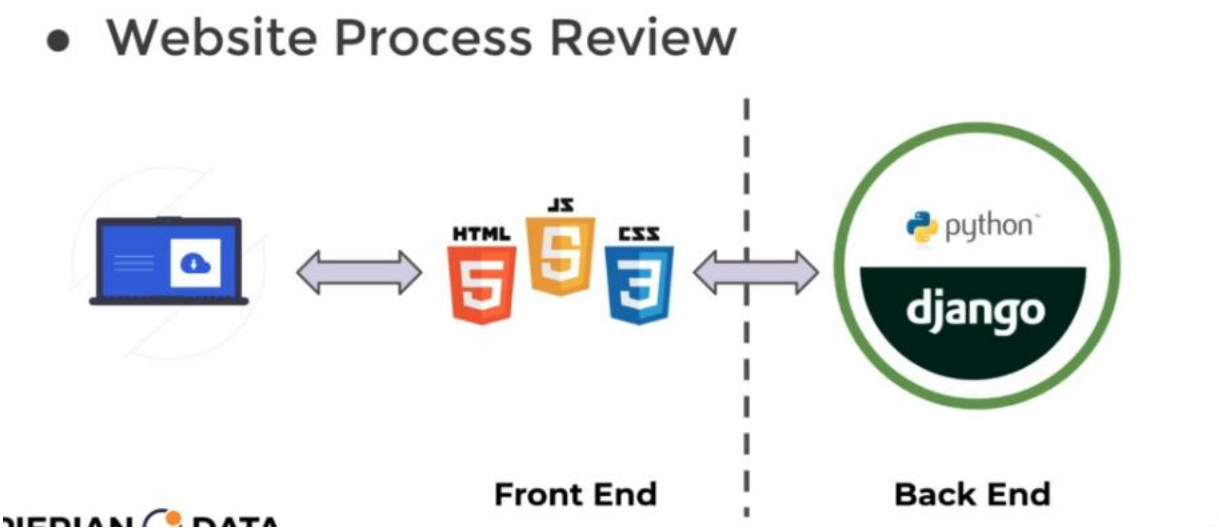
Django

Table of Contents

Section 8: Introduction to Django Framework

1. Introduction to Django Framework



Was created by Adrian Holovaty and Simon Willison who is fan from Django Reinhardt (Jazz - musician)

→ Back-End Framework with Python language for creating web application
→ Django can interact with our web applications to send information to the user of the web application

Why using Django:

Key Features:

- Allows for fast development
- Many common features included
- Updated often and secure
- Very scalable
- Very versatile with Python

Lots of built-in functionality:

- Administration
- Authentication
- Database Interaction
- Security

Allow to use all the methods and modules of Python

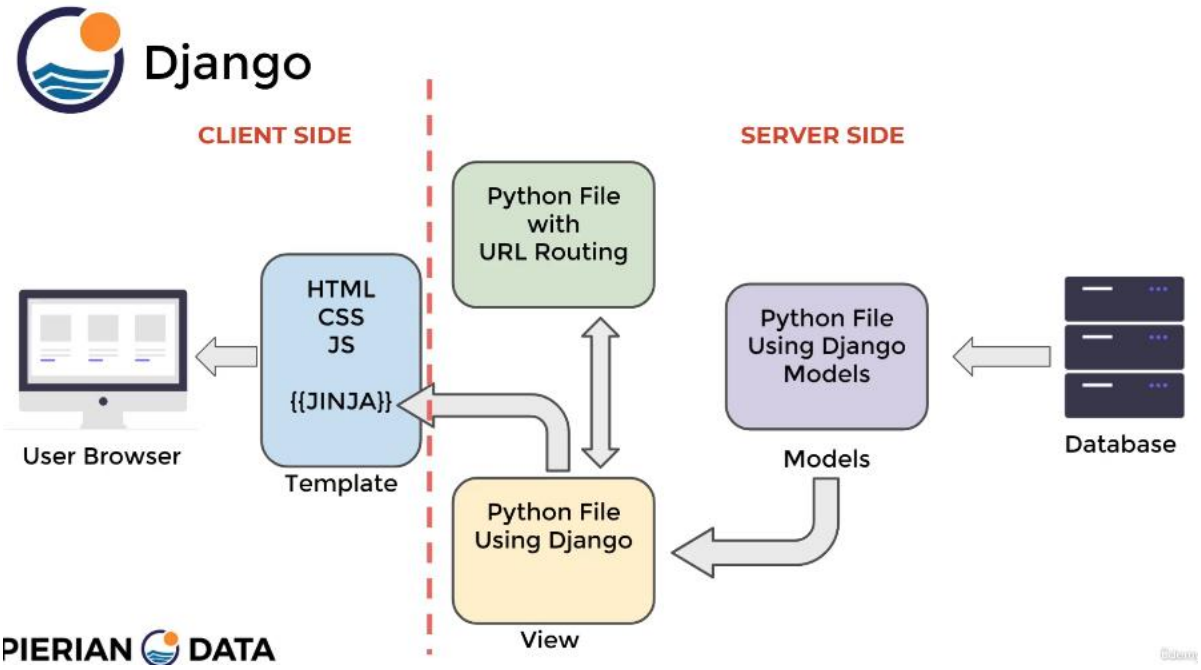
Who uses Django?

- Instagram
- Spotify
- YouTube
- Pinterest
- DropBox
- EventBrite
- And many more

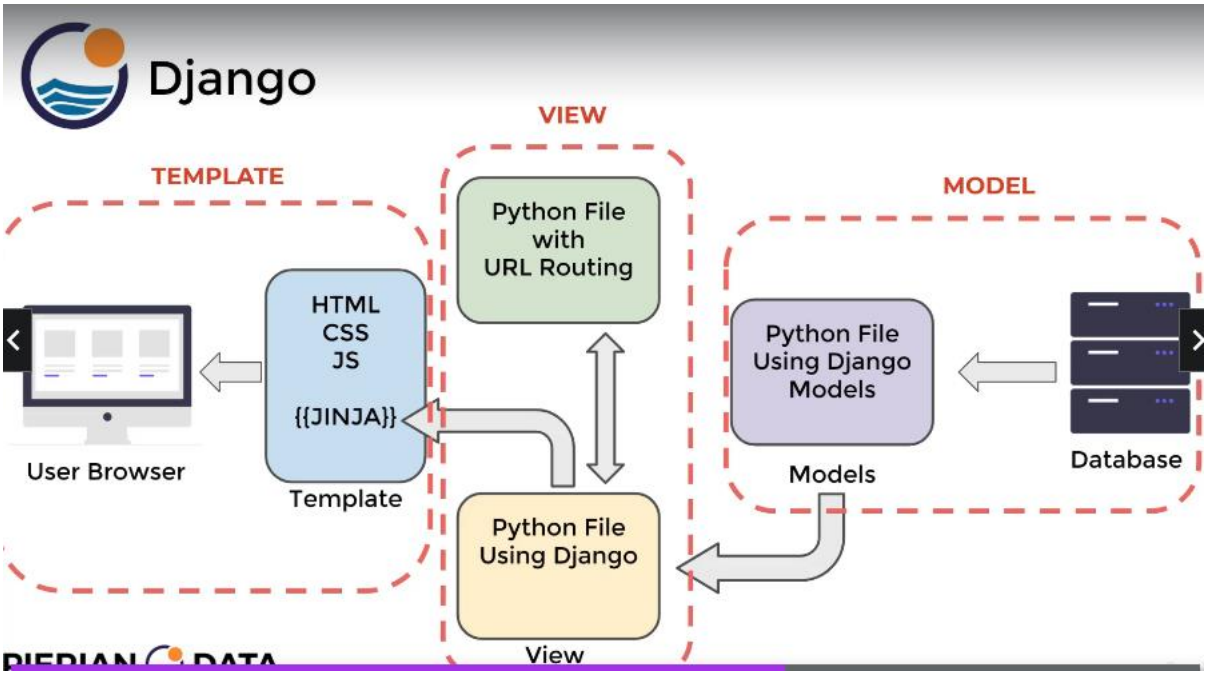
2. How Django works

Key Features of Django:

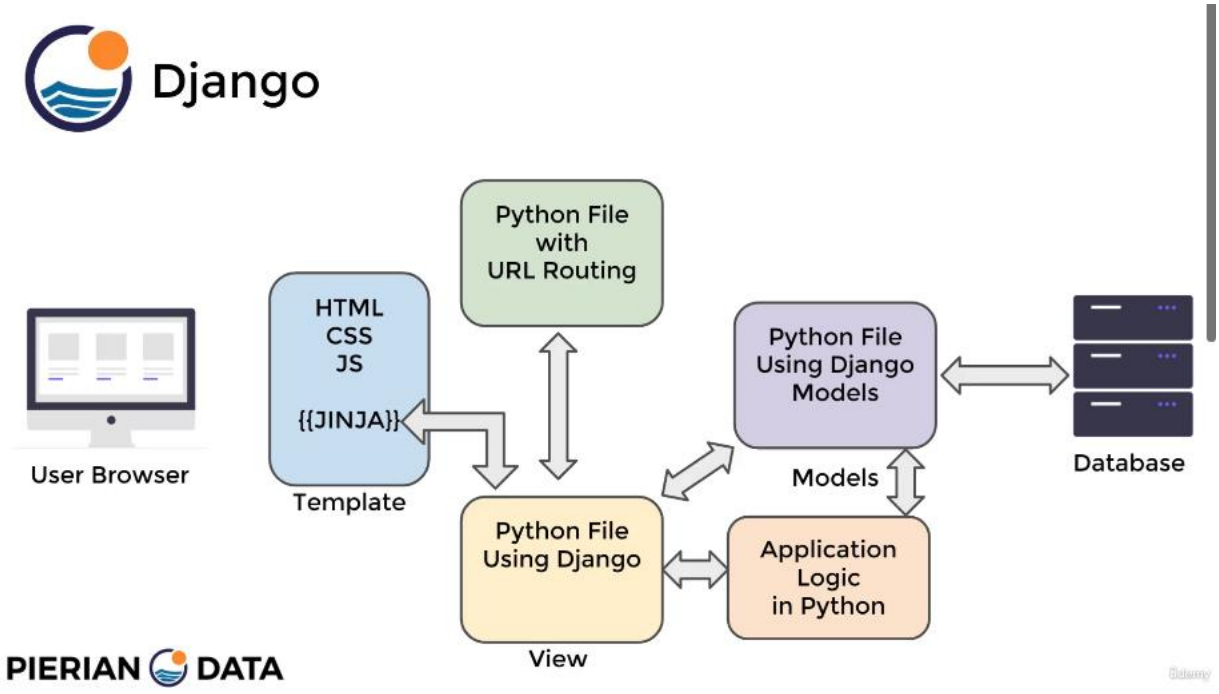
- Model-Template-View (MTV) Structure
  - ORM–Object–relational Mapper
  - Models
  - URLs and Views
  - Templates



Model Section/ View Section/ Template Section



Creating many Applications:



Django structure will also have many more features not shown in this MTV diagram, such as authentication and administration

Django Drawbacks:

- Heavily reliant on idea of Model
- The model is a Python/Django representation of a table in a database
- This makes it very easy to work with querying data, but does add the requirement of understanding Models and setting them up for views

3. First Django Project

At the command prompt, navigate to your desired location and type:

- django_admin → to see available subcommands
- django-admin startproject my_site

This creates the following files and folders

- my_site (root of the project, rename possible, doesn't matter to Django)
    - my_site
    - manage.py (manage the project)

my_site:

- my_site
    - __init.py__
    - settings.py
    - urls.py
    - asgi.py
    - wsgi.py
- manage.py

→ mkdir foldername
→ cd foldername

Make sure your environment is activated by running a new terminal

→ django-admin startproject your_project

Runserver:

→ python manage.py runserver (option: Server number)

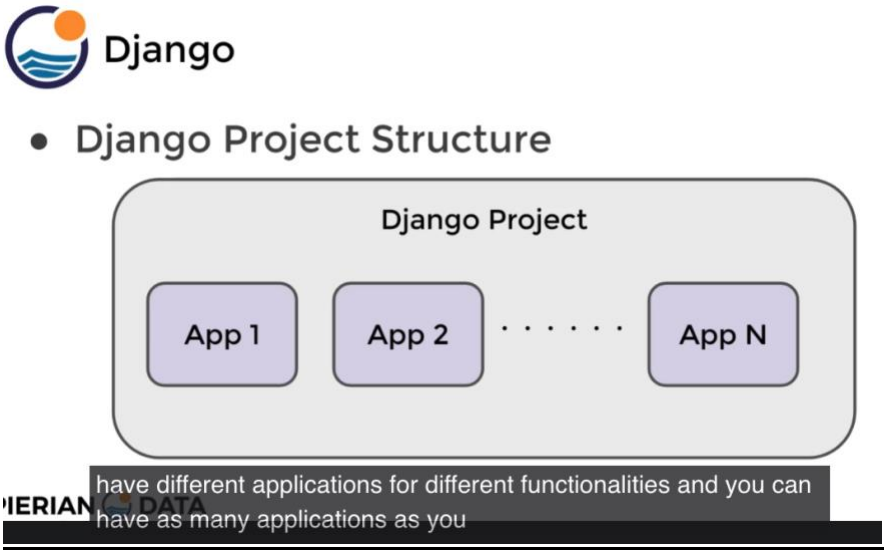
4. First Django Application (Apps)

Django Projects can have separated components called "apps"

- Don't get confused by this nomenclature!
- Typically, a "web app" describes the full website or mobile application on the web
- A "Django app" is a sub-component of a single Django Project (web application)

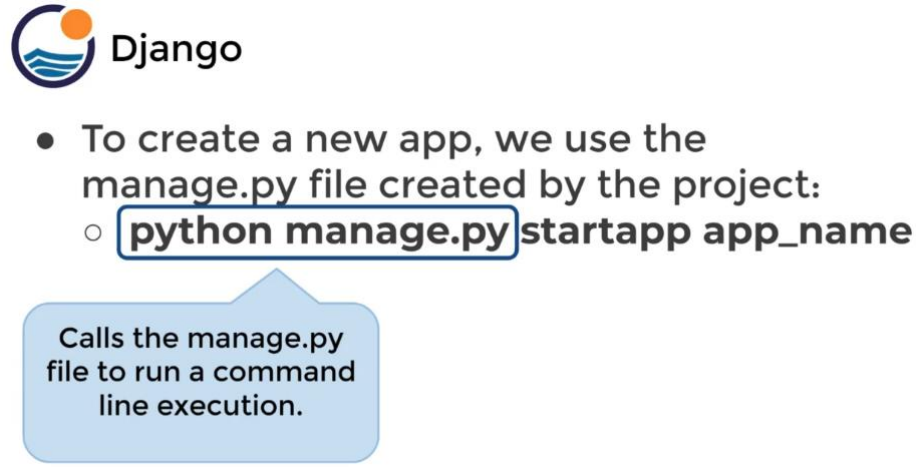
→ Often it becomes much easier to organise your code through the use of apps
→ Each app should cover a different key functionality for your website

→ Also keep in mind that if you are beginning as a solo developer with a simple website, it may make more sense to put everything under a single Django app



→ Managing applications by separating out into Django apps in order to keep these subcomponents organised by functionality

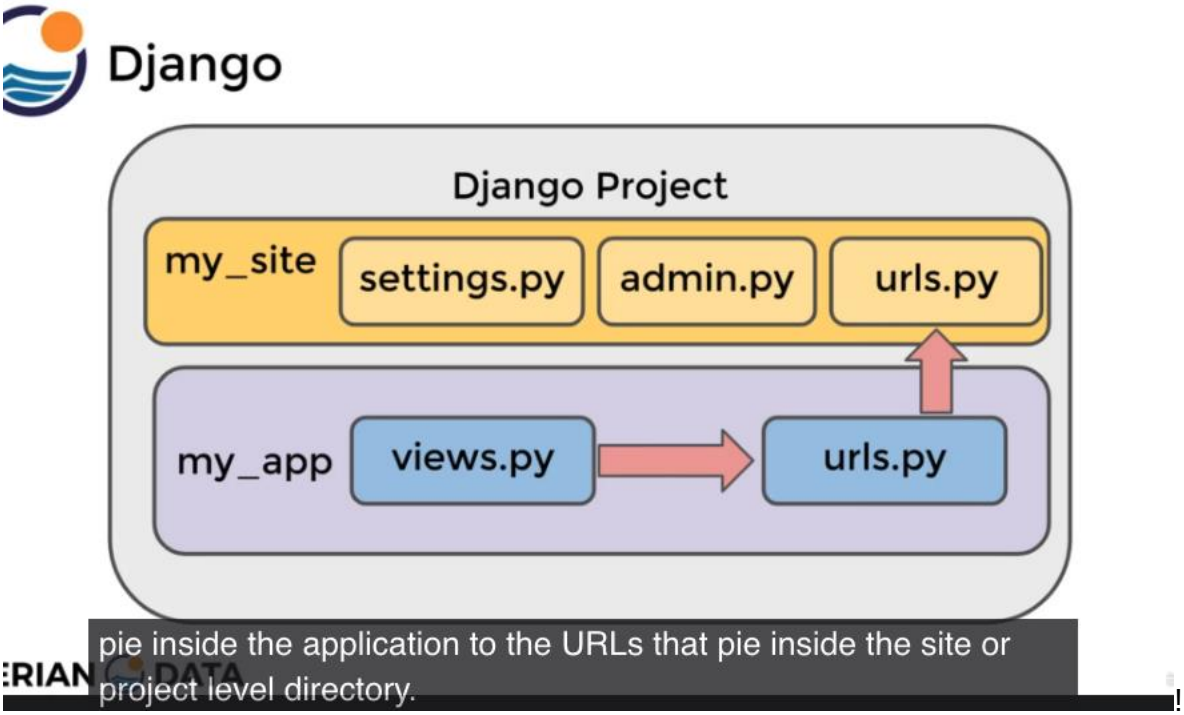
Creating apps:



→ **python manage.py startapp app_name**

Reminder! Make sure you are at the right directory

Creation an example App and connect it to a URL view



Let's begin:

→ Go to my_app (new application)
→ enter index function and import HttpResponse in **views.py**:
Created an Function based view

```
from django.shortcuts import render
from django.http import HttpResponse


# Create your views here.


def index(request):
    return HttpResponse("Hello this is a view inside my app")


# def index(request):
#     return render(request, 'index.html')
```

Creating a file on my_app
→ cd my_app
→ touch urls.py

→ Go to **urls.py**
→ from django.urls import path
And because you are in the same directory as your views.py inside my_app, you can simply say
→ from . import views
(Other way is: from views import index)
→ enter urlpatterns =

```
from django.urls import path
from . import views


urlpatterns = [
    path('', views.index, name='index') #/my_apps --> PROJECT urls.py
]
```

Connect urls.py from my_app with project urls.py:

Follow the instruction by including another URLconf

```
"""stars URL Configuration


The `urlpatterns` list routes URLs to views. For more information please see:
```

```
    https://docs.djangoproject.com/en/4.1/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('my_app/', include('my_app.urls')),
    path('admin/', admin.site.urls),
]
```

Summary:

I created a very simple, function-based view inside of my_app. I created a new URLs that file also inside of my_app. Then I used project level URLs that profile and included and connected the other URL configuration through these following lines of code

NEXT run the sever! Reminder make sure you have saved all your changes

→ **python manage.py runserver**
→ Go on browser http://127.0.0.1:8000/
→ Add on path http://127.0.0.1:8000/my_app/ → "Hello this is a view inside my app"

Section 9: Django – Views, Routing, and URLs

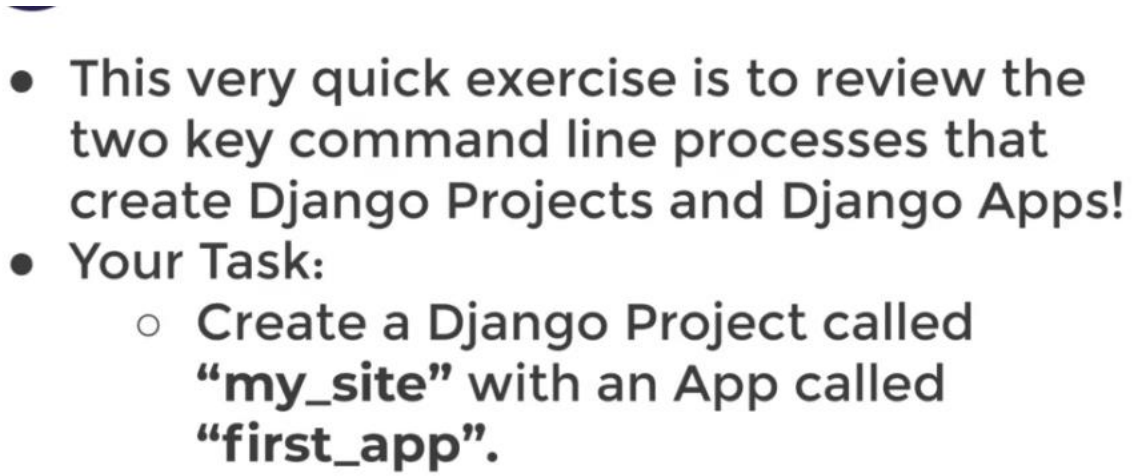
5. Introduction Views, Routing and URLs

Section Overview:

- Project and App review
- Routes and URLs
- Basic Function Views
- Dynamic Views and View Logic
- Redirects and 404s
- URL names and reverse()
- Connecting to Templates

Focus on function-based view

6. Project Application Exercise



Create project:
→ cd foldername
→ django-admin startproject my_site

Create app:
→ cd my_site
→ python manage.py startapp first_app

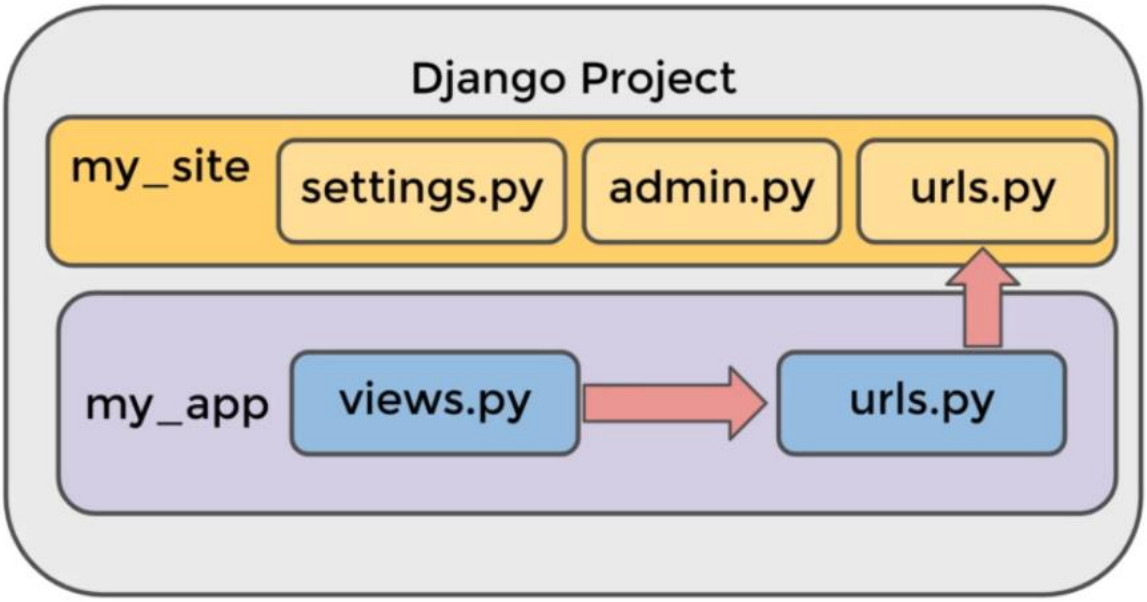
7. Views and URLs Overview

In Django, **Views** dictate *what* information is being shown to the client, and URLs dictate *where* that information is shown on the website

These work in concert so you can think of each View/URL pairing as a web page on the website

- Keep in mind that not every permutation of a webpage can be known in advance, for example, “How many *total* blog posts will a website blog eventually have?”
- Django Views and URLs support a lot of dynamic and logic features to help with this sort of task

- Recall that we have URL configurations at a Project level and at an App level.
- We connect these through the use of **path()** and **include()** Django functions
- A list of view routes is defined in a list variable called **urlpatterns**



Connecting a View to a URL with **path():**

- **Route**
  - String code that contains the URL pattern
  - Django will scan the relevant **urlpatterns** list until if find a matching string route (e.g.,“app/”)
- **View**
  - Once a matching route is found, the view argument connects to a function or view, typically defined in the **views.py** file of the relevant Django app
- **kwargs**
  - Allow us to pass in keyword arguments as dictionary to the view
- **name**
  - Allow us to name a URL in order to reference it elsewhere in Django

8. Function Based Views – Basics

On App:

View.py:

```
from django.shortcuts import render
from django.http.response import HttpResponse

# Create your views here.

def simple_view(request):
    return HttpResponse("Simple View")
```

Connect with function-based view on urls.py:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.simple_view)
]
```

On Project:

urls.py:

```
"""my_site URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/4.1/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
```

```
from django.contrib import admin
from django.http.response import HttpResponse
from django.urls import path, include
# from . import views       #created view.py on project level as third way for creating views

# without using view and url from an app
# showing function to the client by calling it in path without including/importing function from the app
def say_again_hallo(request):
    return HttpResponse("Hallo again")

urlpatterns = [
    path('admin/', admin.site.urls),
    path('first_app/', include('first_app.urls')),
# calling function without string route and include
    path('', say_again_hallo),
#   path('', views.project_level_view)      #created function on views.py (project level) as third way for showing view on browser
]
```

Different:

Direct way without using an app: Only in urls.py project level

Just run server and see the output: http://127.0.0.1:8000/

Direct way without using an app: Created views.py on project level

Just run server and see the output: http://127.0.0.1:8000/

Normal way with app: urls.py and views.py from app level

Runserver and apply http://127.0.0.1:8000/my_app/

9. Dynamic Views – Routing Logic

Possible to use dynamic views by starting to integrate the idea of different python objects within views and using Python code and logic. Furthermore, it is possible to have user dynamically updated routing and views.

Several function-based views:

On views.py app level

```
from django.shortcuts import render
from django.http.response import HttpResponse

# Create your views here.

def simple_view(request):
    return HttpResponse("Simple View")

def different_view(request):
    return HttpResponse("Second View")
```

Connect to URLs on app level:

```
from django.urls import path
from . import views

urlpatterns = [
    path('simple/', views.simple_view),
    path('different/', views.different_view)
]
```

Connect to URLs on project level:

```
from django.contrib import admin
from django.urls import path, include


urlpatterns = [
    path('admin/', admin.site.urls),
    path('first_app/', include('first_app.urls'))
```

```
]
```

Dynamic view with dictionary:

On views.py app level

→ Using a dictionary

articles = {
‘sports’ : ‘Sports Page’,
‘finance’ : ‘Finance Page’,
‘general’ : General Page

→ Function news_view with parameter: topic

```
from django.shortcuts import render
from django.http.response import HttpResponse


# Create your views here.


articles = {'finance' : 'Finance Page',
            'sports' : 'Sports Page',
            'general' : 'General Page',
}


def news_view(request, topic):
    return HttpResponse(articles[topic])
```

Connect to URLs app level:
Allow user to enter topic in browser with races brackets → <>/

```
from django.urls import path
from . import views


urlpatterns = [
    path('<topic>/', views.news_view)
]
```

Connect to URLs on project level:
Same as above!!

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('first_app/', include('first_app.urls'))
]
```

Don't forget to run the server!

Enter after app_name/topic
→ for example topic = finance

Output:

Dynamically updating view by user:

Allow only proper types:

10. Path converters

The following path converters are available by default:

- **`str`** - Matches any non-empty string, excluding the path separator, **`'/'`**. This is the default if a converter isn't included in the expression.
- **`int`** - Matches zero or any positive integer. Returns an **`int`**.
- **`slug`** - Matches any slug string consisting of ASCII letters or numbers, plus the hyphen and underscore characters. For example, **`building-your-1st-django-site`**.
- **`uuid`** - Matches a formatted UUID. To prevent multiple URLs from mapping to the same page, dashes must be included and letters must be lowercase. For example, **`075194d3-6885-417e-a8a8-6c931e272f00`**. Returns a **`UUID`** instance.
- **`path`** - Matches any non-empty string, including the path separator, **`'/'`**. This allows you to match against a complete URL path rather than a segment of a URL path as with **`str`**.

On views.py app level:

```
def add_view(request, num1, num2):
    add_result= num1 + num2
```

```
    result = f"{num1} + {num2} = {add_result}"
    return HttpResponse(result)
```

Connect to URLs on app level:
Important: Assigning types → int for numbers

```
from django.urls import path
from . import views

urlpatterns = [
    path('<str:topic>', views.news_view),
    path('<int:num1>/<int:num2>', views.add_view)
]
```

Connect to URLs on project level:
Same as above!!

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('first_app/', include('first_app.urls'))
]
```

Output:

11. Using ResponseNotFound and 404 Pages

Inevitably there will be a time when a client requests a web page view that does not exist, or the client did not provide the correct URL route

In these cases, we can use **HttpResponseNotFound()** to still return a webpage for the user

Django also has a default 404 page we can display
Later on when we dive deeper into templates we'll see how to create our own custom 404 page

Using HttpResponseNotFound:

With **try and expect command** for raising errors!

On views.py app level

Important → import class HttpResponseNotFound

```
from django.shortcuts import render
from django.http.response import HttpResponse, HttpResponseNotFound

# Create your views here.

articles = {'finance' : 'Finance Page',
            'sports' : 'Sports Page',
            'general' : 'General Page',
}

def news_view(request, topic):
    try:
        return HttpResponse(articles[topic])
    except:
        return HttpResponseNotFound('Topic doesn\'t exist')
```

On URLs app level:
Same as above!

```
from django.urls import path
from . import views

urlpatterns = [
    path('<str:topic>', views.news_view),
    path('<int:num1>/<int:num2>', views.add_view)
]
```

On URLs project level:
Same as above

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('first_app/', include('first_app.urls'))
```

```
]
```

Output:

By entering topic = politics (doesn't exist)

Using Http404:

Hint: raise Http404 and import class Http404

For convenience, and because it's a good idea to have a consistent 404 error page across your site, Django provides an Http404 exception. If you raise Http404 at any point in a view function, Django will catch it and return the standard error page for your application, along with an HTTP error code 404.

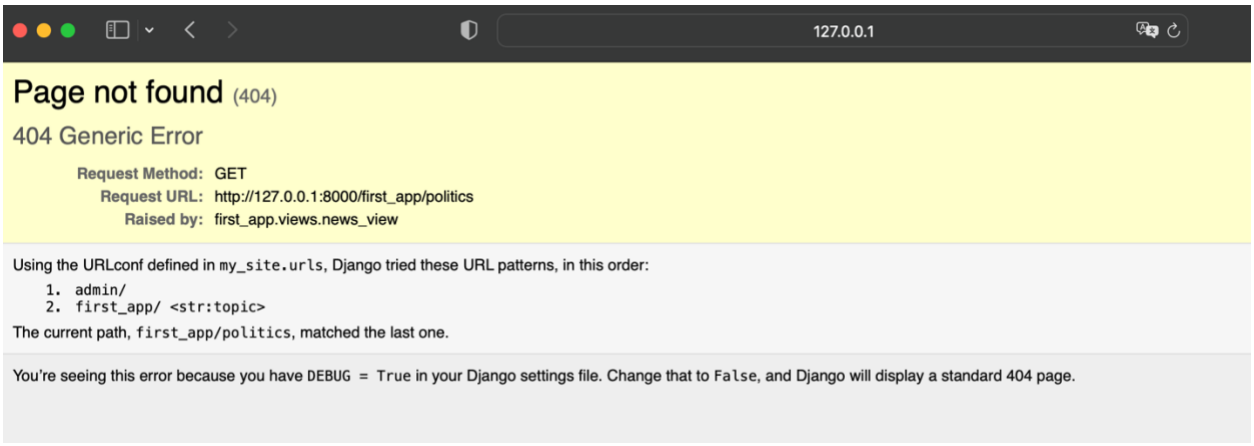
Example usage:

```
from django.shortcuts import render
from django.http.response import HttpResponse, HttpResponseNotFound, Http404

# Create your views here.

articles = {'finance' : 'Finance Page',
            'sports' : 'Sports Page',
            'general' : 'General Page',
}

def news_view(request, topic):
    try:
        return HttpResponse(articles[topic])
    except:
        raise Http404('404 Generic Error') # 404.html
```

In order to show customized HTML when Django returns a 404, you can create an HTML template named 404.html and place it in the top level of your template tree. This template will then be served when DEBUG is set to False.

When DEBUG is True, you can provide a message to Http404 and it will appear in the standard 404 debug template. Use these messages for debugging purposes; they generally aren't suitable for use in a production 404 template.

Not suitable for the general public as too much information with DEBUG = True → **Output:**



→ Therefore, change settings:

```
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True


ALLOWED_HOSTS = []
```

→ DEBUG = FALSE

ALLOWD_HOST = ['127.0.0.1'] # allow your host as show an error: CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.

```
DEBUG = False


ALLOWED_HOSTS = ['127.0.0.1']
```

Output:

Not Found

The requested resource was not found on this server.

