CHAPTER 18

Authentication and Authorization

There comes a point in most projects when you need to be able to restrict access to certain parts of your web site. This might be so that you can create an administration area or because your users are storing private information and want to know their data is password protected. To do this, you will need to implement a security system that can confirm the identity of a user and then restrict the pages each user has access to based on their permissions to each area. These two processes are known as *authentication* and *authorization*.

Authentication is typically performed by asking a visitor for their username and password. If the visitor enters the correct password, you can be fairly sure they are who they claim to be. Once a user has been authenticated, your security system needs some way of remembering who the user is so that they are not asked to enter their username and password the next time they try to access a restricted page. This could be achieved by setting a cookie on the user's browser so that the next time they visit a page, the security system can read a secret code from the cookie and determine which user is accessing the page from the secret code.

Just because the user has been authenticated, it doesn't necessarily mean they should be authorized to access the page they are trying to visit. If, for example, you signed into a Yahoo! Mail account with your username and password, you wouldn't expect to be able to read the e-mails of other users, because you wouldn't be authorized to do so.

Authorization is usually performed by checking the authenticated user has the appropriate permissions to access the page they are trying to visit. The permission check can be as simple as ensuring that a user has in fact signed in but can also be very sophisticated involving checks about the user's roles, group, or even which computer the user is accessing the web site from.

In this chapter, you'll start off by looking at how Pylons handles private data before learning how to implement a basic security system from scratch. You'll then move on to considering an authentication and authorization tool called AuthKit and looking at its main features. In the next chapter, you'll use the knowledge you've gained about AuthKit in this chapter to add a sign-in form and role-based permissions system to the SimpleSite application.

Private Data

The easiest way to prevent *any* user from accessing a controller action is to make that action private. You'll remember from Chapter 9 that Pylons treats any controller method that starts with an underscore (_) character as private and will not dispatch to it as a controller action. This means controller methods beginning with _ are not directly publicly accessible.

Let's create a controller to demonstrate this. Run the following commands to create a test project; you won't need SQLAlchemy or Google App Engine support:

```
$ paster create --template=pylons AuthTest
$ cd AuthTest
$ paster controller example
$ paster serve --reload development.ini
```

Now add the following content to the `example` controller:

```
class ExampleController(BaseController):

    def hello(self):
        return self._result()

    def _result(self):
        return 'Hello World!'
```

If you start the server and visit `http://localhost:5000/example/_result`, you will be shown a 404 Not Found error document, but if you visit `http://localhost:5000/example/hello`, you will see the `Hello World!` message because the public action `hello()` is able to return the value from the private method `_result()`.

A Homegrown Solution

Although making certain controller actions private prevents any user from accessing a method, you will often need to restrict access to just certain users. You can do this by using Pylons' `__before__()` method and session functionality.

Create a new controller for the `AuthTest` project called `homegrown`:

```
$ paster controller homegrown
```

You'll recall that on each request the `__before__()` method of the controller is called before the controller action is called. Let's set up a simple session variable named `user` that will be used to store the username of the authenticated user. If the `user` session variable isn't set, you can assume no user is signed in. If it is set, you can set the `REMOTE_USER` environment variable.

```
class HomegrownController(BaseController):

    def __before__(self, action, **params):
        user = session.get('user')
        if user:
            request.environ['REMOTE_USER'] = user

    def signin(self):
        if len(request.params) > 1 and \
         request.params['password'] == request.params['username']:
            session['user'] = request.params['username']
            session.save()
            return redirect_to(controller='homegrown', action="private")
        else:
            return """\
               <html>
                 <head><title>Please Login!</title></head>
                 <body>
                   <h1>Please Login</h1>
                   <form action="signin" method="post">
```

```
                <dl>
                  <dt>Username:</dt>
                  <dd><input type="text" name="username"></dd>
                  <dt>Password:</dt>
                  <dd><input type="password" name="password"></dd>
                </dl>
                <input type="submit" name="authform" />
                <hr />
              </form>
            </body>
          </html>
        """

    def public(self):
        return 'This is public'

    def private(self):
        if request.environ.get("REMOTE_USER"):
            return 'This is private'
        else:
            return redirect_to(controller='homegrown', action="signin")
```

In this example, you can access `http://localhost:5000/homegrown/public` without signing in, but if you visit `http://localhost:5000/homegrown/private`, you will be redirected to the sign-in form at `http://localhost:5000/homegrown/signin` to sign in. If you enter a username that is the same as the password, you will be shown the private message. You will be able to continue to see the private message until you clear the Pylons session cookie by closing your browser.

