Full Stack with Docker Compose: Frontend, Backend and Database

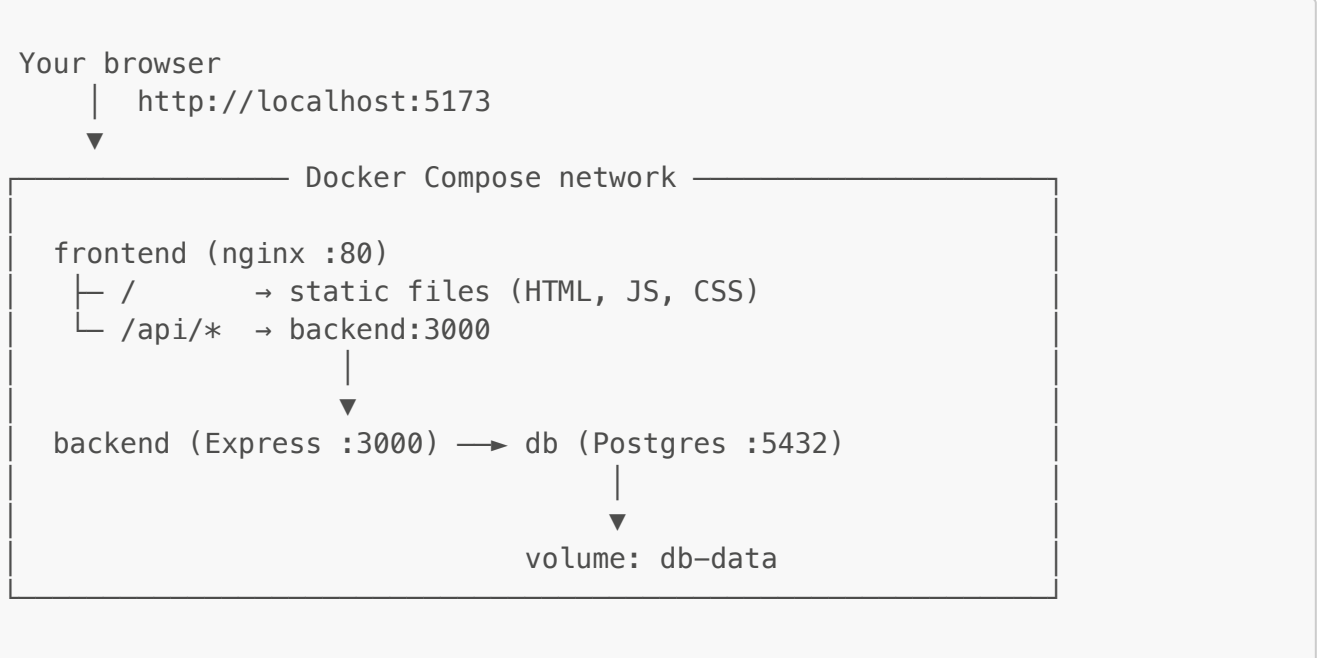
Final step. Until now, users lived in a browser variable and vanished on refresh. Now they go into a real PostgreSQL database.

A browser can't talk to a database directly (it would expose the password to everyone), so we need a backend: a small API between the two. Three services, three containers, all started with one command by Docker Compose.

Start from your `node-auth-app` folder from `DOCKER_GUIDE.md`.

---

1. The architecture



- Only the frontend is reachable from your machine. The backend and database are private to the Compose network.
- Containers find each other by service name: the backend connects to host `db`, nginx forwards to host `backend`.
- The database stores its files in a volume, so data survives when containers are deleted.

