The Definitive Guide to Plone

ANDY MCKAY

Apress®

ISBN (pbk): 1-59059-329-4

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Dominic Shakeshaft

Technical Reviewer: Michel Pelletier

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, John Franklin, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Project Manager: Beth Christmas

Copy Edit Manager: Nicole LeClerc

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Janet Vail

Compositor: Susan Glinert

Proofreader: Christy Wagner

Indexer: Valerie Robbins

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 8

Managing Workflow

One of Plone's many strengths is the workflow component. Workflow fits into one of the core themes of *content management*, which is the separation of logic, content, and presentation. This chapter therefore covers Plone's workflow in detail.

The chapter starts by covering some key definitions related to workflow, as well as the key tools involved, so that you can begin to conceptualize workflows. Once these concepts are clear, I then discuss how to add and edit your own workflows.

Throughout this chapter, I reference simple changes you can make to the workflow that comes straight out of the box with Plone. I also provide a series of examples to help you perform tasks such as creating notifications, moving content, and so on. Finally, I show some of the more advanced features of workflow development and some of the useful tools that are available.

What Is Workflow?

Workflow is a chain of actions or events that occurs on something to achieve an objective. Workflow often expresses business rules that may exist. Every business has different rules and policies about tasks that must happen within that company. Examples of this include the following:

- Before an employee's time sheet is approved, it must be viewed and acknowledged by a supervisor.
- In a widget factory, for each widget assembled, users must be notified of the order and any change in the state of the widget as it passes through the factory.
- Before a Web page is published on a Web site, it must be approved by marketing, approved by the Webmaster, and translated by a linguist.

Workflow separates the logic of these business rules and standardizes the concept of thinking about these changes. By having separate logic, it's now easy for businesses to change the application to fit their business and their business rules. Often applications try to enforce a workflow on a business because the workflow is hard-coded into the application.

Understanding Workflow in Plone

Plone's workflow tool provides certain features and limitations that are key to understanding workflow in Plone. The workflow product used in Plone is DCWorkflow, which is an open-source product released by Zope Corporation. Other workflow systems are available, and some of them are being incorporated into Plone, such as OpenFlow (`http://www.openflow.it`). However, for the moment, DCWorkflow is powerful and simple enough to provide all the functionality most users will need.

DCWorkflow assumes there's one object in the system that's the target of the workflow—for example, one piece of content or one widget. It further assumes that all objects of the same type go through the same workflow. By repurposing content (see Chapter 11 for more on this), you can have similar content use different workflows.

Since the DCWorkflow system is included in Plone, there's nothing extra to install. It's represented in the Zope Management Interface (ZMI) by the `portal_workflow` object.

Conceptualizing a Workflow

Before explaining a workflow, I'll explain a few simple pieces of terminology: states and transitions.

A *state* is information about an item of content at a particular moment in time. Examples of states are private, public, pending, and draft. All workflows have at least one starting state in which all the content starts. The workflow will then move the content through a series of states, either by user interaction or by some automation process. When the content reaches an end state, it'll remain in that state for a long time (usually forever). Content may reach one or more different end states in the process of a workflow.

For that piece of content to move from one state to another, a *transition* is needed. A transition connects a starting state and an ending state. A transition can have lots of different features associated with it, as you'll see later, but for the moment, you just need to know that a transition moves content between two states. Usually a transition is triggered by some external force, such as a user clicking a button on a Web page or a script interacting with a page.

Visualizing a workflow, especially when talking about something as nebulous as content, can be a little confusing. Thinking about an everyday occurrence will help. In this case, the following example shows the workflow of my credit card bill, which I have the joy of getting every month:

1. The credit card company prepares a bill and mails it to me.

2. I get the bill and put it on my desk. Sometimes the bill sits on my desk for quite a while as I wait for the end of month. Occasionally I have to query people about certain expenditures, such as "What were those clothes you bought?"

3. Any serious queries or questions then go back to the credit card company, perhaps causing a new bill to be created (although this happens quite rarely).

4. Usually at the end of the month, when I do all the accounting, I then pay the bill.

From this, then, you can come up with some states. Looking at the previous steps, you'll see you really have no need to create different states for receiving the bill, which includes opening it and putting it on my desk. Similarly, you don't need to bother with every review that happens. Although these are all valid steps that take place, trying to make a workflow for every state would be too cumbersome. Instead, you can summarize the workflow with the following states:

- **Draft**: The credit card bill has been prepared and sent to me.

- **Review**: The credit card bill has been received and is on my desk, being reviewed.

- **Paid**: The credit card bill has been paid, put in my filing cabinet, and forgotten about forever.

Now that you've come up with the states, you can think of the changes that need to occur. For each of these states, you'll have at least one transition that occurs to move the bill from one state to another:

- **Post**: The bank sends the credit card bill.

- **Pay**: I pay the credit card bill.

- **Reject**: Something is wrong on the bill, and it isn't approved.

Figure 8-1 shows this set of transitions and states. In the figure, boxes represent states, with the state written in them. Arrows represent the transitions from one state to the next, with the name of the transition in italics.

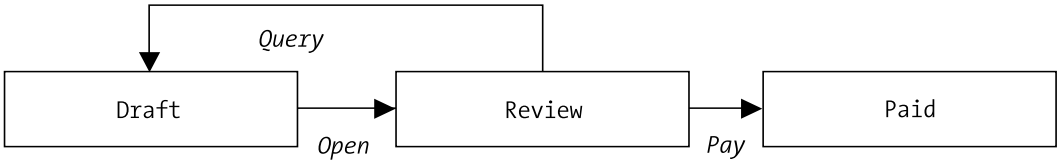


Figure 8-1. A simple state machine for paying credit card bills

You've now extracted this business process of paying a credit card bill into a workflow. The next step is to think about roles and security for this credit card bill. This workflow now contains the business logic for an application for processing credit cards.

Understanding Roles and Security in Workflow

In any complicated system, you'll have users of all roles and groups. These roles give Plone a large amount of flexibility with security, but they also can make it more complicated. Chapter 9 covers security, local roles, and groups, but this section covers some key points about how these topics relate to workflow.

When a piece of content moves from one workflow state to another, the workflow process can change the security settings on that content. The security settings determine what user can perform what action on what piece of content. By manipulating the security settings through workflow, you can cause the security to change on a piece of content through its life cycle. Users from static systems or Zope often get confused because in Zope, all pieces of content have the same security settings throughout their life cycle.

Returning to the credit card example, you can infer the security settings for the credit card bill. One way to represent this is to produce a table that expands the security in general terms for the transitions that can occur at each of the various states, as shown in Table 8-1.

Table 8-1. The Transitions and Entities That Can Make Them

State	Me	Bank
Draft		Post
Review	Pay, Reject	
Paid		

At this stage in Table 8-1, you've seen the transitions and who can make them. You haven't thought about the access that each user has to perform an action on an object at each point. For example, at which point can someone edit the bill, and when can it be viewed? These are called *actions* in Plone terminology, as shown in Table 8-2. I hope that only I have access to my own credit card statements! Likewise, at any stage, the bank is able to view the credit card bill and answer queries on it.

Table 8-2. The Actions and Entities That Can Make Them

State	Me	Bank
Draft		View, Edit
Review	View	View
Paid	View	View

Actually, as it turns out, I can't edit my credit card bill; only the bank can. I can send back my credit bill by rejecting it, but the bank is unlikely to want my edits. In this situation, assume the bank is the owner of the credit card bill. This demonstrates a concept called *ownership*. I may have several credit card bills from several banks, and in each case you can think of the bank as the owner. Each bank owns its own credit card bills, but Bank A isn't the owner of Bank B's bill. Table 8-3 combines the transitions and actions, changing the terms *Me* and *Bank* to *Payee* and *Owner*, respectively.

Table 8-3. The Transitions and Actions Combined, Plus the Roles of People

State	Payee	Owner
Draft		Post, View, Edit
Review	Pay, Reject, View	View
Paid	View	View

Of course, this is a rather contrived example, but it illustrates how you can apply workflow to basic states. More transitions can occur here—for instance, I'd be more than happy for someone else to pay my credit card bill for me—but that's so unlikely that you shouldn't add it to the workflow or security.

Before showing how to create and edit workflows, I'll now show you the default workflows that ship with Plone.

Introducing Plone Workflows

Plone ships with a set of default workflows for your Plone site. These workflows provide a logical way of moving content through a Plone site. A standard Plone site ships with two workflows: the default workflow and the folder workflow. The following sections present each of these in turn.

Default Workflow

Chapter 3 covered the default workflow and the default settings when publishing content. I discussed the security and settings for each state in the workflow. However, a picture is worth a thousand words, so Figure 8-2 shows the workflow state.

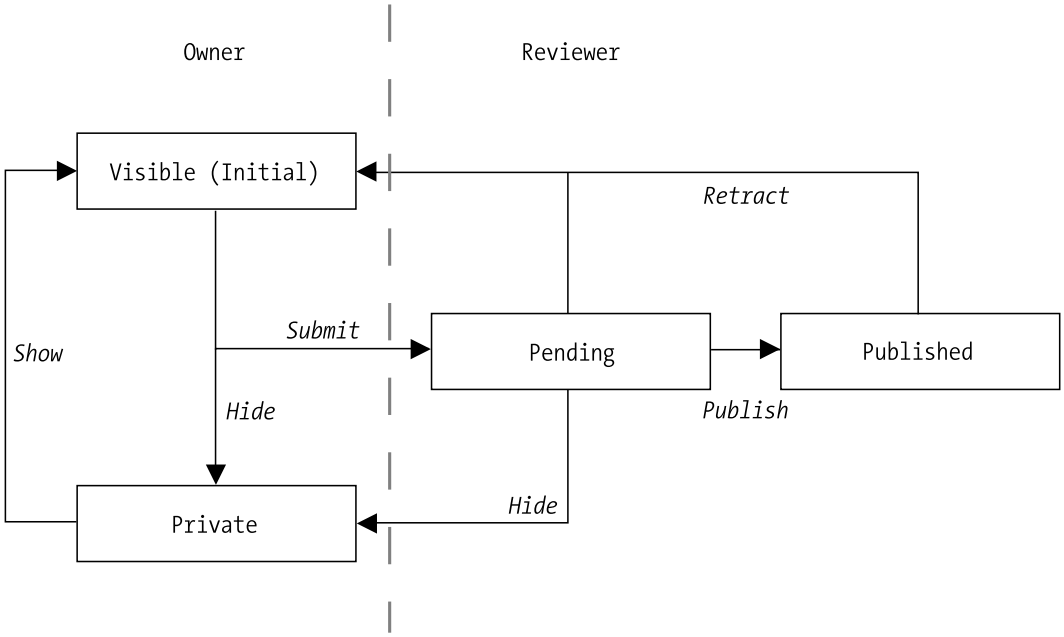


Figure 8-2. The default workflow for content that comes with Plone

Figure 8-2 shows the main states and the transitions. This figure has a gray dotted line that represents a sort of security divider. To the left of the line is where owners of the content usually interact with the content. To the right of the line is where reviewers usually interact with the content.

NOTE The owner of the content is the person who created the content originally. An owner is one particular member of a Plone site. Although many members exist in a Plone site, only one person can be the owner of a piece of content in a Plone site. Because the owner role is calculated when an object is created, the owner role is special.

Just like with the credit card example, an associated set of permissions exists for the default workflow. Table 8-4 outlines all the permissions and the states.

Table 8-4. The Default Workflow Permissions

State	Anonymous	Authenticated	Owner	Manager	Reviewer
Pending	View	View	View	Edit	Edit
Private			Edit	Edit	View
Published	View	View	View	Edit	View
Visible	View	View	Edit	Edit	View

* View refers to the following permissions: Access Contents Information and View
* Edit refers to the following permission: Modify Portal Content

As you can see from Table 8-4, by default only when content is in the private state is it truly hidden from everyone else. When content is in the published state, only the manager can edit it. Later in the "Editing Permissions" section, I'll show you how to change these permissions easily through the Web.

Folder Workflow

I also discussed the folder workflow in Chapter 3, when I covered publishing content with you. However, as I noted in that chapter, no pending state exists for folders. Instead, you have a slightly simpler workflow, as shown in Figure 8-3.

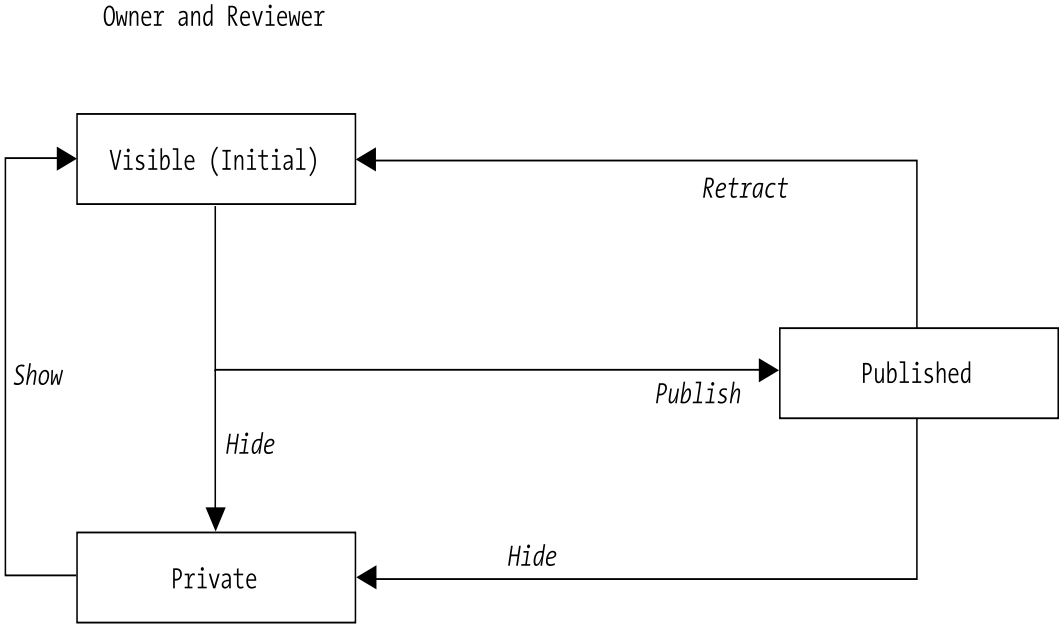


Figure 8-3. The folder workflow for content that comes with Plone

Other Workflows

Numerous workflows are available to a Plone site, including private workflow, community workflow, one-step publication workflow, and so on. ZopeZen comes with a workflow, and PloneCollectorNG also comes with a workflow. DCWorkflow comes with four workflows.

