

## Understanding Containers vs Virtual Machines

### What is a Virtual Machine?

A Virtual Machine (VM) is a complete computer system that runs on top of your physical hardware. Think of it as a computer inside your computer.

Components of a VM:

- Full operating system (e.g., Ubuntu, Windows)
- Virtual hardware (CPU, RAM, disk)
- Hypervisor (software that manages VMs)
- Guest OS kernel

Example: If you run Windows and want to test your app on Linux, you'd install a VM with the entire Linux OS.

### What is a Container?

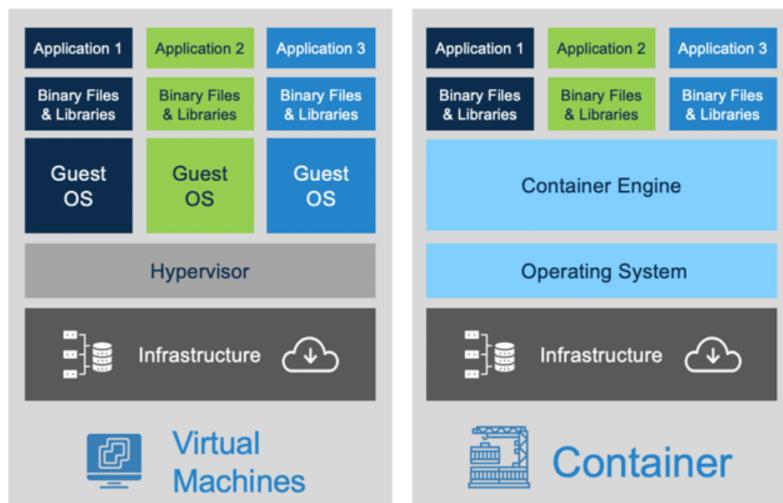
A Container is a lightweight, standalone package that includes everything needed to run your application (code, runtime, libraries), but shares the host OS kernel.

Components of a Container:

- Your application code
- Application dependencies
- Minimal OS utilities
- Shares host OS kernel

### Key Differences

Feature	Virtual Machine	Container
Size	GBs (includes full OS)	MBs (shares OS kernel)
Startup Time	Minutes	Seconds
Resource Usage	Heavy (needs dedicated resources)	Light (shares resources)
Isolation	Complete isolation	Process-level isolation
Portability	Less portable	Highly portable
Use Case	Running different OS	Running applications consistently



## Why Containers for Development?

**Problem:** "It works on my machine" syndrome  
Developer A uses Windows with Python 3.11  
Developer B uses Mac with Python 3.9  
Production server uses Linux with Python 3.10

**Solution:** Containers ensure everyone runs the exact same environment.

### Implementation in your existing Django Project

---

#### I. Install Docker

Step 1: Create a Docker Hub Account

Docker Hub is like GitHub for Docker images - it's where you can store and share your Docker images.

##### a. Creating Your Account

Go to: <https://hub.docker.com/signup>

##### b. Fill in Your Information

Docker ID: Choose a username (e.g., john\_doe, student123)

- This will be used to tag your images*
- Choose carefully - it's hard to change later*
- Avoid spaces or special characters*

Email: Your email address

Password: Create a strong password

##### c. Verify Your Email

- Check your email inbox
- Click the verification link

Step 2: Install Docker on Windows

##### a. Enable Virtualization

Check if virtualization is enabled:

1. Press Ctrl + Shift + Esc (Task Manager)
2. Click "Performance" tab
3. Click "CPU"
4. Look for "Virtualization: Enabled"

If disabled, enable it in BIOS:

1. Restart computer
2. Press F2, F10, Delete, or Esc during startup (varies by manufacturer)
3. Find "Virtualization Technology" or "Intel VT-x" or "AMD-V"
4. Enable it

5. Save and exit
- b. Install WSL 2 (Windows Subsystem for Linux)

Open PowerShell as Administrator and run: \$ wsl –install  
Restart your computer.  
After restart, set WSL 2 as default: \$ wsl --set-default-version 2

#### Step 3: Download Docker Desktop

- a. Visit Docker's website: <https://www.docker.com/products/docker-desktop>

Click "Download for Windows"

#### Step 4: Install Docker Desktop

- Run the installer
- Configuration options:

Check "Use WSL 2 instead of Hyper-V" (recommended)

