



Django pt. 1: Intro, Setup, and Views

Kianoosh Abbasi

CSC309 Fall 2022

So far

- How **web** works

Client/server – request/response - **HTTP**

- **Front-end**

HTML Tags: headers, inputs, etc.

CSS Styles: Selectors, spacing, layout

JavaScript: DOM, elements, Ajax, Fetch API

This week

- Back-end development & frameworks
- Django
 - Setup, simple views, forms, templates
- MVC Design patterns

Back-end development

- What user **can't** see
What does it even mean?
- All logic and processes that happen **behind the scene**
- At the **server-side!**
- Processing the requests, creating responses, data management

Web server

- Listens on specified port(s)
- Handles incoming connections
 - Generates a response
 - Fetches a file
 - Forwards them to corresponding applications
- Load balancing, security, file serving, etc
- Examples: Apache, Nginx

Backend frameworks

- Doing everything from scratch?
Listen on a port, process http requests (path, method, headers, body), retrieve data from storage, process data, create the response
- Not really a good idea!
- A lot of **frameworks** are out there!
A lot of things are pre-implemented

Backend frameworks

- **PHP**: Laravel, CodeIgniter
- **Python**: Django, Flask, FastAPI
- **Javascript**: ExpressJS, Spring
- **Ruby**: Ruby on Rails

Concept is more important than
framework!

Django: a backend framework with Python



DJANGO APPS



Python projects

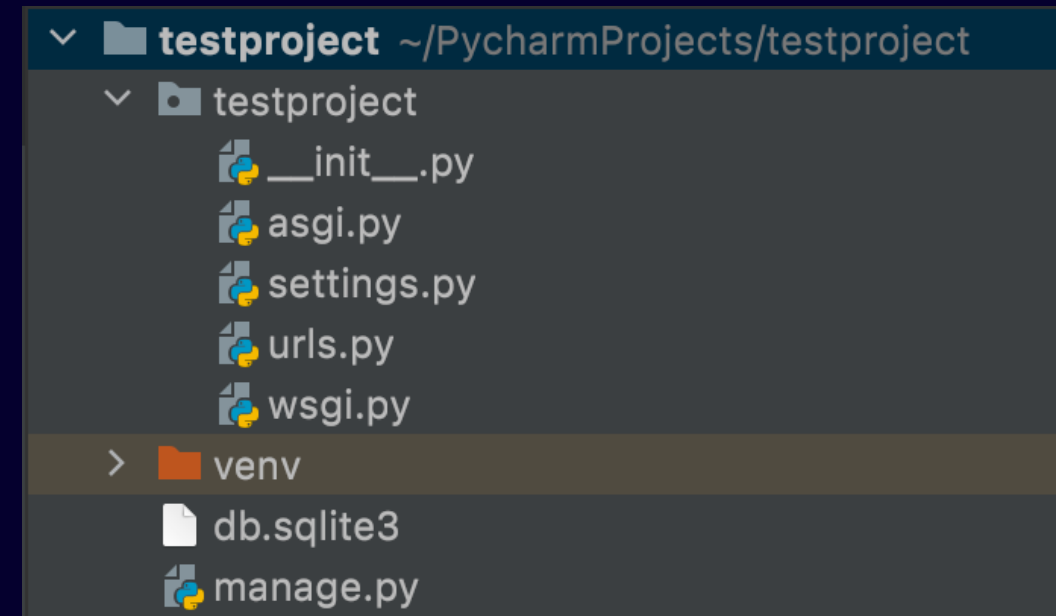
- A big project needs several different **packages!**
- Python's package manager: **pip**
- Command: **python3 -m pip install Django**

Creating a Django project

- Create the folder, environment, and install Django
- Command: `django-admin startproject <name> .`
- Creates the **skeleton** for your work
- <https://docs.djangoproject.com/en/3.2/intro/tutorial01>

Project structure

- Run the project
`python3 manage.py runserver`
- Access the **website** from
`http://localhost:8000`
- Django has a small **development** server



