**Django Notes**

**What is Django**

* Django is a python web server side framework
* Designed to get you started fast
* Is a batteries included framework which means it’s more opinionated in how you build out your app. There are rules you should follow which make it easier to have good designs
* MVT Design Pattern (mode, view, template)
* Django is used by Pinterest, dropbox, Instagram, Spotify

**MVT**

* Diagram

  Description automatically generated
* When someone goes to facebook.com for example, we find a backend function (a view) that matches the url. The view will get some data from the models and render some template back to the user.

**Installation**

* Text

  Description automatically generated
* “python -m venv .venv” creates a virtual environment so that this project is isolated from the rest of your computer
* “.venv\scripts\activate” activates the virtual environment
* “pip install django” installs django
* “django-admin –version” displays django version (env must be activated if django was installed with env activated)
* “django-admin startproject my\_project” creates a startproject folder with a name of my\_project
* “cd my\_project” enters the startproject folder (now inside the root folder)
* “python manage.py runserver” runs the website server at <http://127.0.0.1:8000/>

**Folder structure**

* Graphical user interface, text

  Description automatically generated
* Note that we can move the .venv folder to be inside the my\_project folder (root folder). To do so, we create a requirements.txt file, and create a new .venv inside the root folder and do a pip install.
* Manage.py
  + This file is a command-line utility that lets you interact with your Django project
* Settings.py
  + This file contains the configuration settings for your Django project. Here, you can define your database settings, middleware, installed apps, time zone, static and media file settings, and many other settings that you may require.
* Urls.py
  + This file contains the URL patterns for your project. You can define URL patterns for your apps and map them to the appropriate views.
* Wsgi.py
  + This file contains the entry point for WSGI (Web Server Gateway Interface) which allows a web server to communicate with your Django project. This file is used for deployment on production servers.
* Asgi.py
  + This file is similar to wsgi.py but is used for asynchronous servers like Daphne and uvicorn.
* Init.py
  + This file is empty and tells Python that this directory should be considered as a Python package.
* db.sqlite3
  + This file is the default file-based relational database that comes bundled with Django. It is a lightweight, serverless, and self-contained database engine that stores data in a local file instead of using a separate database server.

**Views and URL Routing**

* In Django, a view is a Python function or class that receives an HTTP request and returns a response. (It is similar to Express.js controllers)
* We will also have to let django know which views get triggered by which routes.
* Inside the urls.py file, we a urlpatterns list which is a variable in Django that is used to define the URL patterns for your application. It is a list of path() or re\_path() function calls that map URL patterns to view functions or class-based views.
* In Django, the path function is used to define URL patterns for views. The path function takes two required parameters and several optional parameters:
* 
* Here's a brief explanation of each parameter:
* route (required):
  + A string that contains the URL pattern to match. This may include variable parts enclosed in angled brackets (< and >), which can capture values from the URL and pass them as arguments to the view function.
* view (required):
  + The view function or class that will handle the request. This can be a reference to a function defined in your views.py module, or a class-based view that defines methods for handling different HTTP methods.
* kwargs (optional):
  + A dictionary of keyword arguments to pass to the view function.
* name (optional):
  + A unique name for the URL pattern that can be used to refer to it in other parts of your application.
* In addition to these parameters, the path function also accepts several optional parameters that can be used to customize how the URL pattern is matched and processed:
* converter:
  + A converter object that can be used to validate and convert the value of the variable captured by the URL pattern.
* regex:
  + A regular expression that can be used to match the URL pattern.
* partial:
  + A Boolean value that indicates whether the URL pattern should match only a partial path.
* prefix:
  + A string that is prepended to the route before matching the URL pattern.
* decorators:
  + A list of decorators that should be applied to the view function.
* These additional parameters provide more flexibility and customization options for defining URL patterns in your Django application.
* Ex:
  + Suppose inside the urls.py file, we have the following
  + Text

    Description automatically generated
  + When we go to <http://127.0.0.1:8000/>, django first looks in the urls.py file within the main folder.
  + Inside the urls.py file, it searches if the URL endpoint is “”.
  + Since <http://127.0.0.1:8000/> has a path of “” which matches path(‘’, home), we execute the home function.
  + Thus, we print out the request  and return an HttpResponse Graphical user interface, application, Teams

    Description automatically generated
* Ex:
* While the previous example works, if we add more routes and views, the main urls.py file will get really complicated and messy. To make our code more organized, we can create applications.
* Note that we can have pass in parameters inside the route
* Ex:
  + Text

    Description automatically generated
  + Notice in the third path, it’ll match a route endpoint that starts with a number. We call this number ‘id’ and access it inside the get\_user view as the second parameter to the function.
  + Thus, if we go to <http://localhost:8000/3/>, we see the following on our webpage: 

**Applications**

* A Django app is a self-contained unit of code that provides a specific set of features or functionality for a Django project.
* Applications can be created by developers to add functionality to their Django projects, and Django also provides several built-in applications that can be used out of the box.
* Applications are designed to be modular and reusable, allowing developers to organize their code into separate components that can be easily shared across multiple projects or open-sourced for others to use.
* To create an application, run this following command inside the root folder
* python manage.py startapp my\_app
* This creates the following “my\_app” folder inside the root folder
* Text

  Description automatically generated with medium confidence
* models.py:
  + This file defines the data models for the application using Django's Object-Relational Mapping (ORM) system. You can define your database schema by creating classes that inherit from Django's Model class.
* views.py:
  + This file contains the logic for handling HTTP requests and returning responses. You can define your application's views by creating functions or classes that handle specific URL patterns.
* admin.py:
  + This file allows you to customize the admin interface for the application. You can define custom admin pages, filters, and actions for your application's data models.
* tests.py:
  + This file contains unit tests for your application. You can use Django's built-in testing framework to create automated tests that ensure your application's functionality works as expected.
* apps.py:
  + This file contains the configuration for the application. You can specify various settings for the application, such as the application name and label, or any dependencies that it requires.
* migrations/:
  + This directory contains database schema migration files. Django's migration framework allows you to modify your application's database schema without losing any data or requiring manual updates.
* You can also create additional files and directories as needed to support your application's functionality (such as a urls.py file)
* All the “startapp” command does is create this folder. Django doesn’t know about it. To let django know about this new app
  + go apps.py file inside the my\_app directory and copy the class name 
  + go to the settings.py file inside the main folder and paste app\_name.apps.ClassName
  + Text

    Description automatically generated
* We previously mentioned how we can use apps to organize our routes/views.
* Ex:
  + Inside my\_app folder’s views.py:
  + Text

    Description automatically generated
  + Notice this views.py file contains a view. In this case, this view is a function called “home” that takes in a request, prints the request, and returns an HttpResponse. We also import the HttpResponse method as needed.
  + Inside my\_app folder, create a urls.py file:
  + Text

    Description automatically generated
  + This file is does normal route to view mapping. Notice we import the my\_app folder’s views.py file via “from . import views” and we access the “home” function inside the views.py file via “views.home”
  + Inside main urls.py:
  + Text

    Description automatically generated
  + In Django, the include function is used to include other URL configurations in your project's urls.py file. The include function takes a string argument that specifies the module containing the URLs you want to include.
  + When we go to [http://localhost:8000/my\_app/](http://localhost:8000/my_app), django first looks in the urls.py file within the main folder.
  + Inside the urls.py file, it searches if the route starts with “admin/” or “my\_app/”
  + Notice [http://localhost:8000/my\_app/](http://localhost:8000/my_app) is a route whose endpoint starts with “my\_app”.
  + Since this “my\_app” endpoint matches path('my\_app/', include("my\_app.urls")), the remaining part of the URL is sent to my\_app/urls.py. Thus, the URL string sent to my\_app/urls.py is “”.
  + Since the URL string sent to blogs/urls.py is “” which matches path('', views.home, name = "home") we run the about method within the views.py file

**Templates**

* Within a view, we could return: HttpResponse('<h1>Blog Home</h1>') to return html
* Theoretically, we could write all our html in that one line with brackets but that would be horrible.
* We can solve this issue by importing html files instead which are called templates
* Creating Templates:
  + create a “templates” directory inside the app folder
  + create a folder inside the this “templates” directory with the same name as your app
  + Inside the “templates/my\_app” folder, create html files which are templates.
  + Text

    Description automatically generated
  + Text

    Description automatically generated
* To let a view know of this template’s existence so that it can return it as a response, we can have following inside the app’s views.py file.
* A screenshot of a computer

  Description automatically generated with medium confidence
* In Django, the render method is used to render a template. We could use HttpResponse(loader.get\_template('my\_app/home.html').render()) which is equivalent but much less conside.
* The render method takes three arguments:
  + request: The HTTP request object that triggered the view.
  + template\_name: The name of the template that should be rendered.
  + context: A dictionary containing the variables and their values that should be passed to the template.
* Now if we go to <http://localhost:8000/my_app/>, we get this output on the webpage: 

**Template Inheritance**

* Like class inheritance, template inheritance allows one html file to inherit tags and elements from other html files
* Creating templates:
  + create two templates, one of the templates will inherit from the other one
  + Text

    Description automatically generated
  + A screenshot of a computer

    Description automatically generated
  + In this parent.html file, my\_block is the name of block. Other templates that extend from parent.html can fill in the code that goes within the my\_block block.
  + Text

    Description automatically generated
  + In this child.html file, the first line indicates that we are inheriting from “parent.html”. The 3rd to 5th lines indicates the content that will be passed into the my\_block block.
  + Now we need a view that renders this “my\_app/child.html” template as shown below
  + Text

    Description automatically generated
  + The return value is the following:
  + A picture containing logo

    Description automatically generated

**Passing Content to Templates**

* To pass this dummy data into our html templates, we pass a dictionary storing the content as the 3rd parameter of the render method as shown below
* Text

  Description automatically generated
* To access this content inside the template, we reference the keys within the dictionary passed to the render method
* Text

  Description automatically generated
* Notice that the dictionary we passed to the render method has a ‘users’ key whose value is a list. We access this list by using the ‘users’ key as shown within the for loop block.
* Notice that we can iterate over the “users” list, access the properties for each individual user and also perform logic with if statements.