#### Step 5: Verify Installation

Open **Command Prompt or PowerShell**: \$ docker –version

## II. Project Structure

```
my_django_project/
├── manage.py
└── myproject/
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── myapp/
    ├── models.py
    ├── views.py
    └── ...
└── requirements.txt
└── Dockerfile
└── docker-compose.yml
└── .dockerignore
```

## III. Create Required Files

```
$ pip freeze > requirements.txt
```

## IV. Building Docker Images for Django

### Core Concepts

Docker Image: A blueprint/template for your application (like a recipe)  
Docker Container: A running instance of an image (like a cake made from the recipe)

### Dockerfile

It contains step-by-step instructions that Docker follows to create your application image. So, everyone who builds from the same Dockerfile gets the exact same image, eliminating "it works on my machine" problems. Instead of manually installing Python, dependencies, and configuring the environment, the Dockerfile does it all automatically.

Create a file named Dockerfile in your project root:

```
FROM python:3.11-slim

ENV PYTHONDONTWRITEBYTECODE=1
ENV PYTHONUNBUFFERED=1

WORKDIR /app

COPY requirements.txt /app/
RUN pip install --upgrade pip && pip install --no-cache-dir -r requirements.txt

COPY . /app/

EXPOSE 8000

CMD ["sh", "-c", "python manage.py migrate && python manage.py runserver 0.0.0.0:8000"]
```

## Dockerignore

Purpose: Tells Docker which files to ignore (like `.`gitignore` for Git)

Create a file named .dockerignore in your project root:

```
*.pyc
__pycache__/
db.sqlite3
*.sqlite3
.env
.git
.gitignore
venv/
env/
```

## Compose YAML

Purpose: Used to define and run multi-container applications and simplify how you manage containers.

Create a file named docker-compose.yml in your project root:

```
version: "3.8"

services:
  web:
    build: .
    image: my-django-app:latest
    command: sh -c "python manage.py migrate && python manage.py runserver 0.0.0.0:8000"
    ports:
      - "8000:8000"
    volumes:
      - .:/app
```

environment:

- DEBUG=1
- USE\_SQLITE=true
- DJANGO\_ALLOWED\_HOSTS=localhost 127.0.0.1 [::1]

The volume mount `./app` ensures the SQLite database file (`db.sqlite3`) is saved on your host machine in your project directory, not inside the container.

## V. Building Your Image

```
$ docker build -t my-django-app:latest .
```

Breakdown:

- `docker build`: Command to build an image
- `-t my-django-app:latest`: Tag (name) the image
- `my-django-app`: image name
- `latest`: version tag

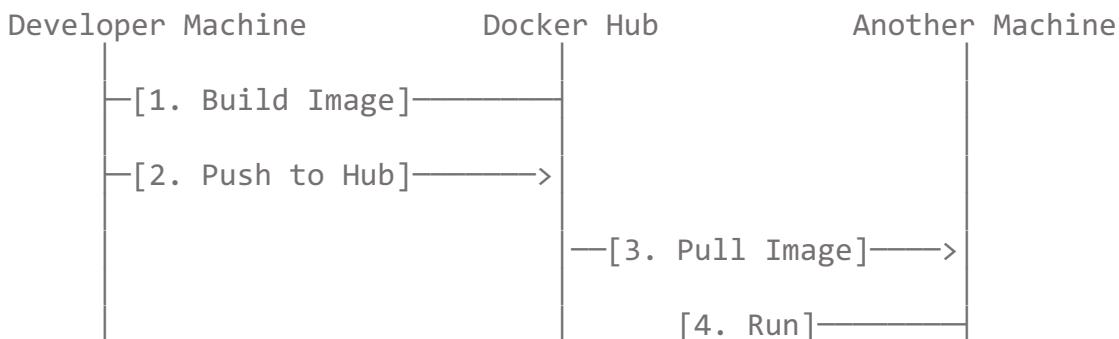
What Happens During Build?

1. Docker reads your Dockerfile
2. Downloads base image (`python:3.11-slim`)
3. Executes each instruction step-by-step
4. Creates layers (each instruction = one layer)
5. Final result: A complete image ready to run

Command	Meaning
<code>docker images</code>	Lists all images currently available on your local machine.
<code>docker rmi my-django-app:latest</code>	Removes the specified image ( <code>my-django-app:latest</code> ) from your local system.
<code>docker history my-django-app:latest</code>	Displays the build history (layers) of the specified image.
<code>docker build --no-cache -t my-django-app:latest .</code>	Builds a new Docker image without using the cache, ensuring a fresh build with the tag <code>my-django-app:latest</code> .