This example works perfectly well for the straightforward case described earlier, but when you start dealing with complex permissions and different authentication methods, it quickly becomes preferable to use an authentication and authorization framework.

AuthKit

AuthKit is a complete authentication and authorization framework for WSGI applications and was written specifically to provide Pylons with a flexible approach to authentication and authorization. AuthKit can be used stand-alone with its user management API or integrated with other systems such as a database. You'll see both of these approaches in this chapter and the next.

AuthKit consists of three main components:

Authentication middleware: Intercepts any permission errors or HTTP responses with a 401 status code and presents a user with a way to sign in. Various authentication methods can be used, and the middleware is also responsible for setting the `REMOTE_USER` environ variable once a user has signed in.

Permission objects: Represent a particular permission that a user may or may not have.

Authorization adaptors: Check the permissions, triggering a `PermissionError`, which is intercepted by the authentication middleware if the permission check doesn't pass. If no `PermissionErrors` are raised, the user is considered to be authorized.

In this chapter, you'll learn about each of these components in turn before looking at AuthKit's more advanced features.

Of course, AuthKit is just one of the authentication and authorization tools available for Pylons. There may be occasions when you want to handle all authentication yourself in your application rather than delegating responsibility to AuthKit. Although AuthKit provides a basic platform on which to build, you should also be willing to look at the AuthKit source code and use it as a basis for your own ideas.

■ **Note** One toolset that is proving to be particularly popular at the time of writing is `repoze.who`, which is part of the repoze project to help make Zope components available to WSGI projects such as Pylons. If you are interested in `repoze.who`, you should visit the web site at `http://static.repoze.org/whodocs/`.

Authentication Middleware

AuthKit is actually very straightforward to integrate into an existing Pylons project. You'll remember from Chapter 3 that a web browser finds out what type of response has been returned from the server based on the HTTP status code. There are two HTTP status codes that are particularly relevant to authentication and authorization. A 401 status code tells the browser that the user is not authenticated, and a 403 status code tells the browser that the user is not authorized (which you may have seen described in error pages as `Forbidden`). AuthKit's authentication middleware works at the HTTP level by responding to 401 status responses so that the authentication middleware can work with *any* application code that is HTTP compliant, regardless of whether AuthKit is used for the authorization checks.

Let's create a new project to use with AuthKit. Run the following commands to install AuthKit and create a test project; again, you won't need SQLAlchemy support:

```
$ easy_install "AuthKit>=0.4.3,<=0.4.99"
$ paster create --template=pylons AuthDemo
$ cd AuthDemo
$ paster serve --reload development.ini
```

To set up the authentication middleware, edit the AuthTest project's `config/middleware.py` file, and add the following import at the end of the existing imports at the top of the file:

```
import authkit.authenticate
```

Then add this line:

```
app = authkit.authenticate.middleware(app, app_conf)
```

just before these lines and at the same indentation level:

```
# Display error documents for 401, 403, 404 status codes (and
# 500 when debug is disabled)
if asbool(config['debug']):
    app = StatusCodeRedirect(app)
else:
    app = StatusCodeRedirect(app, [400, 401, 403, 404, 500])
```

The authentication middleware has to be set up before the error documents middleware because you don't want any 401 responses from the controllers being changed to error pages before the authentication middleware has a chance to present a sign-in facility to the user.

If you had set the `full_stack` option to `false` in the Pylons config file, you would need to add the AuthKit authenticate middleware before these lines; otherwise, it wouldn't get added:

```
if asbool(full_stack):
    # Handle Python exceptions
    app = ErrorHandler(app, global_conf, **config['pylons.errorware'])
```

■ **Tip** You'll remember from Chapter 16 that middleware is simply a component that sits between the server and the controller action that is being called and has an opportunity to change the response that the controller action returns.

Now that the authentication middleware is set up, you need to configure it. AuthKit is designed to be completely configurable from the Pylons config file and has a number of required options that tell AuthKit which authentication method you want to use and how AuthKit should check whether the username and password that have been entered are correct.

Add the following options to the end of the `[app:main]` section:

```
authkit.setup.method = form, cookie
authkit.form.authenticate.user.data = visitor:open_sesame
authkit.cookie.secret = secret string
```

These options set up AuthKit to use form and cookie-based authentication. This also sets up a user named `visitor` with the password `open_sesame`. These options will be passed to the authentication middleware via the `app_conf` argument.