**Configuring Templates**

* We’ve seen the folder structure where for each app, it has a template folder. This is the default way of making templates in django.
* Another folder structure is to have a template folder in the root folder as shown below:
* Graphical user interface, text

  Description automatically generated
* In order to tell django this new location for the templates folder, we have the change the settings.py file:
* Text

  Description automatically generated
* The ‘DIRS’ option is a list of directories where Django should look for projects. The items within the ‘DIRS’ array should be paths to where your templates folder is which is what the above example does by making use of the os.path.join method.
* Now, we render a target within a view as shown below:
* 
* Note that we don’t have to do ‘app\_name/index.html’

**Links in Templates**

* notice that within a template, we can link another template by making use of the url tag: "{% url 'path-name'%}"
* Graphical user interface, text

  Description automatically generated
* Suppose the views.route1 view renders the following template:
* Text

  Description automatically generated
* In the above example, we named the first path to be “route1” and the second path to be “route2”. If we go to the route that matches the first path, we trigger the route1 view and render the above template to produce this webpage output:
* A picture containing text

  Description automatically generated
* We can then click on the link to redirect us to route2. Notice the view that corresponds to the path with a name of “route2” doesn’t necessarily need to return a template, it could return a JSON response.

**Redirect**

* Django provides a built-in redirect() function to redirect users to a different URL within your application.
* To use redirect(), you need to import it from Django's shortcuts module: 
* Then, you can call the redirect() function with the URL you want to redirect to as an argument:
* Text

  Description automatically generated
* In the above example, if trigger the my\_view view, it’ll send back a response that redirects us to youtube.com
* You can also use named URL patterns in redirect().
* Note that redirect() returns an HttpResponseRedirect object, which is a subclass of HttpResponse. So, you can also set the status parameter to control the status code of the response:
* A picture containing text

  Description automatically generated
* In this case, my\_named\_url\_pattern is the name of a URL pattern defined in your urls.py file. As well, this will send a 301 (Moved Permanently) status code to the client.

**Flash Message**

* In Django, a flash message (or a message that is shown only once) is a way to show a message to the user on a redirect to another page.
* To add a flash message in Django, you can use Django's built-in messages framework by importing:
* 
* Ex:
  + Assume that “route1” is the path name for the url that triggers the “route1” view.
  + Text

    Description automatically generated
  + Suppose “my\_app/1.html” has the following chunk of code:
  + Text

    Description automatically generated
  + Now if we go to the url that triggers the “route2” view, messages.success() adds a success message to the current session, which will be displayed on the next request. The first argument to messages.success is the request object, and the second argument is the message text. Similarly, messages.error adds a new error message to the current session. Then we get redirected to the path with a name of “route1” which triggers the “route1” view (from assumption above) which renders the my\_app/1.html template. This template realizes that this session currently has a message so the {% if messages %} block is true. We then loop through each message to output the following on the webpage:
  + 

**Models**

* A model is a class that inherits from Django’s models class and a Django model is a table in your database.
* An instance of a model represents an entry within a table.
* A picture containing text

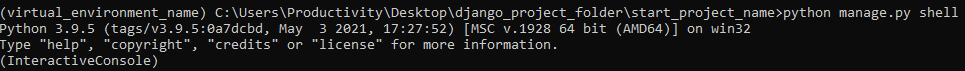
  Description automatically generated
* When we created the Django project, we got an empty SQLite database. It was created in the root folder, and has the filename db.sqlite3. By default, all Models created in the Django project will be created as tables in this database.
* To work with these tables, Django has its own database-abstraction API called an ORM. ORM stands for object relational mapper and allows us to access our database in an OOP way instead of having to use raw SQL. You can use different databases without changing your code to query the database, you just change the settings file.
* Creating Models
  + To create a model, create a Model class inside the models.py file in your app folder
  + Ex:
  + Text

    Description automatically generated
  + Note that Django inserts an id field for your tables, which is an auto increment number (first record gets the value 1, the second record 2 etc.), this is the default behavior of Django, you can override it by describing your own id field.

**Migrate and MakeMigrations**

* When we have described a Model in the models.py file, we must run two command to actually create the table in the database. (migrate and makemigrations)
* Migrate
  + First, we have to run the following command to generate new database migration files based on the changes made to your app’s models.
  + 
  + This will generate a new migration file in the migrations directory of your app folder.
  + Note that we sometimes have to run python manage.py makemigrations <my\_app>
* Makemigrations
  + Second, we have to run the following command so that Django will create and execute an SQL statement, based on the content of the new file in the /migrations/ folder made from the previous migrate command.
  + 
* Note that the makemigrations command only generates migration files based on changes to your models. It doesn't actually apply the changes to the database. To apply the changes, you need to run the migrate command.
* After running the above two commands, we have a table in our database that is described by the Model we made.
* As a side-note: you can view the SQL statement that were executed from the migration above. All you have to do is to run this command, with the migration number: 

**CRUD**

* the django python shell allows us to perform crud operations with models
* Shell Commands:
* to activate the shell, go to the command prompt and type: python manage.py shell
* 
* to use the built-in User models, import the Users via from django.contrib.auth.models import User
* 
* to see all the users in the database, type User.objects.all()
* 
* to return the first user in the database, type User.objects.first()
* 
* to return the last user in the database, type User.objects.last()
* 
* to search for a certain user, type User.objects.filter(user\_attribute = desired\_user\_ attribute)
* 
* to declare a desired user to be a variable, type user\_var = User.objects.filter(user\_ attribute = desired\_user\_ attribute).first()
* 
* to view the desired user, type user\_var
* 
* to view the attributes of the desired user, type user\_var.attributes() (notice the password is hashed)
* Text

  Description automatically generated
* to use the Post models, import it
* 
* to create a new post, type post\_var = Post(title = 'Blog1', content = 'First Post Content!', author = user1)
* 
* to save the new post, type post\_var.save()
* 
* to view all posts, type Post.objects.all()
* 
* notice that the user\_var.author() returns a User, and we can see the properties of that user such as their email
* Text

  Description automatically generated
* to delete a post, type post\_var.delete()
* Text

  Description automatically generated
* to quit the shell, type quit()
* 
* Note that these shell commands are just python scripts, we often include these crud scripts inside a view to handle crud operations with models.
* In Django, get\_object\_or\_404() is a useful shortcut function that tries to retrieve a single object from a model's database table. If the object does not exist, get\_object\_or\_404() raises a Http404 exception, which can be caught by Django's built-in error handling mechanism and rendered as a 404 error page.
* Ex:
  + Suppose you have a model called Article, and you want to retrieve an article with a specific ID.
  + Text

    Description automatically generated

**QuerySets**

* A QuerySet is built up as a list of objects from a database.
* QuerySets makes it easier to get the data you actually need, by allowing you to filter and order the data at an early stage.
* Suppose we have the following Table in our database:
* Table

  Description automatically generated
* To get all the entries, use the .all() method:
  + 
  + mydata looks like:
  + Text, letter

    Description automatically generated
* To return each object as a Python dictionary, with the names and values as key/value pairs, use the values() method:
  + 
  + mydata looks like:
  + Text, letter

    Description automatically generated
* To return only the columns that you specify, use the values\_list() method:
  + 
  + mydata looks like:
  + 
* To return specific entries, use the filter method:
  + 
  + AND
    - 
    - The above return records where lastname is "Refsnes" and id is 2
  + OR
    - 
    - The above return records where firstname is either "Emil" or Tobias"
    - Another common method is to import and use Q expressions:
    - 
    - 
    - In SQL, the above statement would be written like this:
    - 
  + LIKE
    - 
    - The above is the same as the following
    - 
* Order
  + 
  + The above example orders the result first by lastname ascending, then descending on id
  + In SQL, the above statement would be written like this:
  + 

**Admin**

* Django Admin is a CRUD\* user interface of all your models.
* To go to the admin page, to the “admin/”. You should see the following image.
* The main folder’s urlpatterns[] list takes requests with routes of “admin/” and sends them to admin.site.urls, this is why this URL goes to the Django admin log in page.
* Graphical user interface, application

  Description automatically generated
* To be able to log into the admin application, we need to create a superuser (aka an admin).
* This is done by typing this command in the command view: python manage.py createsuperuser
* Then, you must enter: username, e-mail address, (you can just pick a fake e-mail address), and password:
* Text

  Description automatically generated
* Then, make migrations if needed
* Now, if you go to admin panels and login with the credentials of the newly created superuser, you should see the following interface:
* Graphical user interface

  Description automatically generated
* To include a model within the admin panel, go to the admin folder inside the same app folder that the model was created.
* Import “admin” (should automatically be imported) and the model you want to add to the admin panel. Then have the line “admin.site.register(MyModel)”
* Ex:
  + Text

    Description automatically generated
* Now, if we go to the admin panel, we should see the model.
* Graphical user interface

  Description automatically generated with medium confidence
* Now if we click members, we see entries within the Members table.
* Graphical user interface, text, application

  Description automatically generated
* Note that we can change these records to have a more useful name by changing the \_\_str\_\_() method within the Model or by using the list\_details property
* \_\_str\_\_()
  + Ex:
  + Graphical user interface, text, application

    Description automatically generated
  + Output:
  + Graphical user interface, text, application

    Description automatically generated
* List\_details
  + We can control the fields to display by specifying them in in a list\_display property in the admin.py file.
  + Create a new class that inherits from admin.ModelAdmin and specify the list\_display tuple. Then add this new class as the second parameter to admin.site.register
  + Ex:
  + Graphical user interface, text, application

    Description automatically generated
  + Output:
  + Graphical user interface, text

    Description automatically generated
* Via this admin interface, we can perform CRUD operations which should be self-explanatory with the intuitive UI.