## VI. Deploying Your Docker Image to Another Machine

One of Docker's biggest advantages is portability - "build once, run anywhere". This section discusses how to share their Django application with others.



1: Using Docker Hub (Recommended - Like GitHub for Docker Images)

Docker Hub is a free registry where you can store and share Docker images.

### Step 1: Login to Docker Hub from Terminal

On your development machine (where you built the image): \$ docker login

Enter your Docker Hub username and password

You should see: "Login Succeeded"

### Step 2: Tag Your Image

Images must be tagged with your Docker Hub username:

Format: docker tag <current-image-name> <dockerhub-username>/<image-name>:<version>

\$ docker tag my-django-app:latest yourusername/django-app:latest

Example:

docker tag my-django-app:latest reid00/django-app:latest

Understanding Tags:

- **yourusername/** - Your Docker Hub username
- **django-app** - Name of your application
- **latest** - Version tag (can be v1.0, dev, etc.)

### Step 3: Push Image to Docker Hub

\$ docker push yourusername/django-app:latest

You'll see upload progress:

The push refers to repository [docker.io/yourusername/django-app]  
5f70bf18a086: Pushed  
latest: digest: sha256:abc123... size: 1234

### Step 4: Verify Upload

Visit: <https://hub.docker.com/r/yourusername/django-app>

You should see your image listed!

## VII. Installing on Another Machine

Now anyone (or you on a different computer) can run your application!

Prerequisites on Target Machine

Check if Docker is installed: \$ docker --version

If not installed, install Docker:

- Windows/Mac: Download Docker Desktop from docker.com
- Linux: sudo apt-get install docker.io

### Step 1: Pull the Image

On the target machine: \$ docker pull yourusername/django-app:latest

Docker downloads all layers:

```
latest: Pulling from yourusername/django-app
abc123: Pull complete
def456: Pull complete
Status: Downloaded newer image for yourusername/django-app:latest
```

## Step 2: Prepare Configuration Files

Create a project folder and add these files:

docker-compose.yml (SQLite version):

```
version: '3.8'

services:
  web:
    image: yourusername/django-app:latest # Note: 'image' not 'build'
    command: sh -c "python manage.py migrate && python manage.py runserver 0.0.0.0:8000"
    ports:
      - "8000:8000"
    volumes:
      - django_data:/app
    environment:
      - DEBUG=1
      - USE_SQLITE=true
      - DJANGO_ALLOWED_HOSTS=localhost 127.0.0.1 [::1]

volumes:
  django_data:
```

Key Difference: Using image: instead of build: because we're pulling from Docker Hub, not building locally.

## Step 3: Run the Application

- Start the application: \$ docker-compose up -d

Expected output:

```
Creating network "psuspHERE_default" with the default driver
Creating psuspHERE_web_1 ... done
Attaching to psuspHERE_web_1
web_1  | Performing system checks...
web_1  | System check identified no issues (0 silenced).
web_1  | Starting development server at http://0.0.0.0:8000/
```

- Check if running: \$ docker-compose ps
- View logs: \$ docker-compose logs -f web
- Access the app: Open browser: <http://localhost:8000>
- Step 4: Initial Setup (First Time Only)

Create superuser:

```
$ docker-compose exec web python manage.py createsuperuser
```

Run any additional commands

```
$ docker-compose exec web python manage.py collectstatic --noinput
```

## Updating the Application

When you make changes to your app:

### A. On Development Machine

1. Make code changes

2. Rebuild image

```
$ docker-compose build
```

3. Tag with new version

```
$ docker tag my-django-app:latest myusername/django-app:v1.1
```

4. Push new version

```
$ docker push myusername/django-app:v1.1
```

```
$ docker push myusername/django-app:latest
```

### B. On Target Machine

1. Pull new version

```
$ docker-compose pull
```

2. Restart with new image

```
$ docker-compose down
```

```
$ docker-compose up -d
```

3. Run migrations if database changed

```
$ docker-compose exec web python manage.py migrate
```

## Practice:

Apply Docker concepts to containerize your PSUSphere project. Create a Docker image and docker-compose configuration that allows anyone to run your project with a single command.

Submit the GitHub Repository URL containing:

- a. Dockerfile
- b. docker-compose.yml (this is for building image)
- c. requirements.txt
- d. .dockerignore
- e. /for\_client/docker-compose.yml (this is for pulling from DockerHub)
- f. README.md – add instructions to your project

Do not submit the PSUSphere GitHub Repository URL, prepare a separate repository for this activity.