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: