# User authentication and permissions

In this tutorial we'll show you how to allow users to login to your site with their own accounts, and how to control what they can do and see based on whether or not they are logged in and their permissions. As part of this demonstration we'll extend the Library website, adding login and logout pages, and user- and staff-specific pages for viewing books that have been borrowed.

Django provides an authentication and authorisation ("permission") system, built on top of the session framework discussed in the previous tutorial, that allows you to verify user credentials and define what actions each user is allowed to perform. The framework includes built-in models for `Users` and `Groups` (a generic way of applying permissions to more than one user at a time), permissions/flags that designate whether a user may perform a task, forms and views for logging in users, and view tools for restricting content.

In this tutorial we'll show you how to enable user authentication in the Library website, create your own login and logout pages, add permissions to your models, and control access to pages. We'll use the authentication/permissions to display lists of books that have been borrowed for both users and librarians.

The authentication system is very flexible, and you can build up your URLs, forms, views, and templates from scratch if you like, just calling the provided API to login the user. However, in this article we're going to use Django's "stock" authentication views and forms for our login and logout pages. We'll still need to create some templates, but that's pretty easy.

We'll also show you how to create permissions, and check on login status and permissions in both views and templates.

## Enabling authentication

The configuration is set up in the `INSTALLED_APPS` and `MIDDLEWARE` sections of the project file (**library/library/settings.py**), as shown below:

```
INSTALLED_APPS = [
    ...
    'django.contrib.auth',  #Core authentication framework and its default models.
    'django.contrib.contenttypes',  #Django content type system (allows permissions to be associated with models).
    ....

MIDDLEWARE = [
    ...
    'django.contrib.sessions.middleware.SessionMiddleware',  #Manages sessions across requests
    ...
    'django.contrib.auth.middleware.AuthenticationMiddleware',  #Associates users with requests using sessions.
    ....
```

### Creating users and groups

You already created your first user when we looked at the Django admin site in tutorial 4 (this was a superuser, created with the command `python manage.py createsuperuser`). Our superuser is already authenticated and has all permissions, so we'll need to create a test user to represent a normal site user. We'll be using the admin site to create our *locallibrary* groups and website logins, as it is one of the quickest ways to do so.

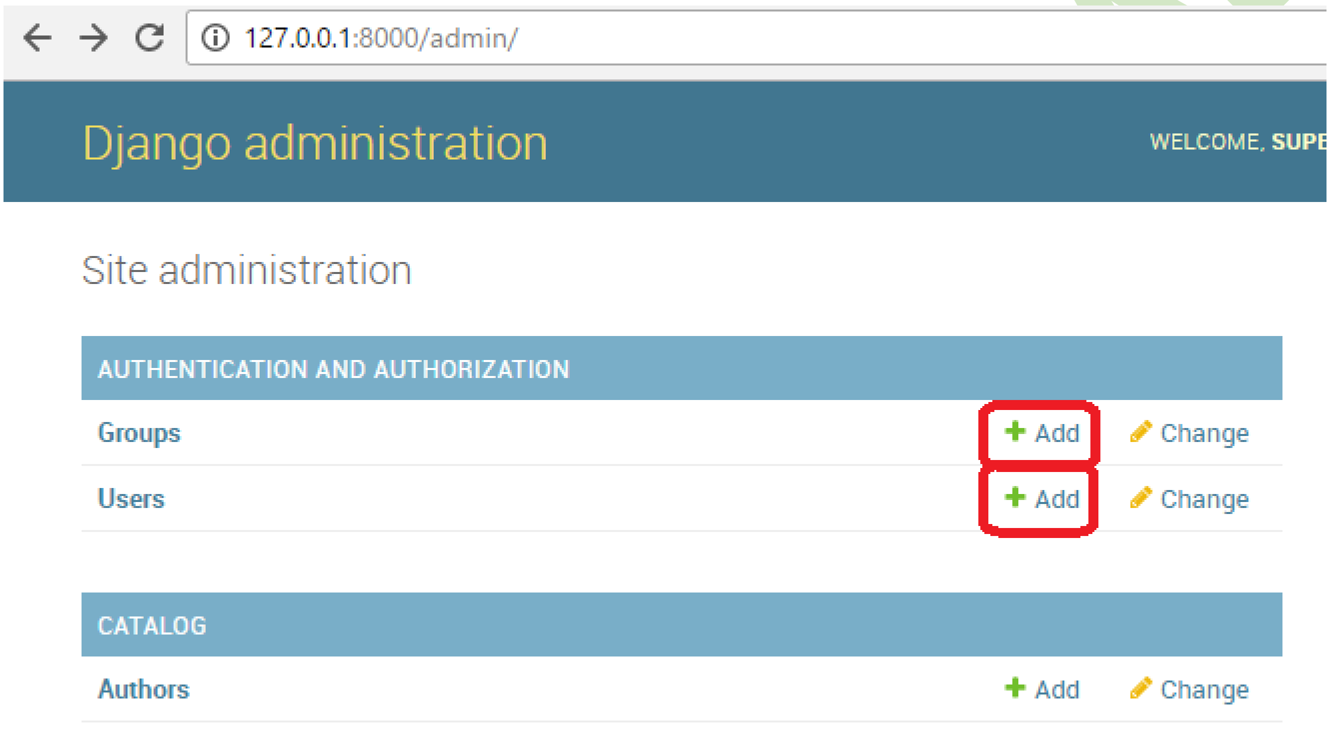
```
from django.contrib.auth.models import User

# Create user and save to the database
user = User.objects.create_user('myusername', 'myemail@crazymail.com', 'mypassword')

# Update fields and then save again
user.first_name = 'John'
user.last_name = 'Citizen'
user.save()
```

