Docker for Beginners: A Practical Guide to Containers

This beginner-friendly tutorial covers the essentials of containerization, helping you build, run, and manage containers with hands-on examples.

Feb 23, 2025 · 14 min read

Moez Ali
Data Scientist, Founder & Creator of PyCaret

TOPICS

Kubernetes

Docker

When I started using Docker, I quickly realized how powerful it was. Imagine setting up your development environment in minutes instead of hours or running applications across different machines without the classic "it works on my machine" problem.

Docker simplifies how we build, ship, and run applications by packaging them into lightweight, portable containers. Whether you're a developer, data scientist, or system administrator, mastering Docker can save you headaches and make your workflows more efficient.

In this tutorial, I'll walk you through the basics—installing Docker, understanding key concepts, and running your first containerized application. By the end, you'll not only know how Docker works but also have hands-on experience using it, setting a strong foundation for more advanced topics. Let's dive in!

What is Docker?

Docker is an open-source containerization platform that simplifies application deployment by packaging software and its dependencies into a standardized unit called a container. Unlike traditional virtual machines, Docker containers share the host OS kernel, making them more efficient and lightweight.

Containers ensure that an application runs the same way in development, testing, and production environments. This reduces compatibility issues and enhances portability across various platforms. Due to its flexibility and scalability, Docker has become a crucial tool in modern DevOps and cloud-native development workflows.

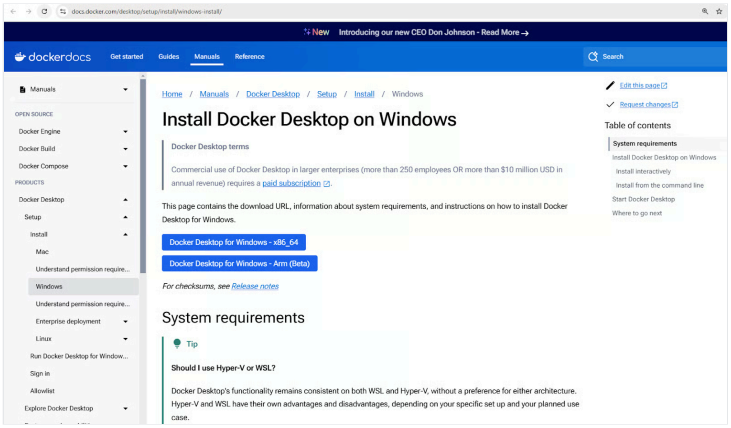
Docker official logo.

Installing Docker

Docker can be installed on various operating systems, including Windows, macOS, and Linux. While the core functionality remains the same across all platforms, the installation process differs slightly depending on the system. Below, you'll find step-by-step instructions for installing Docker on your preferred operating system.

