# Todo Application Project Documentation

## 1. Application and Infrastructure Description

### 1.1 Application Overview

The project implements a full-stack Todo application with the following components:

#### Frontend

* Simple and responsive web interface built with HTML, CSS, and JavaScript
* Located in the /frontend directory
* Features a clean user interface for managing todo items

#### Backend

* Node.js/Express.js REST API
* Features:
  + CRUD operations for todo items
  + Health check endpoints
  + Database integration with PostgreSQL
  + Robust error handling
  + API routing with Express
* Implements proper separation of concerns with:
  + Controllers
  + Services
  + Models
  + Data Access Objects (DAO)
  + Routes

### 1.2 Infrastructure Components

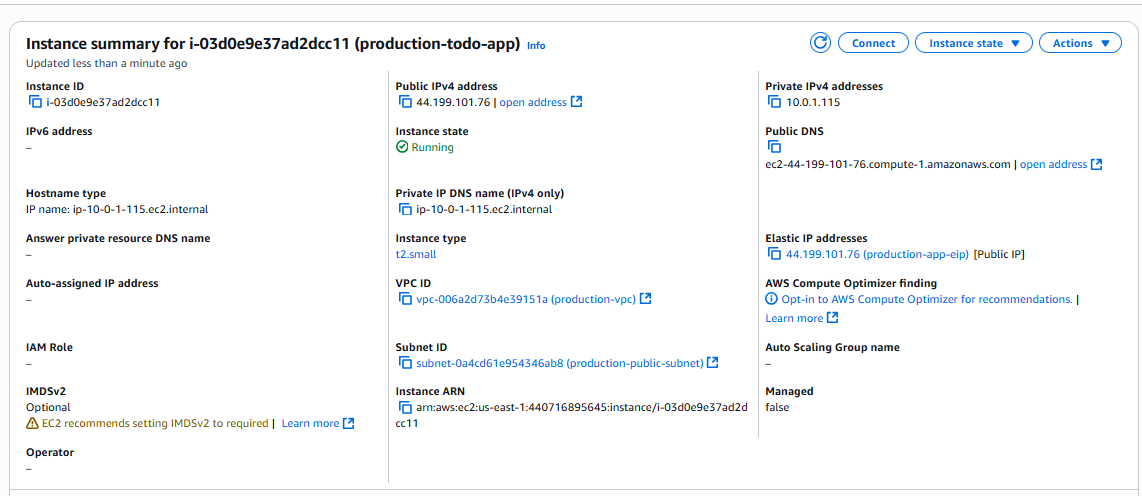
#### Docker Configuration

* Multi-container setup using Docker Compose
* Containers:
  + Frontend container
  + Backend container (Node.js)
  + PostgreSQL database container
* Features:
  + Health checks for services
  + Volume persistence for database
  + Environment variable configuration
  + Network isolation
  + Hot-reload development setup

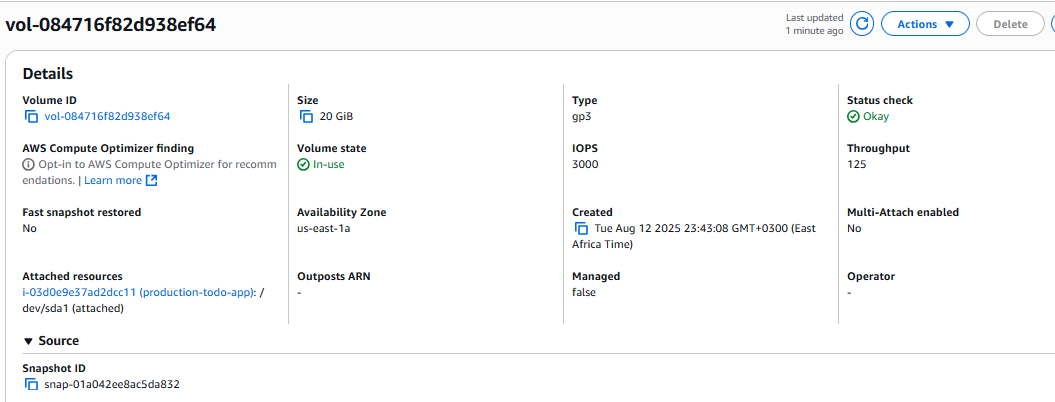
#### AWS Infrastructure (Terraform)