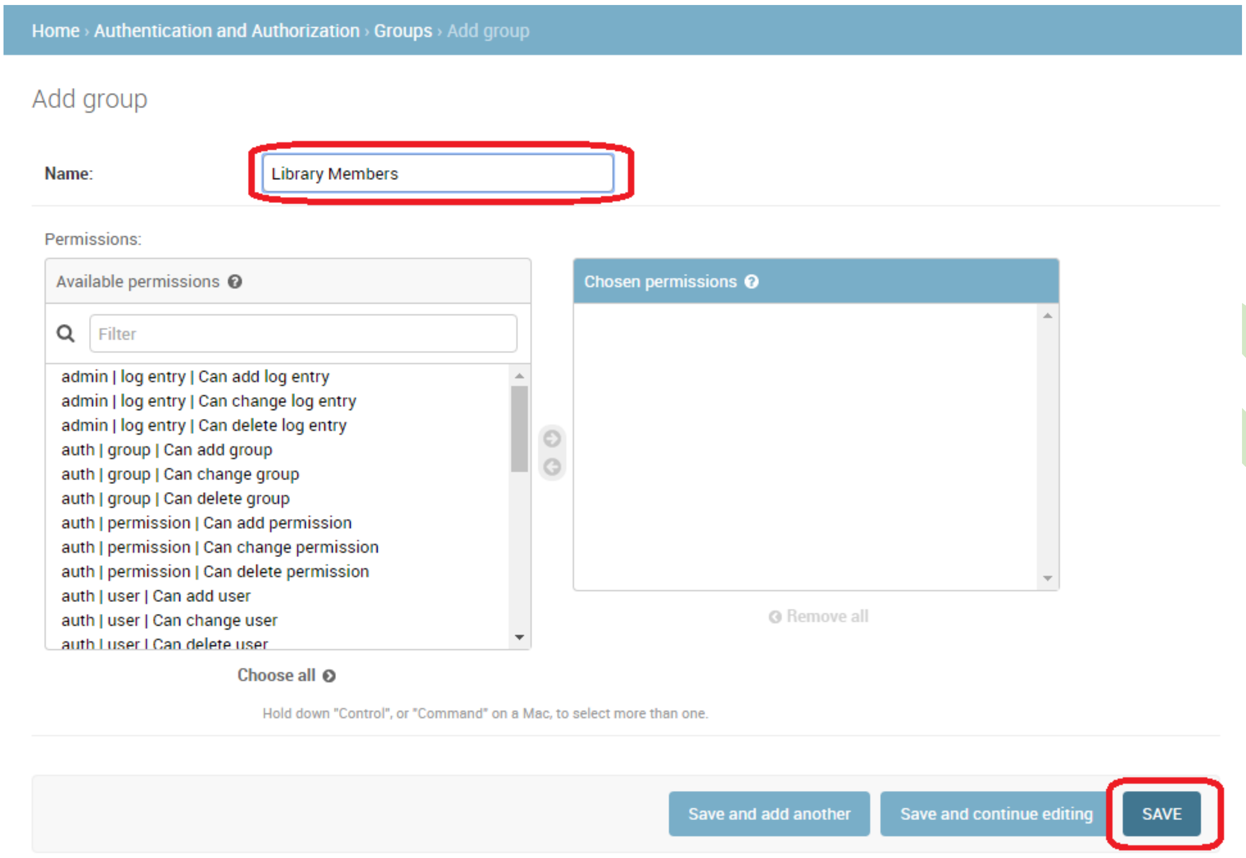
Below we'll first create a group and then a user. Even though we don't have any permissions to add for our library members yet, if we need to later, it will be much easier to add them once to the group than individually to each member.

