Let's go through the process of building Docker images for a Django project from scratch. This involves creating Dockerfiles and understanding how the build process works. Additionally, we'll touch on the role of entrypoint and start scripts in the context of Docker containers.

### Step 1: Project Structure

First, ensure your Django project has a structure that can be containerized. A common structure might look like this:

```

myproject/

|-- src/

| |-- manage.py

| |-- myapp/

| |-- ...

|-- docker/

| |-- local/

| |-- django/

| |-- Dockerfile

| |-- postgres/

| |-- Dockerfile

|-- .envs/

|-- .local/

|-- .django

|-- .postgres

|-- local.yml

|-- entrypoint.sh

|-- start.sh

```

### Step 2: Dockerfiles

Create Dockerfiles for Django and PostgreSQL. These files define how Docker should build the images for your services.

\*\*Dockerfile for Django (`docker/local/django/Dockerfile`):\*\*

```dockerfile

# Use the official Python image as a base

FROM python:3.9

# Set environment variables

ENV PYTHONDONTWRITEBYTECODE 1

ENV PYTHONUNBUFFERED 1

# Set the working directory inside the container

WORKDIR /app

# Copy the requirements file to the container

COPY ./src/requirements.txt /app/requirements.txt

# Install dependencies

RUN pip install --upgrade pip

RUN pip install -r requirements.txt

# Copy the Django project to the container

COPY ./src /app

# Set the entrypoint script

ENTRYPOINT ["./entrypoint.sh"]

```

\*\*Dockerfile for PostgreSQL (`docker/local/postgres/Dockerfile`):\*\*

```dockerfile

# Use the official PostgreSQL image as a base

FROM postgres:latest

# Copy the initialization script to the container

COPY ./docker/local/postgres/init.sql /docker-entrypoint-initdb.d/

```

### Step 3: Environment Files

Create environment files for Django and PostgreSQL (`./.envs/.local/.django` and `./.envs/.local/.postgres`).

\*\*.django Environment File (`./.envs/.local/.django`):\*\*

```dotenv

POSTGRES\_DB=mydatabase

POSTGRES\_USER=myuser

POSTGRES\_PASSWORD=mypassword

POSTGRES\_HOST=postgres

POSTGRES\_PORT=5432

```

\*\*.postgres Environment File (`./.envs/.local/.postgres`):\*\*

```dotenv

POSTGRES\_DB=mydatabase

POSTGRES\_USER=myuser

POSTGRES\_PASSWORD=mypassword

POSTGRES\_HOST=postgres

POSTGRES\_PORT=5432

```

### Step 4: Docker Compose File

Create a Docker Compose file (`local.yml`) that defines your services, networks, and volumes.

```yaml

version: '3.9'

services:

api:

build:

context: .

dockerfile: ./docker/local/django/Dockerfile

volumes:

- .:/app:z

- static\_volume:/app/staticfiles

- media\_volume:/app/mediafiles

ports:

- "8000:8000"

env\_file:

- ./.envs/.local/.django

- ./.envs/.local/.postgres

depends\_on:

- postgres

command: /start

networks:

- mynetwork

postgres:

build:

context: .

dockerfile: ./docker/local/postgres/Dockerfile

volumes:

- local\_postgres\_data:/var/lib/postgresql/data

- local\_postgres\_data\_backups:/backups

env\_file:

- ./.envs/.local/.postgres

networks:

- mynetwork

networks:

mynetwork:

driver: bridge

volumes:

static\_volume:

media\_volume:

local\_postgres\_data: {}

local\_postgres\_data\_backups: {}

```

### Step 5: Entrypoint and Start Scripts

Create entrypoint and start scripts (`entrypoint.sh` and `start.sh`).

\*\*entrypoint.sh:\*\*

```bash

#!/bin/bash

set -o errexit

set -o pipefail

set -o nounset

if [ -z "${POSTGRES\_USER}" ]; then

base\_postgres\_image\_default\_user='postgres'

export POSTGRES\_USER="${base\_postgres\_image\_default\_user}"

fi

export DATABASE\_URL="postgres://${POSTGRES\_USER}:${POSTGRES\_PASSWORD}@${POSTGRES\_HOST}:${POSTGRES\_PORT}/${POSTGRES\_DB}"

python << END

import sys

import time

import psycopg2

suggest\_unrecoverable\_after = 30

start = time.time()

while True:

try:

psycopg2.connect(

dbname="${POSTGRES\_DB}",

user="${POSTGRES\_USER}",

password="${POSTGRES\_PASSWORD}",

host="${POSTGRES\_HOST}",

port="${POSTGRES\_PORT}",

)

break

except psycopg2.OperationalError as error:

sys.stderr.write("Waiting for PostgreSQL to become available...\n")

if time.time() - start > suggest\_unrecoverable\_after:

sys.stderr.write(" This is taking longer than expected. The following exception may be indicative of an unrecoverable error: '{}'\n".format(error))

time.sleep(1)

END

>&2 echo "PostgreSQL is available"

exec "$@"

```

\*\*start.sh:\*\*

```bash

#!/bin/bash

set -o errexit

set -o pipefail

set -o nounset

python manage.py migrate --no-input

python manage.py collectstatic --no-input

exec python manage.py runserver 0.0.0.0:8000

```

### Step 6: Build and Run

Now, you can build and run your services using Docker Compose:

```bash

docker-compose -f local.yml up --build

```

This command tells Docker Compose to build the images (`--build`) and start the services defined in `local.yml`. The `entrypoint.sh` and `start.sh` scripts will be executed as part of the container startup process.

That's it! You've created Dockerfiles, environment files, and scripts to containerize a Django project. Adjustments may be needed based on your specific project structure or requirements, but this should provide a good starting point.

entrypoint:

#!/bin/bash

set -o errexit

set -o pipefail

set -o nounset

if [ -z "${POSTGRES\_USER}" ]; then

base\_postgres\_image\_default\_user='postgres'

export POSTGRES\_USER="${base\_postgres\_image\_default\_user}"

fi

export DATABASE\_URL="postgres://${POSTGRES\_USER}:${POSTGRES\_PASSWORD}@${POSTGRES\_HOST}:${POSTGRES\_PORT}/${POSTGRES\_DB}"

python << END

import sys

import time

import psycopg2

suggest\_unrecoverable\_after = 30

start = time.time()

while True:

try:

psycopg2.connect(

dbname="${POSTGRES\_DB}",

user="${POSTGRES\_USER}",

password="${POSTGRES\_PASSWORD}",

host="${POSTGRES\_HOST}",

port="${POSTGRES\_PORT}",

)

break

except psycopg2.OperationalError as error:

sys.stderr.write("Waiting for PostgreSQL to become available...\n")

if time.time() - start > suggest\_unrecoverable\_after:

sys.stderr.write(" This is taking longer than expected. The following exception may be indicative of an unrecoverable error: '{}'\n".format(error))

time.sleep(1)

END

>&2 echo "PostgreSQL is available"

exec "$@"

**Explanation:**

1. **Shebang (#!/bin/bash):** Indicates that the script should be interpreted using Bash.
2. **set commands:**
   * **set -o errexit**: Causes the script to exit if any command it runs exits with a non-zero status.
   * **set -o pipefail**: Causes a pipeline to produce a failure return code if any command in the pipeline fails.
   * **set -o nounset**: Causes the script to exit if it tries to use an uninitialized variable.
3. **Default PostgreSQL User:**
   * Checks if **POSTGRES\_USER** is empty and sets it to the default PostgreSQL user ('postgres') if it is.
4. **export DATABASE\_URL:**
   * Constructs the **DATABASE\_URL** environment variable using the provided PostgreSQL connection details.
5. **Python Block (python << END ... END):**
   * Uses a Python block to check if PostgreSQL is available. It attempts to connect to PostgreSQL in a loop, waiting until it succeeds or a specified timeout is reached.
6. **>&2 echo "PostgreSQL is available":**
   * Writes a message to standard error indicating that PostgreSQL is available.
7. **exec "$@":**
   * Executes the command passed to the entrypoint script. This allows flexibility in what command is run when the container starts.

start:

#!/bin/bash

set -o errexit

set -o pipefail

set -o nounset

python manage.py migrate --no-input

python manage.py collectstatic --no-input

exec python manage.py runserver 0.0.0.0:8000

**Explanation:**

1. **Shebang (#!/bin/bash):** Indicates that the script should be interpreted using Bash.
2. **set commands:**
   * **set -o errexit**: Causes the script to exit if any command it runs exits with a non-zero status.
   * **set -o pipefail**: Causes a pipeline to produce a failure return code if any command in the pipeline fails.
   * **set -o nounset**: Causes the script to exit if it tries to use an uninitialized variable.
3. **Database Migrations and Collecting Static Files:**
   * **python manage.py migrate --no-input**: Applies database migrations without prompting for input.
   * **python manage.py collectstatic --no-input**: Collects static files without prompting for input.
4. **exec python manage.py runserver 0.0.0.0:8000:**
   * Executes the Django development server, allowing external connections (**0.0.0.0:8000**).

These scripts are commonly used in the context of Docker containers to ensure that the necessary setup steps are performed when the container starts. The **entrypoint** script, in particular, is often used to handle any initialization tasks before the main application command is executed. The **start** script, in this case, is focused on starting the Django development server after performing necessary setup tasks.

local.yml:

version: "3.9"

services:

api:

build:

context: .

dockerfile: ./docker/local/django/Dockerfile

volumes:

- .:/app:z

- static\_volume:/app/staticfiles

- media\_volume:/app/mediafiles

ports:

- "8000:8000"

env\_file:

- ./.envs/.local/.django

- ./.envs/.local/.postgres

depends\_on:

- postgres

- mailhog

command: /start

networks:

- authors-api

postgres:

build:

context: .

dockerfile: ./docker/local/postgres/Dockerfile

volumes:

- local\_postgres\_data:/var/lib/postgresql/data

- local\_postgres\_data\_backups:/backups

env\_file:

- ./.envs/.local/.postgres

networks:

- authors-api

mailhog:

image: mailhog/mailhog:v1.0.0

container\_name: mailhog

ports:

- "8025:8025"

networks:

- authors-api

networks:

authors-api:

driver: bridge

volumes:

static\_volume:

media\_volume:

local\_postgres\_data: {}

local\_postgres\_data\_backups: {}

Explanation:

1. \*\*`version: "3.9"`\*\*: Specifies the version of the Docker Compose file format you're using. This version is associated with the features and syntax available in the file.

2. \*\*`services:`\*\*: Defines the services that make up your application.

- \*\*`api:`\*\*: Configuration for the Django API service.

- \*\*`build:`\*\*: Configuration for building the Docker image.

- \*\*`context: .`\*\*: The build context, the root directory for building the image. In this case, it's the current directory (`.`).

- \*\*`dockerfile: ./docker/local/django/Dockerfile`\*\*: Path to the Dockerfile for building the image.

- \*\*`volumes:`\*\*: Defines volumes to be mounted for the service.

- \*\*`- .:/app:z`\*\*: Mounts the current directory (`.`) to `/app` inside the container, with SELinux context `z`.

- \*\*`- static\_volume:/app/staticfiles`\*\*: Uses the named volume `static\_volume` for static files.

- \*\*`- media\_volume:/app/mediafiles`\*\*: Uses the named volume `media\_volume` for media files.

- \*\*`ports:`\*\*: Maps ports between the host and the container.

- \*\*`- "8000:8000"`\*\*: Maps port 8000 on the host to port 8000 on the container.

- \*\*`env\_file:`\*\*: Specifies environment files to load.

- \*\*`- ./.envs/.local/.django`\*\*: Path to the Django environment file.

- \*\*`- ./.envs/.local/.postgres`\*\*: Path to the PostgreSQL environment file.

- \*\*`depends\_on:`\*\*: Specifies services that this service depends on.

- \*\*`- postgres`\*\*: Depends on the PostgreSQL service.

- \*\*`- mailhog`\*\*: Depends on the MailHog service.

- \*\*`command: /start`\*\*: Overrides the default command to start the container.

- \*\*`networks:`\*\*: Specifies the networks the service should be connected to.

- \*\*`- authors-api`\*\*: Connects to the `authors-api` network.

- \*\*`postgres:`\*\*: Configuration for the PostgreSQL service.

- \*\*`build:`\*\*: Similar to the `api` service, specifies the build context and Dockerfile.

- \*\*`volumes:`\*\*: Defines volumes for PostgreSQL data and backups.

- \*\*`env\_file:`\*\*: Specifies the PostgreSQL environment file.

- \*\*`networks:`\*\*: Connects to the `authors-api` network.

- \*\*`mailhog:`\*\*: Configuration for the MailHog service.

- \*\*`image: mailhog/mailhog:v1.0.0`\*\*: Uses a pre-built MailHog image.

- \*\*`container\_name: mailhog`\*\*: Sets the container name to "mailhog".

- \*\*`ports:`\*\*: Maps port 8025 on the host to port 8025 on the container.

- \*\*`networks:`\*\*: Connects to the `authors-api` network.

