Month 5, Week 4: The Unabridged Learning Guide

Introduction: The Automated Assembly Line

Over the past few weeks, we have mastered the art of building and containerizing a single application. We have a professional, production-grade `Dockerfile` that packages our NestJS API into a portable, consistent unit. However, a real-world application is rarely a single unit. It is a system of collaborating services—an API, a database, a cache, a message queue, and more.

Managing these services individually is complex and error-prone. This week, we learn to become conductors of our own software orchestra. We will master **Docker Compose**, the tool that allows us to define, orchestrate, and run our entire multi-container application with a single command. We will learn how to manage application secrets and configuration professionally using **environment variables**, completely decoupling our code from the environment it runs in.

Then, we will take the final and most crucial step in a senior architect's workflow: **automation**. We will introduce the philosophy of **CI/CD (Continuous Integration & Continuous Deployment)**, the practice of creating an automated assembly line that takes our code from a `git push` to a fully tested and built artifact without manual intervention. We will implement a practical CI pipeline using **GitHub Actions**, creating a safety net that automatically runs our linter and tests on every change, ensuring that our codebase is always in a healthy, deployable state.

By the end of this lesson, you will have the skills to manage a complete application stack, handle configuration like a professional, and automate your workflow, freeing you to focus on what truly matters: building great software.

Table of Contents

Module 1: Docker Compose - The Conductor

The Need for Orchestration Our NestJS API is useless without its partners: a PostgreSQL database to store data and a Redis server for caching. Starting these manually requires a series of long `docker run` commands: `docker run --name db -e POSTGRES_PASSWORD=... -p 5432:5432 -v pgdata:/var/lib/... postgres docker run --name redis -p 6379:6379 redis docker run --name api -p 3000:3000 --link db:db ... my-api` This is complex, hard to remember, and doesn't guarantee the services can talk to each other correctly. Docker Compose solves this by using a declarative YAML file.

Core Concepts: Services, Networks, and Volumes

- **Services:** A service is the definition of a single container in your application stack. It defines the image to use, the ports to map, the environment variables to set, and its relationship to other services.
- **Networks:** By default, Docker Compose creates a single, private "bridge" network for all services defined in your `docker-compose.yml` file. This is a critical feature. It means your `api` container can connect to your `db` container using the simple hostname `db`, without needing to know its internal IP address.
- **Volumes:** Volumes are the preferred mechanism for persisting data. When you define a named volume in your Compose file (like `pgdata`), Docker manages the storage location on the host machine. This is safer and more portable than using a bind mount (linking to a specific folder path on your host).

Anatomy of a `docker-compose.yml` File This file is the single source of truth for your entire application stack's configuration. * **`version:`** Specifies the file format version. `'3.8'` is a common, modern choice. * **`services:`** The top-level key where all your container definitions live. * **`build` vs. `image`:** Use `build: .` for services you build from a local `Dockerfile`. Use `image: postgres:15-alpine` for services that use a pre-built image from a registry like Docker Hub. * **`ports:`** Maps `HOST:CONTAINER` ports. * **`environment:`** Sets environment variables inside the container. * **`volumes:`** Mounts volumes. `named-volume:/path/in/container`. * **`depends_on:`** Controls the startup order. `api` depends on `db`, so Compose will start the `db` container first.

Core Compose Commands

- **`docker-compose up`:** The main command. It reads your `docker-compose.yml`, builds any necessary images, and creates/starts all services.
- **`docker-compose up -d`:** The `-d` (detached) flag is essential for running your stack in the background.
- **`docker-compose down`:** Stops and removes the containers and networks. To also remove named volumes, you must use the `-v` flag: `docker-compose down -v`.
- **`docker-compose build`:** Forces a rebuild of your custom images (e.g., after you change your `Dockerfile`).

Module 2: Managing Configuration & Secrets

The Principle of Decoupling Configuration A well-architected application adheres to the "Twelve-Factor App" methodology, a set of best practices for building modern software. One of its key principles is to **store config in the environment**. Your codebase should be the same across all environments (development, staging, production). The only thing that changes is the configuration, which is injected via environment variables. This prevents secrets from ever being committed to version control.

Using `.env` Files with Docker Compose Docker Compose has built-in support for `.env` files. When you run `docker-compose up`, it will automatically look for a file named `.env` in the same directory and load its key-value pairs. You can then substitute these variables into your `docker-compose.yml` file using the `${VARIABLE_NAME}` syntax.

