Handling Authentication & Authorization

Warning: This page is a work in progress; errors may exist, and additional content is forthcoming.

So far we've built a simple contact manager, and added support for a related model (Addresses). This has shown how to use many of the basics, but there are a few more things you'd want before exposing this to the outside world. One of those is authentication and authorization. Django includes support that works for many projects, which is what we'll use.

Authentication

In order to use the included authentication support, the `django.contrib.auth` and `django.contrib.sessions` applications needs to be included in your project.

Django enables thes by default when you create a project, as you can see in `addressbook/settings.py`.

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    # 'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'contacts',
)
```

In addition to installing the application, the middleware needs to be installed, as well.

```
MIDDLEWARE_CLASSES = (
    'django.middleware.common.CommonMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    # Uncomment the next line for simple clickjacking protection:
    # 'django.middleware.clickjacking.XFrameOptionsMiddleware',
)
```

If you'll recall, during the first run of `syncdb`, Django asked if we wanted to create a superuser account. It did so because we had the application installed already.

The stock Django auth model supports Users, Groups, and Permissions. This is usually sufficient unless you're integrating with an existing authentication backend.

`django.contrib.auth` provides a set of views to support the basic authentication actions such as login, logout, password reset, etc. Note that it includes *views*, but not *templates*. We'll need to provide those for our project.

For this example we'll just add support for login and logout views in our project. First, add the views to `addressbook/urls.py`.

```
urlpatterns = patterns('',
    url(r'^login/$', 'django.contrib.auth.views.login'),
    url(r'^logout/$', 'django.contrib.auth.views.logout'),
```

Both the login and logout view have default template names (`registration/login.html` and `registration/logged_out.html`, respectively). Because these views are specific to our project and not our re-usable Contacts application, we'll create a new `templates/registration` directory inside

of `addressbook`:

```
$ mkdir -p addressbook/templates/registration
```

And tell Django to look in that directory for templates by setting `TEMPLATE_DIRS` in `addressbook/settings.py`.

```
TEMPLATE_DIRS = (
    # Put strings here, like "/home/html/django_templates" or "C:/www/django/tem
    # Always use forward slashes, even on Windows.
    # Don't forget to use absolute paths, not relative paths.
    'addressbook/templates',
)
```

Within that directory, first create `login.html`.

```
{% extends "base.html" %}

{% block content %}

{% if form.errors %}
<p>Your username and password didn't match. Please try again.</p>
{% endif %}

<form method="post" action="{% url 'django.contrib.auth.views.login' %}">
{% csrf_token %}
<table>
<tr>
    <td>{{ form.username.label_tag }}</td>
    <td>{{ form.username }}</td>
</tr>
<tr>
    <td>{{ form.password.label_tag }}</td>
    <td>{{ form.password }}</td>
</tr>
</table>

<input type="submit" value="login" />
<input type="hidden" name="next" value="{{ next }}" />
</form>

{% endblock %}
```

The login template inherits from our `base.html` template, and shows the login form provided by the view. The hidden `next` field allows the view to redirect the user to the page requested, if the login request was triggered by a permission failure.

The logout template, `logged_out.html`, is simpler.

```
{% extends "base.html" %}

{% block content %}

Logged out!

{% endblock %}
```

All it needs to do is provide a message to let the user know the logout was successful.

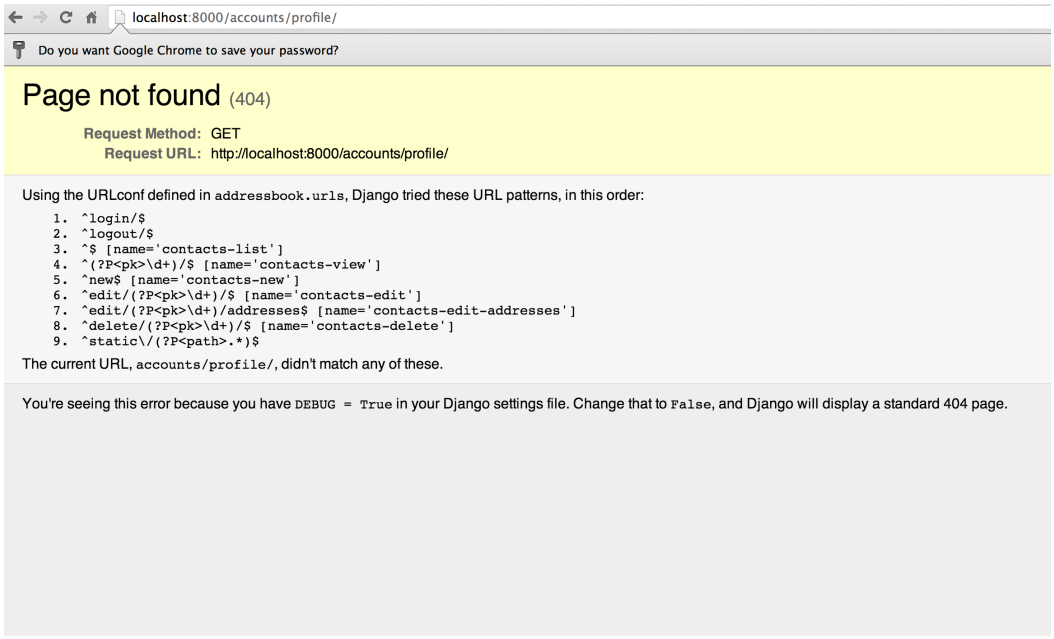
If you run your development server now using `runserver` and visit `http://localhost:8000/login`, you'll see the login page. If you login with bogus credentials, you should see an error message. So let's try logging in with the super user credential you created earlier.

Why no name for the URL patterns?

XXX

Creating an Admin User

XXX



Wait, what? Why is it visiitng `/accounts/profile`? We never typed that. The login view wants to redirect the user to a fixed URL after a successful login, and the default is `/accounts/profile`. To override that, we'll set the `LOGIN_REDIRECT_URL` value in `addressbook/settings.py` so that once a user logs in they'll be redirected to the list of contacts.

