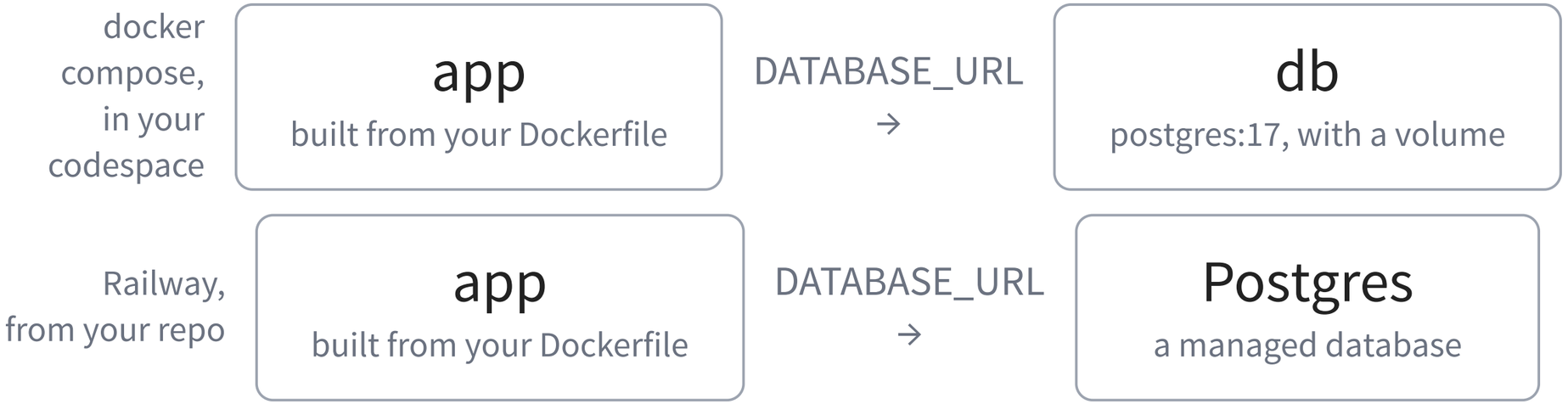
From Compose to Railway

A real database beside your app, then the same two services in the cloud

E190BO · AI-Amplified Prototyping

Compose runs your app and its database side by side, and Railway runs the same two services in the cloud



The app reads its database from `DATABASE_URL`, so one image works with any database

```
def database_url():
    url = os.environ.get("DATABASE_URL", "sqlite:///app.db")
    # psycopg (version 3) is the driver installed, so say so.
    if url.startswith("postgresql://"):
        url = "postgresql+psycopg://" + url.removeprefix("postgresql://")
    return url

engine = create_engine(database_url())
# Say which database this is, with the password starred out.
logging.getLogger("uvicorn.error").info("Database: %r", engine.url)
```

Where it runs	Who sets `DATABASE_URL`
`uvicorn` in your codespace	nobody, so it's SQLite; or you, with `export`
a compose service	the service's `environment:`
Railway	the service's **Variables**

demos/railway-1-compose/src/main.py, the new part

A compose file describes both services, and one command starts them

```yaml
services:
  app:
    build: .
    ports:
      - "8000:8000"
    environment:
      DATABASE_URL: postgresql://e190bo:localdev@db:5432/app
    depends_on:
      db:
        condition: service_healthy
```

```
$ docker compose up --build
 Container …-db-1   Healthy
 Container …-app-1  Started
app-1 | INFO: Database: postgresql+psycopg://e190bo:***@db:5432/app
app-1 | INFO: Uvicorn running on http://0.0.0.0:8000
```

```yaml
  db:
    image: postgres:17
    environment:
      POSTGRES_USER: e190bo
      POSTGRES_PASSWORD: localdev
      POSTGRES_DB: app
    volumes:
      - pgdata:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U e190bo -d app"]
      interval: 2s
      retries: 15

volumes:
  pgdata:
```

demos/railway-1-compose/compose.yaml, in two halves

Inside compose, the database's host is its service name, never `localhost`

`postgresql://` `e190bo` `:` `localdev` `@` `db` `:` `5432` `/` `app`

scheme	user	password	host	port	database
which database	who's asking	the proof	**which machine**	Postgres's port	which database on it

With `@localhost:5432` instead, the app's own container answers, and nothing is listening there:

```
connection to server at "127.0.0.1", port 5432 failed: Connection refused
```

▶ LIVE The ideas now outlive the app's container, because they live in the database's volume

```
$ curl -X POST localhost:8000/ideas -H "Content-Type: application/json" \
    -d '{"title": "Desk fan that follows you"}'
{"id":1,"title":"Desk fan that follows you","body":"","vote_count":0}
$ docker compose down
 Container railway-1-compose-app-1  Removed
 Container railway-1-compose-db-1   Removed
$ docker compose up -d
$ curl localhost:8000/ideas
[{"body":"","id":1,"title":"Desk fan that follows you","vote_count":0}]
```

`docker compose down -v` would delete the volume, and every idea with it.

When `DATABASE_URL` is wrong, one line of the app's log says how

What's wrong: `app` is…	The line that gives it away
wrong password: `Exited (1)`	`FATAL:  password authentication failed for user "e190bo"`
wrong host: `Exited (1)`	`failed to resolve host 'nowhere': [Errno -2] Name or service not known`
`localhost`: `Exited (1)`	`connection to server at "127.0.0.1", port 5432 failed: Connection refused`
not set at all: `Up`	`Database: sqlite:///app.db`

The first three crash before Uvicorn starts, and the error is the last line of the traceback, above SQLAlchemy's link. The fourth runs happily on a file inside its own container, and loses every idea on the next deploy.