3. \*\*`networks:`\*\*: Defines the networks used by the services.

- \*\*`authors-api:`\*\*: Configuration for the `authors-api` network.

- \*\*`driver: bridge`\*\*: Specifies the network driver as "bridge."

4. \*\*`volumes:`\*\*: Defines named volumes used by the services.

- \*\*`static\_volume:`\*\*, \*\*`media\_volume:`\*\*, \*\*`local\_postgres\_data:`\*\*, \*\*`local\_postgres\_data\_backups:`\*\*: Named volumes.

\*\*Command Explanation:\*\*

The `docker compose -f local.yml config` command is used to display the effective configuration that Docker Compose would use based on the specified `local.yml` file. It validates the file, substitutes variables if present, and prints the final configuration. This can be useful for checking if the configuration is correctly interpreted by Docker Compose before actually running the services.

BEFORE attempting to create container, need to run in terminal:

export DOCKER\_REFAULT\_PLATFORM=linux64/amd64

(only necessary for M1 and newer macs)

To build container, run:

docker compose -f local.yml up --build -d --remove-orphans

To bring down the containers:

docker-compose -f local.yml down

To check logs:  
docker-compose -f local.yml logs

To check logs of specific service:  
docker-compose -f local.yml logs service\_name

To inspect volumes:  
docker volume inspect src\_volume\_name\_here

CONSTANTS AND MESSAGES SHELL SCRIPTS

In production, you want to use third party service rather than self-hosting your own database

We are creating a maintenance folder in the docker>local>postgres folder create maintenance folder.

First, modify the Dockerfile inside the postgres folder:

FROM postgres:15-bullseye

COPY ./docker/local/postgres/maintenance /usr/local/bin/maintenance

RUN chmod +x /usr/local/bin/maintenance/\*

RUN mv /usr/local/bin/maintenance/\* /usr/local/bin \

&& rmdir /usr/bin/maintenance

Certainly! Let's break down each line of the Dockerfile for the PostgreSQL service in detail:

```dockerfile

# Use the official PostgreSQL 15 image based on Debian Bullseye

FROM postgres:15-bullseye

```

- `FROM postgres:15-bullseye`: This line specifies the base image for the Dockerfile. It's using the official PostgreSQL 15 image based on Debian Bullseye. This image provides a pre-configured PostgreSQL server environment.

```dockerfile

# Copy the contents of the local directory './docker/local/postgres/maintenance' to '/usr/local/bin/maintenance' in the image

COPY ./docker/local/postgres/maintenance /usr/local/bin/maintenance

```

- `COPY ./docker/local/postgres/maintenance /usr/local/bin/maintenance`: This line copies the contents of the local directory `./docker/local/postgres/maintenance` into the `/usr/local/bin/maintenance` directory in the image. This directory likely contains maintenance scripts or utilities needed for PostgreSQL.

```dockerfile

# Make all files in '/usr/local/bin/maintenance' executable

RUN chmod +x /usr/local/bin/maintenance/\*

```

- `RUN chmod +x /usr/local/bin/maintenance/\*`: This line sets the execute (`+x`) permission on all files in the `/usr/local/bin/maintenance` directory. This step ensures that the scripts or binaries in the maintenance directory can be executed.

```dockerfile

# Move all files from '/usr/local/bin/maintenance' to '/usr/local/bin' and remove the empty directory '/usr/bin/maintenance'

RUN mv /usr/local/bin/maintenance/\* /usr/local/bin \

&& rmdir /usr/bin/maintenance

```

- `RUN mv /usr/local/bin/maintenance/\* /usr/local/bin \`: This line moves all files from `/usr/local/bin/maintenance` to `/usr/local/bin`. After this line, the maintenance scripts are directly in `/usr/local/bin`.

- `&& rmdir /usr/bin/maintenance`: This line removes the empty directory `/usr/bin/maintenance`. The `&&` operator allows running multiple commands in a single `RUN` instruction.

In summary, this Dockerfile sets up a PostgreSQL image based on version 15 with additional maintenance scripts. It copies these scripts into the image, makes them executable, and organizes them in the `/usr/local/bin` directory. The last few commands are cleaning up the structure by moving the scripts directly into `/usr/local/bin` and removing any empty directories.

Create new folder INSIDE maintenance folder called \_sourced (this folder will hold shell scripts). Inside this folder, create a new files:

constants.sh:

#! /usr/bin/env bash

BACKUP\_DIR\_PATH='/backups'

BACKUP\_FILE\_PREFIX='backup'

messages.sh: bash shell functions that generate messages

#! /usr/bin/env bash

message\_newline(){

echo

}

message\_debug(){

echo -e "DEBUG: ${@}"

}

message\_welcome(){

echo -e "\e[1m${@}\e[0m"

}

message\_warning(){

echo -e "\e[33mWARNING\e[0m: ${@}"

}

message\_error(){

echo -e "\e[31mERROR\e[0m: ${@}"

}

message\_info(){

echo -e "\e[37mINFO\e[0m: ${@}"

}

message\_suggestion(){

echo -e "\e[33mSUGGESTION\e[0m: ${@}"

}

message\_success(){

echo -e "\e[32mSUCCESS\e[0m: ${@}"

}

Explanation:

This script defines several functions for displaying messages with different styles and colors in the terminal. Let's break down each part of the script:

```bash

#! /usr/bin/env bash

```

- `#! /usr/bin/env bash`: This line is called a shebang, and it indicates the path to the interpreter that should be used to execute the script. In this case, it specifies that the Bash shell (`bash`) should be used.

The script defines several functions for displaying messages:

```bash

message\_newline(){

echo

}

```

- `message\_newline() { echo; }`: This function, named `message\_newline`, simply echoes a newline character. It's used to print an empty line, creating a separation in the terminal output.

```bash

message\_debug(){

echo -e "DEBUG: ${@}"

}

```

- `message\_debug() { echo -e "DEBUG: ${@}"; }`: This function, named `message\_debug`, prints a message in the terminal with a "DEBUG" label. The `${@}` represents all the function's arguments. The `-e` flag allows interpreting escape sequences, enabling the use of color codes.

```bash

message\_welcome(){

echo -e "\e[1m${@}\e[0m"

}

```

- `message\_welcome() { echo -e "\e[1m${@}\e[0m"; }`: This function, named `message\_welcome`, prints a welcome message in bold. The `\e[1m` sets the text to bold, and `\e[0m` resets the text formatting.

```bash

message\_warning(){

echo -e "\e[33mWARNING\e[0m: ${@}"

}

```

- `message\_warning() { echo -e "\e[33mWARNING\e[0m: ${@}"; }`: This function, named `message\_warning`, prints a warning message in yellow. The color code `\e[33m` sets the text to yellow, and `\e[0m` resets the text formatting.

```bash

message\_error(){

echo -e "\e[31mERROR\e[0m: ${@}"

}

```

- `message\_error() { echo -e "\e[31mERROR\e[0m: ${@}"; }`: This function, named `message\_error`, prints an error message in red. The color code `\e[31m` sets the text to red, and `\e[0m` resets the text formatting.

```bash

message\_info(){

echo -e "\e[37mINFO\e[0m: ${@}"

}

```

- `message\_info() { echo -e "\e[37mINFO\e[0m: ${@}"; }`: This function, named `message\_info`, prints an informational message in white. The color code `\e[37m` sets the text to white, and `\e[0m` resets the text formatting.

```bash

message\_suggestion(){

echo -e "\e[33mSUGGESTION\e[0m: ${@}"

}

```

- `message\_suggestion() { echo -e "\e[33mSUGGESTION\e[0m: ${@}"; }`: This function, named `message\_suggestion`, prints a suggestion message in yellow. The color code `\e[33m` sets the text to yellow, and `\e[0m` resets the text formatting.

```bash

message\_success(){

echo -e "\e[32mSUCCESS\e[0m: ${@}"

}

```

- `message\_success() { echo -e "\e[32mSUCCESS\e[0m: ${@}"; }`: This function, named `message\_success`, prints a success message in green. The color code `\e[32m` sets the text to green, and `\e[0m` resets the text formatting.

In summary, this script provides functions for displaying messages in different styles and colors in the terminal. You can use these functions in other shell scripts to improve the readability of your output and provide visual cues for different types of messages.

Create yes\_no.sh file:

#! /usr/bin/env bash

yes\_no(){

declare desc="Prompt for confirmation. \$\"\{1\}\": confirmation message"

local arg1="${1}"

local response= read -r -p "${arg1} (y/[n])? " response

if [[ "${response}" =~ ^[Yy]$ ]]

then

exit 0

else

exit 1

fi

}

Explanation:

This script defines a function called `yes\_no` in Bash. The purpose of this function is to prompt the user for confirmation, expecting a yes (`y` or `Y`) or no (`n` or `N`) response. Let's break down the script:

```bash

#! /usr/bin/env bash

```

- `#! /usr/bin/env bash`: This is the shebang line, indicating that the script should be executed using the Bash interpreter.

```bash

yes\_no(){

declare desc="Prompt for confirmation. \$\"\{1\}\": confirmation message"

```

- `yes\_no() { declare desc="..."; }`: This line defines the `yes\_no` function. Inside the function, there's a description string stored in the `desc` variable. This string explains the purpose of the function: prompting for confirmation with the confirmation message specified as the first argument (`$1`) when calling the function.

```bash

local arg1="${1}"

```

- `local arg1="${1}"`: This line declares a local variable `arg1` and assigns it the value of the first argument passed to the function (`$1`).

```bash

local response= read -r -p "${arg1} (y/[n])? " response

```

- `local response= read -r -p "${arg1} (y/[n])? " response`: This line prompts the user with the confirmation message specified as the first argument (`${arg1}`). The user's response is stored in the `response` variable. The `-r` option is used with the `read` command to ensure that backslashes are not treated as escape characters.

```bash

if [[ "${response}" =~ ^[Yy]$ ]]

```

- `if [[ "${response}" =~ ^[Yy]$ ]]; then`: This line checks whether the user's response (stored in the `response` variable) matches the regular expression `^[Yy]$`, meaning it's either 'Y' or 'y'. If there's a match, the script exits with a status code of 0 (success).

```bash

then

exit 0

```

- `then exit 0`: If the user's response is 'Y' or 'y', the script exits with a status code of 0, indicating success.

```bash

else

exit 1

```

- `else exit 1`: If the user's response is anything other than 'Y' or 'y', the script exits with a status code of 1, indicating failure.

```bash

fi

}

```

- `fi`: This line marks the end of the `if` statement.

The purpose of this script is to provide a simple way to prompt the user for confirmation and exit the script with an appropriate status code based on the user's response. It's a useful utility function that can be incorporated into other scripts when confirmation is needed.

Create countdown.sh:

#! /usr/bin/env bash

countdown(){

declare desc="A simple countdown."

local seconds="${1}"

local d=$(($(date +%s) + "${seconds}"))

while [ "$d" -ge `date +%s`]; do

echo -ne "$(date -u --date @$(($d - `date +%s` )) +%H:%M:%S)\r";

sleep 0.1

done

}

Explanation:

This script defines a function called `countdown` in Bash. The purpose of this function is to create a simple countdown timer that displays the remaining time in hours, minutes, and seconds. Let's break down the script:

```bash

#! /usr/bin/env bash

```

- `#! /usr/bin/env bash`: This is the shebang line, indicating that the script should be executed using the Bash interpreter.

```bash

countdown(){

declare desc="A simple countdown."

```

- `countdown() { declare desc="..."; }`: This line defines the `countdown` function. Inside the function, there's a description string stored in the `desc` variable. This string explains the purpose of the function: creating a simple countdown.

```bash

local seconds="${1}"

```

- `local seconds="${1}"`: This line declares a local variable `seconds` and assigns it the value of the first argument passed to the function (`$1`). This argument is expected to be the total number of seconds for the countdown.

```bash

local d=$(($(date +%s) + "${seconds}"))

```

- `local d=$(($(date +%s) + "${seconds}"))`: This line calculates the target time (`d`) by adding the current time in seconds (`$(date +%s)`) to the specified number of seconds (`${seconds}`). The result is the future timestamp when the countdown should end.

```bash

while [ "$d" -ge $(date +%s) ]; do

```

- `while [ "$d" -ge $(date +%s) ]; do`: This line starts a `while` loop that continues as long as the target time (`$d`) is greater than or equal to the current time in seconds (`$(date +%s)`). In other words, the loop will continue until the countdown reaches zero.

```bash

echo -ne "$(date -u --date @$(($d - $(date +%s) )) +%H:%M:%S)\r"

```

- `echo -ne "$(date -u --date @$(($d - $(date +%s) )) +%H:%M:%S)\r"`: This line prints the remaining time in the format HH:MM:SS. The `date` command is used to convert the time difference (`$d - $(date +%s)`) into the desired format. The `-ne` options for `echo` are used to ensure that the output does not include a newline, allowing the countdown to be displayed on the same line in the terminal.

```bash

done

```

Sleep 0.1 adds a 0.1 second delay

- `done`: This line marks the end of the `while` loop.

