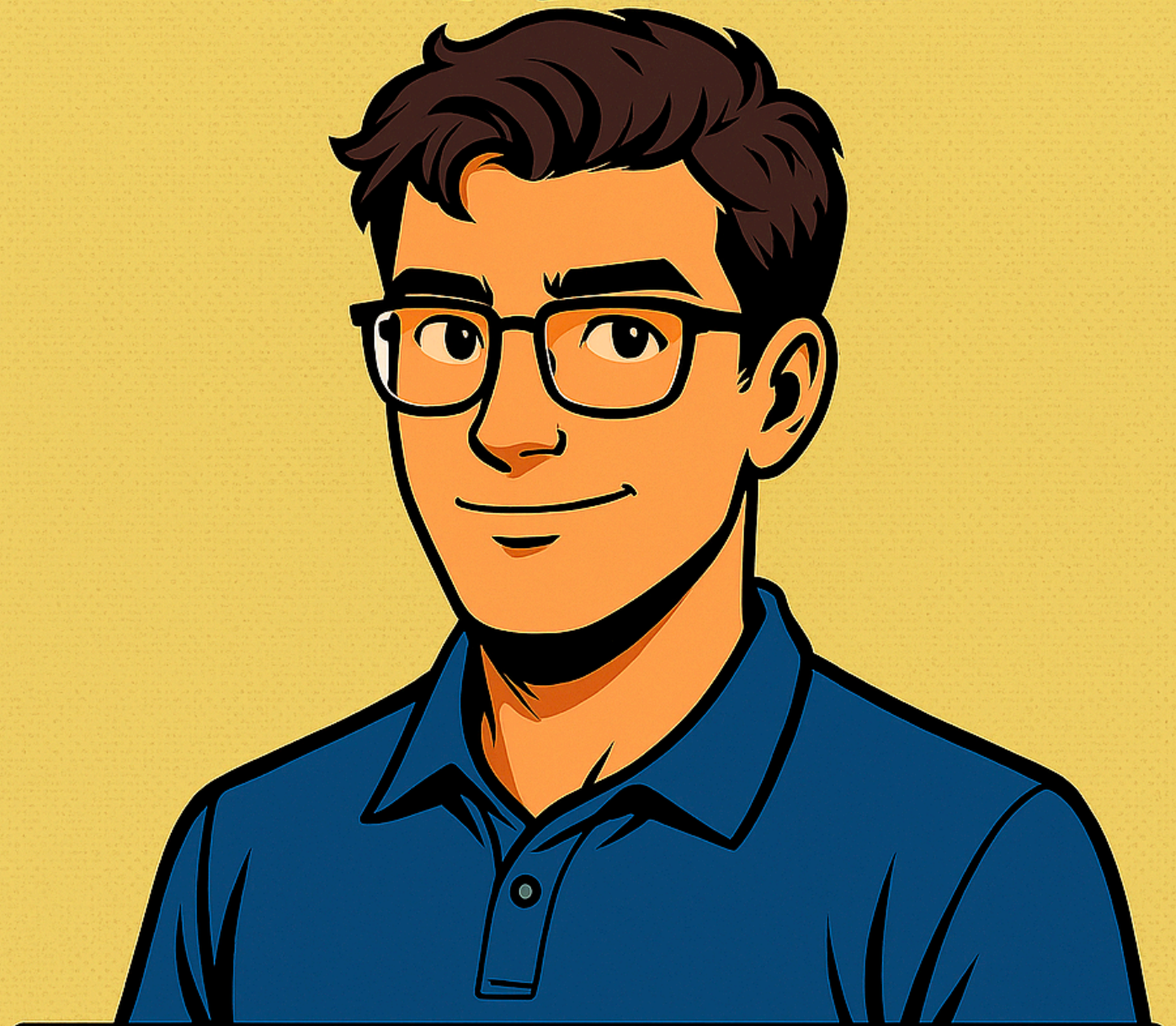
DOCKER PATH



FROM FRONTEND DEVELOPER TO INFRASTRUCTURE ARCHITECT

MIT License

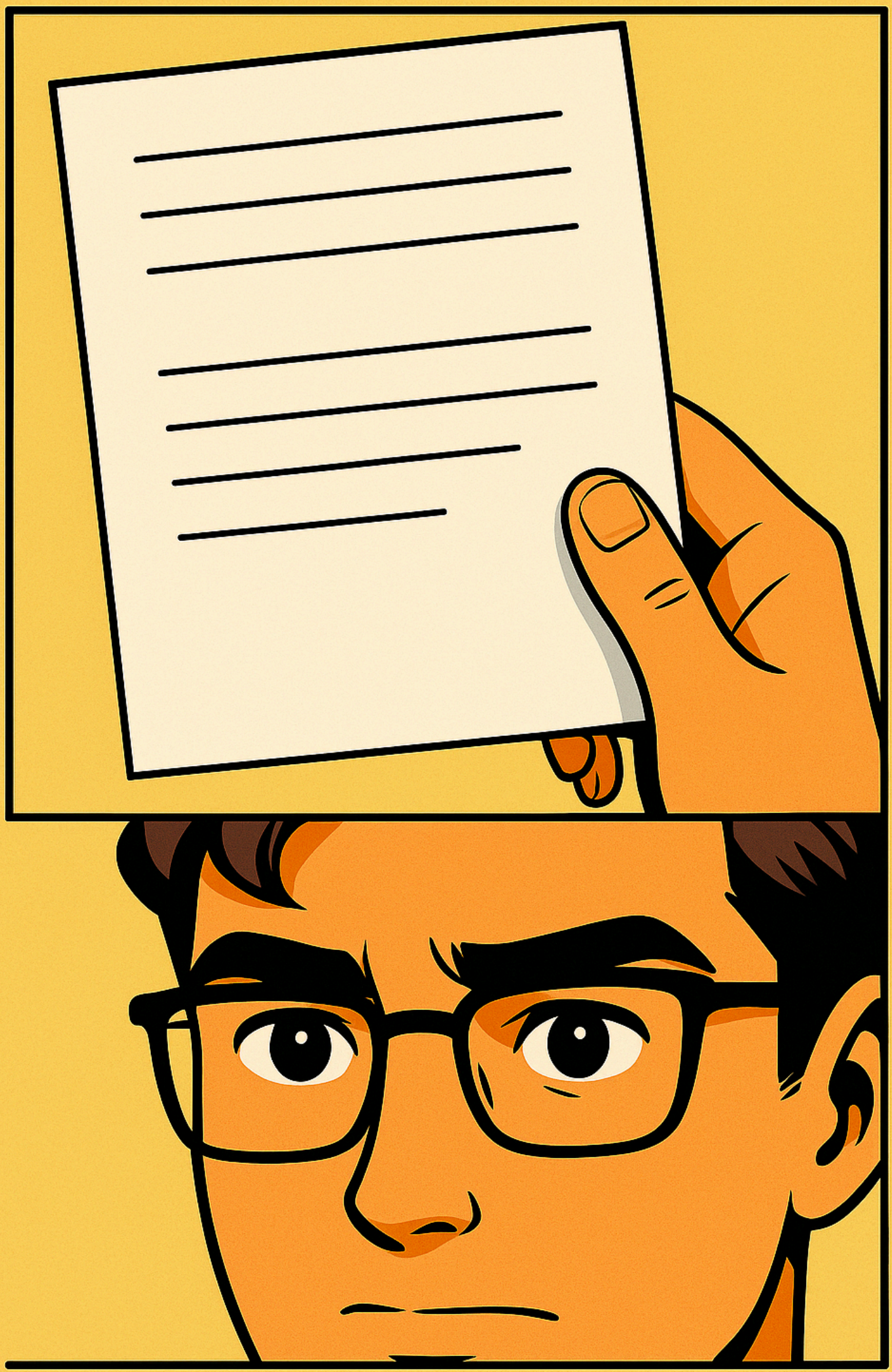
01-First-Contact: Your First Container

You've just joined Innovatech Labs as a junior DevOps engineer. On your first morning, you receive a terse email from your new mentor, Dr. A. Sharma, head of Infrastructure Engineering.

TO: Alex
FROM: Dr. A. Sharma
SUBJECT: First Contact

Welcome to Project Phoenix. Before you touch production systems, you must prove you understand the foundation of everything we build — containers.

Begin with something simple. Launch a single, isolated service. Observe how it behaves. Learn its nature.



What You Will Learn

In this first crucial step, you will learn the absolute basics of containerization. Specifically, you will learn how to pull a Docker image, run it as a container, and understand basic port mapping. This foundational knowledge is vital for everything that follows.

Your Task: Launching Your First Service

Your first task is to get acquainted with Docker by launching a simple web server using an Nginx container. This will confirm your Docker environment is set up correctly and introduce you to the core concepts of Docker images and containers.

Here's what you need to do:

1. Pull the Nginx image: Open your terminal or command prompt and use the Docker CLI to pull the official nginx image from Docker Hub. This image contains everything needed to run an Nginx web server.
   docker pull nginx
2. Run the Nginx container: Execute the nginx image as a container. You'll need to run it in detached mode (meaning it runs in the background) and map a port from your computer to the container's port so you can access the web server.
   - Use the --detach (or -d) flag to run the container in the background.
   - Use the --publish (or -p) flag to map port 8080 on your local machine to port 80 inside the container.
   - Give your container a memorable name using --name my-first-container.
   docker run --detach --publish 8080:80 --name my-first-container nginx
3. Verify container execution:
   - Open your web browser and navigate to http://localhost:8080. You should see the default Nginx welcome page. This confirms your container is running and accessible!
   - You can also use docker ps in your terminal to see a list of running containers and confirm my-first-container is listed.
4. To verify correct execution, run the verify.sh script in the lab work folder.

Core Concepts Explained

In this first crucial step, you will learn the absolute basics of containerization. Specifically, you will learn how to pull a Docker image, run it as a container, and understand basic port mapping. This foundational knowledge is vital for everything that follows.