* Infrastructure as Code using Terraform
* Components:
  + EC2 instances running Ubuntu 20.04
  + VPC configuration
  + Security groups
  + Key pairs for SSH access
  + Availability Zones configuration
* Tagged resources for better management and cost tracking

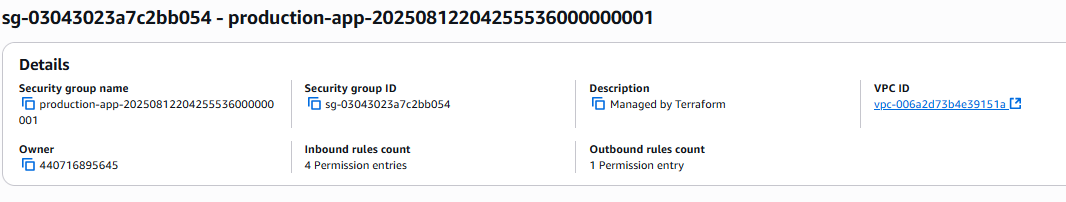
Below is an image of the EC2(T2-small) instance used host the application



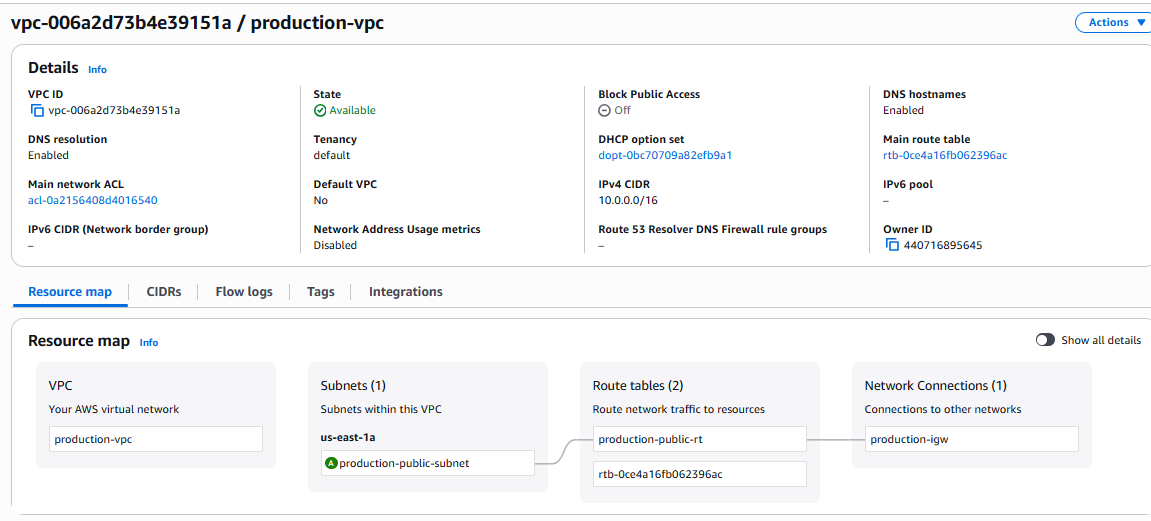
An image of the volume used



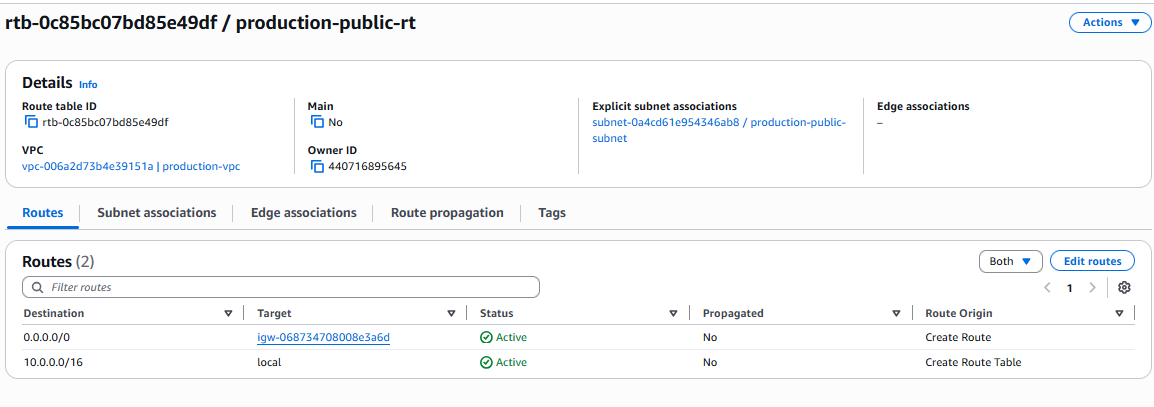
An image of the security group



An image of the vpc



An image of the route table



## 2. Pipeline Design

### 2.1 Development Environment

* Local development setup using Docker Compose
* Hot-reload capabilities for both frontend and backend
* Environment variable management
* Database persistence

### 2.2 Testing Infrastructure

* Comprehensive test suite for backend
* Jest testing framework integration
* Test coverage reporting
* Automated testing in CI pipeline

### 2.3 Deployment Pipeline Stages

1. **Source Control**
   * Git repository management
   * Branch protection rules
   * Code review process
2. **Build Stage**
   * Docker image building
   * Multi-stage builds for optimization
   * Layer caching for faster builds
3. **Test Stage**
   * Unit tests execution
   * Integration tests
   * Code coverage reporting
4. **Infrastructure Deployment**
   * Terraform initialization
   * Infrastructure validation
   * Resource creation/updates
5. **Application Deployment**
   * Container deployment
   * Health check verification
   * Zero-downtime deployment strategy

### 2.4 Tools Used

* Version Control: Git
* Container Runtime: Docker
* Container Orchestration: Docker Compose
* Infrastructure as Code: Terraform
* Cloud Provider: AWS
* Testing Framework: Jest
* API Testing: Postman/curl
* Database: PostgreSQL

## 3. Challenges and Solutions

### 3.1 Database Management

**Challenge**: Ensuring database persistence and proper migration management in a containerized environment.

**Solution**: Implemented Docker volumes for data persistence and added health checks to ensure database availability before starting dependent services.

### 3.2 Infrastructure Security

**Challenge**: Securing the application infrastructure while maintaining accessibility.

**Solution**:

* Implemented security groups with minimal required access
* Used private subnets where possible
* Implemented proper key management
* Added Helmet.js for security headers in the backend

### 3.3 Container Optimization

**Challenge**: Large container sizes and slow build times.

**Solution**:

* Implemented multi-stage builds
* Optimized Dockerfile layers
* Used .dockerignore to exclude unnecessary files

### 3.4 Testing Strategy

**Challenge**: Implementing comprehensive testing while maintaining development speed.

**Solution**:

* Implemented automated testing in CI pipeline
* Added code coverage requirements
* Created specific test environments with Docker Compose

### 3.5 Environment Management

**Challenge**: Managing different configurations across environments.

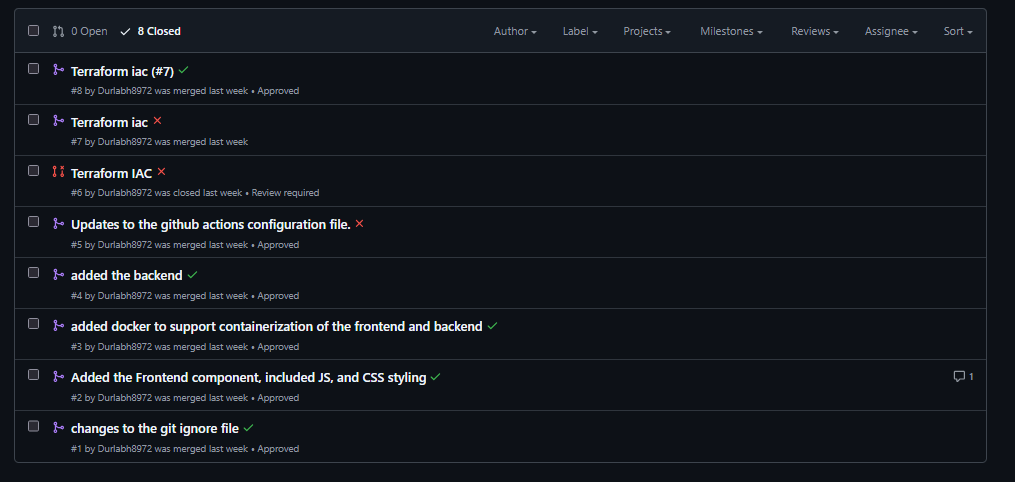
**Solution**:

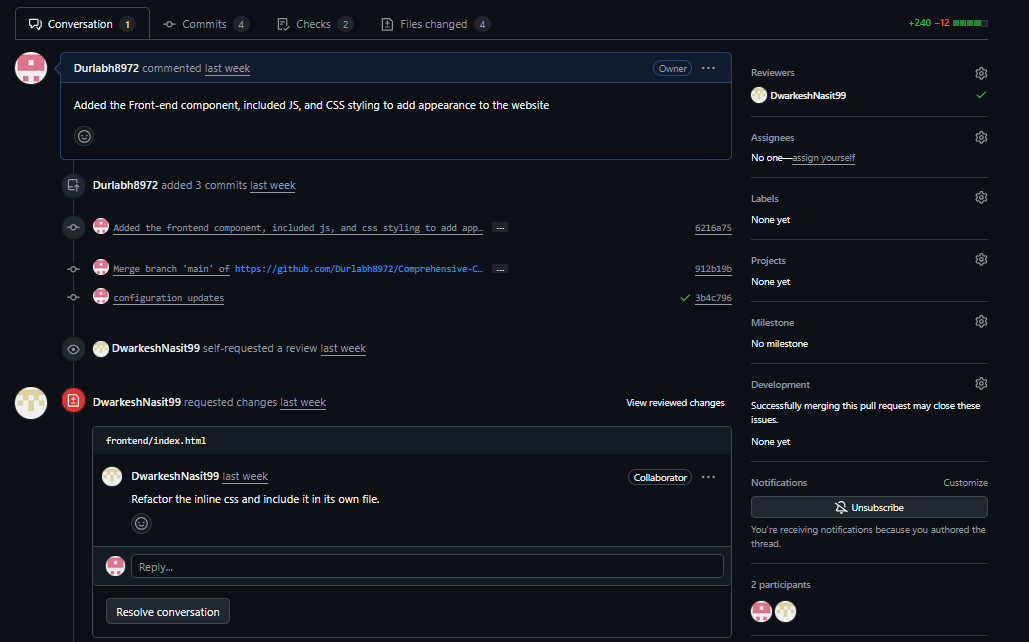
* Implemented environment variable management
* Created separate configuration files for different environments
* Used Docker Compose for local development

## 4. Version Control

## The application and Infrastructure as Code (IaC) templates are hosted in GitHub. Additionally, a structured branching strategy segregated development and production environments and single main branch serves as the stable production-ready code line. The other branches were created from the “develop” branch.

## All merges into the master branch were conducted via a formal Pull Request (PR) process. Each PR was reviewed and approved by a teammate. The screenshots of the completed code reviews and approved PRs are shown below.





## 5. Conclusion

This project demonstrates a comprehensive implementation of a modern web application with a focus on:

* Containerization
* Infrastructure as Code
* CI/CD practices
* Security best practices
* Scalable architecture
* Testing automation

The combination of tools and practices ensures a robust, maintainable, and scalable application that can be easily deployed and managed in a production environment.