In summary, this script defines a function that takes a number of seconds as input and creates a countdown timer. The countdown is displayed in the terminal, updating the remaining time in hours, minutes, and seconds until the countdown reaches zero.

Create Backup Scripts: These are files aimed at restoring the database (if necessary)

In the maintenance folder (one level up from the previous scripts), create a file called backup:

#! /usr/bin/env bash

set -o errexit

set -o nounset

set -o pipefail

working\_dir="$(dirname ${0})"

source "${working\_dir}/\_sourced/constants.sh"

source "${working\_dir}/\_sourced/messages.sh"

message\_welcome "Backing up the '${POSTGRES\_DB}' database..."

if [[ "${POSTGRES\_USER}" == "postgres" ]]; then

message\_error "Backing up as 'postgres' user is not allowed. Assign 'POSTGRES\_USER' env with another one and try again."

exit 1

fi

export PGHOST="${POSTGRES\_HOST}"

export PGPORT="${POSTGRES\_PORT}"

export PGUSER="${POSTGRES\_USER}"

export PGPASSWORD="${POSTGRES\_PASSWORD}"

export PGDATABASE="${POSTGRES\_DB}"

backup\_filename="${BACKUP\_FILE\_PREFIX}\_$(date '+%Y\_%m\_%dT%H\_%M\_%S').sql.gz"

pg\_dump | gzip > "${BACKUP\_DIR\_PATH}/${backup\_filename}"

message\_success "'${POSTGRES\_DB} database backup '${backup\_filename}' has been created successfully and placed in '${BACKUP\_DIR\_PATH}'"

Explanation:

Certainly! Let's break down the complete script:

```bash

#! /usr/bin/env bash

set -o errexit

set -o nounset

set -o pipefail

```

- The shebang line (`#! /usr/bin/env bash`) specifies that the script should be interpreted using the Bash shell. The next three lines (`set -o errexit`, `set -o nounset`, `set -o pipefail`) set various error-handling options:

- `set -o errexit`: Exits the script if any command it runs exits with a non-zero status (error).

- `set -o nounset`: Exits the script if it tries to use a variable that is not set.

- `set -o pipefail`: Exits the script if any command in a pipeline fails.

```bash

working\_dir="${dirname ${0})"

```

- There's a typo in this line. It should be `working\_dir="$(dirname "${0}")"` instead of `working\_dir="${dirname ${0})"`. This line sets the variable `working\_dir` to the directory containing the script.

```bash

source "${working\_dir}/\_sourced/constants.sh}"

source "${working\_dir}/\_sourced/messages.sh}"

```

- These lines source (include) the contents of the `constants.sh` and `messages.sh` files into the script. This is a way to modularize the code by separating constants and messages into external files.

```bash

message\_welcome "Backing up the '${POSTGRES\_DB}' database..."

```

- This line calls a function named `message\_welcome` (which is sourced from `messages.sh`) to print a welcome message indicating that the script is about to back up the PostgreSQL database specified by the `POSTGRES\_DB` environment variable.

```bash

if [[ "${POSTGRES\_USER}" == "postgres" ]]; then

message\_error "Backing up as 'postgres' user is not allowed. Assign 'POSTGRES\_USER' env with another one and try again."

exit 1

fi

```

- This conditional statement checks if the value of the `POSTGRES\_USER` environment variable is set to "postgres". If it is, an error message is displayed, and the script exits with a status code of 1.

```bash

export POSTGRES\_HOST="${POSTGRES\_HOST}"

export PGPORT="${POSTGRES\_PORT}"

export PGUSER="${POSTGRES\_USER}"

export PGPASSWORD="${POSTGRES\_PASSWORD}"

export PGDATABASE="${POSTGRES\_DB}"

```

Certainly! Let's break down the export commands and the last line of the script:

```bash

export PGHOST="${POSTGRES\_HOST}"

export PGPORT="${POSTGRES\_PORT}"

export PGUSER="${POSTGRES\_USER}"

export PGPASSWORD="${POSTGRES\_PASSWORD}"

export PGDATABASE="${POSTGRES\_DB}"

```

- These lines use the `export` command to set environment variables for a PostgreSQL database connection. These environment variables are commonly used by PostgreSQL-related tools and libraries to determine the connection details. Here's what each line does:

- `export PGHOST="${POSTGRES\_HOST}"`: Sets the PostgreSQL host (server) to the value of the `POSTGRES\_HOST` environment variable.

- `export PGPORT="${POSTGRES\_PORT}"`: Sets the PostgreSQL port to the value of the `POSTGRES\_PORT` environment variable.

- `export PGUSER="${POSTGRES\_USER}"`: Sets the PostgreSQL user to the value of the `POSTGRES\_USER` environment variable.

- `export PGPASSWORD="${POSTGRES\_PASSWORD}"`: Sets the PostgreSQL password to the value of the `POSTGRES\_PASSWORD` environment variable. Note that using passwords in environment variables is generally not recommended for security reasons; it's better to use more secure methods like password files or interactive password entry.

- `export PGDATABASE="${POSTGRES\_DB}"`: Sets the PostgreSQL database name to the value of the `POSTGRES\_DB` environment variable.

```bash

backup\_filename="${BACKUP\_FILE\_PREFIX}\_$(date '+%Y\_%m\_%dT%H\_%M\_%S').sql.gz"

```

- This line sets the variable `backup\_filename` to a string that includes a timestamp formatted as "YYYY\_MM\_DDTHH\_MM\_SS", appended to the `BACKUP\_FILE\_PREFIX`, and with a ".sql.gz" extension. The `date '+%Y\_%m\_%dT%H\_%M\_%S'` part generates the current date and time in the specified format.

Then:

pg\_dump | gzip > "${BACKUP\_DIR\_PATH}/${backup\_filename}"

message\_success "'${POSTGRES\_DB} database backup '${backup\_filename}' has been created successfully and placed in '${BACKUP\_DIR\_PATH}'"

create pg\_dump to create a backup file. This backup file will be stored in the backup dir path/backup filename, then display success message to user.

In summary, the export commands set up environment variables for connecting to a PostgreSQL database, and the last line creates a filename for a backup file based on a timestamp and a specified prefix.

In summary, this script sets up error-handling options, determines the working directory, sources external files containing constants and messages, displays a welcome message, checks if the PostgreSQL user is set to "postgres" (which is not allowed), exits the script with an error message if the condition is true, and exports necessary environment variables for a PostgreSQL backup operation.

Create backups file:

#! /usr/bin/env bash

set -o errexit

set -o nounset

set -o pipefail

working\_dir="$(dirname ${0})"

source "${working\_dir}/\_sourced/constants.sh"

source "${working\_dir}/\_sourced/messages.sh"

message\_welcome "These are the backups you have"

ls -lht "${BACKUP\_DIR\_PATH}"

Create Database Restore Scripts: Script that allows us to restore a database from a backup

#! /usr/bin/env bash

set -o errexit

set -o nounset

set -o pipefail

working\_dir="$(dirname ${0})"

source "${working\_dir}/\_sourced/constants.sh"

source "${working\_dir}/\_sourced/messages.sh"

if [[ -z ${1+x} ]]; then

message\_error "Backup filename is not specified. This is a required parameter, please provide one and try again"

exit 1

fi

backup\_filename="${BACKUP\_DIR\_path}/${1}"

if [[ ! -f "${backup\_filename}" ]]; then

message\_error "no backup with the specified backup file name was found. Sheck out the 'backups maintenance script to see if there is one and try again"

exit 1

fi

message\_welcom "Restoring the '${POSTGRES\_DB}' database from the '${backup\_filename}' backup..."

if [[ "${POSTGRES\_USER}" == "postgres" ]]; then

message\_error "Restoring as 'postgres' user is not allowed. Assign a 'POSTGRES\_USER' env with another one and try again"

exit 1

fi

export PGHOST="${POSTGRES\_HOST}"

export PGPORT="${POSTGRES\_PORT}"

export PGUSER="${POSTGRES\_USER}"

export PGPASSWORD="${POSTGRES\_PASSWORD}"

export PGDATABASE="${POSTGRES\_DB}"

message\_info "Dropping the database..."

dropdb "${BGDATABASE}"

message\_info "Creating a new database..."

createdb --owner="${POSTGRES\_USER}"

message\_info "Applying the backup to the new database"

gunzip -c "${backup\_filename}" | psql"${POSTGRES\_DB}"

message\_success "The '${POSTGRES\_DB}' database has been restored successfully from the '${backup\_filename}' backup"

Explanation:

```bash

#! /usr/bin/env bash

set -o errexit

set -o nounset

set -o pipefail

```

- The shebang (`#! /usr/bin/env bash`) specifies that the script should be interpreted using the Bash shell. The next three lines (`set -o errexit`, `set -o nounset`, `set -o pipefail`) set various error-handling options:

- `set -o errexit`: Exits the script if any command it runs exits with a non-zero status (error).

- `set -o nounset`: Exits the script if it tries to use a variable that is not set.

- `set -o pipefail`: Exits the script if any command in a pipeline fails.

```bash

working\_dir="$(dirname ${0})"

```

- This line sets the variable `working\_dir` to the directory containing the script.

```bash

source "${working\_dir}/\_sourced/constants.sh"

source "${working\_dir}/\_sourced/messages.sh"

```

- These lines source (include) the contents of the `constants.sh` and `messages.sh` files into the script. This is a way to modularize the code by separating constants and messages into external files.

```bash

if [[ -z ${1+x} ]]; then

message\_error "Backup filename is not specified. This is a required parameter, please provide one and try again"

exit 1

fi

backup\_filename="${BACKUP\_DIR\_path}/${1}"

```

- This block checks if the first command-line argument (`${1}`) is unset or empty. If it is, an error message is displayed, and the script exits with a status code of 1. Otherwise, it constructs the `backup\_filename` by combining the `BACKUP\_DIR\_path` constant and the specified filename.

```bash

if [[ ! -f "${backup\_filename}" ]]; then

message\_error "No backup with the specified backup filename was found. Check the 'backups maintenance script to see if there is one and try again"

exit 1

fi

```

- This block checks if the specified backup file (`"${backup\_filename}"`) exists. If it doesn't, an error message is displayed, and the script exits with a status code of 1.

```bash

message\_welcome "Restoring the '${POSTGRES\_DB}' database from the '${backup\_filename}' backup..."

```

- This line calls a function named `message\_welcome` (which is sourced from `messages.sh`) to print a welcome message indicating that the script is about to restore the PostgreSQL database specified by the `POSTGRES\_DB` environment variable from the specified backup file.

```bash

if [[ "${POSTGRES\_USER}" == "postgres" ]]; then

message\_error "Restoring as 'postgres' user is not allowed. Assign a 'POSTGRES\_USER' env with another one and try again"

exit 1

fi

```

- This block checks if the value of the `POSTGRES\_USER` environment variable is set to "postgres". If it is, an error message is displayed, and the script exits with a status code of 1.

```bash

export PGHOST="${POSTGRES\_HOST}"

export PGPORT="${POSTGRES\_PORT}"

export PGUSER="${POSTGRES\_USER}"

export PGPASSWORD="${POSTGRES\_PASSWORD}"

export PGDATABASE="${POSTGRES\_DB}"

```

- These lines use the `export` command to set environment variables for a PostgreSQL database connection, similar to what was done in the backup script. These environment variables are necessary for connecting to the PostgreSQL server.

```bash

message\_info "Dropping the database..."

dropdb "${BGDATABASE}"

message\_info "Creating a new database..."

createdb --owner="${POSTGRES\_USER}"

message\_info "Applying the backup to the new database"

gunzip -c "${backup\_filename}" | psql "${POSTGRES\_DB}"

message\_success "The '${POSTGRES\_DB}' database has been restored successfully from the '${backup\_filename}' backup"

```

- These lines use various `message\_` functions to display informative messages about the steps being taken. Here's what each step does:

- `dropdb "${BGDATABASE}"`: Drops (deletes) the existing database specified by `POSTGRES\_DB`.

- `createdb --owner="${POSTGRES\_USER}"`: Creates a new database with the same name as `POSTGRES\_DB` and sets the owner to the user specified by `POSTGRES\_USER`.

- `gunzip -c "${backup\_filename}" | psql "${POSTGRES\_DB}"`: Reads the compressed backup file and pipes it into the `psql` command to apply the backup to the newly created database.

In summary, this script restores a PostgreSQL database from a specified backup file. It performs error checking, displays informative messages, and executes the necessary PostgreSQL commands to drop the existing database, create a new one, and apply the backup.