- Docker Hub: Think of Docker Hub as a public library for Docker images. When you docker pull nginx, you're downloading the Nginx blueprint from this library.
- Image vs. Container: The nginx image is like a blueprint or a recipe for a web server. When you docker run it, you're creating an active, running instance of that blueprint – a container. You can run multiple containers from the same image.
- Port Mapping (-p 8080:80): This is crucial for accessing services inside a container from your host machine. The first number (8080) is the port on your computer (the host), and the second number (80) is the port inside the container where Nginx is listening. This mapping allows traffic coming to http://localhost:8080 to be directed to the Nginx server running on port 80 inside the container.
- Detached Mode (--detach or -d): Running a container in detached mode means it operates in the background, freeing up your terminal for other commands. Without it, your terminal would be attached to the container's output, and you wouldn't be able to type new commands.

Next Steps

Congratulations on launching your first container! In the next exercise, 02-The-Ships-Log, you will learn how to inspect and manage container logs, which is essential for monitoring and debugging your containerized applications.


❯:ǂ%#...
❯·$$@!...
❯·_*%@^...

02: The Ship's Log (Mastering Control)

TO: Alex
FROM: Dr. A. Sharma
SUBJECT: Re: First Contact

Acceptable. You can follow instructions. But creation is easy. Control is harder. An uncontrolled tool is a liability. A server you cannot stop or inspect is a rogue element.

Your next task is to learn to be the master of the asset you just deployed. Start it. Stop it. Listen to what it's telling you by reading its logs. And finally, learn how to decommission it cleanly.

Do not proceed until you have full control.

What You Will Learn

Your mission is to learn the commands that manage a container's lifecycle, using the my-first-container you deployed in the last step.

- How to list running and stopped containers.
- How to stop a running container.
- How to start a stopped container
- How to view the logs of a container.
- How to remove a container.

Here's what you need to do:

Dr. Sharma expects you to have complete mastery. Practice these commands on the my-first-container.

1. Conduct a Status Check: First, see what's currently running. The docker ps command is your control panel.
2. Review the Ship's Log: Every action on the server is logged. You need to know how to read these logs. Use the docker logs command.
3. Cease Operations: A critical skill is knowing how to stop a service. Use the docker stop command. Verify that http://localhost:8080 is now offline.
4. Account for All Assets: The container is stopped, but it still exists. Use the --all flag to see all assets, running or not. docker ps --all
5. Re-activate the Service: An operator must also know how to restart a service. Use docker start. Verify that http://localhost:8080 is back online.
6. Decommission the Asset: For this training exercise, the final step is to decommission the container. This requires stopping it and then removing it permanently with docker rm.
7. To verify correct execution, run the verify.sh script in the lab work folder.


You've just finished familiarizing yourself with the basics of container management when a message from your colleague, Ben, pops up.
new!

TO: Alex
FROM: Ben
SUBJECT: The Blueprint

Hey Alex, I heard you're the one getting into all that Docker stuff. Listen, I'm in a bit of a jam. I built this simple Node.js app to serve as a style guide for the new intern, but I can't get it to run on their machine without a bunch of node version errors. It works on my machine, of course. Any chance your 'Docker magic' can help? I just need a way to package this app so it runs anywhere, no excuses.

What You Will Learn

This is your first real test. Dr. Sharma's exercises are one thing, but solving a real problem for a colleague is another. You realize this is the perfect opportunity to move from being a consumer of images to a creator.

- How to solve the "it works on my machine" problem.
- The basic syntax of a Dockerfile (FROM, WORKDIR, COPY, RUN, CMD).
- How to build a Docker image from a Dockerfile using docker build.
- How to run a container from your own custom image.

Here's what you need to do:

Ben has sent you the code, which you can find in the app directory. It's a very simple Node.js application. Your task is to create a Dockerfile inside that same app directory to containerize it.

1. Create the Dockerfile: In the app directory, create a new file named Dockerfile (no file extension). This file will be the blueprint for your image.
2. Write the Blueprint: Add the following content to your Dockerfile. Each line is an instruction that tells Docker how to build the image, step-by-step.

Dockerfile

```
# 1. Every great project starts from a solid foundation. We'll use an official Node.js base image.
FROM node:16-alpine

# 2. Let's create a dedicated workshop inside the container for our app.
WORKDIR /usr/src/app

# 3. To be efficient, we'll copy the manifest first. This lets Docker cache our dependencies.
COPY package*.json ./

# 4. Now, we run the installer. This is like setting up your local project.
RUN npm install

# 5. With the workshop set up, we can bring in the rest of the source code.
COPY . .

# 6. Finally, we tell the container what to do when it starts.
CMD [ "node", "server.js" ]
```

3. Build the Image: Time to turn your blueprint into a reality. From your terminal, navigate to the 03-The-Blueprint directory. Use the docker build command to create the image. You'll tag it with a name so you can easily refer to it later. Let's name it something Ben will recognize. docker build -t ben-style-guide ./app

4. Run the Containerized App: You've built the image! Now, run it as a container to make sure it works. docker run --detach --publish 8080:8080 --name style-guide-app ben-style-guide
5. Verify It Works: Open your web browser and navigate to http://localhost:8080. You should see the message: "Hello from inside the container! This is our first custom image."