Start the development server and navigate to the admin site in your local web browser (http://127.0.0.1:8000/admin/). Login to the site using the credentials for your superuser account. The top level of the Admin site displays all of your models, sorted by "django application". From the **Authentication and Authorisation** section you can click the **Users** or **Groups** links to see their existing records.



First lets create a new group for our library members.

1. Click the **Add** button (next to Group) to create a new *Group*; enter the **Name** "Library Members" for the group.


Home › Authentication and Authorization › Groups › Add group

Add group

Name: Library Members

Permissions:

Available permissions

Filter

admin | log entry | Can add log entry
admin | log entry | Can change log entry
admin | log entry | Can delete log entry
auth | group | Can add group
auth | group | Can change group
auth | group | Can delete group
auth | permission | Can add permission
auth | permission | Can change permission
auth | permission | Can delete permission
auth | user | Can add user
auth | user | Can change user
auth | user | Can delete user

Choose all

Chosen permissions

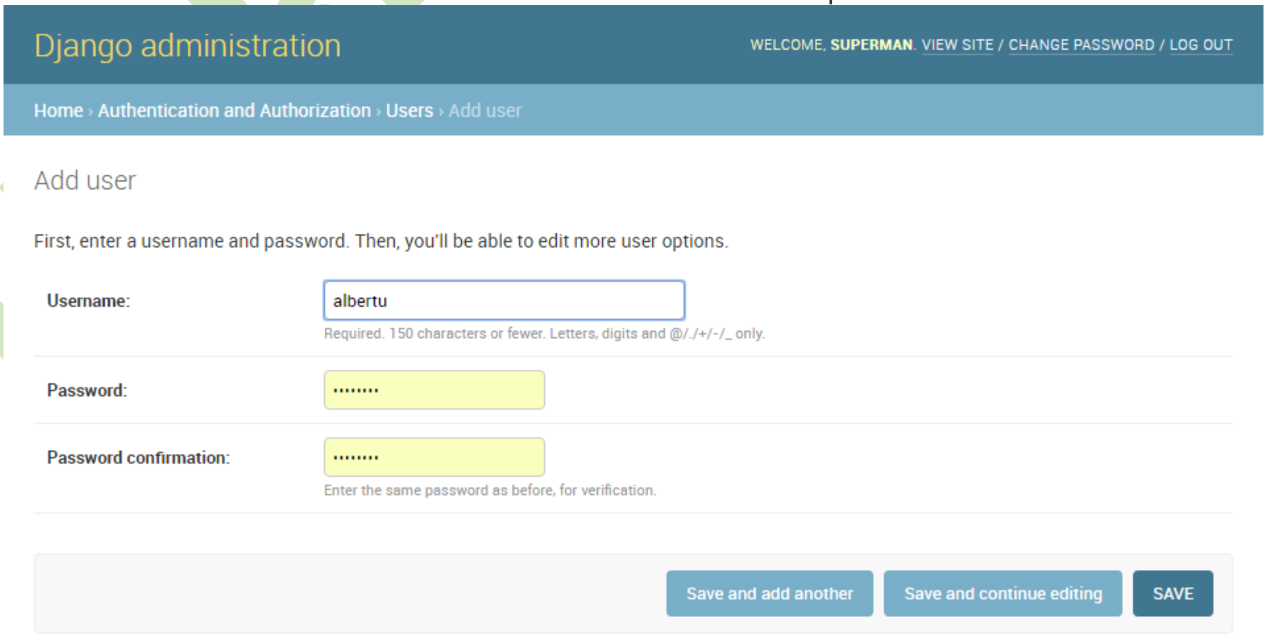
Remove all

Hold down "Control", or "Command" on a Mac, to select more than one.

2. We don't need any permissions for the group, so just press **SAVE** (you will be taken to a list of groups).

Now lets create a user:

1. Navigate back to the home page of the admin site
2. Click the **Add** button next to *Users* to open the *Add user* dialog.


Django administration

WELCOME, **SUPERMAN**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Authentication and Authorization › Users › Add user