Taking requests

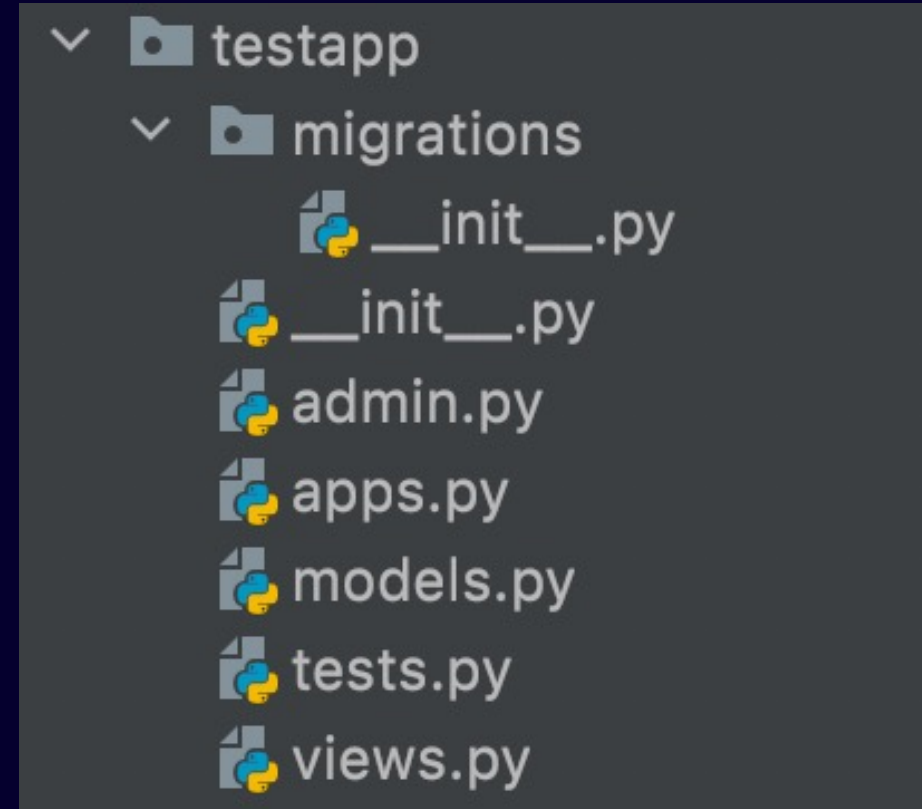
- View: a **piece of code** that runs upon a **request** to a specific endpoint (URL)
- Can be a **function** or a **class**
- How to create a new view?
First, you need to create an **app**

Django apps

- Django is intended for **big** projects
Where **tens** or hundreds of views could exist
- Project's logic is organized by **apps**
- Each app takes care of a set of **related** views, urls, or models
- Example: one app for accounts, one for transactions, one for products, etc.
- Create a new app: **python3 manage.py startapp <name>**
Most times you can do **./manage.py** instead of **python3 manage.py**

App structure

- `models.py`, `migrations`, `admin.py`: next session
- **ALWAYS** add the app name to the end of `INSTALLED_APPS` in project's `settings.py`



Create a new view

- Just write a function in `views.py` that takes an argument: `request`
- Return an `HttpResponse` instance