   It works! You can now confidently go back to Ben and tell him you've solved his problem.
6. To verify correct execution, run the verify.sh script in the lab work folder.

A message from Dr. Sharma arrives as you're about to message Ben.

TO: Alex, Ben
FROM: Dr. A. Sharma
SUBJECT: Re: The Blueprint

I saw you helped Ben. Good initiative. Collaboration is key.

Now, back to Project Phoenix. Our new backend will need a database. A database without data is useless. Your next task is to create a database container where the data persists. The soul of the application must not be ephemeral.

04: The Soul of the Application (Using Volumes)

What You Will Learn

Your mission is to run a postgres database container and use a Docker Volume to ensure its data is persistent.

- The concept of stateless vs. stateful containers.
- What Docker Volumes are and why they are the key to data persistence.
- How to create and manage a named volume (docker volume create).
- How to attach a volume to a container to safeguard its data.

Here's what you need to do:

Dr. Sharma expects you to have complete mastery. Practice these commands on the my-first-container.

1. Dr. Sharma's message is clear: the data is everything. You need to create a volume and attach it to a Postgres container.
   Forge the Soul-Safe (Create a Named Volume): You will create a "named" volume. Think of it as a secure, managed vault for your data that lives outside the container. docker volume create postgres-data
2. Launch the Database Guardian: Run the official postgres container. You must provide a password for the database user and, most importantly, connect your volume to the container's data directory. --mount source=postgres-data,target=/var/lib/postgresql/data: This command tells the container "Store your critical data not inside yourself, but in the postgres-data vault." docker run --detach --name my-database -e POSTGRES_PASSWORD=mysecretpassword --mount source=postgres-data,target=/var/lib/postgresql/data postgres
3. Simulate a Catastrophe: The container is running. Let's test your setup. What if someone accidentally removes the container?
   docker stop my-database
   docker rm my-database
   The container is gone. In a normal setup, all data would be lost.
4. Resurrect the Guardian: Launch a new container with the exact same `docker run` command from Step 2. Because the command points to the same vault (postgres-data), the new container will pick up right where the old one left off, with all the data intact. docker run --detach --name my-database -e POSTGRES_PASSWORD=mysecretpassword --mount source=postgres-data,target=/var/lib/postgresql/data postgres
5. To verify correct execution, run the verify.sh script in the lab work folder.

05: The Local Workshop (A Developer's Insight)

You've successfully created a persistent database, a critical step for Project Phoenix. But something has been bothering you. When you helped Ben back in Step 03, you had to rebuild the entire Docker image every time you made a small code change. It was slow and clunky.
"There has to be a better way," you think to yourself. "How can I work on my code locally and see the changes inside the container instantly?"

After some research, you discover the answer: a **Bind Mount**. Unlike a volume, which is managed by Docker, a bind mount maps a directory from your own computer directly into the container. It's like creating a live portal between your code editor and the running container.

What You Will Learn

Your mission: To prove this concept, you decide to set up a local development environment for a Node.js app that features instant, live reloading.

- The difference between a Docker Volume (for data) and a Bind Mount (for code).
- How to use a bind mount to create an efficient development loop.
- How a file-watching tool like nodemon complements a bind mount.

Here's what you need to do:

You decide to experiment with the app you built for Ben.

1. Upgrade the Toolkit (nodemon): You realize that just seeing the files change isn't enough; the server process inside the container needs to restart. You find a tool called nodemon that does exactly this. You'll add it to the project as a development dependency. In your terminal, navigate to the 05-The-Local-Workshop/app directory.
   cd app
   npm install --save-dev nodemon
   cd ..
   Then, you'll add a dev script to the package.json to run it.
   "scripts": {
   "start": "node server.js",
   "dev": "nodemon server.js"
   },
2. Create a Development Blueprint:
   FROM node:16-alpine
   WORKDIR /usr/src/app
   COPY package*.json ./
   RUN npm install
   COPY . .
   # The key is to run the dev script!
   CMD ["npm", "run", "dev"]
   Now, build the development-focused image from the 05-The-Local-Workshop directory: docker build -t dev-image ./app
3. Open the Portal (Run with a Bind Mount): This is your "Aha!" moment. You'll run the container, but this time you'll use --mount with type=bind to create a live link between your local app directory and the /usr/src/app directory inside the container. docker run --detach --name dev-app --publish 8080:8080 --mount type=bind,source=$(pwd)/app,target=/usr/src/app dev-image

3. Witness the Magic:
   - Visit http://localhost:8080. It shows the original message.
   - Now, open app/server.js in your code editor and change the message inside res.send() to "This is so much faster!"
   - Save the file.
   - Refresh your browser. The change is there instantly. No rebuild. No waiting.
4. Run the verification script to confirm that your live-reloading environment is set up correctly. ./verify.sh

This discovery is a game-changer for your workflow. You make a note to share it with Ben later. Just then, a new memo from Dr. Sharma arrives, pulling you back to the main mission for Project Phoenix.