Final project:

```
node-auth-app/
├── docker-compose.yml        # NEW: starts the 3 services
├── nginx.conf                # NEW: nginx config with /api forwarding
├── Dockerfile.multistage     # updated: copies nginx.conf
├── .dockerignore             # updated
├── src/
│   ├── api.js                # NEW: calls the backend (replaces auth.js)
```

```
│   ├── main.js               # updated
│   └── style.css
├── tests/auth.spec.js        # updated: unique usernames
└── backend/                  # NEW
    ├── Dockerfile
    ├── .dockerignore
    ├── package.json
    └── src/
        ├── server.js         # HTTP routes
        ├── auth.js           # validation, hashing, SQL
        └── db.js             # database connection
```

Check Docker Compose is available (it ships with Docker Desktop):

```
docker compose version
```

Part 1: Create the backend

2. Set up the project

From the `node-auth-app` folder:

```
mkdir -p backend/src
cd backend
npm init -y
npm pkg set type=module scripts.start="node src/server.js"
npm install express pg
cd ..
```

- Express: the most common Node web framework. Maps URLs to functions.
- pg: the PostgreSQL client for Node.
- `type=module` lets us use `import` instead of `require`.
- `npm install` also creates `package-lock.json`, needed by `npm ci` in the Dockerfile.

The important parts of `backend/package.json` (npm adds a few extra fields; they don't matter):

```
{
  "name": "auth-backend",
  "version": "1.0.0",
  "private": true,
  "type": "module",
  "scripts": {
    "start": "node src/server.js"
  },
```

```
  "dependencies": {
    "express": "^5.3.0",
    "pg": "^8.23.0"
  }
}
```

3. The database connection: `backend/src/db.js`

```
// Database connection. The address comes from an environment variable,
// so the same code works locally and inside Docker Compose.
import pg from "pg";

export const pool = new pg.Pool({ connectionString:
process.env.DATABASE_URL });

// If the database restarts, idle connections break. Log it instead of
crashing:
// the pool opens fresh connections on the next query.
pool.on("error", (err) => console.error("Database connection lost:",
err.message));

// Create the table on startup if it doesn't exist yet.
export async function initDb() {
  await pool.query(`
    CREATE TABLE IF NOT EXISTS users (
      username      TEXT PRIMARY KEY,
      password_hash TEXT NOT NULL,
      created_at    TIMESTAMPTZ NOT NULL DEFAULT now()
    )
  `);
}
```

- `DATABASE_URL`: one string with everything needed to connect: `postgres://user:password@host:port/database`. It comes from the environment, so the code contains no passwords and works anywhere.
- `Pool`: keeps a few connections open and reuses them, faster than connecting on every request.
- `initDb`: creates the `users` table on first start. `IF NOT EXISTS` makes it safe to run every time. `PRIMARY KEY` on `username` makes duplicates impossible at the database level.

4. The logic: `backend/src/auth.js`

Same rules as the frontend version, now on the server, now saving to the database:

```
// Business logic: same rules as before, but users are now stored in
PostgreSQL.
import { pbkdf2, randomBytes, timingSafeEqual } from "node:crypto";
import { promisify } from "node:util";
import { pool } from "./db.js";
```

```
const pbkdf2Async = promisify(pbkdf2);
const USERNAME_RE = /^[a-z0-9_]{3,20}$/;
const MIN_PASSWORD_LENGTH = 8;

export class AuthError extends Error {}

// Returns "salt$hash" (both hex) using PBKDF2-SHA256.
async function hashPassword(password, salt = randomBytes(16)) {
  const hash = await pbkdf2Async(password, salt, 100_000, 32, "sha256");
  return `${salt.toString("hex")}$${hash.toString("hex")}`;
}

async function verifyPassword(password, stored) {
  const [saltHex] = stored.split("$");
  const candidate = await hashPassword(password, Buffer.from(saltHex,
"hex"));
  return timingSafeEqual(Buffer.from(candidate), Buffer.from(stored));
}

function validate(username, password) {
  if (!USERNAME_RE.test(username)) {
    throw new AuthError("Username must be 3-20 characters: letters, digits
or underscore");
  }
  if (password.length < MIN_PASSWORD_LENGTH) {
    throw new AuthError(`Password must be at least ${MIN_PASSWORD_LENGTH}
characters`);
  }
  if (!/\d/.test(password)) {
    throw new AuthError("Password must contain at least one digit");
  }
}

const normalize = (username) => String(username).trim().toLowerCase();

export async function register(rawUsername, password) {
  const username = normalize(rawUsername);
  validate(username, String(password));
  try {
    // $1, $2 are parameters: never build SQL by pasting user input into
the string.
    await pool.query("INSERT INTO users (username, password_hash) VALUES
($1, $2)", [
      username,
      await hashPassword(String(password)),
    ]);
  } catch (err) {
    if (err.code === "23505") throw new AuthError("Username already
taken"); // unique violation
    throw err;
  }
  return username;
}
```

```
// Returns the username on success, null on wrong credentials.
export async function login(rawUsername, password) {
  const username = normalize(rawUsername);
  const { rows } = await pool.query("SELECT password_hash FROM users WHERE
username = $1", [username]);
  if (rows.length === 0) return null;
  return (await verifyPassword(String(password), rows[0].password_hash)) ?
username : null;
}
```