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username: albertu

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

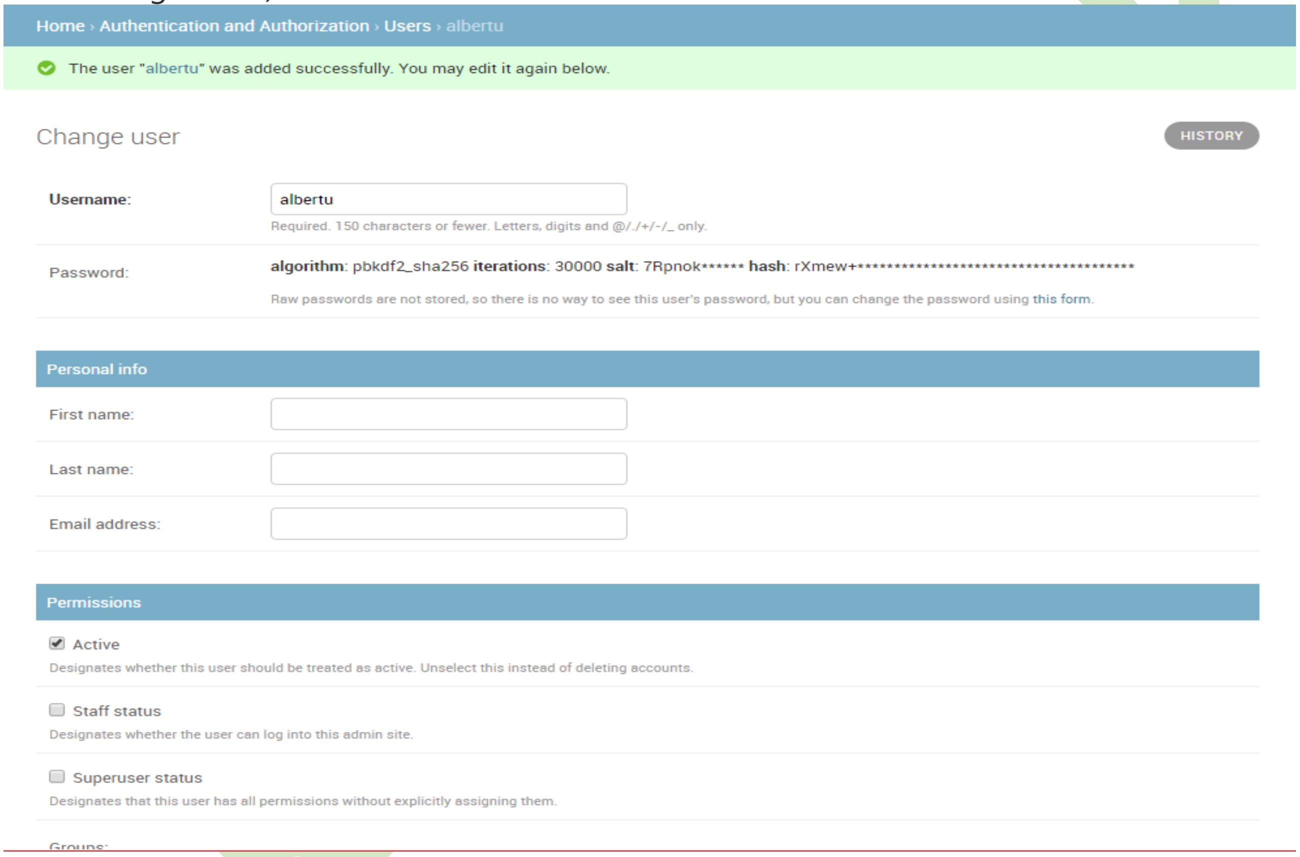
Password: ••••••••

Password confirmation: ••••••••

Enter the same password as before, for verification.

3. Enter an appropriate **Username** and **Password/Password confirmation** for your test user
4. Press **SAVE** to create the user.

The admin site will create the new user and immediately take you to a *Change user* screen where you can change your **username** and add information for the User model's optional fields. These fields include the first name, last name, email address, the users status and permissions (only the **Active** flag should be set). Further down you can specify the user's groups and permissions, and see important dates related to the user (e.g. their join date and last login date).


Home › Authentication and Authorization › Users › albertu

The user "albertu" was added successfully. You may edit it again below.

Change user

| | |
|---|---|
| Username: | albertu<br>Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only. |
| Password: | **algorithm**: pbkdf2_sha256 **iterations**: 30000 **salt**: 7Rpnok****** **hash**: rXmew+****************************************<br>Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form. |

Personal info

| | |
|---|---|
| First name: | |
| Last name: | |
| Email address: | |

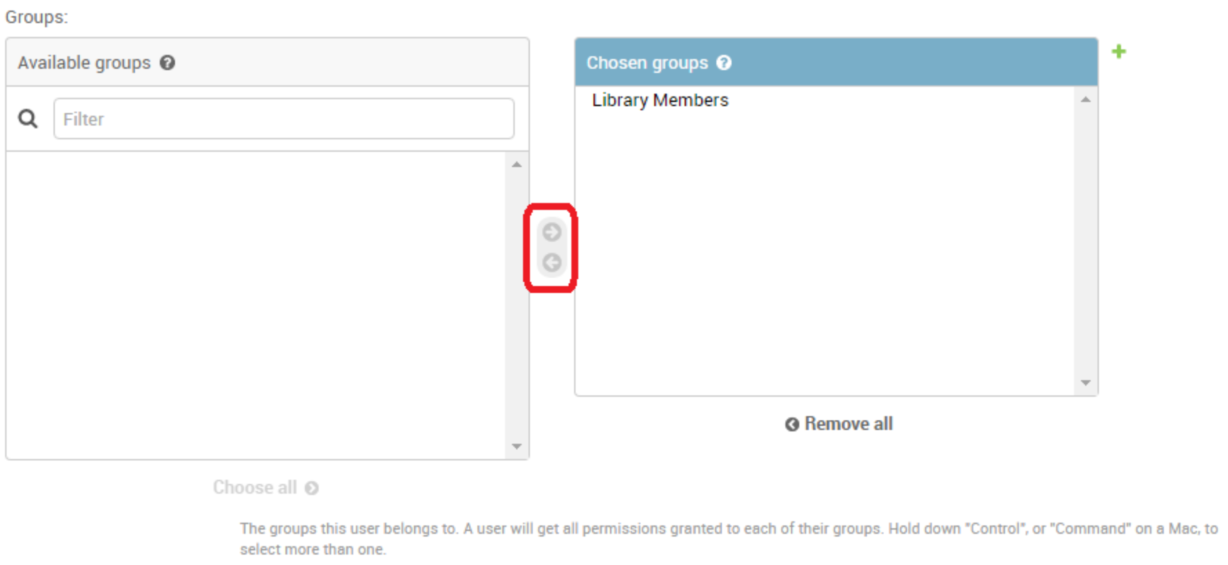
Permissions

[✓] Active
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

[ ] Staff status
Designates whether the user can log into this admin site.

[ ] Superuser status
Designates that this user has all permissions without explicitly assigning them.

5. In the *Groups* section, select **Library Member** group from the list of *Available groups*, and then press the **right-arrow** between the boxes to move it into the *Chosen groups* box.



The groups this user belongs to. A user will get all permissions granted to each of their groups. Hold down "Control", or "Command" on a Mac, to select more than one.

6. We don't need to do anything else here, so just select **SAVE** again, to go to the list of users.

That's it! Now you have a "normal library member" account that you will be able to use for testing (once we've implemented the pages to enable them to login).

## Setting up your authentication views

Django provides almost everything you need to create authentication pages to handle login, logout, and password management "out of the box". This includes a url mapper, views and forms, but it does not include the templates — we have to create our own!

In this section we show how to integrate the default system into the *Library* website and create the templates. We'll put them in the main project URLs.

## Project URLs

Add the following to the bottom of the project urls.py file **library/library/urls.py**) file:

```
#Add Django site authentication urls (for login, logout, password management)
urlpatterns += [
    path('accounts/', include('django.contrib.auth.urls')),
]
```

Navigate to the http://127.0.0.1:8000/accounts/ URL (note the trailing forward slash!) and Django will show an error that it could not find this URL, and listing all the URLs it tried. From this you can see the URLs that will work, for example:

## Template directory

The urls (and implicitly views) that we just added expect to find their associated templates in a directory **/registration/** somewhere in the templates search path.

For this site we'll put our HTML pages in the **templates/registration/** directory. This directory should be in your project root directory, i.e the same directory as as the **catalog** and **library** folders). Please create these folders now.

To make these directories visible to the template loader (i.e. to put this directory in the template search path) open the project settings (**/library/library/settings.py**), and update the `TEMPLATES` section's `'DIRS'` line as shown.

```
TEMPLATES = [
    {
        ...
        'DIRS': ['./templates',],
        'APP_DIRS': True,
        ...
```

## Login template

Create a new HTML file called /**library/templates/registration/login.html**. give it the following contents:

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

{% block content %}

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

{% if next %}
    {% if user.is_authenticated %}
    <p>Your account doesn't have access to this page. To proceed,
    please login with an account that has access.</p>
    {% else %}
    <p>Please login to see this page.</p>
    {% endif %}
{% endif %}

<form method="post" action="{% url 'login' %}">
{% csrf_token %}

<div>
  <td>{{ form.username.label_tag }}</td>
  <td>{{ form.username }}</td>
</div>
<div>
  <td>{{ form.password.label_tag }}</td>
  <td>{{ form.password }}</td>
</div>
```

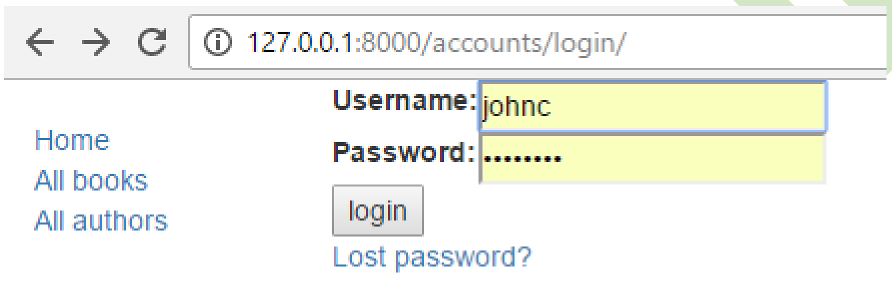
```
<div>
  <input type="submit" value="login" />
  <input type="hidden" name="next" value="{{ next }}" />
</div>
</form>

