To avoid encountering `PermissionError: [Errno 13] Permission denied` when building your Docker image, you need to ensure that the permissions within the image are set correctly during the build process. Here are some steps you can follow:

1. \*\*Specify Permissions in Dockerfile\*\*:

When creating directories or files within your Dockerfile, ensure that you explicitly set the permissions to accommodate the user that will run the application inside the container. This is typically done using `RUN`, `COPY`, or `ADD` commands followed by `chmod`.

For example:

```dockerfile

FROM python:3.9

# Create a directory and set permissions

RUN mkdir -p /var/lib/test/sessions && \

chmod -R 755 /var/lib/test/sessions

# Set the working directory

WORKDIR /app

# Copy your application code

COPY . /app

# Install dependencies and perform other setup tasks

RUN pip install -r requirements.txt

# Specify the command to run your application

CMD ["python", "app.py"]

```

In this example:

- `mkdir -p /var/lib/test/sessions` creates the directory `/var/lib/test/sessions`.

- `chmod -R 755 /var/lib/test/sessions` sets permissions so that the owner has full access (read, write, execute) and group/others have read/execute permissions.

2. \*\*Use User and Group IDs\*\*:

When running processes inside Docker containers, it's a good practice to use user and group IDs that match those outside the container. This avoids permission issues related to different user mappings.

- Use `USER` instruction in Dockerfile to set the user:

```dockerfile

FROM python:3.9

RUN groupadd -r myuser && useradd -r -g myuser myuser

# Create a directory and set permissions

RUN mkdir -p /var/lib/test/sessions && \

chown -R myuser:myuser /var/lib/test/sessions && \

chmod -R 755 /var/lib/test/sessions

# Set the working directory

WORKDIR /app

# Copy your application code

COPY . /app

# Install dependencies and perform other setup tasks

RUN pip install -r requirements.txt

# Specify the command to run your application

CMD ["python", "app.py"]

# Run as non-root user

USER myuser

```

Replace `myuser` with the appropriate user and group names or IDs for your setup.

3. \*\*Avoid Running as Root\*\*:

Running containers as the root user (`USER root`) can lead to permission issues because the host filesystem permissions might not align with those inside the container. Always aim to run containers as non-root users whenever possible.

4. \*\*Build and Run with Correct Permissions\*\*:

When building the Docker image, ensure you have sufficient permissions to access and modify the necessary files and directories on the host system. Use `sudo` if required to execute Docker commands with elevated privileges.

By following these practices, you ensure that your Docker image sets up the necessary permissions correctly during the build process, thereby preventing `PermissionError: [Errno 13] Permission denied` issues when the container runs. Always test your Dockerfile thoroughly to catch and resolve such issues early in the development process.

how to run containers as non-root user? and how to Use sudo if required to execute Docker commands?

Running Docker containers as a non-root user and using `sudo` for Docker commands can be important for security and convenience. Here's a guide to help you manage both:

### Running Containers as a Non-Root User

To run Docker containers as a non-root user, follow these steps:

1. \*\*Add Your User to the Docker Group\*\*:

By default, Docker commands require root privileges. To allow a non-root user to run Docker commands, add the user to the Docker group.

```bash

sudo usermod -aG docker your\_username

```

Replace `your\_username` with the actual username. After running this command, you need to log out and log back in for the group change to take effect.

2. \*\*Verify Docker Group Membership\*\*:

To verify that you’ve been added to the Docker group, you can use the `groups` command:

```bash

groups your\_username

```

You should see `docker` listed among the groups.

3. \*\*Running Containers\*\*:

Once you’re added to the Docker group, you can run Docker containers without needing `sudo`. For example:

```bash

docker run -d --name my\_container my\_image

```

4. \*\*Handling Permissions Inside Containers\*\*:

When running a container as a non-root user, ensure the container's user has the correct permissions to access necessary resources. You can specify a user in the Dockerfile or using the `-u` flag with `docker run`.

\*\*In Dockerfile\*\*:

```Dockerfile

USER myuser

```

\*\*Using `docker run`\*\*:

```bash

docker run -u myuser -d --name my\_container my\_image

```

### Using `sudo` to Execute Docker Commands

If you prefer or need to use `sudo` for Docker commands (e.g., if you haven't added your user to the Docker group), you can do so with these steps:

1. \*\*Running Docker Commands with `sudo`\*\*:

Simply prepend `sudo` to your Docker commands:

```bash

sudo docker run -d --name my\_container my\_image

```

2. \*\*Manage Docker as Root\*\*:

If you need to perform operations that require root access (e.g., modifying Docker configurations or managing Docker service), you can use `sudo` to run Docker commands:

```bash

sudo systemctl restart docker

sudo docker network ls

```

3. \*\*Configure `sudo` for Docker Commands\*\*:

If you want to configure `sudo` for a specific user or group, you can modify the `/etc/sudoers` file using `visudo`. This file defines user permissions for `sudo` commands.

```bash

sudo visudo

```

Add a line to grant specific users or groups permissions to run Docker commands without a password:

```bash

your\_username ALL=(ALL) NOPASSWD: /usr/bin/docker

```

Replace `your\_username` with the actual username.

