CHAPTER 14

SimpleSite Tutorial Part 2

Note You can download the source for this chapter from `http://www.apress.com`.

Now that you've seen a bit more of Pylons and are more familiar with how it works, I'll continue the SimpleSite tutorial. Here are the topics I'll cover in this chapter:

- Adding a comments system to demonstrate how to deal with one-to-many mappings
- Adding a tags system to demonstrate many-to-many mappings as well as how to deal with forms containing multiple check boxes
- Adding a navigation hierarchy involving sections and pages to demonstrate SQLAlchemy's inheritance features as well as a custom Routes setup

Then in Chapter 15, I'll cover JavaScript, Ajax, and YUI to show some improvements that you can make to both the visual appearance of the site and the usability.

There's a lot to cover in this chapter and you might not want to tackle it all in one go. If not feel free to continue with the other chapters and come back to this one later.

Comments System: One-to-Many Mappings

You'd like visitors to the web site to be able to leave comments about each page. The comments will consist of the date they were posted, the name of the poster, their e-mail address, and the comment itself.

I discussed one-to-many mappings in Chapter 7. Situations where one entity (in this case, a page) can have one or more instances of another entity associated with it (in this case, comments) are known as *one-to-many* mappings, and they can all be dealt with in the same way, which in this case is by having a *foreign key* in the comments table represent the ID of the page with which the comment is associated.

Here is the table:

```
comment_table = schema.Table('comment', meta.metadata,
    schema.Column('id', types.Integer,
        schema.Sequence('comment_seq_id', optional=True), primary_key=True),
    schema.Column('pageid', types.Integer,
        schema.ForeignKey('page.id'), nullable=False),
    schema.Column('content', types.Text(), default=u''),
    schema.Column('name', types.Unicode(255)),
    schema.Column('email', types.Unicode(255), nullable=False),
    schema.Column('created', types.TIMESTAMP(), default=now()),
)
```

You'll recall that the table contains an `id` field so that each comment can be uniquely identified and that it contains a `pageid` field, which is a foreign key holding the `id` of the page to which the comment is associated.

The class definition for the comment looks like this:

```
class Comment(object):
    pass
```

The mapper for the page already takes into account that each page could have multiple comments:

```
orm.mapper(Page, page_table, properties={
    'comments':orm.relation(Comment, backref='page'),
    'tags':orm.relation(Tag, secondary=pagetag_table)
})
```

You'll recall that this mapper sets up a `.comments` property on `Page` instances for accessing a list of comments, and it also sets up a `.page` property on `Comment` instances for identifying the page associated with a comment. If you've been following the tutorial, you already added these to your model in Chapter 8.

Planning the Controller

Let's think about the requirements for the controller. You would need the following actions:

`view(self, id)`: Displays a comment for a page

`new(self)`: Displays a form to create a new comment on a page

`create(self)`: Saves the information submitted from `new()` and redirects to `view()`

`edit(self, id)`: Displays a form for editing the comment `id` on a page

`save(self, id)`: Saves the comment `id` and redirects to `view()`

`list(self)`: Displays all comments on a page

`delete(self, id)`: Deletes a comment from a page

The comment controller actions need to know which page the comment is associated with (or will be associated with in the case of `new()` and `create()`) so that they deal with the comments for a particular page only. This means in addition to the ID of the comment the actions are changing, they will also need to know the ID of the page the comment is associated with.

With other frameworks, you might have to use hidden fields in your forms and query parameters in your URLs to keep track of the page ID, but Pylons provides a better method: modifying the routes to keep the page ID as part of the URLs used to route requests to the comment controller's actions.

Modifying the Routes

The URLs you will use will be in this form:

```
/page/1/comment/view/4
```

This URL would result in the comment with ID 4 being viewed on page 1. By setting up Routes to understand this URL and map it to the comment controller you will create in a minute, the issue of how to keep track of the page `id` goes away because it will automatically be added when you use `url_for()` and can always be accessed via `request.urlvars`.

To make this work, you need to add the following routes to config/routing.py immediately after # CUSTOM ROUTES HERE and before the existing map.connect('/{controller}/{action}') route:

```
map.connect(
    '/page/{pageid}/{controller}/{action}',
    requirements=dict(pageid='\d+')
)
map.connect(
    '/page/{pageid}/{controller}/{action}/{id}',
    requirements=dict(pageid='\d+', id='\d+')
)
```

These routes require that both the pageid and id routing variables are integers. Checking this here saves you from having to perform the check in each of the controller actions.

Now that you've learned about the explicit=True option to Routes' Mapper object, let's use this option in the SimpleSite project to disable route memory and implicit defaults as recommended in Chapter 9. Change the Mapper() lines in config/routing.py to look like this, ensuring minimization is also disabled by setting map.minimization = False:

```
map = Mapper(directory=config['pylons.paths']['controllers'],
             always_scan=config['debug'], explicit=True)
map.minimization = False
```

With this change in place, you'll also need to update the section links because when using explicit=True, you no longer need to override the route memory value for id. Edit templates/derived/page/view.html so that the first two links are changed from this:

```
<a href="${h.url_for(controller='page', action='list', id=None)}">All Pages</a>
| <a href="${h.url_for(controller='page', action='new', id=None)}">New Page</a>
```

to the following:

```
<a href="${h.url_for(controller='page', action='list')}">All Pages</a>
| <a href="${h.url_for(controller='page', action='new')}">New Page</a>
```

There's one more subtle place where the change to explicit routing has a consequence: inside the paginator. Luckily, additional keyword arguments passed to the Page constructor are also passed to any calls the paginator makes to h.url_for(). This means you just have to specify controller and list explicitly as keyword arguments to the Page() constructor. Replace the current list() action with this, renaming the records variable to page_q at the same time to reflect that it is really a query object:

```
def list(self):
    page_q = meta.Session.query(model.Page)
    c.paginator = paginate.Page(
        page_q,
        page=int(request.params.get('page', 1)),
        items_per_page = 2,
        controller='page',
        action='list',
    )
    return render('/derived/page/list.html')
```

Creating the Controller

Rather than creating the controller from scratch, let's reuse the page controller you wrote in Chapter 8. Make a copy of it named comment.py in the controllers directory, and then replace every instance of the string page with comment and every instance of the string Page with Comment. If you are on a Linux or Unix platform, these commands will do it for you:

```
$ cd simplesite/controllers
$ cp page.py comment.py
$ perl -pi -w -e 's/page/comment/g; s/Page/Comment/g;' comment.py
```

Now let's do the same with the templates:

```
$ cd ../templates/derived
$ cp -r page comment
$ cd comment
$ perl -pi -w -e 's/page/comment/g; s/Page/Comment/g;' *.html
```

You'll need to correct the new comment controller's `list()` action because some of the **variables** will have been accidentally renamed. Change it to look like this:

```
def list(self):
    comments_q = meta.Session.query(model.Comment)
    c.paginator = paginate.Page(
        comments_q,
        page=int(request.params.get('page', 1)),
        items_per_page = 10,
        controller='comment',
        action='list'
    )
    return render('/derived/comment/list.html')
```

You'll actually use this basic controller template again later in the tutorial when you create a controller to handle tags and sections, so take a copy of the comment controller and call it `template.py.txt` so that you can use it later (you are using a `.py.txt` extension so that the template isn't accidentally treated as a controller):

```
cd ../../../
$ cp comment.py template.py.txt
```

Updating the Controller to Handle Comments

Now that the basic structure of the comment controller is in place, it needs to be updated to correctly handle the fields and relationships of the `comment` table. Comment objects have a `.content` property for the comment text itself, a `.name` property to hold the name of the person who left the comment, and an `.email` property for their e-mail address. You'll need fields for each of these so that a user can leave a comment. Let's start by creating a FormEncode schema. Update the `NewCommentForm` schema to look like this:

```
class NewCommentForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    name = formencode.validators.String(not_empty=True)
    email = formencode.validators.Email(not_empty=True)
    content = formencode.validators.String(
        not_empty=True,
        messages={
            'empty':'Please enter a comment.'
        }
    )
```

The example uses the `allow_extra_fields = True` option so that the form's submit button isn't validated and uses the `filter_extra_fields = True` option so that it isn't included in the results returned when the schema converts the form input to a Python dictionary. A custom error message is used if the user forgets to enter a comment, and the e-mail address uses an `Email` validator to make sure the user enters a string that looks like an e-mail address.

You'll also need to update the `/templates/derived/comment/fields.html` file so it represents the correct fields you'd like users to enter:

```
${h.field(
    "Name",
    h.text(name='name'),
    required=True,
)}
${h.field(
    "Email",
    h.text(name='email'),
    required=True,
    field_desc = 'Use to help prevent spam but will not be published',
)}
${h.field(
    "Comment",
    h.textarea(name='content', rows=7, cols=40),
    required=True,
)}
```

Notice that although the field name is called `content`, it is labeled `Comment`. This is to make it more obvious to the users of the application. After all, they don't need to know that the comment text they enter is actually stored in the `content` column of the table.

Next update the `edit()` action so that the correct values are prepared for the call to `htmlfill.render()`:

```
values = {
    'name': comment.name,
    'email': comment.email,
    'content': comment.content,
}
```

Let's also update the `view.html` template to display the comment information to look more like a comment. Update it to look like this:

```
<%inherit file="/base/index.html"/>

<%def name="title()">Comment</%def>
<%def name="heading()"><h1>Comment</h1></%def>

${c.comment.content}

<p><em>Posted by ${c.comment.name} on ${c.comment.created.strftime('%c')}.</em></p>

<p><a href="${h.url_for(controller='page', action='view', ➥
id=c.comment.pageid)}">Visit the page this comment was posted on.</a></p>
```

Finally, you'll need to update the `list.html` template so that the `pager()` method of the paginator is named `pager()` rather than `commentr()` after the automatic rename and so that the paginator displays information relevant to the comments rather than pages. Here's the updated version:

```
<%inherit file="/base/index.html" />

<%def name="heading()"><h1>Comment List</h1></%def>

<%def name="buildrow(comment, odd=True)">
    <tr class="${odd and 'odd' or 'even'}">
        <td valign="top">
            ${h.link_to(
                comment.id,
                h.url_for(
                    controller=u'comment',
                    action='view',
                    id=unicode(comment.id)
                )
            )}
        </td>
        <td valign="top">
            ${h.link_to(
                comment.name,
                h.url_for(
                    controller=u'comment',
                    action='edit',
                    id=unicode(comment.id)
                )
            )}
        </td>
        <td valign="top">${comment.created.strftime('%c')}</td>
    </tr>
</%def>

% if len(c.paginator):
<p>${ c.paginator.pager('$link_first $link_previous $first_item to $last_item ➥
of $item_count $link_next $link_last') }</p>
<table class="paginator"><tr><th>Comment ID</th><th>Comment Title</th>➥
<th>Posted</th></tr>
<% counter=0 %>
% for item in c.paginator:
    ${buildrow(item, counter%2)}
    <% counter += 1 %>
% endfor
</table>
<p>${ c.paginator.pager('~2~') }</p>
% else:
<p>
    No comments have yet been created.
    <a href="${h.url_for(controller='comment', action='new')}">Add one</a>.
</p>
% endif
```

At this point, you would be able to perform all the usual actions on comments such as add, edit, and remove if it weren't for the fact they also need a `pageid`.

Now you can start the server and see what you have:

```
$ paster serve --reload development.ini
```

Visit `http://localhost:5000/comment/new`, and you should see the comment form shown in Figure 14-1.

