# Django Workshop - Pt 2

Tyler Schwehr

RedRoom CTF

## What we’re going to cover

- Serving static files
- User authentication
- Basic signals

## What are static files

- As you’ve seen in HTML, we often need to reference files that exist on the web server, such as image files, CSS, and JavaScript
- These are static files, they don’t change, they’re just static
- Directly referencing these files introduces a security flaw known as directory traversal
- Django comes with a basic middleware function that allows us to serve static files for testing, but this should not be deployed in a production environment

# Managing static files in Django

- Django has two distinct mechanisms that help serve static files
- One is for serving static files intended for the underlying HTML document
- The other is for serving static files that relate to the data models
- While both of these mechanisms have a lot in common, if you mix up the components it will not serve the files
- Trust me, ~~I've been suffering through this for the last few days~~ *I'm an expert!*

## Modifying our directory

- When serving files intended for the HTML documents such as this, Django will help organise the files
- The first thing we need to do is create a folder named ‘static’, we do this in the relevant app
- Within that folder we create another named after the app
- We can store our files within this folder
- In the settings.py file, locate the static URL, it should look like this

```
STATIC_URL = ‘static/’
```

- Place this line underneath the static URL

```
STATIC_ROOT = BASE_DIR / ‘static/’
```

## Modifying our URL paths

- Now that we've setup our folder structure, we need to configure our URL's to interpret it
- Before we add our code, we nee to import a couple of files

```
from django.conf import settings

from django.conf.urls.static import static
```

- In your website urls.py file add the following line to the end of the URL list

```
static(settings.STATIC_URL,
document_root = settings.STATIC_ROOT)
```

- This is a special type of path that we can reference in our templates

## Inserting files into templates

- Now that we’ve done the heavy lifting, we can add the files into our templates
- To do so we start the document with the {% load static %} statement
- To invoke a specific static file we use this statement:

{% static ‘appName/filename’ %}

- The invocation statement should be treated as a URL and placed accordingly
- If you have files across several apps, Django can still reference any of them, simply use the relevant app name

## Serving static files from models

- We can also serve static files that are associated with a given database entry
- This is really helpful for managing things like product images, and having the ability to upload images
- The website doesn’t store images in the database, because that’s a bad idea, so it stores it in your file directory

## Updating our model

- Firstly, we need to update our model to have a field for an image
- We assign the models.ImageField() to a field
- There are two parameters that we must have
- The first is the ‘upload_to’ parameter, this should receive a path name relevant to the model
- The second is the ‘default’ parameter
- The default parameter should include the file name of the default image
- This file name should also include the path noted in the ‘upload_to’ parameter
- Once you migrate your changes to the database, django will alert you that you need to install a module called ‘Pillow’

# Configuring Django settings

- Now that are model is accepting images, we need to perform very similar steps to before
- We need to make sure that we create two folders
- The first folder should be in the website root directory and named ‘media’
- The second is within the media directory and should be the name that you used in the model
- Now put the default image file within that folder
- In the settings.py file we need to set up the URL and root directory
- We can do so with the following lines

```
MEDIA_URL = ‘media/’

MEDIA_ROOT = BASE_DIR / ‘media/’
```

## Adding the static path

- At the end of the first static path, we want to append another static path
- This time we’re going to reference the media properties, this is the statement

```
+ static(settings.MEDIA_URL, document_root = settings.MEDIA_ROOT
```

## What is pagination

- Simply put pagination is a mechanism we use to break up big pages into smaller pages
- You would have encountered before but perhaps not known the name for it
- A typical place that you'll see it is in product lists where they are broken up into smaller lists
- We use page numbers to progress through the pages
- Django has a module that makes pagination easy, it stores pagination information within the request

## Updating our view

- We want to update our product list view so that only a given number of products are displayed on each page
- Firstly we import the Paginator module from django.core.paginator
- We need to create an instance of our paginator with two parameters, the first is our object list and the second is how many objects to display per page

```
paginator = Paginator(product_list, 3)
```

