Installations

Install conda and create a new conda environment (this is your virtual environment)

```
conda create --name movowReact python=3.10
conda activate movowReact
```

requirements.txt has the following packages that must be pip installed

```
asgiref==3.7.2
Django==4.2.7
django-cors-headers==4.3.1
djangorestframework==3.14.0
psycopg2-binary==2.9.9
python-dotenv==1.0.0
pytz==2023.3.post1
sqlparse==0.4.4
typing_extensions==4.8.0
tzdata==2023.3
```

^ use UTF-8 encoding to save!

To install the above simply be in the same folder as the *requirements.txt* and run

```
pip install -r requirements.txt
```

Make sure to have installed npm for anything React

File Structure Setup

> Create a new Django environment

```
Start by making a new project to house your backend
`django-admin startproject movow`
Change directory into new project, then create an application to deal with your API requests
`django-admin startapp api`
To test if this works, in the same directory as *manage.py* type
`python manage.py runserver`
```

> Create a new React environment

```
Start by making new project folder in the Django project folder
`npx create-react-app frontend`
After changing directory into your frontend, test if this works by typing
`npm start`
```

Both of the methods to check if they work should start a local host with their own default port, close out of all before we continue further

Django Backend Settings Setup

1. Go into the project settings and change the INSTALLED_APPS

   ```
   "django.contrib.admin",
       "django.contrib.auth",
       "django.contrib.contenttypes",
       "django.contrib.sessions",
       "django.contrib.messages",
       "django.contrib.staticfiles",
       "api.apps.ApiConfig",
       "rest_framework",
       "corsheaders",
   ```

   ^ this connects the app we made to the project, enables the use of the rest_frameworks library for our Django project, and corsheaders so that we allow access for communication across front and backend
2. In the same project settings, change the MIDDLEWARE to include corsheaders

   ```
    "django.middleware.security.SecurityMiddleware",
       "django.contrib.sessions.middleware.SessionMiddleware",
       "django.middleware.common.CommonMiddleware",
       "django.middleware.csrf.CsrfViewMiddleware",
       "django.contrib.auth.middleware.AuthenticationMiddleware",
       "django.contrib.messages.middleware.MessageMiddleware",
       "django.middleware.clickjacking.XFrameOptionsMiddleware",
       "corsheaders.middleware.CorsMiddleware",
   ```

3. Modify TEMPLATES so that we can assign any additional directories to look at

   ```
   "DIRS": [
               BASE_DIR / "frontend/build"
           ],
   ```

4. Change the DATABASE to the following (we'll hide this later)

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": "db_name",
        "USER": "db_user",
        "PASS": "db_pass",
        "HOST": "db_host",
        "PORT": "db_port",
    }
    }
}
```

5. Change the STATICFILES_DIRS (helpful when we do a collectstatic)

```
STATICFILES_DIRS = [
    BASE_DIR / "frontend/build/static",
    ]
```

6. Whitelist the ports we'll be using for debugging/developing, so add this to the bottom of *settings.py*

```
CORS_ORIGIN_WHITELIST = (
    'http://localhost:3000',  # React default port = 3000
    'http://localhost:8000',  # Django default port = 8000
    )
```

7. (Optional) Change the time zone so it matches yours (useful when collecting date information)

```
TIME_ZONE = "America/New_York"
```

8. (Optional) Configure your ALLOWEDHOSTS to only be hosted by the domains or localhost IPs

```
ALLOWED_HOSTS = ["127.0.0.1"]
```

Check if everything works ok by running the server once again

Hiding Secrets

To prevent anyone from having access to our secret codes or passwords, we'll add an *.env* file that will be dispersed to any MOVOW developer

In the project directory with *manage.py* create a new file and name it *.env*
Inside of it enter the following credentials we want to keep secret

```
SECRET_KEY="whatever-your-secret-key-is"
DB_NAME="db_name"
DB_USER="db_user"
DB_PASS="db_pass"
DB_HOST="db_host"
DB_PORT="db_port"
```

Then make sure to include these lines in your code to substitute secret keys or passwords

```
import os
import dotenv

dotenv.load_dotenv()
------------------------------------------------
SECRET_KEY = os.getenv('SECRET_KEY')
------------------------------------------------
DATABASES = {