Currently, two workflows come in the PloneWorkflows product in the collective project on SourceForge (`http://sf.net/projects/collective`): the community workflow and one-step publication workflow. The community workflow is similar to the Plone workflow, with a few changes. The one-step publication workflow has two states: private and published.

At the moment, you have no easy way to install and uninstall workflows, and you have no real easy way to transition content between one state and another. For example, if you install the one-step publication workflow into an existing state, you also need to fix the states for all objects and move them into one of the new states. In this case, it's probably simple—everything in a published state should stay as it is, and everything else should move into the private state.

Adding and Editing Workflow

Now that I've discussed the default workflow, I come to the key point that's probably most on your mind: How can you change the defaults? Well, as with most of Plone, you can add, edit, and delete all workflow through the ZMI. The tool that controls workflow is `portal_workflow`. In the following sections, I cover how workflows are assigned and then go through all the settings for a workflow in detail.

Setting Workflows to a Content Type

After clicking *portal_workflow*, you'll see a list of workflow assignments. A feature of DCWorkflow is that each content type has one and only one workflow assigned to it; Figure 8-4 shows these assignments.

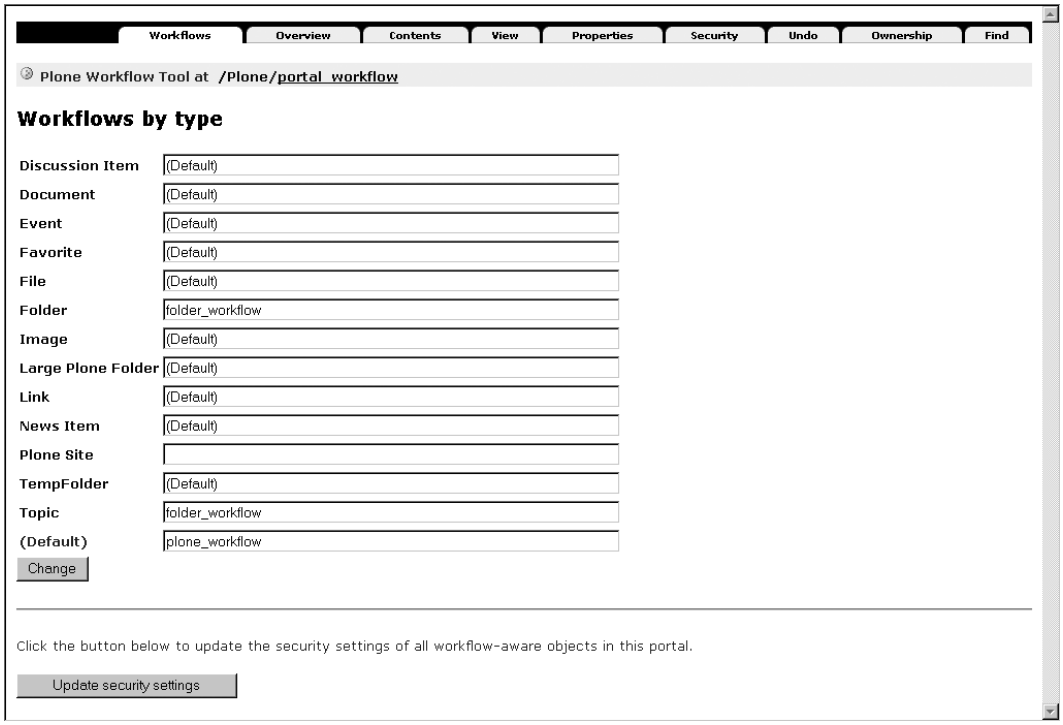


Figure 8-4. The list of workflow by type

On this page you'll see a list of each content type and the workflow that has been applied to it. If a workflow isn't specified (in other words, the value is blank), then no workflow is applied. As an example, the default for the Portal Site type is blank. You really don't want to try transitioning the Plone site itself, just the objects in it. If the value is (Default), the default workflow at the bottom of the page is applied to that content type. In Figure 8-4, for topic and folders, the `folder_workflow` workflow is used, and for all other content types, `plone_workflow` is applied. The names of the workflow refer to the name of workflow objects imported or created inside the workflow tool. For more information on the workflows available, select the Contents tab. This opens a list of workflows that have been loaded into the system, as shown in Figure 8-5.

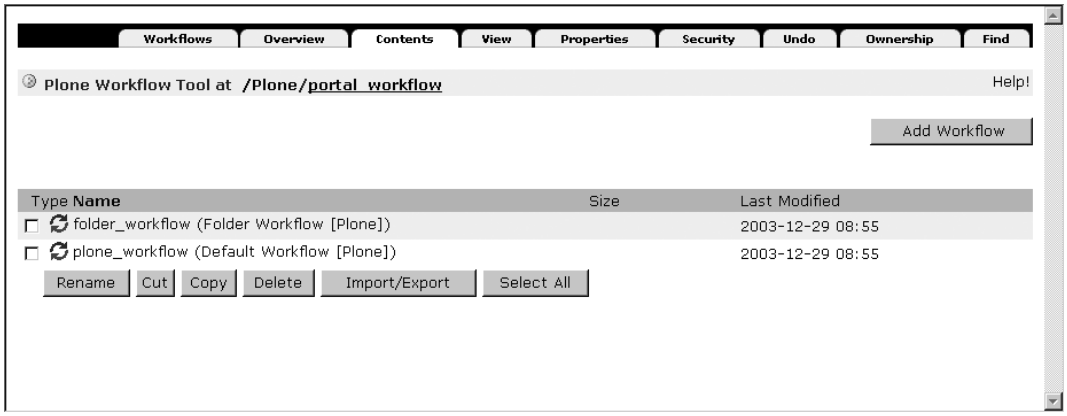


Figure 8-5. Workflows available

You can add workflows by clicking the Add Workflow button. This opens a list of the workflows available; to create a workflow, select a workflow type and enter a workflow name. To create workflow that's empty but that's configurable through the Web, select `dc_workflow` and enter an appropriate name; for example, enter **my_workflow**.

Editing a Workflow

From the Contents tab, you can click a workflow to access the management screens for that workflow: all the states, transitions, and associated features. The series of tabs across the top of the page outlines the functionality of a workflow

quite well: States, Transitions, Variables, Worklists, Scripts, and Permissions. I'll run through each of these tabs and some of the other options available. Unless otherwise mentioned, all the following tabs are accessible from this main workflow page.

Creating or editing workflow can require lots of clicking and can be a little confusing. If you're a developer keen on using the file system, then you can do all this from Python if you want—I cover this for you later in this chapter in the "Writing a Workflow in Python" section.