06: Building Bridges (Container Networking)

TO: Alex
FROM: Dr. A. Sharma
SUBJECT: Architecture

The pieces exist: your application, the database. But they are islands, isolated from each other. An application is a conversation. A frontend must talk to a backend; a backend must talk to a database. By default, my containers provide isolation. This is a feature, not a bug. It is your job to build the bridges between them in a controlled manner.
Your next task is to create a dedicated virtual network and place both your application and database containers on it. They must be able to communicate.

What You Will Learn

Your mission is to make the app and database containers talk to each other by connecting them to the same Docker network.

- The basics of Docker networking.
- How to create a custom bridge network.
- How to attach containers to a network at runtime.
- How Docker provides automatic DNS resolution for containers on the same network.

Here's what you need to do:

Dr. Sharma has given you the blueprint. Now you need to build the bridge.

1. Survey the Land (Create a Docker Network): First, you need to lay the foundation for your bridge. Create a new network for your application. docker network create my-app-net
2. Deploy the First Pier (The Database): Run the postgres container. Crucially, use the --network flag to attach it to the network you just created. Give it a name that your app can use to find it. docker run --detach --name my-database --network my-app-net -e POSTGRES_PASSWORD=mysecretpassword postgres
3. Prepare the Second Pier (The Application): The app directory contains a new version of the application that knows how to talk to a database. You need to build the image for it. Create an app/Dockerfile:
   FROM node:16-alpine
   WORKDIR /usr/src/app
   COPY package*.json ./
   RUN npm install
   COPY . .
   CMD ["node", "server.js"]
   And build it (from the `06-Building-Bridges` directory): docker build -t db-app-image ./app
4. Connect the Bridge (Run the App): Run your application container, attaching it to the same network. This is what completes the connection, allowing the two containers to communicate. docker run --detach --name my-app --network my-app-net --publish 8080:8080 db-app-image
5. Confirm the Connection: Open http://localhost:8080. If the bridge is built correctly, you will see a message showing the current time from the database. This proves your my-app container found and communicated with the `my-database` container across the network.
6. Run the script ./verify.sh to confirm to Dr. Sharma that your architecture is sound.


You watch the successful response from the server. It feels good. But as you look at the series of commands you had to run, you can't help but think it's clumsy. Just then, Ben walks over.

07: The Fleet (Orchestrating with Compose)

What You Will Learn

Your research leads you to the perfect tool: Docker Compose. It's a tool for defining and running multi-container applications using a single, declarative file.

Your mission: Solve your own problem (and Ben's) by defining your application and database as a fleet of services in a docker-compose.yml file.

- How to manage a multi-container application from a single file.
- The syntax of docker-compose.yml to define services, builds, ports
- How to use docker-compose up to launch an entire application stack.

Here's what you need to do:

You'll now codify the architecture you built manually in the last step into a docker-compose.yml file.

1. ft the Fleet Blueprint (docker-compose.yml):
   version: '3.8'

   services:
     app: # You can name this service anything, e.g., 'backend'
       build: ./app
       ports:
         - "8080:8080"
       environment:
         - DB_HOST=db
         - DB_PASSWORD=mysecretpassword
       depends_on:
         - db

     db: # The service name 'db' becomes its hostname
       image: postgres
       environment
         - POSTGRES_PASSWORD=mysecretpassword
       volumes:
         - postgres-data:/var/lib/postgresql/data

     volumes:
       postgres-data:

2. Launch the Fleet: This is the moment of truth. From the 07-The-Fleet directory, run one command to launch everything.
   docker-compose up --build -d Watch as Docker Compose builds your app image, creates a network, creates a volume, and starts both the db and app containers in the correct order.
3. Confirm Operational Status: Navigate to http://localhost:8080. You should see the database time, proving that your entire fleet is operational and communicating correctly.
4. Decommission the Fleet: To stop and remove everything created by Compose, there's an equally simple command. docker-compose down
5. With your stack running, run the verification script to confirm your success.
   ./verify.sh

You show your solution to Ben, who is amazed. You've just taken a massive leap in your DevOps journey. A new message from Dr. Sharma arrives, his timing as impeccable as ever.

08: Lean Manufacturing (Optimizing Images)

TO: Alex
FROM: Dr. A. Sharma
SUBJECT: Efficiency

saw your Compose file. It works. But I also analyzed the image you built for Ben's style guide. It's bloated. It contains development dependencies and other artifacts that have no place in a production environment. For Project Phoenix, our images must be lean, secure, and efficient. They must contain only what is strictly necessary to run the application. Production is not a development playground. Your next task is to learn how to build optimized images using multi-stage builds. I expect a significant reduction in image size.

What You Will Learn

Your mission is to refactor your application's Dockerfile to use a multi-stage build, separating the build environment from the final production environment to create a lean, production-ready image.