    "default": {
        "ENGINE": "django.db.backends.postgresql",
        "NAME": os.getenv("DB_NAME"),
        "USER": os.getenv("DB_USER"),
        "PASSWORD": os.getenv("DB_PASS"),
        "HOST": os.getenv("DB_HOST"),
        "PORT": os.getenv("DB_PORT"),
    }
}
```

API Application Configurations

Here we focus on the api app where we'll we'll configure views, models, urls, and tests

Making Changes to Models

Fill out your model appropriately, we won't be going too into that here, so we can apply those changes and make migrations

```
python manage.py makemigrations
python manage.py migrate
```

^ it should warn you that the frontend build/static directory isn't there and favicon which references to the shortcut icon on the tab, just ignore it for now and make sure everything works

If the models do not have a string representation made of the class model characteristics, add one for printing purposes when we view them admin view like so

```
class Movies(models.Model):
    movie_id = models.IntegerField(primary_key=True)
    movie_title = models.CharField(max_length=255, null=False)
    release_date = models.DateField()

    def __str__(self):
        return f"{self.movie_title} {self.movie_id} {self.release_date}"
```

Creating Superuser and Registering Models for Admin

1. Create the superuser

   ```
   python manage.py createsuperuser
   ```

   ^ follow the prompts after to create your admin user
2. Register models for admin view in *admin.py*

   ```
   from django.contrib import admin
   from .models import *

   admin.site.register(Movies)
   ```

3. To view registered models in admin view, run the server and append '/admin' to your base URL which will prompt you to sign in

Creating Serializers for APIs

To communicate between frontend and backend we need to be able to serialize model information in a way that is readable for our frontend tool

In the api directory create a python file named 'serializers.py' and create a JSON serializer for every model class

```
from rest_framework.serializers import ModelSerializer
from .models import *

class MovieSerializer(ModelSerializer):
        class Meta:
                model = Movies
                fields = '__all__'
```

^ this selects a model to use and specifies which fields to be part of the serializer, for more customization here

Modifying Views to be RESTful

Views would generally render HTML that users would see or redirect to an HTML file located in a static folder and return a status code, but since we're only dealing with moving JSON information around we will just return a serialized response

```
from .models import *
from .serialzers import *

from rest_framework.response import Response
from rest_framework.decorators import api_view

@api_view(['GET'])

def getRoutes(request):

    routes = [
        {
            'Endpoint': '/movies/',
            'method': 'GET',
            'body': None,
            'description': 'Return a list of all the movies'
        },
    ]

    return Response(routes)
```

^ Response here does not render the data, it needs to be serialized before use, with the default status code being 200

api_view is a decorator used to work with function-based views that allows for explicit CRUD methods

We will work on these methods further when we verify we can access our API calls

Creating URL Routing for Backend

The application, by default, comes with no script for URLs so we must create a *urls.py* in the application folder to route specific URL patterns to views

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

urlpatterns = [
        path('', views.getRoutes, name="routes"),
]
```

In this block of code, the urlpatterns are what we use to match the rightmost parts after the base URL to then route according views and actions via paths

We will add more routing as we require more functions

Connect Project's URLs to App URLs

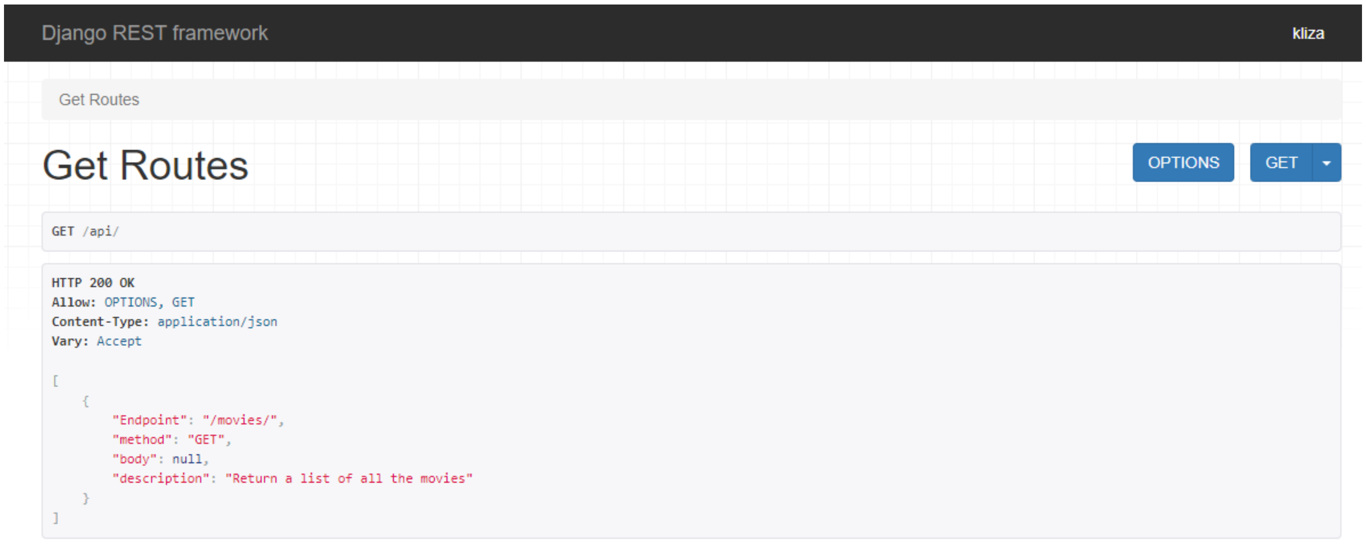
When you run the Django development server by default the screen is the rocket ship, this is because we haven't connected the application's URL routing to the overall project

