**Books Library**

This is the backend for an application which allows users to manage books, book reading status and share them in a community. The library is secured, only logged in users can use the functionalities.

**Technologies Used**

* **Backend**: Java Spring Boot
* **Containerization**: Docker, Docker Compose
* **Load Balancer**: NGINX
* **Database**: PostgreSQL
* **Event Streaming**: Apache Kafka
* **Security**: JWT Authentication

**Micro Services**

1. **Authentication Microservice**

The Authentication Microservice handles user authentication, registration, and JWT-based security. It ensures secure user management and token generation for authentication and authorization.

**Key Features**

* User registration and account creation.
* Secure login with password hashing and validation.
* JWT token generation for authenticated access.
* Public key exposure via JWKS for JWT verification.

**REST API Endpoints**

* POST /register – Registers a new user in the system.
* POST /login – Authenticates a user and returns a JWT token.
* GET /jwks – Provides the JSON Web Key Set (JWKS) containing the public key used for JWT signature verification.

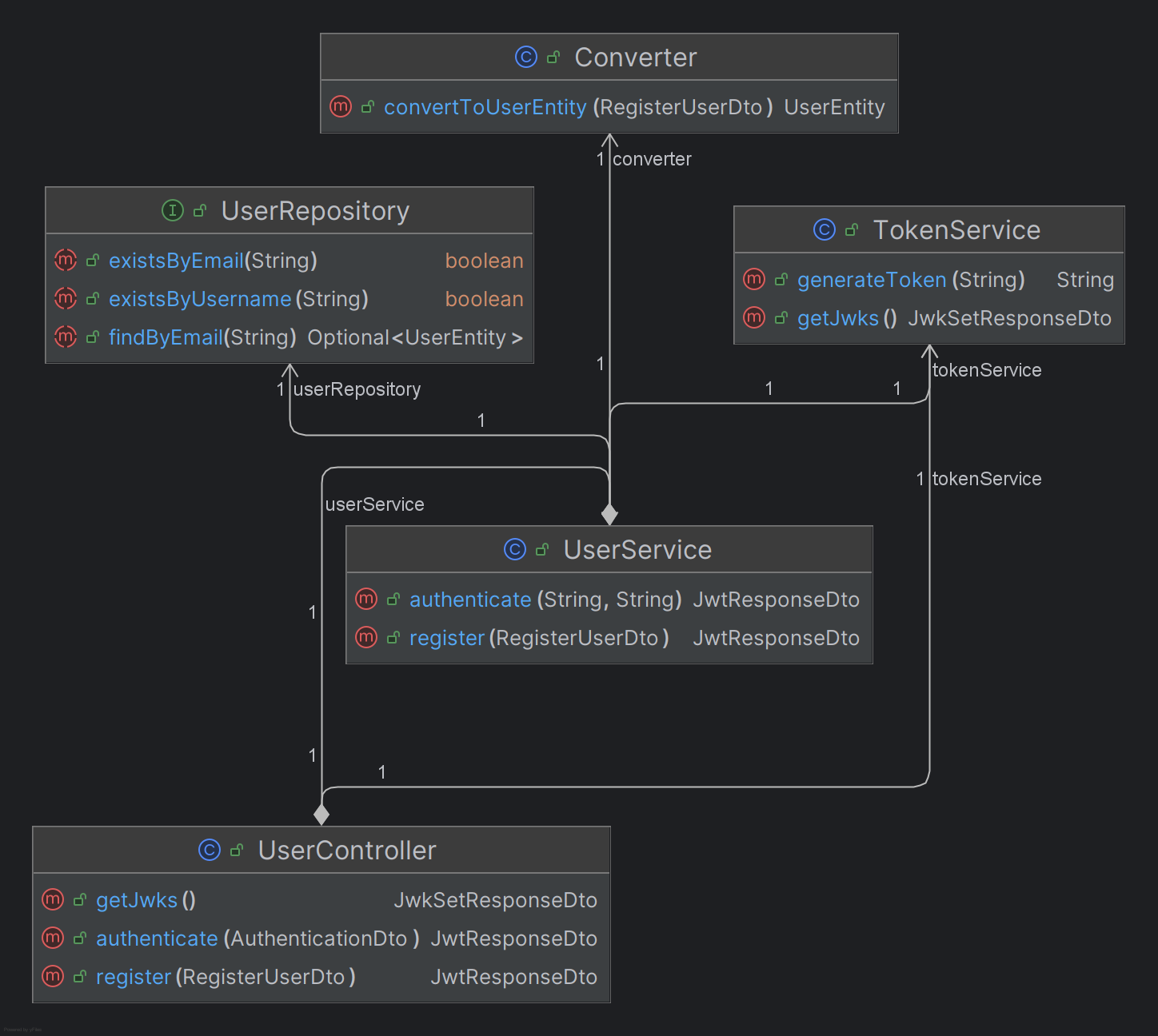
**Core Components**

* UserController – Manages authentication and registration endpoints.
* UserService – Handles business logic for authentication and user management.
* TokenService – Generates and validates JWT tokens using RSA key pairs.
* SecurityConfig – Configures security, CORS, and password encoding.