{# Assumes you setup the password_reset view in your URLconf #}
<p><a href="{% url 'password_reset' %}">Lost password?</a></p>

{% endblock %}
```

This template shares some similarities with the ones we've seen before — it extends our base template and overrides the `content` block. The rest of the code is fairly standard form handling code, which we will discuss in a later tutorial. All you need to know for now is that this will display a form in which you can enter your username and password, and that if you enter invalid values you will be prompted to enter correct values when the page refreshes.

Navigate back to the login page (http://127.0.0.1:8000/accounts/login/) once you've saved your template, and you should see something like this:



Open the project settings (**/library/library/settings.py**) and add the text below to the bottom. Now when you login you should be redirected to the site home page by default.

```
# Redirect to home URL after login (Default redirects to /accounts/profile/)
LOGIN_REDIRECT_URL = '/'
```

## Logout template

If you navigate to the logout url (http://127.0.0.1:8000/accounts/logout/) then you'll see some odd behaviour — your user will be logged out sure enough, but you'll be taken to the **Admin** logout page. That's not what you want, if only because the login link on that page takes you to the Admin login screen (and that is only available to users who have the `is_staff` permission).

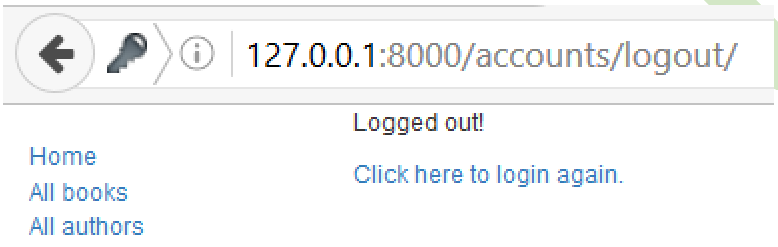
Create and open **/library/templates/registration/logged_out.html**. Copy in the text below:

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

{% block content %}
<p>Logged out!</p>

<a href="{% url 'login'%}">Click here to login again.</a>
{% endblock %}
```

This template is very simple. It just displays a message informing you that you have been logged out, and provides a link that you can press to go back to the login screen. If you go to the logout URL again you should see this page:


127.0.0.1:8000/accounts/logout/

Home
All books
All authors

Logged out!

Click here to login again.

## Testing the new authentication pages

Now that you've added the URL configuration and created all these templates, the authentication pages should now just work!

You can test the new authentication pages by attempting to login and then logout your superuser account using these URLs:

- http://127.0.0.1:8000/accounts/login/
- http://127.0.0.1:8000/accounts/logout/

## Testing against authenticated users

This section looks at what we can do to selectively control content the user sees based on whether they are logged in or not.

## Testing in templates

You can get information about the currently logged in user in templates with the `{{ user }}` template variable (this is added to the template context by default when you set up the project as we did in our skeleton).

Typically you will first test against the `{{ user.is_authenticated }}` template variable to determine whether the user is eligible to see specific content. To demonstrate this, next we'll update our sidebar to display a "Login" link if the user is logged out, and a "Logout" link if they are logged in.

Open the base template (**/library/catalog/templates/base_generic.html**) and copy the following text into the `sidebar` block, immediately before the `endblock` template tag.

```
<ul class="sidebar-nav">

    ...

   {% if user.is_authenticated %}
     <li>User: {{ user.get_username }}</li>
     <li><a href="{% url 'logout'%}?next={{request.path}}">Logout</a></li>
   {% else %}
     <li><a href="{% url 'login'%}?next={{request.path}}">Login</a></li>
   {% endif %}
  </ul>
```

As you can see, we use `if-else-endif` template tags to conditionally display text based on whether `{{ user.is_authenticated }}` is true. If the user is authenticated then we know that we have a valid user, so we call **{{ user.get_username }}** to display their name.

We create the login and logout link URLs using the url template tag and the names of the respective URL configurations. Note also how we have appended ?next={{request.path}} to the end of the URLs. What this does is add a URL parameter next containing the address (URL) of the *current* page, to the end of the linked URL. After the user has successfully logged in/out, the views will use this "next" value to redirect the user back to the page where they first clicked the login/logout link.

