Docker Tutorial for Beginners

A complete guide to learning Docker from scratch with practical examples using a Next.js application.

Chapter 1: What is Docker and Why is it Useful?

What is Docker?

Docker is a platform that allows you to package your applications and all their dependencies into lightweight, portable containers. Think of it like a shipping container for your code - everything your app needs to run is bundled together.

Why is Docker Useful?

1. **Consistency**: Your app runs the same way on your laptop, your colleague's computer, and production servers
2. **Isolation**: Each container is separate from others, preventing conflicts
3. **Portability**: Containers can run anywhere Docker is installed
4. **Efficiency**: Containers share the host OS kernel, making them lighter than virtual machines
5. **Easy Deployment**: Package once, deploy anywhere

Real-World Analogy

Imagine you're moving houses. Instead of packing items individually, you use standardized boxes (containers) that fit perfectly in any moving truck (host system). Docker works the same way for applications.

Chapter 2: Docker Images vs Containers

Docker Images

- **What**: A blueprint or template for creating containers
- **Think of it as**: A recipe or a class in programming
- **Characteristics**: Read-only, immutable, can be shared and reused
- **Example**: `node:18-alpine` (a pre-built image with Node.js 18)

Docker Containers

- **What**: A running instance of an image
- **Think of it as**: A cake made from a recipe, or an object created from a class
- **Characteristics**: Can be started, stopped, modified (temporarily)
- **Example**: Your Next.js app running inside a container

Key Differences

Images	Containers

Images	Containers
Static templates	Running instances
Cannot be changed	Can be modified while running
Stored on disk	Exist in memory
One image → Many containers	Each container from one image

Chapter 3: Writing Your First Dockerfile

A Dockerfile is a text file with instructions to build a Docker image. Let's create one for our Next.js project.

Basic Dockerfile for Next.js App

```
# Use Node.js 18 with Alpine Linux (lightweight)
FROM node:18-alpine

# Set the working directory inside the container
WORKDIR /app

# Copy package.json and package-lock.json first
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code
COPY . .

# Build the Next.js application
RUN npm run build

# Expose port 3000 to the outside world
EXPOSE 3000

# Command to run when container starts
CMD ["npm", "start"]
```

Line-by-Line Explanation

1. `FROM node:18-alpine`
    - Starts with a base image that has Node.js 18 installed
    - Alpine Linux is chosen because it's very small and secure
2. `WORKDIR /app`
    - Sets `/app` as the working directory inside the container
    - All subsequent commands will run from this directory

3. `COPY package*.json ./`
    - Copies `package.json` and `package-lock.json` to the container
    - The `*` is a wildcard that matches both files
    - We copy these first to take advantage of Docker's layer caching
4. `RUN npm install`
    - Installs all dependencies listed in package.json
    - This creates a new layer in the image
5. `COPY . .`
    - Copies all remaining files from your project to the container
    - First `.` is source (your local directory)
    - Second `.` is destination (current WORKDIR in container)
6. `RUN npm run build`
    - Builds the Next.js application for production
    - Creates optimized static files
7. `EXPOSE 3000`
    - Documents that the container will listen on port 3000
    - This is informational - doesn't actually publish the port
8. `CMD ["npm", "start"]`
    - Defines the default command to run when container starts
    - Uses exec form (array) which is preferred over shell form

---

Chapter 4: Essential Docker Commands

Building Images

```
# Build an image from Dockerfile in current directory
docker build -t my-nextjs-app .

# Build with a specific tag
docker build -t my-nextjs-app:v1.0 .
```

Explanation:

- `-t` assigns a name (tag) to your image
- `.` tells Docker to look for Dockerfile in current directory
- Tags help you version your images

Running Containers

```
# Run container in foreground
docker run my-nextjs-app

# Run container in background (detached)
docker run -d my-nextjs-app

# Run with port mapping
docker run -d -p 3000:3000 my-nextjs-app

# Run with custom name
docker run -d -p 3000:3000 --name my-app my-nextjs-app
```

