Django Course Notes

Link : https://www.youtube.com/watch?v=UmljXZIypDc&list=PL-osiE80TeTtoQCKZ03TU5fNfx2UY6U4p&index=1

**Lecture 1**

**L1 aims:**

* **Create basic Django project**
* **Understand basic project structure components.**
* **Learn how to start/stop server (locally hosted)**

“Django-admin” to list admin commands (console).

Use “Django-admin startproject [name]” to create new project. Creates a new Django project that has a complete structure with different files (everything needed to get started).

Structure of Django project:

* Manage.py file allows us to run command line commands
* “projectName” directory containing 4 files
  + Init.py
  + Settings.py – where you change settings/configurations. (See docs more more info)
  + url.py – set up mappings for specified routes.
  + Wsgi.py – how python web app and Django webserver communicate

To run server “python manage.py runserver” command.

To stop server use CNTRL+C (kill process)

To access site go to specified url or localhost:portNumber

**Lecture 2**

**L2 aims:**

* **Adding blog application to our Django site**
* **Setting up basic URL routes..**

Django website project can have multiple applications (eg blog section, store section, etc)

* Good for separating out parts of out website (eg add (drop) blog section into multiple other websites)

Creating blog app >> “python manage.py startapp [appNameHere]”

Many files created at startup >>> get more out of the box then flask >> tradeoff.

Create blog >> views.py >> home method to handle traffic from homepage of blog.

* Takes in request arg
* Returns what we want user to see when they hit this route.

To map URL pattern to home route >> create url.py file >> in url.py map urls to each view function (copy from Django\_project/url.py file)

Import views.py in urls.py file to use ‘home’ function >> “from . import views” (. equals current directory)

To create homepage path use empty path and home view

* “path(‘’, views.home, name=’blog-home’)”.

Use informative names for name keyword arg to ensure route names do not clash across apps on Django site (eg ‘blog-home’ instead of ‘home’).

Django\_project/urls.py tells which routes should be directed to our blog app

* Import path, include functions from Django.urls
* Use include function to reference blog app urls “include(‘blog.urls’)

Now when we open Django site in browser and go to /blog it will map to blog.urls which contains an empty path for homepage. Empty path to indicate no additional url required.

Previously in Django regular expressions used to match paths >> this is not required anymore in later versions of Django as ‘path()’ is now used instead of ‘url()’ .

* <https://stackoverflow.com/questions/47947673/is-it-better-to-use-path-or-url-in-urls-py-for-django-2-0> for example + details.

After running Django site server, Terminal displays debug output (routes attempted to access + http status code response)

When Django encouters include() function it chops off matched part of URL and only sends remaining unmatched part to included urls module for further processing.

* Eg In ‘http://localhost:8000/blog/’ it will send an empty string ‘’ to blogs urls.py to be matched there. In urls.py empty string route is handled by ‘home()’ function which returns http response.

It is a good thing that the url gets passed around like this as if we wanted to change the route to our blog app we can simply change that in one place and it applies to all of those routes.

* Eg to change from /blog/About and to /blog\_dev/About only 1 change is required in Django\_projects/url.py path function.

To map default app to blog app in Django website use empty string in path for blog app

* Path(‘’, include(blog.urls))

**Lecture 3**

**L3 aims:**

* **Learn how to use templates to return more complex HTML code.**
* **See how to pass variables to our templates.**
* **Templates best practices.**

In order to avoid returning all html code in HTTPResponse() method, we can use templates.

First create template directory within blog app directory. By default Django looks for a templates subdirectory in each of our installed apps.

Django convention is to have a folder within an apps template subdirectory that matches the app’s name. Eg blog\_app >> templates >> blog

After creating our home.html and about.html templates, we need to add our blog app to our list of installed apps so that Django knows to look there for a templates directory.

* To do this we add our app configuration (path) to the list of installed apps in the Django projects settings.py module.
* This process is required to get Django to correctly search your templates and installed\_apps is also where Django looks for an app’s models.

To render home.html template when a user navigates to blog homepage point our blog views to use the home template.

* There are a few different ways to load in a template.
  + One way would be to load in a template and then render it and pass that to a HTTPResponse() method.
  + Django provides a shortcut to do all this (render() method imported from Django.shortcuts)

The render(request, template.html, context) function takes the request object as its first argument, template to render as second argument, context as third argument (used to pass info into template).

* To reference template use convention of appname/template.html

The render() function still returns a HTTPResponse in the background. Views always need to return a HTTPResponse or an exception.

Demonstrating context - Fake posts task:

* Create fake post data in views.py
* We can pass posts data into a template by storing it in a dictionary and passing that into the render() method.

Django template engine very similar to jinja2 used in Flask. (see code for examples)

* {% for post in posts %} single braces used for operations eg loops, if-else, etc
* {{ posts.title }} double braces used when accessing variables.

Avoid repeated code across pages, better to have everything that is repeated in a single place so that when we want to make a change we only need to do it in one place.

We can do this using Template inheritance.

* Create base template containing shared html code.
* In child template inherit base template {% extends “blog/base.html” %}
* Create named block in child template that contains unique code within block tags
  + {% block blockName %} uniqueCodeHere {% endblock blockName %}

Why template inheritance is so useful:

* Example: say we want to update entire website to now use bootstrap (https://getbootstrap.com/) we can do this by adding required code to the base template only and through inheritance all other templates will also be updated to use bootstrap classes.

Note on Bootstrap:

* CSS and JS delivered from Content Delivery Network ie there is nothing we need to download.
* Some people use third party packages to use Bootstrap and Django together (this is an additional step not covered in this lecture series)
* We can use bootstrap simply by adding few lines of code into our base template.
  + Meta tags + CSS into head
  + JS into body
* To use Bootstrap specific CSS classes simply add to html as normal. (ex: container class to add padding)

Adding a navigation bar + global styles:

* Add code for navbar and main.
* Uses custom CSS classes >> requires adding files to project.
  + In Django static files like custom CSS classes need to be located within a static directory within our app (App folder >> static folder >> App folder)
* In order to include our css file we need to add {% load static %} to top of our base.html template.
* To load css file we add:
  + <link rel=”stylesheet” type=”text/css” href=”{% static ‘blog/main.css’ %}”>
    - What “static blog/main.css does is generate an absolute URL of the static files and accesses the blog/main/.css

Sometimes to correctly load changes you need to kill webserver and restart it in terminal to reset browser cache and implement CSS changes etc.