Setting the Initial State

To set the initial state, go to the States tab and check out the states available; next to one of the states you'll see an asterisk, as shown in Figure 8-6.

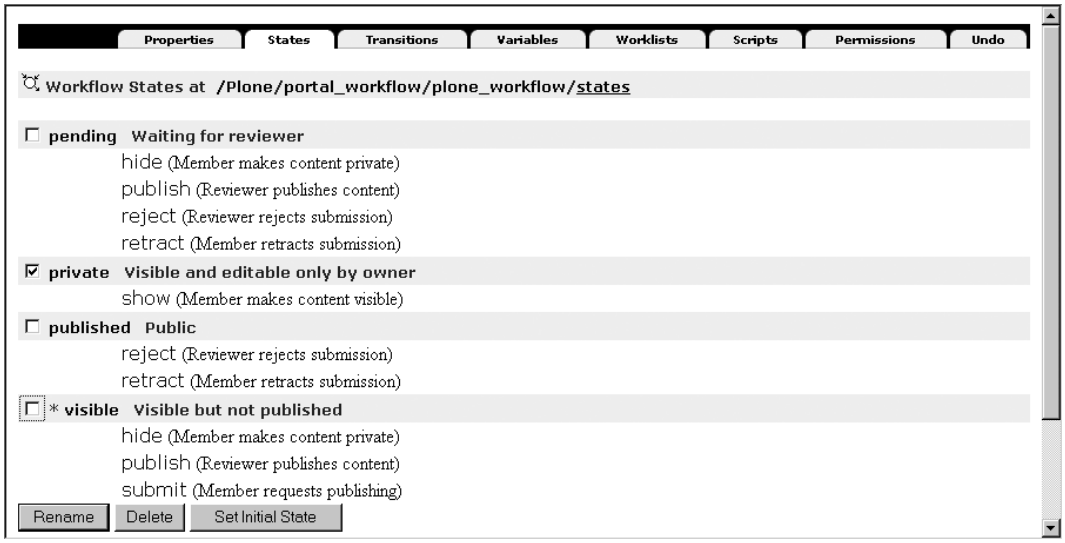


Figure 8-6. Setting the initial state for this workflow

You set the initial state for your workflow on this page by checking the box next to the state and then clicking Set Initial State. All content that uses this workflow will be created with an initial state. Any content that has already been created will remain in its initial state; changing the state afterward won't change that state. You can set only one initial state for each workflow.

How Can You Set the Initial State As Private?

On some sites it may make sense for content to not show up at all or be accessible to users other than administrators and owners only after it has been completed. The best way to do this is to set the default state for the object to something that provides this security—for example, private. In the private state, only reviewers, managers, and owners can actually see the item.

To set the default state to private in the ZMI, click *portal_workflow*, and select the Contents tab. Next, click *plone_workflow*, select the States tab, and then select the default state by checking the box next to *visible*. Finally, click the Save Changes button. New content will now be in the private state and not accessible to the general public.

Editing States

The States tab lists the states that are present in this workflow. At the beginning of this chapter, I explained that a state represents an object at a particular point in time. Each state has an ID that's unique; this is usually a simple verb such as *pending* or *published*. To add a state, enter an ID and click Add at the bottom of the page.

You'll also see the following options:

Title: The title of the state is displayed in your Plone site and is a user-friendly version of the state.

Description: The description of the state is a long description of the state. This isn't currently shown to users but may be in the future.

Possible transitions: This lists all the possible transitions that can occur from the state. This list will show only if you actually have a transition in the system. Simply select the transitions that need to occur on this state. By selecting a transition for this state, you're selecting the start point for this transition to be this state.

To alter a state, enter the changes and then click Save to commit the changes. The Permissions tab will open with the permissions that will be applied to an object while it's in this state. This may mean changing the permissions on an object when it transitions into that state. The form is rather self-explanatory; to enable an anonymous user to view the object, check the boxes that correspond to View and Anonymous and click Save Changes, as shown in Figure 8-7.

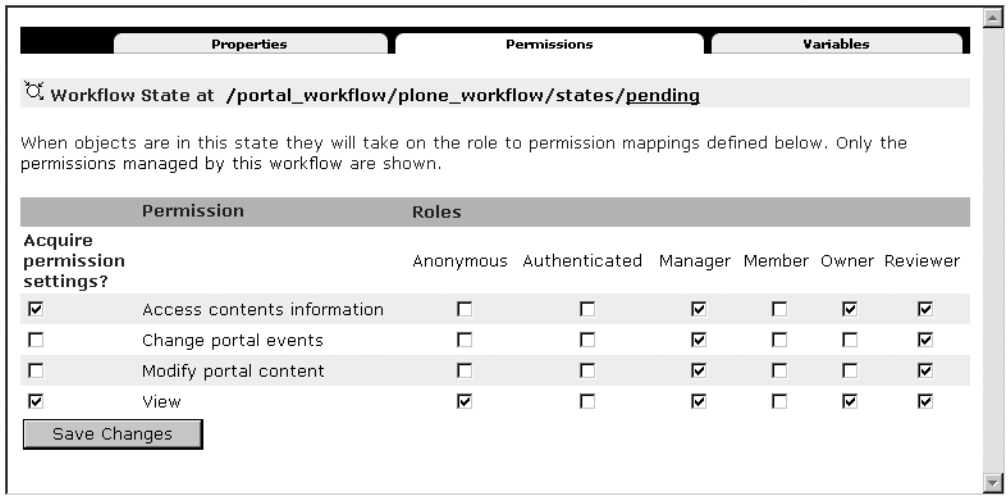


Figure 8-7. State permissions page

If you change the permissions for a particular workflow state, you've created an issue that needs resolving. Any existing content in that state won't have the new workflow permissions set on it. The content will have the old workflow permissions, and you'll need to update them. When you've finished making all your changes, go to the root workflow page and click Update Security Settings, as shown in Figure 8-4. Performing that update may take a while depending upon the number of objects to be altered.

The Variables tab allows you to assign a value to a variable when the object is in this state. The workflow determines the list of variables available to each state. For more information on these, see the "Editing Variables" section.

Editing Transitions

The Transitions tab lists the transitions that will occur in this workflow. In the beginning of this chapter, I showed you how a transition represents the changes that will occur to the object. Each transition has a few variables that are shown on the summary page. To add a transition, enter an ID and click Add at the bottom of the page, as shown in Figure 8-8.