- Next we'll need to extract the page number from the request, we can do that using this code:

```
page = request.GET.get('page')
```

- Using that we tell the paginator to give us the objects that should relate to this page

```
page_data = paginator.get_page(page)
```

- And finally we pass that as context

## Updating our list template

- There are a handful of variables and functions that we can call in our template
- I have updated the command sheet to include the various template elements
- In HTML if we use a href with starts with the question mark, we can call the same page with new parameters

‘?page=1’

- Knowing that, we can build buttons and logic to handle going back a page or going forward
- We can make a button for the first page by simply referencing page 1
- And we can reference the last page by checking how many pages there are in total

## User authentication

- Django already has a user management system implemented
- We're going to utilise it to implement users in our website
- We can extend their user management system to include our requirements
- There are many different approaches but I'm using this one because it's straight forward
- I'm not going to explain what logging in is, because I would hope you all know that one already

## Overview of user authentication

- There are several things that we need to achieve for our user authentication system
- A model for our extended user data
- We need a form to register users
- Signals to listen for user creation
- A form to log in, and a button to log out
- We need a view for each of these
- And a template for logging in
- And a mechanism to detect if a user is logged in

## Setting up our model

- As per our typical model creation process, we create a class in the models.py file, it should inherit from the models.Model class
- Name it as you like, but I try to use the word ‘user’ in the name, so that I know it relates to the User model
- Include attributes as you like, but don’t include firstname, lastname, username, email, or password
- Let’s include a field name ‘role’ which defaults to ‘user’
- Next we want to establish a relationship with the User model, we will need to import the User class
- So we will include a field called ‘user’ and assign the following

models.OneToOneField(User, on_delete = models.CASCADE

## Creating our registration form

- The first thing we need to do is import the ‘User’ model and the ‘UserCreationForm’
- We’ll create our class and name it ‘RegistrationForm’ and inherit from the ‘UserCreationForm’
- The user creation form that we inherit from includes username and password fields
- We want to add any additional fields such as email

- Within our Meta class we want to use the ‘User’ model
- We need to include the following fields as a requirement
  - ‘username’
  - ‘password1’
  - ‘password2’
- Feel free to add additional fields for the User

## Implementing the registration form

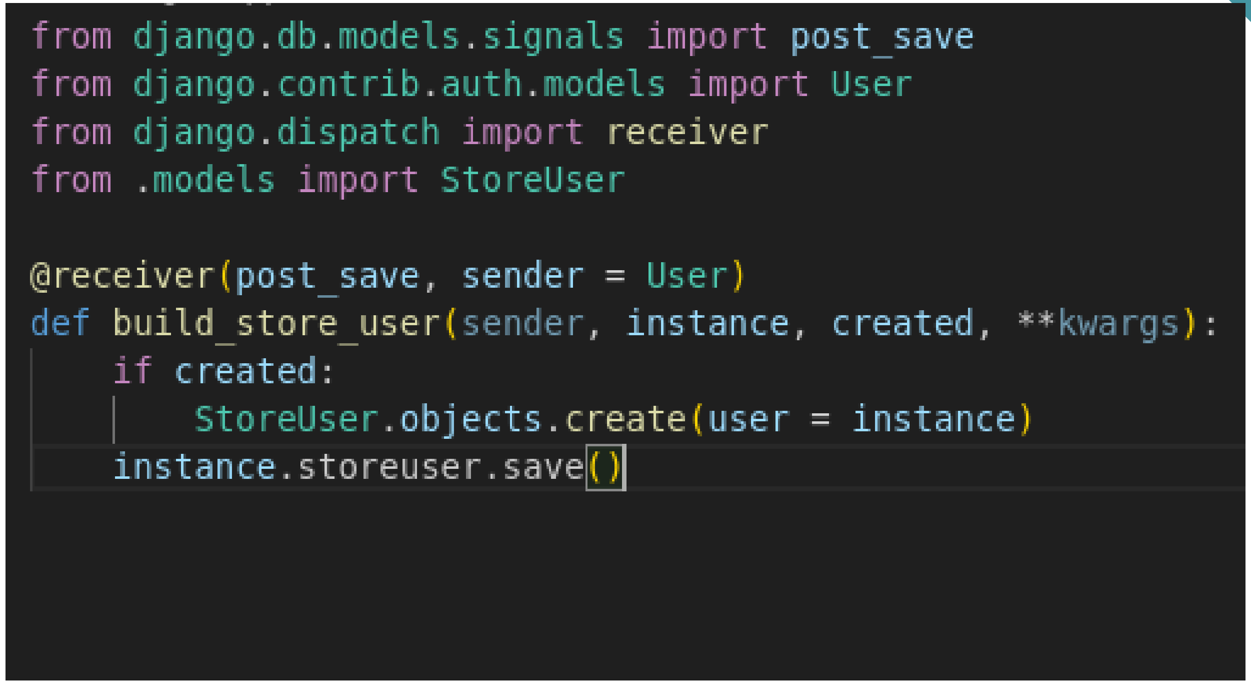
- Make the template as you would for a form, looping over each field
- Create the view in a similar fashion to how we added new items in the last workshop
- We check that the request.method is POST, confirm the form data is valid and save the form
- From there we can do our redirect
- And of course set up the URL pattern