- The importance of small Docker images for security and performance.
- How to use multi-stage builds to create optimized production images.
- How to copy build artifacts from a temporary build stage to a clean production stage.

Here's what you need to do:

Dr. Sharma's directive is clear: make it smaller. You will refactor your Dockerfile to achieve this.

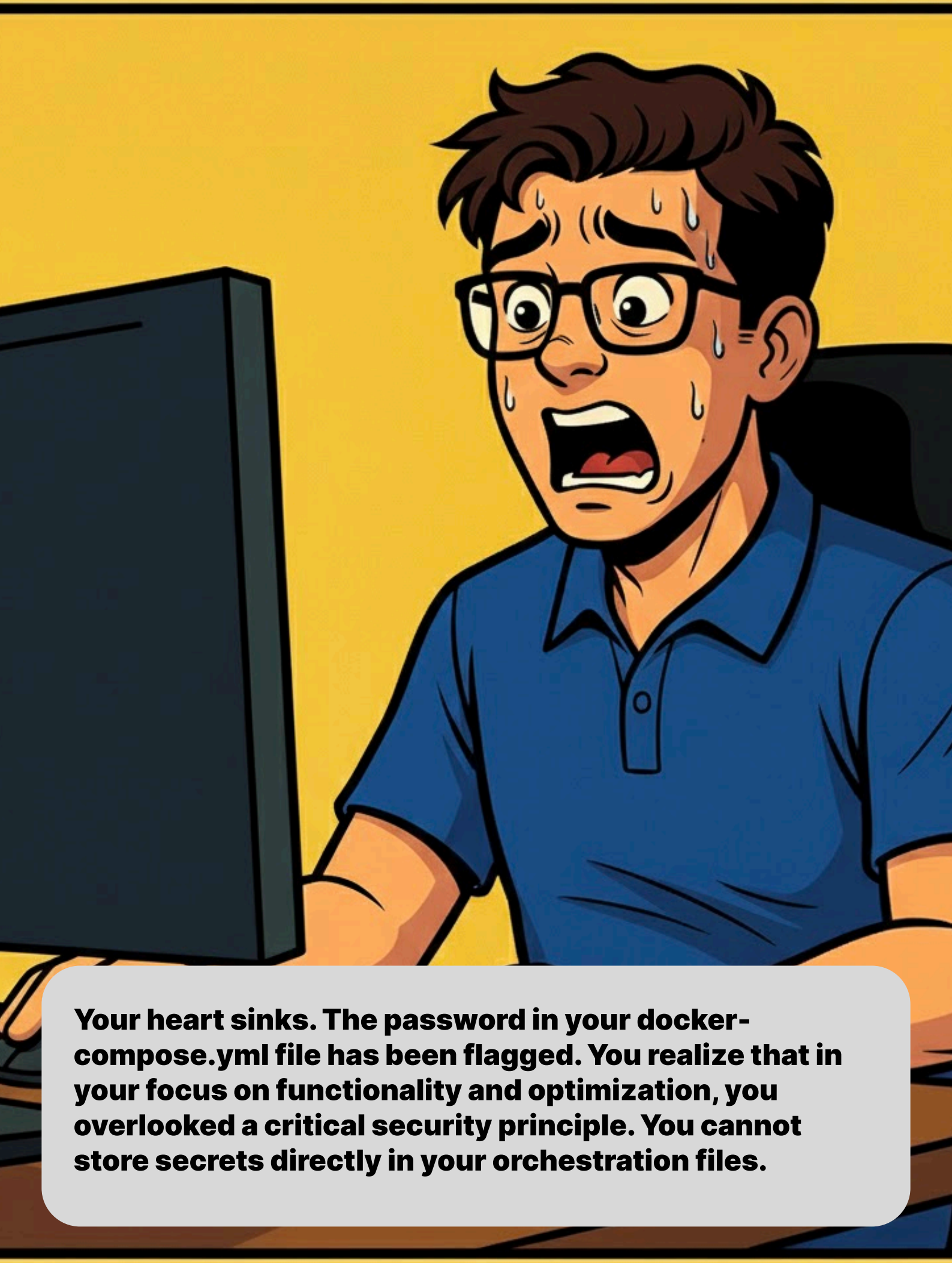
1. Draft the Two-Stage Blueprint: In the app directory, create a new Dockerfile with two distinct stages. The first stage, the builder, will be a workshop where you install everything needed to build the app. The second, production, will be a clean room where you copy only the finished product.
   ```
   # --- STAGE 1: The Workshop (Builder) ---
   FROM node:16-alpine AS builder
   WORKDIR /usr/src/app
   COPY package*.json ./
   # Install everything, including dev dependencies like nodemon
   RUN npm install
   COPY . .
   # --- STAGE 2: The Clean Room (Production) ---
   FROM node:16-alpine
   WORKDIR /usr/src/app
   # Copy only the necessary production artifacts from the builder stage
   COPY --from=builder /usr/src/app/node_modules ./node_modules
   COPY --from=builder /usr/src/app/package.json ./package.jsonCOPY --
   from=builder /usr/src/app/server.js ./server.js
   # This image will NOT contain nodemon or any other dev dependencies
   CMD [ "node", "server.js" ]
   ```
2. Build the Optimized Image: From the 08-Lean-Manufacturing directory, build your new, optimized image. docker build -t optimized-app-image ./app