Steps:

→ In the beginning using HttpResponseNotFound, especially if you are debugging things
→ later on, using Generic Error with Http404
→ linking to our own custom forum for each HTML template

12. Redirects Basics

Sometimes a client user will provide a path that we want to redirect to another webpage on our site

This can be accomplished in Django through the use of the **HttpResponseRedirect()** function

Let's imagine that we want to match our website newspaper article pages to the **numeric** page in the physical newspaper for readers to quickly find an article online

For example, if **Finance** was on page 2 of the newspaper, we want to redirect first_app/2/ to first_app/finance/.

Note!
→ The example shown here is not the typical way we'll redirect pages, it is actually error-prone, which we'll talk about in the lecture

Idea:

 # domain.com/first_app/0 → domain.com/first_app/finance

Views.py app level:

Hint: import HttpResponseRedirect

```
def num_page_view(request, num_page):


    topic_list = list(articles.keys()) # ['finance', 'sports', 'general']
    topic = topic_list[num_page]
```

```
    return HttpResponseRedirect(topic)
```

URLs app level:

```
from django.urls import path
from . import views

urlpatterns = [
    path('<int:num_page>', views.num_page_view), #new path
    path('<str:topic>', views.news_view),
    path('<int:num1>/<int:num2>', views.add_view)
]
```

URLs project level:
Still the same!

13. Reverse URLs and URL Names

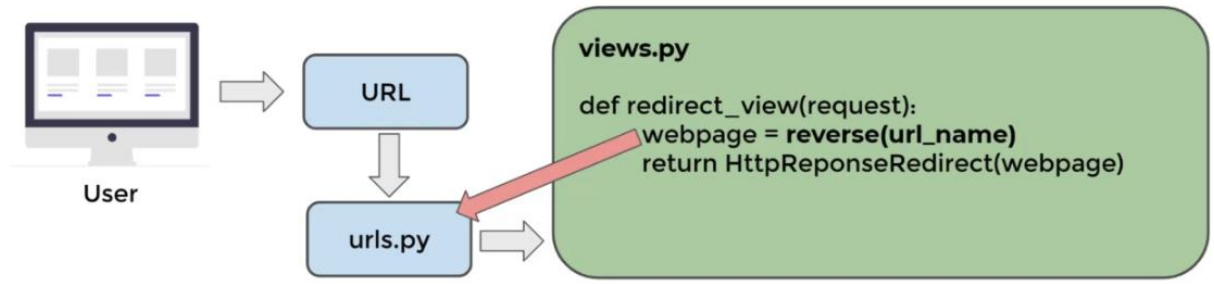
As our website get larger and more complex, we'll keep adding more views and more URLs across the project and apps

This leads to more instances where we'll need to reference existing pages (URLs on across our website)

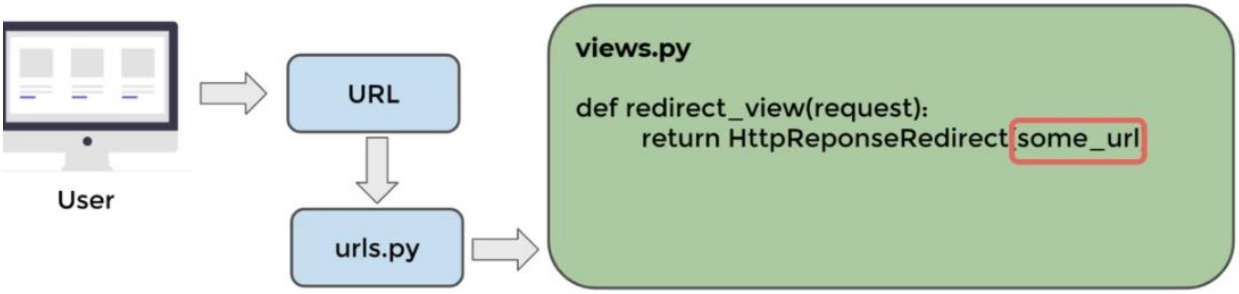
In this lecture we'll explore two key ideas:
URL Path inside the **path()** function can have ***names*** across Django **and** inside of a template

Django has a **reverse()** function to find the corresponding URL path for URL **name**

- Django **reverse()** function!



- What if the request for the URL is in views?



We are inside of views and the idea is to get the path/name/reference by looking what is happening inside URLs for this URLs name!

Let's name a URL and then use it inside a **reverse()** call for Django

On URLs app level:

```
from django.urls import path
from . import views

urlpatterns = [
    path('<int:num_page>', views.num_page_view),
    path('<str:topic>', views.news_view, name='topic-page'),
    # new keyword argument kwargs -> name='topic-page' give the URLs Path a name for reference in views.py
    path('<int:num1>/<int:num2>', views.add_view)
]
```

On views app level:

Hint: args=[…] # always pass argument in a list, even if it's just one item

args=[topic] # see on URLs Path for the argument →'<str:**topic**>'

Important: import reverse()

```
from django.urls import reverse
```

```
def num_page_view(request, num_page):

    topic_list = list(articles.keys()) # ['finance', 'sports', 'general']
    topic = topic_list[num_page]

    webpage = reverse('topic-page', args=[topic])

    return HttpResponseRedirect(webpage)
```

less code →

```
return HttpResponseRedirect(reverse('topic-page', args=[topic]))
```

The idea is: reverse function looks up that URL Path and send then these arguments
→ Redirect is much cleaner, much easier to read and much better for maintenance

14. Connecting a View to a Template

Realistically we don't want to have to manually type out HTML code or HTTP Responses inside our views.py file

Instead we would like to separate out all our templates (HTML files) into a separate directory and have views communicate between this directory and render the templates

Connecting to template directory requires us to inform the Django project settings where to find these templates

In this simple example, we'll store all the templates at a project level

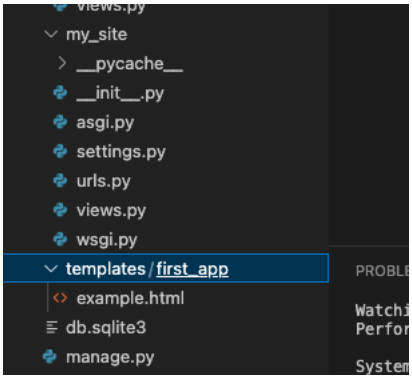
Later on we'll explore how to store templates on a Django app level (recommended way of doing things)

In this lecture we'll:

- Create a new templates directory
- Create an example .html file

- Connect to that .html within the view
- Inform the Django project where to find this template directory inside of settings.py

Created templates folder and first_app folder inside of templates to create an .html file (example.html)



On settings.py:
Import os

→ Templates → os.path.join(BASE_DIR, 'templates')

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR,'templates')], # in order to look fot html files in this directory as well
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

In views.py app level:

Hint: uses render for requesting .html file (import render)

```
from django.shortcuts import render
from django.http.response import HttpResponse, HttpResponseNotFound, Http404, HttpResponseRedirect
from django.urls import reverse

# Create your views here.

def simple_view(request):
    return render(request, 'first_app/example.html') # .html
```

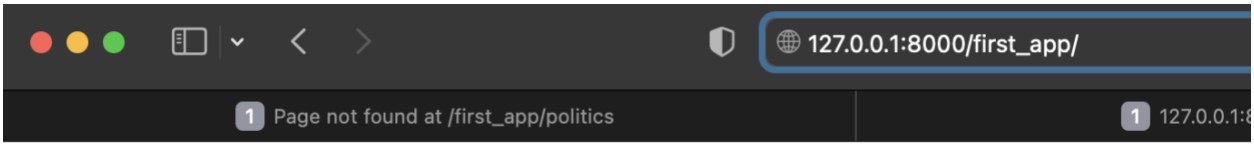
Inside in URLs app level:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.simple_view) # domain.com/first_app
]
```

On example.HTML:

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1> Hallo HTML!!</h1>
</body>
</html>
```

Output:



Hallo HTML!!

Section 10: Django – Templates

15. Django and Templates

In this section we'll begin to explore how to connect our Django views code to templates files (HTML)

This will allow us to take advantage of HTML, CSS, and even JS along with any backend Python logic

Section Overview:

- Template Directories
- Template Rendering
- Django Template Language
    - Context Insertion
    - Filters and Tags
    - URL Names in Templates
    - Templates Inheritance
    - Template Specifics
        - Static Files
        - Custom 404 Templates
    - Templates Exercise

16. Template Directories (Important)

In the previous section we explored a very basic way to connect a view to an existing template file

Usually, however, we would like to separate out template folders based on their application rather than have a single template folder for the entire project

Separating out template directories by app is more ideal because in the future you may want to reuse a Django app in a future project

Having everything in its own app directory makes this reuse easy

In order for the Django Project to be aware of the app's template directory existence, we do need to register the custom Django app in the ***settings.py*** under the **INSTALLED_APPS** variable

Let's describe the process to create a template directory for a custom Django app before we walk through it in our code editor

Important Note: Many of these steps won't do anything for us right now with templates, but are important later on for models, so we'll run them anyways.

- Step One:
  - Setup the Django App
    - Create App Directory with **manage.py startapp** command
    - Create the relevant URLs and Views
    - Map the App URLs to the Project URLs

→ Already done before!

- Step Two:
  - Run the migrate command
    - **python manage.py migrate**
  - This command looks at **INSTALLED_APPS** in settings and creates any necessary database tables

→ DATABASE AND MODELS

- Step Three:
    - Inside of Django App check the apps.py created automatically for you and register the AppConfig class to **INSTALLED_APPS** inside of settings.py

→ Linking to let your project be aware of the app directories, which will eventually let it be aware of the template directory inside the app

- Step Four:
    - Register the app and any database changes with Django by running:
        - **python manage.py makemigrations myapp**
    - Note, this won't be relevant for us until we have actually created models

- Step Five:
    - Run **python manage.py migrate** again to create the model tables in our database
    - Again, not necessary for us yet, but will be once we create models

- Step Six:
    - Create a template directory inside your app directory with structure:
        - my_site
            - my_app
                - templates
                    - my_app
                        - example.html

Why does my_app appear twice?

Often, you'll have multiple template files with the same name (multiple index.html files, one for each app index view page).

Because of the way Django searches for matching template names, to make sure we get the relevant template for an app, we create the app subdirectory underneath the template folder. As Django will choose the first template it finds whose name matches, and if you had a template with the same name in a different application, Django would be unable to distinguish between them. Hence, the best way to point Django at the right one is by namespacing them.

Steps:

One:

→ mkdir projectname
→ cd projectname
→ django-admin startproject my_projectname
→ cd my_projectname
→ python manage.py startapp my_app
→ cd my_app
→ touch urls.py (**create a urls.py file in my_app**)
→ create function in views.py (app level)

```
from django.shortcuts import render


# Create your views here.


def example_view(request):
    # my_app/templates/my_app/example.html (later on telling Django we have templates directory underneath my app)
    return render(request, 'my_app/example.html') # 'my_app/example.html' connect to template
```

→ go to urls.py in my_app
→ code path

```
from django.urls import path
from . import views


urlpatterns = [
    path('', views.example_view)
]
```

→ including URLconf in urls.py (project_level)
→ path(‘my_app/’, include(‘my_app.urls’)) look below:

```
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path, include
```

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('my_app/', include('my_app.urls'))
]
```

Two:

→ on projectname directory
→ python manage.py migrate

```
(base) giales@Gianlucas-MacBook-Air-2 step_site % python manage.py  migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
(base) giales@Gianlucas-MacBook-Air-2 step_site %
```

Three:

→ check the apps.py if app is created automatically for you (**MyAppConfig**)

```
from django.apps import AppConfig


class MyAppConfig(AppConfig): # converts automatically the kenel casing name (this is gonna be my installed app)
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'my_app'
```

→ settings.py (project level)
→ go to list **INSTALLED_APPS**
→ register my_app in the list
→ ‘my_app.apps.MyAppConfig’

```
# Application definition

INSTALLED_APPS = [
    'my_app.apps.MyAppConfig', # Django will connect with templates, views and so on
```

```
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Four:

→ Register the app and any database changes with Django by running → **python manage.py makemigrations my_app**

Because no models:
NO CHANGES!

```
(base) giales@Gianlucas-MacBook-Air-2 step_site % python manage.py makemigrations my_app
No changes detected in app 'my_app'
(base) giales@Gianlucas-MacBook-Air-2 step_site %
```

Five:

→ If changes have been done then push those changes with: → **python manage.py migrate**

```
(base) giales@Gianlucas-MacBook-Air-2 step_site % python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  No migrations to apply.
(base) giales@Gianlucas-MacBook-Air-2 step_site %
```

Six:

→ Create a template directory inside your app directory with structure:

- my_site
  - my_app
    - **templates**
      - **my_app**
        - **example.html**

→ example.html
→ doc: automatically create html code
→ body: <h1>HTML file is connected</h1>

Seven:

→ python manage.py runserver
→ start browser http://127.0.0.1:8000/my_app

17. Variables in Templates

Let's explore how to render templates and send a context variable to the HTML with the Django Template Language

→ create new html in template folder
→ variables.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1> New Variables</h1>
</body>
</html>
```

→ views.py (app level)

```
def variable_view(request):
    return render(request, 'my_app/variables.html')
```

→ urls.py (app level)

```
urlpatterns = [
    path('', views.example_view),
    path('variable/', views.variable_view)
]
```

→ done!!

Function with dictionary to represent the values of them:

On view.py:

```
def variable_view(request):

    my_var = {'first_name': 'Rosallind', 'last_name' : 'Franklin',
            'some_list': [1,2,3], 'some_dict':{'inside_key': 'inside_value'}
    }

    return render(request, 'my_app/variables.html', context=my_var)
```

On variable.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1> New Variables</h1>
    <h2> The name of the princess is: {{first_name}} {{last_name}}</h2>
    <p>Furthermore I wisch my list is: {{some_list}}</p>
    Maybe, I would like to receive the third item of the list: {{some_list.2}}<br>
    A dictionary inside in a dictionary?? Kinda cool: {{some_dict}}<br>
    I wish, I could just see the value of the dict in the dict
    <p>Here we are: {{some_dict.inside_key}}</p>

   <p>command: Django language: {#  just commands    #}</p>
    <p>comannd: HTML languages: <!--   just commands    --></p>

</body>
</html>
```

Output:

New Variables

The name of the princess is: Rosallind Franklin

Furthermore I wisch my list is: [1, 2, 3]

Maybe, I would like to receive the third item of the list: 3
A dictionary inside in a dictionary?? Kinda cool: {'inside_key': 'inside_value'}
I wish I could jsut see the value of the dict in the dict

Here we are: inside_value

command: Django language:

comannd: HTML languages:

For representing variables use two sets of curly braces {{}}

If you have a list or dictionary, you want to reference something inside then use dot notation . and put the the number for the index or the key for the value

18. VS Code Django Extension

Use Django extension for enable Django syntax for clearing the visibility

With extension:

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
```

```
    <h1> New Variables</h1>
    <h2> The name of the princess is: {{first_name}} {{last_name}}</h2>
    <p>Furthermore I wisch my list is: {{some_list}}</p>
    Maybe, I would like to receive the third item of the list: {{some_list.2}}<br>
    A dictionary inside in a dictionary?? Kinda cool: {{some_dict}}<br>
    I wish I could jsut see the value of the dict in the dict
    <p>Here we are: {{some_dict.inside_key}}</p>

   <p>command: Django language: {#  just commands    #}</p>
    <p>comannd: HTML languages: <!--   just commands    --></p>

</body>
</html>
```

Look up to lecture above for the different

19. Filters

Filters are built-in modifiers in Django templating that allow you to quickly apply a change to a variable on the template side, rather than in your Python script

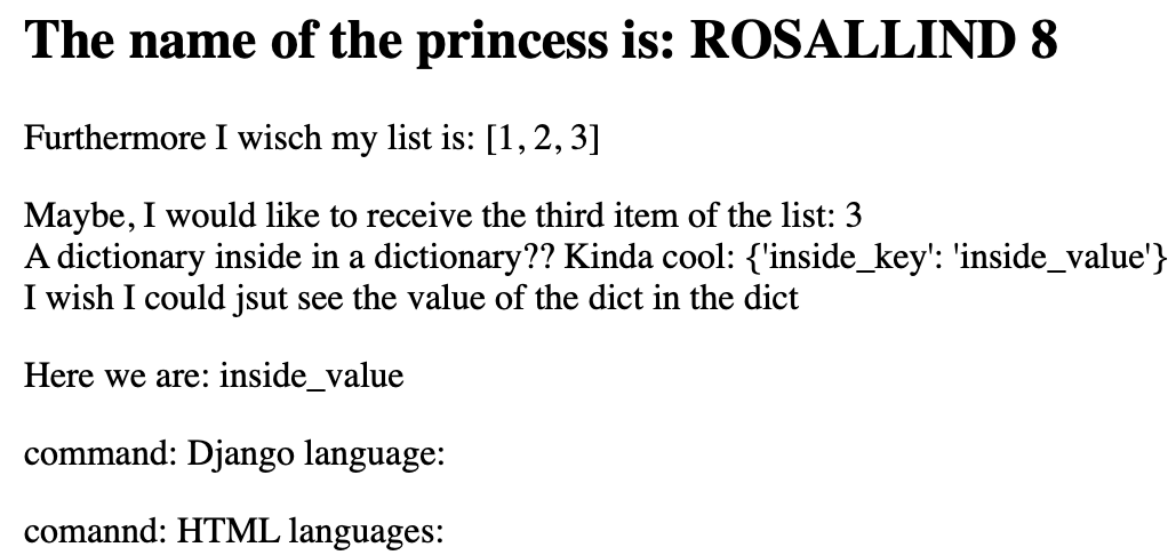
https://docs.djangoproject.com/en/4.1/ref/templates/builtins/

upper or length and so on

```
<body>
    <h1> New Variables</h1>
    <h2> The name of the princess is: {{first_name | upper }} {{last_name | length}}</h2>
    <p>Furthermore I wisch my list is: {{some_list}}</p>
    Maybe, I would like to receive the third item of the list: {{some_list.2}}<br>
    A dictionary inside in a dictionary?? Kinda cool: {{some_dict}}<br>
    I wish I could jsut see the value of the dict in the dict
    <p>Here we are: {{some_dict.inside_key}}</p>

   <p>command: Django language: {#  just commands    #}</p>
    <p>comannd: HTML languages: <!--   just commands    --></p>

</body>
```

```
</html>
```

Output



Also possible: 2 filters for one value

```
my_var = {'first_name': 'rosaLind', 'last_name' : 'Franklin',
```

look at rosaLind!

Change it with two filters at html side

{{first_name | lower | capfirst}}

```
<h1> New Variables</h1>
    <h2> The name of the princess is: {{first_name | lower | capfirst}} {{last_name | length}}</h2>
    <p>Furthermore I wisch my list is: {{some_list}}</p>
    Maybe, I would like to receive the third item of the list: {{some_list.2}}<br>
    A dictionary inside in a dictionary?? Kinda cool: {{some_dict}}<br>
    I wish I could jsut see the value of the dict in the dict
    <p>Here we are: {{some_dict.inside_key}}</p>

   <p>command: Django language: {#  just commands    #}</p>
   <p>comannd: HTML languages: <!--   just commands    --></p>
```

Output:

New Variables

The name of the princess is: Rosalind 8

Furthermore I wisch my list is: [1, 2, 3]

Maybe, I would like to receive the third item of the list: 3
A dictionary inside in a dictionary?? Kinda cool: {'inside_key': 'inside_value'}
I wish I could jsut see the value of the dict in the dict

Here we are: inside_value

command: Django language:

comannd: HTML languages:

20. Tags – For Loops

Django Tags are able to provide further logic at the template in the rendering process.

This includes a lot functionalities, such as for loops, if-else statements, and linking to URLs.

Let's begin by exploring for-loop tags.

Tags: { % …%}

```
<body>

  <h1> New Variables</h1>
  {% for  item in some_list %}

     <h1>{{item}}</h1>
```

```
  {% endfor %}
</body>
```

Output:

New Variables

1

2

3

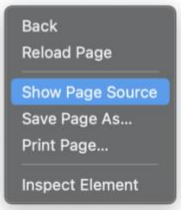
Look at the page source for understanding what the HTML page is showing

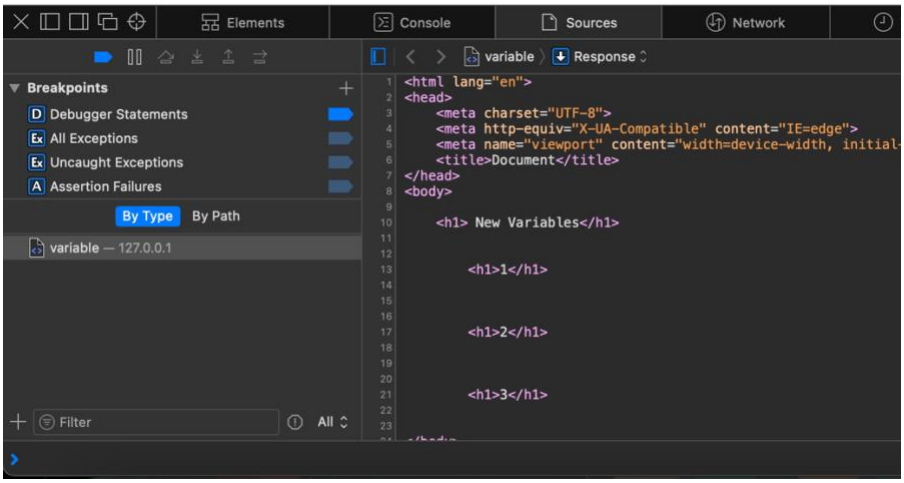
New Variables

1

2

3





Unsorted List with for loop:

```
ody>
```

```
  <h1> New Variables</h1>

  <ul>
  {% for  item in some_list %}

     <li>{{item}}</li>

  {% endfor %}
  </ul>
</body>
</html>
```

Output:

New Variables

- 1
- 2
- 3

HINT: Always checking the source code if you get stuck

Documentation:

https://docs.djangoproject.com/en/4.1/topics/templates/

Looping through the dictionary:

```
<body>

  <h1> New Variables</h1>

  <ul>
  {% for  k, v in some_dict.items %}

     <li>{{k}} : {{v}}</li>
```

```
    {% endfor %}
    </ul>
</body>
</html>
```

21. Tags – If, Elif, Else

Django also has **if, elif, and else** tags

We can use Boolean and comparison operators along with these tags, for example

- **==, or, and, not, >=**

Similar to the Django Tag for **loops,** we will need an **end** Tag **if** statements

- {% if var == True %}
    - HTML CODE
- {% endif %}

https://docs.djangoproject.com/en/4.1/ref/templates/builtins/#if

IF user logged in
THEN print Welcome back
END IF

On views.py

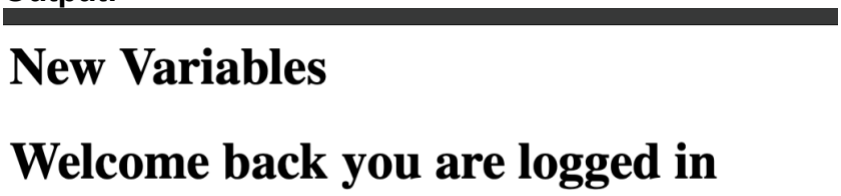
```
def variable_view(request):

    my_var = {'first_name': 'rosaLind', 'last_name' : 'Franklin',
            'some_list': [1,2,3], 'some_dict':{'inside_key': 'inside_value'}, 'user_logged_in' : True
    }

    return render(request, 'my_app/variables.html', context=my_var)
```

On html file

```
<body>

    <h1> New Variables</h1>
```

```
    {% if user_logged_in %}

    <h1>Welcome back you are logged in</h1>

    {% endif %}
</body>
```

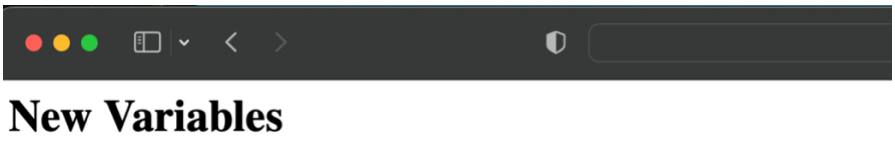
Output:



By false the screen doesn't show anything:

```
my_var = {'first_name': 'rosaLind', 'last_name' : 'Franklin',
            'some_list': [1,2,3], 'some_dict':{'inside_key': 'inside_value'}, 'user_logged_in' : False
    }
```

Output:



If and for loop together:

```
<body>

    <h1> New Variables</h1>
    <ul>
    {% for num in some_list %}
        {% if num == 2 %}
            <li>TWO</li>
```

```
        {% else %}
            <li>{{num}}</li>
        {% endif %}
    {% endfor %}
    </ul>

</body>
</html>
```

Output:

New Variables

- 1
- TWO
- 3

HINT: Space between the operators, variables and statements to avoid errors

22. Tags and URL Names in Templates

Recall in the **path()** function call we could assign names to URLs

This in turn allows us to use the **{% url %}** tag to easily create links to other pages based on their URL name in urls.py

Let's explore how this works!

Step one:

→ Add the app name inside of URLs.py

Urls.py app level
→ app_name = 'my_app'

```
from django.urls import path
from . import views

# register the app namespace
# URL NAMES
app_name = 'my_app'
```

```
urlpatterns = [
    path('', views.example_view),
    path('variable/', views.variable_view)
]
```

Step two:

→ Gives these paths names that we can use

```
urlpatterns = [
    path('', views.example_view, name='example'),
    path('variable/', views.variable_view, name='variable')
]
```

Step three:

→ link url path to the html file

```
<body>

    <h1> VARIABLES HTML TEMPLATE</h1>

    <h1> <a href="{% url 'my_app:example' %}">Click me to go to example</a> </h1>

</body>
```

If html file is on project level:

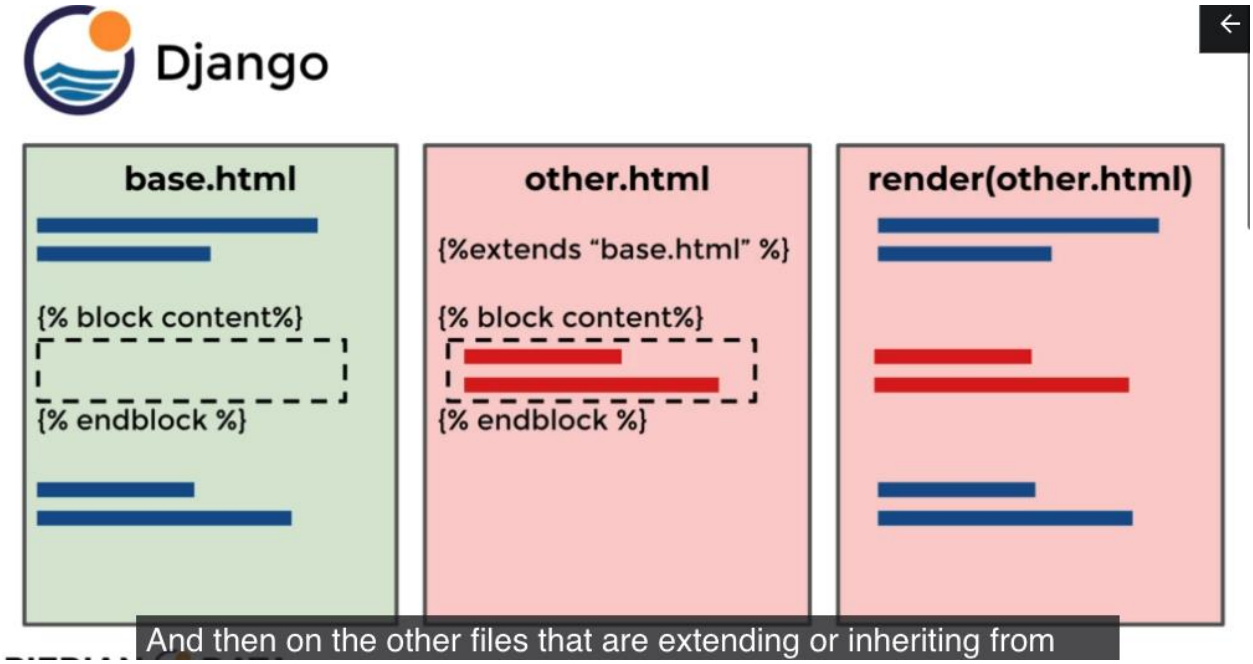
 # without app name

```
<a href="{% url 'example' %}">Click me to go to example</a>
```

23. Templates Inheritance

Typically, you don't want to have every single template hold repetitive information, such as the navigation bar at the top of your website.

Instead, we can **inherit** these components through the use of the Django **{%block%}** tag.



What is pretty common is to have a base that each HTML file on a project level and then base each HTML files on an application level, each of which actually extend or inherit from the project base that each HTML

Step One:

→ Create templates folder on project level
→ Create html file in template folder

Step Two:

→ **import os** in settings.py
→ enter in Template section 'DIRS': **[os.path.join(BASE_DIR, 'templates')]** in order to let Django search template files on project level

Step Three:

→ build generally content like navigation and so on, on base html for all the html sites
→ create content which should only be displayed on base.html site

```
{% block content %}
        …..
{% endblock %}
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>base title</title>
</head>
<body>
    <h1> This is above the block in base.html </h1>
    {% block content %} {# content be named anything else #}


    {# everthing here inside is blank #}


    {% endblock %}
    <h1> This is below the block in base.html </h1>
</body>
</html>
```

Step Four:

inheriting base.html content in my_app html sites

→ go to example.html (html on my_app level)
→ use extend commend in order to inherit the content from base.html **{% extends 'base.html')**

```
{% extends 'base.html' %}


{% block content %}
<h1> This is inside the block in example.html </h1>



{% endblock %}
```

Output on example.html

This is above the block in base.html

This is inside the block in example.html

This is below the block in base.html

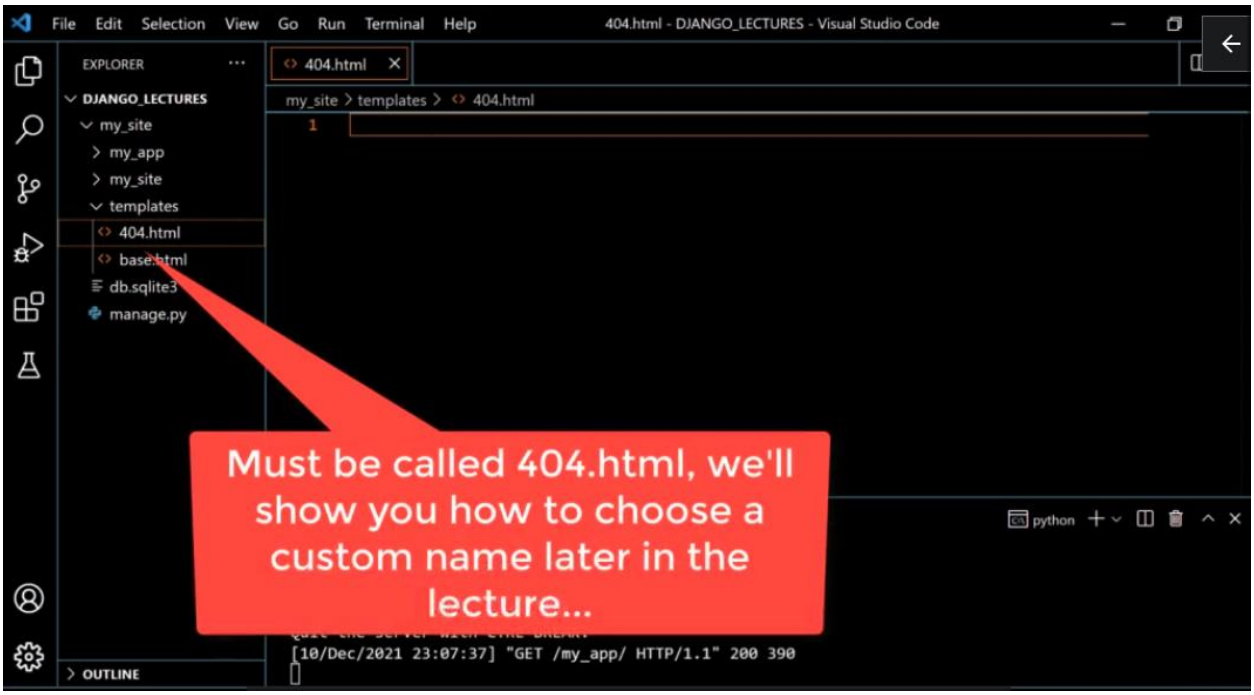
Base.html on project level → see 8. Function Based Views – Basics
See also → 14. Connecting a View to a Template for importing os and set the settings for connecting templates on project level

24. Custom Error Templates

Many pages, such admin or 404 pages have built-in templates provided by Django for your convenience

However, we have the ability to overwrite any of these built-in templates

Let's explore an example of this by overriding the default 404 template



Recommendation:

However, the best way to do it is calling the template 404 each HTML and setting up your handler!
→ so its handler 404, then the pathway handler404 = 'mysite.views.my_custom_page_not_found_view' to find your custom view and then you have a view specifically taking in that exception an registering that status with a particular HTML file

Steps:

→ Create 404.html file in templates folder project level

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>404 ERROR</title>
</head>
<body>
    <h1> Custom 404.html view -> Page not found LOL</h1>
</body>
</html>
```

→ Create views.py on project level

```
from django.shortcuts import render


def base_view(request):
    return render(request, 'base.html')


def my_custom_page_not_found_view(request, exception):

    return render(request, '404.html', status=404)
```

→ add handler404 in URLs project level under path

```
from django.contrib import admin
from django.urls import path, include
```

```
from . import views


urlpatterns = [
    path('admin/', admin.site.urls),
    path('my_app/', include('my_app.urls')),
    path('', views.base_view)
]

handler404 = 'step_site.views.my_custom_page_not_found_view'
```

→ setting DEBUG = False:

In order to show customized HTML when Django returns a 404, you can create an HTML template named 404.html and place it in the top level of your template tree. This template will then be served when DEBUG is set to False.

```
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []
```

→ DEBUG = FALSE

ALLOWD_HOST = [‘127.0.0.1’] # allow your host as show an error: CommandError: You must set settings.ALLOWED_HOSTS if DEBUG is False.

```
DEBUG = False

ALLOWED_HOSTS = [‘127.0.0.1’]
```

Output:

25. Static Files

Most projects will have static files, such as images, JS, or CSS

Django can serve these static files through the use of Tags, instead of having to refer to a full file path

This is similar to the {% url %} tag, but using a {% static %} tag

For this lecture you'll need to download or create some sort of static file

Our example will use a .jpg image, but the same logic applies to any static file

This is just a generic method to let your templates know the location of any static file (.css file, .js file, .jpg file, etc…)

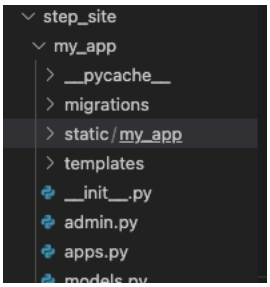
Steps:

→ On settings.py check if static has the path STATIC_URL = 'static/'

```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/4.1/howto/static-files/


# my_app/static/my_app/django.jpg
STATIC_URL = 'static/'
```

→ Create static folder in my_app (app level) and call static!
Hint: create another folder under static → my_app

- Static
    - my_app



→ Add for instance an image.jpg into static/my_app folder

→ link static path to a html file with
{% load static %}

<img src="{% static 'my_app/django.jpg' %}"…

Usable for any source… CSS file, JS file, JPEG, file etc.

