Semester Project Specification

Course: Continuous Integration (S4-CONINT)

Project: Secret Notes

---

1. Context and Motivation

You are working for an Austrian company that wants to provide a secure note-taking service. This service should allow users to create "secret notes" that are stored in a database *only in encrypted form*. Whenever a user retrieves a note, the system must *decrypt* that content on the fly - **but only if the user provides the correct decryption key**.

Your team wants to practice DevOps principles and implement Continuous Integration (CI) and Continuous Deployment/Delivery (CD) to ensure reliable and rapid releases. Your CI/CD pipelines will run on both a **cloud-hosted** solution (e.g., GitHub Actions, GitLab CI, CircleCI, Azure DevOps) *and* a **self-hosted** solution (e.g., Jenkins, self-managed GitLab CI). The plan is to deploy the final "Secret Notes" application on AWS using a blue/green deployment strategy.

---

2. Technical Specifications

2.1 Application Components

1. **Frontend**
   - A web application (e.g., Vue.js, React, or plain HTML, JS and CSS) that allows users to:
     - Create and view notes
     - Provide the correct key or passphrase to decrypt existing notes
   - Dockerized and ready for AWS deployment
2. **Backend**
   - A Node.js (Fastify) web service that:
     - Receives, encrypts, and stores user notes in a relational database (PostgreSQL on AWS RDS)
     - Retrieves and decrypts user notes when the correct key is provided
     - Exposes a REST API to the frontend
   - Dockerized and ready for AWS deployment

2.2 Cloud Infrastructure

- **Amazon Web Services (AWS)**
  - **RDS** – runs a PostgreSQL database for storing notes
  - **EC2** – hosts Docker containers via Docker Compose or another container orchestration approach

2.3 Container Registry and Docker

- **Docker Hub**
  - Stores your Docker images for both the Frontend and Backend

2.4 CI/CD Servers

1. **Cloud-hosted CI**
   - One of: GitHub Actions, GitLab CI (SaaS), CircleCI, or Azure DevOps
2. **Self-hosted CI**
   - Jenkins on AWS EC2

2.5 Code Quality Server

- **SonarQube** (self-hosted on AWS)
  - Used for static code analysis (linting, code smells, duplications, etc.)

2.6 Security Scan

- **Snyk**
  - Performs security checks on your code and dependencies

2.7 Feature Toggle / A/B Testing

- **PostHog**
  - Use for toggling advanced encryption features, UI variations, or experimentation with new user experiences

---

3. Pipeline Requirements

Each pipeline (one for **frontend**, one for **backend**) must have **at least 5 stages**:

1. **Linting Stage**
   - Perform static code analysis via **SonarQube**
   - Perform dependency/security scans via **Snyk**
2. **Unit Testing Stage**
   - Run your test suites with Jest
   - Provide test coverage reports (at least 10 meaningful tests per project)
3. **Build Stage**
   - Build the Docker image (e.g., `docker build -t secret-notes-frontend .` for frontend, similarly for backend)
4. **Deliver Stage**
   - Push the Docker image to **Docker Hub**.

5. **Deploy Stage**
    - Deploy the application on AWS using your Docker images
    - Deploy on staging
    - Implement **Blue/Green Deployment** to minimize downtime and risk
6. **E2E and Performance Testing Stage**
    - Run your test suites with k6 and Playwright on staging environemnt
    - If successful switch **Blue/Green**

Pipeline Behavior

- A **failed stage** stops the pipeline immediately and notifies developers (via Slack, email, or your chosen channel)
- A push to the “main” branch runs **Linting**, **Testing**, and **Build** stages
- A push to the “deploy/production” branch runs **all stages** (including **Deliver**, **Deploy**, and **E2E/Performance Testing**)

---

4. Features

You are expected to develop **three main features** in the "Secret Notes" app:

1. **Feature A: Basic Encryption Storage**
    - Users can create notes that are stored *only* in an encrypted form
    - A user chooses or is assigned an encryption key. The backend uses that key to encrypt and store the content
2. **Feature B: Secure Note Retrieval**
    - When a user tries to read a note, they must provide the correct key. The backend decrypts and returns the plaintext if and only if the provided key matches
    - If an incorrect key is provided, the system should deny access or return an error
3. **Feature C: A/B Toggle for UI**
    - Implement a feature toggle with **PostHog** that controls some aspect of user experience. For instance:
        - Display different UI color themes or note-editing layouts to different user groups for an A/B test
    - Split users into two groups (Group A vs. Group B) so you can gather metrics on which variant provides better usability or performance

---

5. Definition of Done and Backlog

5.1 Definition of Done

- All code changes pass all pipeline stages (Lint, Test, Build, Deliver, Deploy)
- All Code is Tested