### Summary

- \*\*To run Docker containers as a non-root user\*\*, add the user to the Docker group (`docker`) and ensure proper permissions inside containers.

- \*\*To use `sudo` for Docker commands\*\*, simply prepend `sudo` to the commands, and if needed, configure `sudo` settings for convenience.

These practices help in maintaining both security and ease of use when working with Docker.

If you want to configure your Odoo Docker image to start without requiring an initial database connection and then create the database later, you can follow these steps in your Dockerfile:

1. \*\*Extend Odoo Docker Image:\*\*

Start by extending the official Odoo Docker image in your Dockerfile:

```dockerfile

FROM odoo:latest

```

2. \*\*Modify Odoo Configuration:\*\*

Odoo reads its configuration from a file named `odoo.conf`. To delay the database connection, create a placeholder `odoo.conf` file with a dummy database configuration. Later, you'll replace this with the actual database details.

Create a placeholder `odoo.conf` file in your Dockerfile:

```dockerfile

# Create a dummy odoo.conf

RUN echo "[options]" > /etc/odoo/odoo.conf && \

echo "db\_host = db" >> /etc/odoo/odoo.conf && \

echo "db\_port = 5432" >> /etc/odoo/odoo.conf && \

echo "db\_user = odoo" >> /etc/odoo/odoo.conf && \

echo "db\_password = odoo" >> /etc/odoo/odoo.conf && \

echo "db\_name = dummy\_database" >> /etc/odoo/odoo.conf

```

Here, `db\_host` is set to `db`, which is a reference to the PostgreSQL service container hostname in Docker Compose or Kubernetes. Adjust `db\_port`, `db\_user`, `db\_password`, and `db\_name` as per your setup.

3. \*\*Delay Database Initialization:\*\*

To delay database initialization, you can modify the Odoo entrypoint script. By default, the Odoo entrypoint script (`/entrypoint.sh`) initializes the database if it doesn't exist. You can override this behavior by modifying the entrypoint script to skip database initialization until you're ready.

```dockerfile

# Copy modified entrypoint script

COPY ./entrypoint.sh /entrypoint.sh

# Make entrypoint script executable

RUN chmod +x /entrypoint.sh

```

In your `entrypoint.sh`, you can comment out or modify the database initialization step to delay it until the actual database is created:

```bash

#!/bin/bash

set -e

# Delay database initialization - comment out or modify as needed

# /entrypoint\_original.sh "$@"

exec "$@"

```

Ensure that you modify the `entrypoint.sh` script based on your specific requirements and the structure of the original Odoo entrypoint script (`entrypoint\_original.sh`).

4. \*\*Build Docker Image:\*\*

Finally, build your Docker image:

```bash

docker build -t my-odoo-image .

```

5. \*\*Run Docker Container:\*\*

Run your Odoo Docker container:

```bash

docker run -d --name my-odoo-container -p 8069:8069 my-odoo-image

```

6. \*\*Create Database Later:\*\*

After the Odoo container is up and running, you can create the actual database using Odoo’s interface or command-line tools. Update the `odoo.conf` file or Odoo database settings to point to the newly created database.

### Notes:

- \*\*Database Creation:\*\* Odoo itself doesn't provide a built-in mechanism to create databases. You typically use PostgreSQL commands (`createdb`) or database management tools to create databases.

- \*\*Security Considerations:\*\* Ensure you manage database credentials securely, especially in production environments.

By following these steps, you can configure your Odoo Docker image to start without an initial database connection and delay database initialization until a later time when you're ready to create and configure the database.

To remove PostgreSQL from your Ubuntu 24.04 VPS, you can follow these steps. This process involves uninstalling PostgreSQL packages and removing associated files and configurations. Be cautious, as this will remove PostgreSQL and its data if not backed up.

### 1. Stop PostgreSQL Service

First, stop the PostgreSQL service to ensure no processes are using it:

```bash

sudo systemctl stop postgresql

```

### 2. Remove PostgreSQL Packages

You can remove PostgreSQL packages using the `apt` command. The exact packages might vary depending on your installation, but typically you’ll want to remove the main PostgreSQL packages and any additional modules or extensions.