* UserRepository – Manages persistence of user entities.

**Security & Authentication**

* User authentication is managed via an OAuth Authorization Server, which issues JWT tokens.
* RSA key pair is used for asymmetric signing, with the public key retrievable via /jwks.
* Passwords are securely hashed using BCrypt.

This is the corresponding UML diagram generated with Intellij's diagrams plugin:



**2. Book Microservice**

The Book Service is responsible for managing books and their assignments to users. It provides REST API endpoints for book management and integrates authentication and logging mechanisms via Kafka.

**Key Features**

* Book Management: Create, update, delete, and retrieve books.
* Assignment System: Assign books to users and retrieve assigned books.
* Security & Authentication: Uses OAuth2 with JWT for securing endpoints.
* Logging & Monitoring: Logs method executions to Kafka.

**REST API Endpoints**

**Book Management**

* GET /all – Retrieves all books.
* GET /{id} – Retrieves a book by its ID.
* POST /save – Creates a new book.
* PUT /update – Updates a book's details.
* PUT /update-status – Updates a book's status.
* DELETE /delete/{id} – Deletes a book.

**Book Assignment**

* POST /assign/{bookId} – Assigns a book to a user.
* POST /all-assigned-to – Retrieves all books assigned to a specific user.

**User Management**

* GET /user/{username} – Retrieves user details.
* GET /user – Retrieves all users.