3. Analyze the Results: You can inspect the size of your new image and compare it to an unoptimized version. The difference should be noticeable. The devDependencies like nodemon are now gone from the final image, resulting in a smaller, more secure asset.
   docker images optimized-app-image
4. Dr. Sharma expects to see a smaller image. The verification script will build your image and check that its size is below a reasonable threshold for a production application. ./verify.sh


You're feeling good about your lean, optimized image. As you're about to report your success to Dr. Sharma, an automated, high-priority alert appears in your inbox.
Alert
09: Securing the Cargo (Managing Secrets)

Scanner: CodeGuard AI
Finding: Hardcoded Secret Detected in innovatech/project-phoenix/compose/docker-compose.yml
Severity: Action Required

Remediate immediately. Do not commit secrets to version control.



Your heart sinks. The password in your docker-compose.yml file has been flagged. You realize that in your focus on functionality and optimization, you overlooked a critical security principle. You cannot store secrets directly in your orchestration files.

What You Will Learn

Your mission is to remediate this critical security flaw by separating your secrets from your configuration using a .env file.

- Why committing secrets to version control is a critical security risk.
- How to use a .env file to manage secrets for local development.
- How to use the env_file directive in Docker Compose to securely inject secrets into your services.

Here's what you need to do:

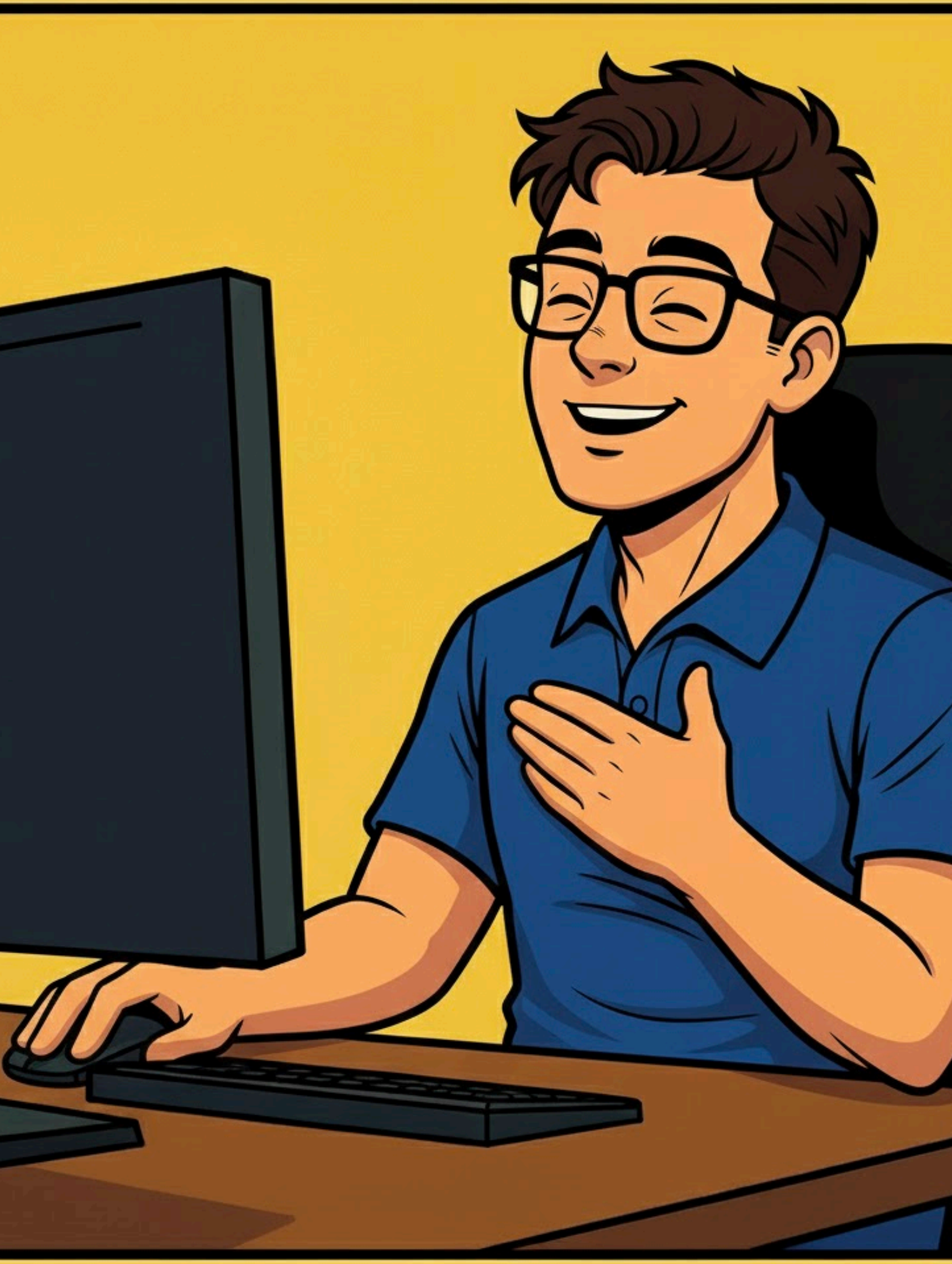
You need to fix the security flaw before anyone notices. You'll use a .env file to hold the password and modify your docker-compose.yml to read from it.