To remove PostgreSQL along with related packages, use:

```bash

sudo apt purge postgresql\\* -y

```

The `\\*` wildcard ensures that all PostgreSQL-related packages are included.

### 3. Remove PostgreSQL User and Group

To clean up PostgreSQL-specific users and groups, you can remove them with:

```bash

sudo deluser postgres

sudo delgroup postgres

```

### 4. Remove PostgreSQL Data Directory

To remove PostgreSQL data directories and other residual files:

```bash

sudo rm -rf /var/lib/postgresql/

sudo rm -rf /var/log/postgresql/

sudo rm -rf /etc/postgresql/

```

### 5. Remove Configuration Files and Dependencies

If you want to remove any configuration files and dependencies that were installed with PostgreSQL, you can use the following command:

```bash

sudo apt autoremove --purge -y

```

### 6. Update Package List

Finally, update your package list to ensure everything is cleaned up:

```bash

sudo apt update

```

### Verification

To verify that PostgreSQL has been removed, you can check if the `psql` command still exists:

```bash

which psql

```

If it’s no longer installed, you’ll get no output or an error indicating that `psql` isn’t found.

---

That’s it! PostgreSQL should now be completely removed from your Ubuntu 24.04 VPS. If you have any specific configurations or additional PostgreSQL-related packages, make sure to remove those as well.

If you've set both the Odoo and PostgreSQL containers to use the `nginx-proxy-manager` default network, it's important to ensure that:

1. \*\*Containers Can Communicate\*\*: Both containers should be able to communicate with each other over the network.

2. \*\*Correct Hostnames\*\*: Your application should use the correct hostname to connect to PostgreSQL.

Here's how you can ensure everything is set up correctly:

### 1. \*\*Check Network Configuration\*\*

Confirm that both the Odoo and PostgreSQL containers are on the same network. You can list the networks and inspect them with:

```bash

docker network ls

docker network inspect nginx-proxy-manager\_default

```

Make sure both containers are listed under the `nginx-proxy-manager\_default` network.

### 2. \*\*Verify Container Names and Hostnames\*\*

Inside a Docker network, containers can communicate using their service names as hostnames. So, if your PostgreSQL container is named `postgresql`, the hostname you should use in your Odoo container’s configuration to connect to PostgreSQL is `postgresql`.

To check the names of your running containers, use:

```bash

docker ps

```

### 3. \*\*Check Container Logs\*\*

Ensure PostgreSQL is running correctly and accepting connections. Use the following commands to check the logs of both containers:

```bash

docker logs <postgresql\_container\_id\_or\_name>

docker logs <odoo\_container\_id\_or\_name>

```

Look for any errors related to PostgreSQL in the logs of both containers.

### 4. \*\*Configuration in Odoo\*\*

Ensure that the Odoo container is configured to connect to the PostgreSQL container using the correct hostname and port. In Odoo's configuration file or environment variables, it should look something like this:

```bash

db\_host = postgresql

db\_port = 5432

db\_user = yourusername

db\_password = yourpassword

```

Alternatively, if using environment variables in Docker Compose, it might look like:

```yaml

version: '3'

services:

odoo:

image: odoo:16

environment:

- HOST=postgresql

- PORT=5432

- USER=yourusername

- PASSWORD=yourpassword

networks:

- nginx-proxy-manager\_default

postgresql:

image: postgres:latest

environment:

POSTGRES\_DB: yourdbname

POSTGRES\_USER: yourusername

POSTGRES\_PASSWORD: yourpassword

networks:

- nginx-proxy-manager\_default

networks:

nginx-proxy-manager\_default:

external: true

```

### 5. \*\*Network Connectivity Test\*\*

Exec into the Odoo container and test the network connectivity to the PostgreSQL container:

```bash

docker exec -it <odoo\_container\_id\_or\_name> bash

```

Inside the container, try to ping the PostgreSQL container:

```bash

ping postgresql

```

You can also test the connection using `nc` (Netcat) or `telnet` to see if the PostgreSQL port is reachable:

```bash

nc -zv postgresql 5432

```

or

```bash

telnet postgresql 5432

```

### 6. \*\*Update PostgreSQL Configuration (If Needed)\*\*

Ensure that PostgreSQL is configured to accept connections from other containers on the network. This usually involves setting `listen\_addresses` in `postgresql.conf` to `'\*'` (allowing connections from any IP) and ensuring the `pg\_hba.conf` file allows connections from the Docker network.

```bash

listen\_addresses = '\*'

```

And in `pg\_hba.conf`:

```bash

host all all 0.0.0.0/0 md5

```

After making changes, restart the PostgreSQL container.