**Signals**

* Signals allow you to write certain functions that get executed automatically when certain actions occur in your Django application.
* For example, you can use signals to execute certain code when an object (a sender) is created, updated or deleted in the database, or when a user logs in or logs out of the application.
* Django signals are a way for decoupling different parts of your application.
* Ex:
* Suppose we have a Profile model. Whenever a User model instance is created/updated, a corresponding profile Model instance is created. To do so, we can use triggers. In this case, the User is the sender and the Profile is the receiver.
* Below is the profile model:
* Text

  Description automatically generated
* Signals should be written in their own file so within the app folder of the receiver model. Below is the signals.py file:
* Text

  Description automatically generated
* The post\_save.connect method allows us to execute a function whenever an instance of the sender model class is saved.
* We will make use of the post\_save.connect method. This method’s first parameter takes a method that takes in three arguments, the first being the “sender”, the second being the “instance”, the third being “created”. The “sender” is the model class that sends the signal. The “instance” is the actual instance of the sender that is being saved. “created” is true if the save created a new entry and false otherwise(for example a save caused by an update instead of creation). The post\_save.connect method takes in a second parameter which is the model class of the sender.
* In our example, we have a create\_profile method that we want to execute whenever a User is saved as indicated in the line “post\_save.connect(create\_profile, sender=User)”. This create\_profile method checks if the save was a result of a new User instance being created and if so, it prints out the create\_profile method’s parameter and saves a new corresponding Profile instance.
* We also need to configure the appys.py file inside the app folder and override/add the ready\_method as shown below:
* Text

  Description automatically generated
* When configuring the apps.py file, make sure we also include our app inside the settings.py file’s INSTALLED\_APPS as my\_app.apps.MyAppConfig and not the “my\_apps” shorthand.
* Now, whenever we create a new User we see that the create\_profile method is executed and prints out the following:
* Text

  Description automatically generated
* And we also see a new Profile entry within our database.
* Note instead of having this  line, we could use the reciever decorator from django.disptach instead.
* So now, our signals.py would look like:
* Text

  Description automatically generated

**Database Connection**

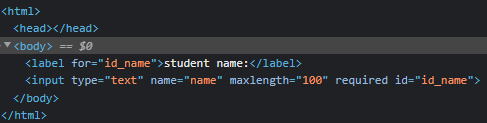
* The settings.py file contains all the project settings along with database connection details. By default, Django works with SQLite, database and allows configuring for other databases as well.
* Ex: connecting to MySQL
  + We need to provide all connection details in the settings file. The settings.py file of our project contains the following code for the database.
  + To connect with MySQL, django.db.backends.mysql driver is used
  + Graphical user interface, text, application

    Description automatically generated
* Connecting with MySQL db hosted on AWS: <https://youtu.be/v3jH1YxJqaY>
* Connecting with Postgres db hosted on AWS: <https://youtu.be/3HPq12w-dww>

**Default Forms**

* Creating Forms
  + We create in a forms.py file inside our app folder (might have to create forms.py yourself)
  + A form is a class that inherits from the built-in forms.Form class
  + Ex:
  + Below is the forms.py file
  + Graphical user interface, text

    Description automatically generated
  + Notice the class has a forms.CharField property (making use of the forms import)
* Passing forms to templates
  + We create an instance of the Form class and pass it to the template view as a context parameter
  + Text

    Description automatically generated
* Rendering forms in templates
  + Note that the above only renders the following html.
  + Assume the below is the “my\_app/create\_student.html” file that the above create\_student view renders.
  + 
  + Webpage output: 
  + Notice when we inspect the page, we see the following:
  + 
  + Notice the form doesn’t render a <form> tag. Instead, each property inside the CreateStudentForm class renders a <label> and <input> tag. Thus, this form only renders the fields of the form.
  + Ex:
    - Graphical user interface, text

      Description automatically generated with medium confidence
    - We can also render a form as\_p which wraps the fields within a p\_tag. There are more options as to what tag to wrap the form fields with, check the documentation.
  + Ex:
    - Text

      Description automatically generated
    - Note that we can access different properties within each field.
* Posting forms in templates
  + To post data via a template’s form, we need to make the template look like the following:
  + Text

    Description automatically generated
  + Notice we surround the form context with a <form> tag that creates a POST request to the url that has a name of ‘create\_student’ when the form is submitted.
  + Also notice that we pass in a csrf\_token.
  + CSRF Token
    - CSRF token (Cross-Site Request Forgery token) is a security mechanism used to prevent unauthorized access to resources in web applications. It is a randomly generated value that is included in a form or a URL parameter with each request made by a user to the server.
    - When a user submits a form, the CSRF token is sent along with the form data. The server checks the CSRF token and only processes the request if the token matches the one stored in the server's session. If the tokens do not match, the server rejects the request, as it may be an attempt to forge a request.
    - In Django, Cross-Site Request Forgery (CSRF) protection is built-in and enabled by default in the framework. Django uses a CSRF token to protect against CSRF attacks.
    - When a user visits a Django website, Django automatically creates a unique CSRF token for that user's session. This token is included in every HTML form that is rendered on the website. When the user submits the form, the CSRF token is included in the request data.
    - Django's middleware automatically checks the CSRF token in the submitted form data against the token in the user's session. If the tokens don't match, Django raises a CSRF verification failure exception, and the request is rejected.
    - To enable CSRF protection in Django, you need to make sure that the middleware is included in your MIDDLEWARE setting in your settings.py file:
    - Text

      Description automatically generated
    - By including the {% csrf\_token %} template tag in your form, Django automatically generates a hidden input field with the CSRF token value, which is included in the form data when the user submits the form.
* Receiving Data
  + Text

    Description automatically generated
  + Now if we submit the form, we trigger the create\_student which has the following output:
  + Text

    Description automatically generated
  + Notice that request.POST includes a csrftoken.
  + This create\_student view checks if the request method was post, then creates a ‘form’ variable based on the request data.
  + Then it checks if the ‘form’ is valid via the is\_valid() method which is used to validate data submitted through an HTML form.
  + If the ‘form’ is valid, then we make use of the cleaned\_data method is a dictionary-like object that is created after the form.is\_valid() method is called on a form instance. It contains the cleaned form data, which is the validated and sanitized data submitted by the user. Once the data has been cleaned and validated, it is stored in the cleaned\_data dictionary. We print out the ‘form\_data’ and notice it’s a dictionary.
  + Then, we create a student entry with the form data and return the following response:
  + 

**Model Form**

* Django ModelForm is a class that is used to directly convert a model into a Django form. If you’re building a database-driven app, chances are you’ll have forms that map closely to Django models. For example, a User Registration model and form would have the same quality and quantity of model fields and form fields. So instead of creating a redundant code to first create a form and then map it to the model in a view, we can directly use ModelForm. It takes as an argument the name of the model and converts it into a Django Form. Not only this, ModelForm offers a lot of methods and features which automate the entire process and help remove code redundancy.
* Suppose we have the following model
* Text

  Description automatically generated
* Creating a ModelForm
* Now we can make a ModelForm based off of the above Model. Below is the forms.py file:
* Text

  Description automatically generated
* The Meta class in a ModelForm is used to provide additional information about the form and its behavior. It's a nested class inside the ModelForm class, and it's defined using the keyword class Meta:
* Here are some of the most common options that can be specified in the Meta class:
  + model: The model that the form is based on. This is a required option.
  + fields: A list of fields that should be included in the form. If this option is not specified, all fields on the model will be included.
  + exclude: A list of fields that should be excluded from the form.
  + widgets: A dictionary of widget instances to use for rendering form fields.
  + labels: A dictionary of labels for form fields.
  + help\_texts: A dictionary of help texts for form fields.
  + error\_messages: A dictionary of error messages for form fields.
  + field\_classes: A dictionary of field classes to use for form fields.
  + localized\_fields: A list of fields that should be localized (i.e., formatted according to the current locale).
  + prefix: A string to use as a prefix for form field names.
  + instance: An instance of the model that the form is editing. If this option is specified, the form will be populated with data from the instance.
* Passing ModelForms to templates, Rendering ModelForms in templates, Posting forms in templates, receiving form POST data is the same as default Forms.

**Static Files**

* In Django, static files are the files that are served directly to the user, such as CSS, JavaScript, images, and other assets. Django provides a built-in way to manage static files that is easy to use and customizable.