## Testing in views

If you're using function-based views, the easiest way to restrict access to your functions is to apply the login_required decorator to your view function, as shown below. If the user is logged in then your view code will execute as normal. If the user is not logged in, this will redirect to the login URL defined in the project settings (settings.LOGIN_URL), passing the current absolute path as the next URL parameter. If the user succeeds in logging in then they will be returned back to this page, but this time authenticated.

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

@login_required
def my_view(request):
    ...
```

Similarly, the easiest way to restrict access to logged-in users in your class-based views is to derive from LoginRequiredMixin. You need to declare this mixin first in the super class list, before the main view class.

```
from django.contrib.auth.mixins import LoginRequiredMixin

class MyView(LoginRequiredMixin, View):
    ...
```

This has exactly the same redirect behaviour as the login_required decorator. You can also specify an alternative location to redirect the user to if they are not authenticated (login_url), and a URL parameter name instead of "next" to insert the current absolute path (redirect_field_name).

```
class MyView(LoginRequiredMixin, View):
    login_url = '/login/'
    redirect_field_name = 'redirect_to'
```

## Example — listing the current user's books

Now that we know how to restrict a page to a particular user, lets create a view of the books that the current user has borrowed.

Unfortunately we don't yet have any way for users to borrow books! So before we can create the book list we'll first extend the BookInstance model to support the concept of borrowing and use the Django Admin application to loan a number of books to our test user.

## Models

First we're going to have to make it possible for users to have a `BookInstance` on loan (we already have a `status` and a `due_back` date, but we don't yet have any association between this model and a User. We'll create one using a `ForeignKey` (one-to-many) field. We also need an easy mechanism to test whether a loaned book is overdue.

Open **catalog/models.py**, and import the `User` model from `django.contrib.auth.models` (add this just below the previous import line at the top of the file, so `User` is available to subsequent code that makes use of it):

```
from django.contrib.auth.models import User
```

Next add the `borrower` field to the `BookInstance` model:

```
borrower = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True)
```

While we're here, lets add a property that we can call from our templates to tell if a particular book instance is overdue. While we could calculate this in the template itself, using a property as shown below will be much more efficient.

```
from datetime import date

@property
def is_overdue(self):
    if self.due_back and date.today() > self.due_back:
        return True
    return False
```

Now that we've updated our models, we'll need to make fresh migrations on the project and then apply those migrations:

```
python3 manage.py makemigrations
python3 manage.py migrate
```

## Admin

Now open **catalog/admin.py**, and add the `borrower` field to the `BookInstanceAdmin` class in both the `list_display` and the `fieldsets` as shown below. This will make the field visible in the Admin section, so that we can assign a `User` to a `BookInstance` when needed.

```
@admin.register(BookInstance)
class BookInstanceAdmin(admin.ModelAdmin):
    list_display = ('book', 'status', 'borrower', 'due_back', 'id')
    list_filter = ('status', 'due_back')

    fieldsets = (
        (None, {
            'fields': ('book','imprint', 'id')
        }),
        ('Availability', {
            'fields': ('status', 'due_back','borrower')
        }),
    )
```

## Loan a few books

Now that its possible to loan books to a specific user, go and loan out a number of `BookInstance` records. Set their `borrowed` field to your test user, make the `status` "On loan" and set due dates both in the future and the past.

## On loan view

Now we'll add a view for getting the list of all books that have been loaned to the current user. We'll use the same generic class-based list view we're familiar with, but this time we'll also import and derive from `LoginRequiredMixin`, so that only a logged in user can call this view. We will also choose to declare a `template_name`, rather than using the default, because we may end up having a few different lists of BookInstance records, with different views and templates.

Add the following to catalog/views.py:

```
from django.contrib.auth.mixins import LoginRequiredMixin

class LoanedBooksByUserListView(LoginRequiredMixin,generic.ListView):
    """
    Generic class-based view listing books on loan to current user.
    """
    model = BookInstance
    template_name ='catalog/bookinstance_list_borrowed_user.html'
    paginate_by = 10

    def get_queryset(self):
        return
BookInstance.objects.filter(borrower=self.request.user).filter(status__exact='o').order
_by('due_back')
```

In order to restrict our query to just the `BookInstance` objects for the current user, we re-implement `get_queryset()` as shown above. Note that "o" is the stored code for "on loan" and we order by the `due_back` date so that the oldest items are displayed first.

## URL conf for on loan books

Now open **/catalog/urls.py** and add a`path()` pointing to the above view (you can just copy the text below to the end of the file).