### Summary

- \*\*Verify both containers are on the same Docker network.\*\*

- \*\*Use the correct hostname (`postgresql` or whatever your container is named) in Odoo’s configuration.\*\*

- \*\*Check container logs for errors.\*\*

- \*\*Test network connectivity between containers.\*\*

- \*\*Ensure PostgreSQL configuration allows connections from the Docker network.\*\*

If issues persist, provide specific error messages or configurations for more detailed troubleshooting.

From the logs you’ve provided, PostgreSQL seems to be starting up correctly, listening on the expected IP addresses and ports, and is ready to accept connections. Here’s a breakdown of the logs and what they indicate:

- \*\*Database Initialization\*\*: `PostgreSQL Database directory appears to contain a database; Skipping initialization` means PostgreSQL detected an existing database and did not perform a fresh initialization.

- \*\*Listening Addresses\*\*: `listening on IPv4 address "0.0.0.0", port 5432` and `listening on IPv6 address "::", port 5432` show that PostgreSQL is configured to listen for connections on both IPv4 and IPv6 on port 5432.

- \*\*Unix Socket\*\*: `listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"` indicates that PostgreSQL is also listening on the Unix socket, which is often used for local connections.

- \*\*Recovery Log\*\*: `database system was not properly shut down; automatic recovery in progress` suggests that PostgreSQL had to recover from an improper shutdown, but it completed the recovery successfully.

- \*\*Ready to Accept Connections\*\*: `database system is ready to accept connections` means the server is up and running.

### Troubleshooting Steps

Given that PostgreSQL seems operational but you're facing connection issues, follow these steps:

#### 1. \*\*Verify Odoo’s Connection Configuration\*\*

Ensure that Odoo is configured with the correct connection details for PostgreSQL. Specifically:

- \*\*Host\*\*: Should be set to the name of the PostgreSQL container if using Docker Compose, or the hostname of the PostgreSQL server if connecting across networks.

- \*\*Port\*\*: Default is 5432.

- \*\*Database Name, User, and Password\*\*: Should match the values you configured for PostgreSQL.

Example connection settings in Odoo might look like:

```bash

db\_host = postgresql

db\_port = 5432

db\_user = yourusername

db\_password = yourpassword

```

Or in Docker Compose:

```yaml

environment:

- HOST=postgresql

- PORT=5432

- USER=yourusername

- PASSWORD=yourpassword

```

#### 2. \*\*Check Docker Networking\*\*

Ensure that the Odoo container can reach the PostgreSQL container. If both containers are on the same network, use the container name as the hostname.

Check network connectivity:

```bash

docker network inspect nginx-proxy-manager\_default

```

Both containers should appear in the list of connected containers.

#### 3. \*\*Test Connectivity from Odoo Container\*\*

Access the Odoo container and test if it can connect to the PostgreSQL container:

```bash

docker exec -it <odoo\_container\_id\_or\_name> bash

```

Inside the Odoo container, try to test connectivity using `nc` or `telnet`:

```bash

nc -zv postgresql 5432

```

or

```bash

telnet postgresql 5432

```

You should receive a message indicating that the connection is successful.

#### 4. \*\*Check PostgreSQL Logs\*\*

Look for any errors in the PostgreSQL logs that might indicate why connections are not being accepted:

```bash

docker logs <postgresql\_container\_id\_or\_name>

```

#### 5. \*\*Verify Database Permissions\*\*

Ensure that the PostgreSQL user has the necessary permissions to access the database. This can be checked within the PostgreSQL container:

```bash

docker exec -it <postgresql\_container\_id\_or\_name> psql -U postgres

```

Run SQL commands to verify users and permissions:

```sql

\du

\l

```

#### 6. \*\*Check Firewall and Security Groups\*\*

If your containers are running on a cloud provider or have specific firewall rules, ensure that there are no rules blocking port 5432.

### Summary

- \*\*Verify\*\* Odoo’s connection configuration for PostgreSQL.

- \*\*Ensure\*\* both containers are on the same network and can communicate.

- \*\*Test\*\* network connectivity between containers.

- \*\*Check\*\* PostgreSQL logs for any errors.

- \*\*Ensure\*\* PostgreSQL permissions are correctly configured.

- \*\*Verify\*\* no firewall rules are blocking the connection.

If you’re still experiencing issues, please provide any specific error messages or configuration details for further assistance.