Final Project

# Microservices Platform for User & Product Management

*Date:*

***August 03, 2025***

*Course:*

***Service Oriented Architecture (SOA915NCC)***

*Prepared by:*

***Thomas Le – 170949218***

*Instructor:*

***Waleed Akram Baig***

# Table of Contents

# 1. Introduction

## Project Overview

This application provides a microservices-based platform where users can register, authenticate, and manage their own product listings. It consists of three core services—web-interface, user-service, and product-service—each running in its own container, communicating over REST, and persisting data in separate MongoDB databases.

### *Goals*

- Design truly single-purpose services with clear API contracts
- Containerize each service using lightweight, multi-stage Docker images
- Orchestrate with Kubernetes (Deployments, Services, HPA, Ingress)
- Enforce fine-grained security via RBAC roles, network policies, and service-account isolation
- Ensure robust observability via Prometheus, Grafana, and centralized logs with Loki
- Validate functionality with unit, integration, and end-to-end tests
- Automate build, test, and deploy via a GitHub Actions CI/CD pipeline

### *Bonus Features:*

- Event-driven cascade delete via RabbitMQ
- Inter-service authentication using signed JWTs
- Two-tier API rate limiting and throttling: Ingress-level limits plus service-level middleware enforcement

# 2. Architecture & Design

## 2.1 System Context

The solution is structured as a three-tier microservices platform, with clear separation of concerns and independent deplorability:

### 1. User Interaction Layer (web-interface)

- A React-based frontend served from its own container.
- Acts as the single entry point for end users, routing requests to the appropriate backend service.
- Communicates over HTTP to the API gateway (Ingress) and includes the user's JWT in each request header.

### 2. API Layer (user-service & product-service)

- user-service handles all user-related operations: registration, login, profile management, and self-deletion.
- product-service manages product CRUD operations, enforcing ownership checks on each action.
- Both services expose RESTful endpoints (/api/v2/users/* and /api/v2/products/*).
- *Authentication:*

    - User-level auth: Requests from the web-interface carry a user JWT signed with the user-service secret (stored in the user-jwt-secret Kubernetes Secret).
    - Inter-service auth: When one service calls another, it uses a separate service JWT signed with the service-auth secret (in the service-jwt-secret Kubernetes Secret), so user tokens aren't reused for internal calls.
- Each service runs in its own Kubernetes Deployment, with a Cluster IP Service for internal routing.

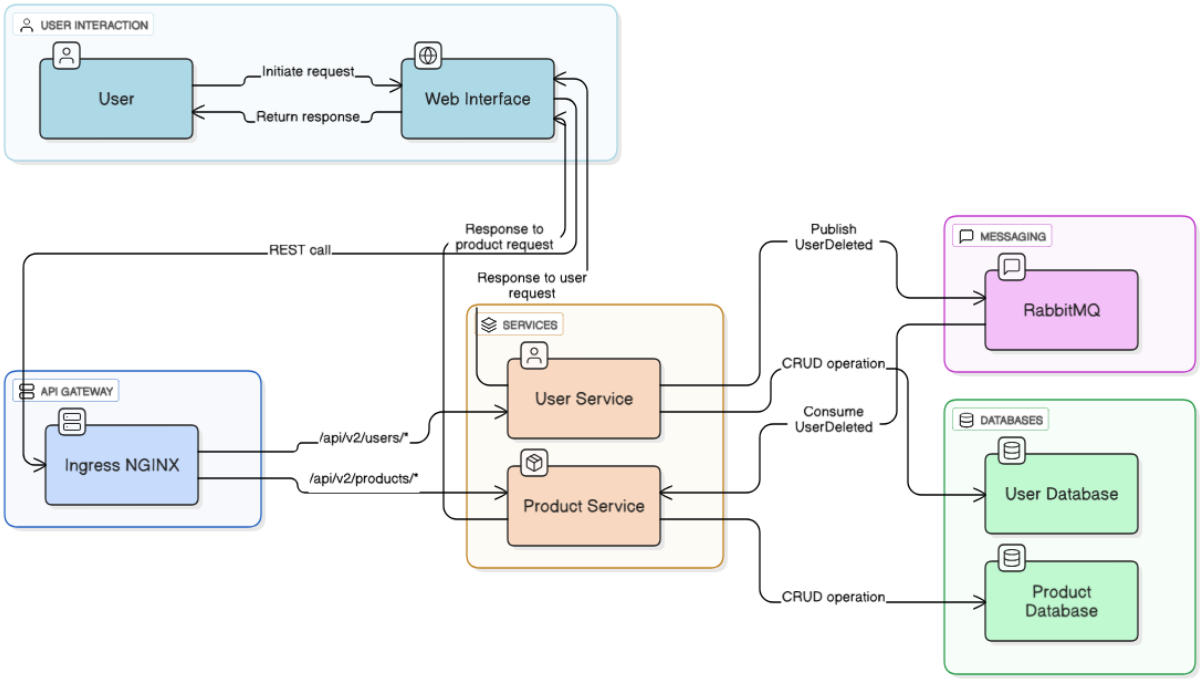
## 3. Persistence Layer (user-mongodb & product-mongodb)

- Two separate MongoDB instances, one per service, ensuring data isolation.
- Each runs as a Stateful Set (or Deployment) with a dedicated PVC; credentials are stored securely in Kubernetes Secrets.

## 4. Infrastructure & Cross-cutting Concerns

- Ingress Controller manages HTTP routing and the first tier of rate limiting.
- RBAC & Network Policies enforce least-privilege access between pods and to the Kubernetes API.
- Prometheus & Grafana collect and visualize metrics; Loki aggregates logs for centralized querying.
- RabbitMQ supports event-driven behaviors, such as cascade-deleting products when a user account is removed.

- CI/CD Pipeline automates build, test, and deploy, including manifest validation and rolling-update deployments via GitHub Actions.



## 2.2 Sequence Diagrams

### 1. Registration

- User → Web Interface: submits email & password to /api/v2/users/register
- Web Interface → User Service: forwards registration request
- User Service → User Database: stores new user
- User Database → User Service: confirms creation
- User Service → Web Interface: returns 201 Created
- Web Interface → User: shows confirmation

### 2. Login

- User → Web Interface: posts credentials to /api/v2/users/login
- Web Interface → User Service: forwards login request
- User Service → User Database: retrieves user record
- User Service → JWT Utils: verifies password & signs token
- User Service → Web Interface: returns JWT
- Web Interface → User: stores token for future calls