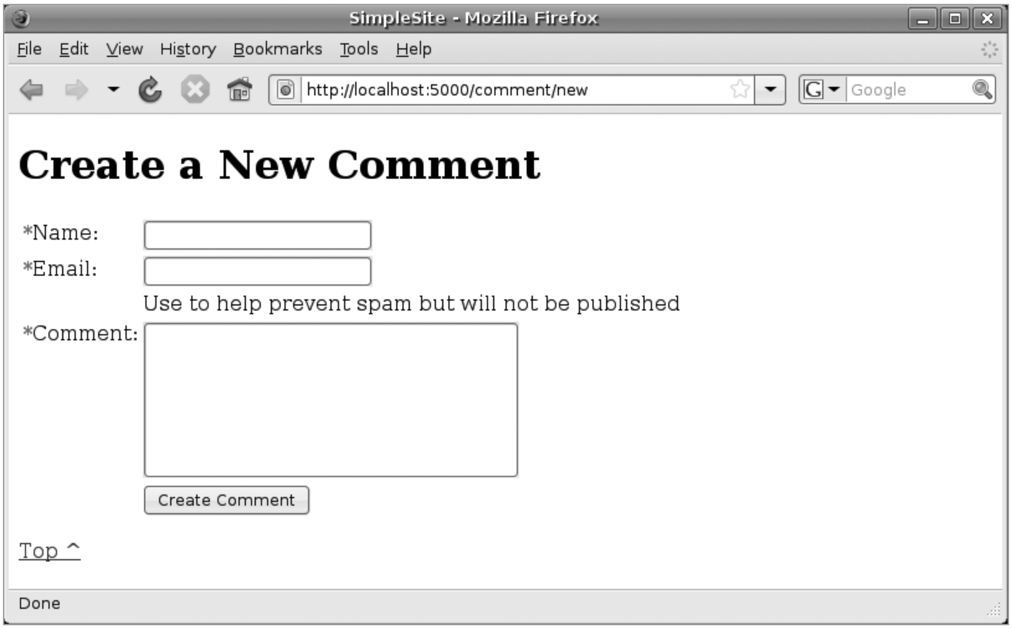

SimpleSite - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/comment/new
Google
Create a New Comment
*Name:
*Email:
Use to help prevent spam but will not be published
*Comment:
Create Comment
Top ^
Done

Figure 14-1. *The create comment form*

Setting the Page ID Automatically

If you try to create a comment at the URL you've just visited, an `IntegrityError` will be raised specifying `comment.pageid may not be NULL` because no page ID has been specified. As I mentioned earlier in the chapter, you'll obtain the page ID from the URL. To set this up, you are going to use the `__before__()` method that gets called before each of the Pylons actions. Add it right at the top of the controller before the `view()` action:

```
class CommentController(BaseController):

    def __before__(self, action, pageid=None):
        page_q = meta.Session.query(model.Page)
        c.page = pageid and page_q.filter_by(id=int(pageid)).first() or None
        if c.page is None:
            abort(404)
```

This code causes the variable `c.page` to be set before any actions are called. If the page ID is not included in the URL or the page doesn't exist, a 404 Not Found response is returned. With this code in place, visiting `http://localhost:5000/comment/new` results in a 404 Not Found response; visiting `http://localhost:5000/page/1/comment/new` correctly displays the new comment form, but the comment will still not save because the form does not yet submit to `http://localhost:5000/page/1/comment/create`. Let's fix that by editing the `new.html` template to change the `h.url_for()` call to include the page ID:

```
<%inherit file="/base/index.html" />
<%namespace file="fields.html" name="fields" import="*"/>

<%def name="heading()">
    <h1 class="main">Create a New Comment</h1>
</%def>
```

```
${h.form_start(h.url_for(pageid=c.page.id, controller='comment', action='create'), ➥
method="post")}
    ${fields.body()}
    ${h.field(field=h.submit(value="Create Comment", name='submit'))}
${h.form_end()}
```

You'll also need to change the `edit.html` template so that the form also includes the page ID:

```
<%inherit file="/base/index.html" />
<%namespace file="fields.html" name="fields" import="*"/>

<%def name="heading()">
    <h1 class="main">Editing ${c.title}</h1>
</%def>

<p>Editing the source code for the ${c.title} comment:</p>

${h.form_start(h.url_for(pageid=c.page.id, controller='comment', action='save', ➥
id=request.urlvars['id']), method="post")}
    ${fields.body()}
    ${h.field(field=h.submit(value="Save Changes", name='submit'))}
${h.form_end()}
```

Let's consider each of the actions of the comment controller in turn to decide how they should behave and how they will need to be modified:

`view()`: The view method needs to be updated to ensure that the comment requested is actually a comment from the page specified in the URL. You can do this by updating the query used in the `view()` action from this:

```
c.comment = comment_q.get(int(id))
```

to the following:

```
c.comment = comment_q.filter_by(pageid=c.page.id, id=int(id)).first()
```

`new()`: This action needs no change since it is responsible only for displaying the form for adding a new comment.

`create()`: This action needs to know the page to which the comment is being added. Just before the comment is added to the session, add the following line to set the page ID:

```
comment.pageid = c.page.id
```

You'll also need to include the page ID in the URL to which the browser is redirected. Since you already learned about the `redirect_to()` function in Chapter 9, let's use it here. Replace the redirect lines with these:

```
# Issue an HTTP redirect
return redirect_to(pageid=c.page.id, controller='comment', action='view', ➥
id=comment.id)
```

`edit()`: The edit action needs a similar modification to the one made to the `view()` method. Although you know which page a comment is associated with, you want to make sure the URL requested has the same page ID as the comment. Change the query from this:

```
comment = comment_q.filter_by(id=id).first()
```

to the following:

```
comment = comment_q.filter_by(pageid=c.page.id, id=id).first()
```

`save()`: Again, you'll want to check that the page ID in the URL is the same as the one in the comment. Since the form doesn't allow you to change the page ID, this can once again be ensured by adding `c.page.id` to the query:

```
comment = comment_q.filter_by(pageid=c.page.id, id=id).first()
```

Replace the redirect lines with this:

```
# Issue an HTTP redirect
return redirect_to(pageid=c.page.id, controller='comment', action='view', ➥
id=comment.id)
```

`list()`: Only comments associated with the current page should be listed, so once again the query is modified to include the page ID. In this case, though, we also have to pass the `pageid` argument, which will in turn get passed to any `h.url_for()` calls in the paginator.

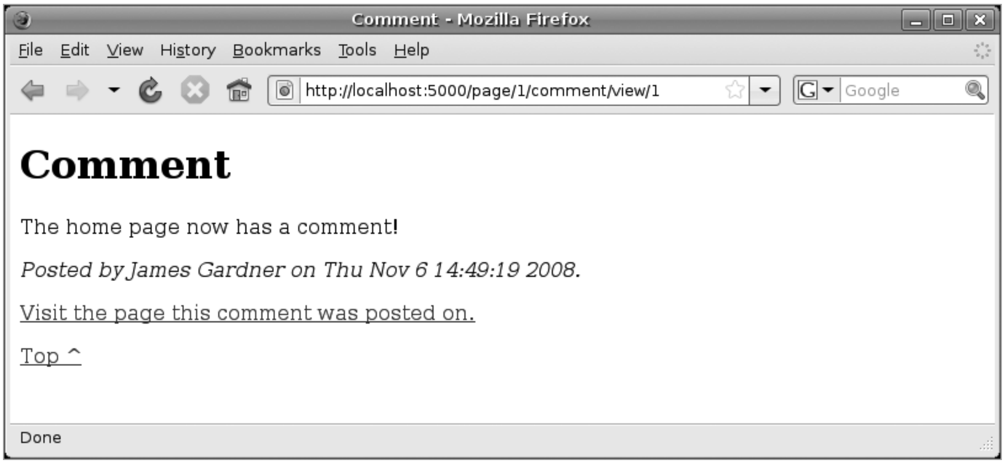
```
def list(self):
    comments_q = meta.Session.query(model.Comment).filter_by(pageid=c.page.id)
    comments_q = comments_q.order_by(model.comment_table.c.created.asc())
    c.paginator = paginate.Page(
        comments_q,
        page=int(request.params.get('page', 1)),
        items_per_page=10,
        pageid=c.pageid,
        controller='comment',
        action='list'
    )
    return render('/derived/comment/list.html')
```

Notice the use of `order_by()` to ensure that the earliest comments are displayed first. I've used the `comment_table` column metadata in the `order_by()` method just to remind you that you can use table metadata as well as class attributes when specifying query arguments, and I've used the `.asc()` method to specify that the results should be specified in ascending order.

`delete()`: Again, this requires only a check that the page ID in the URL is the same as the one in the comment. Since the form doesn't allow you to change the page ID, this can once again be ensured by adding `c.page.id` to the query:

```
comment = comment_q.filter_by(pageid=c.page.id, id=id).first()
```

Now that all the changes have been made, let's test the new controller. Start by adding a new comment to the home page by visiting `http://localhost:5000/page/1/comment/new` and filling in the form. When you click Create Comment, you will see Figure 14-2.


Comment - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/page/1/comment/view/1
Google
Done

Comment

The home page now has a comment!

Posted by James Gardner on Thu Nov 6 14:49:19 2008.

Visit the page this comment was posted on.

Top ^

Figure 14-2. *The first comment*

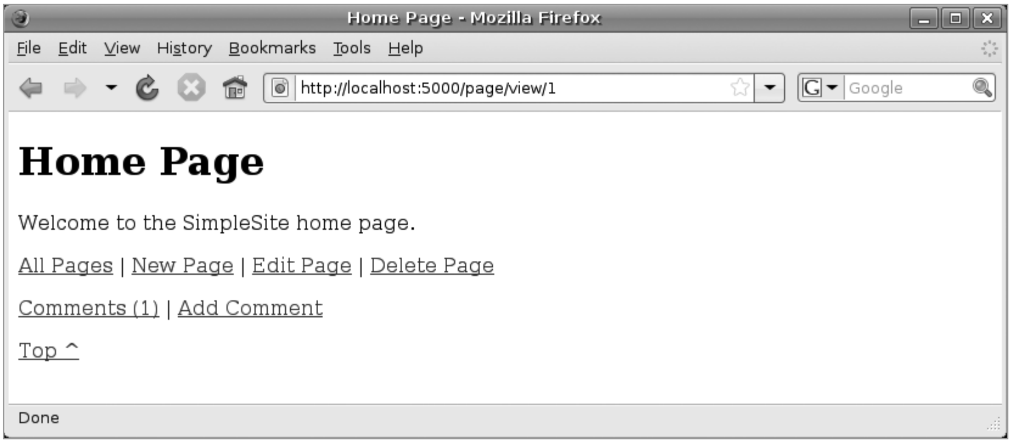
Finally, let's update the comment view template `derived/comment/view.html` so that edit and delete links are added to the footer. Add the following at the end of the template:

```
<%def name="footer()">
## Add our comment links
<p>
  <a href="${h.url_for(pageid=c.page.id, controller='comment', action='edit', ➥
id=c.comment.id)}">Edit Comment</a>
| <a href="${h.url_for(pageid=c.page.id, controller='comment', action='delete', ➥
id=c.comment.id)}">Delete Comment</a>
</p>
## Include the parent footer too
${parent.footer()}
</%def>
```

Make sure you followed the instructions earlier in the chapter to update the `values` variable in the `edit()` action; you will then find you can easily edit or delete comments. There are still no links to display or add comments from the bottom of individual pages. You'll fix that in the next section.

Updating the Page View

SimpleSite will not display a list of comments on the page itself (although you could set it up to do so if you preferred) but will instead display a link at the bottom of each page of the form that says, for example, "Comments (8)" where the number in parentheses is the current number of comments on that page. Users can click this link to view the list of comments. There will also be an Add Comment link so that users can add a comment directly. Figure 14-3 shows what the updated screen will look like.


Home Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/page/view/1
Google

Home Page

Welcome to the SimpleSite home page.

All Pages | New Page | Edit Page | Delete Page

Comments (1) | Add Comment

Top ^

Done

Figure 14-3. *The updated page view screen*

For this to work, you need to modify both the page controller's `view()` action and the template. Let's start with the `view()` action. You need to add a SQLAlchemy query to count the number of pages associated with the page. Add this to the end of the action just before the `return` statement:

```
c.comment_count =  meta.Session.query(model.Comment).filter_by(pageid=id).count()
```

Then modify the `templates/derived/page/view.html` template so the `footer()` def looks like this:

```
<%def name="footer()">
## Then add our page links
<p>
  <a href="${h.url_for(controller='page', action='list')}">All Pages</a>
| <a href="${h.url_for(controller='page', action='new')}">New Page</a>
| <a href="${h.url_for(controller='page', action='edit', ➥
id=c.page.id)}">Edit Page</a>
| <a href="${h.url_for(controller='page', action='delete', ➥
id=c.page.id)}">Delete Page</a>
</p>
## Comment links
<p>
  <a href="${h.url_for(pageid=c.page.id, controller='comment', ➥
action='list')}">Comments (${str(c.comment_count)})</a>
| <a href="${h.url_for(pageid=c.page.id, controller='comment', ➥
action='new')}">Add Comment</a>
</p>
## Include the parent footer too
${parent.footer()}
</%def>
```

Now when you view a page, you will also be able to list or add comments, and by viewing comments individually, you can edit or delete them.

Handling Deleted Pages

Now that comments are related to pages, you need to think about what to do with comments once a page is deleted. Since a comment without the page it is commenting on isn't very useful, you can automatically delete all comments associated with a page when the page itself is deleted.