Installing Docker on Windows

1. Download Docker Desktop for Windows.

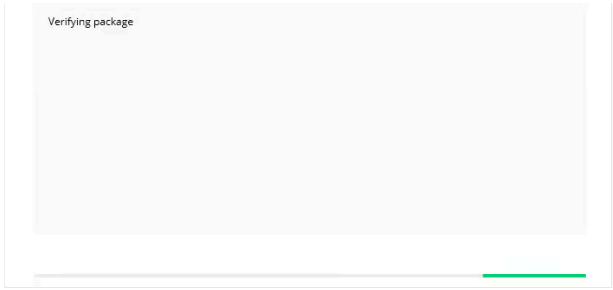


Download Docker Desktop Installer for Windows

2. Run the installer and follow the setup instructions.

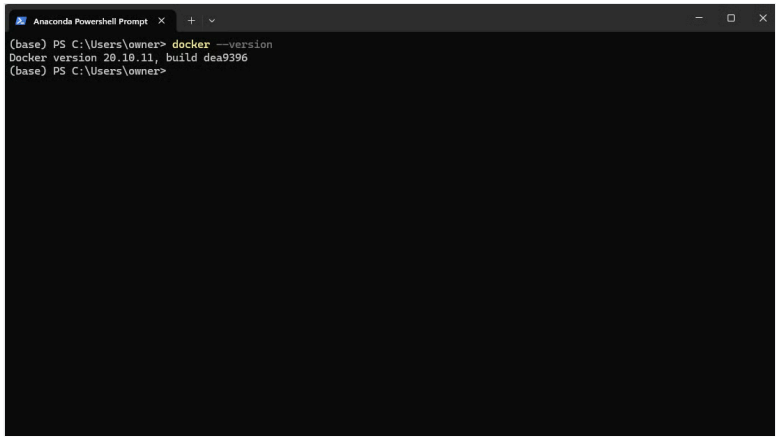
EN

Sale ends in
0d 20h 14m 52s



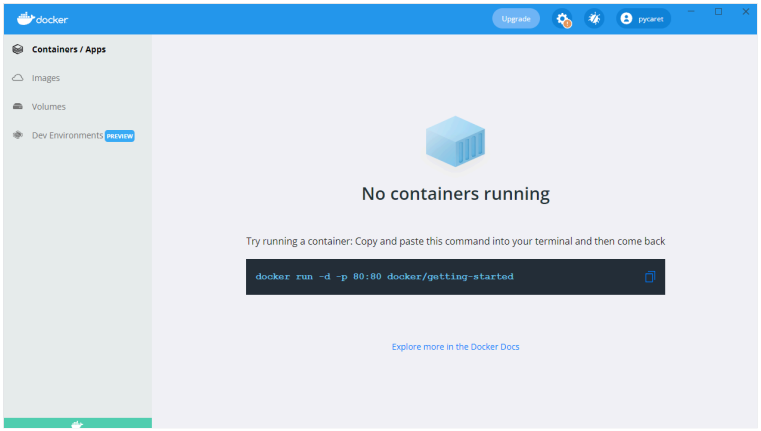
Installing Docker Desktop for Windows

3. Enable WSL 2 integration if prompted.
4. Verify installation by running `docker -version` in PowerShell.



Checking Docker version after installation through Powershell

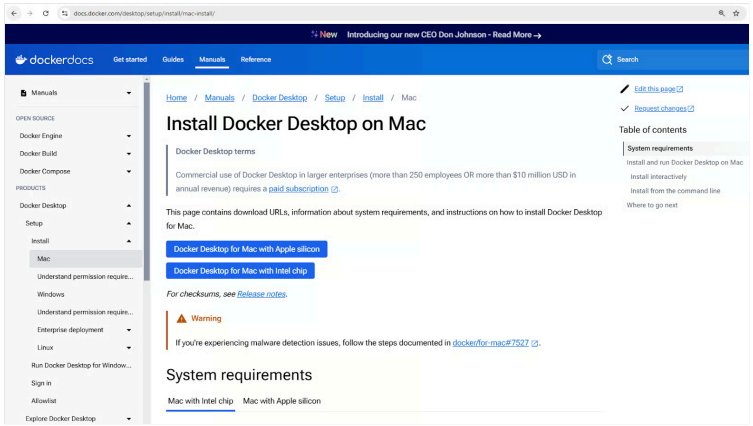
5. Start the Docker Desktop app from your run menu.



Launching Docker Desktop Application on Windows

Installing Docker on macOS

1. Download **Docker Desktop for Mac**.



Download Docker Desktop installer for Mac

2. Open the downloaded `.dmg` file and drag Docker to the Applications folder.
3. Launch Docker and complete the setup.
4. Verify installation using `docker -version` in the terminal.

Installing Docker on Linux (Ubuntu)

1. Update package lists: `sudo apt update`
2. Install dependencies: `sudo apt install apt-transport-https ca-certificates curl software-properties-common`
3. Add Docker's official GPG key: `curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -`
4. Add Docker's repository: `sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"`
5. Install Docker: `sudo apt install docker-ce`
6. Verify installation: `docker –version`

Master Docker and Kubernetes

Learn the power of Docker and Kubernetes with an interactive track to build and deploy applications in modern environments.

Start Track for Free

Basic Docker Concepts

Now that you have Docker installed, you might be eager to jump right in and start running containers. But before we do that, it's important to understand a few key concepts forming the foundation of Docker's work. These concepts will help you navigate Docker more effectively and avoid common beginner pitfalls.

At the heart of Docker are **images**, which serve as blueprints for containers; **containers**, which are the running instances of these images; and **Docker Hub**, a centralized repository for sharing and managing images.

Let's explore each of these concepts in more detail.

Docker images

Docker images are the fundamental building blocks of containers. They are immutable, read-only templates containing everything needed to run an application, including the operating system, application code, runtime, and dependencies.

Images are built using a Dockerfile , which defines the instructions for creating an image layer by layer.

Images can be stored in and retrieved from container registries such as Docker Hub.

Here are some example commands for working with images:

- `docker pull nginx` : Fetch the latest Nginx image from Docker Hub.
- `docker images` : List all available images on the local machine.
- `docker rmi nginx` : Remove an image from the local machine.

Docker containers

A Docker container is a running instance of a Docker image. Containers provide an isolated runtime environment where applications can run without interfering with each other or the host system.Each container has its own filesystem, networking, and process space but shares the host kernel.

Containers follow a simple lifecycle involving creation, starting, stopping, and deletion. Here's a breakdown of common container management commands:

1. Creating a container: `docker create` or `docker run`
2. Starting a container: `docker start`
3. Stopping a container: `docker stop`
4. Restarting a container: `docker restart`
5. Deleting a container: `docker rm`

Let's see a practical example. The following command runs an Nginx container in detached mode (running in the background), mapping port 80 inside the container to port 8080 on the host machine:

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

Explain code POWERED BY datalab

After running this command, Docker will pull the Nginx image (if not already available), create a container, and start it.

To check all running and stopped containers:

```
docker ps -a
```

Explain code POWERED BY datalab

This will display a list of all containers and details like their status and assigned ports.

Docker Hub

Docker Hub is a cloud-based registry service for finding, storing, and distributing container images. Users can push custom images to Docker Hub and share them publicly or privately.

Here are some commands for interacting with Docker Hub:

- `docker login` : Authenticate with Docker Hub.
- `docker push my-image` : Upload a custom-built image to Docker Hub.
- `docker search ubuntu` : Search for official and community images.
- `docker pull ubuntu` : Download an Ubuntu image from Docker Hub.

New to containerization? Get a solid foundation with the Containerization and Virtualization Concepts course.

Running Your First Docker Container

Now that we've covered Docker's core concepts, it's time to put them into action! Let's start by running our first container to ensure Docker is installed correctly and working as expected.

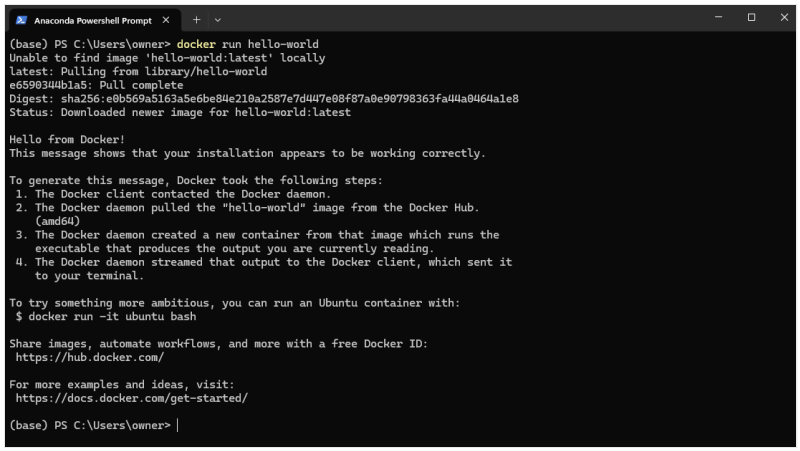
To test your Docker installation, open PowerShell (Windows) or Terminal (Mac and Linux) and run:

```
docker run hello-world
```

Explain code

POWERED BY datalab

This pulls the hello-world image from DockerHub and runs it in a container.



Docker hello-world image example

Now, let's go a step further and run a real-world application—an Nginx web server. Execute the following command:

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

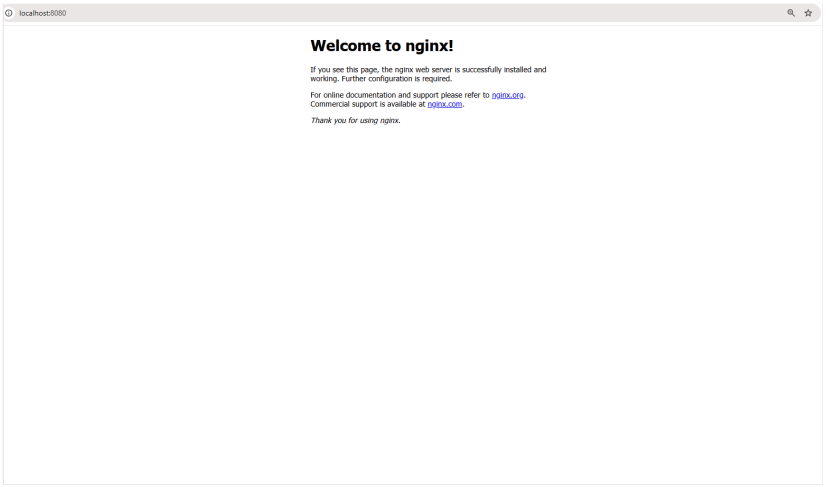
Explain code

POWERED BY datalab

The above command does the following:

- The `-d` flag runs the container in detached mode, meaning it runs in the background.
- The `-p 8080:80` flag maps port 80 inside the container to port 8080 on your local machine, allowing you to access the web server.

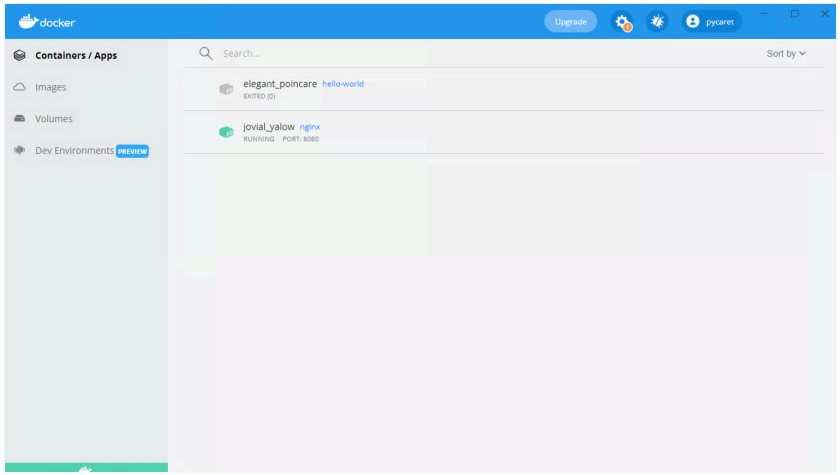
Once the command runs successfully, open a browser and visit: http://localhost:8080



Accessing web server at localhost:8080

You should see the default Nginx welcome page, confirming that your web server is running inside a container!

You will also see a container running in your Docker Desktop:



Nginx container running on port 8080

Building Your First Docker Image

So far, we've been running pre-built images from Docker Hub. But what if you need a custom environment tailored to your application? That's where building your own Docker image comes in.

Creating a Docker image involves writing a Dockerfile , a script that automates image-building. This ensures consistency and portability across different environments. Once an image is built, it can be run as a container to execute applications in an isolated environment.

In this section, we'll learn the fundamentals of writing a Dockerfile, building a custom image, and running it as a container.

Dockerfile basics

A Dockerfile is a script containing a series of instructions that define how a Docker image is built. It automates the image creation process, ensuring consistency across environments. Each instruction in a Dockerfile creates a new layer in the image. Here's a breakdown of an example Dockerfile for a simple Python Flask app:

```
# Base image containing Python runtime
FROM python:3.9

# Set the working directory inside the container
WORKDIR /app

# Copy the application files from the host to the container
COPY . /app

# Install the dependencies listed in requirements.txt
```

```
RUN pip install -r requirements.txt

# Define the command to run the Flask app when the container starts
CMD ["python", "app.py"]
```

Explain code POWERED BY datalab

In the above command:

- `-v my-volume:/app/data` mounts the `my-volume` storage to the `/app/data` directory inside the container.
- Any data stored in `/app/data` will persist even if the container stops or is removed.

Breaking down the Dockerfile above:

- `FROM python:3.9`: Specifies the base image with Python 3.9 pre-installed.
- `WORKDIR /app`: Sets `/app` as the working directory inside the container.
- `COPY . /app`: Copies all files from the host's current directory to `/app` in the container.
- `RUN pip install -r requirements.txt`: Installs all required dependencies inside the container.
- `CMD ["python", "app.py"]`: Defines the command to execute when the container starts.

Building and running the image

Once the Dockerfile is defined, you can build and run the image using the following commands:

Step 1: Build the image

```
docker build -t my-flask-app .
```

Explain code POWERED BY datalab

The above command:

- Uses the current directory (`.`) as the build context.
- Reads the `Dockerfile` and executes its instructions.
- Tags (`-t`) the resulting image as `my-flask-app`.

Step 2: Run the image as a container

```
docker run -d -p 5000:5000 my-flask-app
```

Explain code POWERED BY datalab

The above command:

- Runs the container in detached mode (`-d`).
- Maps port 5000 inside the container to port 5000 on the host (`-p 5000:5000`).