* To do hard refresh and clear cache use CMD+SHIFT+R (mac)

To add some html5 + bootstrap classes that make our blog homepage look better paste in article snippet to home.html template.

If we hardcode the routes in templates anytime we want to change routes we would also have to change the href attribute in our templates.

* Solution is to use the Django URL tag to get the absolute path to a URL pattern
  + Href = {% url ‘blog-home’ %} where blog-home will be the name we assigned to our URL pattern.
  + It is important that we make the URL path names unique as anytime we reference a path name it will go the associated view.
    - example the name for the home path ‘blog-home’ goes to the home view ‘views.home’.
  + This way every time the URL changes we do not have to change all references to it manually.
* Note: If we try to use a URL tag with a URL pattern that does not exist we will get an error.

**Lecture 4**

**L4 aims:**

* **Learn how to access the admin page of Django site**
* **Learn what admin page is used for (basics).**

Admin page is good way to see what data is on your site, gives nice GUI for CRUD on the data.

Before accessing admin page we need to create a superuser.

* Use Command line >> python manage.py createsuperuser.
* Before we can create a user we first need to create the DB to use with this project.

To create DB we run a few DB migration commands (apply changes to DB).

* First DB migration will create DB and add default tables (such as auth\_user table).
* Use “python manage.py makemigrations” to create DB + default tables.
  + Makemigrations detects changes and prepares Django to update the DB, but it doesn’t actually apply those changes.
* In order to apply the migrations use “python manage.py migrate”
* Now we can run earlier command to create superuser (see above).

Cool thing about Django is that it comes loaded with admin site.

* This allows us to do work on the backend that would take lots of effort to implement if we were to do this on our own.
* Example: Automatic added user account security.
  + User passwords are automatically stored as hashed value and on login the password input is hashed and compared to the stored hash value (see user account to verify)
* Can also change email, add firstname/lastname, change user permissions, etc
* Can create user using “+User” button.
  + User is created as active user, but not given staff (access admin site) or superuser statuses (max priviliges).

**Lecture 5**

**L5 aims:**

* **Creating DB tables for blog applications (so that we can create real posts instead of relying on dummy data)**
* **Learn about Django ORM (allows us to access DB in an easy to use OO way, can change DB without altering code).**

DB structure is represented as classes which are referred to as models.

Django automatically creates models.py file

Django already has built in authentication system and already has User model created, so we don’t need to create another Users model.

Create Posts model to represent user posts (Class Post(models.Model))

* To create field in model use “fieldname = models.fieldType(constraintsHere)
  + Eg “title = models.CharField(max\_length=100)”
  + Eg “content = models.TextField() [notice no constraints required]
* For date field you can use constraints:
  + “Auto\_now=True” – sets to datetime now.
  + “Auto\_now\_add=True” – sets to current datetime only when object is created.
  + “default=timezone.now” (requires import timezone)
    - Notice no () after now eg now() as we want to pass function in default parameter.
* We also need author for each post (user who created post)
  + As user is stored in separate Users table First import User model
    - Use “from Django.contrib.auth.models import User”.
  + The Post model and the User model are going to have a 1:many relationship as 1 user can author many posts.
    - Use “author = models.ForeignKey(User, on\_delete=models.CASCADE) to enforce this.
      * “On\_delete” value tells Django what to do when user who created this post is deleted.

After making changes to our DB we need to re-run migrations to apply the changes.

* Use “Python manage.py makemigrations” to detect and prepare changes to be applied.
* The file created by makemigrations command contains information on what will happen when we run the “migrate” command.

Before running migrate command you can view the actual SQL code that will be run on the database. This is good if you are having issues and need to see the SQL code that will be generated when the migrate command is run.

* Remember the app name and the migration number eg blog and 0001
* Use “python manage.py sqlmigrate appName MigrationNumber”
* Using models class allows us to interact with DB without knowing actual SQL code. This is why these ORM are so convenient.

Migrations are useful as it allows us to make changes to our DB schema even after it is created and already contains data.

Without migrations, you would have to connect to your database and type in a bunch of SQL commands or use a graphical tool like PHPMyAdmin to modify the database schema every time you wanted to change your model definition.

Migrations also let us make DB changes over time.

To run Django python shell use “python manage.py shell” In which we can run python code and also work with our Django objects.

* Import both Post and User models to allow us to access data.
* “User.objects.all()” returns all objects in User model (returns QuerySet)
* “User.objects.first()” returns first object all() QuerySet.
* “User.objects.filter(username=’Niral’) when you want to get all objects that match your lookup parameters
* “User.objects.filter(username=’Niral’).first() returns first object in QuerySet returned by filter.
* Use .attributeName to access attribute of a returned object (Eg User, Post ,etc).
* Use User.objects.get(attributeHere=ValueHere) to use  when you want to get a single unique object.

To control how a DB object is described in a QuerySet define the dunder “\_\_str\_\_\_” method in your model class.

To create a post in shell:

* Get a user object “user = User.objects.first()”
* post = Post(title=’Blog 1’, content=’contentHere’,author=user)
* Alternatively you can set author\_id = user\_id instead of author = user.
* post.save() to save post to Post model.

You can also access data from another table through a foreign key.

* Example:
  + author is FK in the Post model (linked to User table PK), so to get a the author’s email (contained in User model) we can use “Post.objects.first().author.email”.

Django adds a special query set to the user model which allows us to get all DB objects (rows) in another table that is linked to a particular user (via a foreign key).

* Example: Use “user.post\_set.all()” to get all posts by the user stored in the user variable.

Using special user query set Django allows you to also create a new post for that user.

* user.post\_set.create(title=’…’, content=’…’)
* Note: we do not have to add author argument as Django auto fills it as it knows what user we want the post’s author field to set to.
* Note: we also do not have to save() the post as Django automatically adds the created post to the Post model for us.

Using DB data in our templates

Import Post models into views.py in our blog app

* From .models import Post (dot represents current package)

Change value in context dictionary of ‘posts’ to “Post.objects.all()”

* Ensure attributes referenced in templates match the name of fields in the Post DB otherwise this wont work.