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello")
```

Map a URL to the view

- Add a **path** to **urlpatterns**
`path('your/path/', hello)`
- Defining all urls in a single file is a **terrible** idea
Makes the urls so messy and disorganized
- Solution: **hierarchical** urls based on apps

Hierarchical URL system

- Create a `urls.py` for each app
Make a `namespace` for each app
- Main `urls.py`:
`path('accounts/', include('accounts.urls'))`
- App's `urls.py`:
`path('hello/', hello)`
- Now access the page through <http://localhost:8000/accounts/hello/>
- **ALWAYS** end your paths with a `slash (/)`

More sophisticated views

- Receive **arguments** through the URL
`path('hello/<str:name>', hello)`
- At the view function
`def hello(request, name):`
- **Extract** request data
`request.method, request.GET, request.POST, request.headers`

Exercise: Create a simple signup form

Form validation

- Email should be valid
- Password must be at least 8 characters
- Username must consist of lowercase letters and digits
- Can be checked at the front-end (a good UX)
- But it **must** always be checked at the backend as well
- User can always **bypass** front-end **restrictions**
 - Inspect element
 - Manual request

Form validation

- If data is **invalid**, an **error** can be returned
- Error 400: **HttpResponseBadRequest**
- Error 403: **HttpResponseForbidden**
- Error 404: **HttpResponseNotFound**

Form success

- On success, a **redirect** is often returned
Redirect to profile page or index page after log in
- Use **HttpResponseRedirect**

HTML Response

- Create a **templates** folder inside the app's directory
- Add an html file there: **hello.html**
- At view, return **TemplateResponse(request, 'hello.html')**
- Django standard: create a **subdirectory** with the same name as the app and put html files there
Template address would be **'<appname>/hello.html'**

Exercise: Serve the signup form from the Django server

Flow of forms

- The **form** and **submission** share the **same** endpoint
- If request's **method** is **GET**, the form itself is returned
- If it's **POST**, the submission is **validated**
Don't forget to add `{% csrf_token %}` to the form
- If form is valid, a **redirect** is returned
- Otherwise, the form with **errors** will be returned!
Not just a simple 400 error, which is a bad UX

Form errors

- Django templates are so **dynamic**!
Data can be passed from the view to the template
- The **context** argument of **TemplateResponse**
`context={'error': 'form is invalid'}`
- Access the variables at the template (html file)
`<h3> {{ error }} </h3>`

Django template language

- Data passed through **context**
`context={'errors': ['err1', 'err2']}`
- **For** loop
`{% for error in errors %}
 <h3> {{ error }} </h3>
{% endfor %}`
- Attribute, method, dictionary lookup, or index access
`{{ errors.0 }} {{ request.GET }} {{ mydict.items }}`

Django template language

- If statement

```
{% if errors %}  
    <h3> There has been errors </h3>  
{% endif %}
```

- Filters