Once running, you can access the Flask application by navigating to http://localhost:5000 in a browser.

Docker Volumes and Persistence

By default, data inside a Docker container is **temporary**—once the container stops or is removed, the data disappears. To persist data across container restarts and share it between multiple containers, Docker provides volumes, a built-in mechanism for managing persistent storage efficiently.

Unlike storing data inside the container's filesystem, volumes are managed separately by Docker, making them more efficient, flexible, and easier to back up.

In the next section, we'll explore how to create and use Docker volumes to ensure data persistence in your containers.

Creating and using Docker volumes

Step 1: Create a volume

Before using a volume, we need to create one. Run the following command:

```
docker volume create my-volume
```

Explain code POWERED BY datalab

This creates a named volume called my-volume , which Docker will manage separately from any specific container.**Step 2: Use the volume in a container**

Now, let's start a container and mount the volume inside it:

```
docker run -d -v my-volume:/app/data my-app
```

Explain code POWERED BY datalab

In the above command:

- `-v my-volume:/app/data` mounts the `my-volume` storage to the `/app/data` directory inside the container.
- Any data stored in `/app/data` will persist even if the container stops or is removed.

Docker Compose for Multi-Container Applications

So far, we've been working with single-container applications, but many real-world applications require multiple containers to work together. For example, a web application might need a backend server, a database, and a caching layer—each running in its own container. Managing these containers manually with separate `docker run` commands can quickly become tedious.

That's where Docker Compose comes in.

What is Docker Compose?

Docker Compose is a tool that simplifies the management of multi-container applications. Instead of running multiple `docker run` commands, you can define an entire application stack using a `docker-compose.yml` file and deploy it with a single command.

Writing a Docker Compose file

Now, let's create a real-world example—a simple Node.js application that connects to a **MongoDB database**. Instead of managing the two containers separately, we'll define them in a `docker-compose.yml` file.

Here's how we define our multi-container setup in Docker Compose:

```
version: '3'
services:
  web:
    build: .
    ports:
      - "3000:3000"
    depends_on:
      - database
  database:
    image: mongo
    volumes:
      - db-data:/data/db
volumes:
  db-data:
```

Explain code

POWERED BY datalab

Breaking down the file above:

- `version: '3'` : Specifies the Docker Compose version.
- `services:` : Defines individual services (containers).
- `web:` : Defines the Node.js web application.
- `database:` : Defines the MongoDB database container.
- `volumes:` : Creates a named volume (`db-data`) for MongoDB data persistence.

Running multi-container applications

Once the docker-compose.yml file is ready, we can launch the entire application stack with a single command:

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

Explain code POWERED BY datalab

The previous command starts both the web and database containers in detached mode (`-d`).

To stop all services, use:

```
docker-compose down
```

Explain code POWERED BY datalab

This stops and removes all containers while preserving volumes and network settings.

Docker Networking Basics

So far, we've focused on running containers and managing storage, but what happens when containers need to communicate with each other? In most real-world applications, containers don't operate in isolation—they need to exchange data, whether a web server talks to a database or microservices interact with each other.

Docker provides a range of networking options to accommodate different use cases, from isolated internal networks to externally accessible configurations.

Ready to level up your Docker skills? Enroll in **Intermediate Docker** to explore multi-stage builds, advanced networking, and more!

What is Docker networking?

Docker networking is a built-in feature that allows containers to communicate with each other, whether on the same host or across multiple hosts in a distributed environment. It

provides network isolation, segmentation, and connectivity options suited for different deployment scenarios.

Docker supports multiple network types, each serving different use cases:

- Bridge (default): Containers on the same host communicate through an internal virtual network. Each container gets its private IP address within the bridge network, and they can reach each other via container names.
    - Example: `docker network create my-bridge-network`
    - Ideal for running multiple containers on a single host that need to communicate securely without exposing services externally.
- **Host**: Containers share the host's networking stack and directly use the host's IP address and ports.
    - Example: `docker run --network host nginx`
    - Useful when you need high performance and don't require network isolation, such as running monitoring agents or low-latency applications.
- **Overlay**: Enables container communication on different hosts by creating a distributed network.
    - Example: `docker network create --driver overlay my-overlay-network`
    - Designed for orchestrated deployments like Docker Swarm, where services span multiple nodes.
- **Macvlan**: Assigns a unique MAC address to each container, making it appear as a physical device on the network.
    - Example: `docker network create -d macvlan --subnet=192.168.1.0/24 my-macvlan`
    - Used when containers need direct network access, such as when integrating legacy systems or interacting with physical networks.