Key Options:

- `-d`: Detached mode (runs in background)
- `-p 3000:3000`: Maps host port 3000 to container port 3000
- `--name`: Gives your container a friendly name

Managing Containers

```
# List running containers
docker ps

# List all containers (including stopped)
docker ps -a

# Stop a running container
docker stop my-app
# or
docker stop <container-id>

# Remove a stopped container
docker rm my-app

# Stop and remove in one command
docker rm -f my-app
```

Managing Images

```
# List all images
docker images

# Remove an image
docker rmi my-nextjs-app

# Remove image by ID
docker rmi <image-id>
```

```
# Remove unused images
docker image prune
```

Viewing Logs and Debugging

```
# View container logs
docker logs my-app

# Follow logs in real-time
docker logs -f my-app

# Execute commands inside running container
docker exec -it my-app /bin/sh

# Run a one-off command
docker exec my-app ls -la
```

Explanation:

- `logs`: Shows what your application is printing to console
- `exec -it`: Interactive terminal inside container
- `/bin/sh`: Shell command (Alpine Linux uses sh instead of bash)

Chapter 5: Port Mapping Explained

Understanding Port Mapping

Containers are isolated environments. To access your app from outside the container, you need to map ports.

```
# Format: -p <host-port>:<container-port>
docker run -d -p 3000:3000 my-nextjs-app
```

Port Mapping Examples

```
# Map host port 8080 to container port 3000
docker run -d -p 8080:3000 my-nextjs-app
# Access via: http://localhost:8080

# Map to different host port
docker run -d -p 4000:3000 my-nextjs-app
# Access via: http://localhost:4000

# Let Docker choose a random host port
docker run -d -P my-nextjs-app
# Check with: docker ps
```

Why Port Mapping?

1. **Isolation**: Container port 3000 is not accessible from outside by default
2. **Flexibility**: You can run multiple containers on different host ports
3. **Security**: Only explicitly mapped ports are accessible

Chapter 6: Cleanup Commands

System Cleanup

```
# Remove all stopped containers, unused networks, images, and build cache
docker system prune

# More aggressive cleanup (includes unused images)
docker system prune -a

# Remove everything with volumes
docker system prune -a --volumes
```

Specific Cleanup Commands

```
# Remove all stopped containers
docker container prune

# Remove unused images
docker image prune

# Remove unused volumes
docker volume prune

# Remove unused networks
docker network prune
```

When to Use Cleanup

- **Regular maintenance**: Run `docker system prune` weekly
- **Low disk space**: Use `docker system prune -a`
- **Fresh start**: Clean everything when testing

Chapter 7: Basic Volume Usage (Optional)

What are Volumes?

Volumes allow you to persist data outside the container and share files between host and container.

Types of Volumes

1. **Bind Mounts**: Link host directory to container directory
2. **Named Volumes**: Docker-managed storage
3. **Anonymous Volumes**: Temporary storage

Practical Examples

```
# Bind mount for development (live code changes)
docker run -d -p 3000:3000 -v $(pwd):/app my-nextjs-app

# Named volume for persistent data
docker run -d -p 3000:3000 -v app-data:/app/data my-nextjs-app

# Read-only bind mount
docker run -d -p 3000:3000 -v $(pwd):/app:ro my-nextjs-app
```

When to Use Volumes

- **Development**: Live reload during coding
- **Data persistence**: Database files, user uploads
- **Configuration**: External config files

Chapter 8: Complete Practical Example

Let's put everything together with our Next.js project.

Step 1: Create the Dockerfile

Create a file named `Dockerfile` in your project root:

```
FROM node:18-alpine
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build
EXPOSE 3000
CMD ["npm", "start"]
```

Step 2: Build the Image

```
# Navigate to your project directory
cd /path/to/your/nextjs-project
```

```
# Build the Docker image
docker build -t dev-logs-app .
```

Step 3: Run the Container

```
# Run the container with port mapping
docker run -d -p 3000:3000 --name my-dev-logs dev-logs-app
```

Step 4: Access Your App

Open your browser and go to: `http://localhost:3000`

Step 5: Monitor and Debug

```
# Check if container is running
docker ps

# View application logs
docker logs my-dev-logs

# Access container shell if needed
docker exec -it my-dev-logs /bin/sh
```

Step 6: Stop and Clean Up

```
# Stop the container
docker stop my-dev-logs

# Remove the container
docker rm my-dev-logs

# Remove the image (optional)
docker rmi dev-logs-app
```

Chapter 9: Common Issues and Solutions

Issue 1: Port Already in Use

Error: `Port 3000 is already allocated`

Solution:

```
# Use a different host port
docker run -d -p 8080:3000 my-nextjs-app

# Or stop the conflicting service
docker ps
docker stop <conflicting-container>
```

Issue 2: Build Fails

Error: `npm install` fails during build

Solutions:

1. Check your `package.json` is valid
2. Ensure you have internet connection during build
3. Try building without cache: `docker build --no-cache -t my-app .`

Issue 3: Container Exits Immediately

Check logs:

```
docker logs <container-name>
```

Common causes:

- Application crashes on startup
- Missing environment variables
- Port conflicts inside container

Chapter 10: Best Practices for Beginners

Dockerfile Best Practices

1. **Use specific base image versions**

```
# Good
FROM node:18-alpine

# Avoid
FROM node:latest
```

2. **Copy package.json first for better caching**

```
COPY package*.json ./
RUN npm install
```

```
COPY . .
```

3. **Use .dockerignore file** Create `.dockerignore` to exclude unnecessary files:

```
node_modules
.git
.env
README.md
```

Container Management

1. **Always name your containers**

```
docker run --name my-app my-image
```

2. **Use environment-specific tags**

```
docker build -t my-app:dev .
docker build -t my-app:prod .
```

3. **Regular cleanup**

```
docker system prune
```

Chapter 11: Quick Reference Commands

Essential Commands Cheat Sheet

```
# Build
docker build -t <image-name> .

# Run
docker run -d -p <host-port>:<container-port> --name <container-name> <image-name>

# List
docker ps          # Running containers
docker ps -a       # All containers
docker images      # All images

# Stop/Remove
docker stop <container-name>
```

```
docker rm <container-name>
docker rmi <image-name>

# Debug
docker logs <container-name>
docker exec -it <container-name> /bin/sh

# Cleanup
docker system prune
```

Next.js Specific Workflow

```
# 1. Build image
docker build -t my-nextjs-app .

# 2. Run container
docker run -d -p 3000:3000 --name nextjs-container my-nextjs-app

# 3. Access app
# Open http://localhost:3000

# 4. View logs
docker logs nextjs-container

# 5. Stop and cleanup
docker stop nextjs-container
docker rm nextjs-container
```

Conclusion

Congratulations! You've learned the fundamentals of Docker:

- ✅ **Understanding**: What Docker is and why it's useful
- ✅ **Core Concepts**: Images vs containers
- ✅ **Dockerfile**: Writing and understanding each instruction
- ✅ **Commands**: Building, running, and managing containers
- ✅ **Networking**: Port mapping for web applications
- ✅ **Cleanup**: Maintaining a clean Docker environment
- ✅ **Practical Skills**: Containerizing a real Next.js application

What's Next?

Once you're comfortable with these basics, you can explore:

- Docker Compose for multi-container applications
- Environment variables and secrets management
- Production deployment strategies
- Container orchestration with Kubernetes

Remember

- Practice with real projects
- Start simple and gradually add complexity
- Use Docker documentation when you need more details
- Join Docker communities for help and best practices

Happy containerizing! 🐳