- `$1, $2`: query parameters. Postgres receives the values separately from the SQL, which prevents SQL injection. Never write `` `... WHERE username = '${username}'` ``.
- `23505`: the Postgres error code for "unique constraint violated". We let the database detect duplicates, then translate the error for the user.
- `timingSafeEqual`: compares hashes in constant time, so attackers can't guess them from response times.
- Passwords are hashed on the server. The database never sees a plain password.

5. The API: `backend/src/server.js`

```
// HTTP layer: a small JSON API. Business logic lives in auth.js.
import express from "express";
import { AuthError, login, register } from "./auth.js";
import { initDb, pool } from "./db.js";

const app = express();
app.use(express.json()); // parse JSON request bodies into req.body

// Is the backend up, and can it reach the database?
app.get("/api/health", async (req, res) => {
  await pool.query("SELECT 1");
  res.json({ status: "ok" });
});

app.post("/api/register", async (req, res) => {
  const { username = "", password = "" } = req.body ?? {};
  try {
    const name = await register(username, password);
    res.status(201).json({ username: name });
  } catch (err) {
    if (err instanceof AuthError) return res.status(400).json({ error:
err.message });
    throw err; // unexpected: Express answers 500
  }
});

app.post("/api/login", async (req, res) => {
  const { username = "", password = "" } = req.body ?? {};
  const name = await login(username, password);
```

```
  if (!name) return res.status(401).json({ error: "Invalid username or
password" });
  res.json({ username: name });
});

// Any error not handled above ends up here: answer in JSON, never leak a
stack trace.
app.use((err, req, res, next) => {
  if (err.type === "entity.parse.failed") return res.status(400).json({
error: "Invalid JSON" });
  console.error(err);
  res.status(500).json({ error: "Internal server error" });
});

const PORT = Number(process.env.PORT ?? 3000);

await initDb();
const server = app.listen(PORT, () => console.log(`Backend listening on
port ${PORT}`));

// `docker compose down` sends SIGTERM: close cleanly instead of waiting to
be killed.
process.on("SIGTERM", () => {
  server.close(() => pool.end().then(() => process.exit(0)));
});
```

Endpoint	Body	Success	Failure
`GET /api/health`		`200 {status: "ok"}`	`500` if the database is unreachable
`POST /api/register`	`{username, password}`	`201 {username}`	`400 {error}`
`POST /api/login`	`{username, password}`	`200 {username}`	`401 {error}`

- Status codes carry meaning: `201` created, `400` bad input, `401` wrong credentials, `500` server problem. The frontend relies on them.
- In Express 5, an error thrown in an `async` route goes to the error handler automatically.
- The error handler at the end guarantees the API always answers in JSON and never shows internal details to users.

6. The backend image: `backend/Dockerfile`

```
FROM node:26-alpine

WORKDIR /app

COPY package.json package-lock.json ./
RUN npm ci --omit=dev          # production dependencies only
```

```
COPY src ./src

ENV NODE_ENV=production
EXPOSE 3000

# Run node directly (not via npm) so it receives the stop signal from
Docker.
CMD ["node", "src/server.js"]
```

`backend/.dockerignore`:

```
node_modules
.dockerignore
Dockerfile
```

- `npm ci --omit=dev`: install only what's needed to run, not to develop.
- `COPY src ./src`: the backend has no build step, the source *is* the app.
- `CMD ["node", ...]` instead of `npm start`: Node gets Docker's stop signal directly, so `server.js` can shut down cleanly.