At this stage, you can test that the middleware is set up and ready to be used. Create a new controller called `auth`:

```
$ paster controller auth
```

and add the following action:

```
def private(self):
    response.status = "401 Not authenticated"
    return "You are not authenticated"
```

If you visit `http://localhost:5000/auth/private`, the 401 status will be returned and intercepted by the AuthKit middleware, and you will see the default sign-in screen for form and cookie authentication displayed at the same URL, as shown in Figure 18-1.

Figure 18-1. *The default sign-in screen when using the form and cookie authentication method*

There is just one more change you need to make in order to properly test your authentication setup. At the moment, the `private` action always returns a 401 HTTP status code, even if a user is authenticated. This means the first time you sign in, you'll see the error document for a 401 response, but on subsequent requests you'll see the sign-in form again.

The authenticate middleware automatically adds a key to the `environ` dictionary named `REMOTE_USER` if a user is authenticated. The value of the key is the username of the authenticated user. Let's use this fact to update the `private()` action so that a user can see the message once they are authenticated:

```
def private(self):
    if request.environ.get("REMOTE_USER"):
        return "You are authenticated!"
    else:
        response.status = "401 Not authenticated"
        return "You are not authenticated"
```

If you start the server and visit `http://localhost:5000/auth/private`, you will now be able to sign in. If you sign in with the username and password you specified in the config file (`visitor` and `open_sesame`), you will see the message `You are authenticated!`. If you refresh the page, you will notice you are still signed in because AuthKit set a cookie to remember you.

To implement a facility for signing out, you will need to add this line to your config file:

```
authkit.cookie.signoutpath = /auth/signout
```

This tells AuthKit that when a user visits the URL `http://localhost:5000/auth/signout`, an HTTP header should be added to the response to remove the cookie. AuthKit doesn't know what else should be included in the response, so you will also need to add a `signout()` action to the controller at that URL so that the visitor is not shown a 404 Not Found page when they sign out:

```
def signout(self):
    return "Successfully signed out!"
```

After you have restarted the server, you can test the sign-out process by visiting `http://localhost:5000/hello/signout`. If you use Firebug to look at the HTTP headers, you'll notice AuthKit has added this to the response headers to sign you out:

```
Set-Cookie    authkit=""; Path=/
```

■ **Caution** Setting the header to remove the AuthKit cookie happens in the AuthKit middleware, *after* the response from the controller action. The controller action itself plays no part in the sign-out process, so it could return the text `You are still signed in`, but the user would still be signed out. More dangerously, if you entered an incorrect path for the `authkit.cookie.signoutpath` option, the user would still get a message saying they are signed out when they are actually still signed in. It is therefore very important that you enter the correct path in the `authkit.cookie.signoutpath` option.

Authorization and Permissions

By using the AuthKit middleware, you have been able to quickly implement a fully working authentication system, but so far you have had to perform the authorization by manually checking `environ["REMOTE_USER"]` to see whether there was a user signed in. AuthKit can help simplify authorization too.

The Authorization Decorator

Let's start by looking at how the `@authorize` decorator can be used with AuthKit permission objects to protect entire controller actions.

Add the following imports to the top of the auth controller:

```
from authkit.authorize.pylons_adaptors import authorize
from authkit.permissions import RemoteUser, ValidAuthKitUser, UserIn
```

Next change the `private()` action to look like this:

```
@authorize(RemoteUser())
def private(self):
    return "You are authenticated!"
```

This code is clearly a lot neater than what you had previously. If you sign out or clear your browser's cookies, you'll see it behaves exactly as it did before, allowing you to view the private message `You are authenticated`, but only after you have signed in.

In this example, the `@authorize` decorator simply prevents the action from being called if the permission check fails. Instead, it raises a `PermissionError`, which is derived from an `HTTPException` and is therefore converted either by Pylons or by AuthKit itself into a response with either a 401 status code or a 403 status code depending on whether the permission failed because of an authentication error or an authorization error. The response is then handled by the authentication middleware triggering a response, resulting in the sign-in screen you've just used.

The `RemoteUser` permission might not be the best permission to use in this case since it simply checks that a `REMOTE_USER` is set and could therefore potentially grant permission to someone who wasn't in the list of users specified in the config file if some other part of the system were to set the `REMOTE_USER` environment variable. Instead, it might be better to use the `ValidAuthKitUser` permission, which does a similar thing but allows only valid AuthKit users. You imported the `ValidAuthKitUser` at the same time you imported `RemoteUser`, so you can use it like this:

```
@authorize(ValidAuthKitUser())
def private(self):
    return "You are authenticated!"
```

You can also use an AuthKit permission object named `UserIn` to specify that only certain users are allowed. Add another user to the config file like this:

```
authkit.form.authenticate.user.data = visitor:open_sesame
                                      nobody:password
```

Then change the way the permission is used like this:

```
@authorize(UserIn(["visitor"]))
def private(self):
    return "You are authenticated!"
```

This time, even if you signed in as `nobody`, you would still not be authorized to access the action because the permission check will authorize only the user named `visitor`.