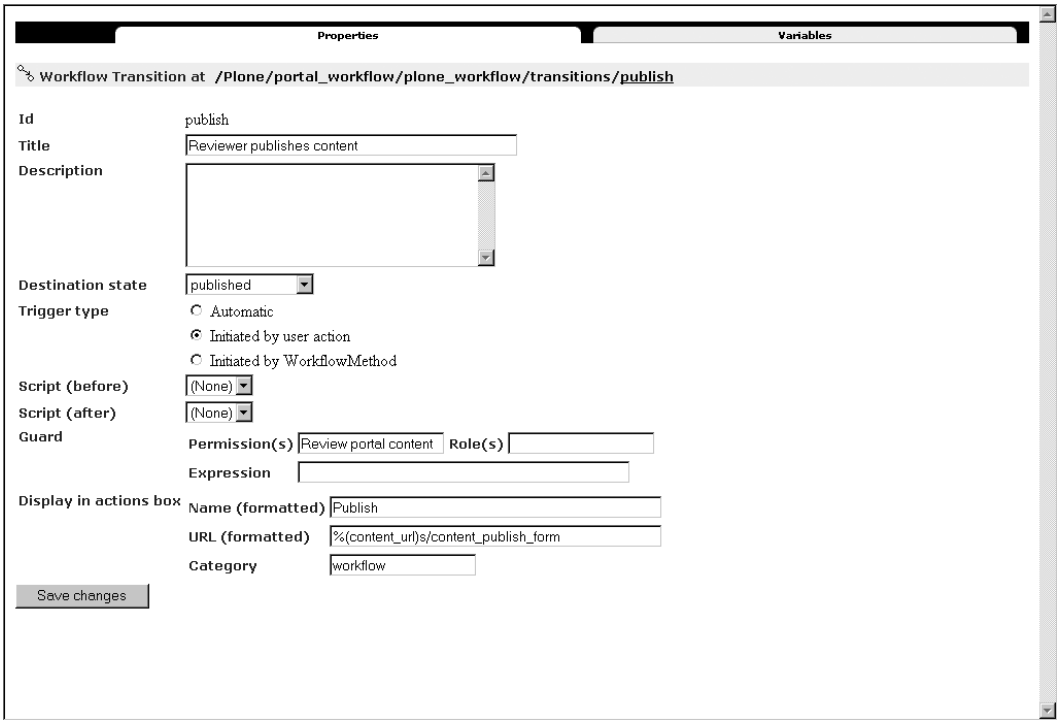


Figure 8-8. Transition details page

If you now click a transition, you'll open the following details for that transition:

Title: This is the title for this transition.

Description: This is the detailed description for this transition.

Destination state: This is the state that will be the target for this transition. The initial source state is defined by assigning the transition to the state.

Trigger type: This indicates how the transition will be triggered. *Automatic* means that this will happen as soon as it moves into this state. *Initiated by user action* is the most common choice and means a user has enacted the transition by clicking a link.

Script (before): This runs this script before this transition occurs.

Script (after): This runs this script after this transition occurs.

Guard: This is the security for this state (explained shortly).

Display in actions box: This is how this transition will be displayed in Plone. Entering a value here also ensures that the transition will be entered as an action. You can then get this transition as an action by querying for actions.

Of these values, the destination state is quite interesting. Although I've already mentioned that transitions normally change state, this isn't required. Because each transition can run scripts and write something into the history, it can be useful sometimes *not* to change state. For an example of this, see the "Using Workflow to Track Changes" section later in this chapter. If your transition does change the state, then select the new state as the destination state.

A transition can have multiple starting points but only one destination; if you need multiple destinations, you'll have to make multiple transitions. You can specify scripts to run before or after this transition. Two common examples are moving an object in workflow and sending an e-mail notification. The "Common Tasks and Examples" section covers both of these examples.

Before any transition can be executed, a security guard checks the entire transition to ensure that the user running the transition has the right to do so. The guard has the following three components:

Permission(s): These are the required permissions. Multiple permissions should have a semicolon (;) to separate them.

Role(s): These are the required roles. Multiple roles should have a semicolon (;) to separate them.

Expression: This is a workflow expression. For more information on this, see the "Editing Workflow Expressions" section later in this chapter. For each value specified, the guard must evaluate as true before continuing. If a test of any of the values fail, then the transition won't execute. Usually you'll find most guards have only one or two values specified.

Editing Variables

The Variables tab lists the variables that will be created and changed in the workflow. I haven't discussed variables much with you up to this point; instead, I've focused on states and transitions. This section covers variables.

It isn't always possible, and I don't recommend that you try, to encapsulate all the information you'll need in a workflow within just states and transitions. Instead, you can use variables to store some workflow-related information. For example, in the credit card bill example, the bill could be paid by several methods (Internet banking, check, and so on). You could store the amount method ($100, for example) in a variable. Should the bill be rejected or altered, that amount would be updated. The point of a variable is to have something that changes between each state and transition.

So, returning to the main workflow page, click the Variables tab to get a list of all the variables. To add a variable, enter a variable ID and click Add at the bottom of the page. To determine what state an object is in at any time, DCWorkflow

stores the current state in a variable on the object. The default name of that variable is `review_state`.

NOTE If you need to change this because it conflicts with another name, you can do so at the bottom of that page. However, doing this will cause all your current objects to lose their state, so be careful about changing that value.

Each workflow variable has the following properties:

- **Description**: This is the variable description.
- **Make available to catalog**: These variables will be placed in a list exposable to the catalog. This doesn't add indexes or metadata to the catalog; you still have to do that manually.
- **Store in workflow**: This determines if the information is to be stored in the workflow or on the object.
- **Variable update mode**: This determines when to update the variable.
- **Default value**: This determines a default value as a string.
- **Default expression**: This is the default value as an expression. If this is present, it'll be used instead of default value (for more information, see the "Editing Workflow Expressions" section later in this chapter).
- **Info. guard**: These are security settings for accessing this variable. These guard settings are similar to the guard settings for a transition; however, the guard occurs when accessing the variable here.

Editing Worklists

The Worklists tab provides access to all the worklists that are assigned in this workflow. A *worklist* is a method of querying the workflow for information about the numbers of objects in that workflow. For example, I'd like to be able to easily ask the workflow for all the outstanding credit card bills I have.

To add a worklist, enter an ID and click Add. Each worklist has the following properties:

- **Description**: This is a description of the worklist.

- **Cataloged variable matches**: This is the value that the worklist must match to be added in this worklist. The variable matched is the workflow state variable given in the variables list (the default variable name for this variable is `review_state`).

- **Display in actions box**: This is information to display on the user interface. Entering a value here also ensures that the transition will be entered as an action. You can then get this transition as an action by querying for actions.

- **Guard**: This is a guard for accessing this worklist.

Returning to the credit card example, if I wanted to know all the credit card bills that need reviewing by me, then I could place this information in a worklist. First, the variable `review_state` would contain the current state for each item. All the credit card bills that need reviewing would be in the `review` state. Second, I'd add a worklist called `review_queue`, and the value for variable would be `pending`. I could now ask the worklist for all the items in the `review_queue`.

Although a worklist is a convenient way of storing this information, Plone doesn't use them. Instead, Plone uses ZCatalog directly to query objects that are workflowed. Since the DCWorkflow worklist uses the catalog tool, the end result is the same.

Editing Scripts

The Scripts tab lists the scripts that are available to this workflow. This list is actually a standard folder in the ZMI, and you can add almost anything there. Since the main reason you'd want to do this would be to add a script to perform advanced handling of transitions, you should add only Script (Python) objects here.