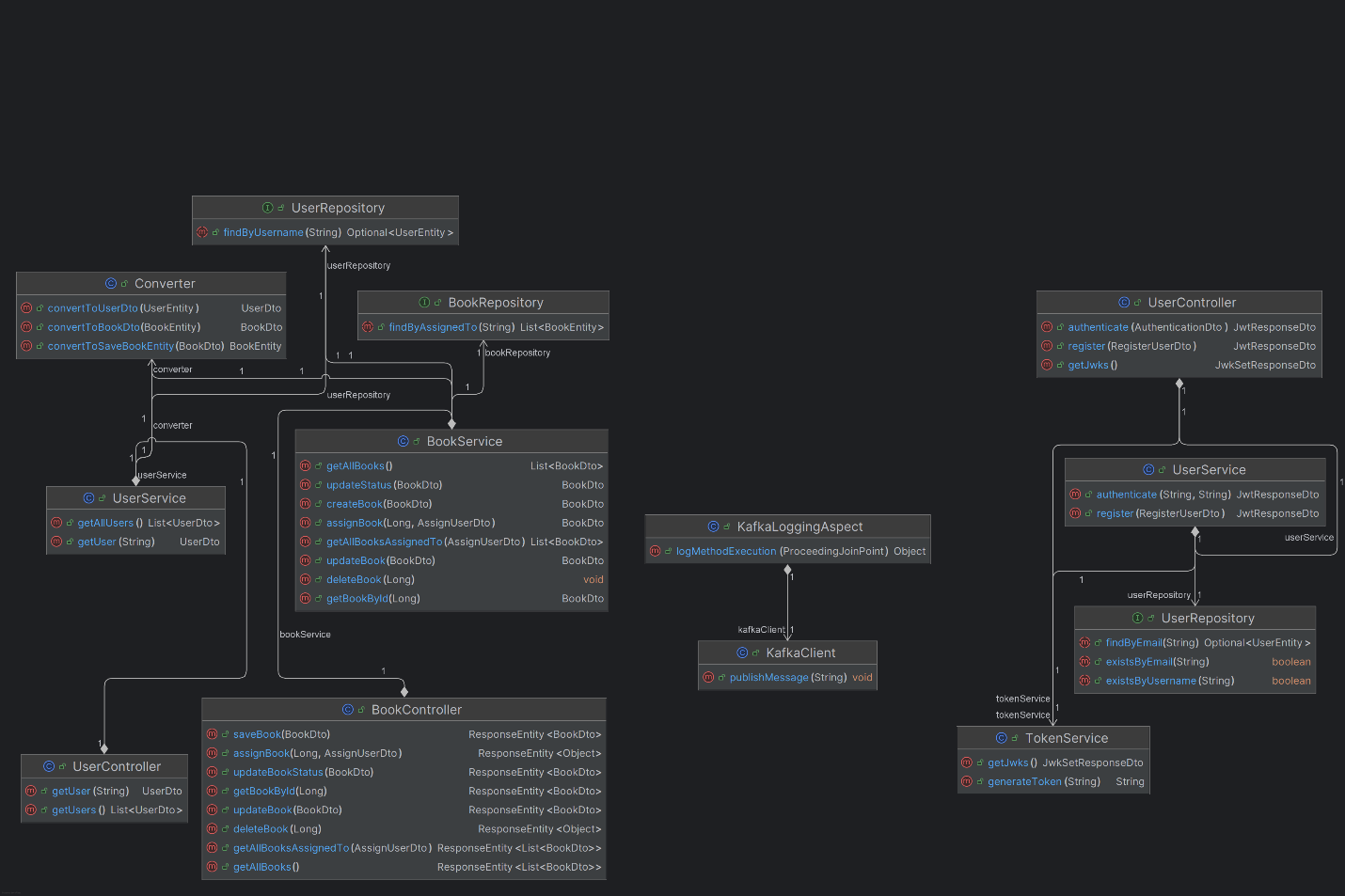
**Security & Authentication**

* Uses OAuth2 Resource Server with JWT authentication.
* Enforces method-level security with @PreAuthorize("isAuthenticated()").
* Allows unrestricted access to WebSocket endpoints (/sba-websocket\*, /topic\*).

**Logging & Kafka Integration**

* Aspect-Oriented Logging: @LogToKafka annotation logs method calls.
* KafkaClient: Publishes logs to a Kafka topic.
* Retry Mechanism: Ensures log delivery via RetryTemplate.

This is the corresponding UML diagram generated with Intellij's diagrams plugin:



* 1. **Logger Microservice**

The **Logger Microservice** is responsible for centralizing logs from other microservices. It listens to log messages published to a **Kafka topic** and records them for monitoring and debugging purposes.

**How It Works**

* The microservice is set up as a **Kafka consumer**, meaning it listens for messages on a specific Kafka topic.
* It receives log messages from other microservices and logs them using **SLF4J**.
* This allows logs from multiple instances of different microservices to be aggregated in a single place.

**Key Component**

* **LoggerListener**:
  + A Kafka consumer that listens to the **log-message** topic.
  + When a new log message arrives, it is processed and logged.
  + The Kafka topic name is configured dynamically using application properties (kafka-topic).

This microservice ensures that logs are captured efficiently and can be accessed for monitoring, debugging, and auditing.

**Nginx as API Gateway and Load Balancer**

In this microservices architecture, **Nginx** is used as an API Gateway and Load Balancer. It acts as a single entry point for all incoming requests, routing them to the appropriate microservices (authentication, book, logger). This setup ensures scalability, security, and efficient request distribution.

**1. API Gateway Role**

Nginx serves as an **API Gateway**, meaning it:

* Exposes a unified entry point (http://localhost:4000).
* Routes API requests to the correct backend services (authentication, book, logger).
* Handles **CORS, rate limiting, and security policies**.
* Hides internal microservices from the client, improving security.