This creates a clean and secure pattern for local development: 1. Define your secrets in a `.env` file. 2. Add `.env` to your `.gitignore` and `.dockerignore` files. 3. Create a `.env.example` file with placeholder values and commit that to Git so other developers know what variables are needed.

Module 3: The Philosophy of CI/CD

Continuous Integration (CI): The Safety Net CI is an automated process where, every time a developer pushes code to a shared repository (e.g., in a pull request), a dedicated server automatically: 1. Checks out the code. 2. Installs all dependencies. 3. Runs a series of quality checks (like linting). 4. Runs the entire automated test suite (unit, integration, E2E). 5. (Optionally) Builds a production artifact, like a Docker image.

If any of these steps fail, the pipeline fails, and the developer is immediately notified. This prevents broken code from ever being merged into the main branch, keeping it stable and deployable at all times.

Continuous Delivery vs. Continuous Deployment (CD)

- **Continuous Delivery:** The CI pipeline runs, and if it succeeds, the application is automatically deployed to a **staging** or testing environment. A final, **manual** approval step (e.g., a manager clicking a "Deploy to Production" button) is required to release to customers.
- **Continuous Deployment:** This is the ultimate goal of automation. If the CI pipeline and all automated tests succeed, the new version of the application is **automatically deployed to production** with no human intervention.

Module 4: Implementing a CI Pipeline with GitHub Actions

Core Concepts of GitHub Actions

- **Workflow:** The top-level YAML file in `.github/workflows/`.
- **Event:** The trigger (e.g., `on: push`).
- **Job:** A set of steps that runs on a runner. Jobs can run in parallel.

- **Runner:** A virtual machine hosted by GitHub that executes your job.
- **Step:** An individual task. `uses:` runs a pre-built Action; `run:` executes a shell command.
- **Action:** A reusable piece of code from the GitHub Marketplace or your own repository. `actions/checkout` and `actions/setup-node` are essential.

Anatomy of a Workflow File The YAML file is declarative. You define the "what," and GitHub Actions figures out the "how." The indentation is critical.

Building a Practical CI Pipeline Our CI pipeline for the NestJS app will have steps for: 1. **Checkout:** Get the code. 2. **Setup Node:** Install the correct Node.js version on the runner. 3. **Caching Dependencies:** A key optimization. This step caches the `node_modules` folder. On subsequent runs, if `package-lock.json` hasn't changed, it can restore the cache instead of running `npm install` from scratch, saving a significant amount of time. 4. **Install Dependencies:** `run: npm ci` (using `ci` instead of `install` is a best practice in CI as it's faster and uses the lock file). 5. **Run Linter:** `run: npm run lint`. 6. **Run Tests:** `run: npm run test`.

Take-Home Assessment: The Complete DevOps Workflow

Objective: To demonstrate mastery of Docker Compose and CI/CD by orchestrating your full application stack and creating an automated testing pipeline.

The Task: You will be given the complete, database-connected NestJS project.

Part 1: Docker Compose Orchestration 1. Create a `docker-compose.yml` file to orchestrate three services: `api`, `db` (PostgreSQL), and `cache` (Redis). 2. Configure the `db` service to use a named volume to persist its data. 3. Configure all three services to restart automatically if they fail (`restart: always`). 4. Create a `.env` file to store all your secrets (database credentials, JWT secret). 5. Ensure your `docker-compose.yml` file reads these secrets from the environment variables. 6. Update your application's data source and cache module configurations to read their credentials from environment variables (e.g., `process.env.DATABASE_USER`). 7. Add a `.env.example` file to your repository.

Part 2: GitHub Actions CI Pipeline 1. Create a `.github/workflows` directory in your project. 2. Inside, create a `ci.yml` file. 3. Define a workflow that triggers on `push` and `pull_request` events to the `main` branch. 4. Create a single job named `build-and-test`. 5. Add the following steps to the job: * Checkout the code. * Set up Node.js version 18. * Add a step to cache your `node_modules` directory based on the `package-lock.json` hash. * Install dependencies using `npm ci`. * Run the linter using `npm run lint`. * Run your complete test suite using `npm run test`.

Submission: Submit your entire updated project, including the `docker-compose.yml`, `.env.example`, and `ci.yml` files, via a Pull Request to your personal assignments repository. GitHub will automatically run your new CI pipeline on the PR. The submission will only be considered complete if the pipeline passes. ‘