**Configuring Static Files**

* STATIC\_URL
  + STATIC\_URL is simply the prefix or URL that is prepended to your static files and is used by the static method in Django templates mostly.
  + By default, the static\_url is set to /static/. This means that any static files in your project's STATICFILES\_DIRS will be served at URLs starting with /static/.
  + Note that STATIC\_URL is simply a url name, it is a not a path. There does not need to exist a directory with the same name as STATIC\_URL.
* STATICFILES\_DIRS
  + STATICFILES\_DIRS is a setting that specifies a list of directories where Django should look for static files.
  + By default, the STATICFILES\_DIRS setting is an empty list, which means that Django only looks for static files in each installed app's ‘static’ directory.
  + However, if you have static files that are shared between apps or that are located in a different directory, you can add those directories to STATICFILES\_DIRS to tell Django to search for static files there as well. For example, if you have static files in a directory named shared\_static in your project's root directory, you can add it to STATICFILES\_DIRS like this: STATICFILES\_DIRS = [os.path.join(BASE\_DIR, 'shared\_static')]
  + This is very similar to TEMPLATES.DIRS in the settings.py file.
* Ex:
* Create a new folder called ‘static’ within our root directory. Within this ‘static’ folder, create ‘js’, ‘styles’, and ‘images’ folders which will hold our js, css, and image files respectively. Note that these folder names could theoretically be anything.
* We can create folders within those ‘js’, ‘styles’, ‘images’ folders as needed. For example, within our ‘images’ folder, we could have a ‘profiles’ folder which contains profile pictures.
* Graphical user interface, text

  Description automatically generated
* Now, we need to tell django that our static files will be stored in the previously mentioned folders.
* To do so, go to settings.py and add the following STATIC\_URL and STATICFILES\_DIRS:
* Text

  Description automatically generated
* In this example, we added the root folder’s ‘static’ directory to the list of directories where django will look for static files. We also set the STATIC\_URL to ‘/assets/’.
* Note that if BASE\_DIR / ‘static’ doesn’t work, try os.path.join(BASE\_DIR, ‘static’) instead.
* Note that it is convention to set the value of STATIC\_URL to be ‘static’, but we will set it to ‘assets’ to emphasize that STATIC\_URL is simply a url name, it is a not a folder path as there will be no ‘assets’ directory in our project. Rather, STATIC\_URL is the name used to represent all the folder paths within STATICFILES\_DIRS.
* Following the previous bullet, this means that if you have a file called index.css within a ‘styles’ folder inside the static directory, its URL will be '/assets/styles.css' even though there is no ‘assets’ folder. But if we go to the <http://localhost:8000/assets/styles/index.css> url, we get the index.css file. We can think of the word ‘assets’ being replaced by each path within STATICFILES\_DIRS. Since the only path within STATICFILES\_DIRS is the root folder’s ‘static’ file, when we go to the <http://localhost:8000/assets/styles/index.css> url, we are looking at the project’s ‘static/styles/index.css’ file. Note that if we try to go to the <http://localhost:8000/static/styles/index.css> url, we get an error since it doesn’t match any of our urls. This is because ‘static/styles/index.css’ is not a url. Thus, <http://localhost:8000/assets/styles/index.css> is an url for the client to use while ‘static/styles/index.css’ is a folder path for the backend to use.

**Connecting CSS Files**

* Within our ‘static/styles’ folder, create a css file. Let’s call it index.css for example.
* Graphical user interface, text

  Description automatically generated
* The index.css could look like the following for example:
* Graphical user interface, application, Teams

  Description automatically generated
* Now, we can add the index.css to a template as shown below:
* A screenshot of a computer

  Description automatically generated with medium confidence
* The {% load static %} template tag is used to load the static file-related template tags in your template. These template tags allow you to easily reference static files in your HTML markup, such as CSS, JavaScript, and image files. To use the {% load static %} template tag, simply include it at the top of your template.
* The {% static %} template tag will generate a URL by making use of the STATIC\_URL setting.
* In the above exemplar template, we load the static file-related template tags at the top with the {% load static %} template.
* Then we link to the index.css file via {% static 'styles/index.css' %}. The keyword static within this template tag indicates we will use our static folder. The location of our static folder is determined by STATIC\_URL and found using STATICFILES\_DIRS. Since we set the STATIC\_URL to be ‘/assets/’, whatever path we have following the static keyword will be relative to this ‘assets’ path. Since we set STATICFILES\_DIRS = [BASE\_DIR / 'static'], whatever path we have following the static keyword will be searched for inside the root folder’s ‘static’ folder. The path that follows this static keyword is 'styles/index.css'. Thus, {% static 'styles/index.css' %} would generate this url: ‘/assets/styles/index.css’. Also, {% static 'styles/index.css' %} would look for 'styles/index.css' inside the root folder’s ‘static’ folder.
* Thus, when we render this template, we get the following webpage output:
* 

**Connecting Images**

* Similar to adding a css file, we will make use of the {% load static %} template tag and {% static %} template tag and pass in the path to our image.
* Ex:
* Suppose we have this ‘sponge.png’ inside the ‘static/images/profiles’ folder.
* Text

  Description automatically generated with medium confidence
* We can add this image as shown below
* A screenshot of a computer

  Description automatically generated with medium confidence
* Output:
* Shape

  Description automatically generated

**Uploading User-Images**

* Suppose we have the following model:
* Graphical user interface, text

  Description automatically generated
* When we save, we get this error:
* Text

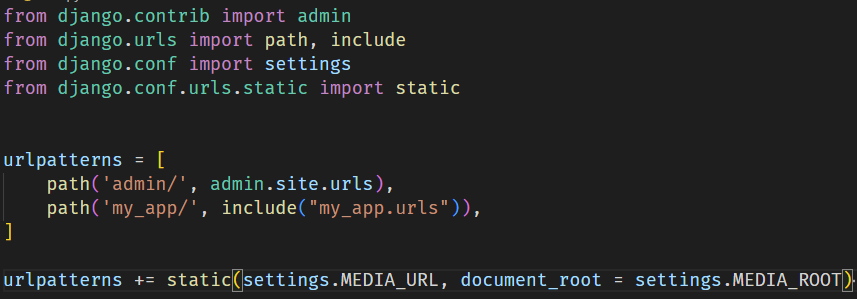
  Description automatically generated
* We get this image because the models.ImageField requires an external package called Pillow which is an image processing library.
* To install pillow, run in cmd: pip install Pillow
* Now, we should no longer see the above error.
* Now, we can upload images but we need to tell django where to store these user uploaded images.
* Suppose that we want to save them in root folder’s static/images folder.
* To do so, we need to set the value of MEDIA\_ROOT. MEDIA\_ROOT defines where user uploaded content is going to be stored. Note that if we don’t define the MEDIA\_ROOT, user uploaded content will be stored in the root folder.
* To change the value of MEDIA\_ROOT, go to settings.py and change the MEDIA\_ROOT as show below:
* 
* Now, suppose we create the below Profile Model instance.
* Graphical user interface, application

  Description automatically generated
* We see the py.png file inside the root folder’s static/images folder as shown below:
* Graphical user interface, application

  Description automatically generated
* Note that if we upload two files with the same name, django automatically adds a string to the end of the second file’s name as shown below:
* 
* Note that both files originally had a name of py.png
* Upload\_to
  + Note that the models.ImageField has an upload\_to arguement to specify where the uploaded files should be stored. By default, the upload\_to argument is relative to your project's MEDIA\_ROOT setting. Note that if the upload\_to path has a path that has not been created, it will be created when when an instance of the model with the ImageField has been created.
  + Graphical user interface, text

    Description automatically generated
  + In this example, the uploaded images will be stored in the MEDIA\_ROOT/myapp/mymodel/images/ directory.
* Note that the above example doesn’t follow the best practices as generally, we don’t want both static images and user-uploaded images to be stored together. Instead, we could create a new ‘media’ folder inside the root folder which will store all of our user-uploaded media files. But in production, django isn’t built to server user-uploaded files. Thus, most people actually disregard storing user-uploaded files in the project and use AWS instead.

**Rendering User-Uploaded-Images**

* Similar to how we set STATIC\_URL which specifies the URL prefixes for static files, we set a MEDIA\_URL.
* MEDIA\_URL specifies the URL prefix for media files, enabling user-uploaded-images to be accessed from the client.
* Note that MEDIA\_URL can be anything we want it to be as it is simply just a prefix name for the URL. BY default, the value of MEDIA\_URL is an empty string.
* Suppose we want to set the MEDIA\_URL to be ‘/images/’. To do so, go to the settings.py file and make the below changes:
* 
* Additionally, we have to go to our main folder’s urls.py and make the following changes:
* 
* “urlpatterns += static(settings.MEDIA\_URL, document\_root = settings.MEDIA\_ROOT)” is to tell django that when a URL with the MEDIA\_URL prefix is requested, Django will look for the corresponding file in MEDIA\_ROOT and serve it to the client.
* Here's what the code does:
  + The urlpatterns variable is a list of URL patterns for your Django project. The += operator adds a new URL pattern to the end of the list.
  + The static() function is a helper function provided by Django that generates a URL pattern for serving static files. It takes two arguments:
    - settings.MEDIA\_URL: This is the URL prefix for serving media files. For example, if MEDIA\_URL is set to '/media/', then any media files uploaded by users will be served from http://localhost:8000/media/.
    - settings.MEDIA\_ROOT: This is the absolute filesystem path to the directory containing your media files.
  + By adding the result of static() to urlpatterns, Django will now serve media files.
* Ex:
  + Now, we can go to <http://localhost:8000/images/py.png>.
  + By including the line “urlpatterns += static(settings.MEDIA\_URL, document\_root = settings.MEDIA\_ROOT)”, we told django that when a URL with the MEDIA\_URL prefix is requested, Django will look for the corresponding file in MEDIA\_ROOT and serve it to the client.
  + This URL has a prefix of ‘images’ which is the value of MEDIA\_URL, thus django will look for the corresponding file in MEDIA\_ROOT and serve it to the client.
  + The path of the file that we are requesting is ‘py.png’. Thus, we go to the MEDIA\_ROOT folder which is the ‘static/images’ folder inside the root folder and search for a py.png file. Now, we are able to get the py.png file, and return it to the client as shown below:
  + Icon

    Description automatically generated with medium confidence
* Note that serving static/media files should only be used in development and not in production, as it can be a security risk to serve static and media files from the same domain as your application. In production, it's recommended to use a separate content delivery network (CDN) or file storage service to serve static and media files.

