DevOps Intern Take-Home Assignment

Branch Loan API

The Story

You've just joined Branch's DevOps team in India as an intern. Branch is a fintech company that provides microloans to underserved populations across emerging markets.

Your manager explains: "We have a Loan API service that works on our laptops, but we need to containerize it, set up CI/CD pipelines, and make it production-ready. Right now, deployments are manual and we have no visibility when things break. Your task is to transform this into a properly automated, scalable service."

Your Mission

Take our existing Loan API service and transform it into a production-ready, containerized application with automated deployment pipelines and proper infrastructure setup.

What We're Giving You

A simple REST API (Node.js/Python/Go - we'll provide) with endpoints like:

- GET /health - Health check
- GET /api/loans - List all loans
- GET /api/loans/:id - Get specific loan details
- POST /api/loans - Create new loan application
- GET /api/stats - Get loan statistics

The API connects to a PostgreSQL database and currently runs on developer machines. It works, but it's not ready for production. Here is the repo. As part of this assignment, please create a fork of this repo and make all the required changes there.

What You Need to Build

Part 1: Containerization

The Challenge: "We need this running in containers so we can deploy consistently across environments."

The Loan API application needs to be containerised and run locally with the following requirements:

1. The application should run in a Docker container
2. It connects to a PostgreSQL database (also in a container)
3. When you visit https://branchloans.com in your browser (on your local machine), you should see the API running
4. It must be served over HTTPS with a valid certificate (self-signed is fine for local development)

Think about how you'll set up the domain locally, how to generate the SSL certificate, and how the containers will communicate with each other.

Part 2: Multi-Environment Setup

The Challenge: "We run this service in development, staging, and production. Each needs different settings, but we don't want to maintain separate setups."

Create a Docker Compose setup that can handle multiple environments with different configurations:

1. **Development environment:**
   - Database: Small PostgreSQL instance
   - API: Debug logging enabled
   - Hot reload for code changes (if applicable)
2. **Staging environment:**
   - Database: Medium PostgreSQL with some resource limits
   - API: Standard logging
   - Should mimic production as closely as possible
3. **Production environment:**
   - Database: Larger PostgreSQL with proper resource limits and health checks
   - API: Structured JSON logging, optimized settings
   - Data persistence (database data should survive container restarts)

The same docker-compose file should work for all three environments. Show us how you'd switch between them and what configuration values would differ. Consider what should be configurable (port numbers, memory limits, log levels, database credentials, etc.).

Part 3: CI/CD Pipeline

The Challenge: "Every code push requires manual building and deployment. We need this automated."

Create a CI/CD pipeline using GitHub Actions that automates the build and release process. Your pipeline should include these stages:

1. **Test Stage:**
   - Run any tests that exist in the codebase
   - If tests fail, stop the pipeline
2. **Build Stage:**
   - Build the Docker image
   - Tag it appropriately (use git commit SHA or branch name)
3. **Security Scan Stage:**
   - Scan the Docker image for vulnerabilities
   - Fail the pipeline if critical vulnerabilities are found
4. **Push Stage:**
   - Push the image to a container registry
   - Use Docker Hub or GitHub Container Registry (ghcr.io) - both are free
   - Only push if all previous stages pass

The pipeline should trigger on:

- Push to main branch
- Pull requests (but don't push images for PRs)

Note: Think about sensitive information like container registry credentials, database passwords, API keys, etc. How should these be handled so they're never exposed in your code or pipeline logs?

Part 4: Documentation

The Challenge: "Other engineers need to understand and run your solution."

Create comprehensive documentation that includes:

README.md:

- How to run the application locally (step-by-step instructions)
- How to switch between different environments (dev/staging/production)
- What each environment variable does
- How the CI/CD pipeline works
- A simple architecture diagram showing how the components connect (can be ASCII art or a simple drawing)

Design Decisions:

- Document why you chose certain approaches
- What trade-offs did you consider?

- What would you improve with more time?

Troubleshooting:
- Common issues someone might face and how to fix them
- How to check if everything is running correctly

Bonus Challenges (Optional)

If you finish early and want to impress us:

Observability & Monitoring

- **Health Check Endpoint:** Add a /health endpoint that actually verifies the service is working (checks database connectivity, not just returns "OK")

- **Structured Logging:** Implement JSON-formatted logs that include useful context (timestamp, log level, request ID, etc.)

- **Prometheus Metrics:** Set up Prometheus to scrape metrics from your application
    - Add a /metrics endpoint to your app that exposes metrics
    - Run Prometheus locally using Docker
    - Show basic metrics like request count, response time, or error rates
    - Bonus: Add a Grafana dashboard to visualize the metrics

Deliverables

Submit a GitHub repository (or zip file) with your complete solution.

Time Expectation

Please spend 4-6 hours on this assignment. We're not timing you, but we trust you to stay within this range. We value quality over quantity—a well-documented, working solution for the core requirements is better than an incomplete attempt at all the bonus features.

Good luck! We're excited to see how you approach this! 🚀