To add a script from the Scripts tab, select Script (Python) from the Add drop-down menu, and give the script an ID. The script is passed to one and only one object, which is the base workflow expression object; for more information on this object, see the "Editing Workflow Scripts" section later in this chapter. For example, if you need to access the actual object in the workflow, you can use a Python script such as the following:

```
##parameters=state_change
obj = state_change.object
```

What happens in this script is up to the developer—you can run almost anything here. You can trigger events, and you can access other workflows and

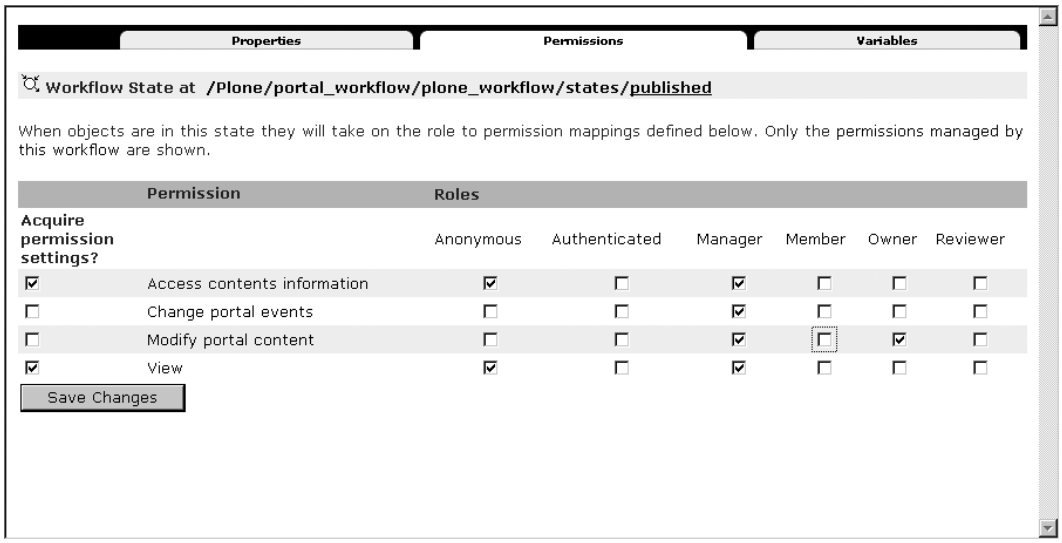
transitions. For some example scripts, see the "Sending E-Mail Notifications" and "Moving Objects" sections later in this chapter. When the script executes, it will execute as the user who initiated the transition. You could assign proxy roles on the script if it needs to happen as someone else. Returning to the transitions, you can assign this script to any number of transitions in the *script (after)* and *script (before)* settings. You can run the script either before or after a transition.

Editing Permissions

The Permissions tab lists the permissions that are managed by this workflow. You've seen these permissions already when examining the states. You set the list of permissions manageable in those states in this tab. To add a new permission, select the permission from the drop-down box and select Add.

How Can You Edit a Published Document?

Well, you can't edit a published document in the default workflow unless you have the manager role. If you allow the owner of the document to edit it, then you really should review it again. However, this seems to be a common request and is a trivial thing to change. In the ZMI, click *portal_workflow*, and select the Contents tab. Then click *plone_workflow*, and select the States tab. Finally, click *published* and then select the Permissions tab. Check the box that corresponds to allowing the owner to modify portal content.



Click Save Changes to save your permissions. Because you've updated the security settings, you'll have to click *portal_workflow*, select the Contents tab, and click *Update security settings*. This will update all the objects in your site and ensure that your permissions have been applied to existing objects. Now owners can edit their documents while they're in the published state.

Editing Workflow Scripts

Scripts are an opportunity for the developer to perform some logic upon a transition. That logic can be almost anything you want. You could be checking that some conditions have been performed (for example, is the document spell checked?) or that some special actions have been performed. When the object is transitioned, the script will be called.

When a script is called, one extra parameter is passed to that script. That extra parameter provides access to all sorts of transition-related elements and attributes. That parameter is called the `state_change` parameter, and it has the following attributes:

status: This is the workflow status.

object: This is the object being transitioned in the workflow.

workflow: This is the current workflow object for the object being transitioned.

transition: This is the current transition object being executed.

old_state: This is original state of the object.

new_state: This is destination state of the object.

kwargs: These are keyword arguments passed to the `doActionFor` method.

getHistory: This is a method that takes no parameters and returns a copy of the object's workflow history.

getPortal: This is a method that takes no parameters and returns the root Plone object.

ObjectDeleted(folder): This tells workflow that the object being transitioned has been deleted; it takes the object to which you'd like to return the user. Pass to the exception the folder you'd like the user to be redirected to (see the "Moving Objects" section later in this chapter).

ObjectMoved(newObject, newObject): This tells workflow that the object being transitioned has moved. Pass to the exception the folder you'd like the user to be redirected to (see the "Moving Objects" section later in this chapter).

WorkflowException: This raises an expectation back to workflow and aborts the transaction (and hence the transition).

getDateTime: This is a method that takes no parameters and returns the `DateTime` object that relates to the transition.

For example, to find out what state is being transitioned to and when, the following is a Script (Python) object that will tell you just that information. This script logs the information about the transition into the log file:

```
##parameters=state_change
st = 'From %s to %s on %s' % (
    state_change.old_state,
    state_change.new_state,
    state_change.getDateTime())
context.plone_log(st)
```

TIP When you're writing a Script (Python) object, you may need to print to the log file to help with debugging. A script called `plone_log` does this, which takes a string and passes it to Plone's logging functions. Hence, calling `context.plone_log` is a useful tool for debugging.

When assigning a script to a transition, you have two choices: `before` and `after`. As the names suggest, a script that's set assigned to `before` runs prior to the transition running. This is suitable for scripts that may check if something should happen prior to the transition running, such as testing that another dependent object or page has been uploaded or there are no spelling errors. The script assigned to `after` runs once the transition completes—although if at any time an uncaught exception is raised on a script, this will cause the transition to fail, the object to remain in its original state, and the exception to display to the user.

Editing Workflow Expressions

Throughout this chapter you've seen values that can be expressed as workflow expressions. For example, the value assigned to a variable is the result of a workflow expression. This expression is nothing special; it's merely a Template Attribute Language (TAL) expression with a few different variables available.

You already learned about TAL expressions in Chapter 5, so you should be familiar with these expression and all their options, such as Python, string, and path expressions.

Unlike the standard TAL expression, a few extra parameters are passed through to the namespace, relating to the particular workflow. The namespace for a workflow expression contains the following:

- **here**: This is the object being transitioned in the workflow.
- **container**: This is the container of the object being transitioned in the workflow.
- **state_change**: This is the state change object referenced in the "Editing Workflow Scripts" section.
- **transition**: This is the transition being executed, identical to `state_change.transition`.
- **status**: This is the original state, identical to `state_change.old_state`.
- **workflow**: This is the workflow for this object.
- **scripts**: These are the scripts available in this workflow.
- **user**: This is the user executing this transition.

Common Tasks and Examples

I'll now present some common tasks you can achieve easily using workflow. When a user causes a workflow transition, this transition runs using that specific user's account. In many of these examples, a normal user may not have the correct permissions to perform the task. For example, members don't normally have the right to access the list of members unless this permission has been explicitly given to them.