```
urlpatterns += [
    path('mybooks/', views.LoanedBooksByUserListView.as_view(), name='my-borrowed'),
]
```

## Template for on loan books

Now all we need to do for this page is add a template. First, create the template file **/catalog/templates/catalog/bookinstance_list_borrowed_user.html** and give it the following contents:

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

{% block content %}
    <h1>Borrowed books</h1>

    {% if bookinstance_list %}
    <ul>

      {% for bookinst in bookinstance_list %}
      <li class="{% if bookinst.is_overdue %}text-danger{% endif %}">
        <a href="{% url 'book-detail' bookinst.book.pk %}">{{bookinst.book.title}}</a>
({{ bookinst.due_back }})
      </li>
      {% endfor %}
    </ul>

    {% else %}
      <p>There are no books borrowed.</p>
    {% endif %}
{% endblock %}
```

This template is very similar to those we've created previously for the `Book` and `Author` objects. The only thing "new" here is that we check the method we added in the model (`bookinst.is_overdue`) and use it to change the colour of overdue items.

When the development server is running, you should now be able to view the list for a logged in user in your browser at http://127.0.0.1:8000/catalog/mybooks/. Try this out with your user logged in and logged out (in the second case, you should be redirected to the login page).

## Add the list to the sidebar

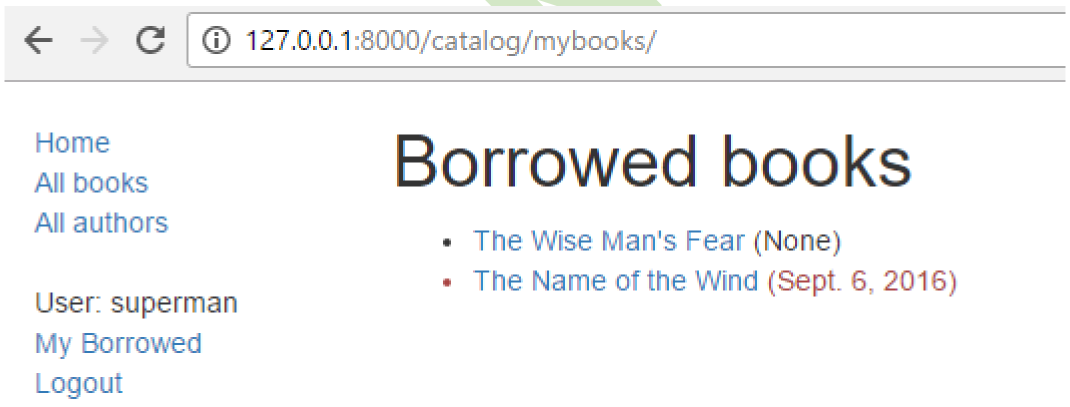
The very last step is to add a link for this new page into the sidebar. We'll put this in the same section where we display other information for the logged in user.

Open the base template (**/library/catalog/templates/base_generic.html**) and add the line in bold to the sidebar as shown.

```
<ul class="sidebar-nav">
   {% if user.is_authenticated %}
   <li>User: {{ user.get_username }}</li>
   <li><a href="{% url 'my-borrowed' %}">My Borrowed</a></li>
   <li><a href="{% url 'logout'%}?next={{request.path}}">Logout</a></li>
   {% else %}
   <li><a href="{% url 'login'%}?next={{request.path}}">Login</a></li>
   {% endif %}
 </ul>
```

## What does it look like?

When any user is logged in, they'll see the *My Borrowed* link in the sidebar, and the list of books displayed as below (the first book has no due date, which is a bug we hope to fix in a later tutorial!).


127.0.0.1:8000/catalog/mybooks/

Home
All books
All authors

User: superman
My Borrowed
Logout

Borrowed books

- The Wise Man's Fear (None)
- The Name of the Wind (Sept. 6, 2016)

## Permissions

Permissions are associated with models, and define the operations that can be performed on a model instance by a user who has the permission. By default, Django automatically gives *add*, *change*, and *delete* permissions to all models, which allow users with the permissions to perform the associated actions via the admin site. You can define your own permissions to models and grant them to specific users. You can also change the permissions associated with different instances of the same model.

Testing on permissions in views and templates is then very similar for testing on the authentication status (and in fact, testing for a permission also tests for authentication).

## Models

Defining permissions is done on the model "`class Meta`" section, using the `permissions` field. You can specify as many permissions as you need in a tuple, each permission itself being defined in a nested tuple containing the permission name and permission display value. For example, we might define a permission to allow a user to mark that a book has been returned as shown:

```
class BookInstance(models.Model):
    ...
    class Meta:
        ...
        permissions = (("can_mark_returned", "Set book as returned"),)
```

We could then assign the permission to a "Librarian" group in the Admin site.

Open the **catalog/models.py**, and add the permission as shown above. You will need to re-run your migrations (call `python3 manage.py makemigrations` and `python3 manage.py migrate`) to update the database appropriately.