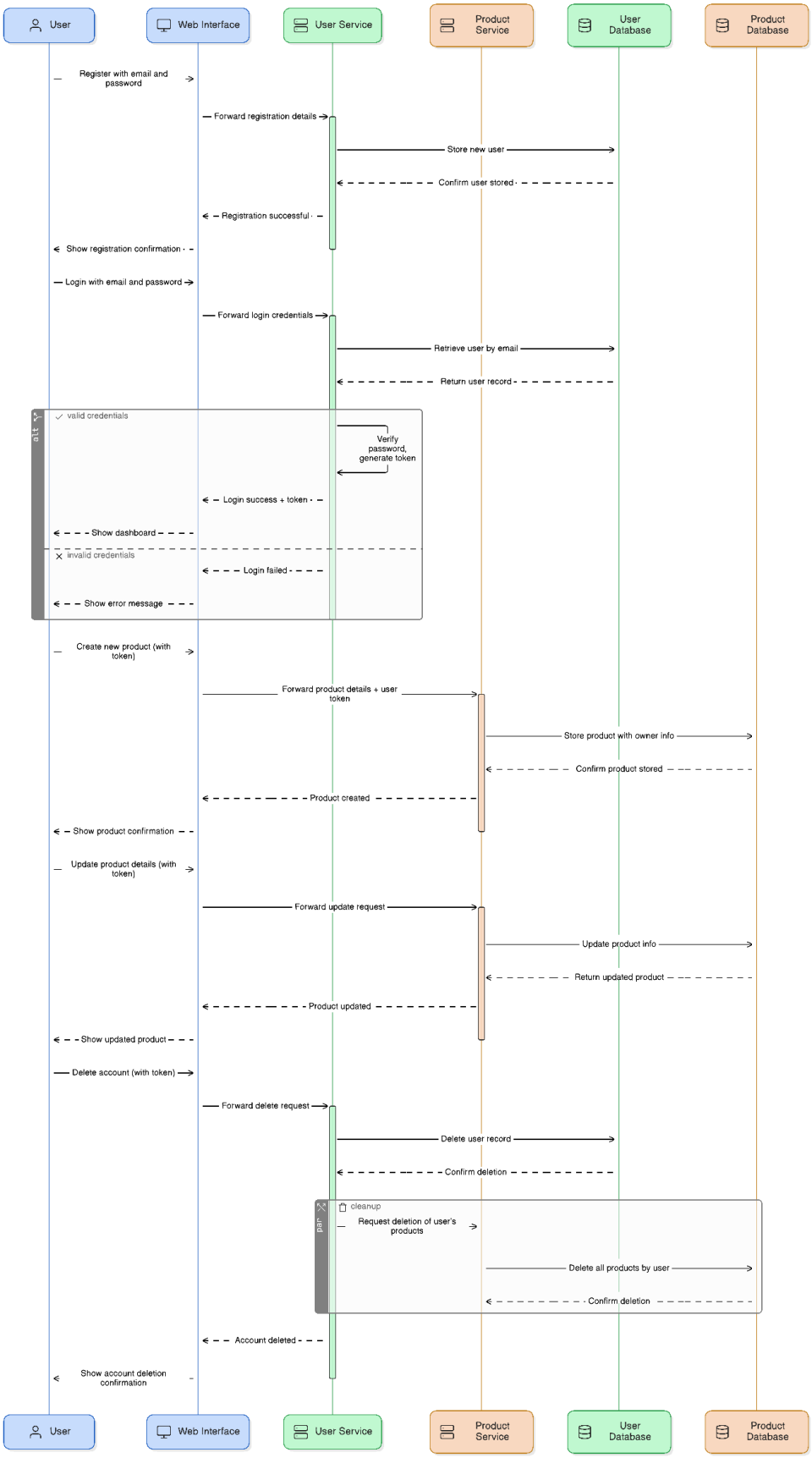
## 3. Create & Update Product

- User → Web Interface: sends POST /api/v2/products with JWT
- Web Interface → Product Service: forwards create request
- Product Service → Product Database: inserts new product (ownerId = userId)
- Product Database → Product Service: confirms creation
- Product Service → Web Interface: returns new product
- Web Interface → User: displays product
- User → Web Interface: sends PUT /api/v2/products/id/{productId} with updated fields & JWT
- Web Interface → Product Service: forwards update request
- Product Service → Product Database: applies update
- Product Database → Product Service: confirms update
- Product Service → Web Interface: returns updated product
- Web Interface → User: displays updated product

## 4. User Deletion & Cascade Delete

- User → Web Interface: deletes own account (DELETE /api/v2/users/me) with JWT

- Web Interface → User Service: forwards delete request
- User Service → User Database: removes user record
- User Database → User Service: confirms deletion
- User Service → RabbitMQ: publishes user.deleted event
- User Service → Web Interface: returns deletion confirmation
- RabbitMQ → Product Service: delivers user.deleted event
- Product Service → Product Database: deletes all products for that user
- Product Database → Product Service: confirms cascade deletion


User
Web Interface
User Service
Product Service
User Database
Product Database
Register with email and password
Forward registration details
Store new user
Confirm user stored
Registration successful
Show registration confirmation
Login with email and password
Forward login credentials
Retrieve user by email
Return user record
alt
valid credentials
Verify password, generate token
Login success + token
Show dashboard
invalid credentials
Login failed
Show error message
Create new product (with token)
Forward product details + user token
Store product with owner info
Confirm product stored
Product created
Show product confirmation
Update product details (with token)
Forward update request
Update product info
Return updated product
Product updated
Show updated product
Delete account (with token)
Forward delete request
Delete user record
Confirm deletion
par
cleanup
Request deletion of user's products
Delete all products by user
Confirm deletion
Account deleted
Show account deletion confirmation

# 3. Docker Implementation

## 3.1 Dockerfile Configuration

The service image is built from the official node:18-alpine base to minimize size and surface area. A dedicated non-root group (hle37) and user (thomas) are created and assigned ownership of the /app directory. Dependencies are installed in two steps—first copying package*.json to leverage Docker's layer cache, then running npm install—before copying the remainder of the application code. File permissions are corrected via chown to ensure the non-root user has read/write access. The container switches to the unprivileged thomas user before exposing port 3001 and starting the server with npm start. This pattern enforces the principle of least privilege and keeps the final image lean.

## 3.2 .dockerignore

The .dockerignore file excludes directories and files irrelevant to runtime:

- Build artifacts (build, dist)
- Test folders (__tests__, coverage)
- Local dependencies (node_modules)
- Kubernetes manifests (k8s)
- Environment files and logs (.env*, *.log)
- Miscellaneous (*.bat, .git*)

By preventing these items from being sent to the Docker daemon, build performance improves and image size is further reduced.

## 3.3 Docker Compose Setup

A single docker-compose.yml brings up the full stack with durable storage, health checks, and correct startup order:

- **Databases:** two MongoDB services (user-service-db on 27017, product-service-db on 27018) each backed by its own named volume and running a ping-based health check.
- **Message Broker:** RabbitMQ with AMQP and management ports exposed, a startup grace period, and health probing.
- **Back-end Services:** user-service (port 3000) and product-service (port 3001) build from their source folders, remain stateless, restart on failure, and only start after their DB and RabbitMQ report healthy—each verifies readiness via its /health endpoint.
- **Web Interface:** builds from ./web-interface, exposes port 80, depends on both back ends, and restarts on failure.

Persistent volumes are declared at the bottom to ensure database data survives container restarts. Startup is as simple as:

```
docker compose up –build
```

## 3.4 Security Best Practices

Images and orchestration adhere to industry-standard hardening guidelines:

- **Non-root execution:** Dockerfiles create and switch to an unprivileged user before starting the app.
- **Secret management:** JWT keys, database credentials, and other sensitive values are never committed to source control; they're supplied at runtime via environment variables or Docker secrets.
- **Minimal footprint:** Multi-stage builds strip out dev packages, and base images are kept lean (Alpine or Distroless) to reduce attack surface.
- **Layer optimisation:** Only necessary files (e.g., compiled code and package.json) are copied into the final image, cutting down on unnecessary layers and potential vulnerabilities.

# 4. Kubernetes Deployment

## 4.1 Prerequisites

- Docker Desktop with Kubernetes integration enabled.
- kubectl v1.33.3, extract to an empty folder (remember to add the path to the system variables).
- Helm v3.17.4, extract to an empty folder (remember to add the path to the system variables). With the prometheus-community and grafana chart repositories added:

  ```
  helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
  helm repo add grafana https://grafana.github.io/helm-charts
  ```
- Node.js v22.17.0 and npm (for running test suites locally)

## 4.2 Deploying to Kubernetes

### 1. Create the namespace

Run kubectl create namespace my-services to establish the my-services namespace for all application resources.