Running containers on custom networks

Let's walk through how to set up and use a **custom bridge network** for container communication.

Step 1: Create a custom network

Before running containers, we first need to create a dedicated network:

```
docker network create my-custom-network
```

Explain code POWERED BY datalab

This command creates an isolated network that containers can join for inter-container communication.

Step 2: Run containers on the network

Now, let's start two containers and connect them to our newly created network:

```
docker run -d --network my-custom-network --name app1 my-app
docker run -d --network my-custom-network --name app2 my-app
```

Explain code POWERED BY datalab

- The `--network my-custom-network` flag attaches the container to the specified network.
- The `--name` flag assigns a unique container name, making it easier to reference.

Both `app11` and `app2` can now communicate using their container names. You can test the connectivity using the `ping` command inside one of the containers:

```
docker exec -it app1 ping app2
```

Explain code POWERED BY datalab

If everything is set up correctly, you'll see a response confirming that the containers can communicate.

Inspecting Docker networks

To verify network configurations and connected containers, use:

```
docker network inspect my-custom-network
```

Explain code POWERED BY datalab

This command provides details about the network, including IP ranges, connected containers, and configurations.

Exposing and publishing ports

When running containers that need to be accessible externally, you can expose specific ports.

For example, to run an Nginx web server and expose it on port 8080 of your local machine, use:

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

Explain code

POWERED BY datalab

This maps port 80 inside the container to port 8080 on the host, making the service accessible via http://localhost:8080.

Best practices for Docker networking

- **Use custom networks**: Avoid using the default bridge network for production deployments to reduce unintended access between containers.
- **Leverage DNS-based discovery**: Instead of hardcoding IP addresses, use container names to enable dynamic service discovery.
- **Restrict external exposure**: Use firewalls or network policies to control service access.
- **Monitor traffic**: Use tools like `docker network inspect`, Wireshark, or Prometheus to analyze network traffic and detect anomalies.
- **Optimize overlay networks**: If deploying in a distributed setup, tune overlay networks for reduced latency by leveraging host-local routing options.

Docker Best Practices and Next Steps

Now that you've learned the fundamentals of Docker, it's time to level up your skills and adopt best practices that will help you build secure, efficient, and maintainable containerized applications.

The following best practices will help you streamline your Docker workflows and avoid common pitfalls.

- **Use official base images:** Always prefer official and well-maintained base images to ensure security and stability. Official images are optimized, regularly updated, and less likely to contain vulnerabilities.
- **Keep images small:** Reduce image size by choosing minimal base images (e.g., `python:3.9-slim` instead of `python:3.9`). Remove unnecessary dependencies and files to optimize storage and pull times.
- **Use multi-stage builds:** Optimize Dockerfiles by separating build and runtime dependencies. Multi-stage builds ensure that only the necessary artifacts are included in the final image, reducing size and attack surface.
- **Tag images properly:** Always use versioned tags (e.g., `my-app:v1.0.0`) instead of `latest` to avoid unexpected updates when pulling images.

- **Scan images for vulnerabilities:** Use security scanning tools like `docker scan`, `Trivy`, or `Clair` to identify and remediate security vulnerabilities in your images before deployment.
- **Manage environment variables securely:** Avoid storing sensitive credentials inside images. Use Docker secrets, environment variables, or external secret management tools like AWS Secrets Manager or HashiCorp Vault.
- **Use .dockerignore files:** Exclude unnecessary files (e.g., `.git, node_modules, venv`) to reduce build context size and prevent accidental inclusion of sensitive files in images.
- **Enable logging and monitoring:** Utilize tools like Prometheus, Grafana, and Fluentd for container logs and monitoring. Inspect logs using `docker logs` and enable structured logging for better observability.

Once you've mastered the basics of Docker, there are plenty of advanced topics to explore. Here are a few areas worth exploring next:

- **Docker Swarm & Kubernetes**: Explore Docker Swarm (built-in clustering) and Kubernetes (enterprise-grade orchestration with auto-scaling and service discovery) for production-grade orchestration.
- **Container security best practices**: To secure containerized applications, follow the CIS Docker Benchmark guidelines and implement Role-Based Access Control (RBAC).
- **CI/CD pipelines with Docker**: Automate image builds, security scans, and deployments using GitHub Actions, GitLab CI, or Jenkins.
- **Cloud-native development**: Leverage Docker with cloud platforms like AWS ECS, Azure Container Instances, and Google Cloud Run for scalable and managed deployments.
- **Data persistence strategies**: For optimal storage management, understand the differences between Docker volumes, bind mounts, and tmpfs.