You could program this code manually in the delete() action of the page controller, but there is actually a better way. SQLAlchemy mappers support the concept of configurable cascade behavior on relations so that you can specify how child objects are dealt with on certain actions of the parents. The options are described in detail at http://www.sqlalchemy.org/docs/05/documentation.html#unitofwork_cascades, but we are simply going to use the option all so that the comments are updated if the page ID changes and are deleted if the page they are for is deleted.

Modify the page mapper in model/__init__.py so that the comments relation has cascade='all' specified like this:

```
orm.mapper(Page, page_table, properties={
   'comments':orm.relation(Comment, backref='page', cascade='all'),
   'tags':orm.relation(Tag, secondary=pagetag_table)
})
```

Try creating a page, adding some comments, and then deleting the page. If you looked at the database table, you'd find that the comments are automatically deleted too.

If you are following along with a SQLite database named development.db, you could check this by connecting to the database with the sqlite3 program:

```
$ sqlite3 development.db
```

Then by executing this SQL:

```
SELECT id, pageid FROM comment;
```

you'd find that there were no comments for the page you just deleted because the SQLAlchemy cascade rules you specified led to SQLAlchemy deleting them for you.

Tags: Many-to-Many Mappings

Now that you've seen how to handle a one-to-many mapping (sometimes called a *parent-child* relationship) between pages and comments, you can turn your attention to the many-to-many mapping between tags and pages. Once again, tags can be created, viewed, updated, or deleted. So, the controller that manipulates them would need the same actions as the page and comment controllers you've created so far. In addition, each page can have multiple tags, and each tag can be used on multiple pages so that tags can't be considered children of pages any more than pages can be considered children of tags.

The way you'll implement this is by once again starting with a simple controller and renaming the core variables with the word *tag*. You'll then tweak the controller so that it correctly handles the columns of the tag table.

After you've done this, users will be able to add, edit, remove, and list tags. I'll then cover how to associate tags with pages. Ordinarily, you would need to create a second controller for handling the adding, editing, listing, and deleting of the *associations* between the page table and the tag table. In this case, though, you'll take a shortcut. Rather than having a second controller to handle the interactions, you will simply display a check box group of all the available tags on each page. Users can then select the tags they want associated with the page, and SQAlchemy will handle how to store those associations in the pagetag table for you automatically.

Creating the tag Controller

Let's start by creating the tag controller from the template copied earlier:

```
$ cd simplesite/controllers
$ cp template.py.txt tag.py
$ perl -pi -w -e 's/comment/tag/g; s/Comment/Tag/g;' tag.py
```

You'll need to correct the new tag controller's `list()` action too because some of the variables will have been accidentally renamed. Change it to look like this:

```
def list(self):
    tag_q = meta.Session.query(model.Tag)
    c.paginator = paginate.Page(
        tag_q,
        page=int(request.params.get('page', 1)),
        items_per_page = 10,
        controller='tag',
        action='list'
    )
    return render('/derived/tag/list.html')
```

Now let's do the same with the templates, but let's use the page templates as a basis:

```
$ cd ../templates/derived
$ cp -r page tag
$ cd tag
$ perl -pi -w -e 's/page/tag/g; s/Page/Tag/g;' *.html
```

Once again, you'll need to update `list.html` to use `c.paginator.pager()`, not `c.paginator.tagr()`.

Now restart the server if you stopped it to make these changes, and let's get started with the updates:

```
$ cd ../../../../
$ paster serve --reload development.ini
```

Tags have a name only, so update the `NewTagForm` schema to look like this:

```
class NewTagForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    name = formencode.validators.String(not_empty=True)
```

Change the `edit()` action so that the values passed to `htmlfill.render()` look like this:

```
values = {
    'name': tag.name,
}
```

Next, change the `fields.html` template so that it looks like this:

```
${h.field(
    "Name",
    h.text(name='name'),
    required=True,
)}
```

Update the tag `view.html` template so it looks like this:

```
<%inherit file="/base/index.html"/>

<%def name="title()">Tag</%def>
<%def name="heading()"><h1>Tag</h1></%def>

${c.tag.name}

<%def name="footer()">
## Add our tag links
<p>
  <a href="${h.url_for(controller='tag', action='edit', id=c.tag.id)}">Edit Tag</a>
| <a href="${h.url_for(controller='tag', action='delete', ➥
id=c.tag.id)}">Delete Tag</a>
</p>
## Include the parent footer too
${parent.footer()}
</%def>
```

Finally, update the `tag/list.html` template so it looks like this:

```
<%inherit file="/base/index.html" />

<%def name="heading()"><h1>Tag List</h1></%def>

<%def name="buildrow(tag, odd=True)">
    <tr class="${odd and 'odd' or 'even'}">
        <td valign="top">
            ${h.link_to(
                tag.id,
                h.url_for(
                    controller=u'tag',
                    action='view',
                    id=unicode(tag.id)
                )
            )}
        </td>
        <td valign="top">
            ${tag.name}
        </td>
    </tr>
</%def>

% if len(c.paginator):
<p>${ c.paginator.pager('$link_first $link_previous $first_item to $last_item ➥
of $item_count $link_next $link_last') }</p>
<table class="paginator"><tr><th>Tag ID</th><th>Tag Name</th></tr>
<% counter=0 %>
% for item in c.paginator:
    ${buildrow(item, counter%2)}
    <% counter += 1 %>
% endfor
</table>
```

```
<p>${ c.paginator.pager('~2~') }</p>
% else:
<p>
    No tags have yet been created.
    <a href="${h.url_for(controller='tag', action='new')}">Add one</a>.
</p>
% endif
```

That's it—the tag controller is complete, so you could now start creating tags by visiting `http://localhost:5000/tag/new`; however, before you do, let's add a few restrictions to what can be used as a tag name.

Constraining Tag Names

You'll put a restriction on tag names to ensure they can be made only from letters, numbers, and the space character and can consist of 20 characters or less. Also, you don't want users to add a tag with a name that already exists. Of course, because of the constraints you set up when defining the model, you know that an exception will be raised if a nonunique tag name is added, but the 500 Internal Server Error page that will be generated doesn't provide a way to let the user fix the error, so you need a FormEncode validator to check for the error before it occurs and to display an appropriate error message if necessary.

First let's create a validator to check for unique tags and update the `NewTagForm` schema to use it. Add this to the top of the tag controller instead of the current `NewTagForm` schema:

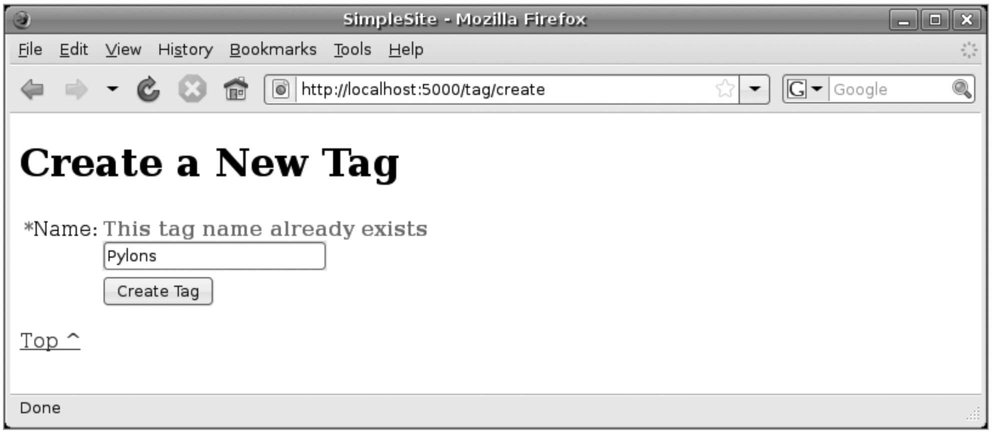
```
import re

class UniqueTag(formencode.validators.FancyValidator):
    def _to_python(self, value, state):
        # Check we have a valid string first
        value = formencode.validators.String(max=20).to_python(value, state)
        # Check that tags are only letters, numbers, and the space character
        result = re.compile("[^a-zA-Z0-9 ]").search(value)
        if result:
            raise formencode.Invalid("Tags can only contain letters, ➥
numbers and spaces", value, state)
        # Ensure the tag is unique
        tag_q = meta.Session.query(model.Tag).filter_by(name=value)
        if request.urlvars['action'] == 'save':
            # Ignore the existing name when performing the check
            tag_q = tag_q.filter(model.Tag.id != int(request.urlvars['id']))
        first_tag = tag_q.first()
        if first_tag is not None:
            raise formencode.Invalid("This tag name already exists", value, state)
        return value

class NewTagForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    name = UniqueTag(not_empty=True)
```

There's quite a lot going on in the `UniqueTag` validator, so let's go through what it does. When the validator is called, the first thing that happens is a check to ensure the tag name is a valid string with 20 characters. If the check passes, a regular expression is used to ensure that only alphanumerics and the space character are used in the name. Next, a SQLAlchemy query object is set up to query any tags with a name equal to the name being validated. What happens next depends on whether the validator is used in the `save()` action decorator or the `create()` action decorator. You'll recall that `request.urlvars` contains all the routing variables matched by Routes, so in this case the action is stored in `request.urlvars['action']`. If this is equal to `'save'`, the `save()` action is being called, and the tag query is filtered to exclude the tag with the same ID as the current request. This prevents the tag save from failing when someone saves a tag without changing its name. If a tag with the same name exists after the query has been set up and filtered, then an `Invalid` exception is raised, which results in an error message above the form field.

With these changes in place, you can visit `http://localhost:5000/tag/new` to test the new tag functionality. If you try to create two tags with the same name, you'll see the error message shown in Figure 14-4.


SimpleSite - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/tag/create
Google

Create a New Tag

*Name: This tag name already exists

Pylons

Create Tag

Top ^

Done

Figure 14-4. *The error message shown when you create two tags with the same name*

That's it! SimpleSite now supports tags, but you can't yet add them to pages. Let's look at this in the next section.

■ **Caution** Sharp-eyed readers might not be too happy with the validator I've just described. In this case, the validator uses `model.Session` and `request`, both of which are request-specific and should ordinarily be passed via the `state` argument to a schema's `to_python()` method, as you'll recall from Chapter 6. In this case, though, all the validation happens behind the scenes in Pylons' `@vailidate` decorator, so there isn't an opportunity to specify a `state` argument. Luckily, both `model.Session` and `request` are special objects that Pylons ensures behave correctly during each request, even in a multithreaded environment, so this example is perfectly OK in this case.

If your validator accessed an object that wasn't thread-safe, you could do the following:

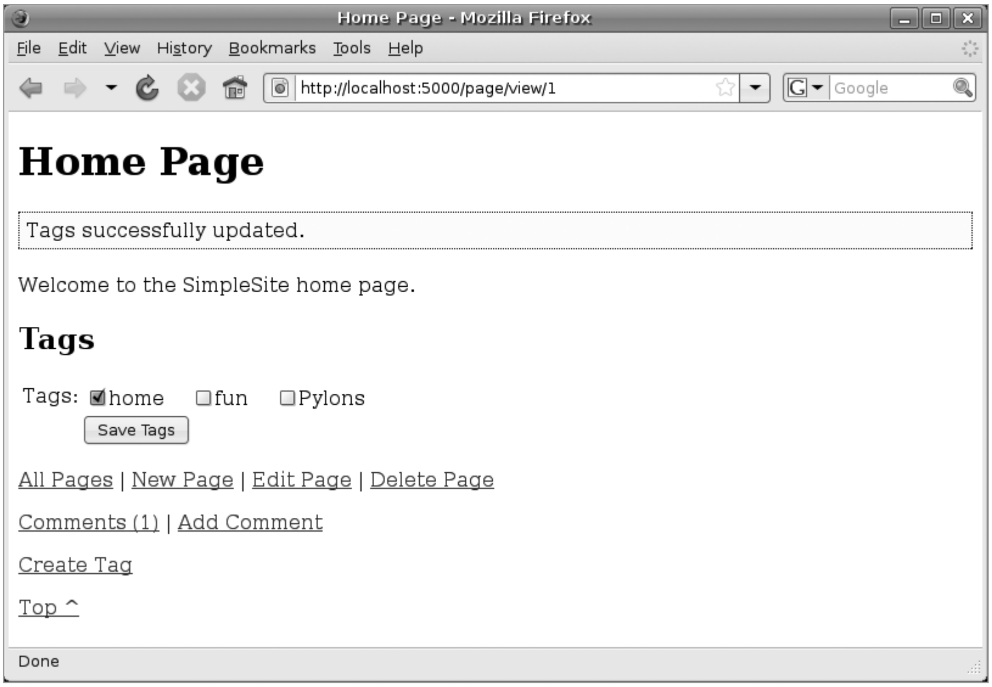
- Assign the non-thread-safe object to the template context global `c` in the controller's `__before__()` method to make it available in the validator's `_to_python()` method before the validator is called.
- Handle the entire validation process manually, explicitly passing a state object to the `to_python()` method as demonstrated in the `process()` action of the example in the "Solving the Repeating Fields Problem" section of Chapter 6 where the template context global `c` is itself used as the `state` argument.
- Use a `StackedObjectProxy` object to give Pylons the responsibility of using the correct version of the object for the particular request that is running.