**Rendering user-uploaded media files within templates:**

* Suppose we have the following index view:
* Text

  Description automatically generated
* Suppose we have the following body inside the index.html page:
* Text

  Description automatically generated
* Note that {{profile.pfp}} is the file path (for the backend) while {{profile.pfp.url}} is the url (for the frontend).
* Now, if we trigger the view, we render the template and get:
* Graphical user interface, text, application

  Description automatically generated
* Note that the above template renders only if all the profiles have a pfp image. If a profile doesn’t have a pfp image, we can’t access it’s url as that an error.
* One solution is to have a default profile picture so all profile have a pfp image.
* Another solution is instead of calling pfp.url, we can create a pfpURL method within our Profile model. This pfpURL method will return the pfp.url if the Profile instance has a pfp, and ‘’ otherwise.
* Thus the modifications will be shown below in the template and model:
* Text

  Description automatically generated
* Text

  Description automatically generated  
  This @property is a decorator that allows you to define a method that behaves like an attribute. This means that you can use the method as if it were a regular attribute, even though it is actually a method that computes and returns a value. This just allows us to call profile.pfpURL instead of profile.pfpURL()
* Now if we trigger the view to render the template, we get the following:
* Logo, company name

  Description automatically generated
* Notice how the last profile instance in the list doesn’t have an pfp.url but it doesn’t cause an error since we use the pfpURL property method instead of pfp.url

**Submitting Image Files with Forms**

* When submitting image files with forms, we have to include the following ‘enctype’ attribute to the form tag as shown below:
* Text

  Description automatically generated
* By setting ‘enctype’ to multipart/form-data, we can send images, pdfs, or other files with this form. This is because enctype="multipart/form-data" tells the browser to encode the form data in a way that can handle binary files.
* Now, within the view, we have access to request.FILES since in the template, we said that form data would include files.
* request.FILES is a dictionary-like object that contains files uploaded by a client in the request. The keys in the dictionary are the names of the file input fields in the HTML form
* Text

  Description automatically generated
* In this example, the view checks whether the request method is POST, which indicates that the form has been submitted. It then retrieves the uploaded file from the request.FILES dictionary using the name of the file input field (myfile in this case). Finally, it does something with the file (in this case, it just renders a template with information about the uploaded file).
* Note that we can also pass request.FILES into a ModelForm as shown in the example below:
* Text

  Description automatically generated

**Static Files In Production**

* If we plan to deploy our project, we should set DEBUG=False but Django has no built-in solution for serving static files in production.
* Note that when using DEBUG = False you have to specify which host name(s) are allowed to host your work. You could choose '127.0.0.1' or 'localhost' which both represents the address of your local machine. We could choose '\*', which means any address are allowed to host this site. This should be change into a real domain name when you deploy your project to a public server.
* Notice if we go to our templates, or even the admin panel, there is no styling.
* Since Django by itself cannot serve static/media files in production, you will have to upload these files to a cloud storage such as AWS or use a third-party library to handle static files.
* WhiteNoise
  + One third-party library is WhiteNoise, which is a Python library, built for serving static files
  + To install WhiteNoise in your virtual environment, type the command below:
  + 
  + To make Django aware of you wanting to run WhitNoise, you have to specify it in the MIDDLEWARE list in settings.py file:
  + Text

    Description automatically generated
* In addition to using whitenoise, static files have to be collected and put in a specified folder before we can use it.
* STATIC\_ROOT
  + First, we need to specify where our static files will be located when DEBUG=False.
  + STATIC\_ROOT determines the path for serving static files in production. When debug is false, django no longer looks into the paths inside STATICFILES\_DIRS to get our static files. Instead, when debug is false, django will look into STATIC\_ROOT for the static files.
  + Thus we need to set the value of STATIC\_ROOT in the settings.py file as shown below:
  + 
  + In the above example, we tell django to look for a ‘staticfiles’ folder inside our root folder for our static files when DEBUG=False.
  + Note that the STATIC\_ROOT folder doesn’t have to exist yet.
* collectstatic
* After setting STATIC\_ROOT we have to tell django to find all the static files in our project and bundle/collect them into the STATIC\_ROOT folder.
* To do so, we run this command: 
* When we run the above command, django will search for all static files in the directories specified in STATICFILES\_DIRS and copy them to the directory specified in STATIC\_ROOT.
* When you run the above command, we may get many files because of the admin user interface, that comes built-in with Django. We want to keep this feature in production, and it comes with a whole bunch of files including stylesheets, fonts, images, and JavaScripts.
* Note that any changes we make to the static files within STATICFILES\_DIRS are not automatically reflected in the new STATIC\_ROOT folder. Thus, anytime we make changes to the static files within STATICFILES\_DIRS, we need to rerun: py.manage.py collectstatic
* Now, we have to connect STATIC\_URL with STATIC\_ROOT.
* urlpatterns += static(settings.STATIC\_URL, document\_root = settings.STATIC\_ROOT)
  + In our main folder’s urls.py file, we have to add the following:
  + 
  + “urlpatterns += static(settings.STATIC\_URL, document\_root = settings.STATIC\_ROOT)” is to tell django that when a URL with the STATIC\_URL prefix is requested, Django will look for the corresponding file in STATIC\_ROOT and serve it to the client. By including this line, django will now serve static files.
  + If we don’t include the above line, it's also possible to serve static files without using the static() function by relying on Django's built-in runserver command. When you run the runserver command, Django automatically sets up a view to serve static files for you. This view is only intended for use during development and is not suitable for production use. So if you're able to serve static files without explicitly adding the static() function to your urlpatterns, it's likely that you're running the runserver command and Django is automatically serving the static files for you. It's important to note that relying on Django's built-in runserver command to serve static files is not recommended for production use. In production, you should use a dedicated web server like Nginx or Apache, or a Content Delivery Network (CDN) to serve static files.
* Now, even though DEBUG=False, because we have used a third-party library, we can see the styling in our templates and admin page.
* However, our media/user-uploaded files do not appear.
* While it is possible to configure a web server like Apache or Nginx to serve media files, Django discourages to serve media files on production from the server. Use cloud services like amazon s3 to server your media file.

**Class Based Views**

* In Django, class-based views (CBVs) are an alternative to function-based views (FBVs).
* There was a point where django only had function based views. There were methods added to take care of common actions but extending functions can be very limiting so to fix this problem django added class based views.
* By using classes we are able to extend our code by utilizing the following:
  + Inheritance: so we can write reusable code and make our application more DRY. (Don’t Repeat Yourself)
  + Built in methods and views to eliminate redundancy for common use cases
  + Separation of code by http method types such as GET and POST.
* Just because CBV have the above advantages, that doesn’t mean you should only use CBV as it can make a view more complicated for no reason. You can use both CBV and FBV in your application, it depends on the use case.
* FBV vs CBV
  + Ex:
  + Let’s take a look at a view that accomplishes the same tasks written as a function and then a class.
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + Because we are using a class based view, we need to add the “as\_view” method for our url resolver. This is because the django url resolver cannot process a class but instead needs a function. To resolve this, we trigger the “as view” method from our “View” class which we inherited from and the “as view” method will call the correct view function depending on the method sent, therefore giving the url resolver a function.
  + As you can see in this example the class based view is much simpler to write and takes care of the magic under the hood, which we cannot see at the moment. This is a very simplistic example but it already demonstrates some powers of the class based view.
  + In this case we inherited from a built in view called ListView which at a minimum just needed a model name or a queryset. The view already knows which template to render, and how to pass in the data into the template. With a function based view we have to take care of each step by first querying the database, passing in our context data and then rendering our template.
* Separating code by http methods
  + In this next example I want to demonstrate how class based views take care of http methods by passing in a post request onto the same view.
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + If you take a look at the two methods you can see that the function based view takes in every http method type, this means if we are sending “post” data then we have to check the request type ourselves to process it correctly.
  + With class based views we have a function for different methods. So if we send a GET request, then the “get” function gets triggered and any logic we have in there gets processed. If we send a “post” request then that request is sent to the “post” function for processing. This makes for much cleaner code by separating our request.
* The “View” Class
  + While django provides us with a number of built in class based views to work with, at the core of all these views in one main view called “View”. This is a class that all other views will inherit from and provides us with the core functionality to make a django class based view.
  + Here we will create a simple view that queries all the products from our database and renders a template.
  + Text

    Description automatically generated
  + Now in this case all we need is a simple view that returns back a template with a queryset of data. Thankfully django already knows that this is a common task so we don’t have to write all the logic ourselves. This is a perfect example of where we may want to use the built in “ListView” turning the above view into this:
  + Text

    Description automatically generated
* CRUD
  + Let’s see what a collection of views would look like for a simple application with CRUD functionality.
  + What I want to do here is give you an example of what a simple to-do application would look like if we utilized the full power of class based views. We will create views to perform the following actions:
    - List out our to do items
    - View a single item
    - Create a new item
    - Update an existing item
    - Delete an item
  + Here’s what each view should look like in our application.
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + The above built in views have been created by the django team to make these types of operations easier for us. Let’s take a look under the hood and see how a view is created and what other attributes and methods it may have.
* How these views are constructed
  + To understand how django class based views work we need to understand how they are constructed. Django class based views are typically a collection of other views and mixins. Let’s take a look at the 5 built in views from our sample application and see how they are made along with how we can customize certain default settings.
  + Mixins are simply classes that contain methods to be used by other classes. We use mixins when we want to when we want to add a particular feature that will be made available in multiple child objects. They should be use to take care of one specific task and should not be extended. When using mixins, order of inheritance matters.
* ListView
  + Text

