

## Docker Essentials Tutorial

### What is a Container?

A container is an isolated process that packages your application with everything it needs to run. Unlike virtual machines, containers share the host system's kernel, making them lightweight and efficient.

#### Key benefits:

- **Self-contained:** Includes all dependencies
- **Isolated:** Minimal impact on host and other containers
- **Independent:** Each container manages separately
- **Portable:** Runs consistently anywhere

#### Quick start:

```
docker run -d -p 8080:80 --name welcome-to-docker docker/welcome-to-docker
```

Visit <http://localhost:8080> to see your running container.

### Understanding Images

Images are immutable templates that define how to build containers. They consist of layered filesystem changes and include all files, binaries, libraries, and configurations needed.

#### Key principles:

- Images cannot be modified once created
- Built as a series of layers
- Can extend existing images by adding new layers

**Finding images:** Docker Hub hosts over 100,000 pre-built images. Search for official images like redis, nginx, or node to get started.

### Docker Compose for Multi-Container Apps

Docker Compose simplifies running applications with multiple components. Instead of managing multiple docker run commands, define everything in a single YAML file.

#### Example compose.yaml:

services:

```
web:
  build: .
  ports:
    - "3000:3000"
```

database:

```
image: postgres
environment:
  POSTGRES_PASSWORD: example
```

#### Commands:

```
docker compose up    # Start all services
```

```
docker compose down  # Stop and remove everything
```

### Publishing Ports

Containers are isolated by default. To access services running inside, publish ports using the -p flag:

**Syntax:** HOST\_PORT:CONTAINER\_PORT

#### Examples:

*# Publish container port 80 to host port 8080*

```
docker run -p 8080:80 nginx
```

*# Let Docker choose an ephemeral port*

```
docker run -p 80 nginx
```

*# Publish all exposed ports*

```
docker run -P nginx
```

### **Overriding Container Defaults**

Customize container behavior using docker run flags:

#### **Common overrides:**

*# Port mapping*

```
docker run -p 5433:5432 postgres
```

*# Environment variables*

```
docker run -e POSTGRES_PASSWORD=mysecret postgres
```

*# Resource limits*

```
docker run --memory=512m --cpus=0.5 postgres
```

*# Custom network*

```
docker network create mynetwork
```

```
docker run --network=mynetwork postgres
```

### **Persisting Data with Volumes**

Containers lose their data when removed. Volumes solve this by storing data outside the container lifecycle.

#### **Creating and using volumes:**

*# Create volume*

```
docker volume create postgres_data
```

*# Use volume*

```
docker run -v postgres_data:/var/lib/postgresql/data postgres
```

*# View volumes*

```
docker volume ls
```

```
docker volume rm postgres_data
```

### **Sharing Local Files**

Bind mounts let you share files between host and container, perfect for development:

#### **Examples:**

*# Mount current directory to container*

```
docker run -v $(pwd):/usr/local/apache2/htdocs/ httpd
```

*# Using --mount (recommended)*

```
docker run --mount type=bind,source=$(pwd),target=/usr/local/apache2/htdocs/ httpd
```

## Multi-Container Applications

Real applications often need multiple services working together. While you could use multiple docker run commands, Docker Compose provides a better approach.

### Benefits of Compose:

- Single configuration file
- Automatic networking between services
- Easy scaling and management
- Consistent environments