Notice dates are not the same on blog app site as in our dummy data\*. This is because the template uses the datetime from the DB without any formatting.

To change to Month Day Year:

* Within template tags there are different default filters that we can use to change our data around (order).
* For dates there is a date filter
  + Use “post.date\_posted|date=”DjangodateFormatString”
  + To get date formatting code see Django date documentation (https://docs.djangoproject.com/en/2.0/ref/templates/builtins/#date)

Edit (post) model data within the admin page of django site.

In Django site admin page to see our Post model, we have to register it with our admin page.

* In our blog app directory >> admin.py
  + Import model using “from .models import Post”
  + Use “admin.site.register(Post)

Now you can interact with Post model from the admin page (create/update/delete/change author etc)

Using convenient default admin GUI we can manipulate our models with ease.

**Lecture 6**

**L6 aims:**

* **Learn how to use HTML forms.**
* **Learn how to validate user input (create user registration page)**

First thing is to think about how the user logic will relate to our project as a whole.

* The user account is going to have its own forms and templates and routes etc so it is logical that it will be separate from the blog itself.
* Best thing to do is to create a new app within our Django project where all of the user related items will be contained in its own section.
* This way we know where to look when we want to add/change/delete user information.

Create new app using “python manage.py startapp AppName” and add it to list of installed apps in Django\_project/settings.py. Name to add can be found in App/app.py.

If you were to create forms from scratch it can get quite complex quite fast. You will have do validation checks, ensure hash passwords match, use regex to ensure valid email etc….

Django takes care of a lot of this for us on the backend. It is similar to the DB models in that we can create python classes and these classes generate html forms for us. Some classes already exist in Django (eg user creation form).

* To create html form using default classes (eg UserCreationForm) remember to Import before.

In app/views.py create form and pass into render function as context variable.

* Render(requests, appName/formName.html, {‘form’:form})

Remember: When creating appName/templates folder also create subdirectory called appName!

Django lets you reference a template in another app from within a different app on the site.

When using a html form we need to add a CSRF token in the html code – a hidden tag that adds a cross site request forgery token that protects our form against certain attacks. If you do not add it your form will not work.

* Use {% csrf\_token %} to add token.

After creating register.html template we now need to create/add the URL pattern.

We could create a URL module in our users app and included it in the Django\_project/urls.py but instead we can use the view directly by importing it in the Django\_project/urls.py

* Use “from users import views as users\_views”. The user\_ is required as you can import multiple views from multiple apps (eg user\_views + blog\_views etc)

Use form method “as\_p” to render form in <p> tags which will split the form up for better aesthetic.

* Use {{ form.as\_p }} in register.html

Writing register user business logic (view):

* To check request method in a view type use “request.method == TYPEHERE” (Eg POST) with conditionals to act/not act on depending on POST payload (eg register view).
  + Remember: when a form is submitted a POST request is sent back to the same route >>> hence checking whether request = POST is required.
* Use “.isvalid()” method on form variable ie form.isvalid() to check if form contents are valid or not.
  + The validated form data will be contained in the “form.cleaned\_data” dictionary (converted into python types automatically).
    - Eg “username = form.cleaned\_data.get(‘username’)
* Flash messages are convenient way to display one-time-alerts to user and will disappear on the next request by app (eg on refresh page).
  + To use flash messages import messages class “from Django.contrib import messages”
  + Several message options (debug,info,success,warning,error)
  + Pass request and string to be displayed to message.method(r,s).
* After form is successfully submitted redirect user to homepage
  + Import redirect “from Django.shortcuts import redirect”
  + Use “redirect(urlPatternName) (eg ‘blog-home’)

Update base template so that any flash messages pop up on any page (as all pages extend base template)

* Display messages above content block
* Use {% if messages %} {% for message in messages %} CODE {% endfor %}{% endif %} to display all messages.
* For action use <div class=”alert-success”> to display message using bootstrap class.
* Bootstrap class shares same name as Django message method tag (eg success) so pass in message tag into class to have different alert-class for different message type.
  + Use “alert-{{ message.tags }}”

If form is submitted with invalid entries but for example clashes with a pre-existing user, the form will rendered again with an descriptive error message (done automatically by Django)

Add form functionality so that it saves user when the form validates

* To save user use “form.save()”
  + This automatically hashes password etc in background for us.

Create a custom form that inherits from UserCreationForm (default user creation form provided by Django)

* To do this create a new form that inherits from UserCreationForm().
* Create the new form.py file in app root directory.
* Import forms, User model, UserCreationForm
* Add custom field(s) (eg email) “email = forms.Emailfield()”
  + ‘required’ kwarg defaults to true and makes field mandatory.
* Create subclass Meta
  + gives us nested namespace for configurations and keeps the configurations in one place.
  + Within configurations we are saying the model that will be affected will be the user model and the fields are the fields that we want in the form and in what order.
  + Use “model = User” as whenever form validates its going to create a new user.
  + Fields list dictates what fields we want in form (from parent + custom fields) and in what order.

To use custom form in a view import formName using “from .forms import formName”

Improving a form’s appearance and displaying validation feedback more clearly for user.

* We can set classes to our form fields in our forms.py file but that is not a good place to do this as it mixes our presentation with back end logic and it would be better if we could implement styling within the template.
* Use 3rd party library CrispyForms to more easily style forms in templates.
  + Allows us to put simple tags in templates that allow us to style forms in a bootstrap fashion.
  + Other CSS frameworks that can be used with CrispyForms aswell.
  + Install CrispyForms using “pip install Django-crispy-forms” in terminal.
    - Post install need to tell Django that this is an installed app >> do this in Django\_projects/settings.py
  + Tell CrispyForms what CSS framework we want to use with it
    - In settings.py set CRISPY\_TEMPLATE\_PACK = ‘bootstrap4’
* Load in CrispyForms tags in template using “{% load crispy\_forms\_tags %}” allowing us to use crispy filter on any of our forms.
* Use “|crispy” to use crispy form filter to handle styling of form (same as using date template tag).
  + There are ways to customise styling further (see crispy docs)
* Looking a html code in registration page you can see that the crispy tag adds many classes to our html form (including bootstrap classes) to give form bootstrap-like aesthetic.
* Now the error messages and corresponding input fields are highlighted in red which indicates the error much more clearly to the user.