Hint: best way to load all these files is to load it to the base.html (project level)

CSS file not found. Django project level

Check Your path again,

If it's correct Follow the Guidelines to Include CSS in Django Project
Static files are intended to wrap CSS files and your images, Django automatically identifies this file.

1. Create static folder in your app folder, same directory as of migrations and template folder
2. Create css Folder and insert it into static Folder
3. Now put your styles.css into css folder
4. Now in your HTML File where you want to include CSS, add {% load static %} On the top of HTML File and **Your Path should be like this** <link rel="stylesheet" href="{% static 'css/styles.css' %}"> in HTML file.
5. Then Make Change To Your settings.py in projectfoldername with-

STATIC_URL = '/static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR,'static')]

#STATIC_ROOT = os.path.join(BASE_DIR, 'assets') *nur wenn es nicht klappt*
 # You static file will be copied to New file created by django as assets.

Then Run this command

python manage.py collectstatic

```
 {% extends 'base.html' %}
{% load static %}


{% block content %}
<h1> This is inside the block in example.html </h1>
<img src="{% static 'my_app/django_reinhardt.jpeg' %}" alt="my image">


{% endblock %}
```

→ refresh the server and run the server again

Hint: If you change or edited anything in settings.py or you are doing something that has a lot do with settings.py just do yourself a favor and then restart the development server

Output:

This is above the block in base.html

This is inside the block in example.html

This is below the block in base.html

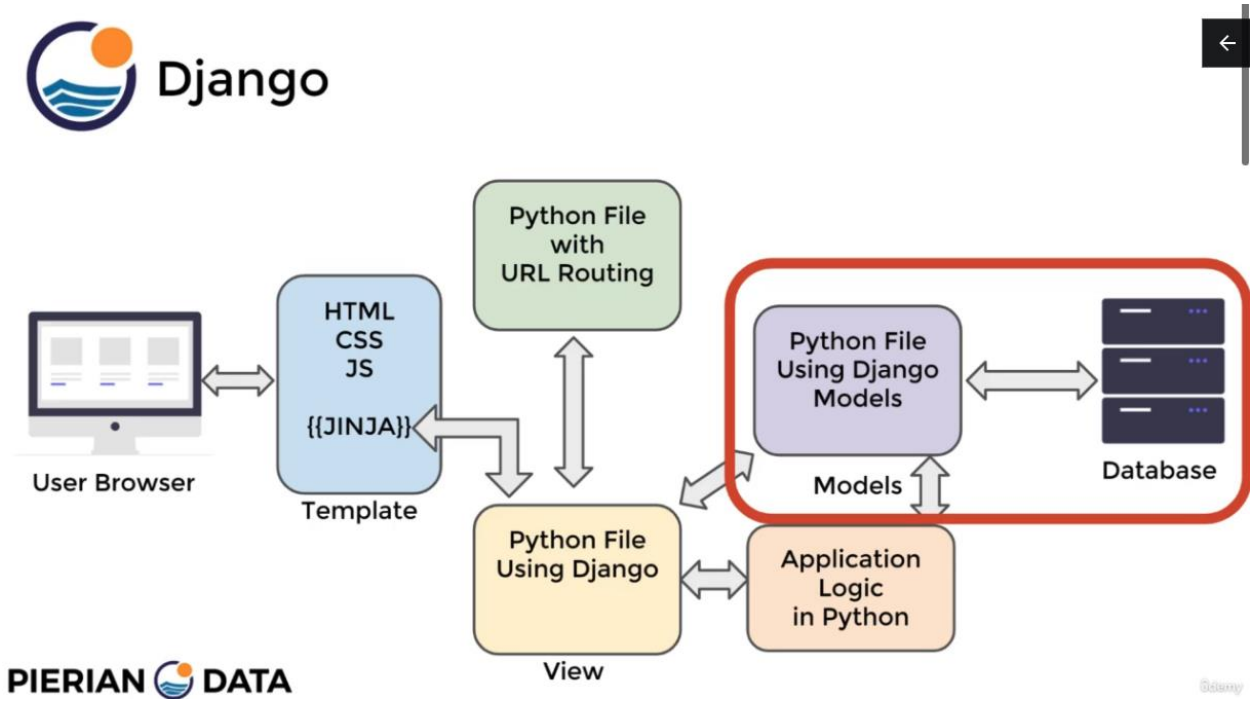
Section 11: Django – Models, Database, and Queries

26. Introduction to Models and Database

Models allow us to interact with a database with Python and Django

This includes the key interactions with database – CRUD:

- CREATE
- READ
- UDATE
- DELETE



In this section we'll be exploring how to store, retrieve, update, and delete data from a SQL based database using Django's built-in tools and functionality

Section Overview

- Database Overview
- Models and Database
- Creating Models and Fields
- Migration
- Data Interaction – CRUD
- Database and Template Interaction

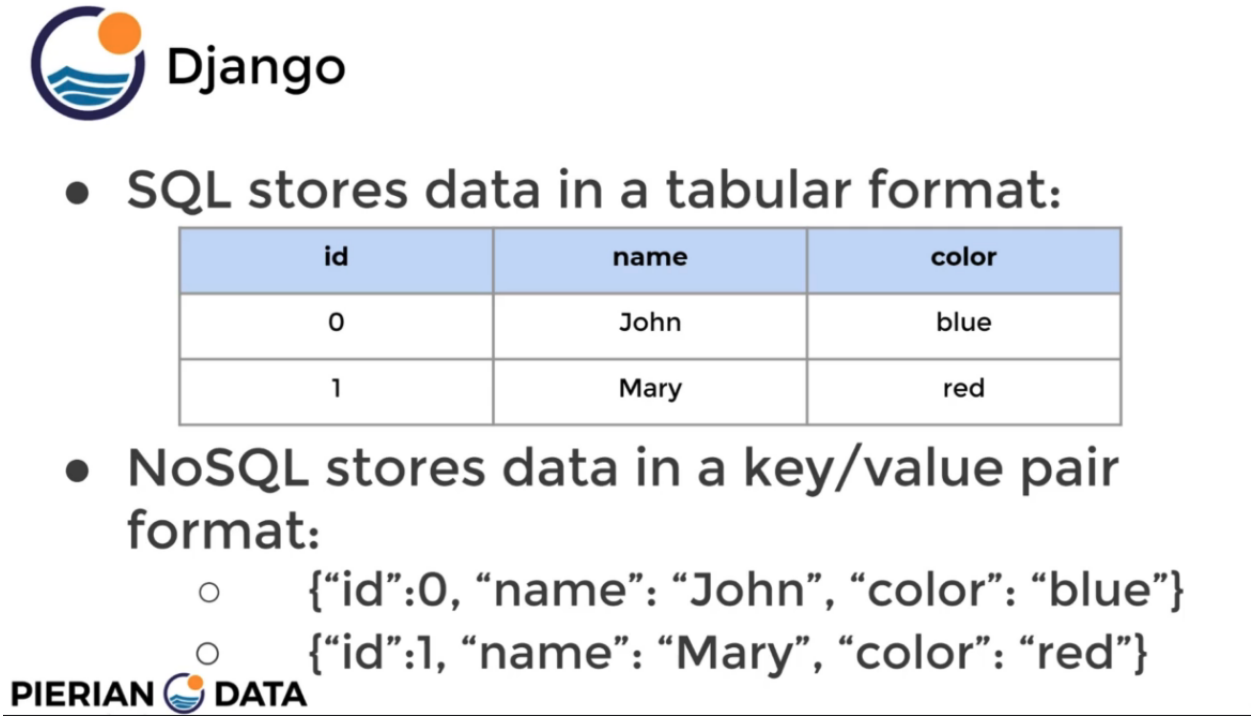
27. Databases Overview

Put simply, databases allow us to store information that we can use on our website.

We should briefly cover how SQL based databases work so we can understand the Django Models analogous operations that interact with the database.

SQL databases are tabular, similar to a spreadsheet, like Excel.

- But we often get questions like:
  "What about NoSQL, such as MongoDB?"
- Let's quickly explore the visual difference between two…



Which is better? NoSQL or SQL

- One format is not better than another, they are simply different
- Django Models is designed to work **really** well with a tabular SQL based format, so that is what we will choose

For most applications SQL will be completely fine, and if you are beginning with Django you should try to stick to SQL before making a jump to NoSQL

You should also carefully consider if NoSQL actually provides a major advantage to your project

Now, what type of SQL to choose?

- There are lots of options!
  - MySQL
  - SQLite
  - PostgreSQL
  - MS SQL
  - …and many more!

Django is pretty agnostic to most major SQL engines with the use of its Django Models system, so switching to another SQL engine is more matter of updating **settings.py** rather than rewriting the actual Python Django code.

Since this is the case, we'll use **SQLite** as its already included and installed along with Python

While typically thought of as a smaller scale SQL engine, for many uses cases SQLite performs fine.

- www.sqlite.org/whentouse.html

From the official website of SQLite:

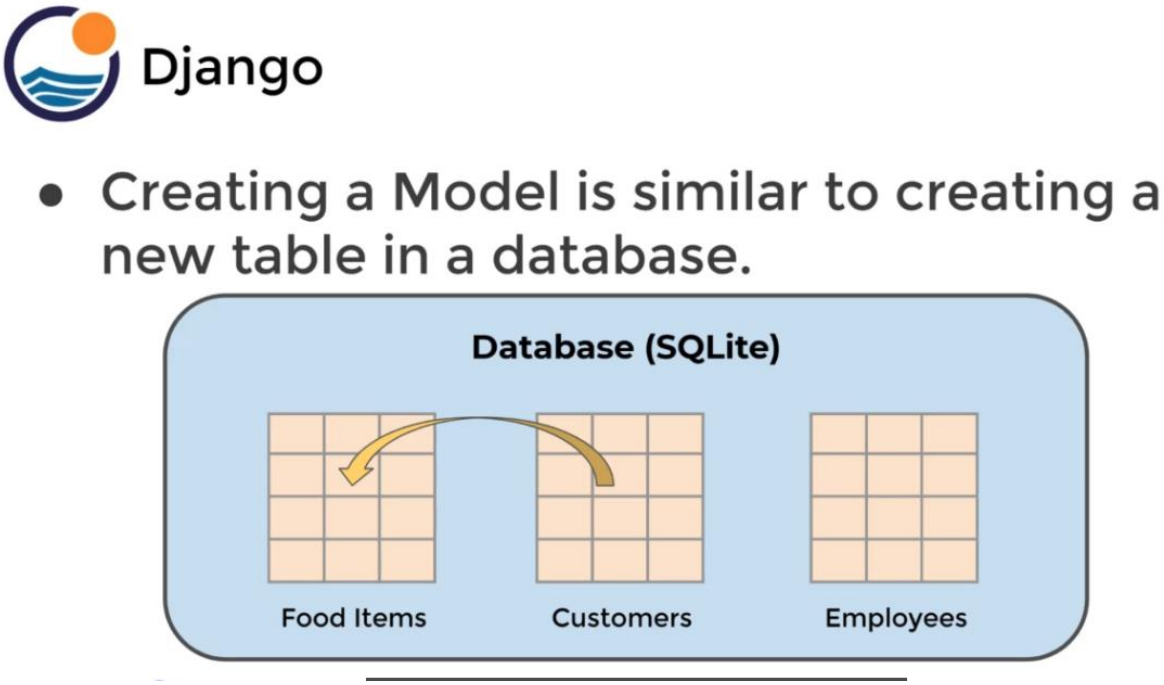
"Generally speaking, any site that gets fewer than 100k hits/day should work fine with SQLite. The 100k hits/day figure is a conservative estimate, not a hard upper bound. SQLite has been demonstrated to work with 10 time that amount of traffic."

28. Models and Database

Django Models are defined inside a Django app (or project) **models.py** file.

The models class operates on a system which directly converts Python based code into SQL commands

This makes it much easier to work with the backend database



Each database table has a name and then columns, where each column will have a specific data type, for example: **character strings** for names or **integers** for ages in years.

Let’s now explore how Django Models works in conjunction with these structures.

- Inside of **models.py**:

```python
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

- Automatically converted to SQL:

```sql
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

IERIAN DATA Again, you don't need to worry about that bottom.

Django Model Key Concepts:

- Inherits from **models** class.
- Uses **fields** to define both data **types** and data **constrains.**
    - For example, you may want to **require** information, like a user’s email address, in which case you can add a **NOT NULL** constraint.
    - You may want to **require** unique entries, like a unique user email (no duplicate accounts) with **UNIQUE** constraint.

- Fields are chosen for data type:

```python
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

- Field arguments specify constraints:

```python
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

- Django Models can also be connected through keys:

```python
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Let's continue our discussion by exploring how to create a model and add fields.

Later on in future section we'll be able to drastically advance our Django abilities by automatically creating templates simply by connecting them to a model (Class Based Views).

29. Models and Fields

In this lecture we will:
- Create a new project an app.
- Register a database in settings.
- Run a migrate command to create the database (**python manage.py migrate**).
- Create an example model with fields.

Let's imagine we've been hired by a dentist office to create some software for their office to keep track of patients.

It would be great if we could store patient information in a database, like their name and age.

To do this, we'll also be exploring the documentation as we code along, including:
- Setting up databases in settings.py.
- Different backend available.
- Different files that are available for Django Models.

Note, in the next lecture we'll continue with a further discussion on migrations, including the **makemigrations** command

Let's get started!

→ Start new Project → django-admin startproject dentist_site
→ Start new app → python manage.py startapp office

→ Exploring Database settings:

Engine can be changed for example to PostgreSQL
Default →

```
# Database
# https://docs.djangoproject.com/en/4.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

https://docs.djangoproject.com/en/4.1/ref/settings/#databases

Other Databases than SQLite

https://docs.djangoproject.com/en/4.1/ref/databases/

→ In order to run the database, it needs to create the database file

→ with migrate command, the database will create

→ **python manage.py migrate**

Migrate command is looking at the installed apps settings and create any necessary database tables according to the settings that you had under the database settings.

→ Django created db.sqlite3 file (actual database)

→ **Open models.py**

→ Create a database with model class

Hint: You can add more columns later on, but you always want to try to get as much information as possible at the very beginning of your project, even if you don't end up collecting it

Hint: You need a bit of experience to figure out what are the fields that are available for us, as well as what is the most

Under Model field reference there a lot of field including the field options and field types that Django offers.

https://docs.djangoproject.com/en/4.1/ref/models/fields/
With validators you can valid the input of the users. For example, nobody can be 1000 years old.

Validation will be introduced later on!

Simple model.py file:

```
from django.db import models


# Create your models here.


class Patient(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    age = models.IntegerField()
```

STEP 1 and STEP 2 are done!

STEP 3 → setting application on settings.py

30. Migration

In general, **migrations** is the act of connecting changes in your Django project or app to the database

This includes things like adding new models within an application, adding a new application, updating models with a new column/ attribute, and more.

You typically see these commands done through the manage.py file

Let's discuss the migrate based commands you can run:

- makemigrations
- migrate
- sqlmigrate

→ **python manage.py makemigration my_app**

- This actually creates (but does not run) the set of instruction that will apply changes to the database.
- Note, the default applications in Django (e.g. Admin, Auth) already have their SQL makemigrations code ready (just not run yet).
- You can actually see these migration files created under:
    - app
        - migrations
            - 0001_initial.py

→ **python manage.py migrate**

- Runs any existing migrations (typically created through the makemigrations command).
- This is actually running the files under the migrations directory from the previous command.

→ **python manage.py sqlmigrate app 0001**

- If you run makemigrations, then you've already created a migration.py code file
- If you wanted to see what the SQL code looked like, you could run the **sqlmigrate** command to view it
- Useful for things like debugging or interfacing with some sort of database admin

Note that typically we won't review the files created under the migrations directory or run **sqlmigrate**, we'll simply run **makemigrations** and **migrate**
You can think of the very first migrate command we run as executing the default **makemigrations** that was already created for your upon creating the project

Steps for migrations:

- Initial projects migrate command
- Create app and create models
- Register app in **INSTALLED_APPS** in **settings.py**
- Run **makemigrations** for new app
- Run **migrate** for new migrations

Let's explore these concepts based on the Patient model class created in the previous lecture!

→ Register app in INSTALLED_APPS in settings.py

```
# Application definition


INSTALLED_APPS = [
    'office.apps.OfficeConfig'
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
```

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

→ Run **makemigrations my_app**
→ python manage.py makemigrations office

Hint: under migration folder → 0001_initial.py created by Django for creating the actual SQL code to interact the Database (Possible for editing)
In order to see what Django has created in SQL use python manage.py sqlmigrate office 0001

```
    - Create model Patient
(base) giales@Gianlucas-Air-2 dentist_site % python manage.py sqlmigrate office 0001
BEGIN;
--
-- Create model Patient
--
CREATE TABLE "office_patient" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "first_name" varchar(30
) NOT NULL, "last_name" varchar(30) NOT NULL, "age" integer NOT NULL);
COMMIT;
```

→ **run migrate again** to run all new data which aren't running yet
→ python manage.py migrate

```
(base) giales@Gianlucas-Air-2 dentist_site % python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, office, sessions
Running migrations:
  Applying office.0001_initial... OK
(base) giales@Gianlucas-Air-2 dentist_site %
```

31. Data Interaction: Creating and Inserting

Inserting new data into a SQL table is easy with Django Models

Since the models are represented by a class, we can easily create a new instance of the class object in Python, and then call the **.save()** method to create an INSERT call to the SQL database.

Alternatively, you can use the built-in .objects.create() method to both create and save the new data entry in a single line.

In instances where you want to create multiple new data entries in bulk, you can use the .objects.bulk_create() method to pass in a list of newly created objects.

Let's explore these 3 methods of creating new data entries with a model:

- Create Object and .save()

- objects.create()
- objects.bulk_create()

https://docs.djangoproject.com/en/4.1/ref/models/querysets/#methods-that-do-not-return-querysets

→ run shell command
→ **python manage.py shell**

User interface interaction with an operating system by using PYTHON

Creating and saving Instance of the object in two steps:

→ create instance: carl = Patient(first_name='carl', last_name='smith', age=30)
→ save instance in database carl.save()

```
hello

In [2]: from office.models import Patient

In [3]: carl = Patient(first_name='carl',last_name='smith',age=30)

In [4]: carl.age
Out[4]: 30

In [5]: carl.age < 20
Out[5]: False

In [6]: carl.save()

In [7]:
```

```
(base) giales@Gianlucas-Air-2 dentist_site % python manage.py shell
Python 3.9.12 | packaged by conda-forge | (main, Mar 24 2022, 23:24:38)
[Clang 12.0.1 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> print(1)
1
>>> from office.models import Patient
>>>  carl = Patient(first_name='carl', last_name='smith', age=30)
  File "<console>", line 1
    carl = Patient(first_name='carl', last_name='smith', age=30)
IndentationError: unexpected indent
>>>  carl = Patient(first_name='carl',last_name='smith',age=30)
  File "<console>", line 1
    carl = Patient(first_name='carl',last_name='smith',age=30)
IndentationError: unexpected indent
>>> carl = Patient(first_name='carl',last_name='smith',age=30)
>>> carl.age < 30
False
>>> carl.save()
>>>
```

→ **creating and saving in database at the same time with objects.create()**
→ Patient.objects.create

```
>>> Patient.objects.create(first_name='susan', last_name='smith', age=40)
<Patient: Patient object (2)>
>>>
```

→ 2 Instances/ Patients are now saved in Database
→ create and saved another Patient

```
>>> Patient.objects.create(first_name='susan', last_name='smith', age=40)
<Patient: Patient object (2)>
>>> Patient.objects.create(first_name='mimi', last_name='bolus', age=36)
<Patient: Patient object (3)>
>>>
```

→ bulk_create() in order to create many Patient at the same time

https://docs.djangoproject.com/en/4.1/ref/models/querysets/#bulk-create

Hint: there are many caveats!

→ The way how to create many Patients at the same time:

>>> mylist=[Patient(first_name='adam', last_name='smith', age=50), Patient(first_name='paul', last_name='marx', age=15)]

>>> Patient.objects.bulk_create(mylist)

→ Patient 4 and 5 are created!

```
>>> Patient.objects.create(first_name='susan', last_name='smith', age=40)
<Patient: Patient object (2)>
>>> Patient.objects.create(first_name='mimi', last_name='bolus', age=36)
<Patient: Patient object (3)>
>>> mylist=[Patient(first_name='adam', last_name='smith', age=50), Patient(first_name='paul', l
ast_name='marx', age=15)]
>>> Patient.objects.bulk_create(mylist)
[<Patient: Patient object (4)>, <Patient: Patient object (5)>]
>>>
```

32. Data Interaction: Using .all() Reading and Querying

Each Model you create comes with a **Manager** that allows you to create a **QuerySet** which can then be used to retrieve entries from the database

Keep in mind that the QuerySet is actually **lazily evaluated,** meaning that it doesn't hit the database until its explicitly asked to grab the information

Recall we used something like:

- **MyModel.objects**
- This is the Django Model Manager
- This Manager can then actually read the database through the use of method calls, like **.all()** and **.get()** and narrow down results with **.filter()** and **.exclude()**

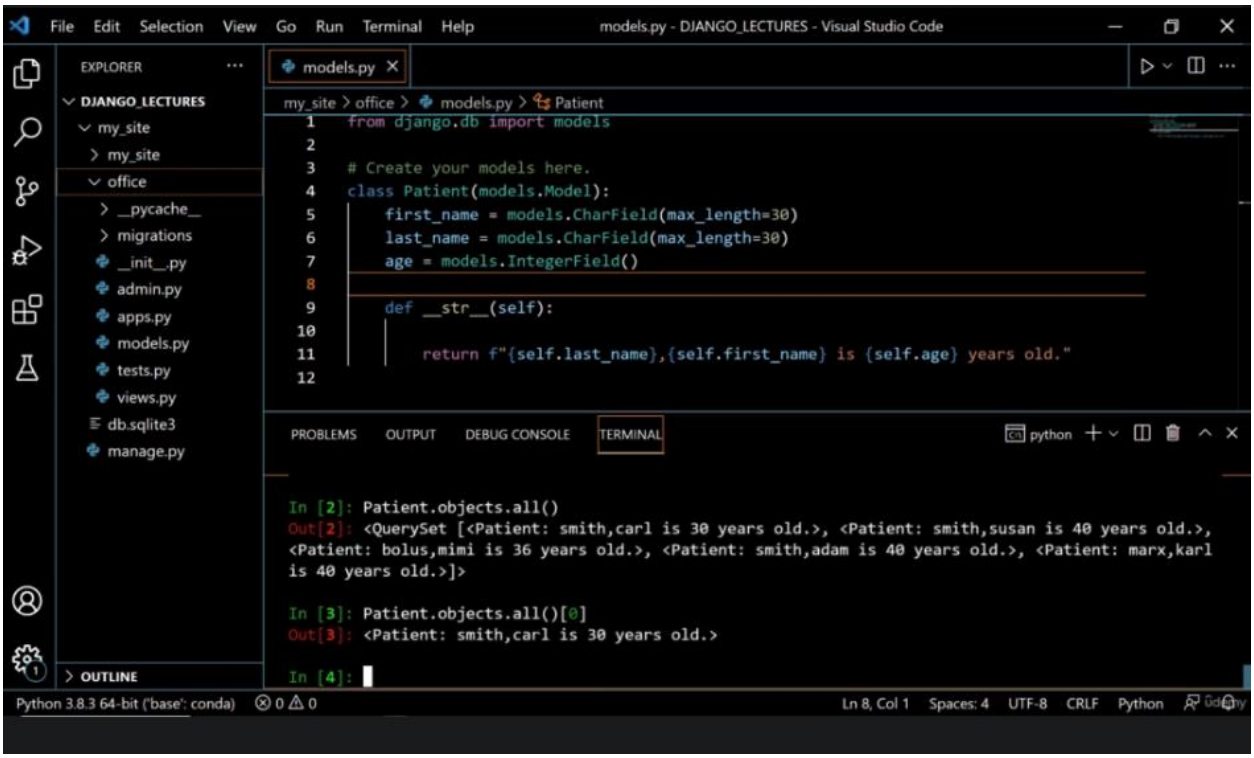
In this lecture we're going to focus on the **.all()** method which allows us to grab all the entries in a database table.

Typically, we won't want everything, so we'll need to filter our results, but we'll discuss that in more detail later on.

Great resource on queries with examples:

- **docs.djangoproject.com/en/4.1/topics/db/queries/**

For now, let's explore the basics of reading in data from the database and what changes we can make to the model to make results more human readable

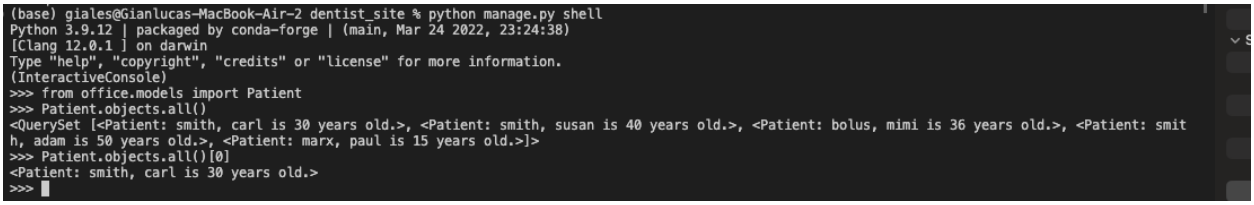


→ python manage.py shell

→ from office.models import Patient

→ Patient.objects.all()
→ Patient.objects.all()[0] # showing only first entry

For readable entries add to models.py → def __str__(self): f"{self.last_name}, {self.first_name} is {self.age} years old"



33. Data Interaction: Filtering filter() and get()

The **.get()** operation allows us to grab a **single** item from the Model table

This is typically reserved for something where you are sure there is only a single unique entry, like the default primary key that is automatically created by Django **(pk=N).**

If we want to further filter our results (rather than grab all or get a single item), we can use the **.filter()** method to narrow down based on conditions.

The **.filter()** method can be chained together.

Django also provides **operators** for QuerySets, which allow us to directly use logic like **AND** and **OR.**

These operators need to be imported from **django.db.models** from the **Q** function

Let's explore the following topics:

- Using **.get()**
- Using **.filter()**
- Using **operators**

Get() and Filter():

→ Patient.objects.get(pk=1)

Hint: SQL Table begins with 1 (Python list with 0)

Error will occur if item exist more than 1 as the get method can only return a single object!

→ Patient.objects.filter(last_name="smith").all() or
Patient.objects.filter(last_name="smith")

→ Double filter: Patient.objects.filter(last_name=”smith”).filter(age=30).all() or Patient.objects.filter(last_name=”smith”).filter(age=30)

```
>>> Patient.objects.get(pk=1)
<Patient: smith, carl is 30 years old.>
>>> Patient.objects.filter(last_name="smith")
<QuerySet [<Patient: smith, carl is 30 years old.>, <Patient: smith, susan is 40 years
old.>, <Patient: smith, adam is 50 years old.>]>
>>> Patient.objects.filter(last_name="smith").all()
<QuerySet [<Patient: smith, carl is 30 years old.>, <Patient: smith, susan is 40 years
old.>, <Patient: smith, adam is 50 years old.>]>
>>> Patient.objects.filter(last_name="smith").filter(age=40)
<QuerySet [<Patient: smith, susan is 40 years old.>]>
>>> Patient.objects.filter(last_name="smith").filter(age=40).all()
<QuerySet [<Patient: smith, susan is 40 years old.>]>
```

Operators:

→ from django.db.models import Q

→ allow us to use: OR → I and AND → & operators

https://docs.djangoproject.com/en/4.1/topics/db/queries/#complex-lookups-with-q-objects

→ Patient.objects.filter(Q(last_name=’smith’) & Q(age=40)).all() or Patient.objects.filter(Q(last_name=’smith’) & Q(age=40))

```
>>> from django.db.models import Q
>>> Patient.objects.filter(Q(last_name='smith')& Q(age= 40)).all()
<QuerySet [<Patient: smith, susan is 40 years old.>]>
>>> Patient.objects.filter(Q(last_name='smith')& Q(age= 40))
<QuerySet [<Patient: smith, susan is 40 years old.>]>
>>>
```

```
>>> Patient.objects.filter(Q(last_name='smith')& Q(age= 30) |  Q(age=40))
<QuerySet [<Patient: smith, carl is 30 years old.>, <Patient: smith, susan is 40 years
old.>]>
>>> Patient.objects.filter(Q(last_name='smith')& Q(age= 30) |  Q(age=40)).all()
<QuerySet [<Patient: smith, carl is 30 years old.>, <Patient: smith, susan is 40 years
old.>]>
>>>
```

34. Data Interaction: Filtering with Field lookups

So far we’ve had to use equality statements in our filtering (age=30 or last_name=’smith’)

But what about more general comparison operators:

- Greater than or less than
- Starts with

For more complex filtering operations we use **field lookups** with a **filter()** call:

Model.objects.filter(name__startswith="s")

https://docs.djangoproject.com/en/4.1/topics/db/queries/#field-lookups

→ Patient.objects.filter(last_name__startswith='s').all()
→ Patient.objects.filter(age__in=[20,30,40]).all()

```
>>> Patient.objects.filter(last_name__startswith='s').all()
<QuerySet [<Patient: smith, carl is 30 years old.>, <Patient: smith, susan is 40 years
old.>, <Patient: smith, adam is 50 years old.>]>
>>> Patient.objects.filter(last_name__startswith='s')
<QuerySet [<Patient: smith, carl is 30 years old.>, <Patient: smith, susan is 40 years
old.>, <Patient: smith, adam is 50 years old.>]>
>>> Patient.objects.filter(age__in=[20,30,40]).all()
<QuerySet [<Patient: smith, carl is 30 years old.>, <Patient: smith, susan is 40 years
old.>]>
>>> Patient.objects.filter(age__in=[20,30,40])
<QuerySet [<Patient: smith, carl is 30 years old.>, <Patient: smith, susan is 40 years
old.>]>
>>>
```

→ in = in a given iterable; often a list, tuple, or queryset
→ gte = greater than or equal to → Patient.objects.filter(age_gte=39).all()
→ gt = greater than
→ lt = less than
→ lte = less than or equal to
→ startswith

More: **https://docs.djangoproject.com/en/4.1/ref/models/querysets/#field-lookups**

Methods that return new QuerySets:

https://docs.djangoproject.com/en/4.1/ref/models/querysets/#exact

→ order_by

order_by(*fields)

By default, results returned by a QuerySet are ordered by the ordering tuple given by the ordering option in the model's Meta. You can override this on a per-QuerySet basis by using the order_by method.

Example:

Entry.objects.filter(pub_date__year=2005).order_by('-pub_date', 'headline')

The result above will be ordered by pub_date descending, then by headline ascending. The negative sign in front of **"-pub_date" indicates descending** order. **Ascending order is implied**. To order randomly, use "?", like so:

→ Entry.objects.order_by('?')

