RadiantIQ

UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

Project's acronym:	RADIANTIQ
Project's title:	RadiantIQ
Start date:	08/03/2024

D2 - RadiantIQ Agile Methodology

WP1:	Software Specification
Task 2.1:	Define Agile Methodology
Submission date:	27/05/2024
Responsible:	RadiantIQ develop team
Version:	2.0
Status:	Completed
Author(s):	Lorenzo Cattai, Gabriele Pernici, Anh Tu Duong, Lorenzo Negut
Deliverable type:	DOCUMENT

Version list:

Version	Authors	Date	Description
0.1	Anh Tu Duong	03/05/2024	Define the scheme of the document

Contents

Acronyms

Acronym	Description
FRS	Functional Requirement Specification
FSD	Functional Specification Document
FLOSS	Free/Libre Open Source Software
LLMs	Large Language Models
UML	Unified Modeling Language
UniTn	University of Trento

1 Introduction

2 Architecture description

2.1 Overview

RadiantIQ is designed using a microservices architecture to ensure scalability, flexibility, and maintainability. This section provides a detailed description of the architecture, including the components, technologies used, and the communication flow between services.

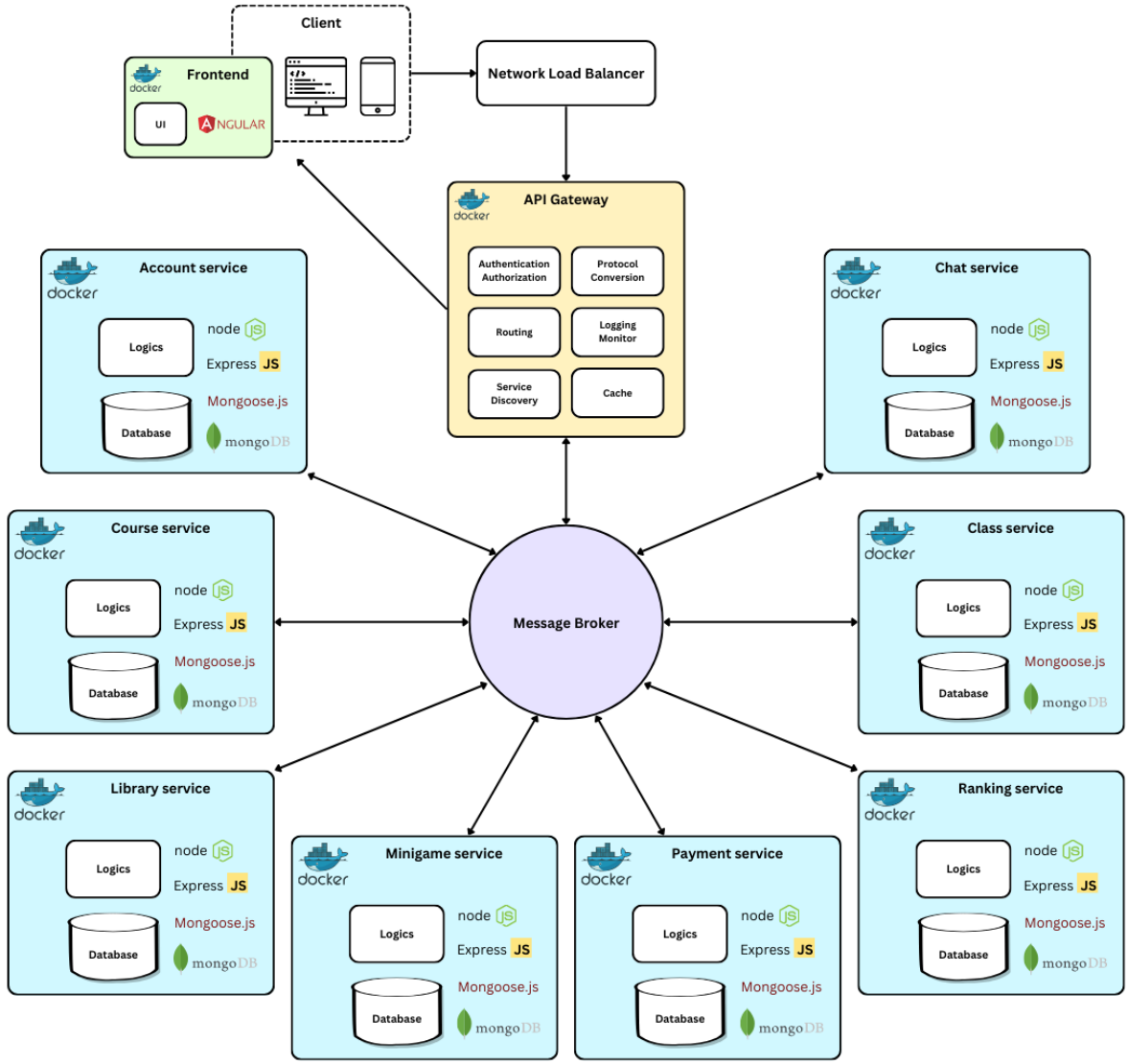

Figure 2: RadiantIQ - Architecture description diagram

2.2 Architectural Components

2.2.1 Frontend

- **Technology Stack**: Angular, NgBootstrap
- **Description**: The frontend is responsible for delivering a responsive and interactive user interface. It communicates with the backend services through the API Gateway.