## What are signals?

- I’m not familiar enough with Python to know how they handle events normally
- Events are very common in Object Oriented Programming; it allows us to achieve loose coupling between objects
- The way Django has implemented this is creating a system they call Signals
- The core idea is straight-forward, when an event occurs it sends a signal, anything listening for that signal receives it
- We’re going to listen for when a new user is registered and create our extended model when that happens
- Overusing signals can create unnecessary complexity within our code though

## Hooking up our signals

- Firstly we need to import the post_save event, the User model, the receiver, and our extended model
- We define our function with the shown parameters
- In our use here we check if the form is new and if it is we create a new StoreUser object and save it
- The receiver decorator states the event type to listen to and where from



```python
from django.db.models.signals import post_save
from django.contrib.auth.models import User
from django.dispatch import receiver
from .models import StoreUser

@receiver(post_save, sender = User)
def build_store_user(sender, instance, created, **kwargs):
    if created:
        StoreUser.objects.create(user = instance)
    instance.storeuser.save()
```

## Handling the Login

- Make the template as you would for a form, looping over each field
- Instead of making our own view, we’re going to import a view from django.contrib.auth and give it an alias
- This is a class based view, which is similar to the function based views that you’ve been using but has additional functionality
- When using a class based view, we must use the method .as_view()
- In settings.py we need to set up our LOGIN_REDIRECT_URL & LOGIN_URL

## Logging out

- Logging out is a little different than logging in, firstly set up a URL path
- In our views we need to import the logout module from django.contrib.auth
- Then we simply call the logout function and pass the request as a parameter
- Then we'll want to return a redirect to send the user somewhere else
- Django does has a message system that you can use to give the user a message about logging out, but I'm not covering that today

## Authentication buttons

- Now that we can log users in and out, let’s make some navbar buttons
- To access authentication information about the user, we can reference the user with ‘request.user’ for views and ‘{{ user }}’ in our templates
- There is a handy method of .is_authenticated that will allow us to check if the user is logged in
- In our navbar we can use the if function to check if the user is logged in or not and give them either a log in button or log out button

## Role constraints in views

- With our quirky process we'll be doing our authentication in two steps
- Firstly we'll check if the user if logged in at all, and this time we'll use a decorator
- Import the login_required decorator from django.contrib.auth.decorators
- Simply apply the decorator the any view that should require it
- Within the view get the users account and check to see if their role is 'admin' or whatever else you named it
- If they have that role perform the view as normal, otherwise redirect them to an access denied page

## Thank you

Tyler Schwehr

RedRoom CTF