messages
- | Enter some integers
  - Should get a sum

# Adding Complexity

- | Task: Add a second button that computes a difference of two numbers.

- | Method:

- 1) Rename the existing button and add the second

```
<input type="submit" name="form.button.add"
 value="Add" />
```

```
<input type="submit" name="form.button.subtract"
 value="Subtract" />
```

- 2) Specify validators and actions for buttons "add" and "subtract"

# Gotcha

- | In IE, you can submit a page with a carriage return. No button will register as having been pressed.
- | You need to always specify validators / action for any button. This controls what happens when a form is submitted with CR.
- | FormController will log a warning if you forget to do this in a .metadata file (Plone 2.0.4 generates lots of these warnings)

# Development on the File System

- | Don't develop in the ZMI!
- | File system procedure for FormController is very similar.
- | Use .cpt, .vpy, and .cpy files
- | Specify actions and validators in a .metadata file

# .metadata

- | Special extra sections to your metadata file

```
[default]
```

```
title=My Title
```

```
[validators]
```

```
validators=my_validator
```

```
[actions]
```

```
action.success=string:my_script
```

- | Gotcha: If you create a .metadata file that can't be parsed, it can prevent an entire skin from loading. You will see an empty directory view.

## More Details

- | Fairly complete documentation in the ZMI
- | Go to `portal_form_controller`, click on the Documentation tab