Fly.io Deployment Guide

FastAPI Blog Application

Complete Step-by-Step Documentation

Learn how to deploy your FastAPI blog to Fly.io
with Docker, persistent storage, and zero cost

Generated on January 30, 2026

Table of Contents

Chapter 1: Introduction to Fly.io

What is Fly.io?

Fly.io is a modern cloud platform that runs your applications close to your users worldwide. Unlike traditional hosting (like a VPS or shared hosting), Fly.io uses containerization to package your application with all its dependencies, ensuring it runs identically everywhere.

Key Features

* Global deployment - Run your app in multiple regions
* Persistent volumes - Store data that survives container restarts
* Automatic HTTPS - Free SSL certificates for your apps
* Zero downtime deployments - Update without taking your app offline
* Generous free tier - Perfect for personal projects

How

Traditional Hosting (VPS/Shared):
- You manage the server OS
- Install dependencies manually
- Configure web servers (nginx, apache)
- Handle security updates
- Fixed to one location

Fly.io (Container Platform):
- Application packaged in a container
- Dependencies included in the container
- No web server configuration needed
- Automatic security updates
- Deploy globally with one command

Why Fly.io for Your Blog?

* SQLite Support: Persistent volumes mean SQLite works perfectly
* No Database Migration: Keep your existing SQLite database
* Always Running: No cold starts unlike serverless platforms
* Free Tier: 3 VMs and 3 GB storage completely free
* Simple Deployment: One command to deploy

Chapter 2: Understanding Docker Fundamentals

What is Docker?

Docker is a platform that packages your application and everything it needs to run into a standardized unit called a container. Think of it as creating a complete, self-contained environment for your application.

The Problem Docker Solves

The Classic Problem: 'It works on my machine!'

Your Computer:
- Windows 11
- Python 3.10.10
- Specific package versions
- Your app works perfectly

Production Server:
- Linux Ubuntu
- Python 3.9.5
- Different package versions
- Your app breaks!

Docker's Solution:
Package everything your app needs (Python version, packages, code) into a container. This container runs identically on any system - your computer, the server, or Fly.io.

Key Docker Concepts

1. Dockerfile

A Dockerfile is a text file containing instructions to build your application container. It's like a recipe that tells Docker:
- What base system to use (e.g., Python 3.10 on Linux)
- What files to copy
- What commands to run
- How to start your application

2. Docker Image

An image is the result of building a Dockerfile. It's a snapshot of your complete application environment, frozen in time. Images are:
- Immutable (never change)
- Portable (run anywhere)
- Layered (efficient storage)

- Versioned (can tag v1, v2, etc.)

3. Docker Container

A container is a running instance of an image. It's like the difference between:
- Image = A program installer (setup.exe)
- Container = The running program

Containers are:
- Isolated from other containers
- Can be started, stopped, restarted
- Disposable (delete and recreate anytime)
- Multiple containers can run from one image

Docker Analogy: Shipping Container

Imagine shipping physical goods:

Without Containers (Old Way):
- Different sizes and shapes
- Hard to stack
- Inefficient loading/unloading
- Items get damaged

With Standard Containers:
- Same size and shape
- Easy to stack
- Efficient transport
- Contents protected

Docker containers do the same for software - standardize how applications are packaged and deployed.

How Docker Works with Fly.io

1. You write a Dockerfile
2. Fly.io builds an image from your Dockerfile
3. Fly.io creates a container from that image
4. The container runs on Fly.io's servers
5. Your app is accessible at your-app.fly.dev

Chapter 3: Dockerfile - Building Your Application Image

Complete Dockerfile for Your Blog

[Dockerfile]

```
FROM python:3.10-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 8080

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Line-by-Line Explanation

Line 1: FROM python:3.10-slim

The FROM instruction sets the base image for your container.

python:3.10-slim means:
- python: Official Python image from Docker Hub
- 3.10: Python version 3.10.x
- slim: Minimal version (smaller size, faster downloads)

This gives you a Linux system with Python 3.10 pre-installed. Alternative options:
- python:3.10 (full version, includes build tools)
- python:3.10-alpine (even smaller, but can have compatibility issues)
- python:3.11-slim (if you want newer Python)

Line 3: WORKDIR /app

WORKDIR sets the working directory inside the container.

What it does:
- Creates /app directory if it doesn't exist
- Changes to that directory (like 'cd /app')
- All subsequent commands run from /app
- When your app runs, it starts in /app

Why /app?
- Convention (most Docker images use it)

- Keeps your code organized
- Separate from system directories

Line 5: COPY requirements.txt .

COPY instruction copies files from your computer into the container.

Syntax: COPY <source> <destination>
- Source: requirements.txt (on your computer)
- Destination: . (current directory = /app)

Why copy requirements.txt first?
Docker uses layers and caching. If requirements.txt hasn't changed, Docker reuses the cached layer instead of reinstalling packages. This makes rebuilds faster.

Line 6: RUN pip install --no-cache-dir -r requirements.txt

RUN executes a command during the image build process.

This command:
- pip install: Install Python packages
- --no-cache-dir: Don't store pip cache (saves ~100 MB)
- -r requirements.txt: Install from requirements file

What gets installed:
- fastapi
- uvicorn
- sqlalchemy
- markdown
- jinja2
- python-multipart
- aiofiles

These packages become part of the image and are available when your app runs.

Line 8: COPY . .

Copies all your application code into the container.

What gets copied:
- app/ directory (your FastAPI code)
- templates/ directory (Jinja2 templates)
- static/ directory (CSS, JS files)
- Everything not in .dockerignore

Why copy code last?
Code changes frequently. By copying it last, you don't invalidate the cached package installation layer. Only the code layer needs to rebuild.

Line 10: EXPOSE 8080

EXPOSE documents which port your application uses.

Important: This is documentation only!
- Doesn't actually open the port
- Doesn't publish the port
- Tells other developers (and Fly.io) which port to use

Why 8080?
- Common convention for web apps

- Fly.io expects this port
- Non-privileged port (doesn't need root)

Your uvicorn server must listen on this port (configured in CMD).

Line 12: CMD

CMD specifies the command to run when the container starts.

Breaking down the command:
- uvicorn: ASGI server (runs your FastAPI app)
- app.main:app: Python path to your FastAPI instance
  * app.main = app/main.py file
  * :app = the 'app' variable in main.py
- --host 0.0.0.0: Listen on all network interfaces
  * Required for Docker (127.0.0.1 won't work)
  * Makes app accessible from outside container
- --port 8080: Listen on port 8080 (matches EXPOSE)

This is the same command you run locally, but with --host 0.0.0.0

Best Practices

* Use specific version tags (python:3.10-slim, not python:latest)
* Copy requirements.txt before code (better caching)
* Use --no-cache-dir to reduce image size
* Use WORKDIR instead of RUN cd
* Use slim or alpine images for smaller size
* Run as non-root user in production (not covered in basic setup)

Chapter 4: Docker Ignore - Optimizing Your Build

What is .dockerignore?

The .dockerignore file tells Docker which files and directories to exclude when copying files into the container. It works just like .gitignore for Git.

Why is this important?
- Faster builds (less data to copy)
- Smaller images (no unnecessary files)
- Security (don't include secrets)
- Correctness (exclude conflicting files)

Complete .dockerignore for Your Blog

[.dockerignore]

```
# Virtual Environment
website_v1/
venv/
env/

# Python Cache
__pycache__/
*.pyc
*.pyo
*.pyd

# Local Database
blogs.db
blogs.db-journal

# Git
.git/
.gitignore
.gitattributes

# Documentation
*.md
*.pdf

# IDE
.vscode/
.idea/
*.swp

# Environment
.env
.env.local
```

```
# Build artifacts
build/
dist/
*.egg-info/
```

Entry-by-Entry Explanation

Virtual Environment Files

website_v1/, venv/, env/

Why exclude:
- Your virtual environment contains 700+ MB of packages
- These are specific to your Windows computer
- Won't work in the Linux container
- Docker installs fresh packages via requirements.txt

Impact: Saves 700 MB and several minutes of build time

Python Cache Files

__pycache__/, *.pyc, *.pyo, *.pyd

Why exclude:
- Bytecode cache files generated by Python
- Platform-specific (Windows vs Linux)
- Will be regenerated in container
- No benefit to copying them

Impact: Saves 5-10 MB and prevents potential issues

Local Database

blogs.db, blogs.db-journal

Why exclude:
- Your local test database
- Should start fresh on Fly.io
- Will be created on the persistent volume
- Prevents overwriting production data

Important: The database on Fly.io lives on the volume, not in the container.

Git Repository

.git/, .gitignore, .gitattributes

Why exclude:
- .git/ contains entire version history (can be 100+ MB)
- Not needed for running the app
- Security: prevents exposing Git history
- .gitignore is only needed during development

Impact: Can save 100+ MB depending on repository size

Documentation Files

*.md, *.pdf

Why exclude:
- README.md, documentation PDFs, etc.
- Only needed by developers
- Not used by the running application
- Reduces image size

Exception: If your blog displays markdown files, don't exclude them!

IDE Configuration

.vscode/, .idea/, *.swp

Why exclude:
- Editor-specific settings
- Not needed to run the app
- Can contain sensitive paths
- Keeps image clean

Environment Variables

.env, .env.local

Why exclude:
- Contains secrets (database passwords, API keys)
- Should NEVER be in Docker image
- Use Fly.io secrets instead
- Security critical!

Remember: Environment variables should be set via 'fly secrets set'

Testing Your .dockerignore

To verify what gets copied:

1. Build the image locally:
   docker build -t test-blog .

2. Inspect the image:
   docker run --rm test-blog ls -lah /app

3. Check size:
   docker images test-blog

You should see:
- No website_v1/ directory
- No __pycache__/ directories
- No .git/ directory
- Image size around 200-300 MB (without exclusions: 1+ GB)

Chapter 5: Fly.toml - Configuring Your Deployment

What is fly.toml?

fly.toml is the configuration file that tells Fly.io how to run your application. It's written in TOML format (Tom's Obvious Minimal Language), which is easy to read and write.

Complete fly.toml for Your Blog

[fly.toml]

```
app = "your-blog-app"
primary_region = "iad"

[build]
  dockerfile = "Dockerfile"

[env]
  PORT = "8080"

[http_service]
  internal_port = 8080
  force_https = true
  auto_stop_machines = false
  auto_start_machines = true
  min_machines_running = 1

[mounts]
  source = "blog_data"
  destination = "/app"
```

Section-by-Section Explanation

App Configuration

app = 'your-blog-app'

Your application name on Fly.io:
- Must be globally unique across all Fly.io
- Becomes your URL: your-blog-app.fly.dev
- Lowercase letters, numbers, hyphens only
- Change 'your-blog-app' to something unique

Example names:
- mdine-personal-blog
- my-fastapi-blog-2024
- techwriter-stories

Primary Region

primary_region = 'iad'

Where your app runs:
- iad = Washington DC, USA (East Coast)
- lax = Los Angeles, USA (West Coast)
- lhr = London, UK
- fra = Frankfurt, Germany
- syd = Sydney, Australia
- nrt = Tokyo, Japan

Choose the region closest to your users for best performance.

Build Section

[build]
  dockerfile = 'Dockerfile'

Tells Fly.io how to build your app:
- dockerfile: Use Dockerfile in your project root
- Alternative: Can specify path like 'docker/Dockerfile'
- Fly.io runs 'docker build' with this file

Other build options:
- image: Use pre-built image from Docker Hub
- builder: Use Fly.io buildpacks (auto-detect)

Environment Variables

[env]
  PORT = '8080'

Environment variables available to your app:
- PORT: Your app listens on this port
- Accessible in Python via os.getenv('PORT')
- Can add more: DATABASE_URL, API_KEY, etc.

For secrets, use 'fly secrets set' instead (encrypted).

HTTP Service Configuration

[http_service]

Controls how Fly.io exposes your app to the internet:

internal_port = 8080
- Port your app listens on inside the container
- Must match the port in your CMD (uvicorn --port 8080)
- Fly.io proxies external traffic to this port

force_https = true
- Automatically redirect HTTP to HTTPS
- Fly.io provides free SSL certificates
- Recommended for security

auto_stop_machines = false
- false: Keep your app running 24/7
- true: Stop when idle (save resources, but slower first request)

auto_start_machines = true

- Automatically start if stopped
- Ensures availability

min_machines_running = 1
- Always keep at least 1 machine running
- Ensures your app is always accessible
- For high availability, set to 2+

Mounts (Persistent Storage)

```
[mounts]
  source = 'blog_data'
  destination = '/app'
```

Critical for SQLite!

source: Name of the volume to mount
- Created with: fly volumes create blog_data
- Persistent storage that survives container restarts
- Must exist before first deployment

destination: Where to mount it in the container
- /app: Same as WORKDIR in Dockerfile
- Your blogs.db will be stored here
- Persists across deployments

How it works:
1. Container starts with /app from image
2. Volume overlays /app
3. Files written to /app go to volume
4. Container restarts, volume remains

Chapter 6: Persistent Storage with Volumes

Why Do We Need Volumes?

Containers are ephemeral (temporary). When a container restarts:
- All files created inside are deleted
- Your blogs.db would be lost!
- Users' blog posts would disappear

Volumes solve this:
- External storage attached to container
- Survives container restarts
- Survives deployments
- Survives crashes

Think of it like:
- Container = Actor playing a role (can be replaced)
- Volume = The script (stays the same)

How Volumes Work

Without Volume:
```
Container
  /app/
    main.py
    models.py
    blogs.db <- DELETED on restart!
```

With Volume:
```
Container              Volume 'blog_data'
  /app/ ----- mounted ---> [Persistent Storage]
    main.py                  blogs.db <- PERSISTS!
    models.py
```

Files in /app are actually stored on the volume.

Creating a Volume

Command:
```
  fly volumes create blog_data --size 1
```

Parameters:
- blog_data: Volume name (must match fly.toml)

- --size 1: Size in GB (1 GB is plenty for a blog)
- Optional: --region iad (specify region)

This creates a 1 GB persistent disk in your chosen region.

Volume Characteristics

* Persistent: Data survives container restarts and deployments
* Regional: Tied to one region (can't move between regions)
* Single-attach: Can only be attached to one machine at a time
* Backed up: Fly.io snapshots volumes daily
* Expandable: Can increase size (but not decrease)

SQL

Your blogs.db file:
1. Created when your app first runs
2. Stored in /app/blogs.db
3. Actually on the volume
4. Persists forever

When you deploy updates:
1. New container is created
2. Volume is mounted to /app
3. blogs.db is still there
4. All blog posts preserved

Chapter 7: Deployment Process Step-by-Step

Prerequisites

* Fly.io account (free signup at fly.io)
* Fly CLI installed on your computer
* Your FastAPI blog code
* Dockerfile, .dockerignore, and fly.toml created

Step

Windows:
1. Download installer from fly.io/docs/flyctl/install
2. Run the installer
3. Restart your terminal

Verify installation:
```
flyctl version
```

Should show something like: flyctl v0.x.x

Step 2: Authenticate

Command:
```
flyctl auth login
```

What happens:
1. Opens browser
2. Login or signup for Fly.io
3. Authorize the CLI
4. Returns to terminal

You're now authenticated and can deploy apps.

Step 3: Launch Your App

Command:
```
flyctl launch
```

This interactive command:
1. Detects your app (sees Dockerfile)
2. Asks for app name (must be unique)
3. Asks for region (choose closest to you)
4. Asks about PostgreSQL (say NO - we use SQLite)

5. Creates fly.toml (or uses existing one)
6. Registers your app with Fly.io

Does NOT deploy yet - just creates the app.

Step 4: Create Volume

Command:
```
flyctl volumes create blog_data --size 1
```

What happens:
1. Creates a 1 GB persistent volume
2. Names it 'blog_data'
3. Creates in your primary region
4. Ready to mount to your app

Output shows:
- Volume ID
- Region
- Size
- Status: created

Step 5: Deploy

Command:
```
flyctl deploy
```

What happens:
1. Packages your code
2. Uploads to Fly.io
3. Builds Docker image (runs Dockerfile)
4. Creates container from image
5. Mounts the volume
6. Starts your app
7. Runs health checks
8. Routes traffic to your app

Takes 2-5 minutes. Watch the output for any errors.

Step 6: Verify Deployment

Check status:
```
flyctl status
```

Open your app:

```
flyctl open
```

View logs:

```
flyctl logs
```

Your app should now be live at https://your-app.fly.dev!

Chapter 8: Managing Your Application

Common Commands

[Bash]

```
# View app status
flyctl status

# View logs (real-time)
flyctl logs

# SSH into container
flyctl ssh console

# List all apps
flyctl apps list

# View app info
flyctl info

# Restart app
flyctl apps restart

# Scale app
flyctl scale count 2

# View volumes
flyctl volumes list

# Deploy updates
flyctl deploy
```

Updating Your App

When you make code changes:

1. Make your changes locally
2. Test locally (uvicorn app.main:app)
3. Run: flyctl deploy
4. Fly.io builds new image
5. Creates new container
6. Mounts existing volume (data preserved)
7. Switches traffic to new container
8. Old container is removed

Your blogs.db and all data remain untouched!

Chapter 11: Cost Analysis and Free Tier

Free Tier Includes

* 3 shared-cpu-1x VMs (256 MB RAM each)
* 3 GB persistent volume storage
* 160 GB outbound data transfer per month
* Unlimited inbound data transfer
* Automatic HTTPS certificates
* Custom domains (unlimited)

Wha

* 1 VM (running your FastAPI app)
* 1 GB volume (for SQLite database)
* ~5-10 GB bandwidth/month (moderate traffic)

You

* 2 more VMs for other projects
* 2 GB more storage
* 150 GB bandwidth

Esti

Blog Posts:
- 1 GB can store ~33,000 medium-length posts
- Writing 1 post/day = 274 years of content

Traffic:
- 160 GB bandwidth = ~2,000,000 page views/month
- Average personal blog: 1,000-10,000 views/month
- You have 200x-2000x headroom

Conclusion: You'll stay in free tier indefinitely!

Chapter 14: Quick Reference Commands

Initial Deployment

[PowerShell]

```
# 1. Install Fly CLI
iwr https://fly.io/install.ps1 -useb | iex

# 2. Login
flyctl auth login

# 3. Launch app
flyctl launch

# 4. Create volume
flyctl volumes create blog_data --size 1

# 5. Deploy
flyctl deploy

# 6. Open app
flyctl open
```

Daily Usage

[Bash]

```
# Deploy updates
flyctl deploy

# View logs
flyctl logs

# Check status
flyctl status

# SSH into container
flyctl ssh console

# Restart app
flyctl apps restart
```

Troubleshooting

[Bash]

```
# View recent logs
flyctl logs --region iad

# Check volume
flyctl volumes list
```

```
# View machine info
flyctl machines list

# Verify database
flyctl ssh console
ls -lah /app
sqlite3 /app/blogs.db .tables
```