### Example workflow:

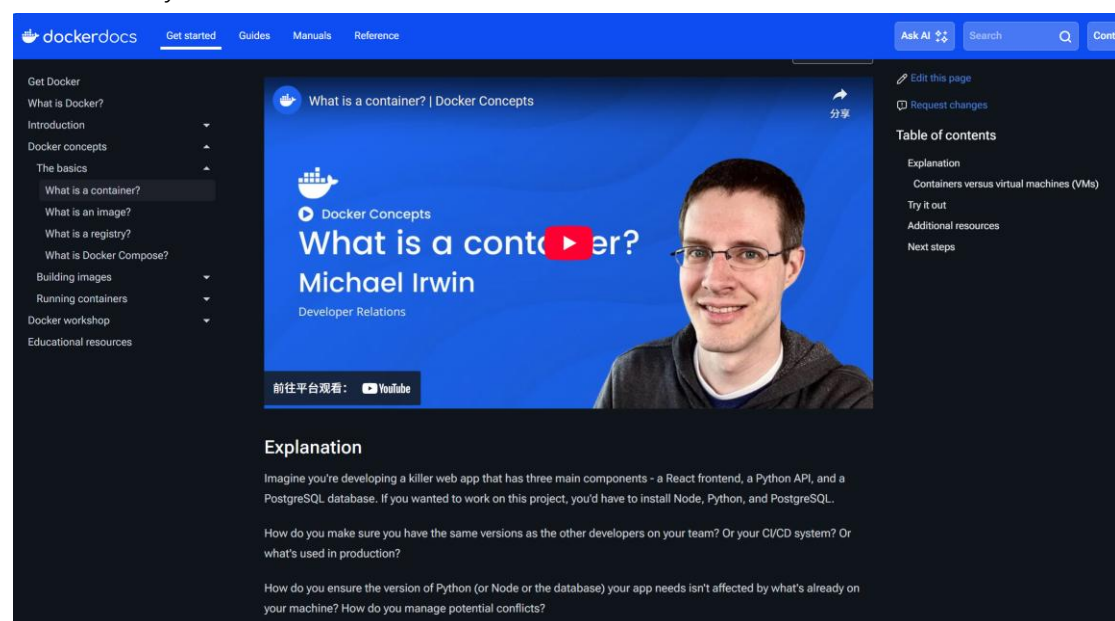
1. Define services in compose.yaml
2. Run docker compose up to start everything
3. Services automatically connect and can communicate
4. Use docker compose down to clean up

### Getting Started

1. **Install Docker Desktop** from [docker.com](https://docker.com)
2. **For Windows:** Enable WSL2 for better performance
3. **Verify installation:** Run docker run hello-world

### Next Steps

- Explore Docker Hub for pre-built images
- Build your own images with Dockerfiles
- Learn about Docker networking
- Study container orchestration with Kubernetes



The screenshot shows the Docker Docs website. The main content area features an article titled "What is a container? | Docker Concepts" by Michael Irwin, a Developer Relations member. The article includes a video player with a red play button and a link to "前往平台观看: YouTube". The left sidebar contains a navigation menu with categories like "Get Docker", "What is Docker?", "Introduction", "Docker concepts", "The basics", "Building images", "Running containers", "Docker workshop", and "Educational resources". The right sidebar has a "Table of contents" section with links to "Explanation", "Containers versus virtual machines (VMs)", "Try it out", "Additional resources", and "Next steps".

## Explanation

Seeing as a [container](#) is an isolated process, where does it get its files and configuration? How do you share those environments?

That's where container images come in. A container image is a standardized package that includes all of the files, binaries, libraries, and configurations to run a container.

For a [PostgreSQL](#) image, that image will package the database binaries, config files, and other dependencies. For a Python web app, it'll include the Python runtime, your app code, and all of its dependencies.

There are two important principles of images:

1. Images are immutable. Once an image is created, it can't be modified. You can only make a new image or add changes on top of it.
2. Container images are composed of layers. Each layer represents a set of file system changes that add, remove, or modify files.

These two principles let you to extend or add to existing images. For example, if you are building a Python app, you can start from the [Python image](#) and add additional layers to install your app's dependencies and add your code. This lets you focus on your app, rather than Python itself.

## Explanation

If you've been following the guides so far, you understand that containers provide isolated processes for each component of your application. Each component - a React frontend, a Python API, and a Postgres database - runs in its own sandbox environment, completely isolated from everything else on your host machine. This isolation is great for security and managing dependencies, but it also means you can't access them directly. For example, you can't access the web app in your browser.

That's where port publishing comes in.