CI/CD Part 1

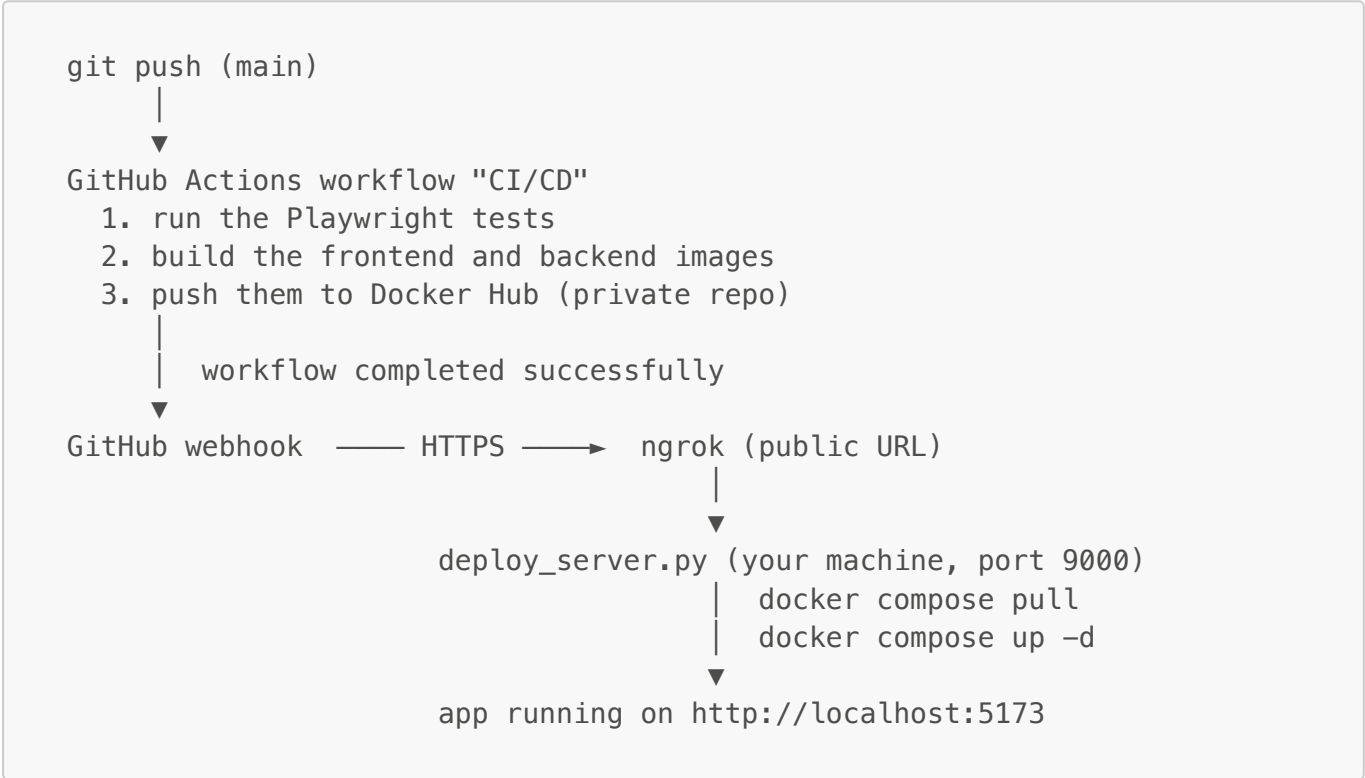
Goal: every push to `main` automatically tests the app, builds its images, publishes them to Docker Hub, and redeploys the app on your machine. No manual step.

- CI (Continuous Integration): every change is tested and built automatically.
- CD (Continuous Deployment): every change that passes is deployed automatically.

Starting point: the `node-auth-app` project from `COMPOSE_GUIDE.md` (frontend, backend, database).

---

1. The pipeline you will build



2. Docker Hub

2.1 Account and private repository

1. Sign up at https://hub.docker.com/signup and verify your email.
2. Repositories → Create a repository:
   - Name: `auth-app`
   - Visibility: Private

> The free plan includes 1 private repository. Both images go into it, told apart by their tag:
>
> - `<your-docker-id>/auth-app:frontend`
> - `<your-docker-id>/auth-app:backend`

2.2 Access tokens

Never use your Docker password in automation. Create personal access tokens instead:

1. Go to https://app.docker.com/ → your avatar (top right) → Account settings → Personal access tokens → Generate new token.
2. Create two tokens:

Description	Access permissions	Used by
`github-actions`	Read & Write	GitHub, to push images
`local-deploy`	Read-only	Your machine, to pull images

3. Copy each token right away: Docker shows it only once.

Give each user the least access it needs: your machine only ever pulls, so it gets a read-only token.