The first two alternatives are the preferred approaches, but see the section "The Registry Manager, StackedObjectProxy, and Pylons Globals" in Chapter 17 if you want to investigate the `StackedObjectProxy` approach.

Adding Tags to Pages

Now that you have a system for adding and editing tags, you need a way of associating tags with pages. As was mentioned earlier, you can choose to do this in two ways. The first is with the `pagetag` controller to provide an interface to allow users to manually add entries to the `pagetag` table to create the associations. If the `tag` table contained more fields or didn't have a column that could be used naturally as a primary key, then this would be a good option. In this case, though, the tag name provides a unique way to specify the tag, so you can simply provide a list of all available tags on each page with a check box next to each, and users can simply select the boxes of the tags they want to use.

Figure 14-5 shows what a page will look like when you've finished this section and saved the tags associated with a page.


Home Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/page/view/1
Google

Home Page

Tags successfully updated.

Welcome to the SimpleSite home page.

Tags

Tags: ☑ home ☐ fun ☐ Pylons

Save Tags

All Pages | New Page | Edit Page | Delete Page

Comments (1) | Add Comment

Create Tag

Top ^

Done

Figure 14-5. *A page containing the tag list*

Let's start by editing the page controller's `view()` action to obtain a list of all the available tag names. Update it to look like this (the lines to add are in bold):

```
def view(self, id=None):
    if id is None:
        abort(404)
    page_q = meta.Session.query(model.Page)
    c.page = page_q.filter_by(id=int(id)).first()
    if c.page is None:
        abort(404)
    c.comment_count = meta.Session.query(model.Comment).filter_by(pageid=id).count()
    tag_q = meta.Session.query(model.Tag)
    c.available_tags = [(tag.id, tag.name) for tag in tag_q]
    c.selected_tags = {'tags':[str(tag.id) for tag in c.page.tags]}
    return render('/derived/page/view.html')
```

In the templates/derived/page/view.html template, add a new form for the tags just before the footer() def. The code is wrapped in a def block because later in the section you'll need to capture its output to use with HTMLFill to populate the fields:

```
<%def name="tags(available_tags)">
    <h2>Tags</h2>
    ${h.form_start(h.url_for(controller='page', action='update_tags', ➥
id=c.page.id), method='post')}
        ${h.field(
            "Tags",
            h.checkbox_group('tags', selected_values=None, align="table", ➥
options=available_tags)
        )}
        ${h.field(field=h.submit(value="Save Tags", name='submit'))}
    ${h.form_end()}
</%def>
```

For this to work, you'll need to add the check box_group() helper to lib/helpers.py:

```
from formbuild.helpers import checkbox_group
```

This form will submit to the page controller's update_tags() action which you'll create in a minute. Once again though, you'll need to validate the result of any form submission. Since the check boxes are effectively a set of repeating fields, you could use a ForEach validator like this:

```
class ValidTagsForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    tags = formencode.foreach.ForEach(formencode.validators.Int())
```

Although this schema checks that the tags have integer values, it doesn't actually check that the values are actually valid for the tags. To do this, you could derive your own validator from the Int validator and override its _to_python() method to check the value using a similar technique to the one used in UniqueTag, but then a separate database call would need to be made for each tag that needed to be validated. Instead, you'll create a *chained validator* that will take the list of integers and validate them all in one go. It looks like this:

```
class ValidTags(formencode.FancyValidator):
    def _to_python(self, values, state):
        # Because this is a chained validator, values will contain
        # a dictionary with a tags key associated with a list of
        # integer values representing the selected tags.
        all_tag_ids = [tag.id for tag in meta.Session.query(model.Tag)]
        for tag_id in values['tags']:
            if tag_id not in all_tag_ids:
                raise formencode.Invalid(
                    "One or more selected tags could not be found in the database",
                    values,
                    state
                )
        return values
```

Add the ValidTags validator to the top of the page.py controller after the existing schema, then add the the ValidTagsForm schema to look like this:

```
class ValidTagsForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    tags = formencode.foreach.ForEach(formencode.validators.Int())
    chained_validators = [ValidTags()]
```

Now we can write the update_tags() action. Add this to the page controller:

```
@restrict('POST')
@validate(schema=ValidTagsForm(), form='view')
def update_tags(self, id=None):
    if id is None:
        abort(404)
    page_q = meta.Session.query(model.Page)
    page = page_q.filter_by(id=id).first()
    if page is None:
        abort(404)
    tags_to_add = []
    for i, tag in enumerate(page.tags):
        if tag.id not in self.form_result['tags']:
            del page.tags[i]
    tagids = [tag.id for tag in page.tags]
    for tag in self.form_result['tags']:
        if tag not in tagids:
            t = meta.Session.query(model.Tag).get(tag)
            page.tags.append(t)
    meta.Session.commit()
    session['flash'] = 'Tags successfully updated.'
    session.save()
    return redirect_to(controller='page', action='view', id=page.id)
```

This code iterates over the real tags twice, deleting any unselected boxes first and adding any new associations from boxes that have just been selected.

Now that the tags are correctly saving, you need to ensure that their values are correctly populated when the page is displayed. To do this, you'll call the tags() def with Mako's special capture() function to capture the HTML from the form and then pass the HTML through HTMLFill to populate the tags. This is a lot like the method you've been using for populating forms, but rather than calling htmlfill.render() in the controller with the whole output from the template, you are just calling it in the template with the form output from the tags() def.

Update the page view.html template to call the tags() def you added earlier in this section. Add this just after the tags() def and before the footer() def:

```
<%!
    from formencode import htmlfill
    from webhelpers.html import literal
%>

% if c.available_tags:
${literal(htmlfill.render(capture(self.tags, c.available_tags), c.selected_tags))}
% endif
```

In this case, it should be safe to use literal() here since the output from self.tags will already be escaped and htmlfill.render() correctly escapes the values passed in. Notice that Mako's capture() function takes the def to capture as the first argument and any arguments to pass to that function as subsequent arguments. If you were to call `capture(self.tags(c.available_tags))`, the `tags()` def would be called, outputting its content to the buffer, and `capture()` would try to call the return value from the def instead of the def itself.

There is one last change you need to make. Let's add a link in the footer to enable users to add new tags:

```
<%def name="footer()">
## Then add our page links
<p>
  <a href="${h.url_for(controller='page', action='list')}">All Pages</a>
| <a href="${h.url_for(controller='page', action='new')}">New Page</a>
| <a href="${h.url_for(controller='page', action='edit', ➥
id=c.page.id)}">Edit Page</a>
| <a href="${h.url_for(controller='page', action='delete', ➥
id=c.page.id)}">Delete Page</a>
</p>
## Comment links
<p>
  <a href="${h.url_for(pageid=c.page.id, controller='comment', ➥
action='list')}">Comments (${str(c.comment_count)})</a>
| <a href="${h.url_for(pageid=c.page.id, controller='comment', ➥
action='new')}">Add Comment</a>
</p>
## Tag links
<p><a href="${h.url_for(controller='tag', action='list')}">All Tags</a>
| <a href="${h.url_for(controller='tag', action='new')}">Add Tag</a></p>
## Include the parent footer too
${parent.footer()}
</%def>
```

Deleting Tags and Pages

When a tag is deleted, it can no longer be used on a page, so all references to that tag need to be removed from the `pagetag` table. Likewise, when a page is deleted, there is little point in keeping track of which tags used to be associated with it, so all references to that page should be removed from the `pagetag` table too.

Add the following line just before `meta.Session.delete(page)` in the `delete()` action in the page controller:

```
meta.Session.execute(delete(model.pagetag_table, ➥
model.pagetag_table.c.pageid==page.id))
```

Now add this just before `meta.Session.delete(tag)` in the `delete()` action of the tag controller:

```
meta.Session.execute(delete(model.pagetag_table, ➥
model.pagetag_table.c.tagid==tag.id))
```

Both controllers will require the following import:

```
from sqlalchemy import delete
```

If you visit the home page at `http://localhost:5000/page/view/1` and create some tags, you'll now be able to tag pages. The application should look like it did in Figure 14-5.

Creating a Navigation Hierarchy

Now that the basic functionality of the web site is in place, I'll cover how to add a navigation structure to turn the application into a full (albeit simple) content management system. In this second part of the chapter, you'll learn about table inheritance and how to structure hierarchical data in SQLAlchemy.

Let's start by thinking about how pages are typically structured on web sites. Pages can usually be thought of as being divided into sections, with each section containing pages and other sections. You will need top-level tabs for the top-level sections and then a navigation menu so that the pages within each section can be displayed. You'll also need a breadcrumb trail so that the user can always navigate to a section higher up in the hierarchy.

The URL for each page will be determined by the URL path info part. If a URL resolves to a section rather than to a page, you need to render the index page for that section. The index page will simply be the page in the section named `index`. You'll also set up the URLs such that those with a trailing slash (`/`) at the end always resolve to sections and those without resolve to a page. Thus, `/dev` will display a page, and `/dev/` will display the `index` page in the `dev` section. Some people find it a little strange to have pages without file extensions, but if you would prefer your pages to have URLs that end in `.html`, feel free to update the code as you work through the examples.

The following is the URL structure to create. The first part represents the URL, the second is the name of the page, and the third part explains whether it is a page or a section:

```
/ Home (Section)
    home Home (Page)
    dev/ Development (Section)
        home Development Home (Page)
        SVN Details (Page)
    Contact (Page)
```

The top-level tabs will therefore show the text *Home*, *Development*, and *Contact*.

Using Inheritance in SQLAlchemy

If you think about how pages and sections might work, you'll notice that both pages and sections will need the following attributes:

- Unique ID
- Display name in the navigation structure
- URL path fragment (for example, `/dev` or `dev.html`)
- Parent section ID
- The ID of the sibling node that this node appears before (or `None` if this is the last node in the section)

Because both the pages and the sections share the same attributes, it makes sense to store them both in the same table. This table will also need a Type column to describe whether the record represents a page or a section.

You can also imagine a situation where other content types are supported, perhaps Word documents or PNG images. Although you won't implement them, you can imagine that any such objects would also need these same attributes. The characteristic that pages, sections, and other types of objects share is that they can all be accessed via a URL and should appear in the navigation structure of the site. In effect, they all can be *navigated*, so let's name the table that will store this information `nav`. The new `nav` table looks like this:

```
nav_table = schema.Table('nav', meta.metadata,
    schema.Column('id', types.Integer(),
        schema.Sequence('nav_id_seq', optional=True), primary_key=True),
    schema.Column('name', types.Unicode(255), default=u'Untitled Node'),
    schema.Column('path', types.Unicode(255), default=u''),
    schema.Column('section', types.Integer(), schema.ForeignKey('nav.id')),
    schema.Column('before', types.Integer(), default=None),
    schema.Column('type', types.String(30), nullable=False)
)
```

You will still want to be able to work with page and section objects in the model, so you'll need to use SQLAlchemy's powerful inheritance tools so that the `Page` and `Section` classes inherit information from a `Nav` class. Replace the existing `Page` class with these three classes:

```
class Nav(object):
    pass

class Page(Nav):
    pass

class Section(Nav):
    pass
```

You'll also need a new mapper for the `Nav` class, and you'll need to tell SQLAlchemy that `Page` and `Section` will inherit from `Nav` and therefore should also have all the same properties a `Nav` object would have. Here's what the updated mappers look like:

```
orm.mapper(Comment, comment_table)
orm.mapper(Tag, tag_table)
orm.mapper(Nav, nav_table, polymorphic_on=nav_table.c.type, ➥
polymorphic_identity='nav')
orm.mapper(Section, section_table, inherits=Nav, polymorphic_identity='section')
orm.mapper(Page, page_table, inherits=Nav, polymorphic_identity='page', properties={
   'comments':orm.relation(Comment, backref='page', cascade='all'),
   'tags':orm.relation(Tag, secondary=pagetag_table)
})
```

The important points to notice are that the `Nav` mapper specifies that `Nav` is polymorphic on `nav_table.c.type`, in other words, that the `type` column will contain a string to specify whether the record is a page or a section. The `Section` and `Page` mappers then specify that they inherit from `Nav` and specify the text to be used in the `nav` table's `type` column to identify them. Now would be a good time to make these changes to your model if you haven't already done so.