→ Patient.objects.order_by('age').all()

```
>>> Patient.objects.order_by('age').all()
<QuerySet [<Patient: marx, paul is 15 years old.>, <Patient: smith, carl is 30 years ol
d.>, <Patient: bolus, mimi is 36 years old.>, <Patient: smith, susan is 40 years old.>,
 <Patient: smith, adam is 50 years old.>]>
>>> Patient.objects.order_by('age')
<QuerySet [<Patient: marx, paul is 15 years old.>, <Patient: smith, carl is 30 years ol
d.>, <Patient: bolus, mimi is 36 years old.>, <Patient: smith, susan is 40 years old.>,
 <Patient: smith, adam is 50 years old.>]>
>>>
```

35. Data Interaction: Updating Models

There may come a time when you need to create a new column or attribute for a model

You can easily update existing models by simply adding a new model class attribute and then migrating those changes

You should note that when adding new fields, the existing entries will need to have some default value inserted (even if it's just null)

In fact, when we attempt to run migrations without taking care of these issues, Django will specifically request us to make a decision

You'll be given two options:

- Choose a default value on the spot when making the migrations file
- Cancel the migration and create a default value within the model

It's usually more robust to have the default live in the model, but each case is different

Let's explore these ideas further in our code, we'll also touch on the idea of using **validators** with fields, which add hard-coded constraints that will rejects non valid entries.

→ **adding new attributes to the table**

Hint: it needs a default, since data already exist in the database
→ here the Patient doesn't have any heart rate value

Django will occur an error:

It is impossible to add a non-nullable field 'heartrate' to patient without specifying a default. This is because the database needs something to populate existing rows.
Please select a fix:
1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
2) Quit and manually define a default value in models.py.
Select an option:
→ option 2: for transparency reasons it is better to put everything in models.py
→ example heartrate = models.IntegerField(default=60)

```python
from django.db import models


# Create your models here.


class Patient(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    age = models.IntegerField()
    heartrate = models.IntegerField(default=60)


    def __str__(self):


        return f"{self.last_name}, {self.first_name} is {self.age} years old."
```

→ python manage.py makemigrations office
→ new python file will create for the changes! → 0002_patient_heartrate

Hint: This file is depended with 0001_initials.py

Validators:

https://docs.djangoproject.com/en/4.1/ref/validators/#built-in-validators

→ from django.core.validators import MaxValueValidator, MinValueValidator
→ …validators=[built_in validators]

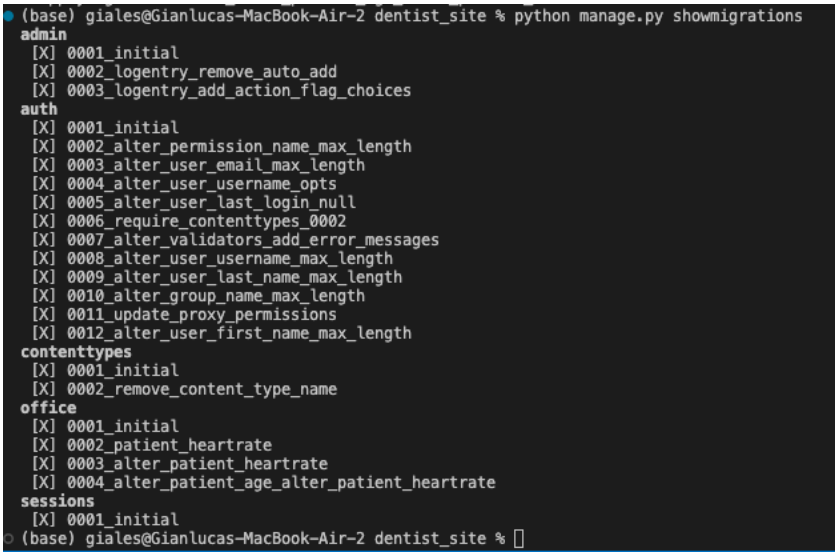
```python
from django.db import models
from django.core.validators import MaxValueValidator, MinValueValidator


# Create your models here.


class Patient(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    age = models.IntegerField(validators=[MinValueValidator(0), MaxValueValidator(120)])
    heartrate = models.IntegerField(default=60, validators=[MinValueValidator(1), MaxValueValidator(300)])


    def __str__(self):


        return f"{self.last_name}, {self.first_name} is {self.age} years old."
```

→ python manage.py makemigrations office
→ python manage.py migrate

Hint: With *python manage.py showmigrations* you can see all migrations and which ones are already migrated and which ones are not

```
(base) giales@Gianlucas-MacBook-Air-2 dentist_site % python manage.py showmigrations
admin
 [X] 0001_initial
 [X] 0002_logentry_remove_auto_add
 [X] 0003_logentry_add_action_flag_choices
auth
 [X] 0001_initial
 [X] 0002_alter_permission_name_max_length
 [X] 0003_alter_user_email_max_length
 [X] 0004_alter_user_username_opts
 [X] 0005_alter_user_last_login_null
 [X] 0006_require_contenttypes_0002
 [X] 0007_alter_validators_add_error_messages
 [X] 0008_alter_user_username_max_length
 [X] 0009_alter_user_last_name_max_length
 [X] 0010_alter_group_name_max_length
 [X] 0011_update_proxy_permissions
 [X] 0012_alter_user_first_name_max_length
contenttypes
 [X] 0001_initial
 [X] 0002_remove_content_type_name
office
 [X] 0001_initial
 [ ] 0002_patient_heartrate
 [ ] 0003_alter_patient_heartrate
 [ ] 0004_alter_patient_age_alter_patient_heartrate
sessions
 [X] 0001_initial
(base) giales@Gianlucas-MacBook-Air-2 dentist_site %
```

After → python manage.py migrate



36. Data Interaction: Updating Entries

Django makes it very easy to update existing entries

You simply grab the existing data entry, update any attributes, then .save() the changes to write the update to the database table

Let's check it out

→ from my_app.models import Object
→ Assign entry to a value with Entry.objects.get(pk=1)
→ Change value with the following attribute → value.last_name='name'
→ value.save() for saving it at the database

Example:

→ python manage.py shell
→ from office.models import Patient
→ Patient.objects.get(pk=1)
→ carl = Patient.objects.get(pk=1)
→ carl
→ carl.last_name
→ carl.last_name = 'django'
→ carl
→ carl.save()
→ Patient.objects.all()

```
(base) giales@Gianlucas-MacBook-Air-2 dentist_site % python manage.py shell
Python 3.9.12 | packaged by conda-forge | (main, Mar 24 2022, 23:24:38)
[Clang 12.0.1 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from office.models import Patient
>>> Patient.objects.get(pk=1)
<Patient: smith, carl is 30 years old.>
>>> carl = Patient.objects.get(pk=1)
>>> carl
<Patient: smith, carl is 30 years old.>
>>> carl.last_name
'smith'
>>> carl.last_name = 'django'
>>> carl
<Patient: django, carl is 30 years old.>
>>> carl.save()
>>> Patient.objects.all()
<QuerySet [<Patient: django, carl is 30 years old.>, <Patient: smith, susan is 40 years old.>, <Patient: bolus,
mimi is 36 years old.>, <Patient: smith, adam is 50 years old.>, <Patient: marx, paul is 15 years old.>]>
>>>
```

37. Data Interaction: Deleting

→ .delete()

```
>>> Patient.objects.all()
<QuerySet [<Patient: django, carl is 30 years old.>, <Patient: smith, susan is 40 years old.>, <Patient: bolus,
mimi is 36 years old.>, <Patient: smith, adam is 50 years old.>, <Patient: marx, paul is 15 years old.>]>
>>> data_point = Patient.objects.get(pk=1)
>>> data_point
<Patient: django, carl is 30 years old.>
>>> data_point.delete()
(1, {'office.Patient': 1})
>>> Patient.objects.all()
<QuerySet [<Patient: smith, susan is 40 years old.>, <Patient: bolus, mimi is 36 years old.>, <Patient: smith, a
dam is 50 years old.>, <Patient: marx, paul is 15 years old.>]>
>>> Patient.objects.get(pk=1)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/Users/giales/micromamba/lib/python3.9/site-packages/django/db/models/manager.py", line 85, in manager_m
ethod
    return getattr(self.get_queryset(), name)(*args, **kwargs)
  File "/Users/giales/micromamba/lib/python3.9/site-packages/django/db/models/query.py", line 650, in get
    raise self.model.DoesNotExist(
office.models.Patient.DoesNotExist: Patient matching query does not exist.
>>>
```

38. Connecting Templates and Database Models

You have a lot of the knowledge and tools needed to make a fully functioning website!

You know all the details necessary to connect a user to a backend database and have the user interact with the data (create, read, update and delete).

In this lecture, we'll explore how we could report back information from the database to the user in a template

However, we want to keep in mind there are two major ideas we have yet to learn about…

Two Major Features:

- **Django Forms**
  - Allows Django to automatically create forms from Python to template

- **Class Based Views**
    - Automatically generates views based on a Model

These two features are so powerful that you should really learn them first *before* jumping straight into using Django based on what you know so far

While we have a lot of capabilities already, **those features will drastically reduce your development time!**

For now let's show a simple example of a template that could be used report back information from a database

Steps:
One:

→ create urls.py file on app level

Two:
On urls.py project level

→ import include and connect urls.py app level with urls.py project level
→ path('office'/, include('office.urls'))

```
from django.urls import path, include


urlpatterns = [
    path('admin/', admin.site.urls),
    path('office/', include('office.urls'))
]
```

Three:
On views.py app level

→ import: from . import **models**
→ create function for the database list

```
from django.shortcuts import render
from . import models


# Create your views here.


def list_patients(request):
```

```
    return 'nothing'
```

Four:
On urls.py app level

→ import: from django.urls import path
→ from . import views
→ add path from function list_patients and give it a name

```
from django.urls import path
from . import views



# domain.com/office --->
urlpatterns = [
    path('',views.list_patients, name='list_patients')


]
```

Five:
On views.py app level

→ connect model object to views.py. with models.Patient.objects.all()
→ continue with function and assign object to a variable
→ use dictionary to send a context variable to the HTML → look at *17. Variables in Templates* for more information

```
from django.shortcuts import render
from . import models


# Create your views here.


def list_patients(request):


    all_patients = models.Patient.objects.all()


    context_list = {'patients' : all_patients}


    return render(request, 'office/list.html', context = context_list) # placeholder html doesn't exist yet
```

Six:
Create templates folder in app folder and html file to templates/office

→ templates
- Office
  - list.html

Seven:
Add content in html file

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    {{patients}}
</body>
</html>)
```

Eight:
→ runserver
→ http://127.0.0.1:8000/office/

Output:

<QuerySet [<Patient: smith, susan is 40 years old.>, <Patient: bolus, mimi is 36 years old.>, <Patient: smith, adam is 50 years old.>, <Patient: marx, paul is 15 years old.>]>

In a list:

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
```

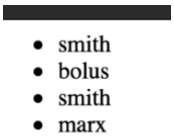
```
<body>
   <ul>
   {% for person in patients %}

   <li>{{person}}</li>

   {% endfor %}
   </ul>
</body>
</html>
```

Output:

- smith, susan is 40 years old.
- bolus, mimi is 36 years old.
- smith, adam is 50 years old.
- marx, paul is 15 years old.

Only last_name:

```
<li>{{person.last_name}}</li>
```



- smith
- bolus
- smith
- marx

Section 12: Django – Admin

39. Introduction to Django Admin Section

One of the most powerful features of Django is its ability to automatically create an Admin interface.

This is a feature meant to be used by the website manager, to have a graphical interface for interacting with data and users on the site.

We've already seen pre-built admin paths in our urls.py file ("/admin") as well as indications of an existing Django Admin app ("**django.contrib.admin"**).

In this section we'll explore how to access the admin panel view and how to configure admin settings.

Keep in mind that the admin panel is really meant for a manager of the website, we won't expect normal users to access the Django Administration interface.

Section Overview:
- Review of Website with Models
- Accessing Django Admin Panel
- Connecting Models to Admin

40. Model and Website – Part one

Let's create an online website that sells cars!

We'll need to create a very simple landing page that can report what cars we have in stock, and we'll also want a way to add or remove cars from our inventory.

Everything shown in this lecture should feel like a review of past concept, we won't introduce the admin panel yet!

To do:

- Create Templates
- Create Views
- Connect with URLs
- Create Model for Cars
- Test CRUD functionality

→ Create new Project my_car_site
→ Create new app cars
→ Create Templates (project level and app level)
→ Create HTMLs files and base.html in templates project level
→ Create Views (Function based View, render html files)

```
from django.shortcuts import render


# Create your views here.


def list_view(request):
    return render(request, 'cars/list.html')
```

```
def add_view(request):
    return render(request, 'cars/add.html')

def delete_view(request):
    return render(request, 'cars/delete.html')
```

→ Connect with URLs (create urls.py file on app level, add app_name to urls.py app level)

App level:

```
from django.urls import path
from . import views

app_name = 'cars'

urlpatterns = [
    path('list/', views.list_view, name = 'list'),
    path('add/', views.add_view, name = 'add'),
    path('delete/', views.delete_view, name = 'delete')
]
```

Project level:

```
"""my_car_site URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/4.1/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
```

```
"""
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('cars/', include('cars.urls'))
]
```

→ Set settings:

- import os and set templates path in settings.py
- add path for new installed app

```
from pathlib import Path
import os
```

```
INSTALLED_APPS = [
    'cars.apps.CarsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR,'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
```

```
    },
  },
]
```

→ Run Migration
→ **python manage.py migration**

→ Set up base.html and link it to the app level html (Set up bootstrap Navigation bar)

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <!-- CSS only -->
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1WTRi"
crossorigin="anonymous">
  <!-- JavaScript Bundle with Popper -->
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-
OERcA2EqjJCMA+/3y+gxIOqMEjwtxJY7qPCqsdltbNJuaOe923+mo//f6V8Qbsw3"
crossorigin="anonymous"></script>
  <title>Cars Project</title>
</head>
<body>
  <nav class="navbar navbar-dark bg-dark">
    <div class="container-fluid">
      <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-
controls="navbarNav" aria-expanded="false" aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarNav">
        <ul class="navbar-nav">
          <li class="nav-item">
            <a class="nav-link" href="{% url 'cars:list' %}">List</a>
          </li>
          <li class="nav-item">
```

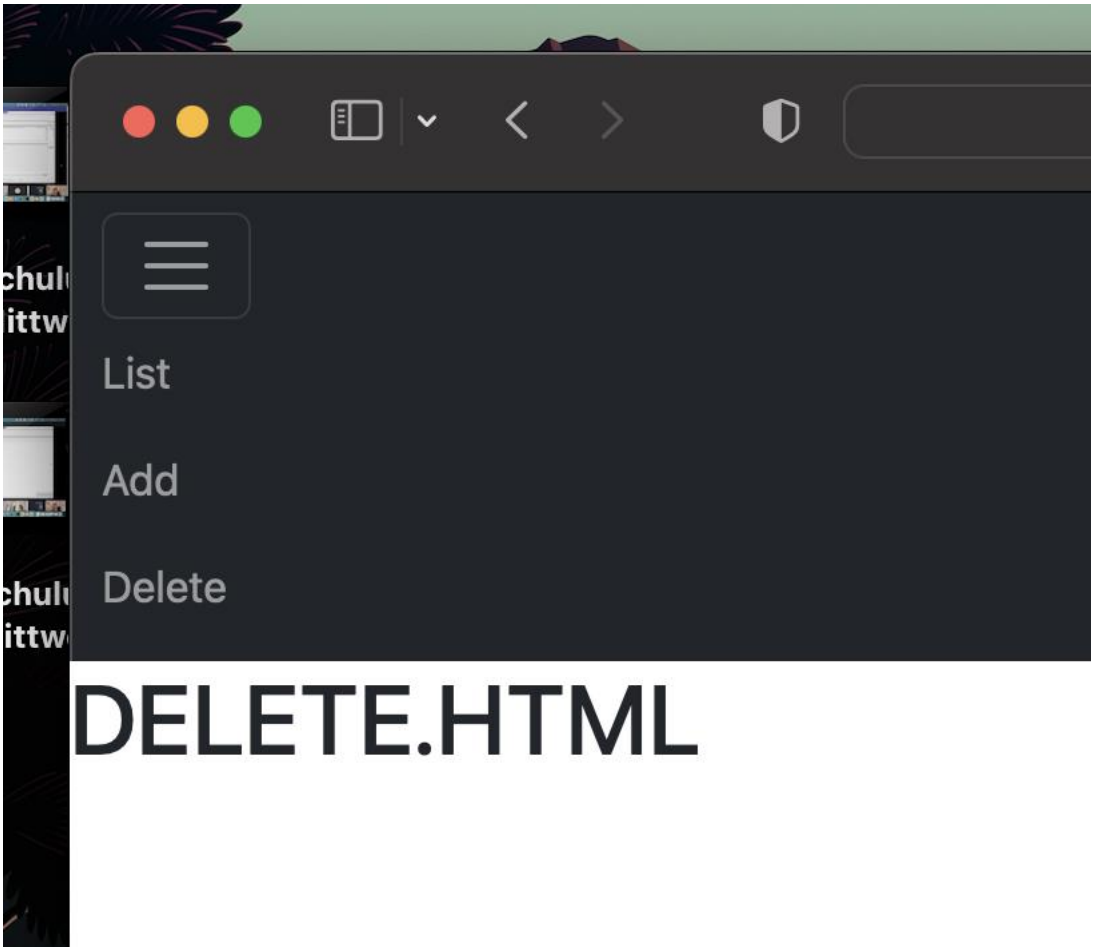
```
            <a class="nav-link" href="{% url 'cars:add' %}">Add</a>
          </li>
          <li class="nav-item">
            <a class="nav-link" href="{% url 'cars:delete' %}">Delete</a>
          </li>
        </ul>
      </div>
    </div>
  </nav>

  {# Block content #}
  {% block content %}


  {% endblock %}
</body>
</html>
```

→ Add {% extends 'base.html' %} to the other .html files

Example: add.html

```
{% extends 'base.html' %}

{% block content %}

<h1>ADD.HTML</h1>

{% endblock %}
```

Output:



41. Model and Website: Part Two

So far, we've completed:

- Basic Views
- Basic URL Routing
- General NavBar
- Empty Templates:
    - List
    - Add
    - Delete

Now we will:

- Create a Car Model.
- Add Functionality to Views connecting to model.
- Create HTML forms to send information back to the Views.

Important Note!

The methods shown here using “raw” HTML forms to accept user information regarding a model in the database are not ideal and later on we will learn about the Django Form system which is much better!

→ Create Car Model (models.py)

```
from django.db import models


# Create your models here.
class Car(models.Model):
    # pk
    brand = models.CharField(max_length = 30)
    year = models.IntegerField()


    def __str__(self):
        return f"Car is {self.brand} {self.year}"
```

→ Make migration
→ **python manage.py makemigrations cars**

Hint: Also, possibly make migration and run migrate just once after creating models

→ Run migration
→ **python manage.py migration**

→ Add model in views.py

```
from django.shortcuts import render
from . import models



# Create your views here.


def list_view(request):
    all_cars = models.Car.objects.all()
    context = {'all_cars':all_cars}


    return render(request, 'cars/list.html', context = context)


def add_view(request):
    return render(request, 'cars/add.html')
```

```
def delete_view(request):
    return render(request, 'cars/delete.html')
```

→ update list.html

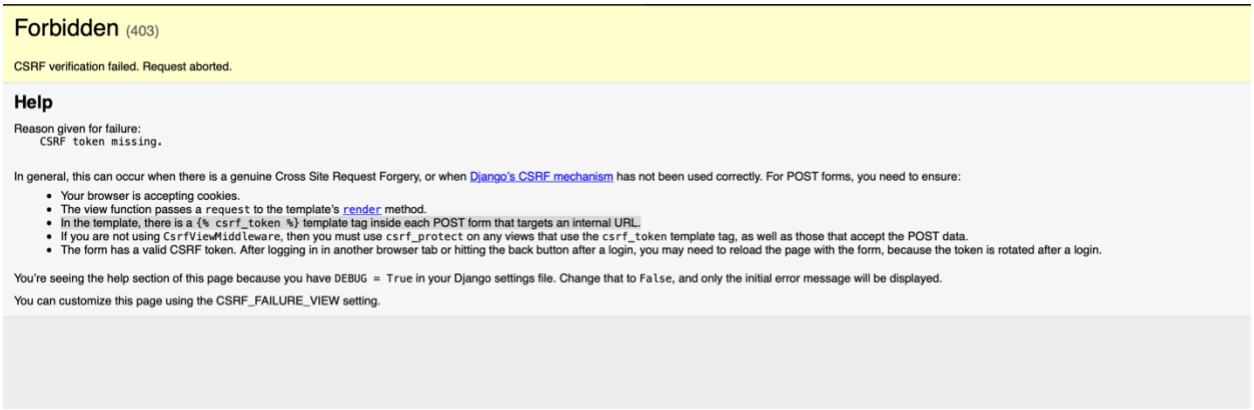
```
{% extends 'base.html' %}

{% block content %}

<div class='container'>
    <h1>LIST.HTML</h1>
        <ul>
            {% for car in all_cars %}
                <li>{{cars}}</li>
            {% endfor %}
        </ul>
</div>

{% endblock %}
```

→ update add.html and use a form

```
{% extends 'base.html' %}

{% block content %}

<div class='container'>

    <h1>ADD.HTML</h1>
    <h2>Add a new car to the database:</h2>
    <form action="" method="POST">
        <div>
            <label for="brand">Brand:</label>
            <input type="text" id="brand" name="brand">
        </div>
        <div>
```

```
        <label for="year">Year</label>
        <input type="text" id="year" name="year">
    </div>
    <input type="submit">
  </form>
{% endblock %}
```

→ Use {% csrf_token %} because Django thinks that is an attacking attack:
See error:



Forbidden (403)

CSRF verification failed. Request aborted.

Help

Reason given for failure:
`CSRF token missing.`

In general, this can occur when there is a genuine Cross Site Request Forgery, or when Django's CSRF mechanism has not been used correctly. For POST forms, you need to ensure:

- Your browser is accepting cookies.
- The view function passes a `request` to the template's `render` method.
- In the template, there is a `{% csrf_token %}` template tag inside each POST form that targets an internal URL.
- If you are not using `CsrfViewMiddleware`, then you must use `csrf_protect` on any views that use the `csrf_token` template tag, as well as those that accept the POST data.
- The form has a valid CSRF token. After logging in in another browser tab or hitting the back button after a login, you may need to reload the page with the form, because the token is rotated after a login.

You're seeing the help section of this page because you have `DEBUG = True` in your Django settings file. Change that to `False`, and only the initial error message will be displayed.

You can customize this page using the CSRF_FAILURE_VIEW setting.

Important add CSRF token:

In the template, there is a `{% csrf_token %}` template tag inside each POST form that targets an internal URL.

Some style changes on form:

```
{% extends 'base.html' %}

{% block content %}

<div class='container'>

    <h1>ADD.HTML</h1>
    <h2>Add a new car to the database:</h2>
    <form action="" method="POST">
        {% csrf_token %}  {# csrf_token to have the submit permission #}
        <div class="form-group">
            <label for="brand">Brand:</label>
```

```
        <input class="form-control" type="text" id="brand" name="brand">
    </div>
    <div class="form-group">
        <label for="year">Year</label>
        <input class="form-control" type="text" id="year" name="year">
    </div>
    <input class="btn btn-primary" type="submit">
</form>
{% endblock %}
```

→ on views.py add request.POST to add_view() function

```
def add_view(request):
    print(request.POST)
    return render(request, 'cars/add.html')
```

Output by submit the form:

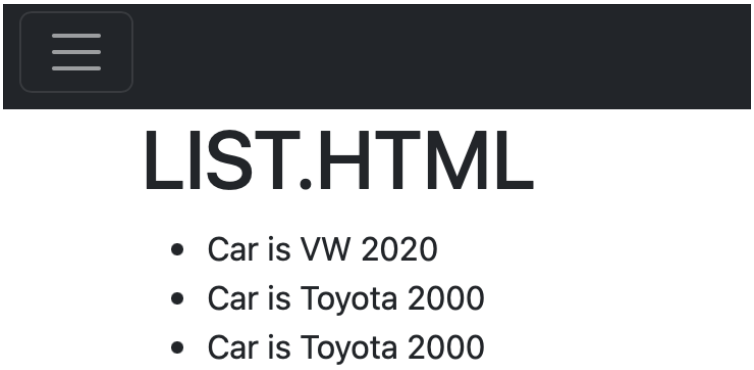
```
System check identified no issues (0 silenced).
October 27, 2022 - 10:35:06
Django version 4.1.2, using settings 'my_car_site.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
<QueryDict: {}>
[27/Oct/2022 10:35:12] "GET /cars/add/ HTTP/1.1" 200 2300
[27/Oct/2022 10:35:14] "GET /cars/list/ HTTP/1.1" 200 1736
<QueryDict: {}>
[27/Oct/2022 10:35:15] "GET /cars/add/ HTTP/1.1" 200 2300
<QueryDict: {'csrfmiddlewaretoken': ['il5B5PWgT626kaZ6Q3PtoVx78rwO3tReNxaFPC5rSQjuJ
ZaSiWrSFWUJvDi7xdqO'], 'brand': ['VW'], 'year': ['2020']}>
[27/Oct/2022 10:35:20] "POST /cars/add/ HTTP/1.1" 200 2300
```

→ if statement for creating objects:
→ import redirect and reverse in views.py

```
from django.shortcuts import render, redirect
from django.urls import reverse
from . import models


# Create your views here.

def list_view(request):
    all_cars = models.Car.objects.all()
```

```
    context = {'all_cars':all_cars}


    return render(request, 'cars/list.html', context = context)


def add_view(request):
    if request.POST:    # if there is a request post
        brand = request.POST['brand']  #  I grab the brand
        year = int(request.POST['year'])    # I grab the year (no exception here)
        models.Car.objects.create(brand = brand, year = year)   # I grab and save that objects in the database
        # if user submitted new car ---> List.html
        return redirect(reverse('cars:list'))
    else:
        return render(request, 'cars/add.html')


def delete_view(request):
    return render(request, 'cars/delete.html')
```

Output:

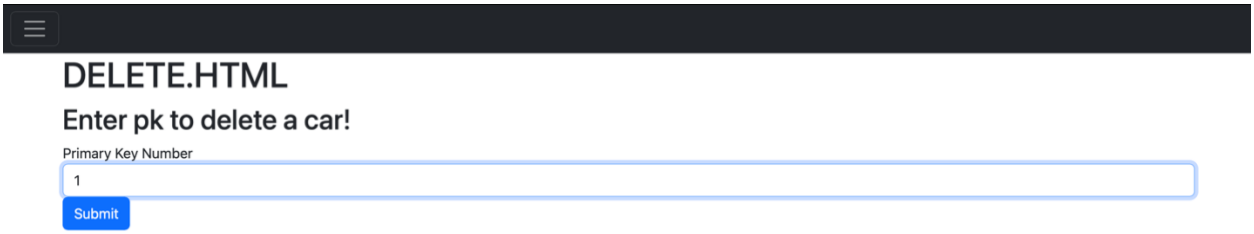


→ update delete.html
→ what is the primary that I wanna delete?

```
def delete_view(request):
    if request.POST:
        pk = request.POST['pk']
        try:
            models.Car.objects.get(pk=pk).delete()
```

```
            return redirect(reverse('cars:list'))
        except:
            print('pk not found')
            return redirect(reverse('cars:list'))
    else:
        return render(request, 'cars/delete.html')
```

→ delete some entries by enter the pk
Output:



42. Django Administration

Let's explore the Django Admin:
- domain.com/admin Extension
- Creating a "superuser"

Whole Django admin system just comes and automatically built itself based off the models and views you create

It's an amazing feature that it going to impress you when you realize you didn't have to do any extra code in order to have this happen!

→ domain.com/admin Extension
http://127.0.0.1:8000/admin

→ Creating a superuser

→ python manage.py createsuperuser
→ enter username, email and password (it is invisible)

43. Django Admin and Models

Let's explore how to register our models to the Django Admin interface

We'll also explore the **ModelAdmin** class which gives us additional functionality with the field presented in the Admin Interface

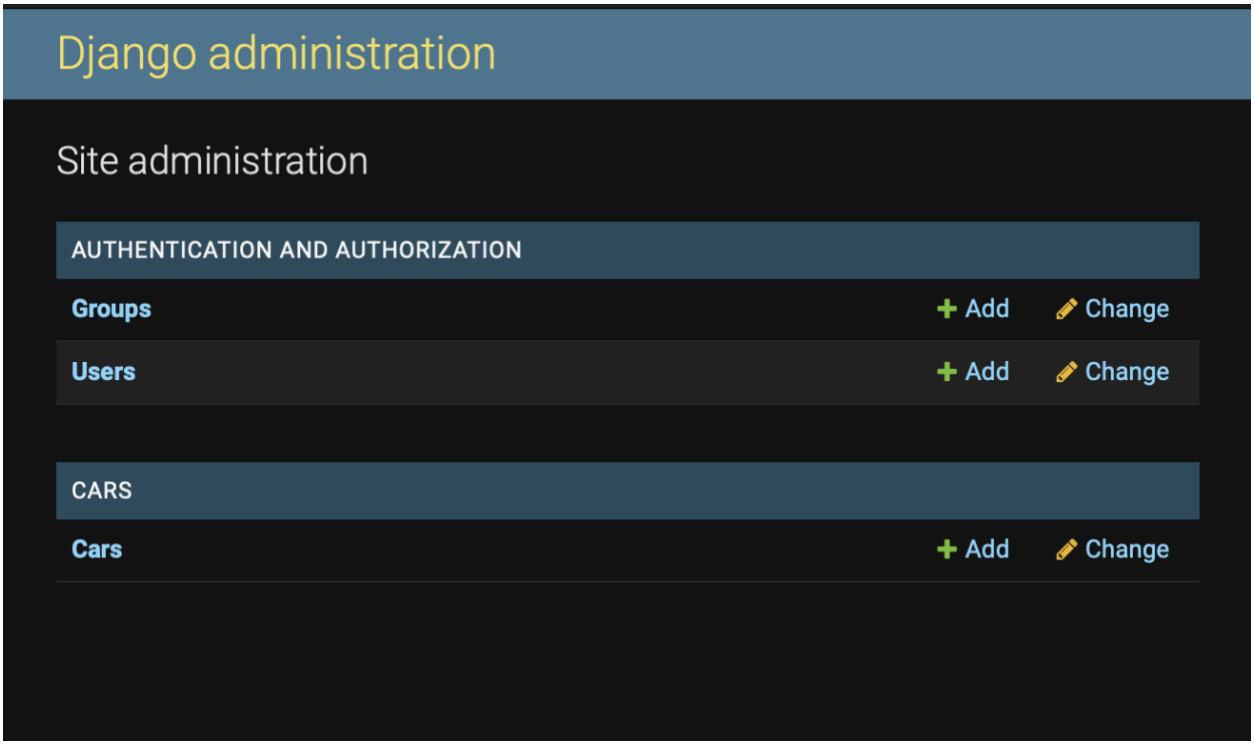
https://docs.djangoproject.com/en/4.1/ref/contrib/admin/
→ add Car model to admin site

On admin.py

admin.site.register(Object)

```
from django.contrib import admin
from cars.models import Car


# Register your models here.


admin.site.register(Car)
```

→ Cars is added

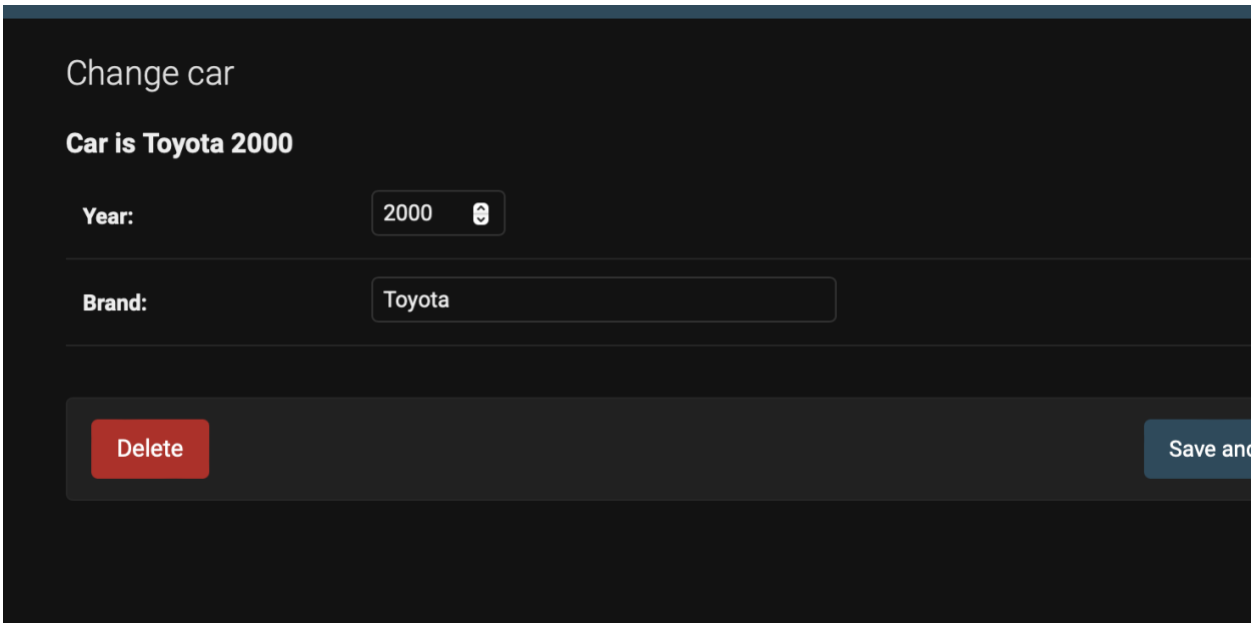


→ with ModelAdmin objects, the admin site can be changed:

ModelAdmin.fields:

```
from django.contrib import admin
from cars.models import Car

# Register your models here.

class CarAdmin(admin.ModelAdmin):
    fields = ['year', 'brand']


admin.site.register(Car, CarAdmin)
```

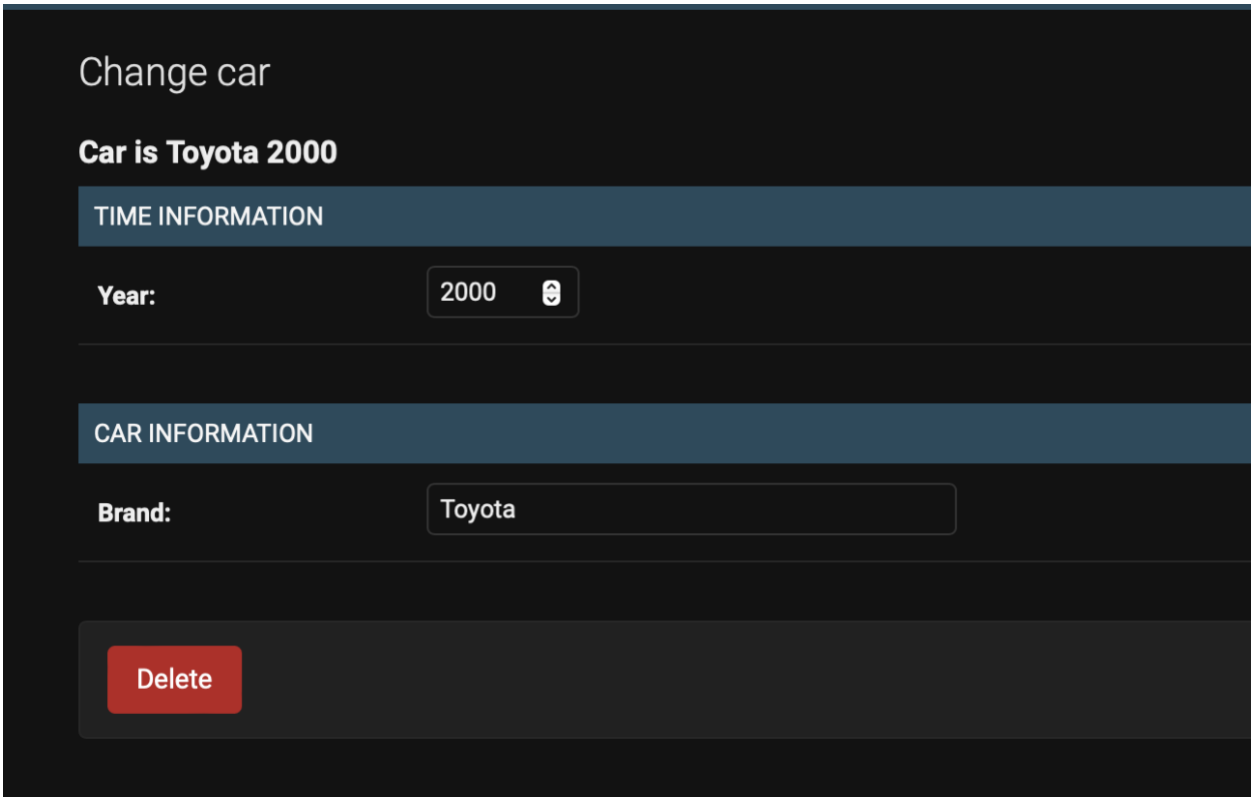
→ year and brand are different order!



→ ModelAdmin.fieldsets:

```
from django.contrib import admin
from cars.models import Car

# Register your models here.

class CarAdmin(admin.ModelAdmin):
    fieldsets = (
        ('TIME INFORMATION', {'fields': ['year']}),
        ('CAR INFORMATION', {'fields': ['brand']})

    )


admin.site.register(Car, CarAdmin)
```

Output:



For more visit the Django documentation webpage

Section 13: Django Forms

44. Introduction to Django Forms Section

We've been able to use HTML forms to allow client users send information in their browser to the backend of our Django application.

The Django website can then Create/Read/Update/Delete information in the database based on the HTML forms.

User interactions based on HTML forms are extremely common across the internet.

However, HTML forms require a lot of processing to connect with Django, especially when we want to later connect these inputs to Models.

Fortunately, Django comes with a built-in Forms class which can be used with Django and Python to create forms and then send that form to the template through a simple Tag call **{{form}}.**

This allows us to rapidly develop forms for the client while only needing to work mainly with Django and Python!

This will be a huge productivity improvement and make our overall website code more readable.

Django Form Section Overview:
- GET, POST, and CSRF Review
- Django Form Class Basics
- Form Fields and Validation
- Form Widgets and CSS Styling
- ModelForms

45. GET, POST, and CSRF Overview

We've already seen that HTTP (Hypertext Transfer Protocol) is the foundation for the method of sending and receiving data over the world wide web.

Recall, HTTPS is simply an encrypted version of HTTP.

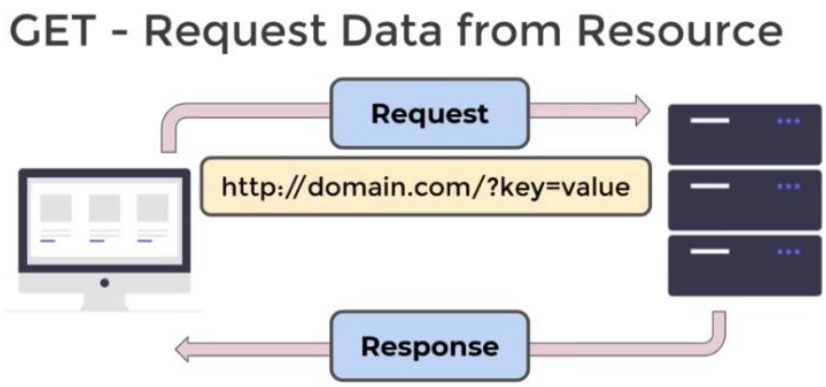
HTTP defines a variety of methods for interactions.

The key methods we need to understand are **GET** and **POST** methods, which we've already seen used in HTML forms.

GET
- Requests data from a specified resource.

POST
- Requests to send data to a server to create/update a resource.

Note the tricky terminology that can be confusing, both GET and POST are HTTP **request methods**, even though you will commonly see GET as *"requesting"* information and POST as *"sending"* information, they are technically **HTTP Requests**.



Notice how the GET request is sent in the

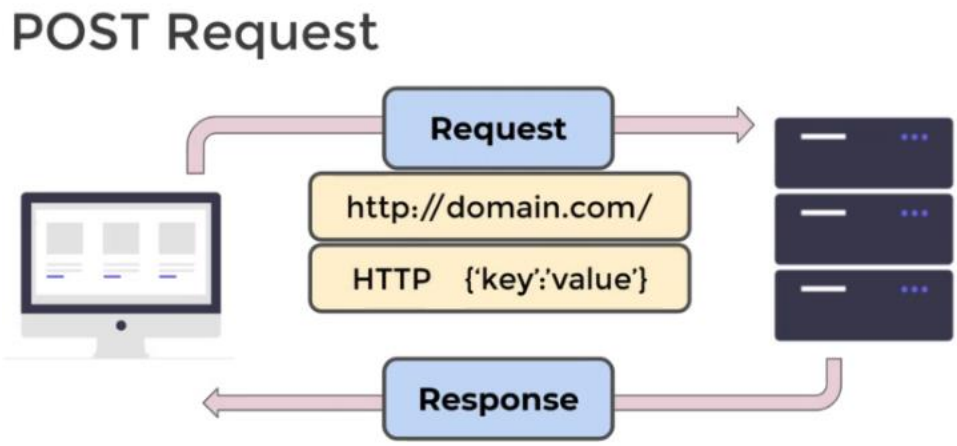
URL. This means a few things:

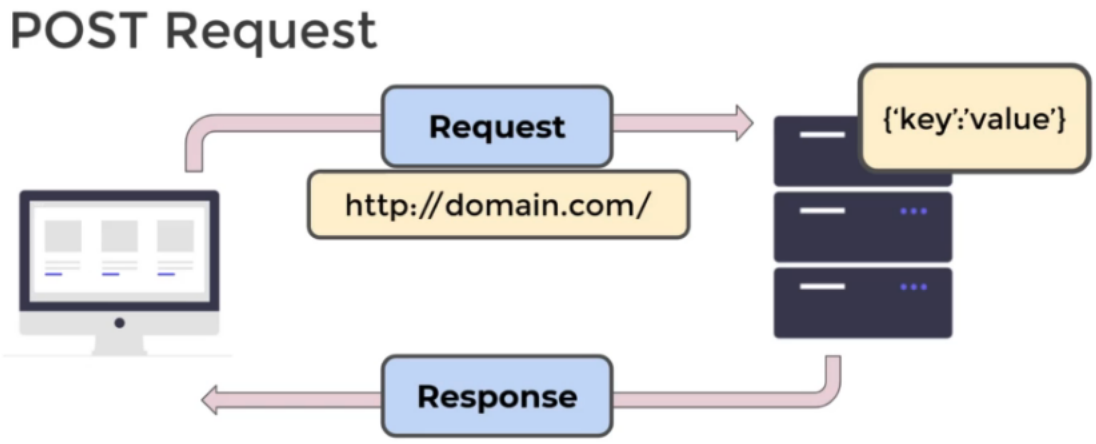
- GET request can be bookmarked
- GET request saved in history
- GET request can be cached
- GET request has length limits

Also GET request can only request data, not modify or update anything

What if we want to send information for the specific purpose of updating some backend information?

We would not want that information in the URL and our main concern is no receiving information back, but instead sending information.

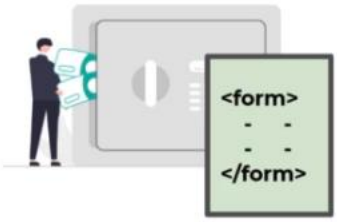




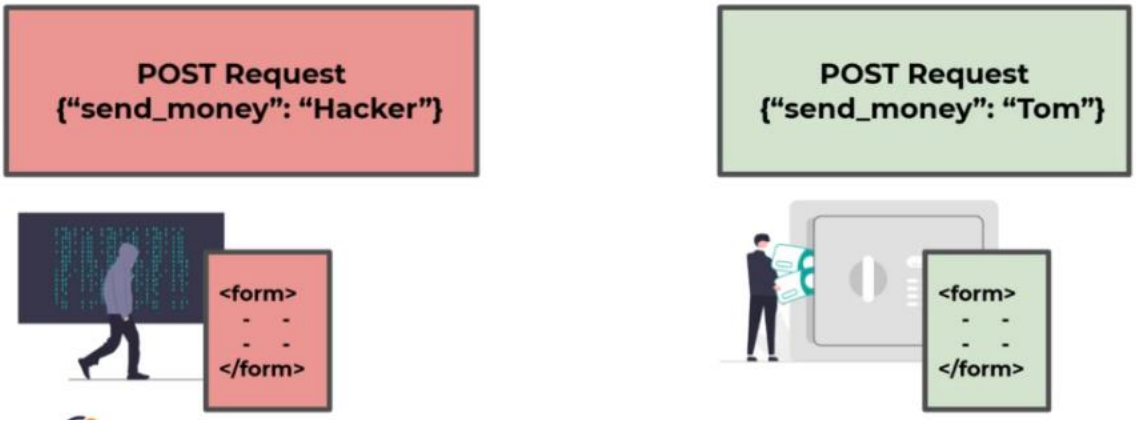
We see that HTML forms are actually quite readable by the browser, how can we make sure the HTML form on the page is being correctly used by the appropriate user?

Could a malicious actor try to ***fake*** an HTML form?
Imagine this forgery attempt...

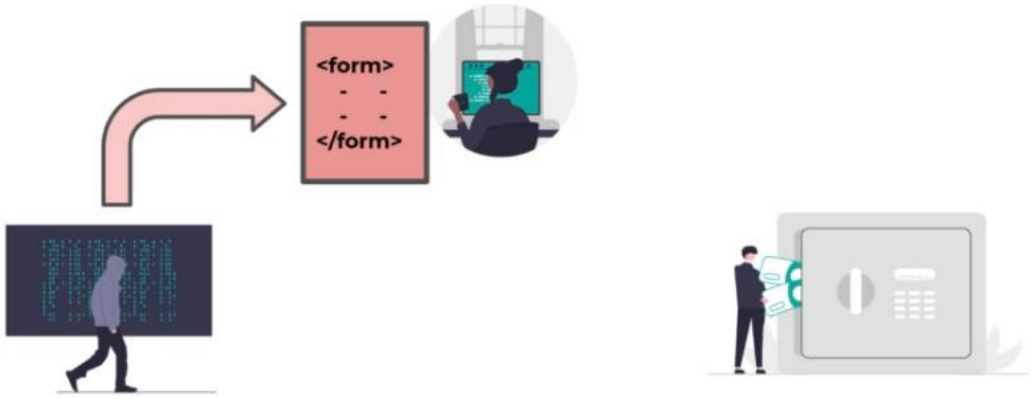
Step #1: Bank Website Form

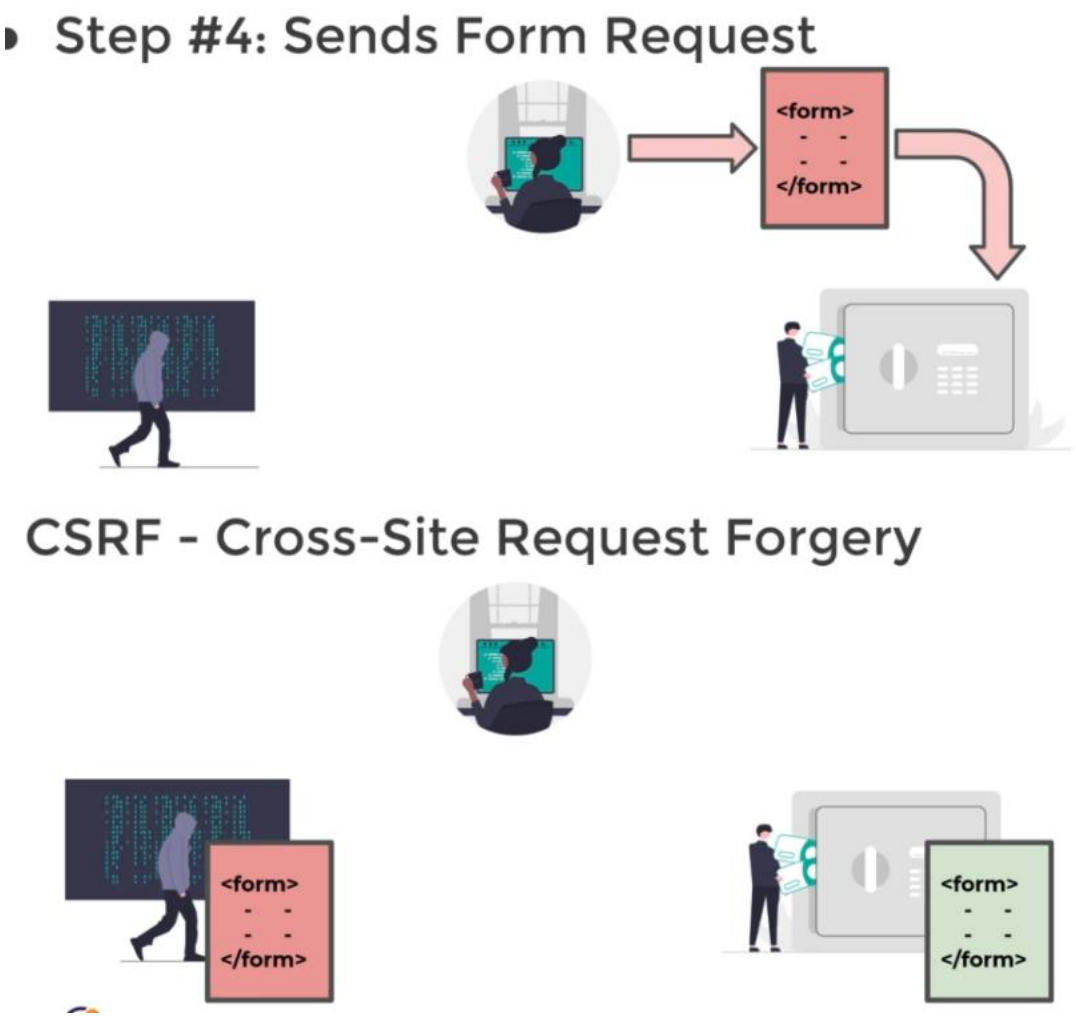


Step #2: Hacker Forges Form



Step #3: Phishing to send form





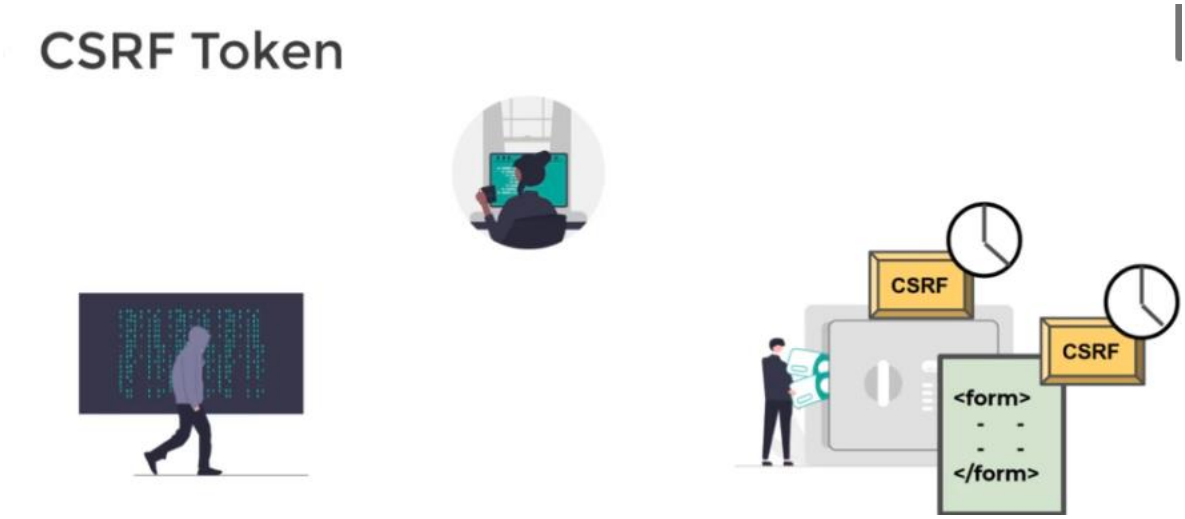
How to prevent this attack?

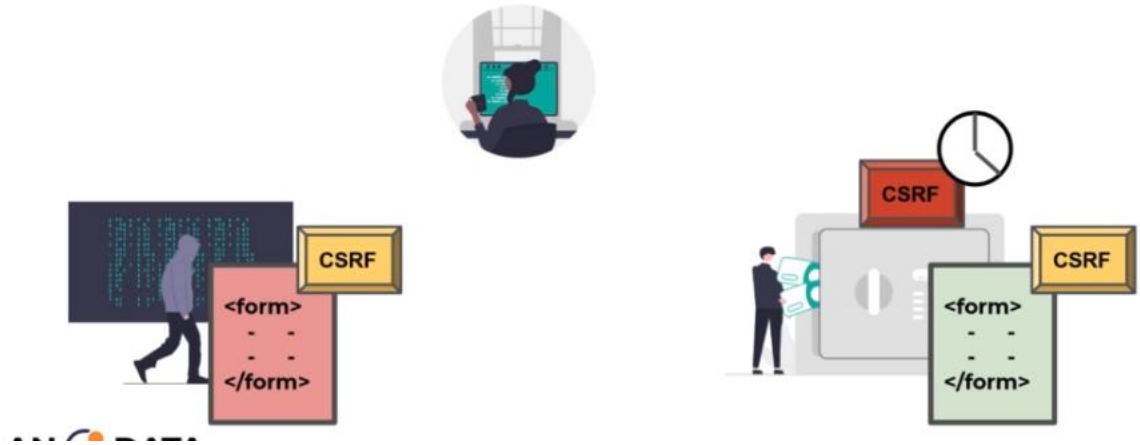
- We could generate a random cryptographic token with every form during each session.
- The server could then confirm if the token matches with the current session.

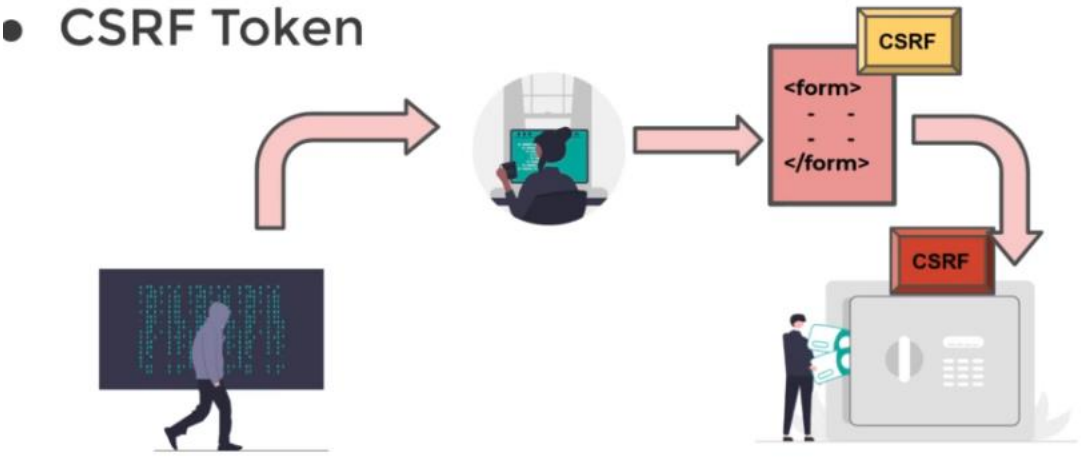
How to prevent this attack?

- Since each session has a unique token, only the true original form would be accepted as authentic.

Imagine a hacker would like to hack a bank account and sending money
→ With time session Token can only be valid for certain period of time
→ The hacker will not be able to post a request for sending money because the token is no longer valid
→ **See pictures below:**







Django creates these CSRF tokens for us automatically with a simple tag call!

We simply remember to provide:

• **{% csrf_token %}**

For more information visit:

https://docs.djangoproject.com/en/4.1/ref/csrf/

46. Django Form Class Basics

Let's create a new Django project and Django app and then we can use Django Forms instead of a manually created HTML form!

Let's quickly cover the steps we will perform in this lecture to create a car rental review feedback form...

Steps:

- Create New Project and App
- Connect Templates, Views, and URLs
- Create a **forms.py** file
- Create a Django Form Class inside forms.py
- Connect Django Form to View for context insertion inside Template

Visit the documentation about forms! Very good!

https://docs.djangoproject.com/en/4.1/topics/forms/

Using form.py

Steps: Start new project!

→ Create Templates
→ Create Views
→ Connect with URLs
→ Set Settings (Install App)
→ Add Content
    → HTMLs
→ Runserver for checking
→ **Create Form**

Hint: 2 things to worry in HTML files:

- **method="POST"**
- **input type="submit"**

```
<body>
    <h1>Rental Review Form</h1>
```

```
    <form action="" method='POST'>


        <input type="submit">


    </form>
</body>
```

- **Create forms.py file in app level** (follow same principle like models.py)

- **Create class ExampleForm(forms.Form)** → For every HTML label input paring you want, you create an attribute inside this class

- Use field to replace label and inputs! More on fields → https://docs.djangoproject.com/en/4.1/topics/forms/#more-on-fields Look at the built-in field in the documentation

```
from django import forms


class ReviewForm(forms.Form):
    first_name = forms.CharField(label = 'First Name', max_length=100)
    last_name = forms.CharField(label= 'Last Name', max_length=100)
    email = forms.CharField(label = 'Email')
    review = forms.CharField(label= 'Please write your review here')
```

- **Passing to views.py**

```
from django.shortcuts import render, redirect
from django.urls import reverse
from cars.forms import ReviewForm
# Create your views here.


def rental_review(request):


    #POST REQUEST --> Form Contest --> Thank You!
    if request.method == 'POST':
        form = ReviewForm(request.POST)
```

```
        if form.is_valid():
            #{'first_name': 'Jose',}
            print(form.cleaned_data)
            return redirect(reverse('cars:thank_you'))
        #Else, Render Form


    else:
        form = ReviewForm()
    return render(request, 'cars/rental_review.html', context={'form':form})


def thank_you(request):
    return render(request, 'cars/thank_you.html')
```

Explanation:

After we create the form class, we go ahead and import into our view. The actual form is imported (ReviewForm).
Then you check, Hey are they actually posting something? (if request.method ==’POST’)
If so, pass that information into the review form and if it’s valid (if form.is_valid(), then I do whatever I want for the information by accessing it through what’s essentially a Python dictionary (form.cleaned_data). And then I am going to redirect them (return redirect(reverse…)) So we will actually see the information inside our terminal.

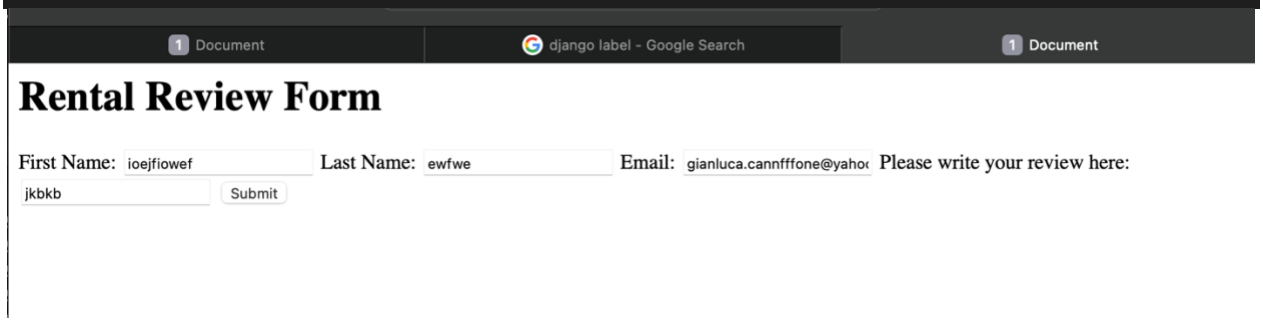
Otherwise, (Else:…) it’s the first time visited page to have a hit, then go ahead and just create the form and then pass it in here as context of the page. Which then means if you come back here to run the review.html, that is being passed in here into the {{form}}

Create HTML:

```
{% csrf_token %}
        {{form}}
```

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
```

```
    <h1>Rental Review Form</h1>
    <form action="" method='POST'>
        {% csrf_token %}
        {{form}}
        <input type="submit">

    </form>
</body>
</html>
```



With print(form.cleaned_data)

```
[28/Oct/2022 08:34:32] "GET /cars/rental_view/ HTTP/1.1" 200 1270
{'first_name': 'ioejfiowef', 'last_name': 'ewfwe', 'email': 'gianluca.cannfffon
e@yahoo.de', 'review': 'jkbkb'}
```

This data can be import or inject straight to the database

47. Django Forms – Templates Rendering

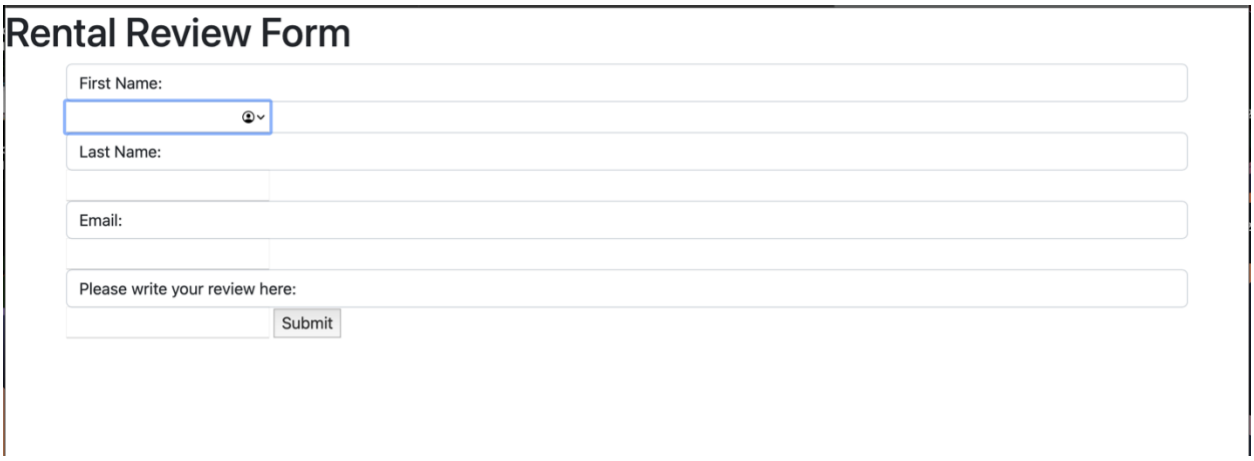
When passing **{{form}}** to the template, we saw that the HTML tags rendered by the Django Form Widgets are all in the same line and don't look visually appealing.

Let's explore a few more details around template rendering inside the .html files.

