Certainly! 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: