EC7204: CLOUD COMPUTING

GROUP PROJECT

By:

EG/2020/3986 Jayasinghe D.M.S.N.

EG/2020/3981 Hettiarachchi P.P.P.

EG/2020/4021 Kavindya P.P.

EG/2020/4034 Kumarasiri L.I.N.

BlendRush - Juice Bar Ordering System

Link to the code: BlendRush

1. Introduction

The BlendRush Juice Bar Ordering System is a cloud-native application designed to handle juice orders online. It demonstrates scalability, high availability, security, and modern deployment practices. The system is built using a microservices architecture to ensure modularity and future extensibility.

Objectivies:

- Enable customers to browse juice menus and place orders online.
- Ensure high availability using cloud deployment and containerization.
- Secure user data and communication.
- Provide automated CI/CD pipelines for reliable deployments.
- Allow easy scaling for peak load times.

Key Features:

- Online ordering with confirmation.
- Menu browsing with categories and details.
- Ensure data security and privacy of user.
- Order confirmation emails.
- Automated deployment using Terraform and GitHub Actions.

2. System Architecture

The system follows a microservices architecture with separate containers for each service, deployed on AWS EC2 instances provisioned using Terraform. An API Gateway routes client requests to backend services.

Major Components :

1. Frontend (React + Tailwind CSS)
    - Single Page Application for customers.
    - Deployed using netlify.
2. API Gateway (Node.js + Express)
    - Entry point for all API requests.
    - Routes requests to appropriate services.
3. User service
    - Manages user registration ,login and authentication.
    - Passwords are securely hashed using bcrypt.

- JWT authentication tokens for session management
- Encrpt user register details before store in database.
- Provides protected routes for user-specific actions.

4. Menu service
    - Manages juice menu items.
    - Stores and retrieves menu data from MongoDB**.**
5. Cart service
    - Manages cart items per user.
    - Stores and retrieves selected item data from MongoDB**.**
6. Order service
    - Handles order creation, tracking, and status updates.
    - Stores order data in MongoDB.
7. Database (MongoDB)
    - Stores user info, menu items, and orders in collections.
8. Deployment Layer
    - Terraform provisions EC2 infrastructure.
    - Docker containers for all services.
    - GitHub Actions CI/CD pipeline automates build and deployment.

3. Implementation Steps

1. **Frontend Development**

- React + Tailwind CSS.
- Pages: Home, Menu, Order, Cart.
- Axios for API calls.

2. **Backend Development**

- Node.js + Express.
- Microservices for Orders, Menu, Users, Cart.
- Mongoose for MongoDB operations.

3. **Security**

- Helmet for HTTP header security.
- CORS for frontend-backend restrictions.
- Encrpt user register details before store in database.
- JWT authentication for protected routes.
- bcrypt for password hashing.

- Secure SSH key pairs for EC2 access.
- AWS Security Groups restricting access to required ports (22, 80, 3000,443).