2.2.2 API Gateway

Responsibilities:

- **Authentication and Authorization**: Verifies user credentials and permissions.
- **Routing**: Forwards requests to the appropriate backend services.
- **Service Discovery**: Determines the location of microservices.
- **Protocol Conversion**: Converts protocols (e.g., HTTP to WebSockets).
- **Logging and Monitoring**: Tracks and logs requests for monitoring and debugging.
- **Caching**: Stores frequently accessed data to improve performance.

2.2.3 Microservices

Each microservice is designed to handle a specific domain within the RadiantIQ platform. They are developed using Node.js and Express, and use MongoDB for data storage. Each service is deployed in a separate container, ensuring isolation and scalability.

- **Account Service**: Manages user accounts, including registration, login, and profile management.
- **Chat Service**: Handles real-time messaging between users.
- **Class Service**: Manages virtual classrooms, including scheduling and attendance.
- **Course Service**: Manages course content, enrollment, and progress tracking.
- **Library Service**: Provides access to educational resources, materials and allows users to create and share collections.
- **Minigame Service**: Create an environments for external developers to create minigames.
- **Payment Service**: Manages payment processing for course enrollments and other transactions.
- **Ranking Service**: Tracks and displays user rankings and achievements.

2.2.4 Message Broker

Description: Facilitates communication between microservices using a publish-subscribe model. Ensures that messages are reliably delivered to the appropriate services.

2.3 Communication Flow

1. **Client Request**:
   - The client (frontend) sends a request to the Network Load Balancer.
   - The Network Load Balancer forwards the request to the API Gateway.
2. **API Gateway Processing**:
   - The API Gateway authenticates the request and checks user authorization.
   - It routes the request to the appropriate service based on the endpoint and request data.
3. **Service Interaction via Message Broker**:
   - The API Gateway forwards the request to the Message Broker.
   - The Message Broker directs the request to the corresponding microservice (e.g., Course Service for course-related operations).

4. **Inter-service Communication**:
    - Services communicate with each other through the Message Broker using GraphQL API. This ensures decoupled and asynchronous communication, enhancing scalability and reliability.
5. **Response Handling**:
    - The target microservice processes the request and sends the response back to the API Gateway through the Message Broker.
    - The API Gateway returns the response to the client.

2.4 Deployment and Scalability

- Each microservice is containerized using Docker, ensuring consistent environments across development, testing, and production.
- Services can be independently scaled based on load and performance requirements.
- Kubernetes (or another orchestration tool) can be used to manage container deployment, scaling, and load balancing.
- Finally the services are deployed on a cloud provider Render.com to expose to the internet.

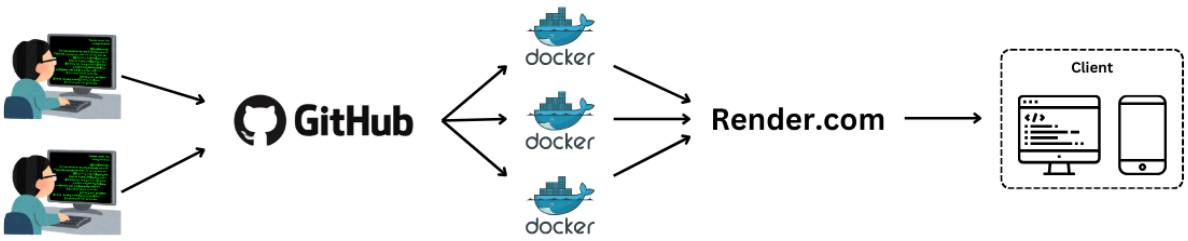

Figure 3: RadiantIQ - Develop and deploy workflow

2.5 Monitoring and Maintenance

- **Logging and Monitoring**: Integrated within the API Gateway and microservices to track performance, errors, and usage patterns.
- **Continuous Integration/Continuous Deployment (CI/CD)**: Using Github and Github Action to automate pipelines for testing and deploying new code changes to ensure rapid and reliable updates.

3 Product backlog

4 Definition of tests

5 Git strategy

6 Definition of done

7 How-to demo

8 Sprint 1

9 Conclusion