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
     <!-- CSS only -->
```

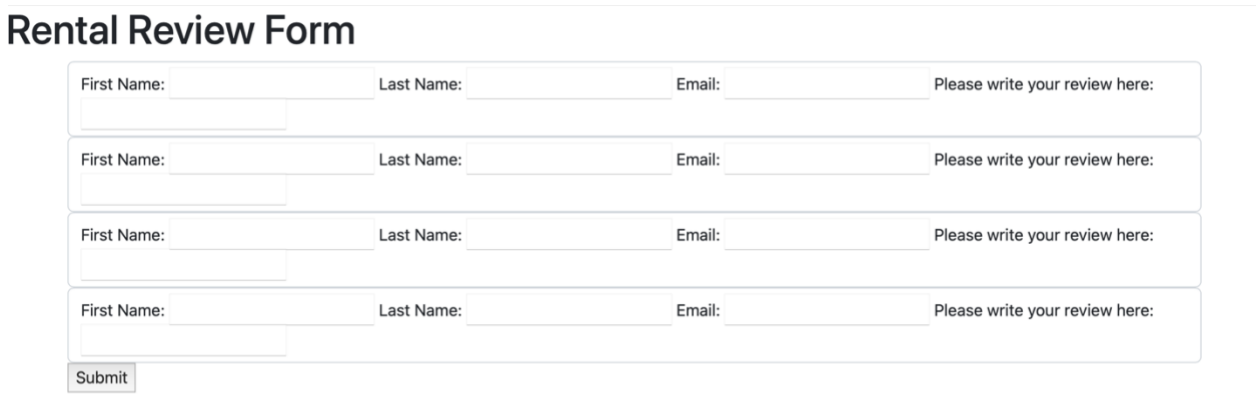
```
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1WTRi"
crossorigin="anonymous">
    <!-- JavaScript Bundle with Popper -->
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-
OERcA2EqjJCMA+/3y+gxIOqMEjwtxJY7qPCqsdltbNJuaOe923+mo//f6V8Qbsw3"
crossorigin="anonymous"></script>
    <title>Document</title>
</head>
<body>
    <h1>Rental Review Form</h1>
    <form action="" method='POST'>
        {% csrf_token %}
    <div class="container">
        {% for field in form %}
            <div class="form-control">
                {{field.label_tag}}
            </div>
            {{field}}
        {% endfor %}
        <input type="submit">
    </div>
    </form>
</body>
</html>
```

Output:



```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
     <!-- CSS only -->
     <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1WTRi"
crossorigin="anonymous">
     <!-- JavaScript Bundle with Popper -->
     <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-
OERcA2EqjJCMA+/3y+gxIOqMEjwtxJY7qPCqsdltbNJuaOe923+mo//f6V8Qbsw3"
crossorigin="anonymous"></script>
    <title>Document</title>
</head>
<body>
    <h1>Rental Review Form</h1>
    <form action="" method='POST'>
        {% csrf_token %}
    <div class="container">
        {% for field in form %}
            <div class="form-control">
                {{form}}
            </div>
        {% endfor %}
        <input type="submit">
    </div>
    </form>
</body>
</html>
```

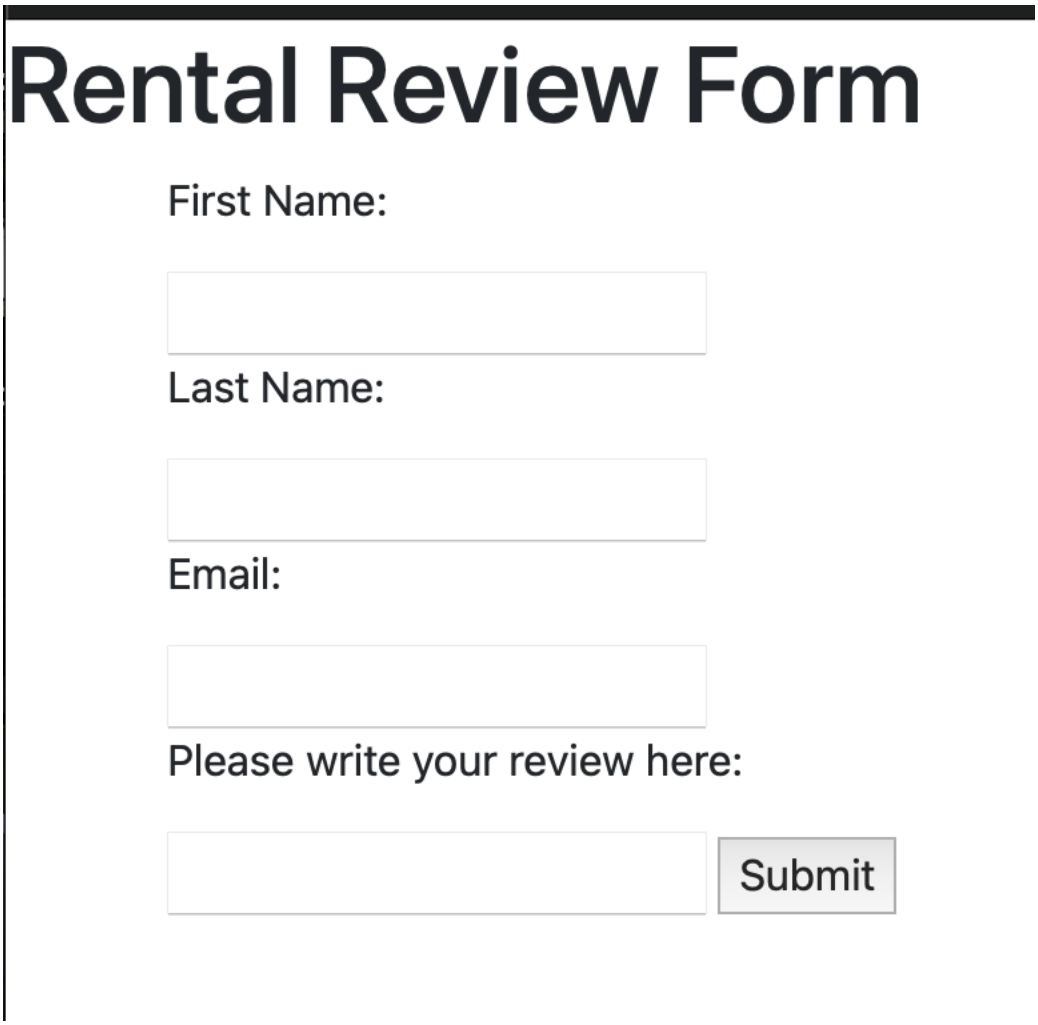
Output:



→ Use mb-3:

```
<body>
    <h1>Rental Review Form</h1>
    <div class="container">
    <form action="" method='POST'>
        {% csrf_token %}
        {% for field in form %}
            <div class="mb-3">
                {{field.label_tag}}
            </div>
            {{field}}
        {% endfor %}
        <input type="submit">
    </form>
    </div>


</body>
```

Output:



Manually:

Two significally commands for forms:

Widget: form.first_name
Label: form.first_name.label_tag (for the particular field)

Easy way:

{{form.as_p}} to let Django add   tags to each field

48. Django Forms – Widget and Styling

Recall that a Form Field inside forms.py ends up generating a Django **widget** which in turn renders the actual HTML form input/label tags.

To have more control over styling and presentation, we can access widget attributes.

We'll begin by linking a static files directory to hold our custom CSS files:

- Create **app/static/app/custom.css** file
- Load static directory in .html
- Link static CSS file connection
- Run migrate to load new app in settings.py file

Simple style class: **custom.css**

```
.myform {
    border: 5px dashed red;
}
```

→ Load static directory in .html
→ Link static CSS file connection

→ {% load static %)
→ <link rel="stylesheet" href="{% static 'cars/custom.css %}">

```
{% load static %}

<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
     <!-- CSS only -->
     <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-Zenh87qX5JnK2Jl0vWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/et3URy9Bv1WTRi"
crossorigin="anonymous">
     <!-- JavaScript Bundle with Popper -->
     <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-
OERcA2EqjJCMA+/3y+gxIOqMEjwtxJY7qPCqsdltbNJuaOe923+mo//f6V8Qbsw3"
crossorigin="anonymous"></script>

     <link rel="stylesheet" href="{% static 'cars/custom.css' %}">
    <title>Document</title>
</head>
```

→ pass in style class in body

→ <div class=”container **myform**”>

```
<body>
    <h1>Rental Review Form</h1>
    <div class="container myform">
    <form action="" method='POST'>
        {% csrf_token %}
        {% for field in form %}
            <div class="mb-3">
                {{field.label_tag}}
            </div>
            {{field}}
        {% endfor %}
        <input type="submit">
    </form>
    </div>

</body>
</html>
```

→ **Run migrate** to load new app in settings.py file

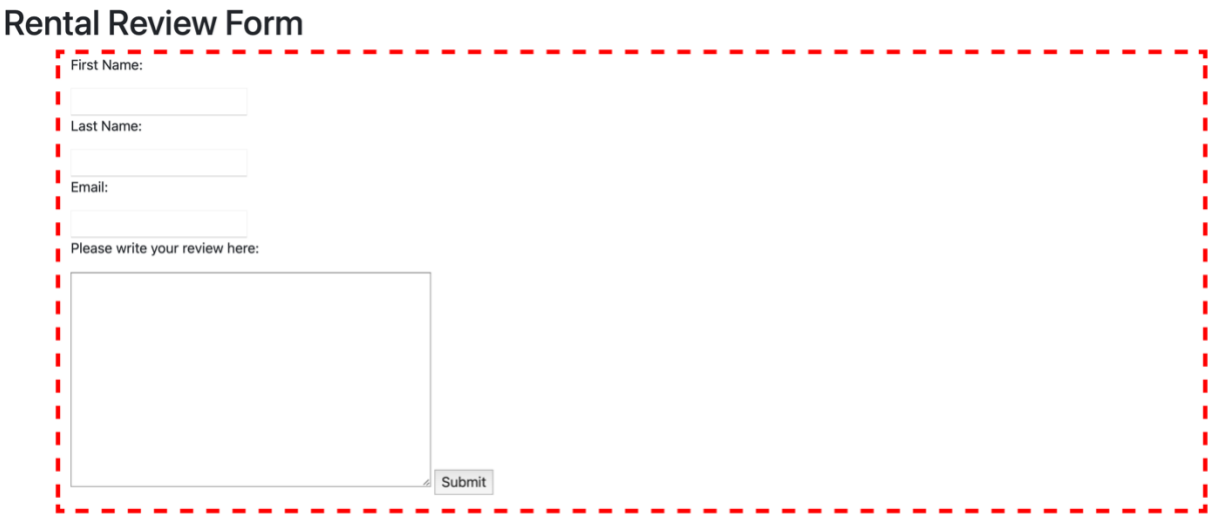
Visit: https://docs.djangoproject.com/en/4.1/ref/forms/fields/#module-django.forms.fields

→ Add textarea() built-in widget

```
from django import forms


class ReviewForm(forms.Form):
    first_name = forms.CharField(label = 'First Name:', max_length=100)
```

```
    last_name = forms.CharField(label= 'Last Name:', max_length=100)
    email = forms.CharField(label = 'Email:')
    review = forms.CharField(label= 'Please write your review here:',
                             widget=forms.Textarea())
```

Output:



→ Styling Textarea:

→ have red dashes only on text area

→ add attrs in **forms.py**

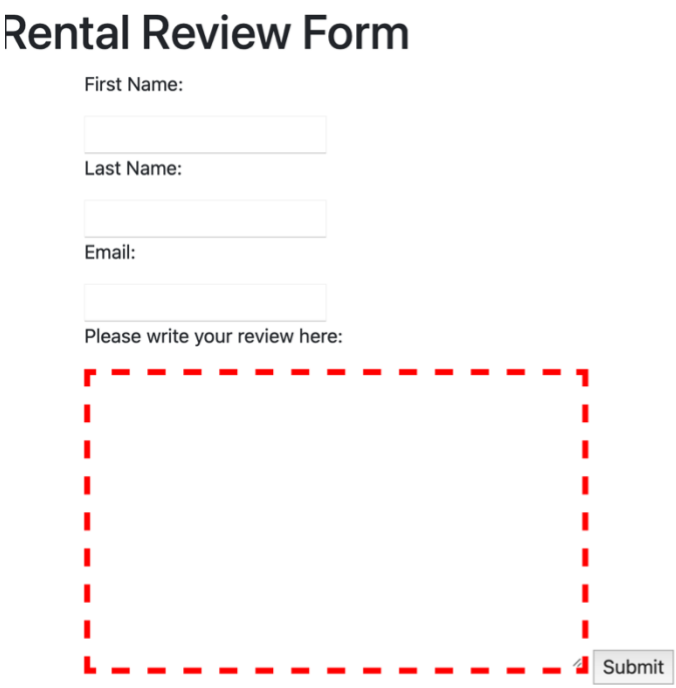
```
from django import forms


class ReviewForm(forms.Form):
    first_name = forms.CharField(label = 'First Name:', max_length=100)
    last_name = forms.CharField(label= 'Last Name:', max_length=100)
    email = forms.CharField(label = 'Email:')
    review = forms.CharField(label= 'Please write your review here:',
                             widget=forms.Textarea(attrs={'class':'myform'}))
```

→ delete myforms in html

```
<body>
```

```
<h1>Rental Review Form</h1>
<div class="container"> → no more myform
<form action="" method='POST'>
    {% csrf_token %}
    {% for field in form %}
        <div class="mb-3">
            {{field.label_tag}}
        </div>
        {{field}}
    {% endfor %}
    <input type="submit">
</form>
</div>

</body>
```

Output:



→ what for attributes exist for textarea (GOOGLE)
→ new attributes:

```
from django import forms


class ReviewForm(forms.Form):
    first_name = forms.CharField(label = 'First Name:', max_length=100)
    last_name = forms.CharField(label= 'Last Name:', max_length=100)
    email = forms.CharField(label = 'Email:')
    review = forms.CharField(label= 'Please write your review here:',
                        widget=forms.Textarea(attrs={'class':'myform', 'rows':'2','cols':'2'}))
```

Output:

https://www.w3schools.com/tags/tag_textarea.asp

Attributes

Attribute	Value	Description
autofocus	autofocus	Specifies that a text area should automatically get focus when the page loads
cols	*number*	Specifies the visible width of a text area
dirname	*textareaname*.dir	Specifies that the text direction of the textarea will be submitted
disabled	disabled	Specifies that a text area should be disabled
form	*form_id*	Specifies which form the text area belongs to
maxlength	*number*	Specifies the maximum number of characters allowed in the text area
name	*text*	Specifies a name for a text area
placeholder	*text*	Specifies a short hint that describes the expected value of a text area
readonly	readonly	Specifies that a text area should be read-only
required	required	Specifies that a text area is required/must be filled out
rows	*number*	Specifies the visible number of lines in a text area
wrap	hard soft	Specifies how the text in a text area is to be wrapped when submitted in a form

49. Django – ModelForm Class

Often, we use forms to directly interact with a model, such as creating a new instance of a data point inside a model.

Fortunately, Django provides the ModelForm class which automatically creates a Form with fields connected to each model field.

→ create Model

```
from django.db import models


# Create your models here.


class Review(models.Model):
    first_name = models.CharField(max_length=30)
```

```
    last_name = models.CharField(max_length=30)
    stars = models.IntegerField()
```

→ add model to admin

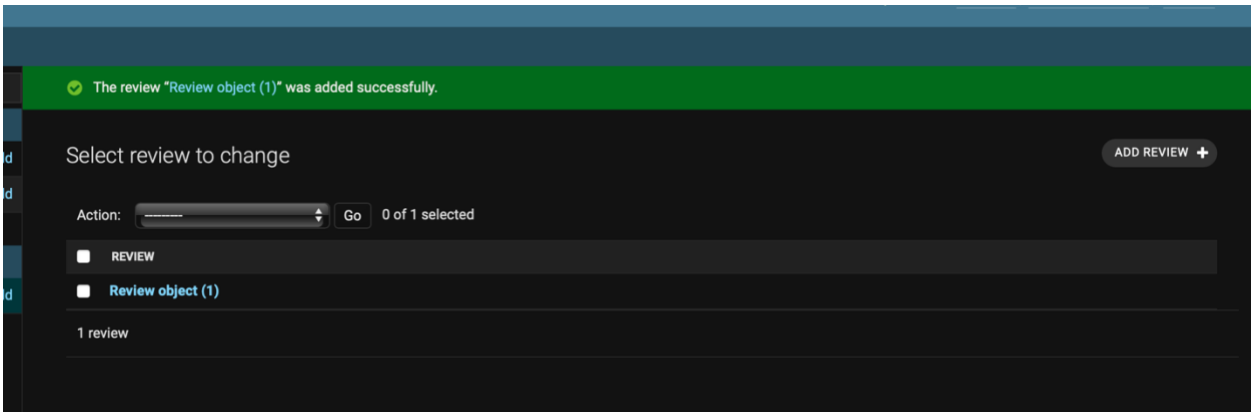
```
from django.contrib import admin
from cars.models import Review

# Register your models here.

admin.site.register(Review)
```

→ make migration: python manage.py makemigrations cars

→ run migrate: python manage.py migrate

→ create superuser (admin): python manage.py create superuser

→ create object via admin:



Creating forms from Models via ModelForm:

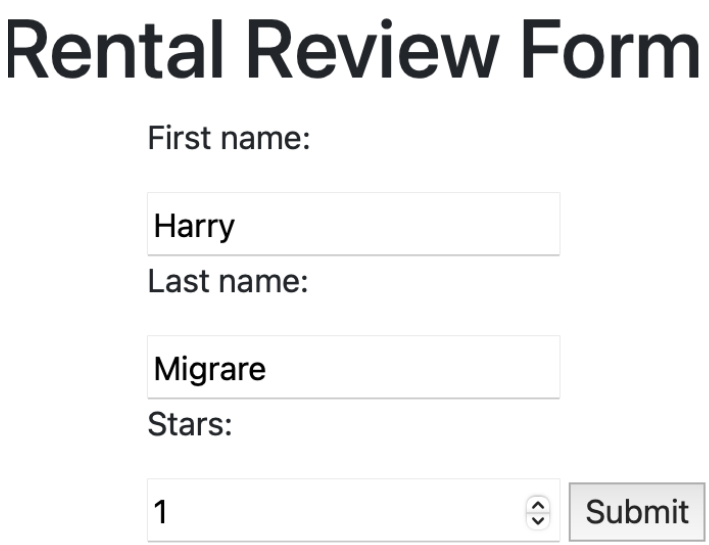
https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/

→ change forms and add ModelForm

```
from django import forms
from cars.models import Review # NEW
from django.forms import ModelForm #NEW

# class ReviewForm(forms.Form):
```

```
#     first_name = forms.CharField(label = 'First Name:', max_length=100)
#     last_name = forms.CharField(label= 'Last Name:', max_length=100)
#     email = forms.CharField(label = 'Email:')
#     review = forms.CharField(label= 'Please write your review here:',
#                         widget=forms.Textarea(attrs={'class':'myform', 'rows':'2','cols':'2'}))

class ReviewForm(ModelForm): # Creating forms from Models # NEW
    class Meta:
        model = Review
        fields = ['first_name', 'last_name', 'stars']
```

Output:



→ add **form.save()** to save data in database
→ This single line, because it is a model form, is going to automatically save what the user passed in as a new instance of the model

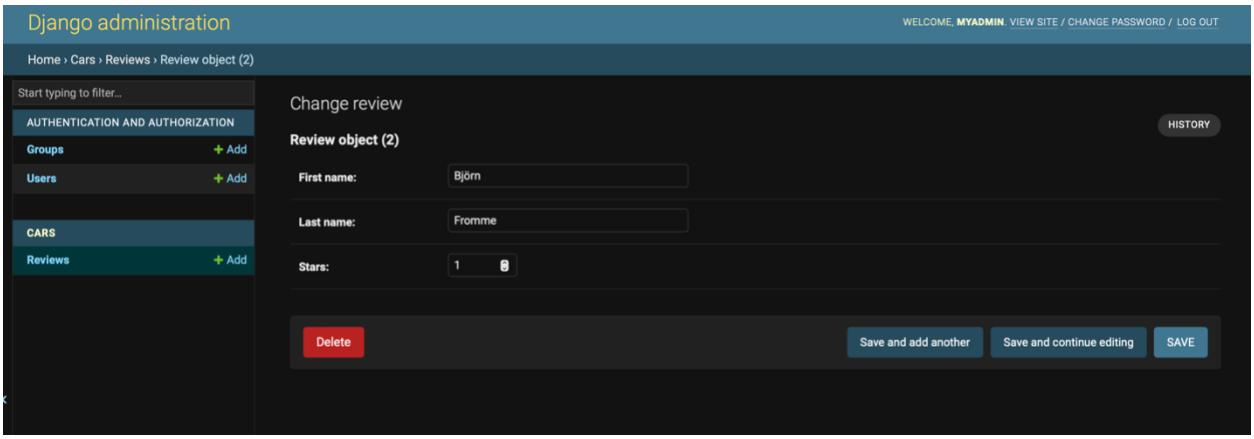
```
from django.shortcuts import render, redirect
from django.urls import reverse
from cars.forms import ReviewForm
# Create your views here.
```

```python
def rental_review(request):

    #POST REQUEST --> Form Contest --> Thank You!
    if request.method == 'POST':
        form = ReviewForm(request.POST)

        if form.is_valid():
            form.save() # This single line, because it is a model form,
            # is going to automatically save what the user passed in
            # as a new instance of the model
            print(form.cleaned_data)
            return redirect(reverse('cars:thank_you'))
        #Else, Render Form
    else:
        form = ReviewForm()
    return render(request, 'cars/rental_review.html', context={'form':form})

def thank_you(request):
    return render(request, 'cars/thank_you.html')
```

→ passed in
Björn Fromme 1:

```
System check identified no issues (0 silenced).
October 28, 2022 - 15:43:51
Django version 4.1.2, using settings 'mysite_form.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
{'first_name': 'Björn', 'last_name': 'Fromme', 'stars': 1}
[28/Oct/2022 15:44:47] "POST /cars/rental_view/ HTTP/1.1" 302 0
[28/Oct/2022 15:44:47] "GET /cars/thank_you/ HTTP/1.1" 200 267
[28/Oct/2022 15:45:23] "GET /admin/ HTTP/1.1" 200 4542
[28/Oct/2022 15:45:24] "GET /admin/cars/review/ HTTP/1.1" 200 6737
[28/Oct/2022 15:45:24] "GET /admin/jsi18n/ HTTP/1.1" 200 3343
[28/Oct/2022 15:45:26] "GET /admin/cars/review/2/change/ HTTP/1.1" 200 7280
[28/Oct/2022 15:45:26] "GET /admin/jsi18n/ HTTP/1.1" 200 3343
```

→ Check on admin site:



50. Django – ModelForms Customization

→ Define field attributes: "__all__" → same as inserting all attributes

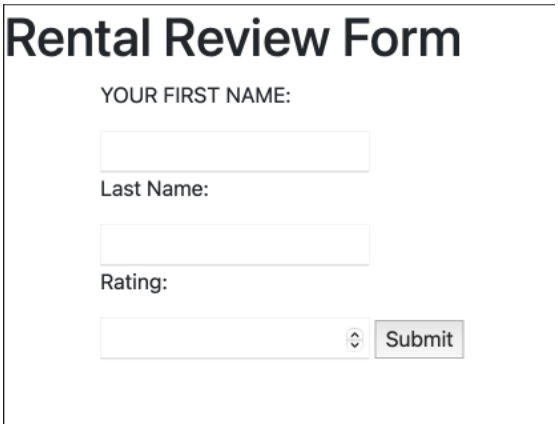
```
from django import forms
from cars.models import Review
from django.forms import ModelForm

# class ReviewForm(forms.Form):
#     first_name = forms.CharField(label = 'First Name:', max_length=100)
#     last_name = forms.CharField(label= 'Last Name:', max_length=100)
#     email = forms.CharField(label = 'Email:')
#     review = forms.CharField(label= 'Please write your review here:',
#                              widget=forms.Textarea(attrs={'class':'myform', 'rows':'2','cols':'2'}))

class ReviewForm(ModelForm): # Creating forms from Models
    class Meta:
        model = Review
        fields = "__all__" # The same as inserting all attributes
```

→ change Labels

```
class ReviewForm(ModelForm): # Creating forms from Models
    class Meta:
        model = Review
        fields = "__all__" # The same as inserting all attributes
```

```
    labels = {
        'first_name':"YOUR FIRST NAME",
        'last_name':'Last Name',
        'stars':'Rating'
    }
```

Output:

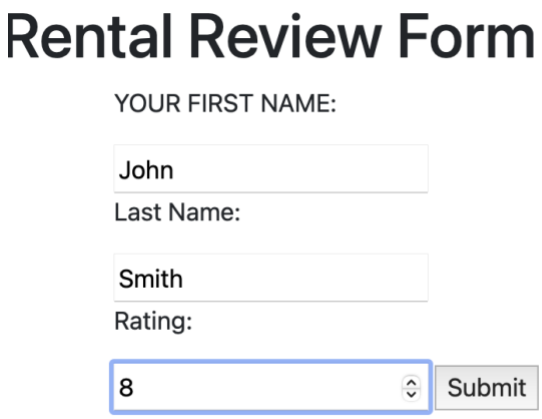


→ Using validators in **models.py**

```
from django.db import models
from django.core.validators import MinValueValidator, MaxValueValidator

# Create your models here.

class Review(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    stars = models.IntegerField(validators = [MinValueValidator(1), MaxValueValidator(5)])
```

→ makemigrations and run migrate again

→ run server again



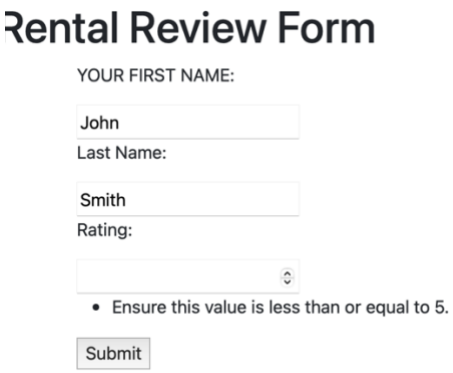
→ Submitting doesn't work!!

→ add {{field.erros}} in html

```
<body>
    <h1>Rental Review Form</h1>
    <div class="container">
    <form action="" method='POST'>
        {% csrf_token %}
        {% for field in form %}
            <div class="mb-3">
                {{field.label_tag}}
            </div>
            {{field}}
            {{field.errors}}
        {% endfor %}
        <input type="submit">
    </form>
    </div>


</body>
```

→ Django is checking through the validation if input is valid and provide a message if input is not valid!

Output:



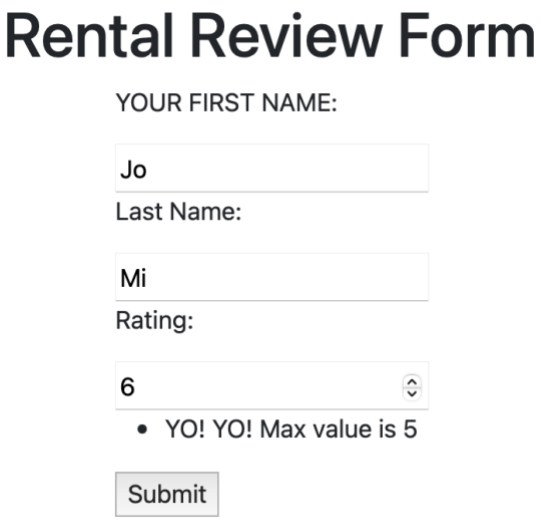
→ go to built-in field classes and look under these classes for error messages keys

https://docs.djangoproject.com/en/4.1/ref/forms/fields/#built-in-field-classes

→ add error_messages → min_value, max_value

```
class ReviewForm(ModelForm): # Creating forms from Models
    class Meta:
        model = Review
        fields = "__all__" # The same as inserting all attributes

        labels = {
            'first_name':"YOUR FIRST NAME",
            'last_name':'Last Name',
            'stars':'Rating'
        }

        error_messages = {
            'stars': {
                "min_value":"YO! Min value is 1",
                "max_value":"YO! YO! Max value is 5"
            }
        }
```

Output:



Section 14: Django Class Based Views

51. Introduction to Class Based Views

So far we've only seen functions inside our views.py file, but just like Forms and Models, Django provides an entire View class system that is very powerful for quickly rendering commonly used views.

Let's see Django developer's own reasoning from the documentation on why to use class based views...

Documentation on Class Based Views:

- Writing web applications can be monotonous, because we repeat certain patterns again and again. Django tries to take away some of that monotony at the model and template layers, but web developers also experience this boredom at the view level
- Django's generic views were developed to ease that pain. They take certain common idioms and patterns found in view development and abstract them so that you can quickly write common views of data without having to write too much code.
- Django CBVs (Class Based Views) come with many pre-built generic class views for common tasks, such as listing all the values for a particular model in a database (ListView) or creating a new instance of a model object (CreateView).
- Once we cover the basics of Class Based Views, we'll go through a "tour" of the key generic class based views and cover the few specifics that are important to know when using them.
- Note! They should be relatively easy to use and understand on their own!

Section Overview:

- Class Based Views Basics
- Generic Views
- TemplateView
- ListView
- DetailView
- CreateView
- DeleteView

52. Django CBV – TemplateView

→ Create Templates folder and html files

Create TemplateView class on views.py

→ from django.views.generic import TemplateView
→ create class for TemplaeView

```
from django.shortcuts import render
from django.views.generic import TemplateView


# Create your views here.


class HomeView(TemplateView):
    template_name = 'classroom/home.html'


class ThankYouView(TemplateView):
    template_name = 'classroom/thank_you.html'
```

Connect TemplateView class to urls.py

→ from school.views import HomeView
→ app_name = ‘school’

→ add instead a function the method **HomeView.as_view()**

```
from django.urls import path
from classroom.views import HomeView, ThankYouView


app_name = 'classroom'


urlpatterns = [
```

```
    path('', HomeView.as_view(), name='home'),  # path expects a function
    path('thank_you/', ThankYouView.as_view(), name='thank_you')
]
```

→ Connect with urls.py project level

```
"""school URL Configuration

The `urlpatterns` list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/4.1/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('classroom/', include('classroom.urls'))
]
```

→ set app at settings.py
→ ‘classroom.apps.ClassroomConfig’
→ Check if templates are working
→ run server

Thank you!!

→ Create a link on home.html which takes you to thank_you.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home</title>
</head>
<body>
    <h1>Welcome to the Classroom</h1>
    <ul>
        <li> <a href="{% url 'classroom:thank_you' %}">Take me to the Classroom</li>
    </ul>
</body>
</html>
```

53. Django CBV – FormView

→ Create forms.py on app level
(as usual)

```
from django import forms
from django.forms import Textarea
```

```
class ContactForm(forms.Form):
    name = forms.CharField()
    message = forms.CharField(widget=Textarea)
```

→ create Template form html

```
<h1>Contact Form</h1>

<form method="Post">
    {% csrf_token %}
    {{form.as_p}}
    <input type = "submit" value = "Submit">
</form>
```

→ Create FormView on views.py

```
from django.shortcuts import render
from django.views.generic import TemplateView, FormView

from classroom.forms import ContactForm

# Create your views here.

class HomeView(TemplateView):
    template_name = "classroom/home.html"

class ThankYouView(TemplateView):
    template_name = "classroom/thank_you.html"

class ContactFormView(FormView):
    form_class = ContactForm
    template_name = "classroom/contact.html"

    # success URL? It is the actual URL NOT a template.html
    success_url = "/classroom/thank_you/"

    # what to do with form
```

```
    def form_valid(self, form):
        print(form.cleaned_data) # it is a dict[key]// form.save() is also possible
        return super().form_valid(form) # similar to ContactForm(request.Post)
```

#What to do with form
→ Ok the form is valid! What do you do with it?
→ Then Use this method call super().form_valid(form) → Inherited from FormView → to essentially take care of things

→ Connect to urls.py
→ import ContactFormView

```
from django.urls import path
from classroom.views import HomeView, ThankYouView, ContactFormView

app_name = 'classroom'

urlpatterns = [
    path('', HomeView.as_view(), name='home'),  # path expects a function
    path('thank_you/', ThankYouView.as_view(), name='thank_you'),
    path('contact/', ContactFormView.as_view(), name='contact')
]
```

→ Create a link on home.html which takes you to contact.html

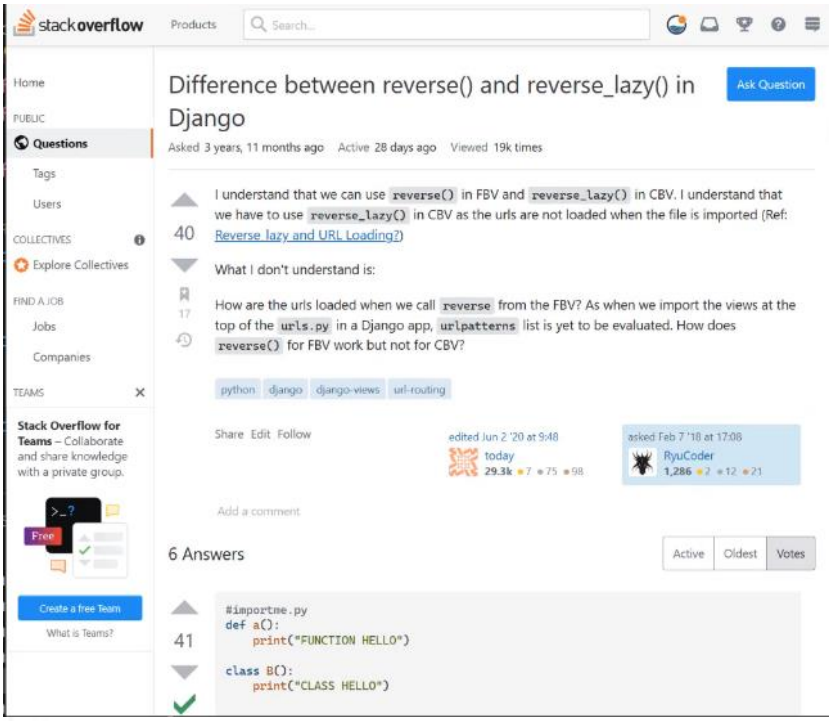
→ check and run server

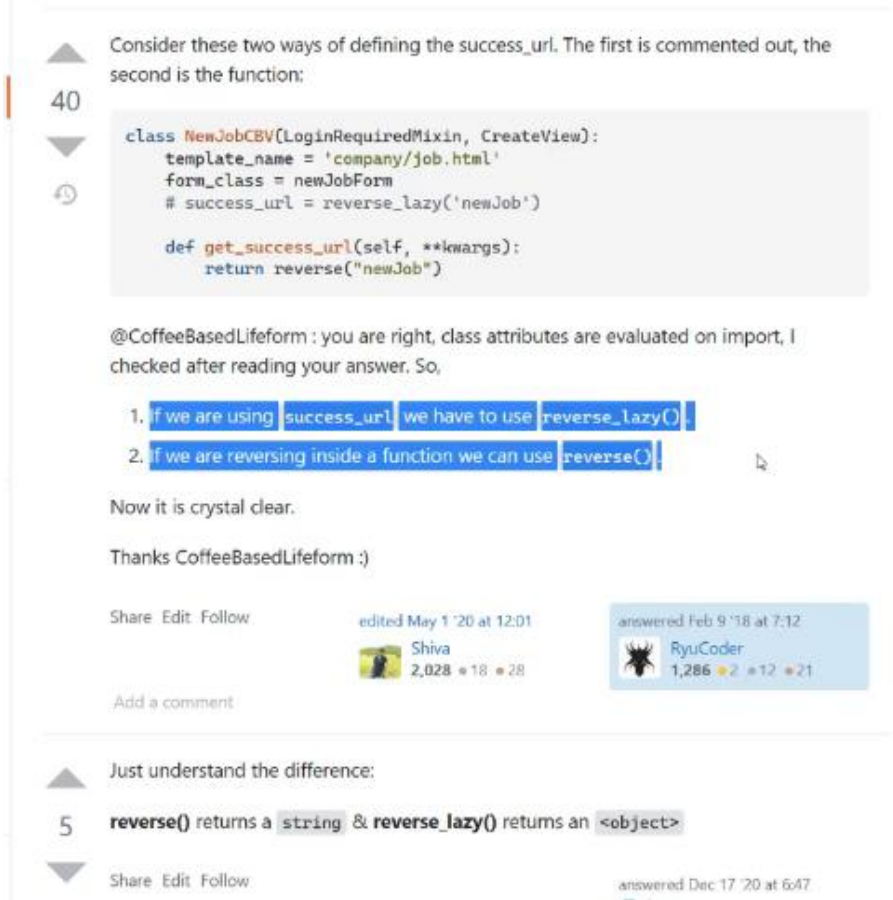
Using reverse_lazy instead to remember URL path! Same like reverse()

```
from django.shortcuts import render
from django.urls import reverse_lazy
from django.views.generic import TemplateView, FormView

from classroom.forms import ContactForm
```

```
# success URL? It is the actual URL NOT a template.html
    # success_url = "/classroom/thank_you/"
    success_url = reverse_lazy('classroom:thank_you')
```

Difference between reverse() and reverse_lazy()





https://www.folkstalk.com/tech/where-to-import-reverse-lazy-in-django-with-code-examples/

54. Django CBV – CreateView

The next few lectures are going to highlight one of the best features of Class Based Views - Model based CBVS.

There are a few operations that are very common with models: Create, Detail, Update, Delete, List.

Django provides CBVs that automatically create the appropriate views, forms, and context objects for predefined template names by simply being connected to a model.

These classes require just a few attributes and automatically do the work for you!

Important Note!