2.3 Log in on your machine

The images are private, so your machine must log in to pull them:

```
docker login --username <your-docker-id>
```

Paste the `local-deploy` token as the password.

---

3. GitHub Actions

3.1 Store the credentials in GitHub

In your GitHub repository: Settings → Secrets and variables → Actions.

Tab	Name	Value
Variables	`DOCKERHUB_USERNAME`	your Docker ID
Secrets	`DOCKERHUB_TOKEN`	the `github-actions` token

Secrets are encrypted and hidden in logs. Variables are for values that aren't sensitive.

3.2 How a GitHub workflow works

- A workflow is a YAML file in `.github/workflows/`, e.g. `.github/workflows/cicd.yml`. GitHub picks it up automatically on the next push.
- `name`: the workflow's name, shown in the Actions tab.
- `on`: what triggers it (a push, a pull request, a schedule...).
- `jobs`: one or more jobs. Each runs on a fresh virtual machine (`runs-on: ubuntu-latest`), which comes with Docker and Docker Compose installed.
- `steps`: run in order inside a job. A step either `uses:` a ready-made action from the Marketplace, or `run:`s shell commands.
- If any step fails, the job stops and the workflow is marked failed.

- Watch runs live in the repository's Actions tab.

3.3 Authenticating to Docker Hub

This step logs the runner in to Docker Hub, using the variable and secret from 3.1:

```
      - name: Log in to Docker Hub
        uses: docker/login-action@v4
        with:
          username: ${{ vars.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}
```

3.4 the workflow

Write `.github/workflows/cicd.yml` so that:

1. The workflow is named exactly `CI/CD` (the deploy server checks this name).
2. It runs on pushes to `main` only.
3. It runs the Playwright tests against the full stack (frontend, backend, database). If a test fails, nothing is pushed.
4. It builds the frontend and backend images and pushes them as `<your-docker-id>/auth-app:frontend` and `<your-docker-id>/auth-app:backend`.
5. It finishes green only if all of the above succeeded.

Checkpoint: push to `main`. The run is green in the Actions tab, and both tags appear in your `auth-app` repository on Docker Hub, with a recent "last pushed" time.

---

4. Run the app from Docker Hub

Your machine no longer builds anything: it runs the images CI produced.

`docker-compose.deploy.yml`

Create a folder outside the project, e.g. `auth-app-deploy/`, containing only a `docker-compose.deploy.yml` that:

1. Runs the same three services as `docker-compose.yml`: `db`, `backend`, `frontend`.
2. Uses your Docker Hub images for `backend` and `frontend`. No `build:`.
3. Keeps the database volume, so users survive redeploys.
4. Publishes the frontend on port 5173.

Checkpoint: from that folder, run `docker compose -f docker-compose.deploy.yml pull` then `docker compose -f docker-compose.deploy.yml up -d`. The app works on http://localhost:5173.

> Stop any stack still running from `COMPOSE_GUIDE.md` first (`docker compose down` in the project folder): both use port 5173.

---

5. The deploy server (provided)

A small Python API that listens on port 9000. When GitHub calls `POST /deploy`, it:

1. checks the request is really from GitHub (signature made with a shared secret),
2. checks the event is a successful run of the `CI/CD` workflow on `main`,
3. runs `docker compose pull` then `docker compose up -d` with your deploy file.