A `/health` that asks the database tells a platform the truth

```python
@app.get("/health")
def health():
    with engine.connect() as conn:
        conn.execute(text("SELECT 1"))
    return {"status": "ok"}
```

```
$ curl localhost:8000/health
{"status":"ok"}
$ docker compose stop db
$ curl localhost:8000/health
Internal Server Error
```

A platform only sends users to a new deploy once its health check answers 200. A `/health` that never touches the database only proves Python is running.

demos/railway-1-compose/src/main.py

A password committed once stays in git history, even after you delete it

```
$ git log --oneline
4c1e2a7 Remove the password
9b03f55 Add config
$ git log -p -S hunter2 | grep DATABASE_URL
-DATABASE_URL = "postgresql://postgres:hunter2@db.example.com:5432/app"
+DATABASE_URL = "postgresql://postgres:hunter2@db.example.com:5432/app"
```

If a real one ever lands in a commit, change the password, and treat the old one as public. A secret lives in the platform’s variables, and nowhere git can see.

Every line of the compose file has a counterpart on Railway

In `compose.yaml`	On Railway
`services:`	a **project**, drawn as a canvas
`app:`, `build: .`	a service from your **GitHub repo**, built from its `Dockerfile`
`db:`, `image: postgres:17`	a **Postgres** service, storage managed for you
`environment:`	the service's **Variables** tab
`postgresql://…@db:5432/app`	`${{ Postgres.DATABASE_URL }}`, at `postgres.railway.internal`
`ports: "8000:8000"`	**Generate Domain**, with `PORT` set for you
`docker compose up --build`	a deploy, on every push to `main`
`docker compose logs app`	the deploy's **build** and **deploy** logs

Step 1: Railway builds the `Dockerfile` in your repo, and the first deploy quietly runs on SQLite

1. **New Project** → **GitHub repo**. The first time, let Railway's GitHub app see your private repo.
2. Pick the repo → **Deploy Now**.

The build log, then the deploy log:

```
Using detected Dockerfile!
INFO:     Database: sqlite:///app.db
INFO:     Uvicorn running on http://0.0.0.0:8080
```

It's up, with no database: the fourth broken `DATABASE_URL`.

Step 2: Adding Postgres puts a second service on the canvas, like db in the compose file

1. On the project canvas: **+ New** → **Database** → **PostgreSQL**.
2. Open the new service's **Variables** tab.

Postgres sets	Means
`PGHOST`, `PGPORT`, `PGUSER`, `PGPASSWORD`, `PGDATABASE`	each part of the connection, on its own
`DATABASE_URL`	all of them in one URL, for other services on the private network
`DATABASE_PUBLIC_URL`	a URL from outside Railway, if you turn public access on

Step 3: A reference variable gives the app the database's URL without anyone copying the password

On **your app's** service → **Variables** → **New Variable**:

```
DATABASE_URL = ${{ Postgres.DATABASE_URL }}
```

Railway stages the change: **Deploy** to apply it. The new deploy log:

```
INFO:     Database: postgresql+psycopg://postgres:***@postgres.railway.internal:5432/railway
INFO:     Uvicorn running on http://0.0.0.0:8080
```

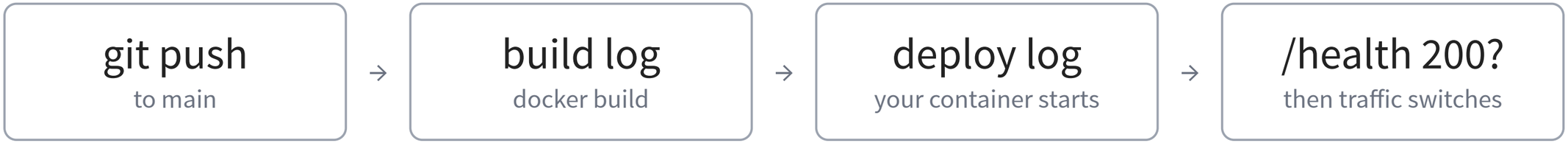
Step 4: A health check path and a generated domain make the deploy safe and public

On your app's service	Set	Compose had
Settings → Healthcheck Path	`/health`	`depends_on`, for the database only
Settings → Networking → Public Networking → Generate Domain	`https://idea-board-production.up.railway.app`	`ports: "8000:8000"`

```
$ curl https://idea-board-production.up.railway.app/health
{"status":"ok"}
```

Open it on your phone, on cell data: that's the internet.

Step 5: From now on, every push to `main` builds and deploys



A failure in the…	Looks like
build log	a typo in the `Dockerfile`; a package that won't install
deploy log	a bad variable; a crash at startup; `/health` never answering 200

A deploy that fails its health check never gets users, so read the failed deploy’s log

> “Railway will query the endpoint until it receives a successful response … Only then will the new deployment be made active and the previous deployment inactive.”
>
> — Railway’s healthcheck docs

Your URL keeps working, served by the **previous** deploy, and its log looks fine. The error is in the **failed** deploy’s log, after up to 5 minutes of retries.

Lab 6 deploys with steps 1, 4 and 5, and the rest is how a real app keeps its data

Step	In Lab 6	Beyond Lab 6
1: deploy the repo's `Dockerfile`	**yes**, from `e190bo-lab-06`	
2–3: Postgres, and a reference variable		a database that outlives every deploy
4: health check path		once `/health` asks the database
4: Generate Domain	**yes**: your live URL, on your phone	
5: every push to `main` deploys	**yes**	

Lab 6's app runs on SQLite inside its container, the quiet case from step 1, so its ideas reset on every deploy.