    Description automatically generated
  + The ListView inherits directly from 1 view and 1 mixin, but all together there are a total of 2 views and 4 mixins that give this view all the attributes and methods that it has.
    - MultipleObjectTemplateResponseMixin
    - TemplateResponseMixin
    - BaseListView
    - MultipleObjectMixin
    - ContextMixin
    - View
  + Each parent view and mixin adds or enhances a set of attributes and methods for our built in view.
  + A picture containing graphical user interface

    Description automatically generated
  + Text

    Description automatically generated
  + template\_name: By default the ListView Looks for a template with the prefix of the model name (task) and the suffix of \_list.html if not otherwise set (task\_list.html). This can be overridden by setting the “template\_name” attribute.
  + context\_object\_name: Override the default queryset name of “object\_list” by setting the “context\_object\_name” attribute. It helps to have a more user friendly name to work with besides just “object\_list”.
  + paginate\_by & ordering: The list view also has pagination and ordering already built in. We can set these methods by setting their attributes like I did in the code example above.
* DetailView
  + Text

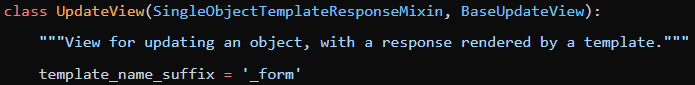
    Description automatically generated
  + The DetailView inherits directly from 1 view and 1 mixin, but all together there are a total of 2 views and 4 mixins that give this view all the attributes and methods that it has.
    - SingleObjectTemplateResponseMixin
    - TemplateResponseMixin
    - BaseDetailView
    - SingleObjectMixin
    - ContextMixin
    - View
  + Graphical user interface

    Description automatically generated
  + Text

    Description automatically generated
  + template\_name = By default the DetailView looks for a template with the prefix of the model name (task) and the suffix of \_detail.html if not otherwise set (task\_detail.html).
  + context\_object\_name: Override the default model name of “object” by setting the “context\_object\_name” attribute.
  + pk\_url\_kwarg: By default the view looks for a primary key in the url called “pk” or a slug field called “slug”. Both of these values can be customized with slug\_field or “pk\_url\_kwarg”
* Create View
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + The CreateView inherits directly from 1 view and 1 mixin, but all together there are a total of 3 views and 6 mixins that give this view all the attributes and methods that it has.
    - SingleObjectTemplateResponseMixin
    - TemplateResponseMixin
    - BaseCreateView
    - ModelFormMixin
    - FormMixin
    - SingleObjectMixin
    - ContextMixin
    - ProcessFormView
    - View
  + Timeline

    Description automatically generated
  + Text

    Description automatically generated
  + template\_name: By default CreateView looks for a template with the prefix of the model name (task) and the suffix of \_form.html if not otherwise set (task\_form.html). This can be overridden by setting the “template\_name” attribute.
  + success\_url: specifies the URL to redirect to after a successful form submission. By default, it redirects to the URL specified in the get\_absolute\_url() method of the created object.
  + form\_class: By default this view creates a model form for us based on the model we specify. As you can see in the example if we don’t specify a model form to use, one will be created by fail for us after we specify the field names. We can use our own model form by creating a mode for and setting the “form\_class attribute”.
* UpdateView
  + 
  + This view by default follows the same template\_naming convention and form principles as the create view. Both views will look for the same form and template unless otherwise specified.
    - SingleObjectTemplateResponseMixin
    - TemplateResponseMixin
    - BaseUpdateView
    - ModelFormMixin
    - FormMixin
    - SingleObjectMixin
    - ContextMixin
    - ProcessFormView
    - View
  + Timeline

    Description automatically generated
  + The template\_name\_suffix attribute of the UpdateView specifies the suffix to use for the template names. By default, the template names are derived from the model name and the view type, and the suffix is \_form.
  + For example, if you have a model called MyModel and you're using an UpdateView to edit instances of that model, the default template name would be myapp/mymodel\_form.html.
  + You can override the default template name by specifying your own template name or template path using the template\_name attribute of the view. For example:
  + Text

    Description automatically generated
  + In the example above, we're using a custom template called my\_custom\_template.html instead of the default template name myapp/mymodel\_form.html. The template\_name\_suffix attribute is not used in this case.
* DeleteView
  + Text

    Description automatically generated
  + The DeleteView inherits directly from 1 view and 1 mixin, but all together there are a total of 3 views and 5 mixins that give this view all the attributes and methods that it has.
    - SingleObjectTemplateResponseMixin
    - TemplateResponseMixin
    - BaseDeleteView
    - DeletionMixin
    - BaseDetailView
    - SingleObjectMixin
    - ContextMixin
    - View
  + A picture containing timeline

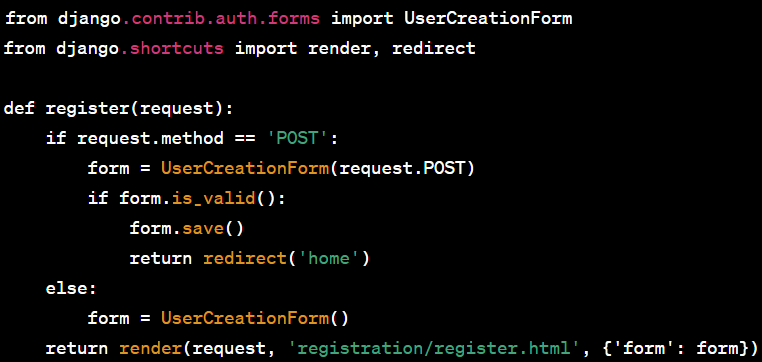
    Description automatically generated
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + template\_name: By default DeleteView look’s for a template with the prefix of the model name (task) and the suffix of \_confirm\_delete.html if not otherwise set (task\_confirm\_delete.html). This can be overridden by setting the “template\_name” attribute.
  + pk\_url\_kwarg: By default the view looks for a primary key in the url called “pk” or a slug field called “slug”. Both of these values can be customized with slug\_field or “pk\_url\_kwarg”
* Django Built in views
  + Django provides use with many built in views we can use and customize. These views are separated into the following categories:
    - Generic Base Views
    - Generic Display Views
    - Generic Editing Views
    - Generic date views
    - Auth Views
  + Listed out below are all the django built in views inside of their category:
    - Auth Views
      * LoginView
      * LogoutView
      * PasswordChangeDoneView
      * PasswordChangeView
      * PasswordResetCompleteView
      * PasswordResetConfirmView
      * PasswordResetDoneView
      * PasswordResetView
    - Generic Base
      * RedirectView
      * TemplateView
      * View
    - Generic List
      * ListView
    - Generic Detail
      * DetailView
    - Generic Edit
      * CreateView
      * DeleteView
      * FormView
      * UpdateView
    - Generic Dates
      * ArchiveIndexView
      * DateDetailView
      * DayArchiveView
      * MonthArchiveView
      * TodayArchiveView
      * WeekArchiveView
      * YearArchiveView

**Authentication**

* By default, django has session based authentication.
* When we register a user, their username and password are saved in the database.
* When a user logins, the credentials are sent to the backend and Django checks the user's credentials against those stored in the database. If the credentials match, Django creates a session token for the user. This session is stored in both the backend and in the browser(search the browser’s dev tools cookies). When we navigate through different pages, that session token will be sent with each request to our backend so we will stay authenticated.
* When a user logs out, Django deletes their session token from the backend and logs them out.