With the class and mapper changes set up, let's think about how you need to modify the page table and what fields you'd like the section table to contain.

In SimpleSite, the section table doesn't need to hold any data other than an ID because all the attributes it needs are already inherited from the `nav` table. The ID for a section has to be the same as the corresponding ID in the nav table so that SQLAlchemy knows how sections and navs are related. This means the ID should be a foreign key. Add the section table like this:

```
section_table = sa.Table('section', meta.metadata,
    schema.Column('id', types.Integer,
        schema.ForeignKey('nav.id'), primary_key=True),
)
```

■**Note** In this particular case, the attributes required suggest you could have simply created a `Section` object and had the `Page` inherit from it rather than having a separate `Nav` object and choosing that `Section` and `Page` inherit from it. The important point to be aware of is that you should also look at how objects you are modeling relate to each other in the real world as well as looking at how their attributes suggest they could be related. Pages aren't really like sections because they can't contain other pages and sections, so it is not wise to structure your model in a way that assumes they are.

The page table remains unchanged because you still want page-specific data stored in the `page` table and navigation information about the page stored in the `nav` table. Once again, though, the page's ID field needs to be a foreign key representing the ID of the record in the `nav` table from which it inherits. Change the definition for the `id` column of the `page` table to this:

```
schema.Column('id', types.Integer, schema.ForeignKey('nav.id'), primary_key=True),
```

With these changes in place, our `Page` and `Section` objects will automatically have all the attributes of `Nav` objects even though the information is physically stored in a different table.

Setting Up Initial Data

Now that the new model structure is in place, you'll need to update the `websetup.py` file so that the project contains more appropriate initial data. Update `websetup.py` so that it looks like this (notice that you drop all the tables first if the function is being called with configuration from `test.ini` as described in Chapter 12):

```
"""Set up the SimpleSite application"""
import logging
import os.path
from simplesite import model

from simplesite.config.environment import load_environment

log = logging.getLogger(__name__)

def setup_app(command, conf, vars):
    """Place any commands to setup simplesite here"""
    load_environment(conf.global_conf, conf.local_conf)

    # Load the models
    from simplesite.model import meta
    meta.metadata.bind = meta.engine
    filename = os.path.split(conf.filename)[-1]
    if filename == 'test.ini':
        # Permanently drop any existing tables
        log.info("Dropping existing tables...")
        meta.metadata.drop_all(checkfirst=True)
# Continue as before
    # Create the tables if they aren't there already
    meta.metadata.create_all(checkfirst=True)

    log.info("Adding home page...")
    section_home = model.Section()
    section_home.path=u''
    section_home.name=u'Home Section'
    meta.Session.add(section_home)
    meta.Session.flush()
    page_contact = model.Page()
    page_contact.title=u'Contact Us'
    page_contact.path=u'contact'
    page_contact.name=u'Contact Us Page'
    page_contact.content = u'Contact us page'
    page_contact.section=section_home.id
    meta.Session.add(page_contact)
    meta.Session.flush()

    section_dev = model.Section()
    section_dev.path=u'dev'
    section_dev.name=u'Development Section'
    section_dev.section=section_home.id
    section_dev.before=page_contact.id
    meta.Session.add(section_dev)
    meta.Session.flush()

    page_svn = model.Page()
    page_svn.title=u'SVN Page'
    page_svn.path=u'svn'
    page_svn.name=u'SVN Page'
    page_svn.content = u'This is the SVN page.'
    page_svn.section=section_dev.id
    meta.Session.add(page_svn)
    meta.Session.flush()

    page_dev = model.Page()
    page_dev.title=u'Development Home'
    page_dev.path=u'index'
    page_dev.name=u'Development Page'
    page_dev.content=u'This is the development home page.'
    page_dev.section=section_dev.id
    page_dev.before=page_svn.id
    meta.Session.add(page_dev)
    meta.Session.flush()

    page_home = model.Page()
    page_home.title=u'Home'
    page_home.path=u'index'
    page_home.name=u'Home'
    page_home.content=u'Welcome to the SimpleSite home page.'
    page_home.section=section_home.id
    page_home.before=section_dev.id
    meta.Session.add(page_home)
    meta.Session.flush()

    meta.Session.commit()
    log.info("Successfully set up.")
```

Now that you have updated the model and written a new `websetup.py`, you need the changes to be reflected in the underlying database. The easiest way of doing this is to create a new database from scratch. Once you've done this, you'll continue with the tutorial.

```
$ mv development.db development.db.old
$ paster setup-app development.ini
```

With these changes in place, IDs are shared between pages and sections. This means that nav ID 1 represents a section, whereas nav ID 2 represents a page. Visiting `http://localhost:5000/page/view/1` will therefore give a 404 Not Found response because there is no page with an ID of 1. In fact, the home page now has an ID of 6.

Creating the Controllers

Now that the model correctly supports the navigation structure, you'll need to think again about the controllers.

I mentioned before that you generally need a controller for every table in your database, but in this case you might think that there isn't a lot of point in having a controller for the navigation table because the `Section` and `Page` objects in the model handle all the functionality for that table anyway. It turns out that it can be useful to have a navigation controller as long as it can't be accessed directly because both the page and section controllers can inherit any functionality that affects only the nav table, such as the ability to move pages or sections. Create the navigation controller and a directory for its templates:

```
$ paster controller nav
$ cd simplesite/templates/derived
$ mkdir nav
```

Now change the `NavController` class. Delete the `index()` action, and add a `__before__()` method that prevents any of the actions you'll add later being called directly as a result of a URL being entered:

```
def __before__(self):
    abort(404)
```

You'll also need to add the following imports:

```
from simplesite import model
from simplesite.model import meta
```

With the nav controller in place, let's start by thinking about the section controller.

Your users will need to be able to add, edit, and remove sections just as they can with pages, but they probably won't need to list all the sections. Let's use the `template.py.txt` file you created earlier in the chapter as a starting point for the new controller:

```
$ cd simplesite/controllers
$ cp template.py.txt section.py
$ perl -pi -w -e 's/comment/section/g; s/Comment/Section/g;' section.py
```

Now let's also create a set of templates:

```
$ cd ../templates/derived
$ cp -r page section
$ cd section
$ perl -pi -w -e 's/page/section/g; s/Page/Section/g;' *.html
```

Delete the `section/list.html` template because you won't need it. Now restart the server if you stopped it to make these changes:

```
$ cd ../../../../
$ paster serve --reload development.ini
```

As usual, let's start by thinking about the FormEncode schema you're going to need. You'll need validators for each of the columns in the `nav` table. To validate the value of the `before` column (which is used to determine the order of pages and sections within a subsection), you'll need a custom validator. Since pages also have a value of `before`, they will need the same validator, so rather than defining the custom validator in the section controller, let's create the validators in the nav controller. Add this to the nav controller after the existing imports:

```
import formencode

class ValidBefore(formencode.FancyValidator):
    """Checks the ID specified in the before field is valid"""
    def _to_python(self, values, state):
        nav_q = meta.Session.query(model.Nav)
        # Check the value for before is in the section
        if values.get('before'):
            valid_ids = [nav.id for nav in nav_q.filter_by(
                section=values['section']).all()]
            if int(values['before']) not in valid_ids:
                raise formencode.Invalid("Please check the section "
                    "and before values", values, state)
        return values

class NewNavForm(formencode.Schema):
    allow_extra_fields = True
    filter_extra_fields = True
    name = formencode.validators.String(not_empty=True)
    path = formencode.validators.Regex(not_empty=True, regex='^[a-zA-Z0-9_-]+$')
    section = formencode.validators.Int(not_empty=True)
    before = formencode.validators.Int()
    chained_validators = [ValidBefore()]
```

The NewNavForm schema handles each of the fields but uses the ValidBefore chained validator to also check that the navigation node specified in before is either None (which means add the new section at the end of the existing section) or is the ID of a navigation node, which does exist in that section. You'll recall that chained validators are run only once after all the individual validators for each of the fields have been checked. The schema also uses a Regex validator to ensure that only allowed characters are used on the path.

Let's now use the NewNavForm schema as a basis for creating a schema for new sections. Add this to the section controller in place of the existing NewSectionForm schema:

```
from simplesite.controllers.nav import NewNavForm, ValidBefore

class UniqueSectionPath(formencode.validators.FancyValidator):
    "Checks that there isn't already an existing section with the same path"
    def _to_python(self, values, state):
        nav_q = meta.Session.query(model.Nav)
        query = nav_q.filter_by(section=values['section'],
            type='section', path=values['path'])
        if request.urlvars['action'] == 'save':
            # Ignore the existing ID when performing the check
            query = query.filter(model.Nav.id != int(request.urlvars['id']))
        existing = query.first()
        if existing is not None:
            raise formencode.Invalid("There is already a section in this "
                "section with this path", values, state)
        return values

class NewSectionForm(NewNavForm):
    chained_validators = [ValidBefore(), UniqueSectionPath()]
```

The UniqueSectionPath validator ensures there isn't another subsection with the same path in the section to which this section is being added. Although this code works well for adding a new section, there are some different constraints when editing a section. Sections cannot be moved to sections that are children of the section being moved. This means you need another validator to ensure that if the section is being edited, the new section is in a valid position. Here's the new validator and an EditNavForm that uses it. Add them to the section controller after the NewSectionForm you just added:

```
class ValidSectionPosition(formencode.FancyValidator):
    def _to_python(self, values, state):
        nav_q = meta.Session.query(model.Nav)
        if values.get('type', 'section') == 'section':
            # Make sure the section we are moving to is not already
            # a subsection of the current section
            section = nav_q.filter_by(id=int(values['section'])).one()
            current_section = nav_q.filter_by(id=request.urlvars['id']).one()
            while section:
                if section.section == current_section.id:
                    raise formencode.Invalid("You cannot move a section to "
                        "one of its subsections", values, state)
                if section.section == 1:
                    break
                section = nav_q.filter_by(id=section.section).first()
        return values

class EditSectionForm(NewNavForm):
    chained_validators = [
        ValidBefore(),
        UniqueSectionPath(),
        ValidSectionPosition()
    ]
```

The ValidSectionPosition validator iterates through each of the parent sections of the section to which you are trying to move the section you are editing. If it reaches the top of the navigation tree without finding the section you are moving, then you are allowed to move the section.

At this point, all the validators that will be shared between pages and sections are in the nav controller, and all the validators that the section needs are in the section controller, but they inherit from those in the navigation controller. You'll make the necessary changes to the page controller later in the chapter, so now let's think about the templates.

Both the page and the section will need the extra fields from the nav table, so create a new file called fields.html in the templates/derived/nav/ directory to be shared by the page and section templates. Add the following content:

```
${h.field(
    "Name",
    h.text(name='name'),
    required=True,
)}
${h.field(
    "Path",
    h.text(name='path'),
    required=True,
)}
```

```
${h.field(
    'Section',
    h.select(
        "section",
        id='section',
        selected_values=[],
        options=c.available_sections,
    ),
    required=True
)}
${h.field(
    "Before",
    h.text(
        "before",
        id='before',
    ),
)}
```

These fields will be used by both the page controller and the section controller. Update the `derived/section/fields.html` file to import and use the fields you've just created:

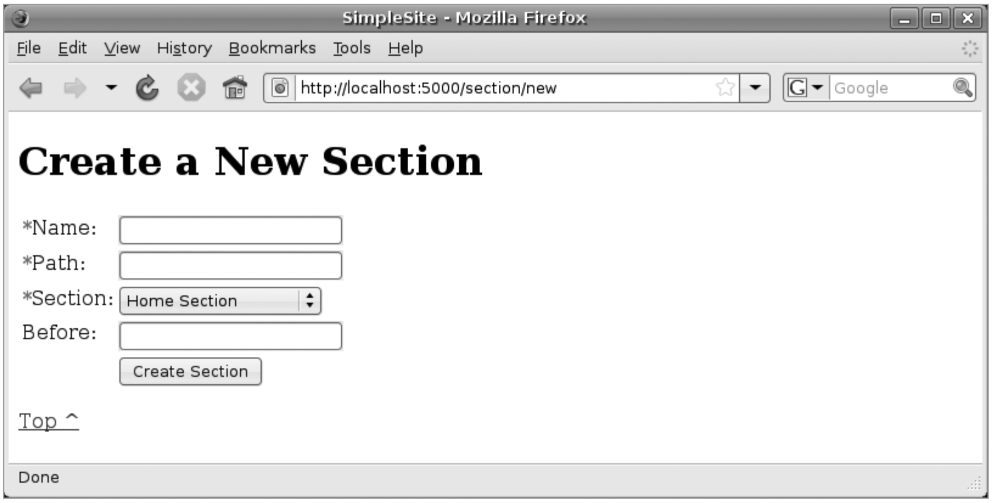
```
<%namespace file="/derived/nav/fields.html" name="fields" import="*"/>
## Nav fields
${fields.body()}
## Section fields would go here if there were any
```

Because of the way the templates are set up, these fields will be used in both the `derived/section/new.html` and `derived/section/edit.html` templates. You'll notice that the section field relies on the value of `c.available_sections`. You haven't set this up yet, so let's do that now by adding the following `__before__()` method to the section controller:

```
def __before__(self, id=None):
    nav_q = meta.Session.query(model.Nav)
    if id:
        nav_q=nav_q.filter_by(type='section').filter(model.nav_table.c.id!=int(id))
    else:
        nav_q = nav_q.filter_by(type='section')
    c.available_sections = [(nav.id, nav.name) for nav in nav_q]
```

Notice that you are using a query based on `model.Nav` here and specifying `type='section'` in the filter rather than querying `model.Section`. This is because the `nav` table contains the `name` column you need access to, but the section table doesn't.