■ **Tip** You don't actually need to instantiate a permission in the decorator itself; you can also create a permission instance elsewhere and use it as many times as you like in different `authorize()` decorators.

You've now seen how to use the `@authorize` decorator to check permissions before an action is called, but there are actually two other ways of checking permissions that automatically raise the correct `PermissionError` if a permission check fails:

- The authorize middleware for protecting a whole WSGI application
- The `authorized()` function for checking a permission within a code block

Let's look at these next.

The Authorization Middleware

To protect a whole application, you can use AuthKit's authorization middleware. You need to set this up in your project's `config/middleware.py` file *before* the authentication middleware to use it. If you set it up after the authentication middleware, any `PermissionError` raised wouldn't get intercepted by the authentication middleware. Let's test this on the AuthDemo project. First import the authorization middleware and `ValidAuthKitUser` permission at the end of the imports at the top of `config/middleware.py`:

```
import authkit.authorize
from authkit.permissions import ValidAuthKitUser
```

Then set up the authorization middleware *before* the authentication middleware:

```
permission = ValidAuthKitUser()
app = authkit.authorize.middleware(app, permission)
app = authkit.authenticate.middleware(app, app_conf)
```

Now every request that comes through your Pylons application will require the user to be signed in as a valid AuthKit user. To test this, add a new action to the auth controller, which looks like this:

```
def public(self):
    return "This is still only visible when you are signed in."
```

Even though this doesn't have an `@authorize` decorator, you still won't be able to access it until you are signed in because of the presence of the authorization middleware. Try visiting `http://localhost:5000/auth/public` to test it.

■ **Caution** Using the authorization middleware in this way will protect only the WSGI applications defined above it in `config/middleware.py`. In this case, the middleware is set up above the `Cascade`, so any requests that are served by the `StaticURLParser` application will not be covered. This means that files served from your project's `public` directory won't be protected. You can protect them too by moving all the AuthKit middleware to below the `Cascade` in `config/middleware.py`.

The Authorization Function

Of course, sometimes you will want to be able to check a permission from within an action or a template. AuthKit provides a function for doing this too named `authorized()`. The function returns `True` if the permission check passes or `False` otherwise. If you comment out the authorization middleware you set up in `config/middleware.py` a few moments ago (leaving the authenticate middleware), you will be able to test this function:

```
# permission = ValidAuthKitUser()
# app = authkit.authorize.middleware(app, permission)
app = authkit.authenticate.middleware(app, app_conf)
```

First edit the `auth` controller to import the `authorized()` function:

```
from authkit.authorize.pylons_adaptors import authorized
```

Then update the `public()` action to look like this:

```
def public(self):
    if authorized(UserIn(["visitor"])):
        return "You are authenticated!"
    else:
        return "You are not authenticated!"
```

Using the `authorized()` function will never actually trigger a sign-in; if you want to trigger a sign-in, you either need to manually return a response with a status code of `401` like this:

```
def public(self):
    if authorized(UserIn(["visitor"])):
       return "You are authenticated!"
    else:
response.status = "401 Not authenticated"
return "You are not authenticated!"
```

or raise the appropriate permission error:

```
def public(self):
    if authorized(UserIn(["visitor"])):
        return "You are authenticated!"
    else:
from authkit.permissions import NotAuthenticatedError
```

You can test this at `http://localhost:5000/auth/public`.

■ **Caution** Although all the permission objects that come with AuthKit can be used with the `authorized()` function, it is possible to create custom permissions that will not work. This is because permissions can perform checks on both the request and the response, and although the authorization decorator and authorization middleware both have access to the response, the `authorized()` function does not and so is not capable of performing checks on permissions that depend on the response.

Protecting a Controller

Sometimes you might want to protect a controller rather than an individual action or an entire WSGI application. Since all Pylons controllers have a `__before__()` method, adding an `@authorize` decorator to `__before__()` is equivalent to protecting the entire controller.

For example:

```
class AuthController(BaseController):

    @authorize(ValidAuthKitUser())
    def __before__(self):
        pass

    def public(self):
        return "This is still only visible when you are signed in."
```

Groups, Roles, and Permissions

So far, you have seen how to authenticate users and how to use permissions to authorize them, but the permissions you've used haven't been very complicated.