1. Examine the Evidence (.env file): A .env file has been provided for you. This file is where your secrets will now live. Notice how it defines the passwords for both the database and the application.
   ```
   # This file should be added to .gitignore!
   POSTGRES_PASSWORD=a_very_secure_password_123
   DB_PASSWORD=a_very_secure_password_123
   DB_HOST=db
   ```
2. Remediate the Vulnerability (docker-compose.yml): Open the docker-compose.yml file. It has been pre-configured to use env_file for both services. This tells Compose to load the variables from the .env file and pass them into the containers.
   ```
   services:
     app:
       build: ./app
       ports:
         - "8080:8080"
       # This line loads the .env file into the container
       env_file:
         - ./.env
       depends_on:
         - db

     db:
       image: postgres
       # This service also needs the password
       env_file:
         - ./.env
       volumes:
         - postgres-data:/var/lib/postgresql/data
   ```

There is nothing for you to change here, but you must understand *why* this is the secure solution.

3. Confirm the Fix: Launch the stack. If the env_file directive works as expected, the application should start up and connect to the database successfully, just as it did before. docker-compose up --build -d
4. Verify in Your Browser: Navigate to http://localhost:8080. The database time proves your application received the password from the .env file and connected successfully.
5. Run the verification script to confirm that the security vulnerability has been remediated and the application is fully functional. ./verify.sh



10: The Phoenix Rises (The Capstone Project)

The final message from Dr. Sharma arrives. This is the one that matters.

TO: Alex
FROM: Dr. A. Sharma
SUBJECT: Final Assessment: Project Phoenix

saw the security alert. I also saw your rapid remediation. Learning from mistakes is a sign of a good engineer. You have learned to build, to manage, to connect, to optimize, and to secure. You have helped your colleagues and strengthened your own workflow. The training is over. Now, the real work begins. I have given you access to the initial source code for the authentication service of Project Phoenix. It is a three-tier application: a frontend, a backend API, and a database. Your final assessment is this: Containerize it. Orchestrate it. am not providing a step-by-step guide. I am providing requirements. You have all the skills you need. Show me you are ready to lead this initiative.

What You Will Learn

This is it. Your mission is to take the provided source code and, from scratch, create the Dockerfiles and Docker Compose file to run the entire application stack, applying all the best practices you have learned.

- Demonstrate mastery of Docker and Docker Compose.
- Architect a containerization strategy for a real-world, multi-tier application.
- Solidify your position as a technical leader at Innovatech.

Project Phoenix: Service Requirements

Dr. Sharma has laid out the specifications. You must meet them all.

1. Orchestration: Create a docker-compose.yml file in the root 10-The-Treasure-Island directory.
2. Secrets Management: Create a .env file in the root directory to store all passwords, users, and other configuration. This file must not contain any hardcoded secrets.
3. Database Service (db): mage: postgres:13-alpine, Data: Must persist in a named volume called capstone-db-data. Configuration: All settings (user, password, db name) must be loaded from your .env file.
4. Backend Service (backend): Build: Must be built from a Dockerfile you create in the backend directory. Optimization: The Dockerfile must use a multi-stage build to create a lean production image. Networking: Must be able to communicate with the db service. Configuration: Must get all database connection details from the .env file.
5. Frontend Service (frontend): Build: Must be built from a Dockerfile you create in the frontend directory. Technology: Use an nginx:alpine image to serve the static index.html and app.js files. Accessibility: Must be accessible from your browser on port 80.

Here's what you need to do:

This is your canvas. You know what to do.

1. **Analyze the source code in the frontend and backend directories.**
2. **Architect and create the Dockerfile for the backend.**
3. **Architect and create the Dockerfile for the frontend.**
4. **Define all secrets and configurations in your .env file.**
5. **Construct the docker-compose.yml file to bring all services, networks, and volumes together.**
6. **Launch the Phoenix: docker-compose up --build -d.**
7. **Verify that http://localhost` shows the frontend, and that clicking the button successfully retrieves and displays data from the database via the backend.**
8. **This is the final test. Run the verification script. It will perform a full integration test on your running application stack. ./verify.sh**

As the verification script returns a success message, you see an email notification from HR and Dr. Sharma.


TO: Alex
FROM: Dr. A. Sharma, HR
SUBJECT: Promotion: Technical Lead, Project Phoenix.

You have completed the DevOps Journey. You are no longer just a developer; you are an architect of modern applications.