To connect our URL routing we made, add this to the project's *urls.py*

```
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/", include("api.urls")),
]
```

Now running the server in DEBUG mode at the base URL will show you the available routing for the project, two of them being `admin/` and `api/`

If you were to search for `http://127.0.0.1:8000/api/` you would then get this masterpiece



```
GET /api/
```

```
HTTP 200 OK
Allow: OPTIONS, GET
Content-Type: application/json
Vary: Accept

[
    {
        "Endpoint": "/movies/",
        "method": "GET",
        "body": null,
        "description": "Return a list of all the movies"
    }
]
```

Frontend React Context

Now that we have made our base API configurations in our backend, now we want to make viewable components using React

React Folder Structure

This is the general folder structure for an advanced project

- assets - where you store any static files (css, svg, imgs)
- components - all components we use when building our application and small (modal, button, etc.)
  - ui - primitive components to distinct from general complex components that are shared
- pages - root level pages where users can go to will live
  - page_name - has components unique to the page (if no feature folder present)

- layouts - for components that are laid out like navigation or page construction
- lib - implementation of facade pattern when a library needs to be updated
- hooks - housing custom hooks
- services - any external service like APIs that the application consumes
- features - mini src folder in which they make pages simpler by isolating the features in a place where anyone can use it (self-encapsulated)
  - Can have components, hooks, services, etc.. of their own
- utils - any useful files that don't belong anywhere else

We'll only implement assets, components, pages, and utils for now

SOLID Principle

Developing code in React should follow this principle (similarly to OOP) to make more readable, maintainable, and testable code

Refer to this article

Single Responsibility
Each component should have one responsibility
Break down complicated actions into smaller components or hooks

Open/Closed
Components should be open for extensions and closed for modifications
Ideal approach is to add to the functions that already exist NOT change them

Liskov Substitution
Composition and props
If you make a component that extends from another component, should be able to use in same place
Parent child relationship should have the child take on parent responsibilities too (inheritance)

Interface Segregation
No client should be forced to depend on methods it does not use
Any other action should be removed completely or moved somewhere else

Dependency Inversion Principle
High level module does not depend on low level modules
Both depend on abstractions, details depend on abstractions but not other way around

Additional Common Mumbo Jumbo

Hooks
Functions that let you “hook into” React state and lifecycle features from function components
Always use the "use" prefix when naming hooks, as it helps to quickly identify them
Don't call hooks from regular JavaScript functions, only from React function components

React Configurations

Now that we know a little bit about React let's start coding our first 'view'
We'll initially do this in Javascript, but should transition to Typescript in the future

Install the following before moving on

```
npm install react-router-dom --save
```

Navigate to `/frontend/src` and make sure to create a `pages/` directory where you will create the *HomePage.js* containing the following

```
const HomePage = () => {
    return (
        <div className='home'>
            <h1>MOVOW Home</h1>
        </div>
    )
}

export default HomePage
```

^ this returns HTML that is currently just a header

Now navigate to *App.js* and replace the default code with the following to represent screen routing