**Lecture 7**

**L7 aims:**

* **Create authentication system to allow users to log in/out.**
* **Set authentication system up so that users need to be logged in to access certain pages.**

Users are not able to log in using their created accounts (as they do not have access rights to allow them to use admin login portal)

Need to create login front-end to allow users to log into their accounts. Django takes care of a lot of the back end (of the login process) for us by default.

In Django\_projects/urls.py:

* import (default) login and logout views
  + Use “from Django.contrib.auth import views as auth\_views”
* Remember to name views to avoid view-name collision across apps (see previous explanation)
* Use path(‘login/’, auth\_view.LoginView.as\_view(), name=’login’) to create login path
* Use path(‘logout/’, auth\_view.LogoutView.as\_view(), name=’logout’) to create logout path.
* LoginView and LogoutView are class based views – explained in later lecture!
* The built in views for login/logout will handle the views and the logic for us but will not handle the templates (good as we can make custom templates to meet our criteria)

Note on Django errors:

Django errors are very useful as they point us in the direction of what we need to change in order to solve the error.

Accessing the ‘/login’ path causes Django to display an error as we have not created a template yet.

* We could create users/templates/registration folder and create a login.html template there (as before) but it would make more sense to have the login.html template alongside our register.html template in the users/templates/user folder.

To pass template to default class based login/logut views add “template\_name=’users/login.html’” as a argument to the as\_view() method in login/logout paths.

* If no template is passed default template is used (it looks like logout page in admin portal) >> the “log in again” link redirects to admin portal login (not ideal).

Login.html template will be similar to register.html template, it will extend the base template and have a form passed in which it will then display.

If you login with a valid user, Django will then attempt to navigate (by default) to the ‘/accounts/profile page. This behaviour can be changed in our settings.py file.

* Add “LOGIN\_REDIRECT\_URL=’blog-home’

Change our register route so that users are redirected to the login page after they successfully register.

As default template used by default login/logout views expose users to admin portal login after they log out, we need to create a login and logout front end that is usable for all types of users.

* Create logout.html template in users/templates/users folder (use login.html as base remove unneeded parts)
  + Redirect to login page.

Changing the navigation bar depending on whether a user is logged in/out.

* Add links to ‘login’ and ‘register’ links using Django url tags >>> {% ‘url ‘urlPathName’ %}
* Change displayed navbar elements based on whether user is logged in or not
  + Django provides ‘user’ variable (model object) that has attribute ‘is\_authenticated’ that allows us to tell if a user is logged in or not.
  + Use is\_authenticated variable in base template to display navbar links based on whether user is logged in.
  + Good practice to give users visual feedback to further assure their current status.

Only allow users who are logged in to access certain pages (Eg user profile page)

* Create view in users/views.py for profile.
* Create profile.html in users/templates/users folder
* Create profile route (path) in django\_projects/url.py to use this view.
* Put a check that only allows users who are logged in to access a page.
  + Use a @login\_required decorator provided by django.
    - Import decorator “from Django.contrib.auth.decorators import login\_required”
    - Add decorator above view you want to enfore login\_required on.
      * If you use class-based views then the process of using login\_required is a little different (covered in later lecture)
    - When user is not logged and attempts to access a route where login is required Django redirects by default to /accounts/login
    - as we have used /login for our login page we need to add a variable LOGIN\_URL=’loginRouteName’ to django\_projects/settings.py file to tell Django to use that as default login route.

Built in to the Django login view is the feature that Django keeps track of the page the non-authenticated user was attempting to access (in url ‘next=/PAGEHERE/’) so that once they have logged in they can be redirected there.

**Lecture 8**

**L8 aims:**

* **Learn how to upload an image (to user profile page)**
* **Learn how to use Django signals to perform specific functions after certain actions**

Default user profile model (provided by Django) does not have field for user profile picture.

Solution: Extend the User model and create a new Profile model that has a 1:1 relationship with the User.

* This will be a new model in our users app.
* See earlier notes on creating new model (Eg Posts model)
* To create 1:1 relationship with User model
  + “user = models.OneToOneField(User, on\_delete=models.CASCADE)”
    - For info on models.CASCADE see earlier notes
* Now we can add any additional fields we want (ie image)
  + Use “image = models.ImageField(default=’default.jpg’, upload\_to=’profile\_pics’”
    - Default dictates the default value to use
    - ‘upload\_to’ dictates directory that images get uploaded to when users upload an image to their profile.
* Create” \_\_str\_\_” (dunder str) method to dictate how we want a profile to be displayed when we print it out (ie in a DB query)

Install Pillow library (library for working with images in python) to allow us to use our image field.

* ‘Pip install Pillow’

Remember:

* use ‘makemigrations’ to apply changes to model and then ‘migrate’ to apply changes to DB.
* To view a new model in our admin site we need to register model in admin.py file
  + Use admin.site.register(ModelName)

As the Profile model now has a 1:1 relationship with the User model we can access a user’s profile from their user object

* In terminal obtain a user object >>> user.profile
* Obtain (default) image attributes
  + User.profile.image.url
  + User.profile.image.width
  + etc

Notes in Images:

* In case of duplicate image files with duplicate names, hash appended to duplicate to ensure names are unique.
* On website we wont be accessing the image directly, instead we access the location of the image as in HTML we use it as the “src” attribute value.

Default location Django saves images is in Django\_project (root) directory which clutters up root directory

* better to add + set MEDIA\_ROOT and MEDIA\_URL to settings.py to change default directory to save images in >>
  + Use ‘MEDIA\_ROOT = os.join(BASE\_DIR, ‘folderToSaveIn)
  + MEDIA\_URL = ‘/folderToSaveIn/’
* Add settings so that image files can be found by browser and viewable from admin site.

Display images on blog app site

* We want profile to hold username, email, profile picture, etc >> add profile html snippet
* Access django’s provided user variable to set username, email, image to that of the logged in user. (see above on how to access image)
* Need to add media routes to our url patterns so that static media files are viewable in browser (different ways to do this in development and production)
  + Add “+ static(…) to url patterns to serve user uploaded static files during development.
  + More info: https://docs.djangoproject.com/en/2.1/howto/static-files/#serving-files-uploaded-by-a-user-during-development

Uploading a default image (for when no image uploaded)

* Add default.jpg to media folder

Want to ensure when a new user is created they automatically get a profile as well.