One common and extremely flexible pattern used in many authorization systems is that of groups and roles. Under this pattern, each user can be assigned any number of roles that relate to tasks they might use the system for. They can also be a member of one group, usually a company or organization.

As an example, if you were designing a content management web site, you might choose the following roles:

Writer: Someone tasked with creating new articles

Editor: Someone with permission to update existing articles

Reviewer: Someone who reviews articles before they are published

Admin: Someone who has permission to set the roles of other users

Certain users might have more than one role, and others might not have any roles at all.

An appropriate use for groups would be when you give access to the same web site to users from different companies and each user is either from one company or the other. Of course, you could implement the same functionality by creating a role for each company and checking which role each user had, but using groups is simpler. If you find you need to assign more than one group to the same person, then you should be using roles, not groups.

■ **Tip** The groups and roles pattern even works when conceptually you have more than one application to which each person should be assigned roles. You can append application names to the front of the roles so that, for example, `intranet_editor` is a role for an editor on an intranet application and `website_editor` is a role for an editor on a public-facing web site application.

You can specify AuthKit groups and roles in the configuration file along with the user information. Here is the configuration to set up roles for four users of our content management system:

```
authkit.form.authenticate.user.data = ben:password1 writer editor reviewer admin
                                      james:pasword2 writer
                                      graham:password3 writer reviewer
                                      philip:password4 editor reviewer
```

As you can see, user information is specified in the format `username:password role1 role2 role3`, and so on. Each new user is on a new line, and roles are separated by a space character. In this example, Ben and Philip are Editors; Ben, James, and Graham are Writers; Ben, Graham, and Philip are Reviewers; and Ben is the only Admin.

If our imaginary content management system was used by two web framework communities, it might be useful to be able to specify which community each user belonged to. You can do this using AuthKit's group functionality:

```
authkit.form.authenticate.user.data = ➥
    ben:password1:pylons writer editor reviewer admin
    simon:password5:django writer editor reviewer admin
```

As you can see, the group is specified after the password and before the roles by using another colon (`:`) character as a separator.

■ **Caution** You have to be a little careful when using groups because your users can never be members of more than one group. In this instance, you would hope that people from both the Django and Pylons communities might contribute to each others' projects, so in this instance it might be more appropriate to create two new roles, `django` and `pylons`, and assign the `pylons` role to Ben and the `django` role to Simon. Then, if at a later date Simon wants to work on Pylons, he can simply be assigned the `pylons` role too.

Groups and roles can be checked in a similar way to other permissions using the authorization middleware, the `@authorize` decorator, or the `authorized()` function.

The two important permissions for checking groups and roles are `HasAuthKitRole` and `HasAuthKitGroup`. They have the following specification:

```
HasAuthKitRole(roles, all=False, error=None)
HasAuthKitGroup(groups, error=None)
```

The `roles` and `groups` parameters are a list of the acceptable role or group names. If you specify `all=True` to `HasAuthKitRole`, the permission will require that all the roles specified in `roles` are matched; otherwise, the user will be authorized if they have been assigned any of the roles specified. Here are some examples of using these permission objects:

```
from authkit.permissions import HasAuthKitRole, HasAuthKitRole, And

# User has the 'admin' role:
HasAuthKitRole('admin')

# User has the 'admin' or 'editor' role:
HasAuthKitRole(['admin', 'editor'])

# User has the both 'admin' and 'editor' roles:
HasAuthKitRole(['admin', 'editor'], all=True)

# User has no roles:
HasAuthKitRole(None)

# User in the 'pylons' group:
HasAuthKitGroup('pylons')

# User in the 'pylons' or 'django' groups:
HasAuthKitGroup(['pylons', 'django'])

# User not in a group:
HasAuthKitGroup(None)
```

It is also possible to combine permissions using the `And` permission class. This example would require that the user was an administrator in the Pylons group:

```
And(HasAuthKitRole('admin'), HasAuthKitGroup('pylons'))
```

In addition to the permissions for roles and groups, AuthKit also comes with permission objects for limiting users to particular IP addresses or for allowing them to access the resource only at particular times of day:

```
# Only allow access from 127.0.0.1 or 10.10.0.1
permission = FromIP(["127.0.0.1", "10.10.0.1"])

# Only allow access between 6pm and 8am
from datetime import time
permission = BetweenTimes(start=time(18), end=time(8))
```

User Management API

All the examples so far have been using the `authkit.users.UsersFromString` driver to extract all the username, password, group, and role information from the config file for use with the permission objects. If you had lots of users, it could quickly become unmanageable to store all this information in the config file, so AuthKit also comes with a number of other drivers.