To add changes to docker containers, take them down and then bring them back up:

Docker-compose -f local.yml down

docker-compose -f local.yml up --build -d --remove-orphans

Migrate postgres database:

docker-compose -f local.yml exec postgres backup

To see existing backups:

Docker-compose -f local.yml exec postgres backups

Which should return:

These are the backups you have

total 4.0K

-rw-r--r-- 1 root root 3.3K Nov 21 20:54 backup\_2023\_11\_21T20\_54\_45.sql.gz

The **-rw-r--r--** string represents the file permissions in a Linux/Unix file system. Here's what each part means:

* The first character indicates the type of file or directory. In this case, it's a regular file, and the character is **-**.
* The next three characters (**rw-**) represent the owner's (user's) permissions. In this example, the owner has read (**r**) and write (**w**) permissions but does not have execute (**x**) permission.
* The next three characters (**r--**) represent the group's permissions. The group has read permission but does not have write or execute permission.
* The last three characters (**r--**) represent the permissions for others (anyone else who is not the owner or in the group). Others also have read permission but do not have write or execute permission.

So, in summary:

* **-rw-r--r--** means read and write permissions for the owner, read-only permission for the group and others.
* The **1** after the permissions indicates the number of hard links to the file.

In the context of your backup file (**backup\_2023\_11\_21T20\_54\_45.sql.gz**), it means that the file is owned by the root user, and the root user has read and write permissions, while others have only read permission.

To restore the database:

docker-compose -f local.yml exec postgres restore backup\_to\_restore\_from

SET UP MAKEFILES:

What are Makefiles?

A Makefile is a special file used to control the build process of a software project. It contains a set of rules and dependencies that specify how the project should be built. Makefiles are commonly used in software development to automate the compilation of source code into executable programs or libraries.

The Make utility reads the Makefile and executes the specified commands to build the target output. Each rule in the Makefile defines a target (what to build), dependencies (what the target depends on), and the commands to execute to build the target.

Here's a simple example of a Makefile:

```make

# Makefile example

CC = gcc

CFLAGS = -Wall

all: myprogram

myprogram: main.c utils.c

$(CC) $(CFLAGS) -o myprogram main.c utils.c

clean:

rm -f myprogram

```

In this example:

- `CC` is a variable defining the compiler.

- `CFLAGS` is a variable defining compiler flags.

- `all` is a target specifying that the default target is "myprogram."

- `myprogram` is a target specifying that the executable "myprogram" depends on the source files "main.c" and "utils.c." The command to build the target is given in the next line.

- `clean` is a target that specifies a command to clean up generated files.

By running the `make` command in the same directory as the Makefile, it will execute the specified commands to build the target (in this case, "myprogram"). The `make clean` command would clean up generated files.

Makefiles are widely used in projects with complex build processes to ensure that the build is executed consistently and efficiently. They are especially common in C and C++ projects, but they can be used for any project with a build process that involves multiple source files and dependencies.

For this project, we will use the Make utility to shorten the docker commands. For this, you create a file (Makefile) that defines a set of tasks to be executed

For example:

Makefiles can be particularly useful for simplifying and shortening Docker commands, especially when you have complex build or deployment processes. Below is a simple example demonstrating how you might use a Makefile with Docker commands:

```make

# Makefile for Docker-related commands

# Variables

IMAGE\_NAME = mydockerimage

CONTAINER\_NAME = mydockercontainer

# Build the Docker image

build:

docker build -t $(IMAGE\_NAME) .

# Run the Docker container

run:

docker run --name $(CONTAINER\_NAME) -p 8080:80 -d $(IMAGE\_NAME)

# Stop and remove the Docker container

stop:

docker stop $(CONTAINER\_NAME)

docker rm $(CONTAINER\_NAME)

# Remove the Docker image

clean:

docker rmi $(IMAGE\_NAME)

```

In this example:

- `IMAGE\_NAME` is a variable representing the name of the Docker image.

- `CONTAINER\_NAME` is a variable representing the name of the Docker container.

You can use the Makefile as follows:

- To build the Docker image, you would run `make build`.

- To run the Docker container, you would run `make run`.

- To stop and remove the Docker container, you would run `make stop`.

- To remove the Docker image, you would run `make clean`.

This helps simplify complex Docker commands and provides a consistent and easy-to-remember interface. Additionally, as your project grows, you can extend the Makefile with additional targets for other Docker-related tasks.

Makefiles are versatile, and you can tailor them to match your specific project requirements. They are particularly helpful for automating and streamlining repetitive tasks in your development workflow.

Create Makefile in root directory:

build:

docker-compose -f local.yml up --build -d --remove-orphans

up:

docker-compose -f local.yml up -d

down:

docker-compose -f local.yml down

show-logs:

docker-compose -f local.yml logs

show-logs-api:

docker-compose -f local.yml logs

makemigrations:

docker-compose -f local.yml run --rm api python manage.py makemigrations

migrate:

docker-compose -f local.yml run --rm api python manage.py migrate

collectstatic:

docker-compose -f local.yml run --rm api python manage.py collectstatic --no-input --clear

superuser:

docker-compose -f local.yml run --rm api python manage.py createsuperuser

down-v:

docker-compose -f local.yml down -v

volume:

docker volume inspect src\_local\_postgres\_data

authors-db:

docker-compose -f local.yml exec postgres psql --username=marmarq --dbname=authors-live

flake8:

docker-compose -f local.yml exec api flake8 .

black-check:

docker-compose -f local.yml exec api black --check --exclude=migrations .

black-diff:

docker-compose -f local.yml exec api black --diff --exclude=migrations .

black:

docker-compose -f local.yml exec api black --exclude=migrations .

isort-check:

docker-compose -f local.yml exec api isort . --check-only --skip venv --skip migrations

isort-diff:

docker-compose -f local.yml exec api isort . --check-only --diff --skip venv --skip migrations

isort:

docker-compose -f local.yml exec api isort . --skip venv --skip migrations

To run commands, just type in terminal: make [command]

For example: make build (to build the images and containers)

USER MODEL: We will create our own custom user model, that means we have to delete the migrations we already made at the beginning. TO achieve this, I need to bring down containers and dump the volumes. To do that:

make down-v

CREATE CUSTOM USER: There are two options to do this.

Option 1: Abstract User.

This is a god option if you are happy with Django’s base User model and only want to remove the username field, for example, or add other fields. The Abstract User class is a full user model, complete with fields as abstract classes, so you can inherit from it and add your own profile fields and methods

Option 2: Abstract Base User.

Use this option if you want to start from scratch by creating a completely new user model with custom fields. This class only contains the authentication functionality, but no fields. These need to be supplied when you subclass.

For this project we will use Option 1, the Abstract User, which we will customize using pseudo primary keys

In the core\_app folder/users, create managers.py file

Define the Custom User Manager in this file:

The Custom User Manager is an interface through which database query operations are provided to Django models.

Use email for authentication instead of a user name

We create a method to validate the email address

Then, we override the default Django create\_user method so it accepts the email as a parameter

Then, override Django’s default create\_superuser method to also accept an email as a parameter

from django.contrib.auth.base\_user import BaseUserManager

from django.core.exceptions import ValidationError

from django.core.validators import validate\_email

from django.utils.translation import gettext\_lazy as \_

class CustomUserManager(BaseUserManager):

def email\_validator(self, email):

try:

validate\_email(email)

return True

except ValidationError:

raise ValueError(\_("You must provide a valid email address"))

def create\_user(self, first\_name, last\_name, email, password, \*\*extra\_fields):

if not first\_name:

raise ValueError(\_("Users must have a first name"))

if not last\_name:

raise ValueError(\_("Users must have a last name"))

if email:

email = self.normalize\_email(email)

self.email\_validator(email)

else:

raise ValueError(\_("Users must have a valid email address"))

user = self.model(first\_name=first\_name, last\_name=last\_name, email=email, \*\*extra\_fields)

user.set\_password(password)

extra\_fields.setdefault("is\_staff", False)

extra\_fields.setdefault("is\_superuser", False)

user.save(using=self.\_db)

return user

def create\_superuser(self, first\_name, last\_name, email, password, \*\*extra\_fields):

extra\_fields.setdefault("is\_staff", True)

extra\_fields.setdefault("is\_superuser", True)

extra\_fields.setdefault("is\_active", True)

if extra\_fields.get("is\_staff") is not True:

raise ValueError(\_("Super user must have is\_staff=True"))

if extra\_fields.get("is\_superuser") is not True:

raise ValueError(\_("Super user must have is\_superuser=True"))

if not password:

raise ValueError(\_("Super user must have a valid password"))

if email:

email = self.normalize\_email(email)

self.email\_validator(email)

else:

raise ValueError(\_("Super user must have a valid email address"))

user = self.create\_user(first\_name, last\_name, email, password, \*\*extra\_fields)

user.save(using=self.\_db)

return user

Explanation

Certainly! Let's break down the `CustomUserManager` code and provide a detailed explanation of its role in creating a custom user in Django:

```python

from django.contrib.auth.base\_user import BaseUserManager

from django.core.exceptions import ValidationError

from django.core.validators import validate\_email

from django.utils.translation import gettext\_lazy as \_

```

These lines import necessary modules and classes from Django, including the `BaseUserManager`, which is a base class for creating custom user managers.

```python

class CustomUserManager(BaseUserManager):

```

Here, you define a custom user manager class (`CustomUserManager`) that inherits from `BaseUserManager`. This class will be responsible for managing user creation.

```python

def email\_validator(self, email):

try:

validate\_email(email)

return True

except ValidationError:

raise ValueError(\_("You must provide a valid email address"))

```

The `email\_validator` method is a utility method to validate email addresses using Django's built-in `validate\_email` function. It raises a `ValueError` with a custom error message if the validation fails.

```python

def create\_user(self, first\_name, last\_name, email, password, \*\*extra\_fields):

```

The `create\_user` method is responsible for creating a regular user. It takes various user-related parameters (first name, last name, email, password) along with additional `\*\*extra\_fields`.

```python

if not first\_name:

raise ValueError(\_("Users must have a first name"))

if not last\_name:

raise ValueError(\_("Users must have a last name"))

if email:

email = self.normalize\_email(email)

self.email\_validator(email)

else:

raise ValueError(\_("Users must have a valid email address"))

```

These lines perform basic validations on the provided user information (first name, last name, email). If any of these checks fail, a `ValueError` is raised with a corresponding error message.

```python

user = self.model(first\_name=first\_name, last\_name=last\_name, email=email, \*\*extra\_fields)

user.set\_password(password)

extra\_fields.setdefault("is\_staff", False)

extra\_fields.setdefault("is\_superuser", False)

user.save(using=self.\_db)

return user

```

This block creates a new user instance (`user`) using the custom user model (`self.model`) with the provided information and extra fields. It sets the user's password using `set\_password` and saves the user to the database.

```python

def create\_superuser(self, first\_name, last\_name, email, password, \*\*extra\_fields):

```

The `create\_superuser` method creates a superuser with additional permissions beyond a regular user.

```python

extra\_fields.setdefault("is\_staff", True)

extra\_fields.setdefault("is\_superuser", True)

extra\_fields.setdefault("is\_active", True)

if extra\_fields.get("is\_staff") is not True:

raise ValueError(\_("Superuser must have is\_staff=True"))

if extra\_fields.get("is\_superuser") is not True:

raise ValueError(\_("Superuser must have is\_superuser=True"))

if not password:

raise ValueError(\_("Superuser must have a valid password"))

if email:

email = self.normalize\_email(email)

self.email\_validator(email)

else:

raise ValueError(\_("Superuser must have a valid email address"))

user = self.create\_user(first\_name, last\_name, email, password, \*\*extra\_fields)

user.save(using=self.\_db)

return user

```

This block sets default values for additional fields specific to superusers, such as `is\_staff`, `is\_superuser`, and `is\_active`. It performs additional checks and validations similar to the `create\_user` method. The superuser is created by calling the `create\_user` method and saving the user to the database.

This `CustomUserManager` provides a structured way to create regular users and superusers, enforcing certain validations and default values. It's a crucial component in creating a custom user model in Django.

Remember, this is designed to work with a custom user model that you define in your Django project. The custom user model typically extends `AbstractBaseUser` and `PermissionsMixin`.

CUSTOM USER MODEL

This application will use UUIDs (universal Unique Identifiers) as primary keys instead of auto incrementing integer primary keys. Pros of UUIDs:

* They are unique across space and time (they are always unique, even if generated computers or times.
* Are not sequential
* Are not predictable
* Not easy to guess
* Not easily incrementable or decrementable

Cons:

* At scale, they can cause massive insert performance issues because the primary key is a clustered index (a clustered index is an index that determines the physical order of the data in a table. Unlike a non-clustered index, which creates a separate structure to store the index and data is stored in a random order, a clustered index organizes the data rows in the table based on the order of the index key). Since UUIDs are not sequential, inserting new records with a UUID clustered index may lead to page splits in the underlying storage. Page splits can negatively impact insert performance because the database engine may need to move existing data to accommodate the new record.
* **Impact on Insert Performance**: Since UUIDs are not sequential, inserting new records with a UUID clustered index may lead to page splits in the underlying storage. Page splits can negatively impact insert performance because the database engine may need to move existing data to accommodate the new record.
* **Fragmentation**: Over time, as records are inserted, deleted, or updated, the table can become fragmented if the UUIDs are not generated in a sequential manner. Fragmentation can lead to decreased query performance.
* **Size**: UUIDs are 128-bit values, which are larger than traditional integer-based primary keys. This can result in larger indexes and data pages.
* There is no quick sort by id chronology available, so the latest items will have to be found by using timestamps, which are slower.