Conclusion

Docker has revolutionized how developers build, ship, and run applications, making it an essential tool for modern software development.

In this tutorial, we covered:

- What Docker is and why it's important
- How to install and run your first container
- Key concepts like images, containers, and networking
- Persistent storage with Docker volumes
- Multi-container applications with Docker Compose
- Best practices for security, performance, and scalability

But this is just the beginning! If you want to deepen your Docker expertise, you can take a beginner-level **Introduction to Docker course**. For more in-depth knowledge, you can take an **Intermediate Docker course** that covers multi-stage builds, Docker networking tools, and Docker Compose. Finally, you can also pursue Docker certification, check out **The Complete Docker Certification (DCA) Guide for 2025** if interested!

Master Docker and Kubernetes

Learn the power of Docker and Kubernetes with an interactive track to build and deploy applications in modern environments.

Start Track for Free

FAQs

How is Docker different from a virtual machine (VM)?

Docker containers share the host OS kernel, making them lightweight and faster, whereas virtual machines (VMs) require a full guest OS, making them heavier and slower. Containers are better for microservices and rapid scaling, while VMs are more suitable for running different OS environments on the same hardware.

What is the difference between Docker CE and Docker EE?

Can Docker run on Windows without using WSL 2?

What are multi-arch Docker images?

How does Docker handle security?

AUTHOR
Moez Ali

Data Scientist, Founder & Creator of PyCaret

TOPICS

Kubernetes Docker

Training more people?

Get your team access to the full DataCamp for business platform.

For Business

For a bespoke solution book a demo.

Learn more about Docker and Kubernetes with these courses!

TRACK

Containerization and Virtualization with Docker and Kubernetes

0 min

Learn the power of Docker and Kubernetes, this interactive track will allow you to build and deploy applications in modern environments.

See Details →

Start Course

COURSE

Introductio

4 hr 38.

Gain an intro

See Details →

See More →

Related

BLOG

How to Learn Docker from Scratch: A Guide for Data...

BLOG

10 Docker Project Ideas: From Beginner to Advanced

TUTORIAL

Kubernetes Tutorial: A Beginner Guide to Deploying Applications

See More →

Grow your data skills with DataCamp for Mobile

Make progress on the go with our mobile courses and daily 5-minute coding challenges.

Download on the App Store GET IT ON Google Play

LEARN

Learn Python

Learn AI

Learn Power BI

Learn Data Engineering

Assessments

Career Tracks

Skill Tracks

Courses

Data Science Roadmap

DATA COURSES

Python Courses

R Courses

SQL Courses

Power BI Courses

Tableau Courses

Alteryx Courses

Azure Courses

AWS Courses

Google Sheets Courses

Excel Courses

AI Courses

Data Analysis Courses

Data Visualization Courses

Machine Learning Courses

Data Engineering Courses

Probability & Statistics Courses

DATALAB

Get Started

Pricing

Security

Documentation

CERTIFICATION

Certifications

Data Scientist

Data Analyst

Data Engineer

SQL Associate

Power BI Data Analyst

Tableau Certified Data Analyst

Azure Fundamentals

AI Fundamentals

RESOURCES

Resource Center

Upcoming Events

Blog

Code-Alongs

Tutorials

Docs

Open Source

RDocumentation

Book a Demo with DataCamp for Business

Data Portfolio

PLANS

Pricing

For Students

For Business

For Universities

Discounts, Promos & Sales

Expense DataCamp

DataCamp Donates

FOR BUSINESS

Business Pricing

Teams Plan

Data & AI Unlimited Plan

Customer Stories

Partner Program

ABOUT

About Us

Learner Stories

Careers

Become an Instructor

Press

Leadership

Contact Us

DataCamp Español

DataCamp Português

DataCamp Deutsch

DataCamp Français

SUPPORT

Help Center

Become an Affiliate

Privacy Policy Cookie Notice Do Not Sell My Personal Information Accessibility Security Terms of Use

© 2025 DataCamp, Inc. All Rights Reserved.