* Current implementation requires manual creation of profile in admin site.
* Solution : use Django signals
  + In users app create signal.py file (sometimes this file is put in models.py file where Profile model is created, but Django docs recommend creating it in users app directory to avoid import issues)
  + In signals.py import post\_save
    - “from Django.db.models.signals import post\_save
  + This is a signal that gets fired after an object gets saved.
    - in our case we want signal to fire when user is created.
  + User model is known as sender (what sends the signal) (remember to import)
  + Create a receiver (function that receives signal and then performs some task)
    - “from django.dispatch import receiver”
  + Import Profile model as receiver will be creating a Profile for a new user.
  + Create a create\_profile(…) function and add logic to create new profile.
    - Eg “Profile.objects.create(…)
  + Add logic to link create\_profile() function to receiver
    - Add @receiver(signal,sender=user) decorator to function.
* How signal logic works:
  + When sender does signal action (post\_save) send signal
  + signal caught by receiver (create\_profile())
    - receiver function takes arguments passed in by signal and runs function
      * arg: instance of the user
      * arg: created (whether user was created)
    - receiver function will create Profile object with user=instance (if created=True).
* Create additional function that saves the user’s profile every time a user object gets saved. (identical to create\_profile() signal-receiver)
* (Django docs recommended) Final step to get signals working: import signals inside of the ready function (within the UsersConfig class) in the users/apps.py file
  + Create “ready()” function in which you import “users.signals”

**Lecture 9**

**L9 aims:**

* **Allow users to edit their profile on profile page**
* **Allow users to upload profile image on their profile page**
* **Set up auto image resizing when user upload image (space saving)**

In forms.py create a model form (allows us to create a form that will work with a specific database model – in our case we want a form that will update our Users model) to allow users to update their profile

* Use “class UserUpdateForm(forms.ModelForm)” to create user update model form.
* See earlier notes on how to create a custom form.
* As profile image is stored in Profile model we need to create a second model form to interact with Profile model.
* Even though we created 2 model forms, on our site it will appear as one single form.

After creating custom model forms import them in views.py.

* Use “from .models import UserUpdateForm, ProfileUpdateForm”

Then create instances of forms in profile view (and pass forms in context dictionary to render function).

Add html for form in profile.html template (copy form from register.html template)

* Add both forms with crispy formatting
  + “{{ u\_form|crispy }}
  + “{{p\_form|crispy }}
* Stacking forms like this in html code make them appear as a single form on page.

We have to add special encoding type to our form in order for it to pass our image data for out profile image properly

* In form add attribute enctype =…
  + <form method=”POST” enctype=”multipart/form-data”>
  + If you don’t add this attribute it can appear that your site is saving the form but it wont actually be saving the image (hard to debug!)

To have fields in profile update form pre-populated with current values (ie current username and email)

* To populate forms with current info
  + Remember that modelForms expect to be working with a specified model object so to prepopulate the form simply pass in the expected model.
    - Add instance=request.users as argument to UserUpdateForm
    - Add instance=request.users.profile as argument to ProfileUpdateForm

If request.method = POST then perform validation checks on payload data, else display pre-populated form as before.

* To pass in POST data into model form pass “requests.POST into model form.
  + “UserUpdateForm(requests.POST, instance=request.user)”
* For ProfileUpdateForm we are also going to be getting some additional POST data (file data coming in with the request) so pass “request.FILES” into form as well.
  + “ProfileUpdateForm(request.POST, request.FILES, request.user)”
* Then perform validation checks on both forms in a conditional.
  + “if u\_form.is\_valid() and p\_form.is\_valid():

If form data passes validation checks Use “.save()” function to save form data to model.

Once form info has been update provide feedback to user that they have updated their profile.

* Use a one-time flash message
* Then redirect them back to the profile page.
  + You need to use redirect here as opposed to letting the execution fall down to render() method due to post-get-redirect-pattern convention.
    - Using redirect avoids the popup that asks user if they want to resubmit the form when they reload the page
      * This is due to the browser making another POST request when the page is reloaded.
    - A GET request avoids making this additional POST request when the page is reloaded (as the browser performs a get when using redirect)

After performing manual check to see if functionality works correctly good practice is to check in admin site that changes have been applied correctly.

Resize image files to save space on disk.

* No use having large image that then gets scaled down to small size.
* Large images cause site to run slower as Django has to send large image to browser every time it appears.
* To resize image use ‘Pillow’ package.
  + Override save() method of our Profile model
    - This save method is the method that runs after our model is saved. It already exists in the parent class but we are overriding it so we can add some additional functionality.
    - Implement the superclass’s (parent) save() method, and then add additional functionality.
    - “super().save()” – runs parent class’s save method.
  + Open image for the current profile instance and resize
    - Img = Image.open(self.image.path) – opens image
    - Use conditional check to see if image height + width > limit (300px)
      * If over limit resize image to limit (300px)
        + Use “img.thumbnail(output\_size)
  + To save resized image use img.save() to overwrite the path of the current image.
    - Use “img.save(self.image.path)”
  + Note on resizing image: lots of different methods, some more complex, there are definitely trade-offs to each method. Also 3rd party packages that will do resizes etc for you more efficiently (for larger applications more beneficial)

After resizing you could add code to do additional cleaning such as delete previous profile picture file etc.

Display the image of the author beside each post on blog homepage.

* Add src={{ post.author.profile.image.url }} to image html element in home.html template. Add bootstrap and custom classes for styling.

**Lecture 10**

**L10 aims:**

* **Adding ability for users to add posts that will be displayed on homepage.**
* **Use class based views in order to work with our Post model**
* **See how Django class based views can be useful.**

So far we have been using only function based views (url >> url-pattern >> view loads html page and handles logic)

Class based views have much more built in functionality that will handle a lot of backend logic for us.

Different kinds of class based views (list views, detail views, create views, update views, delete views and more)

* Eg on a blog you will have page that lists all blog posts >> use a list view here.
* Eg Youtube has a subscription page that lists all videos from subscribed channels >> use list view here.
* Eg Clicking on a video in subscription list would take us to different page that contains more details (content, comments, etc) about the video >> use detail view here.
* Eg Ability to update/delete video on your channel >> use update/delete view here

Django tries to predict some of this common behaviours (Eg list,detail,update, etc) and provides generic views that handle a lot of the backend logic for us.