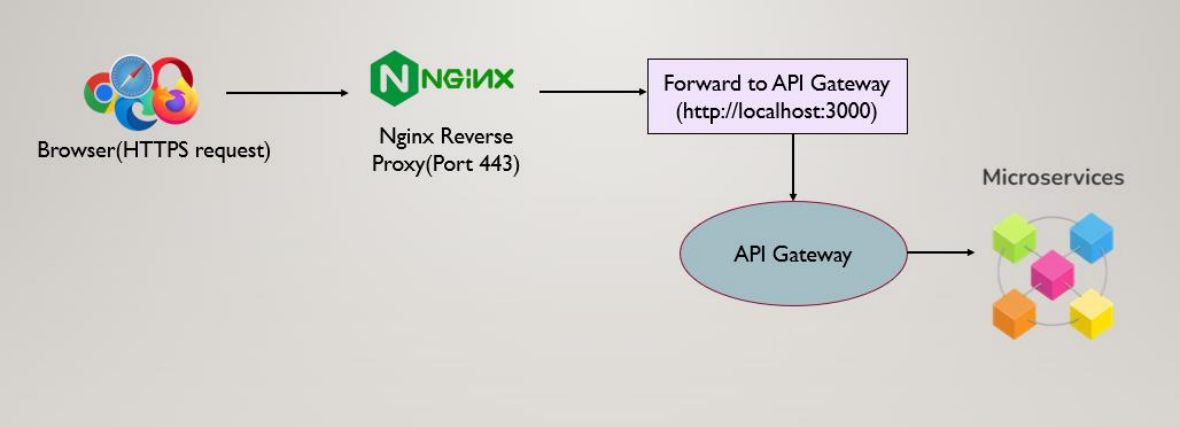

Figure 1: Security Workflow

4. **Deployment**
   - Terraform
     - Creates EC2 instances for backend.
     - Configures networking and security groups.
   - Docker
     - Each microservice were containerized.
     - Docker Compose used locally for service linking.
   - GitHub Actions:
     - CI/CD pipeline builds images, pushes to EC2.
     - Deploys containers automatically on commit.
   - AWS EC2:
     - Backend exposed on https://3.86.210.165/api.
   - Frontend Deployment
     - Frontend was deployed using Netlify.
     - Frontend exposed on https://blendrush.netlify.app/.

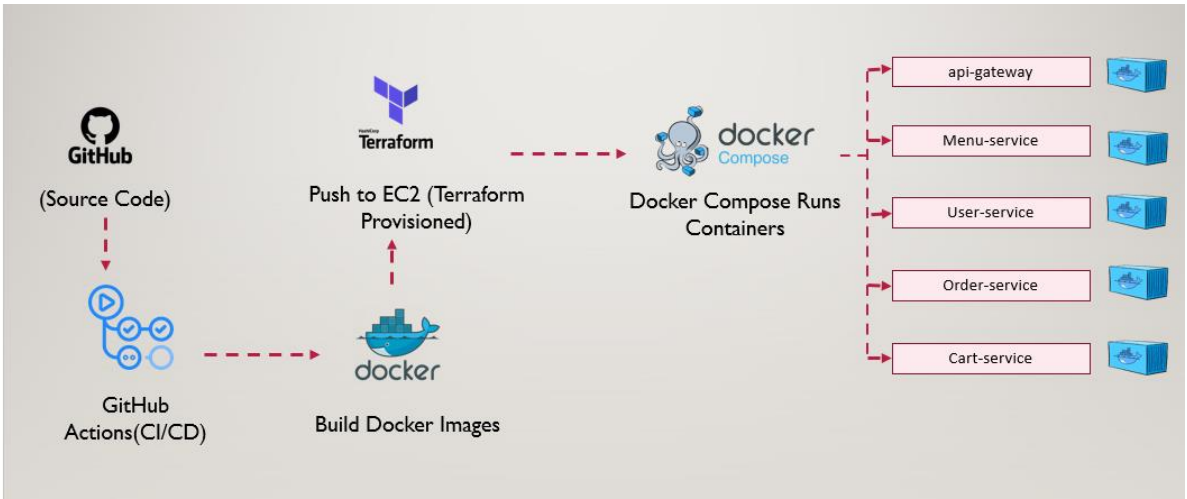

Figure 2: Deployment Workflow

4. Challenges Faced

- Setting up Docker containers to work together and communicate properly was tricky at first.
- Configuring SSH key permissions correctly on Windows was problematic during EC2 access.
- Terraform remote-exec provisioner initially failed due to authentication issues.
- The GitHub Actions workflow sometimes failed due to incorrect Dockerfile paths, missing dependencies, or timing issues during build and deploy stages.
- Making sure the app can handle more users by adding more containers was a learning process.
- Handling large files (Terraform providers) caused Git push failures until fixed with .gitignore.
- Managing multiple containers and ensuring inter-service communication required debugging.
- Integrating automated tests in the pipeline sometimes caused delays and false failures, slowing down deployment.
- When deployment failed, rolling back to previous stable versions was not always straightforward.

5. Lessons Learned

- Setting up deployment pipelines and environment configurations early saves a lot of time later.
- Setting up Terraform scripts correctly from the start avoids repeated key pair mismatches.
- Containerizing services makes it easier to run the app consistently on any machine or cloud platform.
- Automating build and deployment with GitHub Actions speeds up delivery and reduces manual errors.
- Automated tests should be reliable to avoid false failures that block deployments.
- Having a way to quickly revert deployments avoids long downtime during failures.
- Keeping good notes on deployment steps and fixes helps future maintenance and team collaboration.
- Infrastructure-as-Code (IaC) is powerful for reproducible environments.