At this point you'll be able to see the form for creating a new section by visiting `http://localhost:5000/section/new`. This is shown in Figure 14-6, but the section won't save correctly yet.


SimpleSite - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/section/new
Google

Create a New Section

*Name:
*Path:
*Section: Home Section
Before:
Create Section

Top ^

Done

Figure 14-6. *The create section form*

Simply adding the section to the table isn't enough. Because this is a hierarchy, you also need to modify the node this node appears before to update its value of `before` to point to the ID of the section the user is adding so that the ordering is correct. You could add a function to perform this task as a helper, but in this case the work mainly has to do with the model, so it would be better to add it there.

Rather than simply adding the function to the model module itself, you're going to add it as a static method to the `Nav` class. A static method in Python is one that is associated with the class itself and not the instance of a class. As such, it doesn't have a `self` argument. Update the `Nav` class in `model/__init__.py` to look like this:

```
class Nav(object):
    @staticmethod
    def add_navigation_node(nav, section, before):
        nav_q = meta.Session.query(Nav)
        new_before = nav_q.filter_by(section=section, before=before).first()
        if new_before is not None and new_before.id != nav.id:
            new_before.before = nav.id
```

You can now access this functionality as `model.Nav.add_navigation_node()` in your controllers without needing any additional imports, and it is clear what the functionality does. The `Section` and `Page` classes will also inherit this method, although you'll use the version attached to `Nav` to keep what is happening more explicit.

Update the section controller's `create()` action so that the navigation structure is correctly updated when a section is added by calling `model.Nav.add_navigation_node()`. Since a section isn't a lot of use on its own, let's also generate an index page for the section. To do this, you need to flush the session so that the `section` object gets assigned an ID. You can then use the ID to help create the index page. The finished code looks like this with the new lines bold:

```
@restrict('POST')
@validate(schema=NewSectionForm(), form='new')
def create(self):
    # Add the new section to the database
    section = model.Section()
    for k, v in self.form_result.items():
        setattr(section, k, v)
    meta.Session.add(section)
    model.Nav.add_navigation_node(section, self.form_result['section'],
        self.form_result['before'])
    # Flush the data to get the session ID.
    meta.Session.flush()
    index_page = model.Page()
    index_page.section = section.id
    index_page.path = 'index'
    index_page.title = 'Section Index'
    index_page.name = 'Section Index'
    index_page.content = 'This is the index page for this section.'
    meta.Session.add(index_page)
    meta.Session.commit()
    # Issue an HTTP redirect
    return redirect_to(controller='section', action='view', id=section.id)
```

Since you can't actually *see* a section, there isn't a lot of point in having the `create()` action redirect to a view of it. For this reason, delete the `view()` action and the `view.html` template because you won't use them. For the time being, after you create a new section, you'll get the error `"Action u'view' is not implemented"`. You'll fix this later in the chapter.

Now that you can create sections, let's think about editing them. First you'll need to update the `edit()` action so that the `values` reflect those of a section. It should look like this:

```
values = {
    'name': section.name,
    'path': section.path,
    'section': section.section,
    'before': section.before,
}
```

When you save a section, there is a chance you are moving it to another section. If this is the case, you need to formally remove it from the node hierarchy before adding it in the new location. You already have an `add_navigation_node()` method, so here's the `remove_navigation_node()` static method. Add this to the `Nav` class in `model/__init__.py` too:

```
@staticmethod
def remove_navigation_node(nav):
    nav_q = meta.Session.query(Nav)
    old_before = nav_q.filter_by(section=nav.section, before=nav.id).first()
    if old_before is not None:
        old_before.before = nav.before
```

Update the section controller's `save()` action to look like this. Make sure you update the redirect code and change the schema in the `@validate` decorator to use the `EditSectionForm` you created earlier.

```
@restrict('POST')
@validate(schema=EditSectionForm(), form='edit')
def save(self, id=None):
    section_q = meta.Session.query(model.Section)
    section = section_q.filter_by(id=id).first()
    if section is None:
        abort(404)
    if not(section.section == self.form_result['section'] and \
        section.before == self.form_result['before']):
        model.Nav.remove_navigation_node(section)
        model.Nav.add_navigation_node(section, self.form_result['section'],
           self.form_result['before'])
    for k,v in self.form_result.items():
        if getattr(section, k) != v:
            setattr(section, k, v)
    meta.Session.commit()
    session['flash'] = 'Section successfully updated.'
    session.save()
    # Issue an HTTP redirect
    return redirect_to(controller='section', action='view', id=section.id)
```

Once again, when you save a section after editing, you will be redirected to the nonexistent view() action. You'll fix this later too.

Next you'll need to look at the delete() action. Ideally you should not be able to delete a section while it still contains pages or subsections. However, if you deleted all the pages a section contained, then there would be no page on which to display a link to delete the section. Instead, you will set things up so that deleting a section also deletes its index page, but you can't delete a section if any other pages or sections exist within the section you're deleting. Update the delete() action so that it looks like this:

```
def delete(self, id=None):
    if id is None:
        abort(404)
    section_q = meta.Session.query(model.Section)
    section = section_q.filter_by(id=id).first()
    if section is None:
        abort(404)
    nav_q = meta.Session.query(model.Nav)
existing = nav_q.filter_by(section=id, type='section').filter(
        model.Page.path != 'index').first()
    if existing is not None:
        return render('/derived/section/cannot_delete.html')
    index_page = nav_q.filter_by(section=id, path='index', type='page').first()
    if index_page is not None:
    model.Nav.remove_navigation_node(index_page)
    meta.Session.delete(index_page)
model.Nav.remove_navigation_node(section)
    meta.Session.delete(section)
    meta.Session.commit()
    return render('/derived/section/deleted.html')
```

You'll need to create the derived/section/cannot_delete.html file with this content:

```
<%inherit file="/base/index.html"/>

<%def name="heading()"><h1>Cannot Delete</h1></%def>

<p>You cannot delete a section which still contains pages or subsections
other than the index page. Please delete the pages and subsections
first.</p>
```

That's it—now you can also delete sections, but before we move on, let's add the section links to the page footer so that users can access the functionality you've just implemented without having to type the URL directly. Edit templates/derived/page/view.html so that the footer() def looks like this:

```
<%def name="footer()">
## Then add our page links
<p>
  <a href="${h.url_for(controller='page', action='list')}">All Pages</a>
| <a href="${h.url_for(controller='page', action='new', ➥
section=c.page.section, before=c.page.before)}">New Page</a>
| <a href="${h.url_for(controller='page', action='edit', ➥
id=c.page.id)}">Edit Page</a>
| <a href="${h.url_for(controller='page', action='delete', ➥
id=c.page.id)}">Delete Page</a>
</p>
## Comment links
<p>
  <a href="${h.url_for(pageid=c.page.id, controller='comment', ➥
action='list', id=None)}">Comments (${str(c.comment_count)})</a>
| <a href="${h.url_for(pageid=c.page.id, controller='comment', ➥
action='new', id=None)}">Add Comment</a>
</p>
## Section links
<p>
  <a href="${h.url_for(controller='section', action='new', ➥
section=c.page.section, before=c.page.before)}">New Section</a>
| <a href="${h.url_for(controller='section', action='edit', ➥
id=c.page.section)}">Edit Section</a>
| <a href="${h.url_for(controller='section', action='delete', ➥
id=c.page.section)}">Delete Section and Index Page</a>
</p>
## Tag links
<p><a href="${h.url_for(controller='tag', action='list')}">All Tags</a>
| <a href="${h.url_for(controller='tag', action='new')}">Add Tag</a></p>
## Include the parent footer too
${parent.footer()}
</%def>
```

You'll notice that the call to h.url_for() to the section controller's new() action contains some extra arguments, section and before. When Routes' h.url_for() function gets passed arguments, and it doesn't recognize them; it will simply add them as parameters to the query string. In this case, the arguments represent information about the current page that can be used on the new() action to automatically populate some of the values. The URL generated might look like /section/new?section=1&before=7. To take advantage of these arguments, you will have to update the section controller's new() action to look like this:

```
def new(self):
    values = {}
    values.update(request.params)
    if values.has_key('before') and values['before'] == u'None':
        del values['before']
    return htmlfill.render(render('/derived/section/new.html'), values)
```

The `h.url_for()` call to create a new page has also had a similar change. It now takes both the section and the before value of the current page as arguments too, so now you can turn your attention to updating the page controller.

The Page Controller

Let's start by updating the page controller's `new()` action to accept the variables that will be passed to it when a user clicks the New Page link now that you've updated the arguments to `h.url_for()` in the page view template footer. The new action should look like this:

```
def new(self):
    values = {}
    values.update(request.params)
    if values.has_key('before') and values['before'] == u'None':
        del values['before']
    return htmlfill.render(render('/derived/page/new.html'), values)
```

The page controller also needs the same fields and functionality as the section controller. This is not surprising since both sections and pages inherit from the `Nav` class. Edit the `derived/page/fields.html` file to include fields from the `derived/nav/fields.html` file. Add the following at the top of the before the existing fields:

```
<%namespace file="/derived/nav/fields.html" name="fields" import="*"/>
## Nav fields
${fields.body()}
## Page fields
```

Now that the extra fields are in place, let's change the `NewPageForm` schema in the page controller to inherit from `NewNavForm` instead of `formencode.Schema` so that it has all the validators of `NewNavForm` as well as its own. You'll also need a `UniquePagePath` validator to ensure there isn't already a page with the same name in the current section. Add this to the top of `page.py` replacing the existing `NewPageForm` schema:

```
from simplesite.controllers.nav import NewNavForm, ValidBefore

class UniquePagePath(formencode.validators.FancyValidator):
    def _to_python(self, values, state):
        nav_q = meta.Session.query(model.Nav)
        query = nav_q.filter_by(section=values['section'],
            type='page', path=values['path'])
        if request.urlvars['action'] == 'save':
            # Ignore the existing id when performing the check
            query = query.filter(model.Nav.id != int(request.urlvars['id']))
        existing = query.first()
        if existing is not None:
            raise formencode.Invalid("There is already a page in this "
                "section with this path", values, state)
        return values
```

```
class NewPageForm(NewNavForm):
    allow_extra_fields = True
    filter_extra_fields = True
    content = formencode.validators.String(
        not_empty=True,
        messages={
            'empty':'Please enter some content for the page. '
        }
    )
    heading = formencode.validators.String()
    title = formencode.validators.String(not_empty=True)
    chained_validators = [ValidBefore(), UniquePagePath()]
```

Notice that the `NewPageForm` schema also has the same `ValidBefore()` chained validator as the `NewSectionForm`.