Home view would be a good candidate for a list view as it lists all blog posts.

* First “from django.views.generic import ListView”
* Use “class PostListView(ListView)” to create list view.
* In list view need ‘model’ variable that tells the list view what model to query in order to create the list.
  + Set “model = Post”
* When we use class based views we cant simply pass the view into the path() function like with function based views. We have to convert it into an actual view
  + Use “PostListView.as\_view()”
* By default class based views look for templates with a specific naming pattern “<app>/<model>\_<viewType>.html”
* We could create template with this naming convention, but instead its better to change which template that we want this view to use.
  + Add “template\_name = ‘blog/home.html’ in list view class.
* This still wont work as by default the items to loop over in the template are called “object\_list” so either we change “posts” >> “object\_list” in home.html or set a variable in our list view class to tell it to loop over an object with a different name (posts).
  + Add variable “context\_object\_name = ‘posts’”

Correct ordering of posts on homepage.

* Add variable “ordering = [‘-date\_posted’] to order from newest >> oldest (remove minus sign to get oldest >> newest ordering.

Difference between function view and list view (home view example)

* In a class based view we are setting a few variables
* In a function view we have to render a function and explicitly pass in some information.
* You can get a working class based view with a single line of code if you stick to the default naming conventions perfectly (eg looping over “object\_list”).

Creating a list view following perfect naming conventions (for post details)

* Ie no additional variable definitions required in class (apart from model)

Use detail class based view for each object (post) we are going to be looking at the details of each object.

* First “from Django.views.generic import DetailView” in blog/views.py
* Import ‘PostDetailView’ view in urls.py so you can pass view into path function.
* We need to create a url pattern that contains a variable which is then passed into view (in background).
* Django gives us ability to use variables in path function url argument.
  + Use ‘post/<int:pk>/’ to tell Django that we expect an integer after the ‘post variable in url and to use that as the primary key (pk).
  + Use ‘path(‘post/<int:pk>/’, PostDetailView.as\_view(), name=post-detail)’
  + Defining what we expect the type of the ‘pk’ variable to be allows Django to discard anything else passed as ‘pk’ in the url.
  + By specifying the ‘pk’ variable in the url allows django to use the ‘pk’ value from the url in our view function.
  + Following expected (default) naming conventions and naming url variable ‘pk’ allows the detail view to grab that specific object (without adding” ‘renaming’ variable to our view class) as that is what it expects the variable to be named.

Create template to display post details

* Remember by default template reference is expected to be called “<app>/<model>\_<viewType>.html”
  + so the detail view expects “blog/Post\_detail.html”
* Create file in blog/templates/blog as “Post\_detail.html”
  + Paste + modify code from ‘home.html’ (see vid around 16:00)
* When dealing with detail views it expects the context of this template to be called object >>> change all ‘posts’ references to ‘object’ in ’post\_detail.html’ (as we copied html from home.html).
* Remember: Verify changes in browser!

Adding links to individual posts (pages) on homepage (home.html)

* Add “href={% url ‘post-detail %}” (Django url tag) in <a> tag containing post title.
* We also have to add post.id as ‘pk’ parameter required in URL.
  + “href={% url ‘post-detail’ post.id %}”

Going to url for which a post does not exist ie “/post/<non-existant-post-number>” results in ‘404 Page not found’ http status code.

Next create a create, update, and delete views so that we can do all of the required user actions on the front end of our blog app site.

Tip - use parentheses to break up an import line into multiple lines!

Add code to allow users to create new posts

* First “from django.views.generic import CreateView”
* Add “class PostCreateView(CreateView):”
* Define model variable
  + “model = Post” (sets model to query for objects)
* Define fields we want in form
  + “fields = [‘title’, ‘content’]”
* In ‘urls.py” create url-pattern for CreateView
  + “path(‘post/new/’, PostCreateView.as\_view(), name=’post-create’)”

Creating a post creation template

* As CreateView and UpdateView share a template django expects the template name to be “<model>\_form.html” for our PostCreateView.
  + Create file “post\_form.html” in “blog/templates/blog”
* PostCreateView expects form to be (default) named “form”
  + Form will be printed out using crispy forms formatting and as template code is similar to our register.html template so copy code and make necessary modifications (legend + submit button + link)

The main use case for class based views:

* they are very useful as they allow us to add url routes without having to deal with a lot of the back end logic (eg creating a form etc). In our example (above) instead we just pass a model and form parameters and the class based view will implement the required code for us.

Attempting to create a post with our newly created ‘/posts/new’ page results in “IntegrityError - NOT NULL constraint failed: blog\_post.author\_id” which occurs as our post is being added to the Post model without an ‘author’ value which cannot be null.

We can solve this by overriding the “form\_valid” method for our PostCreateView which will allow us to add the author before the form is submitted.

* In view add function “form\_valid(self, form)” in which the author is set to the current user
  + Use “form.instance.author = self.request.user” to set author attribute.
  + Return “super().form\_valid(form)” which calls superclass’s form\_valid method that we have overridden, passing in the form.
* By overriding the view’s form\_valid() method, setting the form’s author attribute to the current user, before calling the superclass’s form\_valid(form) with the form (edited to include an author value) we are able to satisfy all the constraints to create a new post entry in our Post model.

Creating a post however will still not work as we get an “ImproperlyConfigured” error which states that the post was created successfully but it doesn’t know where we want to be redirected to now (see Exception value for full details)

Ideally we would redirect to the post detail page of the post we just created.

* The way to tell Django to find the url of a model object is to create a “get\_absolute\_url” method that returns the path to any specific instance.
* We create a “get\_absolute\_url” method in the Post model so that Django knows how to get the url location of a specific post.
  + First we obtain the URL of a specific route
    - “from Django.urls import reverse”
    - To do this we use the reverse function which returns the full url to that location as a string
      * “reverse(<routeName>, additional args)
      * Eg: “reverse(‘post-detail’, kwargs={‘pk’:self.pk})
    - We don’t use redirect as we only want to obtain the full url string of a post NOT go to that url (for which redirect() would be used)
  + Then we let the view handle the redirect for us (ie go to the url location).

If you wish to send user to homepage after creating a post you can define a success attribute called “success\_url” and just set that to the homepage instead.