To avoid these disadvantages, we use a Pseudo Primary Key:

A "Pseudo Primary Key" is not a standard term used in the context of databases. In this case it could refer to a mechanism or strategy employed to address certain challenges associated with using UUIDs (Universally Unique Identifiers) as primary keys in databases.

When dealing with UUIDs in databases, some common challenges include:

1. \*\*Non-Sequential Nature\*\*: UUIDs are typically generated to be globally unique, but they are not guaranteed to be sequential. This can lead to issues related to index fragmentation and suboptimal storage performance, especially when used as clustered indexes.

2. \*\*Size\*\*: UUIDs are 128-bit values, which are larger than traditional integer-based primary keys. This can result in larger indexes and data pages.

3. \*\*Insert Performance\*\*: Inserting new records with UUIDs as primary keys may result in page splits and impact insert performance, especially when using the UUID as a clustered index.

To address these challenges, a common approach is to introduce a separate, sequential, and more compact primary key that serves as a clustered index, often referred to as a "Pseudo Primary Key." This key is used for managing the physical order of records in the database. Meanwhile, the UUID remains as a unique identifier for external use or as a surrogate key.

Here's a basic example in the context of a database table:

```sql

CREATE TABLE MyTable (

PseudoPrimaryKey INT IDENTITY(1,1) PRIMARY KEY,

UUIDColumn UNIQUEIDENTIFIER DEFAULT NEWID(),

-- Other columns...

);

```

In this example, `PseudoPrimaryKey` is an integer-based identity column that serves as a clustered index, providing sequential ordering for the records. The `UUIDColumn` is still present, providing a globally unique identifier.

This approach allows for efficient storage and retrieval of data while preserving the benefits of using UUIDs for uniqueness. Queries that require sequential access or range scans can leverage the pseudo primary key, while the UUID remains available for situations where a globally unique identifier is needed.

It's important to note that the specific implementation may vary based on the database management system (DBMS) being used.

Implementing Pseudo Primary Keys for this Project:

Users/models.py file: Create user model using the pseudo pks

This is how you create the pseudo pk in Django User model:

class User(AbstractBaseUser, PermissionError):

pkid = models.BigAutoField(primary\_key=True, editable=False)

id = models.UUIDField(default=uuid.uuid4, editable=False, unique=True)

1. **id = models.BigAutoField(primary\_key=True)**: This line creates an **id** field, which is a **BigAutoField**. **BigAutoField** is similar to **AutoField**, but it is designed for larger integer values. Setting **primary\_key=True** indicates that this field will serve as the primary key for the model, making it the pseudo primary key.
2. **uuid = models.UUIDField(default=uuid.uuid4, editable=False, unique=True)**: This line creates a **uuid** field, which is a **UUIDField**. It has the following attributes:
   * **default=uuid.uuid4**: This sets a default value for the UUID field, using the **uuid4** function to generate a random UUID.
   * **editable=False**: This means that the UUID field won't be editable in Django admin or forms.
   * **unique=True**: This ensures that each user will have a unique UUID.
3. **objects = CustomUserManager()**: This line associates a custom manager (**CustomUserManager**) with the model. The custom manager may include methods for creating users, such as **create\_user** and **create\_superuser**.

The combination of the **id** field and the **uuid** field provides a mechanism for efficient storage and retrieval of records. The **id** serves as the pseudo primary key for ordering records in the database, while the **uuid** remains a globally unique identifier.

This setup is beneficial for scenarios where you need sequential ordering for certain database operations while retaining the advantages of using UUIDs for uniqueness. It's a common approach to address the challenges associated with UUIDs as primary keys in databases.

Email field will be used instead of username for authentication. db\_index=True is used to create an index for the email field

This is the finished model:

import uuid

from django.db import models

from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin

from django.utils import timezone

from django.utils.translation import gettext\_lazy as \_

from .managers import CustomUserManager

class User(AbstractBaseUser, PermissionError):

pkid = models.BigAutoField(primary\_key=True, editable=False)

id = models.UUIDField(default=uuid.uuid4, editable=False, unique=True)

first\_name = models.CharField(verbose\_name=("first name"), max\_length=50)

last\_name = models.CharField(verbose\_name=("last name"), max\_length=50)

email = models.EmailField(verbose\_name=("email\_address"),db\_index=True, unique=True)

is\_staff = models.BooleanField(default=False)

is\_active = models.BooleanField(default=True)

date\_joined = models.DateTimeField(default=timezone.now)

USERNAME\_FIELD = 'email'

REQUIRED\_FIELDS = ["first\_name", "last\_name"]

objects = CustomUserManager()

class Meta:

verbose\_name = \_("user")

verbose\_name\_plural = \_("users")

def \_\_str\_\_(self):

return self.first\_name

@property

def get\_full\_name(self):

return f"{self.first\_name.title()} {self.last\_name.title()}"

@property

def get\_short\_name(self):

return self.last\_name

This is what each section is doing:

```python

import uuid

from django.db import models

from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin

from django.utils import timezone

from django.utils.translation import gettext\_lazy as \_

from .managers import CustomUserManager

```

- `import uuid`: Imports the `uuid` module, which is used for generating unique identifiers.

- `from django.db import models`: Imports the Django models module, which is used to define the structure of the database tables.

- `from django.contrib.auth.models import AbstractBaseUser, PermissionsMixin`: Imports the base classes for creating a custom user model.

- `from django.utils import timezone`: Imports the Django timezone module.

- `from django.utils.translation import gettext\_lazy as \_`: Imports a function `\_` for translation of verbose names.

```python

class User(AbstractBaseUser, PermissionsMixin):

```

- `class User(AbstractBaseUser, PermissionsMixin)`: Defines a custom user model that inherits from `AbstractBaseUser` and `PermissionsMixin`. This means the `User` model will have the functionalities of a user in the Django authentication system.

```python

pkid = models.BigAutoField(primary\_key=True, editable=False)

```

- `pkid = models.BigAutoField(primary\_key=True, editable=False)`: Defines a big integer field `pkid` as the primary key for the model. It is an automatically incremented field and is not editable.

```python

id = models.UUIDField(default=uuid.uuid4, editable=False, unique=True)

```

- `id = models.UUIDField(default=uuid.uuid4, editable=False, unique=True)`: Defines a UUID field `id` with a default value of a randomly generated UUID using `uuid.uuid4()`. This field is not editable and must be unique.

```python

first\_name = models.CharField(verbose\_name=("first name"), max\_length=50)

last\_name = models.CharField(verbose\_name=("last name"), max\_length=50)

email = models.EmailField(verbose\_name=("email\_address"), db\_index=True, unique=True)

```

- `first\_name`, `last\_name`, and `email`: Define fields for the user's first name, last name, and email. `db\_index=True` creates an index on the email field for faster lookups, and `unique=True` ensures that each email address is unique.

```python

is\_staff = models.BooleanField(default=False)

is\_active = models.BooleanField(default=True)

```

- `is\_staff` and `is\_active`: Boolean fields indicating whether the user is a staff member (admin) and whether the user account is active.

```python

date\_joined = models.DateTimeField(default=timezone.now)

```

- `date\_joined`: A DateTime field indicating when the user account was created, with a default value set to the current time.

```python

USERNAME\_FIELD = 'email'

REQUIRED\_FIELDS = ["first\_name", "last\_name"]

```

- `USERNAME\_FIELD`: Specifies that the email field is used as the unique identifier for authentication.

- `REQUIRED\_FIELDS`: Lists the additional fields required when creating a superuser (in addition to the email and password).

```python

objects = CustomUserManager()

```

- `objects`: An instance of the `CustomUserManager` class, which manages the creation and handling of user objects.

```python

class Meta:

verbose\_name = \_("user")

verbose\_name\_plural = \_("users")

```

- `Meta`: A class inside the model used to specify metadata. Here, it sets the verbose names for the singular and plural forms of the model.

```python

def \_\_str\_\_(self):

return self.first\_name

```

- `\_\_str\_\_`: A method that returns a human-readable representation of the model instance. Here, it returns the user's first name.

```python

@property

def get\_full\_name(self):

return f"{self.first\_name.title()} {self.last\_name.title()}"

```

- `@property`: Decorator to define a property method.

- `get\_full\_name`: A property method that returns the full name of the user in title case.

```python

@property

def get\_short\_name(self):

return self.last\_name

```

- `get\_short\_name`: Another property method that returns the user's last name.

Go to authorsapi/settings/base.py, and at the end of the file (above logging), add:

AUTH\_USER\_MODEL = "users.User"

This tells Django the location of our user model

USER CREATION ADMIN FORM:

TO register inn the Admin, we need to subclass the user creation form and the user change form. These are the forms used to create and edit users.

Go to core\_apps/users/ create forms.py

from django import forms

from django.contrib.auth import forms as admin\_forms

from django.contrib.auth import get\_user\_model

User = get\_user\_model()

#subclass the user change form:

class UserChangeForm(admin\_forms.UserChangeForm):

class Meta(admin\_forms.UserChangeForm.Meta):

model = User

class UserCreationForm(admin\_forms.UserCreationForm):

class Meta(admin\_forms.UserCreationForm.Meta):

model = User

fields = ('first\_name', 'last\_name', 'email')

error\_messages = {

"duplicate\_email":"A user with this email already exists"

}

#override the default clean email method to check if email already exists:

def clean\_email(self):

email = self.cleaned\_data["email"]

try:

User.objects.get(email=email)

except User.DoesNotExist:

return email

raise forms.ValidationError(self.error\_messages["duplicate\_email"])

1. Import necessary modules and get the user model:

```python

from django import forms

from django.contrib.auth import forms as admin\_forms

from django.contrib.auth import get\_user\_model

User = get\_user\_model()

```

- `django.contrib.auth.forms`: It contains various forms related to authentication.

- `get\_user\_model()`: It returns the User model that is active in the current project.

2. Subclass the user change form:

```python

class UserChangeForm(admin\_forms.UserChangeForm):

class Meta(admin\_forms.UserChangeForm.Meta):

model = User

```

- This class inherits from `admin\_forms.UserChangeForm`, which is a default form for changing user information in the Django admin.

- `Meta` class is used to specify metadata about the form, and it inherits metadata from the base `UserChangeForm.Meta` class. It sets the model to your custom User model.

3. Subclass the user creation form:

```python

class UserCreationForm(admin\_forms.UserCreationForm):

class Meta(admin\_forms.User.CreationForm.Meta):

model = User

fields = ('first\_name', 'last\_name', 'email')

```

- Similar to the `UserChangeForm`, this class inherits from `admin\_forms.UserCreationForm`, which is a default form for creating a new user in the Django admin.

- `Meta` class is used to specify metadata about the form, and it inherits metadata from the base `User.CreationForm.Meta` class. It sets the model to your custom User model and includes the specified fields.

4. Override the default `clean\_email` method:

```python

def clean\_email(self):

email = self.cleaned\_data["email"]

try:

User.objects.get(email=email)

except User.DoesNotExist:

return email

raise forms.ValidationError(self.error\_messages["duplicate\_email"])

```

- This method is used to clean and validate the email field.

- It checks if a user with the given email already exists in the database. If not, it returns the cleaned email.

- If a user with the given email exists, it raises a `ValidationError` with a custom error message.

This `forms.py` is designed to work with Django's admin and provides custom forms for changing and creating users, including additional validation for duplicate emails. The use of Django's built-in forms and conventions makes it easy to extend and customize user-related functionality.

USERADMIN: Once the forms are created, we need to tell Django Admin to use these forms by subclassing the user admin in the users admin.py file:

from django.contrib import admin

from django.contrib.auth.admin import UserAdmin as BaseUserAdmin

from django.utils.translation import gettext\_lazy as \_

from .forms import UserChangeForm, UserCreationForm

from .models import User

class UserAdmin(BaseUserAdmin):

ordering = ["email"]

form = UserChangeForm

add\_form = UserCreationForm

model = User

list\_display = ["pkid", "id", "email", "first\_name", "last\_name", "is\_staff", "is\_Active" ]

list\_display\_links = ["pkid", "id", "email"]

list\_filter = [

"email", "is\_staff", "is\_active"

]

fieldsets = (

(\_("Login Credentials"), {"fields":("email", "password")}),

(\_("Personal Info"), {"fields":("first\_name", "last\_name")}),

(\_("Permissions and Groups"), {"fields":("is\_active", "is\_staff", "is\_superuser","groups", "user\_permissions")}),

(\_("Important Dates"), {"fields":("last\_login", "date\_joined")}),

)

add\_fieldsets = (

(None, {

"classes":("wide",),

"fields":("email", "first\_name", "last\_name", "password1", "password2"),

})

)

search\_fields = ["email", "first\_name", "last\_name"]

admin.site.register(User, UserAdmin)

Explanation:

```python

from django.contrib import admin

from django.contrib.auth.admin import UserAdmin as BaseUserAdmin

from django.utils.translation import gettext\_lazy as \_

from .forms import UserChangeForm, UserCreationForm

from .models import User

```

1. `from django.contrib import admin`: Import the Django admin module.

2. `from django.contrib.auth.admin import UserAdmin as BaseUserAdmin`: Import the default `UserAdmin` from Django's auth module as `BaseUserAdmin`.

3. `from django.utils.translation import gettext\_lazy as \_`: Import the `\_` function for translations.

```python

class UserAdmin(BaseUserAdmin):

ordering = ["email"]

form = UserChangeForm

add\_form = UserCreationForm

model = User

```

4. `class UserAdmin(BaseUserAdmin):`: Create a custom `UserAdmin` class that inherits from `BaseUserAdmin`.

5. `ordering = ["email"]`: Set the default ordering of users in the admin panel based on the email field.

6. `form = UserChangeForm`: Use the `UserChangeForm` from your forms.py for updating user details.

7. `add\_form = UserCreationForm`: Use the `UserCreationForm` from your forms.py for creating new users.

8. `model = User`: Set the model for this admin class to your custom `User` model.

```python

list\_display = ["pkid", "id", "email", "first\_name", "last\_name", "is\_staff", "is\_active"]

list\_display\_links = ["pkid", "id", "email"]

list\_filter = ["email", "is\_staff", "is\_active"]

```

9. `list\_display`: Display these fields in the user list view in the admin panel.

10. `list\_display\_links`: Make these fields clickable to go to the user detail view.

11. `list\_filter`: Add filters on the right side of the user list view for these fields.

```python

fieldsets = (

(\_("Login Credentials"), {"fields":("email", "password")}),

(\_("Personal Info"), {"fields":("first\_name", "last\_name")}),

(\_("Permissions and Groups"), {"fields":("is\_active", "is\_staff", "is\_superuser", "groups", "user\_permissions")}),

(\_("Important Dates"), {"fields":("last\_login", "date\_joined")}),

)

```

12. `fieldsets`: Organize the fields in the user detail view into collapsible sections.

```python

add\_fieldsets = (

(None, {

"classes":("wide",),

"fields":("email", "first\_name", "last\_name", "password1", "password2"),

})

)

```

13. `add\_fieldsets`: Define the fields and layout for the user creation view.

```python

search\_fields = ["email", "first\_name", "last\_name"]

```

14. `search\_fields`: Enable searching for users based on these fields in the admin panel.

```python

admin.site.register(User, UserAdmin)

```

15. Register your custom `User` model with the custom `UserAdmin` class in the admin site.

This setup customizes the appearance and behavior of the Django admin panel for your custom user model.

CONFIGURE YASG:

In Django Rest Framework (DRF), \*\*drf-yasg\*\* stands for "Yet Another Swagger Generator." It is a third-party Django package that generates real Swagger/OpenAPI 2.0 specifications from a Django Rest Framework API. Swagger/OpenAPI is a standard for documenting and interacting with RESTful APIs.

The main features of drf-yasg include:

1. \*\*Automatic Documentation\*\*: drf-yasg automatically generates documentation for your API based on your DRF serializers, views, and viewsets. This documentation is presented in a user-friendly Swagger UI.

2. \*\*Interactive API Exploration\*\*: Swagger UI provides an interactive interface that allows users to explore and interact with your API directly from the documentation. Users can make requests to endpoints and see responses in real-time.

3. \*\*Browsable API Integration\*\*: drf-yasg integrates seamlessly with DRF's browsable API views. This means that the same views that provide the browsable HTML interface will also generate Swagger documentation.

4. \*\*Customization\*\*: You can customize the generated Swagger documentation by providing additional metadata, modifying settings, or using decorators on your DRF views.

To use drf-yasg in your Django project, you typically need to:

1. Install the package: `pip install drf-yasg`.

2. Add it to your `INSTALLED\_APPS` in your Django settings.

3. Configure the Swagger UI settings in your project's URL configuration.

Here's a basic example of how to set up drf-yasg in your `urls.py`:

```python

from django.contrib import admin

from django.urls import path, re\_path, include

from rest\_framework import permissions

from drf\_yasg.views import get\_schema\_view

from drf\_yasg import openapi

schema\_view = get\_schema\_view(

openapi.Info(

title="Your API",

default\_version='v1',

description="Your API description",

terms\_of\_service="https://www.yourapp.com/terms/",

contact=openapi.Contact(email="contact@yourapp.com"),

license=openapi.License(name="Your License"),

),

public=True,

permission\_classes=(permissions.AllowAny,),

)

urlpatterns = [

path('admin/', admin.site.urls),

path('api/', include('your\_app.urls')), # Include your DRF API URLs here

path('swagger/', schema\_view.with\_ui('swagger', cache\_timeout=0), name='schema-swagger-ui'),

# Add any other paths as needed

]

```

In this example, visiting the '/swagger/' endpoint in your browser will display the Swagger UI documentation for your API. You can customize the `openapi.Info` parameters to provide information about your API.

In this project, the urls.py looks like this:

from django.contrib import admin

from django.urls import path, include

from django.conf import settings

from drf\_yasg import openapi

from drf\_yasg.views import get\_schema\_view

from rest\_framework import permissions

schema\_view = get\_schema\_view(

openapi.Info(

title="Authors Haven API",

default\_version="v1",

description="API endpoints for Authors Haven API Class",

contact=openapi.Contact(email="marilynjmarquez@gmail.com"),

license=openapi.License(name="MIT")

),

public=True,

permission\_classes=(permissions.AllowAny),

)

urlpatterns = [

path(settings.ADMIN\_URL, admin.site.urls),

path("redoc/", schema\_view.with\_ui("redoc", cache\_timeout=0)),

]

admin.site.site\_header = "Authors Haven API Admin"

admin.site.site\_title = "Authors Haven API Admin Portal"

admin.site.index\_title = "Welcome to Authors Have API Portal"

Explanation:

1. \*\*`get\_schema\_view` and `schema\_view`\*\*: These are part of the drf-yasg package and are used to generate the Swagger/OpenAPI schema for your API. The `get\_schema\_view` function is called to create a schema view, and it is configured with information such as the API title, version, description, contact details, and license. This schema view will be used to render both Swagger UI and ReDoc.

2. \*\*`urlpatterns`\*\*: This is a list of URL patterns for your project. Let's go through each item:

- \*\*`path(settings.ADMIN\_URL, admin.site.urls)`\*\*: This sets up the Django admin interface at the URL specified in `settings.ADMIN\_URL`. It uses the standard Django admin site.

- \*\*`path("redoc/", schema\_view.with\_ui("redoc", cache\_timeout=0))`\*\*: This creates a URL pattern for the ReDoc documentation. ReDoc is an alternative to Swagger UI for rendering OpenAPI documentation. It uses the `schema\_view` with the UI set to "redoc."

3. \*\*Admin Site Customization\*\*: The following lines customize the Django admin site appearance:

- \*\*`admin.site.site\_header`\*\*: Sets the text displayed as the header on the admin site.

- \*\*`admin.site.site\_title`\*\*: Sets the text displayed as the title in the browser tab for the admin site.

- \*\*`admin.site.index\_title`\*\*: Sets the text displayed on the main admin index page.

In summary, this `urls.py` configuration includes the Django admin site, Swagger UI, and ReDoc. The Swagger UI is not explicitly mentioned here, but it is available at the default '/swagger/' endpoint due to the default configuration provided by drf-yasg. ReDoc is available at the "/redoc/" endpoint. The admin site is also included with custom branding. The `schema\_view` is a central piece for generating OpenAPI documentation and is configured with metadata about your API.

After finishing this, you need to REBUILD CONTAINERS AND RUN MIGRATIONS

In the terminal (from the ROOT DIRECTORY) enter:

make build

WHEN RUNNING DOCKER IN AN M2 MACHINE and get SCRAM error:

Export DOCKER\_DEFAULT\_PLATFORM=linux/amd64

(do this every time after restarting the machine)

SUPERUSER:

Email: marilynjmarquez@gmail.com

Name: Marilyn

Last Name: Marquez

Password: 12483351MM

**NGINX as reverse Proxy/Load Balancer**

NGINX configuration: Add nginx folder to Docker directory, create Dockerfile and default.conf file inside the nginx folder

nginx/Dockerfile:

FROM nginx:1.23.3-alpine

RUN rm /etc/nginx/conf.d/default.conf

COPY ./default.conf /etc/nginx/conf.d/default.conf

Configure nginx in default.conf:

upstream api {

server api:8000;

}

server {

client\_max\_body\_size 20M;

listen 80;

location /api/v1/ {

proxy\_pass http://api;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

proxy\_redirect off;

}

location /supersecret {

proxy\_pass http://api;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

proxy\_redirect off;

}

location /redoc/ {

proxy\_pass http://api;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

proxy\_redirect off;

}

location /staticfiles/ {

alias /app/staticfiles/;

}

location /mediafiles/ {

alias /app/mediafiles/;

}

}

}

Certainly, let's break down the Nginx configuration script line by line:

1. `upstream api { server api:8000; }`:

- This defines an upstream block named `api` with a single server named `api` running on port `8000`.

- The `upstream` directive is used to define groups of servers that can be referenced by the `proxy\_pass` directive later in the configuration.

2. `server {`:

- Starts the configuration block for a server.

3. `client\_max\_body\_size 20M;`:

- Sets the maximum allowed size of the client request body to 20 megabytes.

4. `listen 80;`:

- Configures the server to listen on port 80.

5. `location /api/v1/ { ... }`:

- Defines a location block for requests to the path `/api/v1/`.

- `proxy\_pass http://api;`: Passes the request to the upstream server defined in the `api` block.

- `proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;`: Sets the `X-Forwarded-For` header to the original client IP address.

- `proxy\_set\_header Host $host;`: Sets the `Host` header to the original host value.

- `proxy\_redirect off;`: Disables automatic redirection.

6. `location /supersecret { ... }`:

- Defines a location block for requests to the path `/supersecret`.

- Similar to the `/api/v1/` location block.

7. `location /redoc/ { ... }`:

- Defines a location block for requests to the path `/redoc/`.

- Similar to the other location blocks.

8. `location /staticfiles/ { alias /app/staticfiles/; }`:

- Defines a location block for requests to the path `/staticfiles/`.

- Uses the `alias` directive to map requests to the local filesystem path `/app/staticfiles/`.

9. `location /mediafiles/ { alias /app/mediafiles/; }`:

- Defines a location block for requests to the path `/mediafiles/`.

- Uses the `alias` directive to map requests to the local filesystem path `/app/mediafiles/`.

10. `}`:

- Ends the server configuration block.

11. `}`:

- Ends the `upstream` block.

This Nginx configuration acts as a reverse proxy for an API server running on port 8000, handles different paths, sets appropriate headers, and manages static and media files. It's a common setup for deploying Django applications with Nginx. The `upstream` block allows you to easily scale your application by adding more servers if needed.

**NGINX COMPOSE FILE**

Make changes to local.yml:

Change ports: “8000:8000” to expose “8000”

(exposes the port without publishing it to the host)

We are exposing port 8000 to other service in the compose file. In our case, we want the nginx server to have access to this port.

Add nginx service in the same local.yml file:

After mailhog, add:

nginx:

restart: always

depends\_on:

- api

volumes:

- static\_volume:/app/staticfiles

- media\_volume:/app/mediafiles

build:

context: ./docker/local/nginx

dockerfile: Dockerfile

ports:

- "8080:80"

networks:

- authors-api

**CELERY SETUP**

Celery needs a message broker and a result backend.

Message Broker : intermediary program used as the transport for producing or consuming tasks

Result Backend: Used to store the result of a Celery task. In production we will use Redis

Redis: In-memory data structure that can be used as a database cache and message broker.

Redis stores data in-memory, making it a fast and efficient tool for storing and retrieving data

Flower: real-time web application monitoring and admin tool for celery

We will use Django Celery Email, which is a package that allows you to use celery to send asynchronous emails from your Django application

To setup celery, start by adding the following to base.txt:

redis==4.5.1

celery==5.2.7

flower==1.2.0

django-celery-email==3.0.0

Watch files is used to watch out for changes in files and runs a command when the files change. We will use it to watch our celery files. In local.txt file:

watchfiles==0.18.1

Configure celery: In authors\_api, add a file called celery.py:

import os

from celery import Celery

from django.conf import settings

#TODO: change in production

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "authors\_api.settings.local")

#Name celery instance:

app = Celery("authors\_api")

app.config\_from\_object("django.conf:settings", namespace="CELERY")

app.autodiscover\_tasks(lambda: settings.INSTALLED\_APPS) #automaticaly finds tasks in all installed Django applications

Certainly! Let's break down the code in your `celery.py` file:

```python

import os

from celery import Celery

from django.conf import settings

```

- `import os`: This imports the Python `os` module, which provides a way of using operating system-dependent functionality, such as reading or writing to the file system.