To solve this permission issue, where noted, some of the following Script (Python) objects have been given a slightly different role. To set a proxy role on a script, access the Proxy tab on an object and then select the user to run the script, as shown in Figure 8-9.

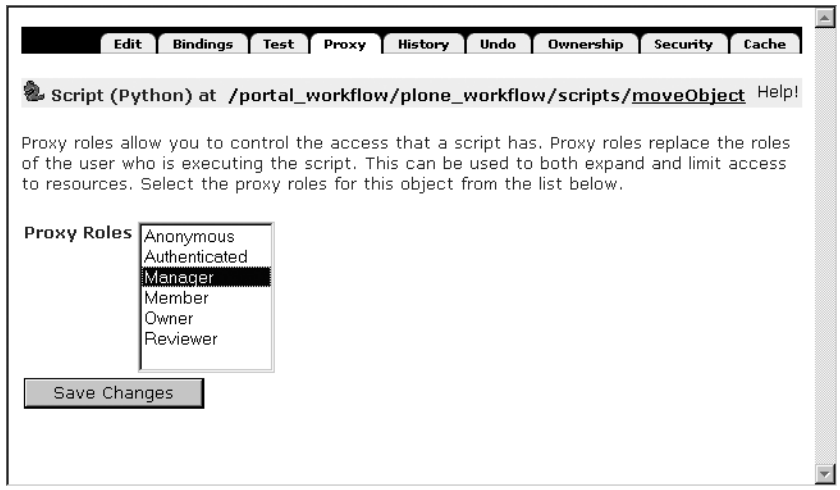


Figure 8-9. Setting proxy settings on a script

You would, of course, make sure your scripts executed with the minimum roles needed, depending exactly upon what your script does.

Example Workflow Expressions

The following are some useful examples of workflow expressions that can be used in various places.

To get the comments, or an empty string, with this transition, use the following:

```
python:state_change.kwargs.get('comment', '')
```

To obtain the title of the folder that the object is in, use this:

```
container/Title
```

To test if the old state is review state, use this:

```
python: state_change.old_state == 'review'
```

To get the user executing this transition, use this:

```
user/getId
```

So if you wanted to track who the last user to transition an object was, you could add a `last user` variable into the workflow. You do this by going to the

workflow and clicking the Variables tab. Then add the variable `last_user`. If you set the `Default expr` variable to `user/getId`, each time the object changed, that value would be stored for you.

Using Workflow to Track Changes

For one particular application a client of mine wanted to keep track of every time an item was edited and any reasons for that edit so that when auditing the item later, there would be a comment for each change. Thanks to workflow, this was quite easy to achieve.

In this case, the workflow had only one state—but actually this will work for almost any workflow. To this one workflow, a transition was added called `edit`. That transition didn't actually change the object's state; the destination state for that transition was (`Remain in state`), meaning no change occurred.

When an object is edited, a method is called to perform the change. For example, when a document is edited, the method called is `document_edit.cpy`. You can find that script by clicking *portal_skins*, clicking *plone_form_scripts*, and clicking *document_edit*. All that's needed is to add one line to that script before the last line:

```
context.portal_workflow.doActionFor(new_context,
 'edit', comment='')
```

The `doActionFor` method of `portal_workflow` performs the transition given (in this case, `edit`) for the object passed in (in this case, `context`). Each time the object is edited, that `edit` transition will fire. That will cause a line to be added to the comments list showing who edited the object, when it was added, and any comments associated with it.

When an object is edited, there are actually no comments, so to be a little more advanced, you'd have to modify the document's edit template to include a comments field. You could then access this comments list by going to the State tab, where the list of changes displays at the bottom.

Moving Objects

One useful ability is moving an object during the workflow. For example, you could move all press releases into a folder called `Press Release` each time you publish one. Content could be created and edited anywhere and then on publishing moved into that folder. The example script in Listing 8-1 moves the object being workflowed into the `Members` folder. To add this script, go into the workflow

tool in the ZMI, and select the Scripts tab. Then select Script (Python) from the drop-down box. Give the new object the name **moveObject**, and then enter Listing 8-1 into this script.

Listing 8-1. Moving an Object

```
##parameters=state_change
# get the object and its ID
obj = state_change.object
id = obj.getId()

# get the src folder and the destination folder
dstFldr = context.portal_url.Members
srcFldr = obj.aq_parent

# perform the move
objs = srcFldr.manage_cutObjects([id,])
dstFldr.manage_pasteObjects(objs)

# get the new object
new_obj = dstFldr[id]

# pass new_obj to the error, *twice*
raise state_change.ObjectMoved(new_obj, new_obj)
```

You need to do a few more things; first, assign this script to a transition. I'd normally use such a script in the publish transition. To do this, go to that transition and assign the value of `script (after)` to `moveObject`.

Second, one other small problem exists: This script moves objects into the `Members` folder. You'll probably have a better destination in mind, of course. To perform such a move, a user has to have the appropriate rights to move objects between these folders. Normally, only a manager can move objects into the `Members` folder. So you need to give the script the proxy role of manager. You can do this by clicking Scripts, clicking *moveObject*, and selecting the Proxy tab. Assign the role of manager to this script. You can find more information about security and local roles in Chapter 9.

Looking at the code, first the script gets the object and the object's ID from the transition namespace. Then it gets the source and destination folders. Then it utilizes Zope's `ObjectManager` Application Programming Interface (API) to perform the copy and paste. You could, of course, determine these folders programmatically—perhaps based on the user performing the transaction or on the type of content being moved. Finally, you get the object and pass it to an exception `ObjectMoved`.

The `ObjectMoved` exception is a special exception to DCWorkflow. By passing the new object twice as parameters into the exception, the new object will be passed up to the Plone front end. This is critical so that when the user is sent to the object in response to the change, it's to the new location of the object, not the old one. Of course, you may want to write a function that moves the function back after rejecting the object, perhaps to the member's home folder.

Another special case, and a more unusual one, is to delete an object in workflow. Usually deleting an object is an action from the containing object, so it's unusual to see in workflow. For this task, you can raise an `ObjectDeleted` exception. Listing 8-2 shows the script to perform a delete.

Listing 8-2. Deleting an Object

```
##parameters=state_change

# get the object
obj = state_change.object
id = obj.getId()

# get the parent folder, delete the object
srcFldr = obj.aq_parent
srcFldr.manage_delObjects([id,])

# raise the object deleted method and pass
# the folder you want to return to
raise state_change.ObjectDeleted(srcFldr)
```

You could call this script `deleteObject` and successfully delete objects in the workflow. Again, by ensuring the error is raised, Plone will know what to do; in this case, it takes the user to the folder containing that object.

Sending E-Mail Notifications

If you have a Web site that a user doesn't visit regularly, then putting information on the site about what has to be reviewed and when is rather pointless. You can turn workflow into a rudimentary notification system by using it to send e-mails to the users. The notification channel of e-mail is just one simple example; this could also be an instant message, a text message delivered to a phone, and so on. I'll leave other options to your imagination.

In this example, you'll send e-mail via the `MailHost` object on the server to every user who has the reviewer role in the system, telling them about a new item that has been submitted for review. This is actually a more complicated script

than the ones I've shown you so far, since it runs through a few steps: defining the variables, finding the account name of every reviewer, finding an e-mail, and sending an e-mail. Listing 8-3 shows the script.