```
LOGIN_REDIRECT_URL = '/'
```

Now that we can log in and log out, it'd be nice to show the logged in user in the header and links to login/logout in the header. We'll add that to our `base.html` template, since we want that to show up everywhere.

```
<body>
  <div>
    {{ user }}
    {% if user.is_anonymous %}
    <a href="{% url 'django.contrib.auth.views.login' %}">login</a>
    {% else %}
    <a href="{% url 'django.contrib.auth.views.logout' %}">logout</a>
    {% endif %}
  </div>
```

Authorization

Having support for login and logout is nice, but we're not actually using it right now. So we want to first make our Contact views only available to authenticated users, and then we'll go on to associated contacts with specific Users, so the application could be used for multiple users.

Django includes a suite a functions and decorators that help you guard a view based on authentication/authorization. One of the most commonly used is login_required. Unfortunately, applying view decorators to class based views remains a little cumbersome. There are essentially two methods: decorating the URL configuration, and decorating the class. I'll show how to decorate the class.

Class based views have a `dispatch()` method that's called when an URL pattern matches. The `dispatch()` method looks up the appropriate method on the class based on the HTTP method and then calls it. Because we want to protect the views for all HTTP methods, we'll override and decorate that.

In `contacts/views.py` we'll create a class mixin that ensures the user is logged in.

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator

class LoggedInMixin(object):

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super(LoggedInMixin, self).dispatch(*args, **kwargs)
```

This is a *mixin* because it doesn't provide a full implementation of a view on its own; it needs to be *mixed* with another view to have an effect.

Once we have it, we can add it to the class declarations in `contacts/views.py`. Each view will have our new `LoggedInMixin` added as the first superclass. For example, `ListContactView` will look as follows.

```
class ListContactView(LoggedInMixin, ListView):

    model = Contact
    template_name = 'contact_list.html'

    def get_queryset(self):

        return Contact.objects.filter(owner=self.request.user)
```

Just as `LOGIN_REDIRECT_URL` tells Django where to send people *after* they log in, there's a setting to control where to send them when they *need* to login. However, this can also be a view name, so we don't have to bake an explicit URL into the settings.

```
LOGIN_URL = 'django.contrib.auth.views.login'
```

Checking Ownership

Checking that you're logged in is well and good, but to make this suitable for multiple users we need to add the concept of ownership. There are three steps for

1. Record the Owner of each Contact
2. Only show Contacts the logged in user owns in the list
3. Set the Owner when creating a new one

First, we'll go ahead and add the concept of an Owner to the Contact model.

In `contacts/models.py`, we add an import and another field to our model.

```
from django.contrib.auth.models import User
...
class Contact(models.Model):

    first_name = models.CharField(
        max_length=255,
    )
    last_name = models.CharField(
        max_length=255,

    )

    email = models.EmailField()

    owner = models.ForeignKey(User)

    def __str__(self):

        return ' '.join([
            self.first_name,
            self.last_name,
        ])

    def get_absolute_url(self):

        return reverse('contacts-view', kwargs={'pk': self.id})
```

Because Django doesn't support migrations out of the box, we'll need to blow away the database and re-run syncdb.

XXX Perfect segue for talking about South

Now we need to limit the contact list to only the contacts the logged in User owns. This gets us into overriding methods that the base view classes have been handling for us.

For the list of Contacts, we'll want to override the `get_queryset` method, which returns the Django QuerySet of objects to be displayed.

```
class ListContactView(LoggedInMixin, ListView):

    model = Contact
    template_name = 'contact_list.html'

    def get_queryset(self):

        return Contact.objects.filter(owner=self.request.user)
```

The remaining views are responsible for showing only a single object – the Contact (or its addresses). For those we'll create another mixin that enforces authorization.

```
from django.core.exceptions import PermissionDenied
...
class ContactOwnerMixin(object):

    def get_object(self, queryset=None):
        """Returns the object the view is displaying.

        """

        if queryset is None:
            queryset = self.get_queryset()

        pk = self.kwargs.get(self.pk_url_kwarg, None)
        queryset = queryset.filter(
            pk=pk,
            owner=self.request.user,
        )

        try:
            obj = queryset.get()
        except ObjectDoesNotExist:
            raise PermissionDenied

        return obj
```

`ContactOwnerMixin` overrides the `get_object()` method, which is responsible for getting the object for a view to operate on. If it can't find one with the specified primary key and owner, it raises the `PermissionDenied` exception.

> **Note:** This implementation will return HTTP 403 (Forbidden) whenever it cannot find the a Contact with the requested ID and owner. This will mask legitimate 404 (Not Found) errors.

We'll use the `ContactOwnerMixin` in all of our views. For example, `ContactView` will look as follows:

```
class ContactView(LoggedInMixin, ContactOwnerMixin, DetailView):

    model = Contact
    template_name = 'contact.html'
```

Note that the order of inheritance is important: the superclasses (`LoggedInMixin`, `ContactOwnerMixin`, `DetailView`) will be checked in the order listed for methods. By placing `LoggedInMixin` first, you're guaranteed that by the time execution reaches `ContactOwnerMixin` and `DetailView`, you have a logged in, authenticated user.

Review

- XXX