The same file works on macOS and Windows. It uses only Python's standard library, so there's nothing to install. Save it as `deploy_server.py` in your `auth-app-deploy/` folder:

```
"""Deploy server: receives GitHub webhooks (through ngrok) and redeploys
the app.

When the CI/CD workflow finishes successfully on main, GitHub calls POST
/deploy.
This server checks the request really comes from GitHub, then runs:
    docker compose -f <COMPOSE_FILE> pull
    docker compose -f <COMPOSE_FILE> up -d

Works on macOS and Windows. Python 3.10+, standard library only (no pip
install).
"""
import hashlib
import hmac
import json
import os
import shutil
import subprocess
import sys
import threading
from datetime import datetime
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer

# Settings come from environment variables (see the guide for how to set
them).
SECRET = os.environ.get("WEBHOOK_SECRET", "")
COMPOSE_FILE = os.environ.get("COMPOSE_FILE", "docker-compose.deploy.yml")
WORKFLOW_NAME = os.environ.get("WORKFLOW_NAME", "CI/CD")
BRANCH = os.environ.get("BRANCH", "main")
PORT = int(os.environ.get("PORT", "9000"))

deploy_lock = threading.Lock()  # one deploy at a time; a second one waits
its turn


def log(message):
    print(f"[{datetime.now():%H:%M:%S}] {message}", flush=True)


def signature_ok(body, header):
    """GitHub signs each delivery with the webhook secret (HMAC-SHA256)."""
    expected = "sha256=" + hmac.new(SECRET.encode(), body,
hashlib.sha256).hexdigest()
```

```
    return hmac.compare_digest(expected, header or "")


def run(args):
    log("Running: " + " ".join(args))
    result = subprocess.run(args, capture_output=True, text=True,
encoding="utf-8", errors="replace")
    output = (result.stdout + result.stderr).strip()
    if output:
        print(output, flush=True)
    return result.returncode == 0


def deploy(commit):
    with deploy_lock:
        log(f"Deploying commit {commit[:7]}...")
        compose = ["docker", "compose", "-f", COMPOSE_FILE]
        if run(compose + ["pull"]) and run(compose + ["up", "-d"]):
            log(f"Deploy finished: commit {commit[:7]} is live on
http://localhost:5173")
        else:
            log("Deploy FAILED (see the output above)")


def reason_to_ignore(payload):
    """Return why this workflow_run event should NOT trigger a deploy, or
None."""
    run_info = payload.get("workflow_run", {})
    if payload.get("action") != "completed":
        return f"action is '{payload.get('action')}', waiting for
'completed'"
    if run_info.get("name") != WORKFLOW_NAME:
        return f"workflow '{run_info.get('name')}' is not
'{WORKFLOW_NAME}'"
    if run_info.get("head_branch") != BRANCH:
        return f"branch '{run_info.get('head_branch')}' is not '{BRANCH}'"
    if run_info.get("conclusion") != "success":
        return f"workflow conclusion is '{run_info.get('conclusion')}'"
    return None


class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        self.reply(200, "Deploy server is running")  # handy check in a
browser

    def do_POST(self):
        if self.path != "/deploy":
            return self.reply(404, "Not found")

        body = self.rfile.read(int(self.headers.get("Content-Length", 0)))
        if not signature_ok(body, self.headers.get("X-Hub-Signature-256")):
            log("Rejected a request with an invalid signature")
            return self.reply(401, "Invalid signature")
```

```
        event = self.headers.get("X-GitHub-Event")
        if event == "ping":  # sent by GitHub when the webhook is created
            log("Ping from GitHub: the webhook is connected")
            return self.reply(200, "pong")
        if event != "workflow_run":
            return self.reply(202, f"Ignored event '{event}'")

        try:
            payload = json.loads(body)
        except json.JSONDecodeError:
            return self.reply(400, "Body is not JSON (set the webhook
content type to application/json)")

        reason = reason_to_ignore(payload)
        if reason:
            log(f"Ignored: {reason}")
            return self.reply(202, f"Ignored: {reason}")

        # GitHub expects an answer within 10 seconds: deploy in the
background.
        commit = payload["workflow_run"].get("head_sha", "")
        threading.Thread(target=deploy, args=(commit,),
daemon=True).start()
        self.reply(202, "Deploy started")

    def reply(self, status, text):
        data = (text + "\n").encode()
        self.send_response(status)
        self.send_header("Content-Type", "text/plain; charset=utf-8")
        self.send_header("Content-Length", str(len(data)))
        self.end_headers()
        self.wfile.write(data)

    def log_message(self, *args):
        pass  # silence the default per-request log lines; we log what
matters


if __name__ == "__main__":
    if not SECRET:
        sys.exit("Set the WEBHOOK_SECRET environment variable first (see
the guide).")
    if not shutil.which("docker"):
        log("Warning: 'docker' was not found. Is Docker Desktop installed
and running?")
    if not os.path.isfile(COMPOSE_FILE):
        log(f"Warning: compose file not found: {COMPOSE_FILE}")

    log(f"Deploy server listening on http://localhost:{PORT}  (GitHub calls
POST /deploy)")
    log(f"Deploys successful '{WORKFLOW_NAME}' runs on '{BRANCH}' using
{COMPOSE_FILE}")
    try:
```

```
        ThreadingHTTPServer(("127.0.0.1", PORT), Handler).serve_forever()
    except KeyboardInterrupt:
        log("Stopped")
```