The first driver you will learn about is the `UsersFromFile` driver from the `authkit.users` module. This driver expects to be given a filename from which to load all the user information. For example, if you stored your user information in the file `C:\user_information.txt`, you might set up your config file like this:

```
authkit.form.authenticate.user.type = authkit.users:UsersFromFile
authkit.form.authenticate.user.data = C:/users_information.txt
```

The `users_information.txt` file should have the user data specified in the same format as has been used so far in this chapter with one user per line. For example:

```
ben:password1 writer editor reviewer admin
james:pasword2 writer
graham:password3 writer reviewer
philip:password4 editor reviewer
```

Of course, you may want to perform checks on the users in your application code as well as on permissions. AuthKit allows you to do this too via a key in the environment called `authkit.users`, which is simply an instance of the particular instance class you are using.

You can use it like this:

```
>>> users = request.environ['authkit.users']
>>> users.user_has_role('ben', 'admin')
True
>>> users.user_has_group('ben', 'django')
False
>>> users.list_roles()
['admin', 'editor', 'reviewer', 'writer']
```

The full API documentation is available on the Pylons web site.

Cookie Options

The AuthKit cookie-handling code supports quite a few options:

`authkit.cookie.name`: The name of the cookie; the default is `auth_tkt`.

`authkit.cookie.includeip`: Should be `True` or `False`. If `True`, the IP address of the user is also included in the encrypted ticket to prevent the same cookie from being used from a different IP address and hence to try to improve security.

`authkit.cookie.signoutpath`: A path that, when visited, will cause the cookie to be removed and the user to therefore be signed out. The application should still display a page at this path; otherwise, the user will see a 404 page and think there is a problem.

`authkit.cookie.secret`: A string you can set used to make the encryption on the cookie data more random. You should set a secret and make sure it isn't publically available.

`authkit.cookie.enforce`: If a cookie `expires` param is set and this is set to `True`, then there will also be server-side checking of the expire time to ensure the user is signed out even if the browser fails to remove the cookie.

`authkit.cookie.params.*`: The available options are `expires`, `path`, `comment`, `domain`, `max-age`, `secure`, and `version`. These are the values described in RFC 2109, but for convenience `expires` can be set as the number of seconds and will be converted automatically.

So, for example, to have a cookie that expires after 20 seconds with a cookie name `test` and the comment `this is a comment`, you would set these options:

```
authkit.cookie.secret = random string
authkit.cookie.name = test
authkit.cookie.params.expires = 20
authkit.cookie.params.comment = this is a comment
```

If you want a more secure cookie, you can add these options:

```
authkit.cookie.enforce = true
authkit.cookie.includeip = true
```

The first option enforces a server-side check on the cookie expire time as well as trusting the browser to do it. The second checks the IP address too and will work only if the request comes from the same IP address the cookie was created from.

AuthKit's default cookie implementation is based on the Apache `mod_auth_tkt` cookie format. This means the cookie itself contains the username of the signed-in user in plain text. You can set another option called `authkit.cookie.nouserincookie` if you'd prefer AuthKit used a session store to hold the username, but this option breaks compatibility with `mod_auth_tkt` and means you have to move the AuthKit authentication middleware to above the `SessionMiddleware` in your project's `config/middleware.py` file so that AuthKit can use the Beaker session.

```
authkit.cookie.nouserincookie = true
```

Alternative Methods

So far, our discussion about authentication, groups, and roles has centered around AuthKit's User Management API. AuthKit also provides six ways of authenticating users including HTTP basic and digest authentication, and of course, it provides APIs for implementing your own authentication method. To use one of the alternative authentication methods, you just need to change the AuthKit options in your config file.

As an example, to use HTTP digest authentication, you could change the AuthKit options in your config file to look like this:

```
authkit.setup.method = digest
authkit.digest.authenticate.user.data = visitor:open_sesame
authkit.digest.realm = 'Test Realm'
```

Now when you access a controller action that causes a permission check to fail, the browser will display an HTTP digest dialog box prompting you to sign in. Once you are signed in, you will be able to access the controller action as before. The process of handling the sign-in and setting the `REMOTE_USER` variable is done automatically for you by AuthKit as before, so the rest of your application can remain unchanged.

AuthKit has similar setups for HTTP basic authentication and OpenID authentication and also has other plug-ins that can redirect users to a URL to sign in or forward the request to a controller inside your Pylons application.

