Docker Core Concept (quick view):-

Concept	What it is	Analogy
Dockerfile	Instructions to build	Recipe
Docker Image	Built package/template	Packed food
Docker Container	Running instance	Eating the food

1. Dockerfile (the instructions)

A **Dockerfile** is a **text file** that tells Docker how to build an image.

** It contains:**

- Base image (`FROM`)
- Dependencies (`RUN`)
- Files (`COPY`)
- Startup command (`CMD`)

Example:

```
FROM node:18
WORKDIR /app
COPY . .
RUN npm install
CMD ["npm", "start"]
```

 Think:

“How should my app environment be created?”

---

2. Docker Image (the built package)

A **Docker Image** is the **output of a Dockerfile**.

It is:

- Read-only
- Reusable
- Portable

□ It includes:

- Your code
- Dependencies
- Runtime

Example:

```
docker build -t myapp .
```

□ Think:

"This is a ready-to-use app snapshot"

---

3. Docker Container (the running app)

A **Docker Container** is a **running instance of an image**.

It is:

- Live
- Isolated
- Executable

Example:

```
docker run myapp
```

□ Think:

"This is my app actually running"

How they connect

```
Dockerfile → docker build → Image → docker run → Container
```

Let's Go with details

What is a Dockerfile?

A **Dockerfile** is a **text file with instructions** that tells Docker:

 "How to build an image for my application"

What it actually does

It defines:

- Base environment (Python, Node, etc.)
- Dependencies
- Project files
- Commands to run your app

Dockerfile Example (for Django):-

```
FROM python:3.10

WORKDIR /app

COPY . .

RUN pip install -r requirements.txt

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Line-by-line explanation

FROM

```
FROM python:3.10
```

- Base image
- Like choosing an OS + runtime

📁 WORKDIR

```
WORKDIR /app
```

- Sets working directory inside container

📋 COPY

```
COPY . .
```

- Copies your project files into container

📦 RUN

```
RUN pip install -r requirements.txt
```

- Installs dependencies
- Runs during **image build**

▶️ CMD

```
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

- Command to run when container starts

How Dockerfile fits into your setup

🐳 Docker:

1. Looks for `Dockerfile` in `./backend`
2. Executes instructions
3. Creates an **image**

4. Runs it as a **container**

Final one-line definition

A **Dockerfile** is a script that defines how to build a Docker image for your application.

Docker Image (the blueprint)

A **Docker image** is a *read-only template* used to create containers.

It contains:

- Your application code
- Runtime (e.g., Python, Node.js)
- Libraries and dependencies
- Environment setup

 You can think of it like a **class in programming** or a **snapshot of a system**.

Example:

- An image might say: “Run a Node.js app with Express installed”

You typically create images using a **Dockerfile**.

? Real-world flow

```
docker build -t myapp .
docker run myapp
```

□ Docker Container (the running instance)

A **Docker container** is a *running instance of a Docker image*.

It is:

- Lightweight
- Isolated
- Executable

□ If an image is a *blueprint*, a container is the *actual house built from it*.

Key points:

- You can run multiple containers from the same image
- Each container has its own filesystem, network, and processes
- Containers can be started, stopped, deleted easily

□ Docker Networking (how containers talk)

Docker networking allows containers to communicate:

- With each other
- With the host machine
- With the internet

Common network types:

- **Bridge (default)**
  Containers on the same bridge can talk via IP or container name

- **Host**
  Container shares the host's network (no isolation)
- **None**
  No network access
- **Custom networks**
  Best practice for apps—containers can refer to each other by name

 Example:

- A backend container connects to a database container via:

  `db:5432`

Docker Compose (multi-container orchestration)

Docker Compose is a tool to define and run *multiple containers together*.

Instead of running many commands, you define everything in a single file:

- `docker-compose.yml`

It lets you:

- Define services (app, database, cache, etc.)
- Configure networks and volumes
- Start everything with one command

 Think of it as a **project manager for containers**.

Example :-

docker-compose.yml file inside root folder

Backend Service (Django)

```
backend:
  build: ./backend
  container_name: django_backend
```

```
  ports:
    - "8000:8000"
  volumes:
    - ./backend:/app
  command: python manage.py runserver 0.0.0.0:8000
```

Frontend Service (Vite React)

```
frontend:
  build: ./frontend
  container_name: react_frontend
  ports:
    - "5173:5173"
  volumes:
    - ./frontend:/app
    - /app/node_modules
  depends_on:
    - backend
  command: npm run dev -- --host
```

Run everything:

```
docker-compose up
```

Networking (implicit but important)

Even though not written, Docker Compose creates a network like:

```
project_default
```

Inside that network:

- `backend` → reachable as `backend`
- `frontend` → reachable as `frontend`

 That's why:

```
http://backend:8000
```

works **inside containers only**

□ Simple Analogy (ties it all together)

- **Docker Image** → Recipe □
- **Docker Container** → Cooked dish □
- **Docker Networking** → Table where dishes interact □
- **Docker Compose** → Full restaurant system managing everything □