```
{{ errors|length }} {{ name|default="TBD" }}
```

- More tags (technically functions)

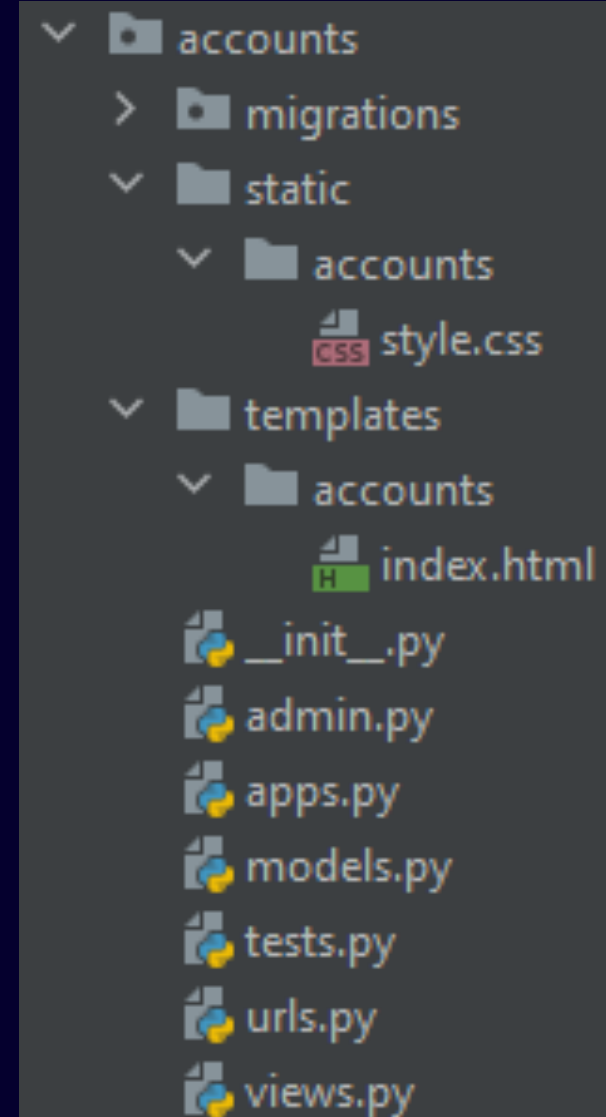
```
{% csrf_token %}
```

Static files

Visit <https://docs.djangoproject.com/en/3.2/howto/static-files/>

- The html response is **dynamic**
Created at each **request**, after template language is **compiled**
- Other files are **static**
CSS, JavaScript, images

- Put static files under **static** directory of each **app**
Standard is to put them under a **subdirectory** with the app's name
- separate css, js, img
- Access them at **template**
... `href="{% static 'accounts/style.css' %}"`
Add `{% load static %}` to the top
- Translated to
`/static/accounts/style.css`

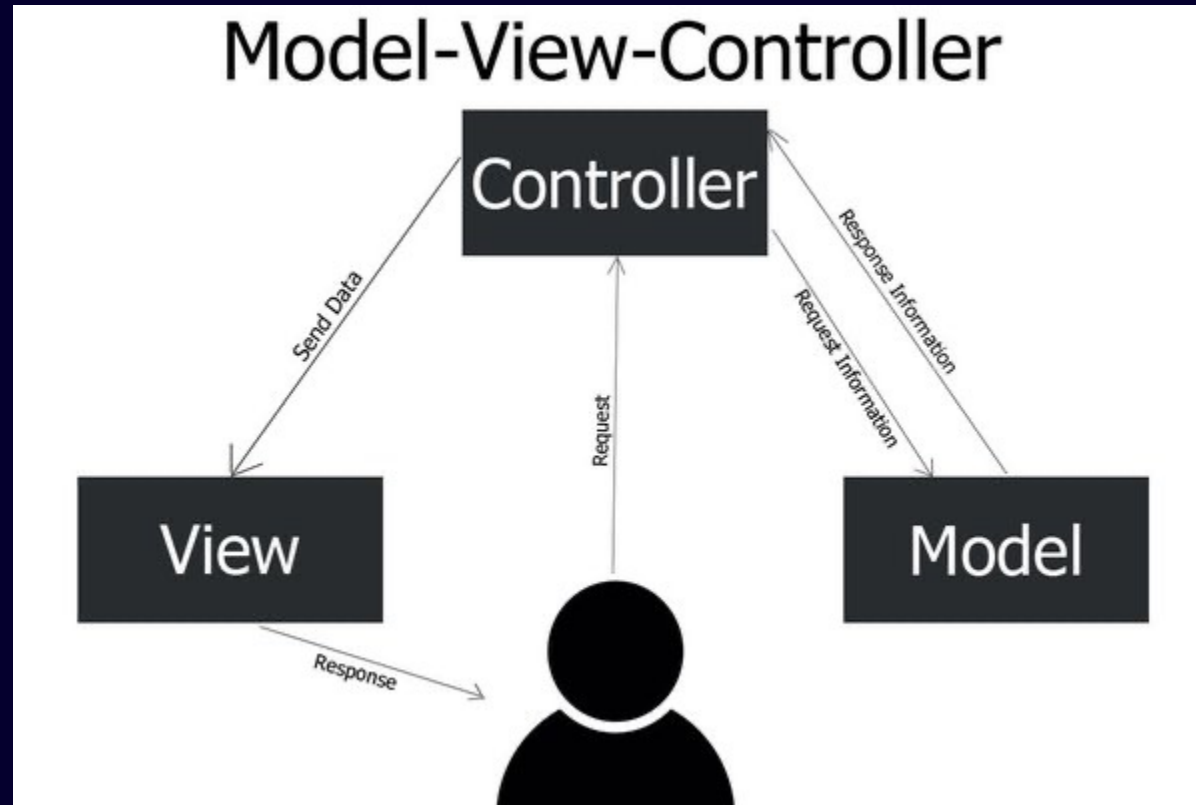


Static files

- Some static files don't really **belong** to an app
Global style, **bootstrap**, font, etc.
- Custom **directories** for static files
Usually a root **static** folder in the project
- At **settings.py**

```
STATICFILES_DIRS = [  
    BASE_DIR / "static"  
]
```

MVC



Source: <https://www.quora.com/Is-Django-an-MVC-or-an-MVT-framework>

Django's architecture



Source: <https://data-flair.training/blogs/django-architecture/>

Django's architecture

- **Almost** an MVC framework
- Naming differences
 - Django's **view**: MVC's **controller**
 - Django's **template**: MVC's **view**
- Parts of **controller** already implemented by framework
 - URL **dispatcher**
- Django **templates** are more than just presentation
 - Capable of adding some **logic**

This week

- Back-end development & frameworks
- Django
 - Setup, simple views, forms, templates
- MVC Design patterns

Next week

- Working with a **database**
ORM and models
- **Authentication**
User model, sessions, login
- **Class-based** views
- **Admin** panel