Listing 8-3. Sending an E-Mail Notification

```
##parameters=state_change
# the objects we need
object = state_change.object
mship = context.portal_membership
mhost = context.MailHost
administratorEmailAddress = context.email_from_address

# the message format, %s will be filled in from data
message = """
From: %s
To: %s
Subject: New item submitted for approval - %s

%s

URL: %s
"""
```

This sets up the message and objects you need. Apart from the object being transitioned, you'll also need a reference to the membership tool `portal_membership` and the Simple Mail Transfer Protocol (SMTP) server via `MailHost`. The message is easily configurable to send an e-mail in any format you like.

You then use the `listMembers` method of the `portal_membership` object to get a list of members. For each member, you can then see if the reviewer role is in the list of roles for that user by calling the `getRoles` method:

```
for user in mship.listMembers():
    if "Reviewer" in mship.getMemberById(user.id).getRoles():
```

The astute reader will note that looping through every member in a Plone site could be a little slow if you have thousands of users. In the next chapter, you'll modify this script to pull the list of users from a specific group.

There's no point in sending an e-mail if you don't have a user's e-mail address, so you should check here that there's a valid e-mail first. Now all that's left is to format the e-mail and send it. For this you can use Python's string replacement functionality and pass in four parameters that correspond to the `%s`

in the `message` variable set at the beginning of the script. After this replacement, the `msg` variable will contain the e-mail you want to send. To send the e-mail, simply call the `send` method of the `MailHost` and pass through the e-mail string:

```
if user.email:
    msg = message % (
         administratorEmailAddress,
         user.email,
         object.TitleOrId(),
         object.Description(),
         object.absolute_url()
         )
    mhost.send(msg)
```

This will result in the following e-mail being sent:

```
From: administrator@agmweb.ca
To: andy@agmweb.ca
Subject: New item submitted for approval - Plone's great

We all know Plone is a great product, but with the newest release
it's gotten even better...

URL: http://agmweb.ca/Members/andym/News_Item_123
```

Appendix B shows the full listing for this script.

Using PloneCollectorNG

PloneCollectorNG is a bug tracker that's available for Plone. You'll find many other issue trackers out there, but this is the one I use and recommend for Plone. In fact, writing an issue tracker seems to be a common thing for developers to do. One of the really nice things about workflow is that it enables your users to significantly change the way an application works. As a developer, developing products hooking into DCWorkflow allows your application to remain flexible. You can find PloneCollectorNG at `http://www.zope.org/Members/ajung/PloneCollectorNG`.

The product adds a series of content types during installation; one of them is PloneIssueNG, which is an *issue* (or bug report). Rather than hard-coding exactly how the issue moves through the database, a separate workflow is assigned to the issue. That workflow contains appropriate states, transitions, variables, and worklists.

At any stage you can find out what state an object is in by calling the `getInfoFor` method of `portal_workflow`. This useful method accepts an object and the variable to be looked up. In PloneCollectorNG's workflow, that variable is called `state`, and in Plone workflow, it's called `review_state`. For example, to find the state for an object, you use this:

```
portal_workflow.getInfoFor(obj, "state")
```

You can find possible states for an object by examining the state's object directly from the workflow, like so:

```
portal_workflow['pcng_workflow'].states._mapping.keys()
```

The result of this is that if your user wants to have a simple issue-tracking system, then modifying this workflow through the Web is relatively trivial (if, when the application was developed, the workflow tools have been considered). Compare this to another popular bug-tracking system, Bugzilla, where changing a state or a transition requires hours and hours of a Perl programmer's time to find all the hard-coded references to a bug's state.

Distributing and Writing Workflow

If you've got a great workflow for your application, you have a couple of different ways to write and distribute workflow. The following sections close the discussion of workflow by presenting a couple of these options.

Writing Through the ZMI

Probably the simplest but most laborious way to write workflow is to use the ZMI. Although the ZMI drives many people crazy, it's a simple way to set up the options. Unfortunately, once you've started writing through the ZMI, you're stuck in that paradigm. In other words, there's no easy to edit or alter that workflow on the file system. I discussed editing a workflow through the Web with you earlier in this chapter, of course.

To export a workflow from the ZMI, click *portal_workflow* and select the Contents tab. Select the created workflows you'd like to export by checking the boxes on the left of the ZMI, and then click *import/export*. At the top part of the export page, select *Download to local machine,* and click *export*. A file with extension `.zexp` will be created that can be saved and redistributed. Selecting XML Format will provide a file in Extensible Markup Language (XML) format with an `.xml` extension.

If you're provided a workflow in a the `.zexp` or `.xml` format, then importing the workflow into your Plone is straightforward. Place that file in the import directory of Zope on the file system. This can be the instance home directory or the Zope directory.

Then click *portal_workflow*, select the Contents tab, and click *import/export*. At the bottom part of the page, you'll see a small form that takes an import filename. Enter the name of the filename there, and leave *Take ownership of imported objects* selected. Click the Import button to import the workflow. The workflow will now be imported and given the name specified in the export.

Writing a Workflow in Python

Using Python is probably the favorite way of programmers to write a workflow, since it can all be done in Python and easily distributed. First, make a Python module on the file system. At the top of the file, import the appropriate tools, as follows:

```
from Products.CMFCore.WorkflowTool import addWorkflowFactory
from Products.DCWorkflow.DCWorkflow import DCWorkflowDefinition
```

Second, make a function that creates the workflow. Appendix A lists the API for writing a workflow in a little more detail. But you could just cheat and look at all the great examples available in the PloneWorkflow's project in the collective (`http://sf.net/projects/collective`), or even the ones contained in Plone. For example:

```
def sample(id):
    """ Sample workflow """
    ob = DCWorkflowDefinition(id)
    ob.states.addState('private')
    ob.states.addState('public')
    # add transitions
    return ob
```

Finally, register the workflow in the system, like so:

```
addWorkflowFactory(sample,
                   id='sample_workflow',
                   title='Sample workflow')
```

This script will need to be as part of a product installation. Chapter 12 covers writing and installing products.

Now, of course, a shortcut is available, which is called `DCWorkflowDump`. This will take the code from the ZMI and dump it into a Python module for you. You can find the source code for `DCWorkflowDump` in the collective at `http://sf.net/projects/collective`, but you can also find a zip file of the code on the Plone book Web site at `http://plone-book.agmweb.ca`.

To install `DCWorkflowDump`, unzip the file and copy the directory called `DCWorkflowDump` into the `Products` directory of your Plone installation. To check that you're in the right directory, your `Products` directory should also contain a directory for `DCWorkflow`, among other things. Then restart your Plone instance.

Once you've restarted Plone, go to the particular workflow in the ZMI, and you'll notice a new tab called *dump*. Click that page to get the dump screen, and then click *Dump it!* to dump the workflow to the screen. This will take your workflow and format it in Python for you. Save that file to your product, and you now have a Python file you can manipulate. This product is a great tool because it allows you to create the workflow in the ZMI and then distribute and alter it through Python.