- Because the classes are designed to be simple, these views **require** a template name to follow a specific pattern, for example:
    - model_form.html
        - teacher_form.html
- This factor is often confusing to students because it seems like Django "magically" knew a template .html file existed, but it's just looking for a specific naming convention pattern.

→ create a Model

```
from django.db import models


# Create your models here.


class Teacher(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    subject = models.CharField(max_length=30)


    def __str__(self):
        return f'{self.first_name} {self.last_name} teaches {self.subject}'
```

On views.py

→ import class **CreateView**
→ from django.views.generic import CreateView

→ import model
→ from classroom.models import Teacher

```python
from django.shortcuts import render
from django.urls import reverse_lazy
from django.views.generic import TemplateView, FormView, CreateView
from classroom.models import Teacher
from classroom.forms import ContactForm

# Create your views here.

class HomeView(TemplateView):
    template_name = "classroom/home.html"

class ThankYouView(TemplateView):
    template_name = "classroom/thank_you.html"

class TeacherCreateView(CreateView):
    model = Teacher
    # model_form.html
    # .save()
    fields = "__all__"
    success_url = reverse_lazy('classroom:thank_you')




class ContactFormView(FormView):
    form_class = ContactForm
    template_name = "classroom/contact.html"

    # success URL? It is the actual URL NOT a template.html
    # success_url = "/classroom/thank_you/"
    success_url = reverse_lazy('classroom:thank_you')

    # what to do with form
    def form_valid(self, form):
```

```
        print(form.cleaned_data)
        return super().form_valid(form) # similar to ContactForm(request.Post)
```

Students often ask is how does it actually know what template to connect to notice?

→ Create teacher_form.html

```
<h1> TEACHER FORM </h1>
<form method='POST'>
    {% csrf_token %}
    {{form.as_p}}
    <input type="submit" value="Submit">
</form>
```

TeacherCreateView looks up at the particular Templates with model_form.html through the help of CreateView and model = Teacher

In this case it looks up at teacher_form.html and it will auto create a model form for you, an then we'll set that up

→ another cool thing is the second you hit the submit button, its pretty much automatically going to hit save after all, the fields are validated
→ so it's doing a lot of work for you

CAVEATE → It is only creating a new instance in that model

→ connect to html for taking you to the create_teacher page

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home</title>
</head>
<body>
    <h1>Welcome to the Classroom</h1>
    <ul>
        <li><a href="{% url 'classroom:thank_you' %}">Take me to the Classroom</a></li>
```

```
        <li><a href="{% url 'classroom:contact' %}">Take me to the Contact Form</a></li>
        <li><a href="{% url 'classroom:create_teacher' %}">Create New Teacher Page</a></li>
    </ul>
</body>
</html>
```

→ connect to url

```
from django.urls import path
from classroom.views import HomeView, ThankYouView, ContactFormView, TeacherCreateView

app_name = 'classroom'

urlpatterns = [
    path('', HomeView.as_view(), name='home'),  # path expects a function
    path('thank_you/', ThankYouView.as_view(), name='thank_you'),
    path('contact/', ContactFormView.as_view(), name='contact'),
    path('teacher/', TeacherCreateView.as_view(), name= 'create_teacher')
]
```

→ don't forget makemigration and then migrate it again

OUTPUT → IT CREATE NEW INSTANCE FOR TEACHER!

Created Gianluca Cannone Django
Mattia Virgilio Musik

But how you can see it? Look at the next lecture

55. Django CBV – ListView

Create a new view that can list all the instances and any particular model

→ import ListView
→ create TeacherListView class and inherit from ListView

→ same as TeacherCreateView →

```
from django.shortcuts import render
```

```python
from django.urls import reverse_lazy
from django.views.generic import TemplateView, FormView, CreateView, ListView
from classroom.models import Teacher
from classroom.forms import ContactForm

# Create your views here.

class HomeView(TemplateView):
    template_name = "classroom/home.html"

class ThankYouView(TemplateView):
    template_name = "classroom/thank_you.html"

class TeacherCreateView(CreateView):
    model = Teacher
    # model_form.html
    # .save()
    fields = "__all__"
    success_url = reverse_lazy('classroom:thank_you')

class ContactFormView(FormView):
    form_class = ContactForm
    template_name = "classroom/contact.html"

    # success URL? It is the actual URL NOT a template.html
    # success_url = "/classroom/thank_you/"
    success_url = reverse_lazy('classroom:thank_you')

    # what to do with form
    def form_valid(self, form):
        print(form.cleaned_data)
        return super().form_valid(form) # similar to ContactForm(request.Post)

class TeacherListView(ListView):
    # model_list.html
    model = Teacher
    # context_object_name = 'teacher_list'
```

→ Create template list
→ teacher_list.html because ListView class is looking to model_list.html

```
<h1>List of Teachers (ListView)</h1>


<ul>
   {% for teacher in object_list %}
      <li>{{teacher.first_name}} {{teacher.last_name}}</li>


   {% endfor %}
</ul>
```

What is object_list??

You can change the name of the list: context_object_name = “teacher_list”

RECOMMENDATION : if you get more views then you should change it for readability

```
class TeacherListView(ListView):
   # model_list.html
   model = Teacher
   # context_object_name = 'teacher_list'
```

→ Connect to urls.py

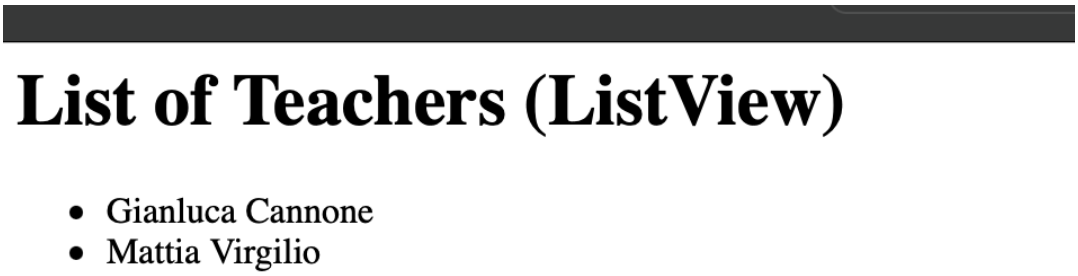
```
from django.urls import path
from classroom.views import (HomeView, ThankYouView,
                        ContactFormView, TeacherCreateView,
                        TeacherListView)


app_name = 'classroom'


urlpatterns = [
   path('', HomeView.as_view(), name='home'),  # path expects a function
   path('thank_you/', ThankYouView.as_view(), name='thank_you'),
   path('contact/', ContactFormView.as_view(), name='contact'),
   path('create_teacher/', TeacherCreateView.as_view(), name='create_teacher'),
```

```
    path('list_teacher/', TeacherListView.as_view(), name='list_teacher')
]
```

→ Connect to home.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home</title>
</head>
<body>
    <h1>Welcome to the Classroom</h1>
    <ul>
        <li><a href="{% url 'classroom:thank_you' %}">Take me to the Classroom</a></li>
        <li><a href="{% url 'classroom:contact' %}">Take me to the Contact Form</a></li>
        <li><a href="{% url 'classroom:create_teacher' %}">Create New Teacher Page</a></li>
        <li><a href="{% url 'classroom:list_teacher' %}">List Teacher Page</a></li>
    </ul>
</body>
</html>
```

Output:

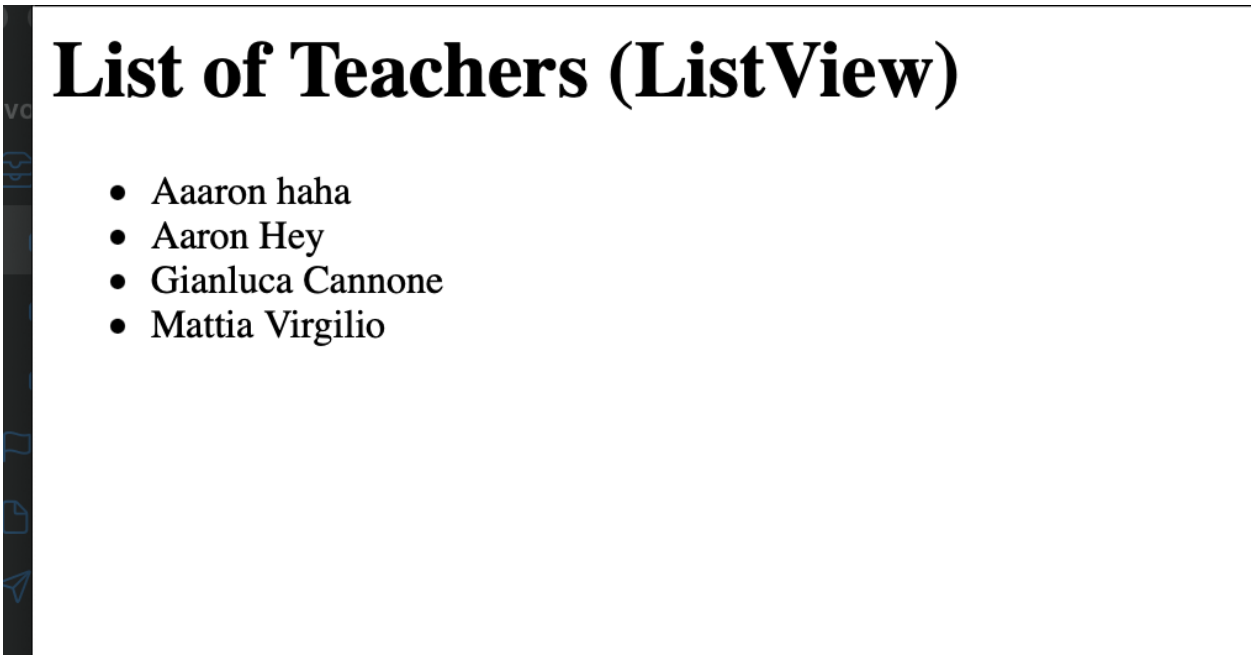


→ Customizing the list view

Overriding queryset (default Teacher.objects.all()

→ queryset = Teacher.objects.order_by('first_name')

```
class TeacherListView(ListView):
    # model_list.html
    model = Teacher
    # context_object_name = 'teacher_list'
    queryset = Teacher.objects.order_by('first_name')
```



56. Django DBV – DetailView

→ Create DetailView class
→ import DetailView

```
from django.shortcuts import render
from django.urls import reverse_lazy
from django.views.generic import TemplateView, FormView, CreateView, ListView, DetailView
from classroom.models import Teacher
from classroom.forms import ContactForm
```

```python
# Create your views here.

class HomeView(TemplateView):
    template_name = "classroom/home.html"

class ThankYouView(TemplateView):
    template_name = "classroom/thank_you.html"

class TeacherCreateView(CreateView):
    model = Teacher
    # model_form.html
    # .save()
    fields = "__all__"
    success_url = reverse_lazy('classroom:thank_you')

class ContactFormView(FormView):
    form_class = ContactForm
    template_name = "classroom/contact.html"

    # success URL? It is the actual URL NOT a template.html
    # success_url = "/classroom/thank_you/"
    success_url = reverse_lazy('classroom:thank_you')

    # what to do with form
    def form_valid(self, form):
        print(form.cleaned_data)
        return super().form_valid(form) # similar to ContactForm(request.Post)

class TeacherListView(ListView):
    # model_list.html
    model = Teacher
    # context_object_name = 'teacher_list'
    queryset = Teacher.objects.order_by('first_name')

class TeacherDetailView(DetailView):
    # Main Idea: RETURN ONLY ONE MODEL ENTRY PK
    # model_detail.html
```

```
    model = Teacher
    #pk --> {{teacher}} (What pk are they actually asking for?)
```

→ Create template teacher_detail.html

```
<h1>Detail View of Teachers</h1>


{{teacher}}
```

→ Connect with list → teacher_list.html

```
<h1>List of Teachers (ListView)</h1>


<ul>
    {% for teacher in object_list %}
        <li> <a href="/classroom/teacher_detail/{{teacher.id}}">{{teacher.first_name}} {{teacher.last_name}}</a></li>


    {% endfor %}
</ul>
```

→ Connect to URLs
→ import TeacherDetailView class

```
from django.urls import path
from classroom.views import (HomeView, ThankYouView,
                             ContactFormView, TeacherCreateView,
                             TeacherListView, TeacherDetailView)


app_name = 'classroom'


# domain.com/classroom/teacher_detail/2/
urlpatterns = [
    path('', HomeView.as_view(), name='home'),  # path expects a function
    path('thank_you/', ThankYouView.as_view(), name='thank_you'),
    path('contact/', ContactFormView.as_view(), name='contact'),
    path('create_teacher/', TeacherCreateView.as_view(), name='create_teacher'),
    path('list_teacher/', TeacherListView.as_view(), name='list_teacher'),
    path('teacher_detail/<int:pk>', TeacherDetailView.as_view(), name='detail_teacher')
]
```

<int:pk> Input will be expected! Since it is connected with list
→ by enter the teacher it will lead you to the datail_view site
→ {{teacher.id}}! In html file

List of Teachers (ListView)

- Aaaron haha
- Aaron Hey
- Gianluca Cannone
- Mattia Virgilio

Detail View of Teachers

Gianluca Cannone teaches Django

57. Django CBV – Update

It's kind of the mix of detail view and create view

Create view → you will be filling out a form to up the information

Detail view → because a specific entry will be changed

All what we are doing here is connecting to the same form we had for create view. Except this time, we are pre filling it with the information for a particular primary key.

→ Create UpdateView class on views.py
→ Don't forget to import UpdateView

```
from django.shortcuts import render
from django.urls import reverse_lazy
from django.views.generic import TemplateView, FormView, CreateView, ListView, DetailView, UpdateView
from classroom.models import Teacher
from classroom.forms import ContactForm

# Create your views here.

class HomeView(TemplateView):
    template_name = "classroom/home.html"

class ThankYouView(TemplateView):
    template_name = "classroom/thank_you.html"

class TeacherCreateView(CreateView):
    model = Teacher
    # model_form.html
    # .save()
    fields = "__all__"
    success_url = reverse_lazy('classroom:thank_you')

class ContactFormView(FormView):
    form_class = ContactForm
    template_name = "classroom/contact.html"

    # success URL? It is the actual URL NOT a template.html
    # success_url = "/classroom/thank_you/"
    success_url = reverse_lazy('classroom:thank_you')

    # what to do with form
```

```
    def form_valid(self, form):
        print(form.cleaned_data)
        return super().form_valid(form) # similar to ContactForm(request.Post)

class TeacherListView(ListView):
    # model_list.html
    model = Teacher
    # context_object_name = 'teacher_list'
    queryset = Teacher.objects.order_by('first_name')

class TeacherDetailView(DetailView):
    # Main Idea: RETURN ONLY ONE MODEL ENTRY PK
    # model_detail.html
    model = Teacher
    #pk --> {{teacher}} (What pk are they actually asking for?)

class TeacherUpdateView(UpdateView):
    # SHARE model_form.html → PK
    model = Teacher
    fields = "__all__"
    success_url = reverse_lazy('classroom:list_teacher')
```

→ Connect to Urls.py
→ don't forget to import UpdateView class

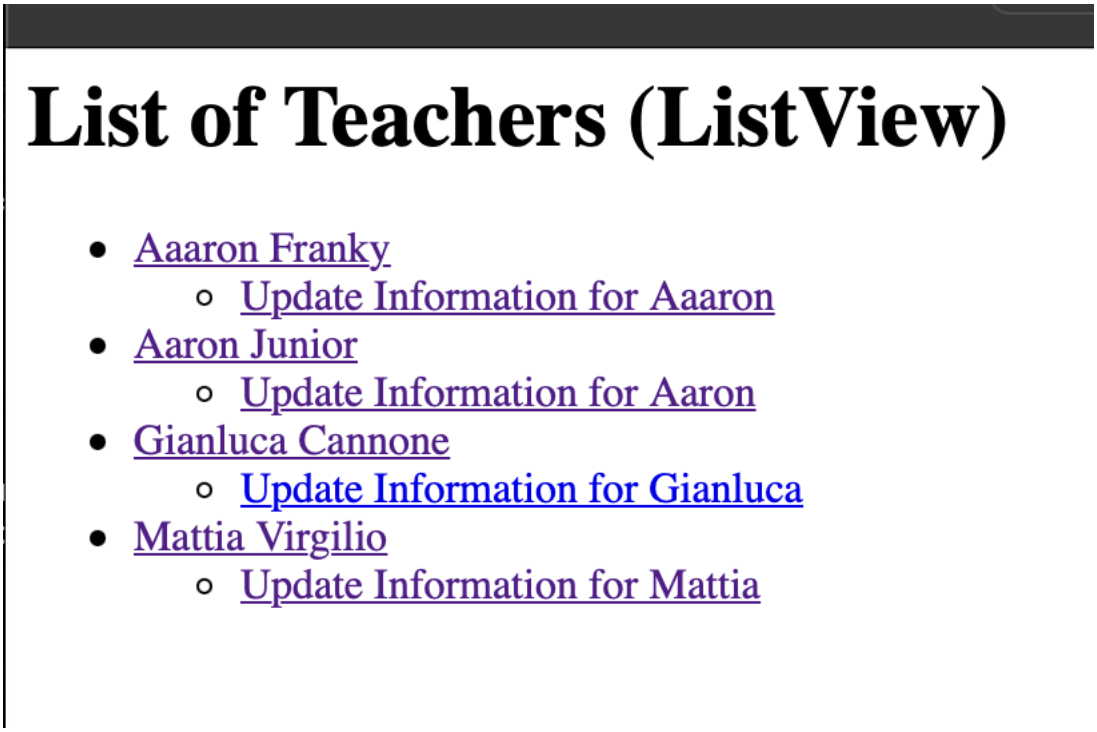
```
from django.urls import path
from classroom.views import (HomeView, ThankYouView,
                             ContactFormView, TeacherCreateView,
                             TeacherListView, TeacherDetailView,
                             TeacherUpdateView)

app_name = 'classroom'

# domain.com/classroom/teacher_detail/2/
urlpatterns = [
    path('', HomeView.as_view(), name='home'),  # path expects a function
```

```
    path('thank_you/', ThankYouView.as_view(), name='thank_you'),
    path('contact/', ContactFormView.as_view(), name='contact'),
    path('create_teacher/', TeacherCreateView.as_view(), name='create_teacher'),
    path('list_teacher/', TeacherListView.as_view(), name='list_teacher'),
    path('teacher_detail/<int:pk>', TeacherDetailView.as_view(), name='detail_teacher'),
    path('update_teacher/<int:pk>', TeacherUpdateView.as_view(), name='update_teacher')
]
```

→ Connect with teacher_list.html

```
<h1>List of Teachers (ListView)</h1>


<ul>
    {% for teacher in object_list %}
        <li> <a href="/classroom/teacher_detail/{{teacher.id}}">{{teacher.first_name}} {{teacher.last_name}}</a></li>
        <ul>
            <li> <a href="/classroom/update_teacher/{{teacher.id}}">Update Information for {{teacher.first_name}}</a></li>
        </ul>
    {% endfor %}
</ul>
```

Output:



58. Django CBV – Delete

→ Create DeleteView class

```
from django.shortcuts import render
from django.urls import reverse_lazy
from django.views.generic import TemplateView, FormView, CreateView, ListView, DetailView, UpdateView,
DeleteView
from classroom.models import Teacher
from classroom.forms import ContactForm

# Create your views here.

class HomeView(TemplateView):
    template_name = "classroom/home.html"

class ThankYouView(TemplateView):
    template_name = "classroom/thank_you.html"

class TeacherCreateView(CreateView):
```

```
    model = Teacher
    # model_form.html
    # .save()
    fields = "__all__"
    success_url = reverse_lazy('classroom:thank_you')

class ContactFormView(FormView):
    form_class = ContactForm
    template_name = "classroom/contact.html"

    # success URL? It is the actual URL NOT a template.html
    # success_url = "/classroom/thank_you/"
    success_url = reverse_lazy('classroom:thank_you')

    # what to do with form
    def form_valid(self, form):
        print(form.cleaned_data)
        return super().form_valid(form) # similar to ContactForm(request.Post)

class TeacherListView(ListView):
    # model_list.html
    model = Teacher
    # context_object_name = 'teacher_list'
    queryset = Teacher.objects.order_by('first_name')

class TeacherDetailView(DetailView):
    # Main Idea: RETURN ONLY ONE MODEL ENTRY PK
    # model_detail.html
    model = Teacher
    #pk --> {{teacher}} (What pk are they actually asking for?)

class TeacherUpdateView(UpdateView):
    # SHARE model_form.html --- PK
    model = Teacher
    fields = "__all__"
    success_url = reverse_lazy('classroom:list_teacher')
```

```
class TeacherDeleteView(DeleteView):
    # Form --> Confirm Delete Button
    # default template name:
    # model_confirm_delete.html
    model = Teacher
    success_url = reverse_lazy('classroom:list_teacher')
```

→ Connect to URLs.py

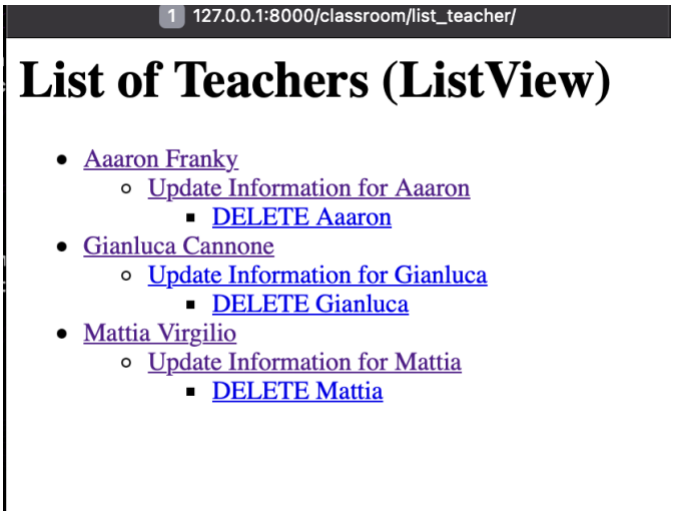
```
from django.urls import path
from classroom.views import (HomeView, ThankYouView,
                             ContactFormView, TeacherCreateView,
                             TeacherListView, TeacherDetailView,
                             TeacherUpdateView, TeacherDeleteView)

app_name = 'classroom'

# domain.com/classroom/teacher_detail/2/
urlpatterns = [
    path('', HomeView.as_view(), name='home'),  # path expects a function
    path('thank_you/', ThankYouView.as_view(), name='thank_you'),
    path('contact/', ContactFormView.as_view(), name='contact'),
    path('create_teacher/', TeacherCreateView.as_view(), name='create_teacher'),
    path('list_teacher/', TeacherListView.as_view(), name='list_teacher'),
    path('teacher_detail/<int:pk>', TeacherDetailView.as_view(), name='detail_teacher'),
    path('update_teacher/<int:pk>', TeacherUpdateView.as_view(), name='update_teacher'),
    path('delete_teacher/<int:pk>', TeacherDeleteView.as_view())
]
```

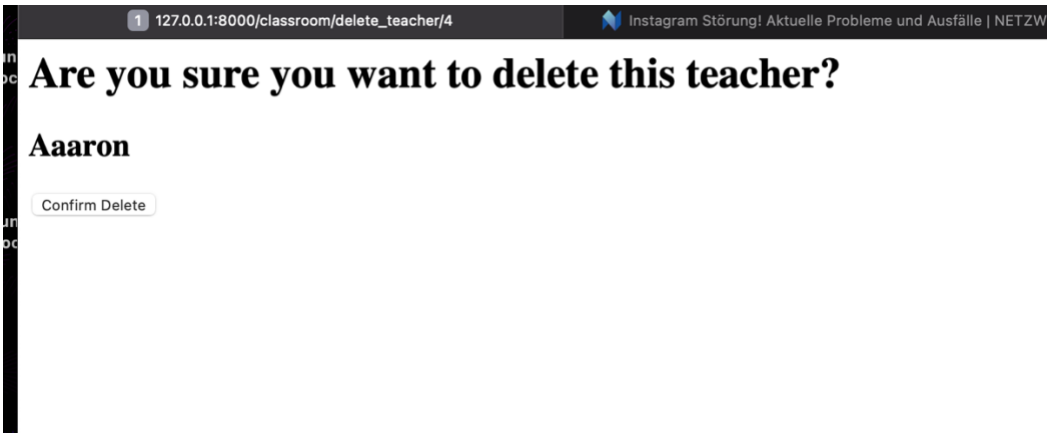
→ Create html. File → **teacher_confirm_delete.html**

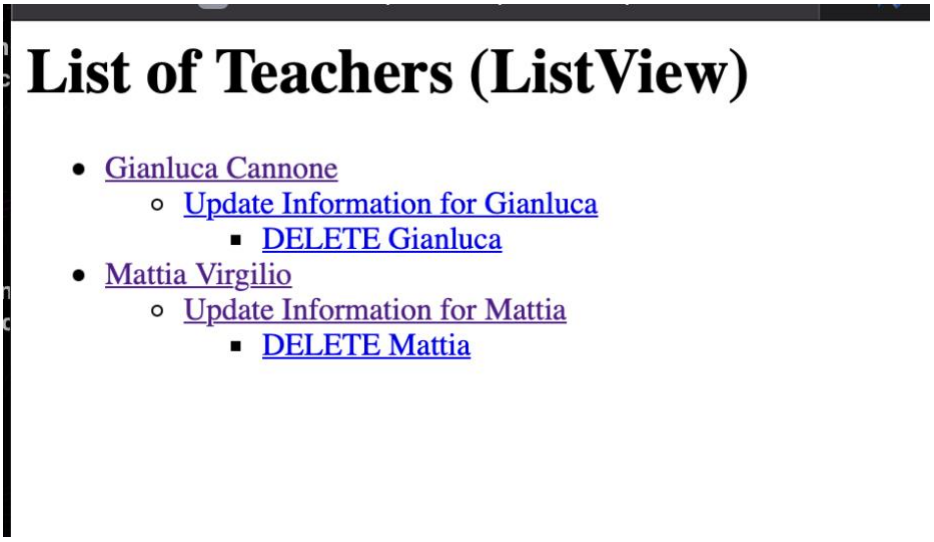
```
<h1>Are you sure you want to delete this teacher?</h1>
<h2>{{teacher.first_name}}</h2>

<form method="POST">
    {% csrf_token %}
```

```
    <input type="submit" value="Confirm Delete">
</form>
```

Output:







CAVEATS: Class-based views do not cover every specific function. Therefore, a function-based view must be created for certain scenarios

Section 15: User Authentication and Session

59. Project Skeleton

Create a project "Library"

- Create Project Library
- Create App Catalog
- Create Templates
- Create View and import class Template View (CBV)
- Connect URLs.py (App level)
- Connect URLs.py (Project Level)
- Set app on Settings.py

60. Model Setup

Let's continue setting up our library with some models, we'll need:

- Book
- Genre
- Language
- Author
- BookInstance - Specific physical copy

Users of our library will eventually be able to check out a BookInstance, we can have multiple BookInstances of the same Book.

- For example, multiple physical book copies of the book "Catcher in the Rye".

```
from django.db import models
from django.urls import reverse

# Create your models here.

class Genre(models.Model):

    name = models.CharField(max_length = 150)

    def __str__(self):
        return self.name

class Language(models.Model):
    name = models.CharField(max_length=200)

    def __str__(self):
        return self.name


class Book(models.Model):

    title = models.CharField(max_length=200)
    author = models.ForeignKey('Author', on_delete = models.SET_NULL, null=True)
    summary = models.TextField(max_length=600)
    isbn = models.CharField('ISBN', max_length=13, unique=True)
    genre = models.ManyToManyField(Genre)
    language = models.ForeignKey('Language', on_delete = models.SET_NULL, null =True)

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse("book_detail", kwargs={"pk": self.pk})
```

```python
class Author(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)
    date_of_birth = models.DateField(null=True, blank=True)

    class Meta:
        ordering = ['last_name', 'first_name']

    def get_absolute_url(self):
        return reverse("author_detail", kwargs={"pk": self.pk})

    def __str__(self):
        return f"{self.last_name}, {self.first_name}"

import uuid

class BookInstance(models.Model):

    id = models.UUIDField(primary_key=True, default=uuid.uuid4)
    book = models.ForeignKey('Book', on_delete= models.RESTRICT, null=True)
    imprint = models.CharField(max_length=200)
    due_back = models.DateField(null=True, blank=True)

    LOAN_STATUS = (
        ('m', 'Maintenance'),
        ('o', 'On Loan'),
        ('a', 'Available'),
        ('r', 'Reserved')
    )

    status = models.CharField(max_length=1, choices=LOAN_STATUS, blank=True, default='m')

    class Meta:
        ordering = ['due_back']

    def __str__(self):
```

```
return f'{self.id} ({self.book.title})'  # type: ignore
```

Book and author should have his own model!!

An author can have written many books! So we can link with ForeignKey to the books that they have written

What happens if delete this book?

Would you also delete the author, or would you delete the book if you happened to delete the author? one way or the other.
If you happen to on_delete the author you will set it as null for this book

CASCADE is the other method → IF author will be deleted the book will be as well

Class Meta:
→ dictate behavior inside the admin view
→ for instance sorted by ordered by

get_absolute_url(self)
→ Its going to return the URL to access a record for this book

Many to many connections:
models.ManyToManyField(class)
if a model has more model connection at the same time
→ genre = models.ManyToManyField(Genre)

Import uuid:
→Unique identifier generator
→ It just generate the unique ID

So essentially what's happening here is the book instance represents a specific copy of a book that someone might borrow and includes information maybe about whether the copies available or what date it's actually expected back
So we should know, hey this particular book is actually checked out for the library

RESTRICT
→ This essentially restricts you from deleting a book if you still have a book instead

→ So, what happens when you actually do a deletion of a book?
So, you have to delete all the book instances first before you're allowed to delete that book

61. Admin Setup

Let's now:

- Register our Models
- Create a Superuser
- Create Example Instances

```
from django.contrib import admin
from catalog.models import Author, Genre, Language, Book, BookInstance

# Register your models here.

admin.site.register(Author)
admin.site.register(Genre)
admin.site.register(Language)
admin.site.register(Book)
admin.site.register(BookInstance)
```

→ python manage.py createsuperuser
User: admin
Password: password

Create example Instances on admin page

62. Page Setup

→ Create IndexView and set context (attributes)

```
from django.shortcuts import render
from django.views.generic import TemplateView, CreateView
from catalog.models import Book, Author, BookInstance, Genre, Language

# Create your views here.

class IndexView(TemplateView):
    template_name = 'catalog/index.html'
    # Attribute!
    extra_context = {'num_books': Book.objects.all().count(),
                     'num_instances': BookInstance.objects.all().count(),
```

```
                'num_instances_avail': BookInstance.objects.filter(status__exact = 'a').count()}

    # Alternative in CBV with method
    # def get_context_data(self, **kwargs):
    #     context = super().get_context_data(**kwargs)
    #     context['num_books'] = Book.objects.all().count()
    #     context['num_instances'] = BookInstance.objects.all().count()
    #     context['num_instances_avail'] = BookInstance.objects.filter(status__exact = 'a').count()
    #     return context

    # FBV
# def index(request):

    # num_books = Book.objects.all().count()
    # num_instances = BookInstance.objects.all().count()a
    # num_instances_avail = BookInstance.objects.filter(status__exact = 'a').count()

    # context = {
    #     'num_books': num_books,
    #     'num_instances': num_instances,
    #     'num_instances_avail': num_instances_avail
    # }

    # return render(request, 'catalog/index.html', context = context)
```

Difference between get_context_data vs extra_context

0

They both do the same functionality, except that in `get_context_data` to get prev data you should call `super` and update that dict; cause you don't want to lose prev data in context.

But `extra_context` will do the same but no need to call super and you can call it in url like `TemplateView.as_view(extra_context={'title': 'Custom Title'})`.

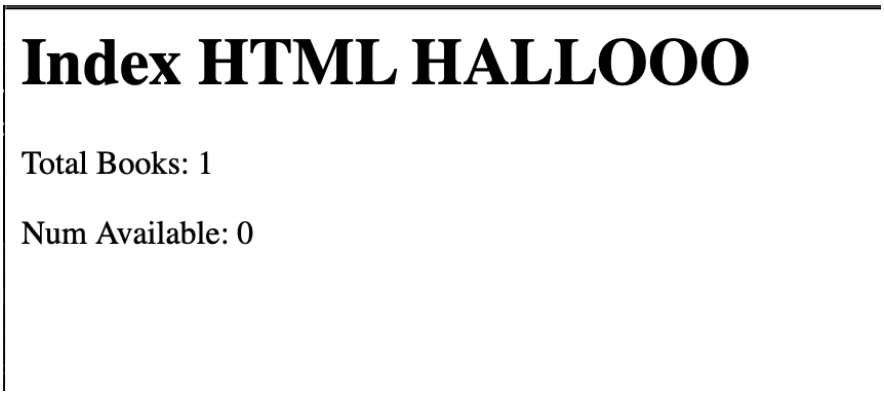
So to sum up, `extra_context` is just a faster with less code to update your context data in CBV. Also to **note**, `extra_context` will be used as an attribute, but `get_context_data` is the method for CBVs.