Functional Testing

As you'll remember from Chapter 12, creating functional tests for key controllers is highly recommended. If your controller actions are protected by AuthKit, you will need to emulate a user being signed in by setting the `REMOTE_USER` environment variable when setting up your tests. Here's how to set up a user called `james` by passing the `extra_environ` argument when using the `get()` method to simulate a GET request:

```
from authdemo.tests import *

class TestAuthController(TestController):

    def test_index(self):
        response = self.app.get(
url(controller='auth', action='public'),
            extra_environ={'REMOTE_USER':'james'}
        )
        # Test response...
        assert "you are signed in" in response.body
```

Alternatively, AuthKit 0.4.1 supports two further options for helping you handle tests:

```
authkit.setup.enable = false
authkit.setup.fakeuser = james
```

The first of these options completely disables the AuthKit authenticate middleware and makes all the authorization facilities return that the permission check passed. This enables you to test your controller actions as though AuthKit wasn't in place. Some of your controller actions might expect a `REMOTE_USER` environment variable, so this is set up with the value of the `authkit.setup.fakeuser` option, if it is present. Both these options should be used only in `test.ini` and never on a production site.

How useful the previous options are depends on the complexity of your setup. For more complex setups, you are likely to want to specify environment variables on a test-by-test basis using the `extra_environ` argument.

■ **Caution** Always remember to set `authkit.setup.enable = true` on production sites; otherwise, all authentication is ignored and authorization checks always return that the user has the appropriate permission. Although useful for testing, this makes all your controller actions public, so it is not a good idea on a production site.

General Security Considerations

So far, everything we have discussed has been around how to authenticate a user and check their permissions in order to authorize them to perform certain actions. A very important aspect of this is making sure the authentication is secure and that no one can get hold of the usernames and passwords your users use. Security as a whole is a huge topic on which many books have been written, but you can take three steps to greatly reduce the risk of an intruder obtaining a user's username and password:

- Ensure other users of the system cannot read passwords from your Pylons config file. This is especially relevant if you are deploying your Pylons application in a shared hosting environment where other users of the system might be able to read database passwords and other information from your `development.ini` file.
- Ensure a user never sends their password in any form over an unencrypted connection such as via e-mail or over HTTP.
- Never directly store a user's password anywhere in your system.

Secure Sockets Layer

If you are using a form and a cookie or HTTP basic authentication, then the user's password is transmitted in plain text to the server. This means that anyone on the network could potentially monitor the network traffic and simply read the username and password. It's therefore important that on a production site using these authentication methods you set up a secure connection for the authentication using Secure Sockets Layer (SSL) to encrypt the communication between the browser and server during authentication.

Even HTTP digest authentication, which does use some encryption on the password, isn't particularly secure because anyone monitoring the network traffic could simply send the encrypted digest and be able to sign onto the site themselves, so even if you are using digest authentication, it is worth using SSL too.

To set up an SSL certificate, you need two things:

- A private key
- A certificate

The certificate is created from a certificate-signing request that you can create using the private key.

The private key can be encrypted with a password for extra security, but every time you restart the server, you will need to enter the password. The certificate-signing request (CSR) is then sent to a certificate authority (CA) that will check the details in the certificate-signing request and issue a certificate. Anyone can act as a certificate authority, but most browsers will automatically trust certificates only from the main certificate authorities. This means that if you choose a lesser-known certificate authority or you choose to sign the certificate yourself, your users will be shown a warning message asking them whether they trust the certificate.

For production sites, you should always choose one of the major certificate authorities such as VeriSign or Thawte because virtually all browsers will automatically trust certificates issued by them. CAcert.org is an initiative to provide free certificates, but these are not trusted by all commonly used browsers yet.

On a Linux platform with the `openssh` package installed, you can run the following command from a terminal prompt to create the key:

```
$ openssl genrsa -des3 -out server.key 1024
```

You'll see output similar to the following:

```
Generating RSA private key, 1024 bit long modulus
.....................++++++
.................++++++
unable to write 'random state'
e is 65537 (0x10001)
Enter pass phrase for server.key:
```

If you want to use a key without a passphrase, you can either leave out the `-des3` option or create an insecure version from the existing key like this:

```
$ openssl rsa -in server.key -out server.key.insecure
```

You should keep the private key `server.key` private and not send it to anyone else.