Now modify the page controller's `create()` action so that new pages are added in the correct place in the hierarchy and the redirect code is updated.

```
@restrict('POST')
@validate(schema=NewPageForm(), form='new')
def create(self):
    # Add the new page to the database
    page = model.Page()
    for k, v in self.form_result.items():
        setattr(page, k, v)
    meta.Session.add(page)
    model.Nav.add_navigation_node(page, self.form_result['section'],
        self.form_result['before'])
    meta.Session.commit()
    # Issue an HTTP redirect
    return redirect_to(controller='page', action='view', id=page.id)
```

And just as with the section controller, update the redirect code and change the lines in the `save()` action before the line `for k,v in self.form_result.items():`

```
@restrict('POST')
@validate(schema=NewPageForm(), form='edit')
def save(self, id=None):
    page_q = meta.Session.query(model.Page)
    page = page_q.filter_by(id=id).first()
    if page is None:
        abort(404)
    if not(page.section == self.form_result['section'] and \
        page.before == self.form_result['before']):
        model.Nav.remove_navigation_node(page)
        model.Nav.add_navigation_node(page, self.form_result['section'],
            self.form_result['before'])
    for k,v in self.form_result.items():
        if getattr(page, k) != v:
            setattr(page, k, v)
    meta.Session.commit()
    session['flash'] = 'Page successfully updated.'
    session.save()
    # Issue an HTTP redirect
    return redirect_to(controller='page', action='view', id=page.id)
```

Now you'll need to update the `edit()` method so that it also populates all the new values:

```
values = {
    'name': page.name,
    'path': page.path,
    'section': page.section,
    'before': page.before,
    'title': page.title,
    'heading': page.heading,
    'content': page.content,
}
```

For this to work, you'll need a similar `__before__()` action used in the section controller to set `c.available_sections`:

```
def __before__(self):
    nav_q = meta.Session.query(model.Nav)
    c.available_sections = [(nav.id, nav.name) for nav in ➥
nav_q.filter_by(type='section')]
```

You'll also need to update the `delete()` action. This is much easier than it is for sections; simply add the following line:

```
def delete(self, id=None):
    if id is None:
        abort(404)
    page_q = meta.Session.query(model.Page)
    page = page_q.filter_by(id=id).first()
    if page is None:
        abort(404)
    meta.Session.execute(delete(model.pagetag_table,
        model.pagetag_table.c.pageid==page.id))
    model.Nav.remove_navigation_node(page)
    meta.Session.delete(page)
    meta.Session.commit()
    return render('/derived/page/deleted.html')
```

Changing the Routing

At this point you now have all the functionality necessary for a fully working web site. These are the only missing elements:

- The ability to enter a proper URL rather than the ID of the page you want to view
- Navigation controls such as tags, a menu, and breadcrumbs

Before I get into too much detail about the code, I'll discuss exactly how these things will be implemented. You want a setup where the URL appears as if it is mapping to a directory structure of sections and subsections. So, rather than visiting `/page/view/4` to view the SVN page in the development section, you will be able to access it as `/dev/svn`. To do this, you need to get Routes to understand your alternative URL structure. You'll also want some navigation components. You'll use a set of tabs for the main navigation. Any page or section that is in the home section will be displayed on these tabs. For any page that isn't in the home section, a navigation menu will be also generated to display the links in that section. Finally, you'll have a breadcrumb trail so that users can see where they are in the navigation hierarchy. There's a lot to do, so let's get started.

You'll implement both these elements, but you'll start with the routing. Ideally, you want to be able to specify a route that will handle a URL not already matched by the other routes. You can do this by using a wildcard part, as you learned in Chapter 9. One approach could be to add a route like this as the last route in SimpleSite's `config/routing.py` file:

```
map.connect('*url', controller='page', action='nav')
```

This would redirect any URL not already matched by the other routes to the page controller's `nav()` action from where the appropriate dispatch can be performed; however, there is also a slightly neater solution that involves having the page or section ID calculated as part of the matching process. This avoids needing to use a controller action for dispatch.

Create a named route called `path` as the last route in the route map, and specify a function condition on the route called `parse()` and a filter on the route named `build()`. Conditions and filters are advanced Routes functionality that I discussed in Chapter 9. Here's how the route map should look, with the new route shown in bold:

```
def make_map():
    """Create, configure and return the routes Mapper"""
    map = Mapper(directory=config['pylons.paths']['controllers'],
                 always_scan=config['debug'], explicit=True)
    map.minimization = False

    # The ErrorController route (handles 404/500 error pages); it should
    # likely stay at the top, ensuring it can always be resolved
    map.connect('/error/{action}', controller='error')
    map.connect('/error/{action}/{id}', controller='error')

    # CUSTOM ROUTES HERE

    map.connect(
        '/page/{pageid}/{controller}/{action}',
        requirements=dict(pageid='\d+'),
    )
    map.connect(
        '/page/{pageid}/{controller}/{action}/{id}',
        requirements=dict(pageid='\d+', id='\d+'),
    )
    map.connect('/{controller}/{action}')
    map.connect('/{controller}/{action}/{id}')
    map.connect('path', '*url', conditions={'function':parse}, _filter=build)
    return map
```

Add the `parse()` and `build()` functions to the top of the `config/routing.py` file before the `make_map()` function:

```
from simplesite import model

def parse(environ, result):
    url = result.pop('url')
    try:
        environ['simplesite.navigation'] = navigation_from_path(url)
    except NoPage, e:
        result['controller'] = 'nav'
        result['action'] = 'nopage'
        result['section'] = e.section
        result['path'] = e.path
    except NoSection, e:
        result['controller'] = 'nav'
        result['action'] = 'nosection'
        result['section'] = e.section
        result['path'] = e.path
    except NotFound, e:
        # This causes the route to not match
        return False
    else:
        result['controller'] = 'page'
        result['action'] = 'view'
        result['id'] = environ['simplesite.navigation']['page'].id
    return True

def build(routing_variables):
    controller = routing_variables.get('controller')
    action = routing_variables.get('action')
    id = routing_variables.get('id')
    del routing_variables['id']
    routing_variables['url'] = model.Nav.nav_to_path(id)
    return routing_variables
```

When Routes can't match any URL against the other routes, the `'path'` named route you've just added gets tested. This causes the `parse()` condition to be called, which in turn calls the `navigation_from_path()` function with the current URL as its argument.

I'll show you the `navigation_from_path()` function in a moment, but let's think about what it has to do. Its main job is to match the URL entered against a section or page that already exists so that the correct routing variables can be set up. If the URL doesn't match an existing section or a page, the function should ideally determine whether it is possible to create a page or section at that URL. If it is possible, you'll need some mechanism to let the user know they can create a section or page. If it isn't, a 404 Not Found response should be returned.

It turns out that performing these checks requires the `navigation_from_path()` function to look up each part of the URL to check that it exists and to determine whether it is a section or page. Since these checks are already being performed, it makes sense for the same function to also gather the information that will be required to generate the navigation components you'd like to use in the site including top-level tabs, a menu, and breadcrumbs. This is precisely what the function does, returning a dictionary with the following keys:

`breadcrumbs`: A list of all the sections in the navigation hierarchy up to the current node, followed by the final page or section. Each item in the list has an attribute added called `path_info`, which is the full URL `PATH_INFO` to that page or section that can be used to help generate links.

`menu`: A list of all the pages and sections in the section to which the URL resolves.

`tabs`: The pages and sections in the topmost section. Used in the main navigation tabs.

`page`: The page object for the page the URL resolves to or the `index` page if the URL resolves to a section.

This dictionary returned is then added to the `environ` dictionary as the `simplesite.navigation` key so that it can be accessed in the rest of the application.

■ **Note** Some people would argue that this sort of functionality is better implemented as Web Server Gateway Interface middleware. You'll learn about middleware in Chapter 16 and are free to reimplement the previous functionality a different way if you prefer.

The `navigation_from_path()` function is shown here together with the `menu()` function it relies on and three `Exception` classes that are used as part of the process. The code looks like this and should be added to the top of `config/routing.py` after the `build()` function:

```
class NoPage(Exception):
    pass

class NoSection(Exception):
    pass

class NotFound(Exception):
    pass

def navigation_from_path(path_info):
    result = {}
    nav_q = model.meta.Session.query(model.Nav)
    path_parts = path_info.split('/')
    result['breadcrumbs'] = []
    if path_info.endswith('/'):
        path_info += 'index'
    path_parts = path_info.split('/')
    for path in path_parts[:-1]:
        s = nav_q.filter_by(type='section', path=path).first()
        if s:
            result['breadcrumbs'].append(s)
        else:
            if path_info.endswith('/index') and \
                len(result['breadcrumbs']) == len(path_info.split('/'))-2:
                exception = NoSection('No section exists here')
                exception.section = result['breadcrumbs'][-1].id
                exception.path = path_parts[-2]
                raise exception
            else:
                raise NotFound('No section can be created here')
    result['page'] = nav_q.filter_by(type='page',
        section=result['breadcrumbs'][-1].id, path=path_parts[-1]).first()
    if result['page'] is None:
        if len(result['breadcrumbs']) == len(path_info.split('/'))-1:
            exception = NoPage('No page exists here')
            exception.section = result['breadcrumbs'][-1].id
            exception.path = path_parts[-1]
            raise exception
        else:
            raise NotFound('No page can be created here')
    result['breadcrumbs'].append(result['page'])
    # Add the path_info
    cur_path = ''
    for breadcrumb in result['breadcrumbs']:
        cur_path +=breadcrumb.path
        breadcrumb.path_info = cur_path
        if isinstance(breadcrumb, model.Section):
            breadcrumb.path_info = cur_path + '/'
            cur_path += '/'
    result['menu'] = menu(nav_q, result['breadcrumbs'][-2].id,
        result['breadcrumbs'][-2].path_info)
    result['tabs'] = menu(nav_q, result['breadcrumbs'][0].id,
        result['breadcrumbs'][0].path_info)
    return result

def menu(nav_q, sectionid, path_info):
    # There might also be child sections
    last = None
    navs = [nav for nav in nav_q.filter_by(section=sectionid).order_by(
        model.nav_table.c.before.desc()).all()]
    for nav in navs:
        if nav.before is None:
            # This is our last node
            last = nav
            break
    menu_dict = dict([[nav.before, nav] for nav in navs])
    if not last:
        raise Exception('No last node found')
    # Iterate over the nodes building them up in the correct order
    menu = [last]
    while len(menu) < len(navs):
        id = menu[0].id
        if not menu_dict.has_key(id):
            raise Exception("This section doesn't have an item %s to go "
                "before %r id %s"%(id, menu[0].name, menu[0].id))
        item = menu_dict[menu[0].id]
        menu.insert(0, item)
    f_menu = []
    for menu_item in menu:
        menu_item.path_info = path_info + menu_item.path
        if isinstance(menu_item, model.Section):
            menu_item.path_info += '/'
        elif menu_item.path_info.endswith('/index'):
            menu_item.path_info = menu_item.path_info[:-5]
        f_menu.append(menu_item)
    return f_menu
```

As you can see, the `navigation_to_path()` function looks at each part of path to check that it exists, building up a list of breadcrumbs as it does. If it matches a page or section, it will also generate data structures for top-level tabs and a navigation menu containing links to other sections and pages in the same section as the section to which the URL resolves.

If the URL entered can't be matched, the function checks to see whether it could represent a section if that section was created. If it does, a `NoSection` exception is raised. This is caught in the `parse()` function and results in the nav controller's `nosection()` action being called. A similar thing happens if the URL resolves to a page that could exist if it were created, only a `NoPage` exception is raised, eventually resulting in a call to the nav controller's `nopage()` action.

If the URL doesn't resolve to a page or section and the component above it doesn't exist either, then a `NotFound` exception is raised, causing the `parse()` function to return `False`, which in turn tells Routes that the `'path'` named route hasn't matched. This results in a 404 Not Found page being displayed as normal.

Let's implement the `nosection()` and `nopage()` actions. Replace the `NavController` class with this (you don't need the `__before__()` method anymore):

```
class NavController(BaseController):

    def nopage(self, section, path):
        return render('/derived/nav/create_page.html')

    def nosection(self, section, path):
        return render('/derived/nav/create_section.html')
```

You'll also need to create the templates on which these actions rely. Create `derived/nav/create_page.html` like this:

```
<%inherit file="/base/index.html"/>

<%def name="heading()"><h1>Create Page</h1></%def>
<p><a href="${h.url_for(controller='page', action='new',
    section=c.section, path=c.path)}">Create a new page here</a>.</p>
```

and create `derived/nav/create_section.html` like this:

```
<%inherit file="/base/index.html"/>

<%def name="heading()"><h1>Create Section</h1></%def>
<p><a href="${h.url_for(controller='section', action='new',
    section=c.section, path=c.path)}">Create a new section here</a>.</p>
```

Now when you visit a URL that doesn't exist but for which a page or a section could be created, you will be shown a page with a link allowing you to create it.