- `from celery import Celery`: This imports the `Celery` class from the Celery library. `Celery` is used to create a Celery application instance.

- `from django.conf import settings`: This imports the `settings` module from Django's configuration. This allows the Celery application to use the same settings as your Django project.

```python

# TODO: change in production

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "authors\_api.settings.local")

```

- `os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "authors\_api.settings.local")`: This sets the `DJANGO\_SETTINGS\_MODULE` environment variable to specify the settings module for the Django project. In this case, it's set to `"authors\_api.settings.local"`. This line is a TODO reminder to change this setting in production, as typically you would want to use different settings for development and production.

```python

# Name celery instance:

app = Celery("authors\_api")

```

- `app = Celery("authors\_api")`: This creates an instance of the `Celery` class named "authors\_api". This instance will be used to configure and run the Celery worker.

```python

app.config\_from\_object("django.conf:settings", namespace="CELERY")

```

- `app.config\_from\_object("django.conf:settings", namespace="CELERY")`: This configures the Celery application using the settings from the Django project. It looks for settings prefixed with `CELERY\_` in your Django settings module.

Certainly! Let's break down the code in your `celery.py` file:

```python

import os

from celery import Celery

from django.conf import settings

```

- `import os`: This imports the Python `os` module, which provides a way of using operating system-dependent functionality, such as reading or writing to the file system.

- `from celery import Celery`: This imports the `Celery` class from the Celery library. `Celery` is used to create a Celery application instance.

- `from django.conf import settings`: This imports the `settings` module from Django's configuration. This allows the Celery application to use the same settings as your Django project.

```python

# TODO: change in production

os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "authors\_api.settings.local")

```

- `os.environ.setdefault("DJANGO\_SETTINGS\_MODULE", "authors\_api.settings.local")`: This sets the `DJANGO\_SETTINGS\_MODULE` environment variable to specify the settings module for the Django project. In this case, it's set to `"authors\_api.settings.local"`. This line is a TODO reminder to change this setting in production, as typically you would want to use different settings for development and production.

```python

# Name celery instance:

app = Celery("authors\_api")

```

- `app = Celery("authors\_api")`: This creates an instance of the `Celery` class named "authors\_api". This instance will be used to configure and run the Celery worker.

```python

app.config\_from\_object("django.conf:settings", namespace="CELERY")

```

- `app.config\_from\_object("django.conf:settings", namespace="CELERY")`: This configures the Celery application using the settings from the Django project. It looks for settings prefixed with `CELERY\_` in your Django settings module.

In summary, this `celery.py` file sets up a Celery application instance, configures it to use the Django project's settings, and specifies the settings module to be used. It's an essential part of integrating Celery with a Django project for handling asynchronous tasks.

Next, need to amend \_\_init\_\_.py:

from .celery import app as celery\_app

\_\_all\_\_ = ("celery\_app",)

This modification in the `\_\_init\_\_.py` file is related to how Django and Celery work together. Let me break it down:

1. \*\*`from .celery import app as celery\_app`\*\*: This line imports the `app` object from the `celery.py` file in the same package (directory). The `as celery\_app` part is just creating an alias for the imported object. This line essentially brings the Celery application instance (`app`) into the namespace of the package.

2. \*\*`\_\_all\_\_ = ("celery\_app",)`\*\*: This line is setting the `\_\_all\_\_` attribute of the package. In Python, `\_\_all\_\_` is a list of public names that should be imported when `from module import \*` is used. In this case, it's specifying that when someone does `from your\_package import \*`, only `celery\_app` should be imported. This is a way to explicitly state what is considered the public API of your package.

By doing this, you make it convenient for other parts of your project to import the Celery application instance from the package. For example, in your Django project, if you need to use Celery, you can now import the Celery app instance like this:

```python

from your\_package import celery\_app

```

It's a common practice to organize the Celery configuration in a separate file (`celery.py`) and then make the Celery app instance easily accessible throughout the project by including it in the package's `\_\_init\_\_.py`.

Add Redis and Celery environment variables:

In envs/local/.env\_django:

CELERY\_BROKER=redis://redis:6379/0

DOMAIN=localhost:8080

EMAIL\_PORT=1025

CELERY\_FLOWER\_USER=admin

CELERY\_FLOWER\_PASSWORD=Pass123456

Add at the end of local.py:

EMAIL\_BACKEND = "djcelery\_email.backends.CeleryEmailBackend"

EMAIL\_HOST = env("EMAIL\_HOST", default="mailhog")

EMAIL\_PORT = env("EMAIL\_PORT")

DEFAULT\_FROM\_EMAIL = "marilynjmarquez@gmail.com"

DOMAIN = env("DOMAIN")

SITE\_NAME="Authors Haven"

Add celery worker to base.py:

CELERY\_BROKER\_URL = env("CELERY\_BROKER")

CELERY\_RESULT\_BACKEND = CELERY\_BROKER\_URL

CELERY\_ACCEPT\_CONTENT = ["json"]

CELERY\_TASK\_SERIALIZER = ["json"]

CELERY\_RESULT\_SERIALIZER = ["json"]

CELERY\_RESULT\_MAX\_RETRIES = 10

CELERY\_TASK\_SEND\_SENT\_EVENT = True

if USE\_TZ:

CELERY\_TIMEZONE = TIME\_ZONE

1. `CELERY\_BROKER\_URL = env("CELERY\_BROKER")`

- This line sets the Celery broker URL to the value of the `CELERY\_BROKER` environment variable. It uses the `env` function to retrieve the value from the environment.

2. `CELERY\_RESULT\_BACKEND = CELERY\_BROKER\_URL`

- This line sets the result backend for Celery tasks to the same URL as the broker. The result backend is where Celery will store the results of asynchronous tasks.

3. `CELERY\_ACCEPT\_CONTENT = ["json"]`

- Specifies the list of content types that the worker will accept. In this case, it's set to JSON, indicating that the worker can accept tasks serialized in JSON format.

4. `CELERY\_TASK\_SERIALIZER = ["json"]`

- Defines the serialization method for task messages. Here, it's set to JSON, so task messages will be serialized in JSON format.

5. `CELERY\_RESULT\_SERIALIZER = ["json"]`

- Similar to `CELERY\_TASK\_SERIALIZER`, this sets the serialization format for task results to JSON.

6. `CELERY\_RESULT\_MAX\_RETRIES = 10`

- Sets the maximum number of retries that Celery will attempt if a task result fails.

7. `CELERY\_TASK\_SEND\_SENT\_EVENT = True`

- Enables the sending of a task-sent event when a task message is sent. This can be useful for monitoring and tracking the progress of tasks.

8. `if USE\_TZ: CELERY\_TIMEZONE = TIME\_ZONE`

- If the `USE\_TZ` setting in your Django project is enabled (which means you're using time zone support), this line sets the Celery timezone to the value specified in `TIME\_ZONE`. This ensures consistency between Django and Celery in handling time zones.

These settings help configure Celery to work with your Django project, specifying how tasks are serialized, where results are stored, and other related configurations.

Finally, add “djcelery\_email” to the THIRD\_PARTY\_APPS in local.py

**DOCKER COMPOSE CONFIG (CELERY AND REDIS)**

Edit local.yml:

version: "3.9"

services:

api:

build:

context: .

dockerfile: ./docker/local/django/Dockerfile

volumes:

- .:/app:z

- static\_volume:/app/staticfiles

- media\_volume:/app/mediafiles

expose:

- "8000"

env\_file:

- ./.envs/.local/.django

- ./.envs/.local/.postgres

depends\_on:

- postgres

- mailhog

command: /start

networks:

- authors-api

postgres:

build:

context: .

dockerfile: ./docker/local/postgres/Dockerfile

volumes:

- local\_postgres\_data:/var/lib/postgresql/data

- local\_postgres\_data\_backups:/backups

env\_file:

- ./.envs/.local/.postgres

networks:

- authors-api

mailhog:

image: mailhog/mailhog:v1.0.0

container\_name: mailhog

ports:

- "8025:8025"

networks:

- authors-api

redis:

image: redis:7-alpine

networks:

- authors-api

celery\_worker:

build:

context: .

dockerfile: ./docker/local/django/Dockerfile

command: /start-celeryworker

volumes:

- .:/app

env\_file:

- ./.envs/.local/.django

- ./.envs/.local/.postgres

depends\_on:

- redis

- potgres

- mailhog

networks:

- authors-api

flower:

build:

context: .

dockerfile: ./docker/local/django/Dockerfile

command: /start-flower

volumes:

- flower-data:/data

env\_file:

- ./.envs/.local/.django

- ./.envs/.local/.postgres

ports:

- "5555:5555"

depends\_on:

- redis

- potgres

- mailhog

networks:

- authors-api

nginx:

restart: always

depends\_on:

- api

volumes:

- static\_volume:/app/staticfiles

- media\_volume:/app/mediafiles

build:

context: ./docker/local/nginx

dockerfile: Dockerfile

ports:

- "8080:80"

networks:

- authors-api

networks:

authors-api:

driver: bridge

volumes:

static\_volume:

media\_volume:

local\_postgres\_data: {}

local\_postgres\_data\_backups: {}

flower\_data: {}

After local.yml is done, go to docker/local/Django/Dockerfile and create commands to start celery worker and flower --above COPY . ${APP\_HOME}:

ARG PYTHON\_VERSION=3.11.6-bullseye

FROM python:${PYTHON\_VERSION} as python

FROM python as python-build-stage

ARG BUILD\_ENVIRONMENT=local

RUN apt-get update && \

apt-get install --no-install-recommends -y \

build-essential \

libpq-dev \

gettext && \

apt-get purge -y --autoremove -o APT::AutoRemove::RecommendsImportant=false && \

rm -rf /var/lib/apt/lists/\*

WORKDIR /requirements

COPY ../../requirements/base.txt .

COPY ../../requirements/local.txt .

RUN pip wheel --wheel-dir /usr/src/app/wheels -r base.txt

RUN pip wheel --wheel-dir /usr/src/app/wheels -r local.txt

FROM python as python-run-stage

ARG BUILD\_ENVIRONMENT=local

ARG APP\_HOME=/app

ENV PYTHONDONTWRITEBYTECODE 1

ENV PYTHONUNBUFFERED 1

ENV BUILD\_ENV ${BUILD\_ENVIRONMENT}

WORKDIR ${APP\_HOME}

COPY --from=python-build-stage /usr/src/app/wheels /wheels

RUN pip install --no-cache-dir --no-index --find-links=/wheels/ /wheels/\* && \

rm -rf /wheels/

COPY ./docker/local/django/entrypoint /entrypoint

RUN sed -i 's/\r$//g' /entrypoint

RUN chmod +x /entrypoint

COPY ./docker/local/django/start /start

RUN sed -i 's/\r$//g' /start

RUN chmod +x /start

COPY ./docker/local/django/celery/worker/start /start-celeryworker

RUN sed -i 's/\r$//g' /start-celeryworker

RUN chmod +x /start-celeryworker

COPY ./docker/local/django/celery/worker/start /start-flower

RUN sed -i 's/\r$//g' /start-flower

RUN chmod +x /-flower

COPY . ${APP\_HOME}

ENTRYPOINT ["/entrypoint"]

Next: Create a celery folder in Docker/local/Django, create celery folder, and inside that folder, create worker and flower folders. Inside each one of these folders, create start file:

In celery/worker/start: This is. Abash script that will run celery worker for our application using the “watch files” command, which watches for changes in Python files and automatically restarts the celery worker. Watch files should only be used for development, since the purpose of these files is to monitor for changes in source files and automatically restart processes when changes are detected. It is useful to speed up development, but in production it introduces potential instability and security risks within production environments. In production you should use system or supervisor to manage and monitor celery workers

celery/worker/start:

#!/bin/bash

set -o errexit

set -o nounset

exec watchfiles celery.\_\_main\_\_.main --args '-A authors api.celery worker -l INFO'

Explanation:

```bash

#!/bin/bash

```

- This line specifies that the script should be interpreted using the Bash shell.

```bash

set -o errexit

```

- This line sets the errexit option, which makes the script exit immediately if any command it runs exits with a non-zero status (indicating an error).

```bash

set -o nounset

```

- This line sets the nounset option, which treats unset variables as an error and causes the script to exit.

```bash

exec watchfiles celery.\_\_main\_\_.main --args '-A authors api.celery worker -l INFO'

```

- The `exec` command replaces the current process with a new one, and `watchfiles` is likely a tool or alias that watches for changes in files and automatically restarts a process when changes are detected.

- `celery.\_\_main\_\_.main` is the main entry point for running Celery commands.

- `--args` is used to pass arguments to the Celery worker.

- `'-A authors api.celery worker -l INFO'` are the arguments passed to the Celery worker. Breaking it down:

- `-A authors` specifies the Celery application module (the `-A` flag is followed by the module path).

- `api.celery` is the path to the Celery instance within the specified module.

- `worker` indicates that this command is for running a Celery worker.

- `-l INFO` sets the logging level to INFO, which controls the amount of information printed to the console.