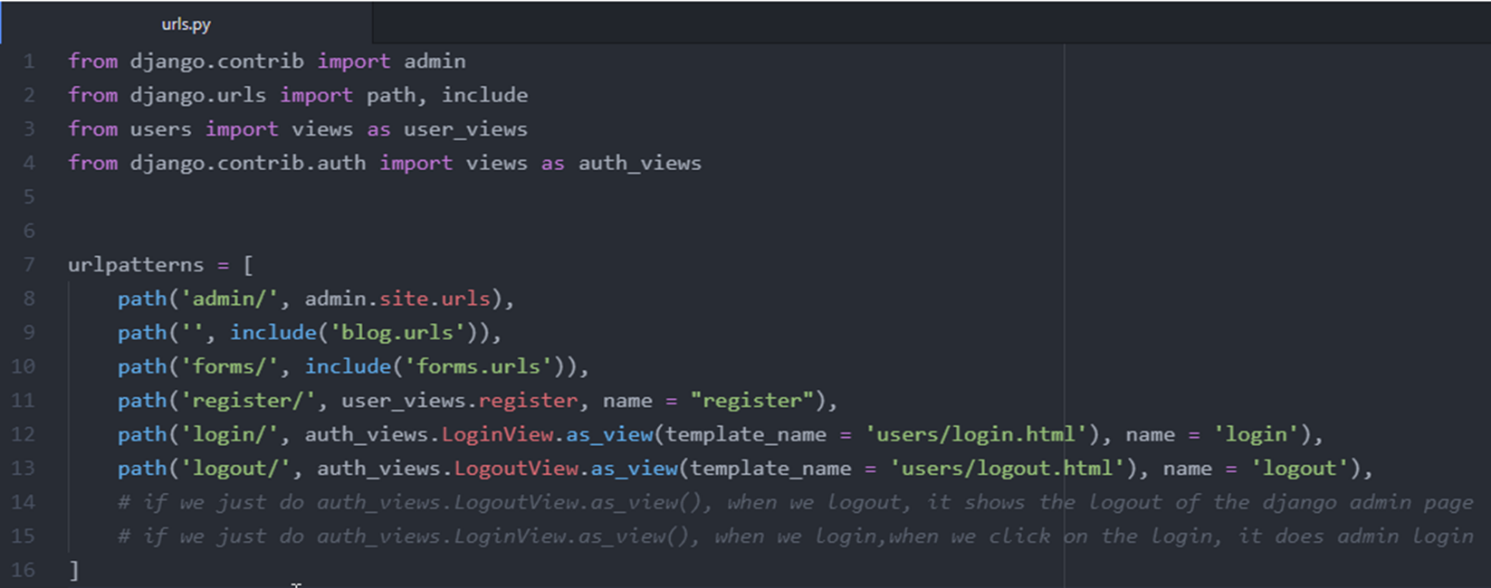
**User Registration**

* This ‘register’ view handles user registration.
* 
* In this example, the register() function handles both GET and POST requests. When a GET request is received, a new instance of UserCreationForm is created and passed to the registration template. When a POST request is received, the form data is validated and if valid, a new user is created and the user is redirected to the homepage.
* Rending this UserCreationForm in the template is the same as rendering any other form in a template. Thus, the ‘registration/register.html’ template could look like:
* Text

  Description automatically generated

**Login and Logout**

* Text

  Description automatically generated
* In the code above, the login\_view function handles user authentication. It checks if the request is a POST request and attempts to authenticate the user using the provided username and password via the authenticate method. This method returns a User object if the authentication is successful, and None if it fails. If the authentication is successful, the function calls the login function to log the user in. This logs in the user and creates a session for them, which enables them to access protected resources or perform certain actions that require authentication. Then this view redirect them to the home page. If the authentication fails, an error message is displayed.
* The logout\_view function logs the user out and redirects them to the login page.
* Class Based Views
  + We can use CBVs to implement login/logout
  + 
  + Then within ‘users/login.html’, we can access the login form as via the ‘form’ key passed to the template as a context (automatically done by the CBV under the hood)
  + Then within settings.py, we set the value of LOGIN\_REDIRECT\_URL to the name of the path that the user should be redirected to upon login.
  + 
  + By default, LoginView uses the django.contrib.auth.forms.AuthenticationForm form to handle user authentication, but you can also customize the form by subclassing AuthenticationForm or setting the form\_class attribute on LoginView.
  + After a successful login, LoginView will redirect the user to the URL specified by the success\_url attribute, or to the next query parameter if it is present in the login form submission.
  + LoginView also provides some other methods and attributes that you can use to customize its behavior, such as redirect\_authenticated\_user (which controls whether authenticated users are redirected to success\_url or a different URL) and authentication\_form (which allows you to use a custom form class for user authentication).

**Restricting Pages With is\_authenticated**

* In Django, you can use template tags to check if a user is authenticated or not, and display content based on their authentication status. The two most common template tags used for this purpose are {% if user.is\_authenticated %} and {% if user.is\_anonymous %} (they are opposites of each other).
* The {% if user.is\_authenticated %} template tag is true if the user is authenticated.
* The {% if user.is\_anonymous %} template tag is true user is not authenticated.
* Ex:
  + Text

    Description automatically generated
  + In this example, If the user is authenticated, a welcome message is displayed with their username. If the user is not authenticated, a message is displayed asking them to log in.
* Ex:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + In this example, the message is only displayed if the user is not authenticated. If the user is authenticated, this message is not displayed.
* Note that we can also use user.is\_authenticated within views. For example, suppose we never want an authenticated user to see the sign up page. We can start the view with something like the below:
* Text

  Description automatically generated

**Restricted Pages With @login\_required and Authorization**

* In Django, you can restrict access to certain pages based on the user's authentication status using the login\_required decorator.
* The login\_required decorator can be applied to a view function to restrict access to authenticated users only. If an unauthenticated user tries to access the view, they will be redirected to the login page.
* To use the login\_required, decorator, we must import it via:
* Text

  Description automatically generated
* Now, let’s suppose we have a Rooms model. Each room belongs to a user, so only that user should be authorized to modify it.
* Ex:
* Text

  Description automatically generated
* This deleteRoom view firsts checks if the user is authenticated since we included the login\_required decorator. If the user is not logged in, they are sent to the url path with a name of ‘login’. If they are logged in, then the function is executed. This function gets the room specified by the pk. Then it checks if that room does not belong to the user who made the request via the line “request.user != room.host”. If the room does not belong to the user, a response error is sent. Otherwise, the room is deleted.
* Note that we if we don’t pass login\_url argument for @login\_required, we can still specify the login\_url in the settings.py file as shown below:
* 
* Additionally, suppose we have a template that renders all the rooms in our database. However, we want to render “edit” and “delete” buttons for rooms that belong to the current user. Then we would have something like the following:
* Text

  Description automatically generated
* Notice the use of {% if request.user == room.host %}

**Authorization with @user\_passes\_test**

* The user\_passes\_test decorator is a function provided by Django's contrib.auth.decorators module that can be used to restrict access to views based on a custom test function.
* The decorator takes a single argument, which is the test function. This function should take a user object as its argument and return True if the user passes the test and should be granted access, or False if the user fails the test and should be denied access.
* This decorator also takes in a second optional parameter which is the url the user gets redirected to if they fail the test. By default, the user will be redirected to the login page upon test fail.
* Ex:
  + Text

    Description automatically generated
* In this example, the my\_view function is decorated with user\_passes\_test, and the test function is defined as a lambda that checks if the user is a superuser. If the user is a superuser, the view code will be executed. If not, they will be redirect to ‘/login/’.

**Updating User**

* In forms.py have the following:
* Text

  Description automatically generated
* In our specific example, we only allow the user to update their username as that is the only item in the ‘fields’ array.
* Note that if we set fields = ‘\_\_all\_\_’, it’ll display all the user fields like email address, staff status, active, etc.
* Text

  Description automatically generated
* login\_required decorator is used to ensure that the user is logged in before accessing the view.
* update\_user\_profile function handles both GET and POST requests.
* On GET request, it creates a UserProfileForm instance with the current user's data and renders the user\_profile\_update.html template with the form.
* On POST request, it validates the submitted form and updates the user's data. If the form is valid, it saves the form and redirects the user to the ‘index’ page with a success message. If the form is invalid, it renders the user\_profile\_update.html template with the form and validation errors.
* Text

  Description automatically generated
* The above is the template that simply renders the form passed to the template via context. This form makes a post request upon submission.
* The webpage output will look like:
* 
* Note that instead of using a ModelForm for the User, we can use the following form which does basically the same thing:
* Text

  Description automatically generated

**Custom User Model**

* Django has a built-in User model with pre-set fields. There are different ways to customize this User model.
* One solution would be to create a one to one relationship between a User and Profile model.
* Another solution (the solution that these notes discuss) is to modify the User model.
* Django allows you to define a custom user model that can be used to replace the default user model provided by Django. Defining a custom user model gives you the ability to add or remove fields from the user model, change the way authentication works, and generally tailor the user model to suit your needs.
* Creating a custom user model:
  + In your app's models.py file, create a new class that inherits from AbstractBaseUser and PermissionsMixin. These are built-in Django classes that provide basic user functionality and permissions.
  + Define the fields for your custom user model. You can add any fields you like, but at a minimum you should include an email field, as this will be used as the primary identifier for your users.
  + Define the USERNAME\_FIELD and REQUIRED\_FIELDS attributes for your custom user model. The USERNAME\_FIELD specifies which field to use as the primary identifier for your users (in this case, the email field). This USERNAME\_FIELD will be used in the authentication system such as ‘authenticate()’ and ‘login’.
  + The REQUIRED\_FIELDS specifies which fields are required when creating a user.
  + Text

    Description automatically generated
  + Note in the above, we must set the ‘email’ field to be unique as email will be used for authentication so two users cannot have the same email.
  + Register your custom user model with Django by adding the following code to your app's apps.py file:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Finally, update the AUTH\_USER\_MODEL setting in your project's settings.py file to point to your custom user model:
  + 
  + This above setting tells django to use our custom User model instead of the built-in user model.
  + Note that custom\_auth is the name of our app.
  + To see this custom User model in the admin panel, register it as shown below:  
    Text

    Description automatically generated

**Pagination**

* Pagination is a technique used to split large sets of data into smaller, more manageable chunks, typically for display purposes. In Django, pagination is implemented using the built-in Paginator class, which allows you to split a queryset into pages and display them using a variety of methods.
* To use pagination in Django, you'll first need to import the Paginator class from the django.core.paginator module:
* 
* Next, you'll need to create a Paginator object, passing in the queryset you want to paginate and the number of items you want to display per page:
* 
* 
* In this example, we set a max of 2 posts per page
* to return the total number of pages, type: p.num\_pages. posts 1 and 2 are on page 1, posts 3 and 4 are on page 2, and post 5 is on page 3, totalling to 3 pages
* 
* We can loop through the pages:
* Text

  Description automatically generated
* to get the first page in a page variable, type: p1 = p.page(1)
* 
* return the page variable with: p1
* 
* return the page variable’s number with: p1.number
* 
* return the post in the page variable with: p1.object\_list
* 
* return if the current page has a previous page with: p1.has\_previous()
* 
* return if the current page has a next page with: p1.has\_next()
* 
* return the page number of the next page with: p1.next\_page\_number()
* 

**Django REST Framework (DRF)**

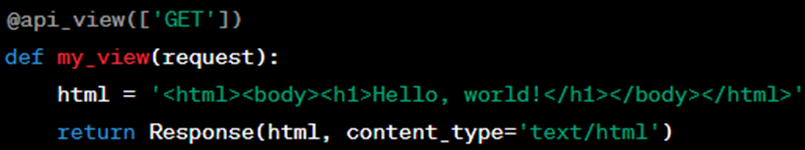
* Django REST Framework is a powerful and flexible toolkit for building Web APIs. It is built on top of the Django web framework and provides a set of reusable tools and patterns for creating RESTful APIs.
* Django REST Framework provides support for common HTTP methods such as GET, POST, PUT, PATCH, and DELETE, as well as support for content negotiation, authentication, and permissions. It also provides serialization and deserialization of data in various formats, including JSON and XML.