→ Continue with index.html

```
<html lang="en">
<head>
```

```
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home</title>
</head>
<body>

    <h1>Index HTML HALLOOO</h1>
    <p> Total Books: {{num_books}}</p>
    <p> Num Available: {{num_instances_avail}}</p>
</body>
</html>
```

Output:



→ Create Class BookCreate and inherit from CreateView

HINT: import CreateView

```
from django.views.generic import TemplateView, CreateView
```

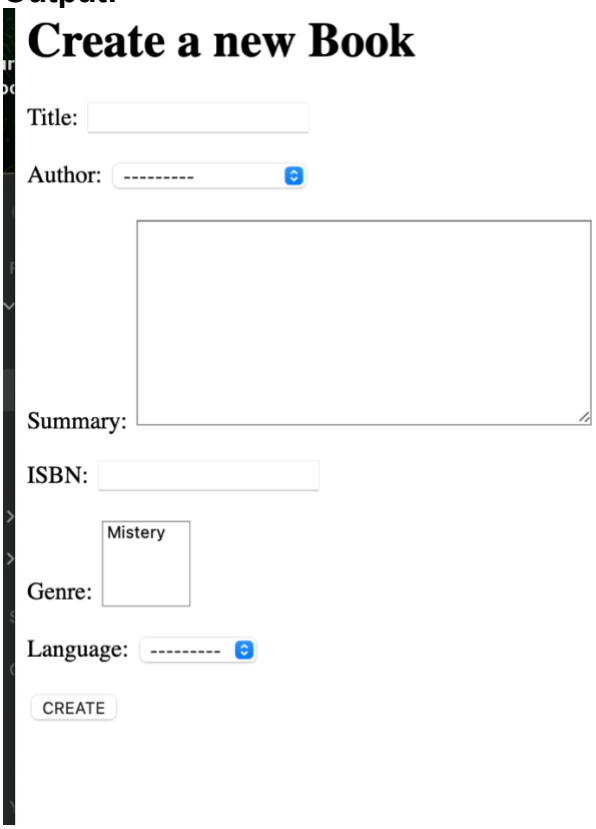
```
class BookCreate(CreateView):   #model_form.html
    model = Book
    fields = '__all__'
```

→ Create model_form.html

```
<h1> Create a new Book</h1>


<form method="Post">
```

```
    {% csrf_token %}
    {{form.as_p}}
    <input type="submit" value='CREATE'>
</form>
```

Output:



→ Create DetailView

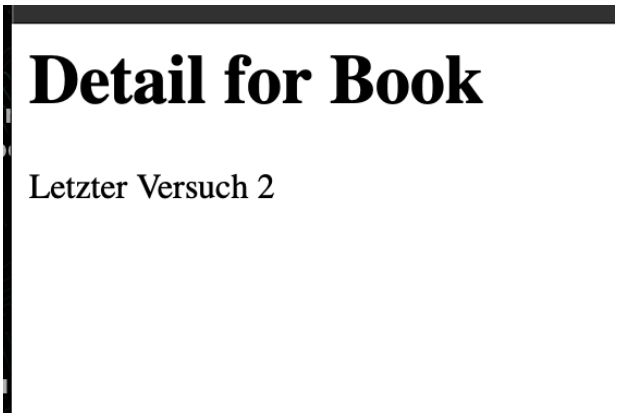
```
from django.shortcuts import render
from django.urls import reverse_lazy
from django.views.generic import TemplateView, CreateView, DetailView
from catalog.models import Book, Author, BookInstance, Genre, Language


# Create your views here.


class IndexView(TemplateView):
    template_name = 'catalog/index.html'
    # Attribute!
    extra_context = {'num_books': Book.objects.all().count(),
```

```
                'num_instances': BookInstance.objects.all().count(),
                'num_instances_avail': BookInstance.objects.filter(status__exact = 'a').count()}

    # Alternative in CBV with method
    # def get_context_data(self, **kwargs):
    #     context = super().get_context_data(**kwargs)
    #     context['num_books'] = Book.objects.all().count()
    #     context['num_instances'] = BookInstance.objects.all().count()
    #     context['num_instances_avail'] = BookInstance.objects.filter(status__exact = 'a').count()
    #     return context

    # FBV
# def index(request):

    # num_books = Book.objects.all().count()
    # num_instances = BookInstance.objects.all().count()a
    # num_instances_avail = BookInstance.objects.filter(status__exact = 'a').count()

    # context = {
    #     'num_books': num_books,
    #     'num_instances': num_instances,
    #     'num_instances_avail': num_instances_avail
    # }

    # return render(request, 'catalog/index.html', context = context)

class BookCreate(CreateView):   #model_form.html
    model = Book
    fields = '__all__'

    # success_url = reverse_lazy('catalog:index')
    # Default --> DetailView

class BookDetail(DetailView):
    model = Book
```

→ Connect to Urls.py

```
from django.urls import path
from . import views

# app_name = 'catalog'

urlpatterns = [
    # path('', views.index, name='index')
    path('', views.IndexView.as_view(), name='index'),
    path('create_book/', views.BookCreate.as_view(), name='create_book'),
    path('book/<int:pk>', views.BookDetail.as_view(), name = 'book_detail')

]
```

Output:



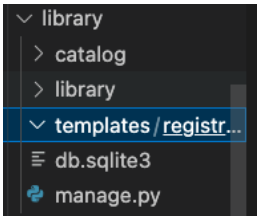
63. User Authentication with Django User Model

We've set up some very simple views and skeleton code for the basis of our Library website.

Clearly, we could keep adding more views/templates/urls to add more pages, but let's focus on adding **users**.

→ On admin site
→ Create Group
→ Create User

→ Create Templates on project level
Registration and logging in… don't happen at an application level
It happens at a site level, which is why underneath templates a default **registration folder** has to be created!



→ **CREATE login.html** underneath registration

→ Connect with URL.py

```
from django.contrib import admin
from django.urls import path, include
from django.views.generic import RedirectView

urlpatterns = [
    path('admin/', admin.site.urls),
    path('catalog/', include('catalog.urls')),
    path('', RedirectView.as_view(url='catalog/')),
    path('accounts/', include('django.contrib.auth.urls'))
]
```

RedirectView → lead to default catalog

Different function with accounts/ ….

accounts/ login/ [name= 'login']
accounts/ logout/ [name= 'logout']
accounts/ password change/ [name= 'password_change']
accounts/ password change/done/ [name= 'password change_done]
accounts/ password reset/ [name= 'password_reset']
accounts/ password reset/done/ [name= 'password _reset_done']
accounts / reset/<uidb64>/<token>/ [name= 'password _reset_confirm'] accounts/ reset/done/ [name= 'password_reset_complete j

→ Set settings.py and import os
→ [os.path.join(BASE_DIR, 'templates')]

→ create content on login.html

```
{% comment %} check errors  {% endcomment %}
{% if form.erros %}
    <p> Your username or password was incorrect. Try again.</p>
{% endif %}

{% if next %}
    {% if user.is_authenticated %}
        <p> You dont have permission for this page</p>

    {% else %}
        <p> Please login to see this page </p>
    {% endif %}

{% endif %}

<form method='POST' action="{% url 'login' %}">
{% csrf_token %}
{{form.username.label_tag}}
{{form.username}}


{{form.password.label_tag}}
{{form.password}}
<input type="submit" value='login'>
<input type="hidden" name='next' value="{{next}}">
</form>

{% comment %} -----USER logged in but no access
----USER not logged in
---- FORM LOGGIN{% endcomment %}
```

is_authenticated:

Read-only attribute which is always True (as opposed to AnonymousUser.is_authenticated which is always False). This is a way to tell if the user

has been authenticated. This does not imply any permissions and doesn't check if the user is active or has a valid session. Even though normally you will check this attribute on request.user to find out whether it has been populated by the AuthenticationMiddleware (representing the currently logged-in user), you should know this attribute is True for any User instance.

Output:

http://127.0.0.1:8000/accounts/login/

→ It takes you to : accounts/profile/

→ SET settings.py

```
STATIC_URL = 'static/'

# Default primary key field type
# https://docs.djangoproject.com/en/4.1/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'

LOGIN_REDIRECT_URL = '/'
```

Try to login again it leads you to catalog base template → index.html

HINT: without having RedirectView add:

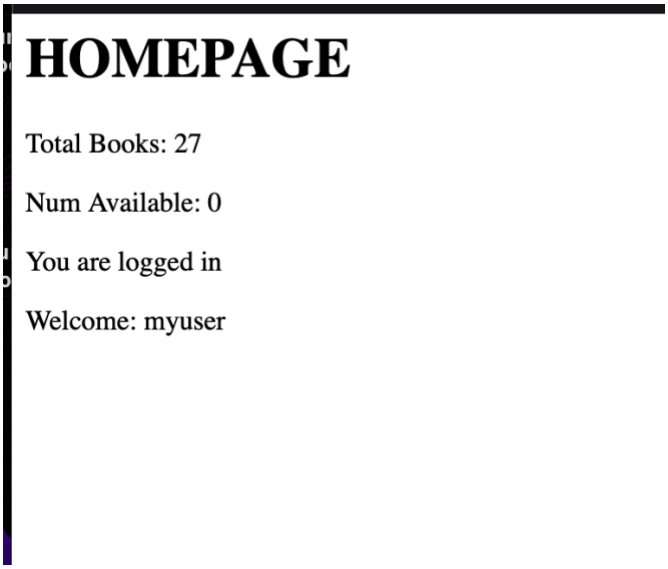
```
LOGIN_REDIRECT_URL = '/catalog'
```

64. User Authentication on Views

→ Add if condition for log in and logout in index.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home</title>
</head>
<body>

    <h1>HOMEPAGE</h1>
    <p> Total Books: {{num_books}}</p>
    <p> Num Available: {{num_instances_avail}}</p>
    {% if user.is_authenticated %}
        <p> You are logged in </p>
        <p> Welcome: {{ user.get_username}} </p>
    {% else %}
        <p> You are not logged in </p>
    {% endif %}
</body>
</html>
```

→ logging to the system



→ Create logged_out.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Home</title>
</head>
<body>
    <p> You have logged_out.html </p>
    <a href="{% url 'login' %}">CLick here to Login!</a>

</body>

</html>
```

I already checking if user is authenticated
I do not want to show a log out button if the user hasn't logged in yet. Be careful when you're designing your pages and templates that you are always displaying a logout or login button. That should actually be conditionally chosen based off if the user is authenticated or not.

In index.hmtl → If user is authenticated they are logged in. Welcome User and give user the possibility to log out!

Lots different ways I can do this

One simple way:

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

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Authentication

→ On index.html

```
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Home</title>
</head>
<body>

   <h1>HOMEPAGE</h1>
   <p> Total Books: {{num_books}}</p>
   <p> Num Available: {{num_instances_avail}}</p>
   {% if user.is_authenticated %}
      <p> You are logged in </p>
      <p> Welcome: {{ user.get_username}} </p>
      <a href="{% url 'logout' %}?next={{request.path}}">Logout Here</a>
   {% else %}
      <p> You are not logged in </p>
      <a href="{% url 'login' %}?next={{request.path}}">Login Here</a>
   {% endif %}
</body>
</html>
```

With next method:

On both login and logout Django will take you back to the homepage

HOMEPAGE

Total Books: 27

Num Available: 0

You are logged in

Welcome: myuser

Logout Here

→ **By login out**

HOMEPAGE

Total Books: 27

Num Available: 0

You are not logged in

Login Here

→ Without having the next method it takes you to the logged_out.

What is better? Depends on how you want it!

You have been logged out!

CLick here to Login!

FBV Lock webpages with Logging requirements!

→ **Using Decorater**
→ Create new html file
→ my_view.html

```
<h1> Only Logged In User can see the content </h1>
```

→ **On views.py import:**

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

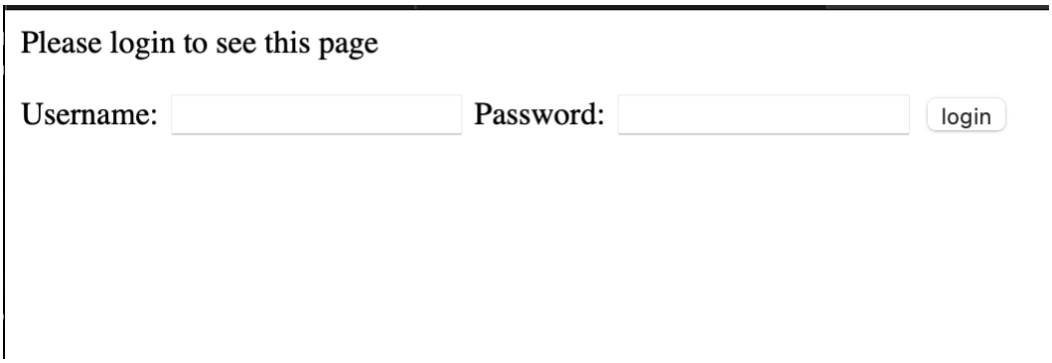
```
@login_required
def my_view(request):
    return render(request, 'catalog/my_view.html')
```

→ **Connect to URLs**

Without @login_required decorator!

Only Logged In User can see the content

With @login_required decorator!



Doesn't let the user see the content anymore!
After login it let you see the content again!

CBV Lock webpages with Logging requirements!

→ Import LoginRequiredMixin

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

→ Inherit from class

```
class BookCreate(LoginRequiredMixin,CreateView):   #model_form.html
    model = Book
    fields = '__all__'

    # success_url = reverse_lazy('catalog:book_detail')
    # Default --> DetailView
```

→ by enter /catalog/create_book

→ **You have to Login to create a book!!**

65. User Registration and Forms

→ Registration an User with class based forms

Import UserCreationForm:

```
from django.contrib.auth.forms import UserCreationForm
```

→ Create a class with CreateView

```

class SignUpView(CreateView):
    form_class = UserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'catalog/signup.html'
```

→ Connect Urls.py

```
from django.urls import path
from . import views

# app_name = 'catalog'

urlpatterns = [
    # path('', views.index, name='index')
    path('', views.IndexView.as_view(), name='index'),
    path('create_book/', views.BookCreate.as_view(), name='create_book'),
    path('book/<int:pk>', views.BookDetail.as_view(), name = 'book_detail'),
    path('myview/', views.my_view, name= 'myievw'),
    path('signup/', views.SignUpView.as_view(), name='signup')


]
```

→ Create Template signup.html

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Register</title>
</head>
<body>
    <h1>Register New User Here</h1>
```

```
  <form method="POST">

    {% csrf_token %}
    {{form.as_p}}
    <input type="submit" value="Signup">
  </form>
</body>
</html>
```

66. User Specific Page

Creating specific page for a user

On models.py

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

→ add borrower and use User

```
class BookInstance(models.Model):

  id = models.UUIDField(primary_key=True, default=uuid.uuid4)
  book = models.ForeignKey('Book', on_delete= models.RESTRICT, null=True)
  imprint = models.CharField(max_length=200)
  due_back = models.DateField(null=True, blank=True)
  borrower = models.ForeignKey(User, on_delete=models.SET_NULL, null=True, blank=True)

  LOAN_STATUS = (
    ('m', 'Maintenance'),
    ('o', 'On Loan'),
    ('a', 'Available'),
    ('r', 'Reserved')
  )

  status = models.CharField(max_length=1, choices=LOAN_STATUS, blank=True, default='m')

  class Meta:
    ordering = ['due_back']
```

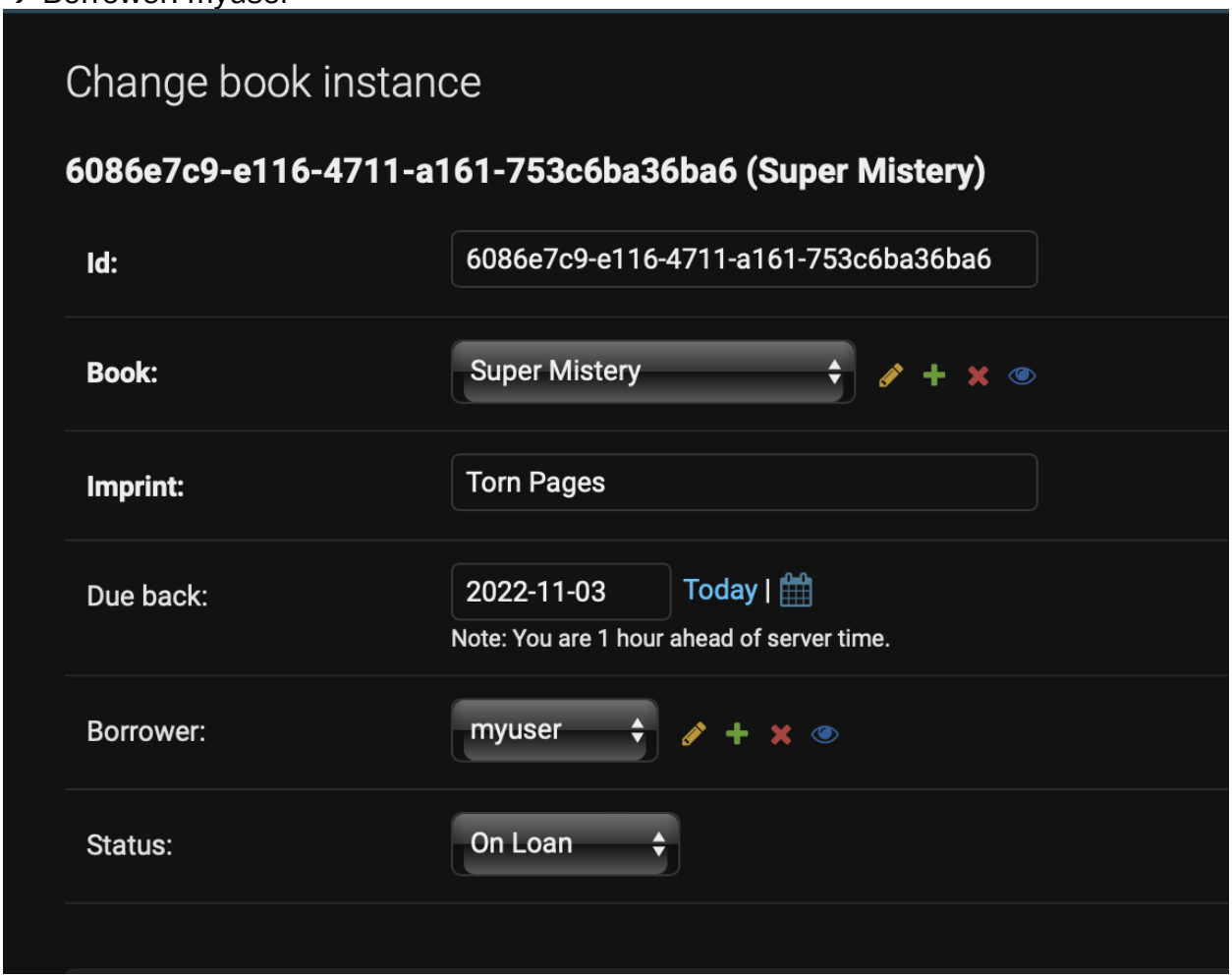
```
def __str__(self):

    return f'{self.id} ({self.book.title})'  # type: ignore
```

→ makemigrations

→ run migrate

→ Change book instance on admin view site

→ Borrower: myuser



Set up a page where myuser can log in an then actually see the books that they have checked out

```
class CheckedOutBooksByUserView(LoginRequiredMixin, ListView):
    # List all BookInstances But I will filter based off currently logged in user session
```

```
    model = BookInstance
    template_name = 'catalog/profile.html'
    paginate_by = 5 # 5 book instances per page

    def get_queryset(self):
        return BookInstance.objects.filter(borrower = self.request.user)
```

→ Explanation:

```
def get_queryset(self):
        return BookInstance.objects.filter(borrower = self.request.user)
```

Essentially all that means is when a person is logged in and they are able to reach this template profile.html then their request as they visit that page is going to have their user information

You can order by something

→ connect to urls.py

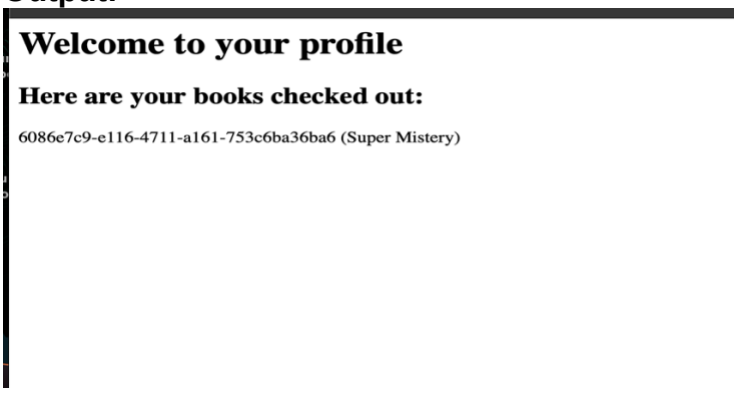
```
from django.urls import path
from . import views


# app_name = 'catalog'


urlpatterns = [
    # path('', views.index, name='index')
    path('', views.IndexView.as_view(), name='index'),
    path('create_book/', views.BookCreate.as_view(), name='create_book'),
    path('book/<int:pk>', views.BookDetail.as_view(), name = 'book_detail'),
    path('myview/', views.my_view, name= 'myievw'),
    path('signup/', views.SignUpView.as_view(), name='signup'),
    path('profile/', views.CheckedOutBooksByUserView.as_view(), name ='profile')



]
```

→ Create profile.html

```
{% comment %} bookinstance_list generic --> Listview  {% endcomment %}
```

```
<h1> Welcome to your profile </h1>
<h2> Here are your books checked out: </h2>


{% for book in bookinstance_list %}
    <p>{{book}}</p>


{% endfor %}
```

Output:



Section 16: Django Linode Deployment

67. Introduction Linode Deployment

Let's explore how to deploy our Django application to the web so anyone can visit our site!

Let's think about our key requirements...

Key Requirements:

- Anyone can visit our website online.
- We don't want to concern ourselves about uptime or resiliency.
- Need to support Python/Django.
- Need to connect to cloud server.
- Need to be able to push updates to our code and have version control.

Cloud Service Provider:

- Hosts a computer/server with our Django application.
- There are ***many*** cloud service providers!
- Often it is a trade-off between ease of use and price.

Cloud Service Provider:

- A cloud provider with an nice balance between price and ease of use is **Linode.**

- We can use Linode to setup an online server that contains our Django Web Application.

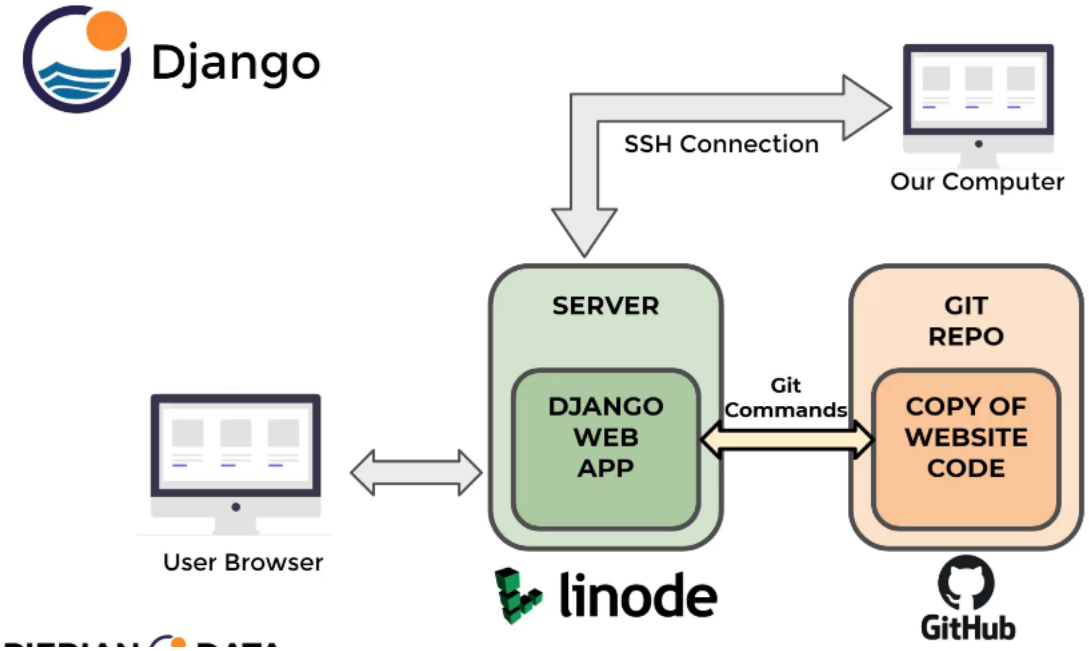
Pierian Data Link for $100 Linode Credit:
www.linode.com/lp/try/?ifso=pierian

IMPORTANT NOTE:
- Any major cloud service provider will require your credit card information in case you go beyond the credit or free tier limits.
- By continuing with this section of the course, you understand that you may be charged if you exceed the limits.

IMPORTANT NOTE:
- What we show in this lecture should easily fall within the $100 credit limit, but cloud services charge monthly, and eventually the credits will be used up!

IMPORTANT NOTE:
- If you have any questions on pricing or charge information, contact your cloud hosting provider.
- We will **NOT** give any advice regarding payments or credit limits.

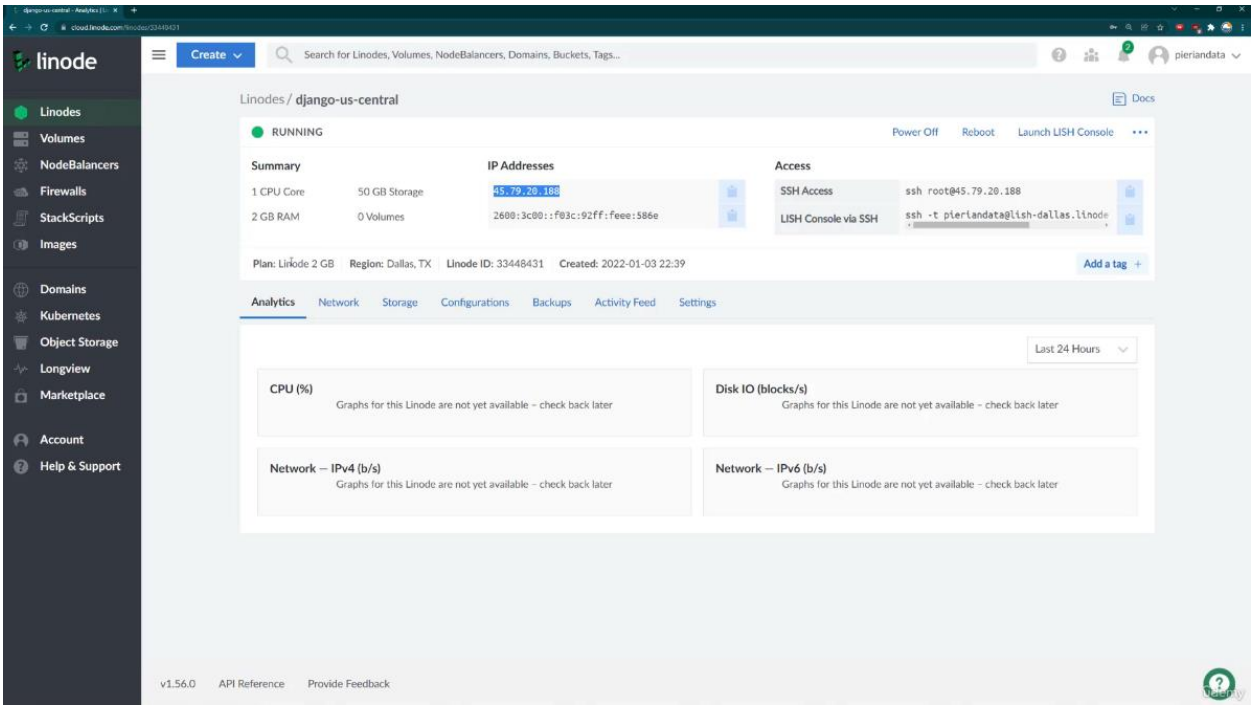
Version Control:
- We want a way to edit our websites code and keep track of changes.
- For this we use a system called **git** that keeps track of versions, and we use **GitHub** to store our code.



- We'll start off by setting up our DjangoWeb Application using Linode's marketplace.

- This will do a lot of setup work for us automatically, by starting a new Django project and linking it to an IP address on the web at a port.

68. Linode Setup



69. SSH Connection

- SSH (Secure SHell) allows us to securely connect to the Linode server that is hosting our Django project.

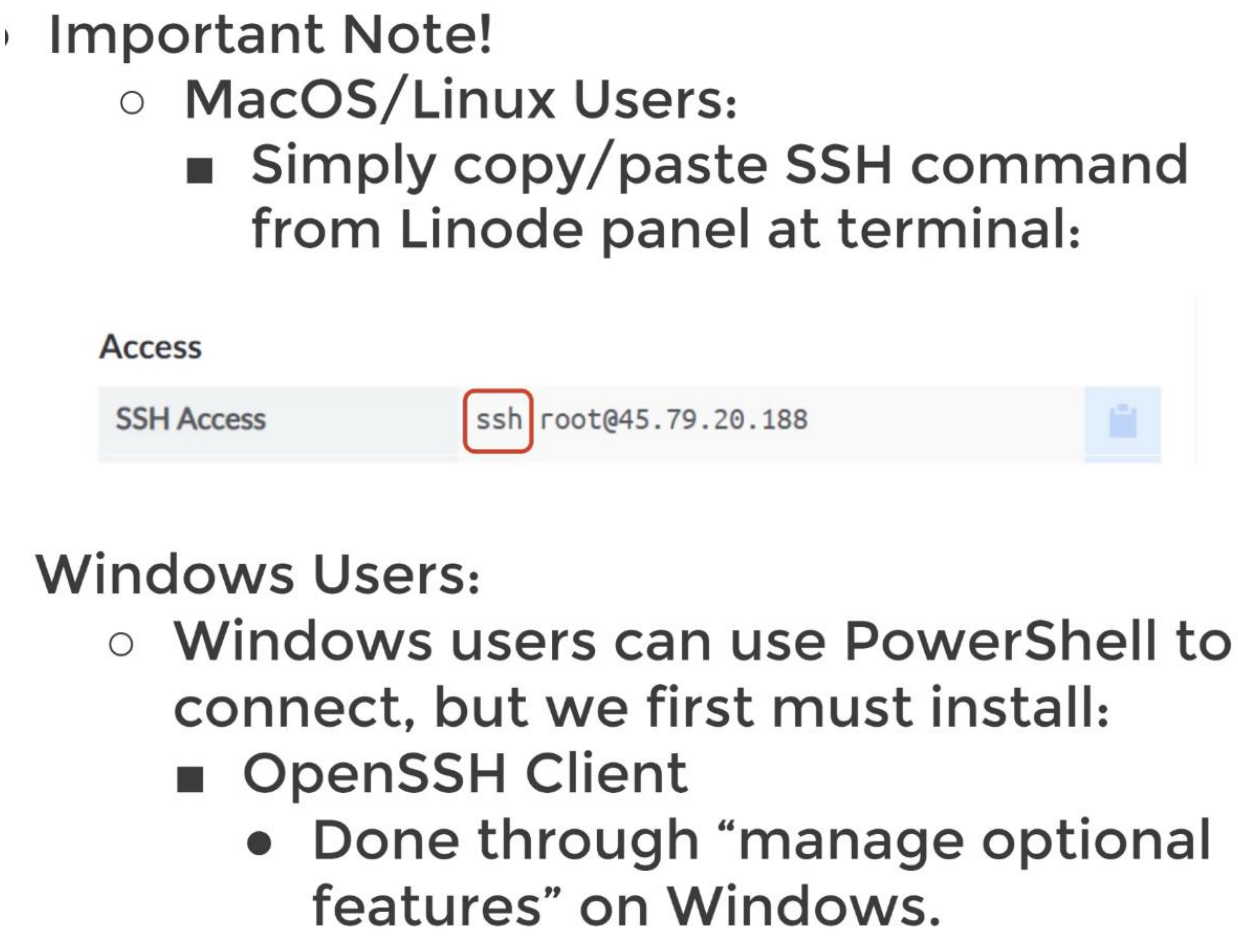
- We don't want just anybody to be able to visit this computer without permission, so we do it through SSH which requires the password you setup during the Linode Setup process in the previous lecture.

- SSH can be intimidating for first time users, but it's actually a simple way of connecting to a computer through the internet securely.
- Just point to the computer (IP Address) and confirm permission through the Password.

- Once we connect through SSH, we'll have access to the Debian Linux shell located at our Linode Server.

- This means we'll be able to use Linux Commands at the Terminal for installations and setups.

Important Note!
→ Detailed instructions on connecting to Linode through SSH:
- www.linode.com/docs/guides/ connect-to-server-over-ssh-on-mac/
- www.linode.com/docs/guides/ connect-to-server-over-ssh-on-wind ows/



→ We'll quickly install OpenSSH Client on a Windows machine, get SSH access, and then all users can follow along at the Debian terminal with the same commands.

70. Version Control with git and GitHub

The final step is to be able to store a copy of our code on GitHub (either public or private) and then update our server's

Django project code with any code we've updated on GitHub.
Create a free GitHub account before continuing.

Important Note:

- This lecture assumes some very basic knowledge of git and GitHub and its overall use for version control.
- If you have never used GitHub or git before, do the GitHub tutorial first! docs.github.com/en/get-started/ quickstart/hello-world

We'll need to perform the following steps:

- SSH Connection to Linode Server
- Install git on Linode
- Create GitHub Repository
- Connect Linode git repo to GitHub
- Connect to Github Repo Locally
- Push/Pull Changes from LocalComputer to Linode Server