Part 2: Connect the frontend

7. Replace the in-memory store: `src/api.js`

Delete `src/auth.js` (validation and hashing now live in the backend) and create `src/api.js`:

```
// Talks to the backend. Replaces the in-memory UserStore from auth.js.
// Paths start with /api: nginx forwards those requests to the backend
container.

export class ApiError extends Error {}

async function post(path, body) {
  let res;
  try {
    res = await fetch(`/api${path}`, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(body),
    });
  } catch {
    throw new ApiError("Server unavailable, please try again"); // no
network at all
  }
  // 5xx: backend crashed, or nginx can't reach it (502 Bad Gateway).
  if (res.status >= 500) throw new ApiError("Server unavailable, please try
```

```
again");
  const data = await res.json().catch(() => ({}));
  return { ok: res.ok, status: res.status, data };
}

// Returns the normalised username, or throws ApiError with the server's
message.
export async function register(username, password) {
  const res = await post("/register", { username, password });
  if (!res.ok) throw new ApiError(res.data.error ?? "Registration failed");
  return res.data.username;
}

// Returns the username on success, null on wrong credentials.
export async function login(username, password) {
  const res = await post("/login", { username, password });
  if (res.ok) return res.data.username;
  if (res.status === 401) return null;
  throw new ApiError(res.data.error ?? "Login failed");
}
```

- `fetch("/api/...")`: a relative URL, so the request goes to the same server that served the page (nginx), which forwards it to the backend. No CORS setup needed.
- The frontend no longer validates or hashes: it shows the server's error message. The server is the single source of truth.

8. Update `src/main.js`

Replace the top of the file:

```
// UI layer: renders the pages and reacts to the forms. Users are stored by
the backend.
import "./style.css";
import * as api from "./api.js";

// UI state only. Users now live in the database, behind the backend API.
let currentUser = null; // username once logged in
let flash = null;       // one-time message: { type: "error" | "success",
text }
```

Replace the whole `onSubmit` function:

```
async function onSubmit(event, mode) {
  event.preventDefault();
  const form = event.target;
  const { username, password } = Object.fromEntries(new FormData(form));
  form.querySelector("button").disabled = true; // wait for the server

  try {
```

```
      if (mode === "register") {
        const name = await api.register(username, password);
        flash = { type: "success", text: `Account created for ${name}. Please
log in.` };
        return go("/login");
      }
      const name = await api.login(username, password);
      if (name) {
        currentUser = name;
        return go("/welcome");
      }
      flash = { type: "error", text: "Invalid username or password" };
    } catch (err) {
      if (!(err instanceof api.ApiError)) throw err;
      flash = { type: "error", text: err.message };
    }
    render();
  }
```

The rest of `main.js` (rendering, navigation) is unchanged.

9. Forward `/api` to the backend: `nginx.conf`

Create `nginx.conf` in the project root:

```
server {
    listen 80;

    # The built frontend (index.html, assets/)
    root /usr/share/nginx/html;
    index index.html;

    location / {
        try_files $uri $uri/ /index.html;
    }

    # API calls go to the backend container. "backend" is its service name
    # in docker-compose.yml: Compose makes it resolvable as a hostname.
    location /api/ {
        proxy_pass http://backend:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

`proxy_pass` makes nginx a reverse proxy: the browser only talks to nginx, nginx talks to the backend on its behalf.

Update `Dockerfile.multistage` to use it:

```
# ---------- Stage 1: build the static files ----------
FROM node:26-alpine AS builder

WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci
COPY . .
RUN npm run build            # outputs plain HTML/CSS/JS to /app/dist