So, in summary, this script is designed to run a Celery worker and uses `watchfiles` to automatically restart the worker when changes are detected in the relevant files. The options at the beginning (`set -o errexit` and `set -o nounset`) ensure that the script is more robust by exiting on errors and treating unset variables as errors.

celery/flower/start:

#!/bin/bash

set -o errexit

set -o nounset

exec watchfiles celery.\_\_main\_\_.main \

--args \

"-A authors\_api.celery -b \"${CELERY\_BROKER}\" flower --basic\_auth=\"${CELERY\_FLOWER\_USER}:${CELERY\_FLOWER\_PASSWORD}\""

Explanation:

1. `#!/bin/bash`: This line is called a shebang. It specifies the interpreter that should be used to execute the script. In this case, it's set to use the Bash shell (`bash`).

2. `set -o errexit`: This line sets the `errexit` option, which makes the script exit immediately if any command it runs exits with a non-zero status (indicating an error).

3. `set -o nounset`: This line sets the `nounset` option, which treats unset variables as an error when they are referenced.

4. `exec watchfiles celery.\_\_main\_\_.main \` : The `exec` command replaces the current shell process with the command that follows it. In this case, it runs the `watchfiles` command with the arguments that follow.

5. `--args \` : The `--args` flag is used to pass arguments to the command that `watchfiles` will execute. The backslash (`\`) at the end of the line indicates that the command continues on the next line.

6. `"-A authors\_api.celery -b \"${CELERY\_BROKER}" flower --basic\_auth=\"${CELERY\_FLOWER\_USER}:${CELERY\_FLOWER\_PASSWORD}\""`: This is the command that `watchfiles` will execute. Let's break it down further:

- `"-A authors\_api.celery -b \"${CELERY\_BROKER}"`: This part specifies the Celery application and the broker URL. The broker URL is obtained from the `CELERY\_BROKER` environment variable.

- `flower --basic\_auth=\"${CELERY\_FLOWER\_USER}:${CELERY\_FLOWER\_PASSWORD}\""`: This part runs Flower, which is a real-time web-based monitoring tool for Celery. It specifies basic authentication using the `CELERY\_FLOWER\_USER` and `CELERY\_FLOWER\_PASSWORD` environment variables.

In summary, this script sets up and runs a command using `watchfiles` to monitor changes and restarts the specified Celery worker with Flower for monitoring. It also includes some error-handling options to exit the script if errors occur.

Now we need to rebuild the containers because we added environmental variables.

To check if Flower is working:

Localhost:5555

CELERY\_FLOWER\_USER=admin

CELERY\_FLOWER\_PASSWORD=Pass123456

DJANGO FUNCTIONALITY

Go to: core\_apps/common/models

In models.py: Create Abstract model

In Django, an abstract model is a way to define a model that doesn't create a database table for itself. Instead, it is intended to be used as a base class for other models. Abstract models are useful when you want to factor out common fields and behavior among multiple models without having a separate database table for the abstract model.

Here are the key points about abstract models in Django:

1. **No Database Table:** An abstract model does not have a database table associated with it. It exists solely to provide fields and methods that can be reused by other models.
2. **Inherits Fields and Methods:** Other models can inherit from the abstract model, gaining access to its fields and methods.
3. **Meta Options:** An abstract model can have its own **Meta** class with options just like a regular model. However, the **abstract** option must be set to **True**.

models.py/Timestamped model:

c

lass TimeStampedModel(models.Model):

pkid = models.BigAutoField(primary\_key=True, editable=False)

id = models.UUIDField(default = uuid.uuid4, editable=False, unique=True)

created\_at = models.DateTimeField(auto\_now\_add=True)

updated\_at = models.DateTimeField(auto\_now=True)

class Meta:

abstract=True

ordering = ["-created\_at", -"updated\_at"]

Next, create the models in the Profile application:

Create separate Users and Profile applications because of separation of concerns. Users handles fields and concerns related to authentication, profile can have models and functionality that relates to other fields.

profiles/models.py:

from django.contrib.auth import get\_user\_model

from django.db import models

from django.utils.translation import gettext\_lazy as \_

from django\_countries.fields import CountryField

from phonenumber\_field.modelfields import PhoneNumberField

from core\_apps.common.models import TimeStampedModel

User =get\_user\_model()

class Profile(TimeStampedModel):

class Gender(models.TextChoices):

MALE = "M", \_("Male")

FEMALE = "F", \_("Female")

OTHER = "O", \_("Other")

user = models.OneToOneField(User, on\_delete=models.CASCADE, related\_name='profile')

phone\_number = PhoneNumberField(verbose\_name=('phone number'), max\_lenght=30, default="+14132345678")

about\_me = models.TextField(verbose\_name=\_("about me"), default="say something about yourself")

gender = models.CharField(

verbose\_name = \_("gender"),

choices = Gender.choices, default=Gender.OTHER, max\_length=20

)

country = models.CountryField(verbose\_name = \_("country"), default = "KE", blank=False, null=False)

city = models.CharField(

verbose\_name=\_("city"),

max\_length=180,

default="Boston",

blank=False,

null=False

)

profile\_photo = models.ImageField(verbose\_name=\_("profile photo"), default="/profile\_default.png")

twitter\_handle = models.CharField(verbose\_name=\_("twitter handle"), max\_length=20, blank=True)

followers = models.ManyToManyField("self", symmertrical=False, related\_name="following", blank=True)

def \_\_str\_\_(self):

return f"{self.user.first\_name}'s Profile'"

def follow(self, profile):

self.followers.add(profile)

def unfollow(self, profile):

self.followers.remove(profile)

def check\_following(self, profile):

return self.followers.filter(pkid=profile.pkid).exists()

Explanation:

Certainly! Let's go through the code line by line:

```python

from django.contrib.auth import get\_user\_model

from django.db import models

from django.utils.translation import gettext\_lazy as \_

from django\_countries.fields import CountryField

from phonenumber\_field.modelfields import PhoneNumberField

from core\_apps.common.models import TimeStampedModel

```

1. Import necessary modules and classes:

- `get\_user\_model`: Retrieves the User model currently in use.

- `models`: Module for defining Django models.

- `gettext\_lazy as \_`: A utility for making strings translatable.

- `CountryField`: A field for storing countries.

- `PhoneNumberField`: A field for storing phone numbers.

- `TimeStampedModel`: A custom model that includes timestamp fields.

```python

User = get\_user\_model()

```

2. Get the User model. `User` now represents the user model currently in use in the Django project.

```python

class Profile(TimeStampedModel):

```

3. Define a model named `Profile` that inherits from `TimeStampedModel`. This model will represent user profiles.

```python

class Gender(models.TextChoices):

MALE = "M", \_("Male")

FEMALE = "F", \_("Female")

OTHER = "O", \_("Other")

```

4. Define choices for the `gender` field using `models.TextChoices`. The available choices are Male, Female, and Other.

```python

user = models.OneToOneField(User, on\_delete=models.CASCADE, related\_name='profile')

```

5. Create a one-to-one relationship with the `User` model. This field is used to link a user to their profile.

```python

phone\_number = PhoneNumberField(verbose\_name=('phone number'), max\_length=30, default="+14132345678")

```

6. Define a field to store the user's phone number using the `PhoneNumberField`.

```python

about\_me = models.TextField(verbose\_name=\_("about me"), default="say something about yourself")

```

7. Create a text field for a user to provide information about themselves.

```python

gender = models.CharField(

verbose\_name=\_("gender"),

choices=Gender.choices, default=Gender.OTHER, max\_length=20

)

```

8. Define a character field for the user's gender, using the choices defined earlier.

```python

country = models.CountryField(verbose\_name=\_("country"), default="KE", blank=False, null=False)

```

9. Create a field to store the user's country, using the `CountryField`.

```python

city = models.CharField(

verbose\_name=\_("city"),

max\_length=180,

default="Boston",

blank=False,

null=False

)

```

10. Define a character field for the user's city.

```python

profile\_photo = models.ImageField(verbose\_name=\_("profile photo"), default="/profile\_default.png")

```

11. Create an image field for the user's profile photo.

```python

twitter\_handle = models.CharField(verbose\_name=\_("twitter handle"), max\_length=20, blank=True)

```

12. Define a field for the user's Twitter handle.

```python

followers = models.ManyToManyField("self", symmetrical=False, related\_name="following", blank=True)

```

13. Create a many-to-many relationship field for the user's followers.

```python

def \_\_str\_\_(self):

return f"{self.user.first\_name}'s Profile'"

```

14. Define a string representation for the `Profile` object.

```python

def follow(self, profile):

self.followers.add(profile)

```

15. Define a method to add a follower to the user's profile.

```python

def unfollow(self, profile):

self.followers.remove(profile)

```

16. Define a method to remove a follower from the user's profile.

```python

def check\_following(self, profile):

return self.followers.filter(pkid=profile.pkid).exists()

```

17. Define a method to check if a user is following another user.

That covers the entire `Profile` model. It represents a user's profile with various fields and methods for handling followers and following.

SIGNALS:

Use signals to ensure that a user’s profile is created whenever we add a new user. Signals are a way for different parts of the app to communicate with each other without having to know about each other directly. Signals allow one part of the application to send a message (called a signal) to another part of your application without the sender and receiver having to be couled

In Profiles, create a signals.py file:

import logging

from django.db.models.signals import post\_save

from django.dispatch import receiver

from authors\_api.settings.base import AUTH\_USER\_MODEL

from core\_apps.profiles.models import Profile

logger = logging.getLogger(\_\_name\_\_)

@receiver(post\_save, sender=AUTH\_USER\_MODEL)

def create\_user\_profile(sender, instance, created, \*\*kwargs):

if created:

Profile.objects.create(user=instance)

logger.info(f"{instance}'s profile has been created")

Explanation:

Certainly! This is a Django signal that's being used to automatically create a user profile whenever a new user is created. Let me break it down for you:

```python

import logging

from django.db.models.signals import post\_save

from django.dispatch import receiver

from authors\_api.settings.base import AUTH\_USER\_MODEL

from core\_apps.profiles.models import Profile

# Get an instance of a logger

logger = logging.getLogger(\_\_name\_\_)

# Define a signal receiver function

@receiver(post\_save, sender=AUTH\_USER\_MODEL)

def create\_user\_profile(sender, instance, created, \*\*kwargs):

# Check if the user instance was just created

if created:

# Create a corresponding profile for the user

Profile.objects.create(user=instance)

# Log a message indicating that the profile has been created

logger.info(f"{instance}'s profile has been created")

```

Explanation:

1. \*\*Import Statements:\*\*

- `import logging`: Import the Python `logging` module to log messages.

- `from django.db.models.signals import post\_save`: Import the `post\_save` signal from Django, which is triggered after a model's `save()` method is called.

- `from django.dispatch import receiver`: Import the `receiver` decorator to connect the signal to a receiver function.

- `from authors\_api.settings.base import AUTH\_USER\_MODEL`: Import the user model specified in the Django project's settings.

- `from core\_apps.profiles.models import Profile`: Import the `Profile` model, which is assumed to be in the `core\_apps.profiles.models` module.

2. \*\*Logger Setup:\*\*

- `logger = logging.getLogger(\_\_name\_\_)`: Create a logger instance to log messages. The logger is named after the current module.

3. \*\*Signal Receiver Function:\*\*

- `@receiver(post\_save, sender=AUTH\_USER\_MODEL)`: Decorate the `create\_user\_profile` function to be a receiver of the `post\_save` signal for the specified user model (`AUTH\_USER\_MODEL`).

- `def create\_user\_profile(sender, instance, created, \*\*kwargs)`: The receiver function takes the sender (the model class), the instance being saved, a boolean `created` indicating if the instance was just created, and additional keyword arguments.

- `if created:`: Check if the user instance was just created.

- `Profile.objects.create(user=instance)`: If created, create a new profile instance associated with the user instance.

- `logger.info(f"{instance}'s profile has been created")`: Log an informational message indicating that the user's profile has been created.

This signal is connected to the `post\_save` signal of the specified user model (`AUTH\_USER\_MODEL`). When a new user is created, the `create\_user\_profile` function is called, creating a corresponding profile for the user and logging the event.

Next, modify core\_app/profile/apps.py (this is to let Django know about the signals you created):

from django.apps import AppConfig

from django.utils.translation import gettext\_lazy as \_

class ProfilesConfig(AppConfig):

default\_auto\_field = "django.db.models.BigAutoField"

name = "core\_apps.profiles"

verbose\_name = \_("Profiles")

def ready(self):

from core\_apps.profiles import signals

Then, register models in the admin.py:

from django.contrib import admin

from .models import Profile

# Register your models here.

class ProfileAdmin(admin.ModelAdmin):

list\_display = ["pkid", "id", "user", "gender", "phone\_number", "city", "country"]

list\_display\_links = ["pkid", "id", "user"]

list\_filter = ["id", "pkid"]

admin.site.register(Profile, ProfileAdmin)

With the containers running, run migrations:

make makemigrations

make migrate