Run it

First, generate a random secret. You'll paste the same value into GitHub in step 7:

```
python3 -c "import secrets; print(secrets.token_hex(32))"      # macOS
py -c "import secrets; print(secrets.token_hex(32))"           # Windows
```

macOS (Terminal), from the `auth-app-deploy` folder:

```
export WEBHOOK_SECRET="paste-your-secret-here"
export COMPOSE_FILE="$PWD/docker-compose.deploy.yml"
python3 deploy_server.py
```

Windows (PowerShell), from the `auth-app-deploy` folder:

```
$env:WEBHOOK_SECRET = "paste-your-secret-here"
$env:COMPOSE_FILE = "$PWD\docker-compose.deploy.yml"
py deploy_server.py
```

Leave it running. Open http://localhost:9000: it should say *Deploy server is running*.

Variable	Default	Meaning
`WEBHOOK_SECRET`	*(required)*	Shared secret with the GitHub webhook
`COMPOSE_FILE`	`docker-compose.deploy.yml`	Compose file to deploy
`WORKFLOW_NAME`	`CI/CD`	Only this workflow triggers a deploy
`BRANCH`	`main`	Only this branch triggers a deploy
`PORT`	`9000`	Port the server listens on

6. ngrok: a public address for your machine

GitHub is on the internet and can't reach `localhost`. ngrok opens a secure tunnel: a public HTTPS URL that forwards to a port on your machine.

1. Create a free account: https://dashboard.ngrok.com/signup
2. Install ngrok:
    - macOS: `brew install ngrok`
    - Windows: download it from https://ngrok.com/download

3. Copy your authtoken from https://dashboard.ngrok.com/get-started/your-authtoken, then connect the agent to your account (once per machine):

```
ngrok config add-authtoken <your-authtoken>
```

4. In a new terminal, start the tunnel to the deploy server:

```
ngrok http 9000
```

ngrok shows a line like:

```
Forwarding   https://your-name.ngrok-free.dev -> http://localhost:9000
```

That `https://...` address is your public URL. Free accounts get a fixed dev domain**, so the address stays the same when you restart ngrok.

> Tip: open http://127.0.0.1:4040 to see every request that goes through the tunnel, with headers and body.

---

7. The GitHub webhook

A webhook is GitHub calling a URL of yours when something happens in the repository.

In your repository: Settings → Webhooks → Add webhook:

Field	Value
Payload URL	`https://your-name.ngrok-free.dev/deploy`
Content type	`application/json`
Secret	the same value as `WEBHOOK_SECRET`
Which events?	select individual events → untick *Pushes*, tick Workflow runs
Active	ticked

Click Add webhook. GitHub immediately sends a ping: the deploy server logs *Ping from GitHub: the webhook is connected*, and the webhook's Recent Deliveries tab shows a green tick.

> Why "Workflow runs" and not "Pushes"? A push event fires the moment you push, before the images are built. *Workflow runs* fires when the workflow finishes, and the deploy server only deploys when it finished with success.

---

8. Run the whole pipeline

Everything must be running, in this order:

Terminal	Command
1	`deploy_server.py` (step 5)
2	`ngrok http 9000` (step 6)

Docker Desktop must be running too.

Now make a visible change, e.g. the *Welcome* heading text in `src/main.js`, then commit and push to `main`. Watch:

1. GitHub → Actions: the `CI/CD` run starts, then turns green.
2. Docker Hub: both tags show a new "last pushed" time.
3. Webhook → Recent Deliveries: a `workflow_run` delivery, answered `202`.
4. Deploy server terminal: `Deploying commit ...`, the pull and up output, then `Deploy finished`.
5. http://localhost:5173: log in. Your change is live.

Then break a test on purpose and push again: the workflow turns red, nothing is pushed, and the deploy server logs *Ignored: workflow conclusion is 'failure'*.

---

9. Deliverable:

A Loom video link (https://www.loom.com) of your entire process running, from push to containers running locally