# ---------- Stage 2: serve them with nginx ----------
FROM nginx:stable-alpine

# Our config: serve the files and forward /api/ to the backend.
COPY nginx.conf /etc/nginx/conf.d/default.conf

# Only the built files are copied. Node, npm and node_modules stay behind.
COPY --from=builder /app/dist /usr/share/nginx/html

EXPOSE 80
# nginx's default command starts the server; no CMD needed.
```

10. Keep the frontend image clean: `.dockerignore`

The frontend image is built from the project root, which now contains `backend/`. Exclude it, along with the test files:

```
# Never copy these into the frontend image: they are rebuilt inside it, or
not needed.
node_modules
dist
test-results
playwright-report
tests
playwright.config.js
backend
docker-compose.yml
.git
.env
*.md
Dockerfile*
.dockerignore
```

Part 3: Docker Compose

11. Describe the stack: `docker-compose.yml`

Create it in the project root:

```yaml
# Three containers that work together: database, backend API, frontend.
# Start everything:  docker compose up --build
services:
  db:
    image: postgres:18-alpine
    environment:
      POSTGRES_USER: app
      POSTGRES_PASSWORD: app_password # fine for local learning, never for production
      POSTGRES_DB: auth
    volumes:
      - db-data:/var/lib/postgresql # named volume: data survives container restarts
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U app -d auth"]
      interval: 5s
      timeout: 3s
      retries: 10

  backend:
    build: ./backend
    environment:
      DATABASE_URL: postgres://app:app_password@db:5432/auth
      PORT: 3000
    depends_on:
      db:
        condition: service_healthy # wait until Postgres accepts connections

  frontend:
    build:
      context: .
      dockerfile: Dockerfile.multistage
    ports:
      - "5173:80" # host:container. Only the frontend is reachable from your machine.
    depends_on:
      - backend

volumes:
  db-data:
```

Key	Meaning
`services`	One entry per container. The name (`db`, `backend`, `frontend`) is also its hostname on the network.
`image`	Use a ready-made image from Docker Hub (Postgres).

Key	Meaning
`build`	Build an image from a Dockerfile: a folder (`./backend`), or a `context` + `dockerfile`.
`environment`	Environment variables. The official Postgres image creates the user and database from them on first start.
`volumes` (in a service)	Mount the named volume `db-data` where Postgres keeps its files.
`healthcheck`	How Docker knows Postgres is *ready*, not just *started*. `pg_isready` is a tool inside the Postgres image.
`depends_on`	Start order. `condition: service_healthy` waits for the healthcheck to pass, so the backend never connects too early.
`ports`	Publish to your machine, `host:container`. Only the frontend has one.
`volumes` (top level)	Declares the named volume. Docker creates and manages it.

> Postgres 18 note: the volume is mounted at `/var/lib/postgresql`. Older tutorials use `/var/lib/postgresql/data`, which is only correct for Postgres 17 and older.

> Passwords are written in plain text here to keep things simple. Real projects keep them in a `.env` file (never committed) or a secrets manager. See exercise 1.

12. Start everything

```
docker compose up --build
```

- `up`: create and start all services, in dependency order: `db` → (healthy) → `backend` → `frontend`.
- `--build`: (re)build the `backend` and `frontend` images first. Use it after any code change.

Logs from the three containers are mixed, each line prefixed with the service name. Wait for:

```
backend-1   | Backend listening on port 3000
```

Open http://localhost:5173. Stop everything with `Ctrl+C`.

To run in the background instead:

```
docker compose up --build -d     # detached
docker compose ps                # status of each service
docker compose logs -f backend   # follow one service's logs (Ctrl+C to
stop following)
```

13. Confirm it works

In the browser: register `alice` / `secret123`, log in, log out. Try a short password and a duplicate username: the error messages now come from the backend.

Ask the backend directly (from inside its container, since its port isn't published):

```
docker compose exec backend wget -qO- http://localhost:3000/api/health
```

`exec` runs a command inside a running service. Expect `{"status":"ok"}`.

Look inside the database:

```
docker compose exec db psql -U app -d auth -c "SELECT username, created_at
FROM users;"
```

`alice` is there, and if you also select `password_hash`, you'll see only a salted hash.

Run the Playwright tests. First, update `tests/auth.spec.js`: the database now *remembers* users between runs, so registering `alice` would fail the second time with *Username already taken*. Each test now registers a unique user. Replace the file with:

```
import { test, expect } from "@playwright/test";

// The database keeps users between runs, so "alice" would already exist
the
// second time. Each test registers a new, unique username instead.
const uniqueName = () => `user_${Math.random().toString(36).slice(2, 10)}`;

// Helper: fill the form on the current page and submit it.
async function submitForm(page, username, password, button) {
  await page.getByLabel("Username").fill(username);
  await page.getByLabel("Password").fill(password);
  await page.getByRole("button", { name: button }).click();
}

let username;

test.beforeEach(async ({ page }) => {
  username = uniqueName();
  await page.goto("/#/register");
  await submitForm(page, username, "secret123", "Register");
  await expect(page.getByRole("alert")).toHaveText(`Account created for
${username}. Please log in.`);
});

// Happy path
test("registered user can log in", async ({ page }) => {
```

```
  await submitForm(page, username, "secret123", "Log in");

  await expect(page).toHaveURL(/#\/welcome$/);
  await expect(page.getByRole("heading", { name: `Welcome, ${username}!`
})).toBeVisible();
});

// Sad path
test("login with invalid credentials shows an error", async ({ page }) => {
  await submitForm(page, username, "wrong-pass1", "Log in");

  await expect(page.getByRole("alert")).toHaveText("Invalid username or
password");
  await expect(page).toHaveURL(/#\/login$/);
  await expect(page.getByRole("heading", { name: "Welcome"
})).toBeHidden();
});
```

Then, with the stack running:

```
npx playwright test
```

Expect `2 passed`. Run it a second time: still passes.

14. Persistence: the volume in action

Command	Containers	Volume (data)
`docker compose stop`	stopped, kept	kept
`docker compose down`	**deleted**	kept
`docker compose down -v`	deleted	**deleted**

Try it:

```
docker compose down             # delete all containers
docker compose up -d            # brand new containers
```

Log in as `alice`: **it works**. The containers are new, but the database files were in the `db-data` volume.

```
docker compose down -v          # also delete the volume
docker compose up -d
```

Log in as `alice`: *Invalid username or password*. The data is gone.

```
docker volume ls                # list volumes; yours is named
<folder>_db-data
```

Command summary

Command	Purpose
`docker compose up --build`	Build and start everything
`docker compose up -d`	Start in the background
`docker compose ps`	Show services and their status
`docker compose logs -f <service>`	Follow a service's logs
`docker compose exec <service> <cmd>`	Run a command inside a service
`docker compose restart <service>`	Restart one service
`docker compose down`	Stop and delete containers (keeps data)
`docker compose down -v`	Same, and delete volumes (wipes data)

---

15. Exercises

1. Secrets out of the file. Create a `.env` file with `POSTGRES_PASSWORD=...`, and use `${POSTGRES_PASSWORD}` in `docker-compose.yml` (for both `db` and the `DATABASE_URL`). Compose reads `.env` automatically. Add `.env` to `.gitignore` and `.dockerignore` so it stays out of Git and Docker builds.
2. Resilience. With the stack running, `docker compose restart db`. Log in during and after the restart. What does the user see? Check `docker compose logs backend`.
3. Connect a database tool. Add `ports: ["5432:5432"]` to `db` and connect with a client like DBeaver or pgAdmin. Why is it safer to leave this port unpublished in production?
4. See the first-run trap. Change the password in both places in `docker-compose.yml` (`POSTGRES_PASSWORD` and `DATABASE_URL`), then `docker compose up -d`. The backend can't connect. Why? (Hint: Postgres only reads these variables when the volume is empty.) How do you fix it?
5. New endpoint. Add `GET /api/users/count` returning the number of registered users, and show it on the welcome page.