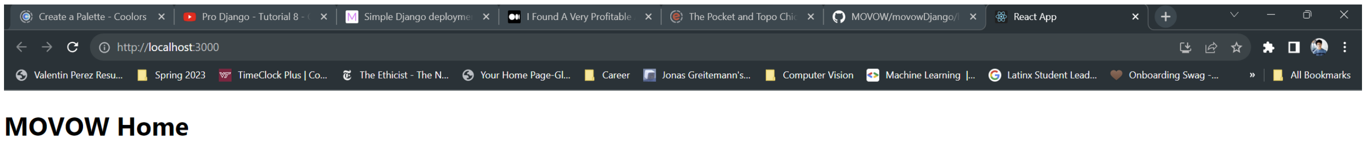
```
import {
  HashRouter as Router,
  Route,
  Routes
} from 'react-router-dom';
import HomePage from './pages/HomePage'
import './App.css';

function App() {
  return (
    <Router>
      <Routes>
        <Route path='/' exact Component={HomePage}/>
      </Routes>
    </Router>
  );
}

export default App;
```

^ this creates the initial pathing to make sure we can create and route pages via react-router-dom

To test if all this works, make sure to be in the `frontend/` directory and run

```
npm start
```

You should see this



MOVOW Home

Gluing Backend and Frontend

By now you should have a static home page, but it does us no good if we can't show any information from our database in a dynamic way

Navigate to the *urls.py* in the `movow/` project folder and match this code

```
from django.contrib import admin
from django.urls import path, include
from django.views.generic import TemplateView

urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/", include("api.urls")),
    path("", TemplateView.as_view(template_name='index.html')),
]
```

^ the last urlpattern makes it so that the default page when running the server connects to the React build static folder which has its own routing

Now modify the *HomePage.js* file in your `frontend/pages/` folder to display the movies data in your database

```
import React, { useState, useEffect } from 'react';

const HomePage = () => {
    const [movies, setMovies] = useState([]);

    useEffect(() => {
        getMovies();
    }, []);

    const getMovies = async () => {
        try {
            const response = await fetch('/api/movies/');
            console.log(response)
            const data = await response.json();
            setMovies(data);
        } catch (error) {
            console.error('Error fetching movies:', error);
        }
    };

    return (
        <div className='home'>
            <h1>MOVOW Home</h1>
            <p>Movies In Database: {movies.length}</p>

            <div className='movies-list'>
                {movies.map((movie, index) => (
                    <div key={index} className='movie-item'>
                        <p>Title: {movie.movie_title}, Release Date: {movie.release_date}</p>
                    </div>
                ))}
            </div>
        </div>
    );
};

export default HomePage;
```

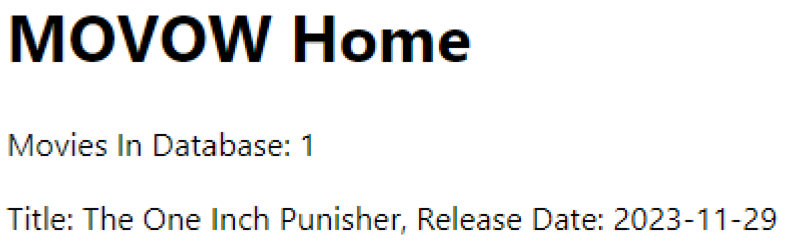
^ this block of code uses the useState to retrieve database data and useEffect makes that happen with the getMovies arrow function, which is then mapped in the return HTML

You might be asking how do these two connect because you would have to run the Django server on its own port and the React server on another which would make the async fetch call not work

To combat this (run both on one server) and apply the changes we made to *urls.py* so that it correctly looks at the `frontend/build/index.html`, we must build the project and then run the development server starting in the `frontend/` directory

```
npm run build
cd ..
python manage.py runserver
```

Doing this should fix the fetch call so that it accesses the right api call when running both programs and should show this page by default (right now with one sample entry)



MOVOW Home

Movies In Database: 1

Title: The One Inch Punisher, Release Date: 2023-11-29

Creating the Gitignore

We want to avoid pushing our *.env* or cache files to the database, so we'll create some rules that prevent them from being pushed onto the remote repo

In the same directory as *manage.py* create a file named *.gitignore*

```
.idea/
venv/
movow/db.sqlite3
movow/api/__pycache__/
movow/movow/__pycache__/
movow/api/migrations/__pycache__/
movow/.env
```

React should already have its own gitignore