More Django

User Authentications & View Classes

User Authentication in Django

Django has a built-in system for Authentication and Authorization.

- Authentication: verifies who the user is
- Authorization: specifies what the use can do

It's included by default in '**django.contrib.auth**' and '**django.contrib.contenttypes**'

Built-in Auth

Django combines its Authentication and Authorization into its auth system, which supports:

- Users
- Permissions (binary yes/no flags)
- Groups
- Password hashing system
- forms and view tools for logging in users and restricting access
- pluggable backend

Read Up Yourselves

Just in case this is easier, you can read all the Auth documentation at https://docs.djangoproject.com/en/1.8/topics/auth/

User Objects

User objects define the who of who is able to interact with your system. Users have, by default:

- username
- password
- email
- first_name
- last_name

Creating Users

To create a new user, simply import django.contrib.auth.models and call User's create_user() function.

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

user1 = User.objects.create_user('bob', 'bob@email.com', 'bobpass')

myuser = YourUser(user=user1, address='323 A St')
```

Superusers: A note

Superusers are a type of User that are allowed to do direct database manipulations and log directly into the admin interface.

You can only make them using manage.py.

Authenticating Users

To authenticate a user, simply use the built-in authenticate() method:

```
from django.contrib.auth import authenticate

# A simple login function
def auth(username, secret_pass):
    user = authenticate(username=username, password=secret_pass)
    if user is not None:
        if user.is_active:
            # Valid user
        else:
            # auth worked, but account is disabled
    else:
        # the authentication failed due to wrong something
```

Sessions

Sessions are per-site-visit concept - every time you visit a site, you create a session.

Session information is stored in the **request** object your views receive and pass around.

You can have **anonymous sessions** (the default on Django), or you can **login** a user to associate a specific **user** with a session.

Logging In

Once a user is authenticated, you can log them into the system to associate them with the current session:

```
from django.contrib.auth import authenticate, login

# A simple login function
def simpleLogin(username, secret_pass, request):
    user = authenticate(username=username, password=secret_pass)
    if user is not None:
        if user.is_active:
            login(request, user)
        else:
            # auth worked, but account is disabled
    else:
        # the authentication failed due to wrong something
```

Storing with Sessions

You can store things in the session object using the built-in methods and properties. The **request.session** object is a dictionary.

```
# Adding something to the session:

request.session[‘fav_color’] = ‘red’

# Then, on another view:

print(“Favorite color is:”, request.session.get(‘fav_color’)
```

Limiting Access to Users

The easiest way to ensure only logged in users see your content, is to check if the user is authenticated:

request.user.is_authenticated()

You can also use the login_required decorator:

```
@login_required
def some_view(request):
    pass
```

Login Required Requirements

To use the login_required decorator, you must have your **settings.LOGIN_URL** set to point to where you want the user redirected if they're not logged in (typically a login page).

View Classes

Views

In Django, a view, by default, is a function that takes a **request** and returns a **response.**

A view could also be a **class** that still takes a request and returns a response, but may also do more complex logic in the middle.

View Classes

There are several built-in View classes you can leverage, both for inheritance and mixins.

All View classes must inherit from the default View class

Some simple Views:

- TemplateView
- RedirectView
- ListView

View Classes as_view()

If you use a View class, you **must** set the urlconf to use the class as a view:

```
urlpatterns = [
    url(r'^about/',AboutView.as_view()),
]
```

as_view() function provides function-like entry into the view for Django.

TemplateView

Sometimes you just need Django to throw the template page up, with little to no additional logic. **TemplateView** does just this:

```
from django.views.generic import TemplateView

class AboutView(TemplateView):

    template_name = “about.html”
```

Even Easier Generic Views

When using a Generic View and not modifying it, you can even just skip making a custom view altogether, like so:

```
urlpatterns = [
    url(r'^about/',
        TemplateView.as_view(template_name="about.html")),
]
```

Why Other Class-based Views?

- Leverage built-in views in Django that make hooking templates to models easier
- write custom POST or GET handlers (for APIs or after login/logout actions)
- handle different user permissions for a page
- modularize your controller logic

ListView Example

ListViews hook to a Model and just passes all the data for the model to the template:

```
class PublisherList(ListView):
    model = Publisher
```

```
{% block content %}
    <h2>Publishers</h2>
        {% for publisher in object_list %}
            <p>{{ publisher.name }}</p>
        {% endfor %}
{% endblock %}
```

More Data

If you want to pull more data for a ListView (or other generic view), simply implement custom a **get_context_data** method that pulls more info:

```
class PublisherDetail(DetailView):
    model = Publisher
    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super(PublisherDetail, self).get_context_data(**kwargs)
        # Add in a QuerySet of all the books
        context['book_list'] = Book.objects.all()
        return context
```

Custom Queries

Instead of just grabbing ALL the items of that Model type, you can set a ListView to use a specific subset, using **queryset**

```
class BookList(ListView):
    queryset = Book.objects.order_by('-publication_date')
    context_object_name = 'book_list'
```

Dynamic Custom Queries

To make your query sets for views dynamic, you can take advantage of the **get_queryset** function, and also leverage arguments passed via the urlconf:

```
class PublisherBookList(ListView):


    template_name = 'books/books_by_publisher.html'


    def get_queryset(self):
        self.publisher = get_object_or_404(Publisher, name=self.args[0])
        return Book.objects.filter(publisher=self.publisher)
```

Lots and Lots of Generic Views

Django has a variety of Generic Views which you can leverage to create your view classes. It's up to you to investigate and figure out what works best for your needs.

Read more at:

https://docs.djangoproject.com/en/1.8/topics/class-based-views