To view the page, the attributes `c.menu`, `c.tabs`, and `c.breadcrumbs` must be set. Add the lines in bold to the end of the page controller's `view()` method to obtain the values calculated during the processing of the routes and set them for use in the template.

```
def view(self, id=None):
    if id is None:
        abort(404)
    page_q = meta.Session.query(model.Page)
    c.page = page_q.filter_by(id=int(id)).first()
    if c.page is None:
        abort(404)
    c.comment_count = meta.Session.query(model.Comment).filter_by(pageid=id).count()
    tag_q = meta.Session.query(model.Tag)
    c.available_tags = [(str(tag.id), tag.name) for tag in tag_q]
    c.selected_tags = {'tags':[tag.id for tag in c.page.tags]}
    c.menu = request.environ['simplesite.navigation']['menu']
    c.tabs = request.environ['simplesite.navigation']['tabs']
    c.breadcrumbs = request.environ['simplesite.navigation']['breadcrumbs']
    return render('/derived/page/view.html')
```

For this to work, you need to use the named route `'path'` when generating URLs to the page or section controller's `view()` actions so that the `build()` filter function can generate the correct URL.

Update all the calls to `redirect_to()` in the page controllers to this:

```
return redirect_to('path', id=page.id)
```

Update all the calls to `redirect_to()` in the section controller to look like this:

```
return redirect_to('path', id=section.id)
```

This build() function relies on a nav_to_path() static method that you should add to the Nav class in your model after the existing static methods:

```
class Nav(object):

    ... existing methods ...

    @staticmethod
    def nav_to_path(id):
        nav_q = meta.Session.query(Nav)
        nav = nav_q.filter_by(id=id).one()
        path = nav.path
        if nav.type=='section':
            path += '/'
        while nav.section is not None:
            nav = nav_q.filter_by(type='section', id=nav.section).one()
            path = nav.path+'/'+path
        return path
```

There are two other places that need updating to use the new route. Edit templates/derived/page/list.html, and replace these lines:

```
h.url_for(
    controller=u'page',
    action='view',
    id=unicode(page.id)
)
```

with the following:

```
h.url_for('path', id=page.id)
```

Then edit templates/derived/comment/view.html, and update the link back to the page the comment was posted on to look like this:

```
<p><a href="${h.url_for('path', id=c.comment.pageid)}">➥
Visit the page this comment was posted on.</a></p>
```

At this point, everything is in place to test the new code, but you are advised to create a new database because the navigation structure is fairly fragile if the validators aren't in place and because it is possible that as you've been building and testing the functionality you may have introduced some errors.

Delete the database and run this:

```
$ paster setup-app development.ini
```

Start the server again, visit http://localhost:5000/, and you should see the home page exactly as if you had visited http://localhost:5000/page/view/6 before making the routing changes.

Adding the Navigation Elements

Now all you need to do is add the navigation elements to the pages. Start by editing templates/base/index.html to add this import to the top:

```
<%namespace name="navigation" file="/component/navigation.html" import="*" />\
```

Then add templates/component/navigation.html with the following content:

```
<%!
import simplesite.model as model
%>
<%def name="breadcrumbs()">
    % if c.page and c.page.id != 1:
    <div id="breadcrumbs"><p>${render_breadcrumbs(c.breadcrumbs)}</p></div>
    % endif
</%def>

<%def name="tabs()">
    % if c.tabs:
    <div id="maintabs">
        <ul class="draglist">
            ${render_list(c.tabs, c.breadcrumbs[1].path,
                type_=c.breadcrumbs[1].type, id='li1_', class_='list2')}
        </ul>
    </div>
    % endif
</%def>

<%def name="menu()">
% if len(c.breadcrumbs) > 2:
    <div id="menu">
        <h2>Section Links</h2>
        <ul class="draglist">
                ${render_list(c.menu, c.breadcrumbs[-1].path,
                    type_=c.breadcrumbs[1].type, id='li1_', class_='list2')}
        </ul>
    </div>
% endif
</%def>

<%def name="render_list(items, current, id, class_)">
% for item in items:
    % if item.path == current and item.type == type_:
<li class="${class_} active" id="${id}${str(item.id)}"><span class="current"><a
    href="${item.path_info}" id="current">${item.name}</a></span></li>\
    % else:
<li class="${class_}" id="${id}${str(item.id)}"
    onclick="document.location ='${item.path_info}'"
><span><a href="${item.path_info}">${item.name}</a></span></li>\
    % endif
% endfor
</%def>

<%def name="render_breadcrumbs(breadcrumbs)">
    % for i, item in enumerate(breadcrumbs):
    % if i < len(breadcrumbs) - 1:
        <a href="${item.path_info}">${item.name}</a> &gt;
    % elif isinstance(c.breadcrumbs[-1], model.Section):
        ${item.name} &gt;
    % else:
        ${item.name}
    % endif
    % endfor
</%def>
```

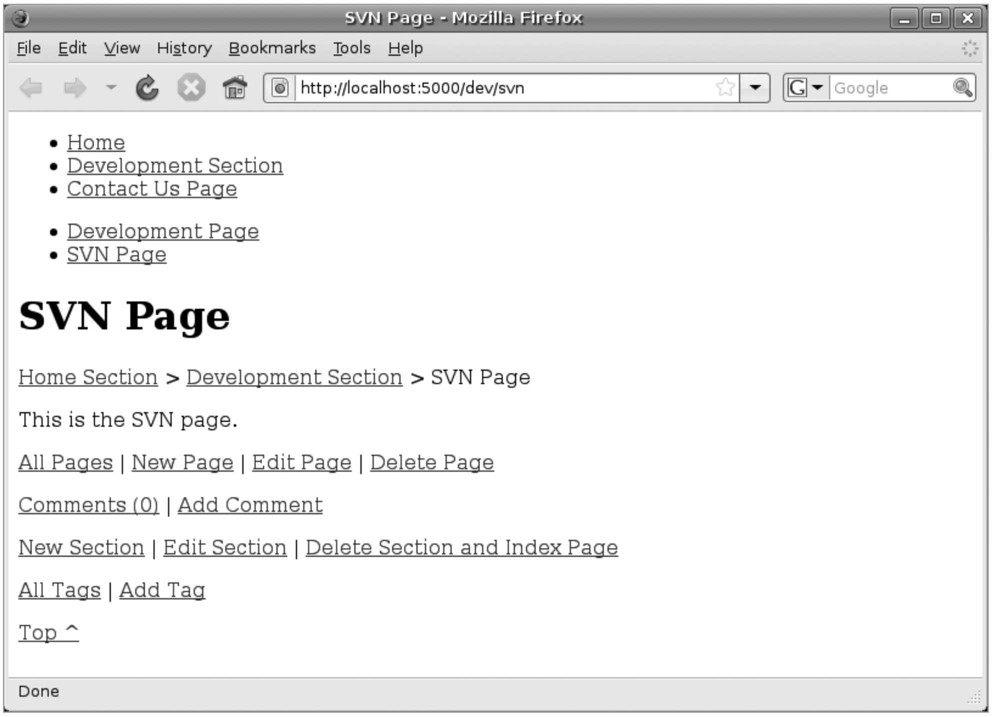
Finally, edit `templates/base/index.html` again to replace the following defs:

```
<%def name="tabs()"></%def>
<%def name="menu()"></%def>
<%def name="heading()"><h1>${c.heading or 'No Title'}</h1></%def>
<%def name="breadcrumbs()"></%def>
```

with the following versions:

```
<%def name="tabs()">${navigation.tabs()}</%def>
<%def name="menu()">${navigation.menu()}</%def>
<%def name="heading()"><h1>${c.heading or 'No Title'}</h1></%def>
<%def name="breadcrumbs()">${navigation.breadcrumbs()}</%def>
```

With these changes in place, as shown in Figure 14-7, you can test the navigation components you've created.


SVN Page - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/dev/svn
Google
Done

- Home
- Development Section
- Contact Us Page

- Development Page
- SVN Page

SVN Page

Home Section > Development Section > SVN Page

This is the SVN page.

All Pages | New Page | Edit Page | Delete Page

Comments (0) | Add Comment

New Section | Edit Section | Delete Section and Index Page

All Tags | Add Tag

Top ^

Figure 14-7. *The breadcrumbs, main links, and section links*

Adding Some Style

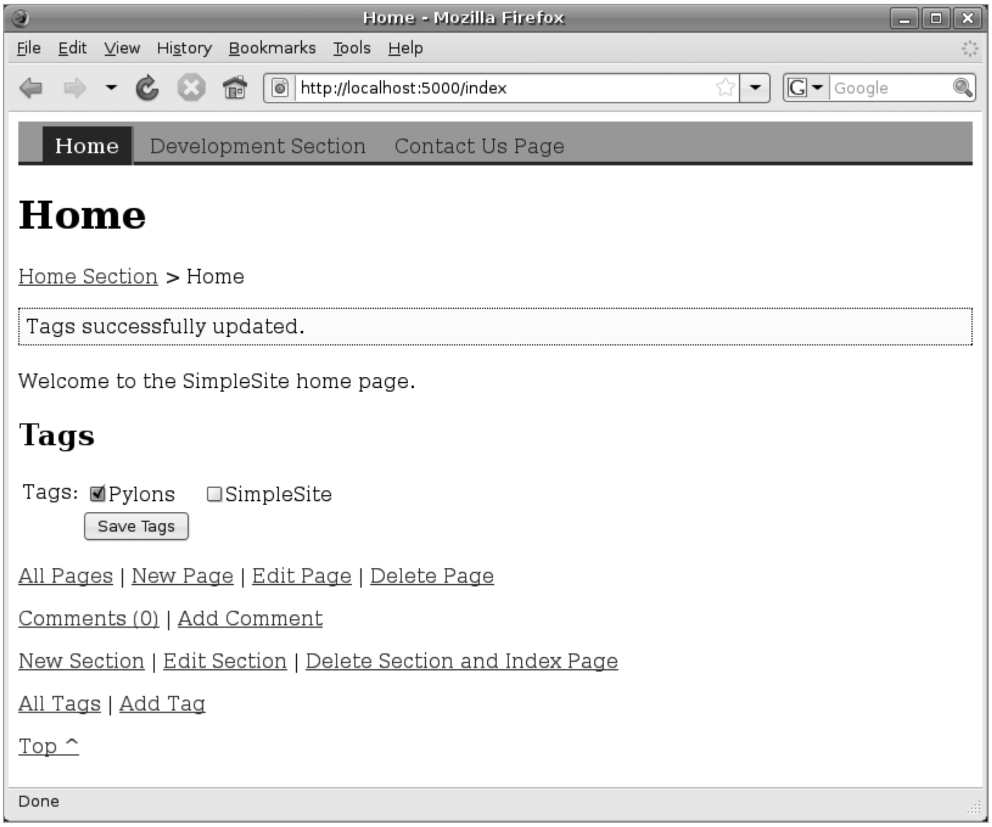
Now that all the functionality for the SimpleSite is in place, let's add some style to `public/css/main.css`. It would be good if the navigation tabs looked like tabs rather than a bulleted list. These styles will fix this; add them to the end of the file:

```
#maintabs ul {
    margin: 0px;
    padding: 0px;
    height: 23px;
}
#maintabs {
    background: #87AFD7;
    border-bottom: 3px solid #113958;
    margin: 0;
    padding: 10px 0 0px 17px;
}
#maintabs li {
    list-style: none;
    margin: 0;
    display: inline;
}
#maintabs li a {
    padding: 6px 10px;
    margin-left: 3px;
    border-bottom: none;
    text-decoration: none;
}
#maintabs li a:link { color: #113958; }
#maintabs li a:visited { color: #113958; }
#maintabs li a:hover {
    color: #000;
    background: #fff;
    border-color: #227;
}
#maintabs li a#current
{
    background: #113958;
    color: #fff;
    font-weight: bold;
    border-right: 2px solid #468AC7;
}
```

■Tip If you find yourself frequently styling bulleted lists in this way, a useful site is listamatic at `http://css.maxdesign.com.au/listamatic/`; it provides quite a few different styles to apply to the same style sheet.

At this point, all the core functionality of SimpleSite is in place. You can add comments, tag pages, create sections and subsections, and move pages and sections around. Now is a good time to test the application to check that it behaves as you expect it to and that you haven't made any mistakes.

Figure 14-8 shows what the application looks like with some tags added.



Home

Home Section > Home

Tags successfully updated.

Welcome to the SimpleSite home page.

Tags

Tags: ☑Pylons ☐SimpleSite

All Pages | New Page | Edit Page | Delete Page

Comments (0) | Add Comment

New Section | Edit Section | Delete Section and Index Page

All Tags | Add Tag

Top ^

Figure 14-8. *CSS and tags*

Summary

You accomplished an awful lot in this chapter. You implemented a full comment and tag system, used SQLAlchemy's sophisticated inheritance features, shared code between different validators and templates, and built some sophisticated extensions to Routes.

In the next chapter, you'll learn about JavaScript and CSS. You'll then update SimpleSite to use a CSS grid. You'll add some Ajax so that the `before` text field is implemented as a select field whose values change when you select a different section, and you'll add some animation to the flash message.