### 2. Install the Ingress controller

Apply the official NGINX Ingress manifest from GitHub:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/cloud/deploy.yaml
```

## 3. Deploy and patch the Metrics Server

- *Deploy the Metrics Server*

  ```
  kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
  ```

- *Read the current deployment manifest*

  ```
  kubectl get deployment metrics-server -n kube-system -o yaml
  ```

- *Copy the content to k8s/custom/metrics-server-patch.yaml*

- *Edit and add the --kubelet-insecure-tls flag under*

  ```
  spec:
       containers:
            - args:
  ```

- *Reapply with*

  ```
  kubectl apply -f k8s/custom/metrics-server-patch.yaml
  ```

## 4. Install the Prometheus, Grafana & Loki stack

*In a monitoring namespace (create it if needed), run:*

```
helm install prometheus-stack prometheus-community/kube-prometheus-stack --namespace monitoring --create-namespace --set nodeExporter.enabled=false
```

```
helm install loki-stack grafana/loki-stack --namespace monitoring
--create-namespace --set promtail.enabled=true
```

## 5. Apply shared configuration

*In the my-services namespace, deploy all environment and infrastructure manifests by applying:*

- configmaps/ (contains configmaps.yaml)
- secrets/ (contains secrets.yaml)
- ingress/ (ingress-configuration.yaml, ingress-service.yaml, nginx-configuration.yaml)
- rabbitmq/ (rabbitmq-deployment.yaml, rabbitmq-service.yaml)
- rbac/ (product-service-account.yaml, product-service-role.yaml, user-service-account.yaml, user-service-role.yaml)
- networking/ (deny-all.yaml; allow-*.yaml files)

*Then patch the Loki stack to enable JSON log parsing:*

```
helm upgrade loki-stack grafana/loki-stack -n monitoring -f
k8s/custom/promtail-values.yaml
```

## 6. Deploy application services

*Apply each service's Kubernetes manifests:*

- **ServiceAccounts & RoleBindings:** user-rolebinding.yaml, product-rolebinding.yaml

- **HorizontalPodAutoscalers:** user-hpa.yaml, product-hpa.yaml, web-hpa.yaml
- **PersistentVolumeClaims:** mongo-pvc.yaml
- **Deployments & Services:** user-deployment.yaml, user-service.yaml, product-deployment.yaml, product-service.yaml, web-deployment.yaml, web-service.yaml, mongo-deployment.yaml, mongo-service.yaml

*Finally, verify that each rollout completes successfully by running kubectl rollout status on each Deployment, then confirm all pods and services are healthy with:*