Notice: the large amount of functionality that Django provides for us after just filling in a few lines of code (creating the create view + setting a few parameters/methods).

* This would have taken a lot more work using other frameworks.
* This would have taken more work using function based views also.

We shouldn’t allow users to create a post unless they are logged in.

* If users try to access the ‘posts/new/’ route to create a post they should be redirected back to the homepage.
* Cannot use a @login\_required(…) decorator on classes so instead we use a ‘login mixin’ a class that we inherit from that adds the logic functionality to the view.

Using Django Mixins (login example)

* In blog/views.py “from Django.contrib.auth.mixins import LoginRequiredMixin”
* Have view you wish to require login inherit from the LoginRequireMixin class
  + Ensure all Mixin’s are to the left of the inherited view argument.
  + “Class PostCreateView(LoginRequireMixin, CreateView):”

Create an UpdateView to update posts on front end

* First “from Django.views.generic import UpdateView”
* Create update view (copy create post view as code similar)
* Import PostUpdateView into urls.py
* Create url path in urls.py named ‘post-update’
  + To know which post to update we need to pass in url variable ‘pk’
  + Use url ‘post/<int:pk>/update/’ in path(…) function
* Since we provide ‘pk’ to url to the post that we want to update, Django update view will take care of the rest of the logic in the background.
* (mentioned earlier) The UpdateView and CreateView share the same ‘post\_form.html’ template, so no additional template is required.

When building an large app it is important to consider the ways that a user could attempt to maliciously use your app.

* Example: if we do not check that the current user is the author of a post they could attempt to edit other user’s posts.
  + Solve this issue by using ‘UserPassesTestMixin’
    - Import UserPassesTestMixin
    - In view create method testFunc() used by UserPassesTestMixin to validate whether current uses passes a “test” (ie is the author of a post).
      * Get the exact post we are currently updating using method of the UpdateView “get\_object()”.
        + Use post = self.get\_object()
      * Use conditional to get if current user = post.author
        + If true >> return true
        + Else >> return false

Create a PostDeleteView for deleting posts.

* DeleteView is very similar to DetailView
* First import DeleteView (see above for full import code)
* To ensure user is logged in and is the author of the post they are trying to delete inherit from LoginRequiredMixin and UserPassesTestMixin
* Create route with URL argument ‘post/<int:pk>/delete/’.
  + “path(‘post/<int:pk>/delete’, PostDeleteView.as\_view(), name=’post-delete’)”
* Template that PostDeleteView expects is just a page that asks if we want to delete the post named ‘post\_confirm\_delete.html’.
  + Copy html code from post\_form.html (simple form)
  + Remove {% load crispy\_forms\_tags %} and {{ form|crispy }} as this template does not require a form be passed in.
  + Add appropriate styles via classes to ensure user is clear regarding options.
    - Add cancel link that takes user back to ‘post-detail’ route.
  + Note: When error occurs post will not be deleted
  + In PostDeleteView define success\_url attribute (variable) determining route to redirect to if deletion is successful.
    - “success\_url= ‘/’”

Finally make a few changes to blog app site to get the Create, Update, Delete post functionality working a bit better.

* Add link in navigation bar for user (that is logged in) to create a post
  + Use “ <a class=”nav-item nav-link” href=”{% url ‘post-create’ %}”>Create Post</a>”
* Add links in post\_detail.html to update and delete post.
  + Use conditional to only show update and delete links if current user = author of post.
    - {% if object.author == user %}:

**Lecture 11**

**L11 aims:**

* **Add pagination to blog app site to avoid loading too many posts at once**
* **Learn how to create a page of posts filtered by a specific user (also paginated)**

Rather then loading all posts which slows down site loading better to add pagination allowing for a certain number of posts to be loaded and then provide links to additional (pages) posts.

Paginator example (shell):

-‘from Django.core.paginator import Paginator”

- “posts = ['1','2','3','4','5']”

- “p = Paginator(<list>,<objectsPerPage>)”

- “p.num\_pages” attribute containing number of pages

- “p1 = p.page(1)” Stores current page object in variable.

- “p1.number” the page number of page variable.

- “p1.object\_list” attribute containing all objects contained in page object.

- “p1.has\_previous()” returns boolean indicating whether there is previous page in Paginator object.

- “p1.next\_page\_number()” returns number of the next page in Paginator object.

- “p1.previous\_page\_number()” returns number of the previous page in Paginator object.

- “p1.number|add: ‘<number>’ “ adds positive or negative number to the page number.

- “p1.paginator’ get paginator of page object.

Remember: class based views take care of a lot of the background logic when loading a view.

* Pagination is no exception, to use pagination we define a variable within the view
  + In PostListView add “paginate\_by = <objectsPerPage>”
  + Adding this variable simply implements the pagination, it does not however create links to other pages (you can manually access then by passing page parameter in url).

Add links in home.html template to allow navigation to other pages.

* Class based view already passes in the context required to access pages in home.html
* Add pagination links at bottom of page (under the looping over posts)
  + Use “{% if paginated %} to check if page supports pagination
  + Use “page\_obj” to refer to current page object (default name expected)
  + Several ways to style paginated links
    - If current page object has previous page(s) then add <a> tags containing links for ‘first’ and ‘previous’ page.
    - Loop over all pages and add <a> tags with link for page if page number is within +-3 of current page number.
    - If current page object has is not final page (ie has next page) then add <a> tags for ‘next’ and ‘last’ pages.
    - (see <https://www.youtube.com/watch?v=acOktTcTVEQ&t=269s> for detailed explanation.

Add link to user’s name on post which then displays posts only by that user.