Once you have a key, you can use it to generate a certificate-signing request like this:

```
$ openssl req -new -key server.key -out server.csr
```

The program will prompt you to enter the passphrase if you are using a secure key. If you enter the correct passphrase, it will prompt you to enter your company name, site name, e-mail, and so on. Once you enter all these details, your CSR will be created, and it will be stored in the `server.csr` file. The common name you enter at this stage must match the full domain name of the site you want to set up the certificate for; otherwise, the browser will display a warning about a domain mismatch. It is worth checking with the certificate authority about the exact details it requires at this stage to ensure the CSR you generate is in the format required by the certificate authority.

Now that you have your certificate-signing request `server.csr`, you can send it to the certificate authority. The CA will confirm that all the details you've entered are correct and issue you a certificate that you can store as `server.crt`.

If you want to sign the certificate yourself, you can do so with this command:

```
$ openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt
```

■ **Caution** If you are planning on using your secure server in a production environment, you probably need a CA-signed certificate. It is not recommended to use self-signed certificates because of the warnings the browser will show the user.

Now that you have your private key and the certificate, you can use them to set up your server. How to set up SSL depends on the server you are using to deploy your Pylons application. You should consult your server's documentation for more information.

■ **Note** If you are using Apache, there is a good entry in the Pylons Cookbook describing how to set up SSL. You can find it at `http://wiki.pylonshq.com/display/pylonscookbook/Setting+up+Apache+and+SSL+for+Pylons`.

The Pylons server (started with the `paster serve` command) also supports SSL as long as you install the pyOpenSSL package. The Paste HTTP server requires the certificate and the private key to be added to the same file, so if you want to use your certificate and the key you generated earlier, you need to create a `server.pem` file like this:

```
$ cat server.crt server.key > server.pem
$ chmod 400 server.pem
```

Edit your project's `development.ini` file, and change the `[server:main]` section so it uses the following options marked in bold:

```
[server:main]
host = 127.0.0.1
ssl_pem = server.pem
port = 443
```

Port 443 is the default port for HTTPS. If you restart the server and visit `https://localhost/`, you should be able to access your Pylons application. You'll need the pyOpenSSL package installed though. Make sure the URL starts with `https` and not `http`; otherwise, you won't be able to connect to your Pylons application. Also make sure you change the host to `0.0.0.0` if you want to be able to access the server from a different machine on the network.

Encrypted Passwords

As was mentioned earlier, it is a good idea never to store users' passwords anywhere on your system in case the worst happens and someone breaks in and steals that information. One way to avoid this is to set up a function to encrypt the passwords before they are stored. One drawback to this approach is that if the user forgets their password, you are not able to send them a reminder by e-mail because you do not ever store the original password. Instead, you have to randomly generate a new password for them and send them an e-mail asking them to sign in with the new password and then change the password to something more memorable.

You can set up encrypted passwords with form authentication by adding the following to your AuthKit config:

```
authkit.setup.method = form, cookie
authkit.cookie.secret = secret string
authkit.form.authenticate.user.data = visitor:9406649867375c79247713a7fb81edf0
authkit.form.authenticate.user.encrypt = authkit.users:md5
authkit.form.authenticate.user.encrypt.secret = some secret string
```

The `authkit.form.authenticate.user.encrypt.secret` option allows you to specify a string that will be used to make the password encryption even harder to break.

Once this option is enabled, you will need to manually convert all existing passwords into their encrypted forms. Here is a simple program that converts the passwords you have used so far in this chapter into their encrypted forms using the secret `"some secret string"`. As you can see the encrypted password in the previous example corresponds to `password1`:

```
from authkit.users import md5

passwords = [
    'password1',
    'password2',
    'password3',
    'password4',
    'password5',
]

for password in passwords:
    print md5(password, "some secret string")
```

The result is as follows:

```
9406649867375c79247713a7fb81edf0
4e64aba9f0305efa50396584cfbee89c
aee8149aca17e09b8a741654d2efd899
2bc5a9b5c05bb857237d93610c98c98f
873f0294070311a707b941c6315f71f8
```

You can try replacing the passwords in the config file with these passwords. As long as you set the config options listed here, everything should still work without the real passwords being stored anywhere. This is more secure because an attacker would still need the original password in order to sign in, even if they knew the encrypted version.

Summary

Although the tools and techniques you've learned about in this chapter will enable you to write fairly advanced authorization and authentication systems that will be adequate for a lot of cases, you may sometimes need to do things slightly differently. AuthKit also supports custom authentication functions and a SQLAlchemy driver for storing the user information.

In the next chapter, you'll apply some of the knowledge you've gained in this chapter to the SimpleSite application you've been working on throughout the book, and you'll see some more techniques involving AuthKit, including how it integrates with SQLAlchemy and how to use a template to give the sign-in screen a theme.