```
kubectl get all -n my-services
```

# 4.3 Rollout Update

When a service is updated, I generate a unique Docker image tag—composed of the Git commit SHA and the current UTC timestamp—to guarantee immutability. The typical workflow is:

## 1. Generate the tag

- Retrieve the short SHA via git rev-parse --short HEAD.
- Append the UTC timestamp in YYYYMMDDHHMMSS format (e.g. with Python's datetime.now(UTC).strftime("%Y%m%d%H%M%S")).
- Combine them into SHA-TIMESTAMP.

## 2. Build the image

- Run docker build -t <service-name>:<tag> <service-directory> to produce the new image.

## 3. Update the Deployment

- Execute kubectl set image deployment/<service-name> <service-name>=<service-name>:<tag> -n my-services to point the Deployment at the new image.

## 4. Watch the rollout

- Monitor progress with kubectl rollout status deployment/<service-name> -n my-services until the update completes successfully.

# 4.4 Running the Test Suites

### *Unit Tests (100% coverage)*

All unit tests reside in each service's __tests__ folder. They cover authentication logic, authorization checks, rate-limiting middleware, token signing, model validation, and route handlers. Example test files include auth.test.js, authorize.test.js, signServiceToken.test.js, userRoutesV2.test.js, productRoutesV2.test.js, and others. To run them:

- Change into the user-service directory and run npm test.
- Repeat in the product-service directory.

### *Integration Tests*

Located in the root __tests__ folder, the integration suite verifies end-to-end interaction between services and databases. It includes integration-create-product.test.js and integration-delete-user-products.test.js. Before running:

- Build and start all containers with docker-compose up -d --build.
- Then execute:
    - npx jest __tests__/integration-create-product.test.js --runInBand --detectOpenHandles
    - npx jest __tests__/integration-delete-user-products.test.js --runInBand --detectOpenHandles

### *End-to-End Tests*

Also in the root __tests__ folder, the user-product-flow.test.js script exercises the full user-to-product lifecycle (registration, login, product CRUD, account deletion, cascade delete). After bringing up the stack via docker-compose up -d --build, run:

- npx jest __tests__/user-product-flow.test.js --runInBand --detectOpenHandles

All suites are integrated into CI to run automatically on every push, ensuring that unit, integration, and E2E tests pass before any deployment.

## 4.5 CI/CD (GitHub Actions)

Continuous integration and delivery are implemented via three GitHub Actions workflows placed in the .github/workflows directory:

- Validate Kubernetes Manifests (kubeval.yml)
- User Service Unit Tests (user-service.yml)
- Product Service Unit Tests (product-service.yml)
- Integration & End-to-End Tests (integration-user-delete.yml)

These workflows trigger on every push to main and on pull requests targeting main. All jobs run on ubuntu-latest runners and perform the following steps:

## 1. Checkout & Environment Setup

The repository is checked out and Node.js (v18) is configured for the runner.

## 2. Dependency Installation

Service-specific (user-service or product-service) and root-level dependencies are installed via npm install.

## 3. Unit Testing

- user-service.yml and product-service.yml execute their respective unit test suites, located under each service's __tests__ folder.
- A lightweight Docker image is built locally for validation but is not pushed to any external registry.

## 4. Integration & End-to-End Testing

- integration-user-delete.yml installs Docker Compose, builds all services, and starts the full stack (docker compose up -d --build).
- Health checks poll the user-service and product-service endpoints until readiness.
- Jest-based integration and end-to-end tests run against the live stack to verify scenarios such as user deletion and cascade-delete behavior.

## 5. Deployment

Kubernetes manifests (located under each service's k8s/ folder) reference the locally built images with imagePullPolicy: IfNotPresent. After successful tests, manifests are applied to the my-services namespace using kubectl apply -n my-services -f <service>/k8s/, and rolling updates are confirmed by running kubectl rollout status for each Deployment.

## 6. Status Reporting

Workflow outcomes appear automatically in GitHub's status checks.

# 5. Innovation & Extras

In addition to the core requirements, this project incorporates three standout features that demonstrate professional-grade design and deliver tangible business value

## a) Event-Driven Cascade Delete via RabbitMQ

- **What it is:** When a user deletes their account, the user-service publishes a user.deleted event to RabbitMQ. The product-service subscribes to that queue and automatically removes all products owned by the deleted user.
- **Value:**
  - Decouples services—no tight coupling or synchronous HTTP calls for cleanup.
  - Improves reliability and resilience: if the product-service is temporarily down, events queue up and are processed once it recovers.
  - Simplifies future extensions (e.g., sending a “user.deleted” notification email) by adding new subscribers.

## b) Inter-Service Authentication with Separate JWTs

- **What it is:**
  - User-level tokens (for requests from the web UI) are signed with the user-jwt-secret.
  - Service-level tokens (for calls between user-service and product-service) use a different service-jwt-secret.
  - Both secrets are stored in Kubernetes Secrets and injected at runtime.

- **Value:**
  - Ensures that internal service calls can't be forged with stolen user tokens.
  - Enables fine-grained permission scoping and key rotation per trust boundary.
  - Lays groundwork for mutual TLS if needed in future releases.

## c) Two-Tier API Rate Limiting & Throttling

- **Ingress-Level Limits:** NGINX Ingress annotations enforce a global cap on requests (limit-rps, limit-connections) at the edge, protecting the cluster from volumetric attacks.
- **Service-Level Middleware:** Each service implements its own rate limiter (e.g., fixed window or token bucket) per user or API key, ensuring fair usage and preventing abuse even if an attacker circumvents the ingress.
- **Value:**
  - Shields core infrastructure from spikes and DDoS-style bursts.
  - Provides better user experience by isolating misbehaving clients.
  - Offers layered defense: if one layer is misconfigured or overwhelmed, the other still protects the backend.