* Create list view called UserPostListView (that inherits from ListView) (similar to PostListView)
* Need to add filter to view that only gets posts by a specific user. (default returns all posts on blog)
  + In order to modify the QuerySet that the view returns we can override the function “get\_queryset” and change the QuerySet from within this method.
    - Within this method we obtain the user to filter by from the URL. If the user does not exist return a 404 http status code.
    - To do this use” get\_object\_or\_404” function with User model passing in the username from the url (Remember: import both first).
      * Use “user=get\_object\_or\_404(User, username=self.kwargs.get(‘username’))”
    - Return the query “Post.objects.filter(author=user)”
    - Overriding the” get\_queryset” method overrides the default query remove variables that are used in the default query (eg ordering etc) and instead add to query code as method (eg ordering variable >>> .order\_by(…))
* Create url route (path) to ‘user/<str:username>/’ remember to add url variable called username (as we use it in custom get\_queryset method)
  + “path('user/<str:username>/', UserPostListView.as\_view(), name='user-posts')”
* Create template for UserPostListView view called ‘user\_posts.html’ (similar to home.html)
  + Keep pagination logic as paginiation can occur if user has enough posts.
  + Add <h1> heading tag as title of page containing username and number of posts
    - Use “{{ view.kwargs.username }}” to get username from URL (like above)
    - Use “{{ page\_obj.paginator.count }} to get total number of posts across all pages.
    - (see code/video for details)
* Update href of post author <a> tag to go to user-posts url path.
  + Use Django url tag “{% url ‘user-posts’ post.author.username %} (post or object depending on view type)

**Lecture 12**

**L12 aims:**

* **Add functionality allowing users to reset their password via email.**

Django has default functionality that generates a secure token that only allows a specific user to reset their password.

Like loginView and logoutView the password reset views are built in to the auth views and do not require manual view creation (as we use the default fields ie user+password and do not require custom fields)

Creating password reset views

Create route that provides a form for the user to fill out that sends a password reset instruction to the email (provided in form).

* In django\_projects/urls.py create url path ‘password-reset/’
* Use “Path(‘password-reset/’, PasswordResetView.as\_view(template\_name=’users/password\_reset.html’), name=’password-reset’)”

Creating password\_reset.html template

* Copy html code from login.html and make required changes (change text)

Create route that tells user to check inbox for password reset email (uses PasswordResetDone default view).

* In Django\_projects/urls.py create url path ‘password-reset/done’
* Use “Use “Path(‘password-reset/done/’, PasswordResetDoneView.as\_view(template\_name=’users/password\_reset\_done.html’), name=’password\_reset\_done’)”

Create password\_reset\_done.html

* Copy logout.html code and make required changes (create div containing instructive text telling user to check email)

If we test /password-reset route now (fill in form and submit it) it will give us an error as it will try to reference a path that does not exist yet.

* Reverse for ‘password\_reset\_confirm’ not found.
* The error occurs as Django tries to create a url to the ‘password\_reset\_confirm’ route passing other parameters.
* The name of the template that threw this error is ‘password\_reset\_email.html (a template that Django is using in the background to create the email to send to the user)
* When it tries to create the email it runs into an error as we don’t have a route called password\_reset\_confirm/ (which it attempts to go to passing in 2 parameters uidb64 and token)
* ‘Uidb64’ and ‘token’ parameters need to be within our url (as the view requires them) so that django knows that the person who is accessing the password reset page is the same person who requested the password reset (adds layer of security to these routes).
  + Uidb64 is the user’s id encoded in base64
  + Token is the token to check that the password is valid

Solution: create route for password-reset-confirm expecting uid64 and token parameters to url.

* Use (default auth view) PasswordResetConfirmView
* Set “template\_name= users/password\_reset\_confirm.html”
* “path(‘password-reset-confirm/<uidb64>/<token>/’, auth\_views.PasswordResetConfirmView(template\_name=’users/password\_reset\_confirm.html’), name=’password\_reset\_confirm’)”

Create password\_reset\_confirm.html

* Similar to password\_reset.html (copy + make changes)

Testing ‘/password-reset’ route still does not work and throws a “ConnectionRefusedError”.

* This error is less informative then the previous seen
* What is happening is Django is trying to send an email but is failing (as we do not have an email server configured to send email)

Solution: gmail (one of many solutions)

* look online for others eg setting up localhost to be used as an email server with Django [https://docs.djangoproject.com/en/2.1...](https://www.youtube.com/redirect?event=video_description&v=-tyBEsHSv7w&q=https%3A%2F%2Fdocs.djangoproject.com%2Fen%2F2.1%2Ftopics%2Femail%2F%23configuring-email-for-development&redir_token=ekU-LFuC8MrLiBbQ9d0E7W8f4PZ8MTU1MTUxMzg3MUAxNTUxNDI3NDcx))
* Gmail better example of how it is done in production (and gives better practice on how to send email)

Gmail solution:

* Depending on gmail configuration you may have to edit settings to expect sign in’s from a python (3rd party) application.
  + If you have 2 factor authentication create a password specifically for the application that you want to sign in from.
  + Else set up access from third party application (google this for details)
* Add to ‘django\_projects/settings.py’
  + ‘EMAIL\_BACKEND = Django.core.mail.backends.smtp.EmailBackend’
  + ‘EMAIL\_HOST = smtp.gmail.com’
  + ‘EMAIL\_PORT = 587’
  + ‘EMAIL\_USE\_TLS = True’
  + To access the environment variables containing username + password:
    - ‘EMAIL\_HOST\_USER = os.environ.get(‘EMAIL\_USER’)’
    - ‘EMAIL\_HOST\_PASSWORD = os.environ.get(‘EMAIL\_PASS’)’
* Adding environment variables containing username and password good practice as then sensitive information is not stored in files and instead on local machine.
  + To do this in .bash/profile add new line add “export <Name>=<Value>”
  + Detailed explanation: <https://www.youtube.com/watch?v=5iWhQWVXosU>

Testing ‘/password-reset’ route still does not work and throws a “NoReverseMatch” error which is due to there being no ‘password-reset-complete’ route created.

Solution: create password-reset-complete

* In ‘django\_projects/urls.py create route for ‘password-reset-complete/’ which uses PasswordResetCompleteView and template password\_reset\_complete.html
  + “path(‘password-reset-complete/’, PasswordResetCompleteView.as\_view(template\_ name=’users/password\_reset\_complete.html’, name=’password\_reset\_complete)”

Create password\_reset\_complete.html template similar to password\_reset\_done.html ([informative - does not contain any forms])

* Similar to password\_reset\_done.html (copy + modify: change text, add link to sign in page )

Finally add link to password reset page on blog app site on login page

* Add to login.html template after login button

There are lots of additional improvments you can make to you web app including:

* Write unit tests for application
* Deploy application on multiple different platforms
* Learn how to send longer running requests to a message queue and make it asynchronous.
* Add a commenting system
* Add a search feature
* And many more!

Questions (to be answered):

* Why no urls.py in users app folder?