- Encrypted data is verified to be unreadable without the correct key
- Documentation is up to date with instructions on setup, usage, and encryption logic
- Blue/Green deployment is successful with no downtime or data loss during a new version release
- Feature toggles and A/B tests can be turned on/off in **PostHog** without redeploying the entire application

5.2 Refined Backlog and User Stories

1. **User Story 1: Create Secret Note**
    - *As a user, I want to create a secret note so that my text is stored securely.*
    - **Acceptance Criteria**:
        - The user enters text and a key/passphrase
        - The note is encrypted on the server and stored in the database
        - UI confirms the note is created securely
2. **User Story 2: Read Secret Note**
    - *As a user, I want to retrieve a note so that I can view its content when I have the correct key.*
    - **Acceptance Criteria**:
        - The user enters the same key used to encrypt
        - The system decrypts the note if the key is valid, returning the plaintext to the user
        - The system rejects or returns an error if the key is invalid
3. **User Story 3: Manage Feature Toggles**
    - *As a DevOps engineer, I want to toggle specific UI so I can run A/B tests.*
    - **Acceptance Criteria**:
        - A toggle in PostHog allows enabling or disabling a second UI
        - A/B testing splits users into at least two groups with distinct experiences

---

6. Docker Files

- **Dockerfile (Backend)**
    - Installs Node.js dependencies
    - Copies backend code
    - Runs production build (if needed) and exposes necessary ports
- **Dockerfile (Frontend)**
    - Installs dependencies for Vue.js/React/Plain-HTML-JS-CSS
    - Builds the production bundle
    - Serves static files using nginx or a lightweight web server

---

7. Project Documentation

A comprehensive documentation (in the official FHTW template) must describe:

1. **Applications + Docker Files**
    - Explanation of the chosen frameworks (Vue, React, or Express)
    - Dockerfile instructions, ports, environment variables
2. **Version Control**
    - Branching strategy (main, feature branches, deploy/production branch).
3. **Infrastructure (AWS)**
    - EC2 instance configuration
    - RDS configuration for PostgreSQL
    - Network details and security groups
4. **Docker Hub**
    - Repository setup and naming conventions
    - Image tagging strategy
5. **CI Servers**
    - Details of both pipelines (cloud-hosted and self-hosted)
    - Screenshots or logs showing successful and failing runs
6. **Code Quality Server**
    - SonarQube setup, including typical metrics collected
7. **Security Scan Server**
    - Snyk usage, how to interpret and fix vulnerabilities found
8. **Feature Toggle & A/B Test Server**
    - PostHog setup for toggles and experiments
    - Example toggles with instructions on how to enable/disable them
9. **Each Stage of the Pipeline**
    - Detailed explanation of Lint, Test, Build, Deliver, Deploy, E2E/Performance Testing
    - What happens if a stage fails (notification)
10. **Features and Refined User Stories**
    - Detailed acceptance criteria
    - Implementation details
11. **A/B Testing**
    - How to set up a new split test for a different UI feature or encryption flow
12. **Blue/Green Deployment**

    - How you implement it on AWS (e.g., Docker Compose definitions swapped behind a load balancer)

13. **Logging and Monitoring (Outlook)**

    - Proposed approach for logging, analytics, and potential monitoring solutions (e.g., CloudWatch, Splunk, ELK)

---

8. Blue/Green Deployment Description

Your deployment pipeline must support Blue/Green deployment on AWS. The general process:

1. **Blue Environment**: Currently running version of “Secret Notes.”
2. **Green Environment**: New version of the “Secret Notes” app is deployed here first
3. **Test Green**: Verify that encryption, decryption, and feature toggles are working as expected
4. **Switch Traffic**: Shift all user traffic from Blue to Green if tests pass
5. **Decommission Blue**: Stop the Blue environment or keep it as a fallback until fully confident in the new release

---

9. Summary of Requirements

1. **Two CI/CD Pipelines**

    - One for **backend**, one for **frontend**
    - Each pipeline has **5 stages**: Linting, Testing, Build, Deliver, Deploy
    - Each pipeline is configured on:
        - **One cloud-hosted** CI platform (GitHub Actions, GitLab CI SaaS, CircleCI, or Azure DevOps)
        - **One self-hosted** CI solution (Jenkins)

2. **SonarQube** for code quality analysis

3. **Snyk** for security scanning

4. **PostHog** for toggles and A/B tests

5. **Blue/Green Deployment** on AWS

6. **At least 10 real tests** per project

7. **Documentation** (in FHTW template) covering everything from Docker usage to feature toggles

> **Important**: Failing to use the provided structure or skipping any of the main technologies (SonarQube, Snyk, PostHog, Docker Hub, AWS, Blue/Green) may result in an immediate negative grade.

Good luck with your “Secret Notes” Semester Project!