**Installing DRF**

* Install django
* Then install DRF via: pip install djangorestframework
* Then go to settings.py and add ‘rest\_framework’ to installed apps
* Text

  Description automatically generated

**Response**

* The Response object is used to create an HTTP response that will be returned to the client.
* The Response object is imported from: rest\_framework.response
* Response object will automatically serialize a dictionary as JSON and set the Content-Type header of the response to application/json.
* We can also pass an HTML string to Response constructor, along with a content\_type argument specifying that the content is HTML.
* An example of this is shown below:
* 

**API Views as FBV**

* In Django, the @api\_view decorator is used in combination with function-based views to create web APIs using Django REST Framework. The @api\_view decorator is used to define the allowed HTTP methods for a view function.
* Ex:
* Text

  Description automatically generated
* In this example, we import the api\_view decorator from Django REST Framework and apply it to the my\_view function, which takes a request argument and returns a Response object. We specify that the my\_view function is only allowed to handle GET requests by passing the ['GET'] argument to the @api\_view decorator.

**API Views as CBV**

* We can also write our API views using class-based views
* APIView
  + APIView is a base class for all other class-based views in Django REST Framework and is designed to be subclassed to provide specific behavior for your API endpoints.
  + Text

    Description automatically generated
  + In this example, we define a BookList API view that subclasses APIView. The get method returns a list of all the books in the database, serialized using the BookSerializer class. The post method creates a new book from the data provided in the request, if the data is valid, and returns the serialized book data with a status of 201 (Created). If the data is invalid, it returns an error message with a status of 400 (Bad Request).
  + To use this view, you need to wire it up to a URL by defining a URL pattern that points to the BookList view. Here's an example of how to do this in urls.py:
  + Text

    Description automatically generated
* Mixins
  + Mixins are a way to add specific behavior to a class-based view by defining reusable chunks of functionality that can be mixed in with other views.
  + Mixins can be combined to create more complex views that provide a mix of functionality. For example, you could combine the ListAPIView and CreateAPIView mixins to create a view that provides both list and create functionality in a single endpoint.
  + Ex:
  + Text

    Description automatically generated
  + In this example, we define two views, BookList and BookDetail, which use different combinations of mixins to provide different functionality. The BookList view uses the ListCreateAPIView mixin, which combines the ListAPIView and CreateAPIView mixins to provide both list and create functionality in a single endpoint. The BookDetail view uses the RetrieveUpdateDestroyAPIView mixin, which combines the RetrieveAPIView, UpdateAPIView, and DestroyAPIView mixins to provide read, update, and delete functionality for a single object.

**Serializers**

* In Django REST Framework, a serializer is a component that is used to convert complex data types, such as Django model instances or Python data structures, into a format that can be easily rendered into JSON or XML, and vice versa.
* Serializers also handle deserialization, allowing parsed data to be converted back into complex types.
* A serializer field defines how a particular attribute of an object should be serialized and deserialized. Serializer fields allow you to control the format, validation, and representation of the data that is passed through the serializer.
* Serializers should be written in their own serializers.py file. These serializers can be placed inside your app’s folder.
* Django REST Framework provides two main types of serializers: ModelSerializer and Serializer.
* ModelSerializer
  + ModelSerializer is a shortcut for creating serializers that deal with model instances and provide default implementations for common operations such as create, update, and deletion. It automatically generates fields based on the model fields and can handle relationships between models.
  + With ModelSerializer, you can quickly create a serializer that maps closely to a model and includes all of its fields, with minimal code.
  + Here is an example of a ModelSerializer for a simple Book model:
  + Text

    Description automatically generated
  + In this example, we create a BookSerializer class that inherits from serializers.ModelSerializer. We specify the Book model as the model attribute of the Meta class and list the fields we want to include in the serialized representation of the model. In this case, we include the id, title, author, and published\_date fields.
  + Note that if we want to serializer all the fields, we set ‘fields’ equal to ‘\_\_all\_\_’.
* Serializer
  + Serializer is a more generic serializer that does not rely on model instances. It is used to serialize and deserialize arbitrary Python objects, such as dictionaries or lists. With Serializer, you define the fields and their types explicitly.
  + Ex:
  + Text

    Description automatically generated
  + In this example, we define a DictionarySerializer class that inherits from serializers.Serializer. We define two fields, key and value, and specify their types as CharField. When we serialize a dictionary using this serializer, it will produce a serialized representation with key and value fields for each key-value pair in the dictionary.
  + Serializer fields also support many options for customizing their behavior, such as required, allow\_null, min\_value, max\_value, min\_length, max\_length, and more. You can use these options to control how the data is validated and processed, and to enforce constraints on the data that is being serialized or deserialized.

**Nested Serializers**

* Nested serializers are used to serialize and deserialize data that involves relationships between models. Nested serializers allow you to include related models as nested representations within a serializer.
* Here's an example of how to use nested serializers:
* A screenshot of a computer

  Description automatically generated with medium confidence
* In this example, we define a BookSerializer and an AuthorSerializer for the Book and Author models, respectively. The BookSerializer includes the id, title, and author fields, while the AuthorSerializer includes the id, name, and books fields.
* The books field in the AuthorSerializer is a nested serializer that uses the BookSerializer to serialize a list of books associated with an author. many=True is used to specify that there can be multiple books associated with an author. read\_only=True is used to ensure that the nested serializer is only used for read operations and not for creating or updating books.
* To use these serializers, you can retrieve an author and their associated books from the database and serialize them as follows:
* A screenshot of a computer

  Description automatically generated with low confidence
* This will serialize the author and their associated books as a nested representation. The resulting serialized\_data will contain the author's id, name, and a list of books associated with the author, where each book has its own id and title.
* Nested serializers allow you to easily handle relationships between models in your APIs and return complex, nested data structures in a consistent and efficient manner.

**Using Serializers in Views**

* Serializers can be used in views to perform serialization and deserialization of data.
* Ex:
* Text

  Description automatically generated
* In Django REST Framework, many=True is a parameter that can be passed to a serializer when serializing multiple objects. It is used to specify that the serializer should be used in a list context to serialize multiple objects instead of a single object context. By default, a serializer expects to work with a single object and many is set to False. Using many=True is particularly useful when you want to serialize a queryset or a list of objects that belong to the same model. It allows you to serialize the objects in a consistent and efficient manner without having to repeat the serialization code for each individual object.
* In this example, we define a book\_list view function that only accepts GET and POST requests because of the @api\_view decorator. When a GET request is received, all books in the database are retrieved using Book.objects.all(), serialized using the BookSerializer, and returned as a response.
* When a POST request is received, the data is deserialized using the BookSerializer, validated using the is\_valid() method, and if the data is valid, saved to the database using the save() method. The serialized data is returned as a response with a status code of 201 (Created) if the data is saved successfully, and the validation errors are returned with a status code of 400 (Bad Request) if the data is invalid.

**CRUD**

* Text

  Description automatically generated
* Text

  Description automatically generated

**Permissions**

* DRF includes several built-in permission classes that can be used to restrict access to API endpoints based on various criteria. Some of the common built-in permission classes are:
  + AllowAny: Allows unrestricted access to the API endpoint.
  + IsAuthenticated: Allows access only to authenticated users.
  + IsAdminUser: Allows access only to users with admin privileges.
  + IsAuthenticatedOrReadOnly: Allows authenticated users to perform any action, while anonymous users are only allowed to read data.
  + DjangoModelPermissions: Allows access based on the Django model-level permissions.
  + DjangoObjectPermissions: Allows access based on the Django object-level permissions.
* If a permission check fails, a PermissionDenied exception is raised, and the view or viewset returns a 403 Forbidden response to the client.
* Ex:
* Text

  Description automatically generated
* This will ensure that only authenticated users can access the get method of the MyView class.
* Custom Permissions
  + DRF allows you to define custom permissions to control access to your API views. You can create custom permissions by subclassing the BasePermission class and overriding its has\_permission and/or has\_object\_permission methods.
  + Here's an example of a custom permission that only allows access to a view if the request user is the owner of the object being accessed:
  + Text

    Description automatically generated
  + In the above example, the has\_object\_permission method is overridden to check if the object being accessed has an owner attribute that matches the request user. If the check returns True, the method returns True, allowing the user to access the view. Otherwise, the method returns False, denying access.
  + To use this custom permission, you can add it to the permission\_classes attribute of the view or viewset you want to protect:
  + Text

    Description automatically generated

**CORS**

* <https://pypi.org/project/django-cors-headers/>

**Deployment**

* Deploy Django on EC2 <https://youtu.be/uiPSnrE6uWE>
* Hosting static/media files: <https://www.youtube.com/watch?v=f64Ue2C39Ag&t=939s>