**2. Load Balancing**

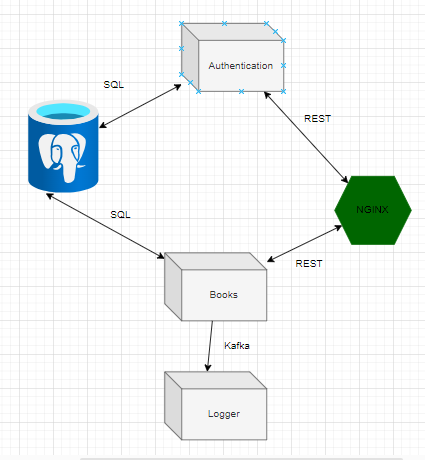
Load balancing helps distribute traffic efficiently across multiple instances of a service, preventing any single instance from being overwhelmed.

* The book microservice is deployed with **2 replicas** (deploy.replicas: 2), meaning there are two running instances.
* Nginx will **distribute incoming requests** across these instances.
* This improves system availability and performance.

**3. Reverse Proxy**

Nginx functions as a **reverse proxy**, meaning it:

* Receives client requests.
* Determines which microservice should handle the request.
* Forwards the request and returns the response to the client.



**Docker tutorial**

Docker is a platform for developing, shipping, and running applications inside lightweight, isolated containers. Containers include everything needed to run an application, making them portable and consistent across different environments.

**2. Installing Docker**

**Windows & Mac:**

1. **Download Docker Desktop** from Docker’s official website: https://docs.docker.com/engine/install/
2. **Install it** by following the on-screen instructions.
3. **Enable WSL2 Backend (for Windows users).**
4. **Start Docker Desktop** and ensure it is running by executing:

docker –version

**3. Basic Docker Commands**

* **Check Docker version:**

docker --version

* **List running containers:**

docker ps

* **List all containers (including stopped ones):**

docker ps -a

* **Start a container:**

docker start <container\_id>

* **Stop a container:**

docker stop <container\_id>

* **Remove a container:**

docker rm <container\_id>

* **Remove an image:**

docker rmi <image\_id>

* **View logs for a container:**

docker logs <container\_id>

* **Execute a command inside a running container:**

docker exec -it <container\_id> sh

**4. Using Docker Compose**

**What is Docker Compose?**

Docker Compose is a tool for defining and running multi-container Docker applications. It uses a docker-compose.yml file to configure application services.

**Setting Up Your Application with Docker Compose**

**Step 1: Create the docker-compose.yml File**

We will use my existing docker-compose.yml configuration.

version: "3.8"  
  
networks:  
 app-network:  
 driver: bridge  
  
services:  
 postgres-db:  
 image: postgres:17.2-alpine3.21  
 restart: always  
 environment:  
 POSTGRES\_PASSWORD: "postgres"  
 volumes:  
 - ./postgres-db-data:/var/lib/postgresql/data  
 ports:  
 - "5432:5432"  
 healthcheck:  
 test: [ "CMD-SHELL", "pg\_isready -U postgres" ]  
 interval: 10s  
 timeout: 5s  
 retries: 5  
 networks:  
 - app-network  
  
 authentication:  
 image: authentication:latest  
 restart: always  
 build:  
 dockerfile: Dockerfile  
 context: ./authentication  
 environment:  
 - POSTGRES\_HOST=postgres-db  
 - POSTGRES\_PORT=5432  
 - POSTGRES\_DATABASE=postgres  
 - POSTGRES\_USERNAME=postgres  
 - POSTGRES\_PASSWORD=postgres  
 - APPLICATION\_PORT=8080  
 ports:  
 - "8080"  
 deploy:  
 replicas: 1  
 depends\_on:  
 postgres-db:  
 condition: service\_healthy  
 volumes:  
 - ./logs/authentication/:/var/log/authentication  
 networks:  
 - app-network  
  
 product:  
 image: product:latest  
 restart: always  
 build:  
 dockerfile: Dockerfile  
 context: ./product  
 environment:  
 - POSTGRES\_HOST=postgres-db  
 - POSTGRES\_PORT=5432  
 - POSTGRES\_DATABASE=postgres  
 - POSTGRES\_USERNAME=postgres  
 - POSTGRES\_PASSWORD=postgres  
 - APPLICATION\_PORT=8060  
 - KAFKA\_SERVER=kafka  
 - KAFKA\_PORT=9092  
 - JWT\_ISSUER\_URI=http://localhost:8080  
 - JWK\_SET\_URI=http://host.docker.internal:4000/authentication/jwks  
 - RABBIT\_HOST=rabbitmq  
 - RABBIT\_PORT=5672  
 ports:  
 - "8060"  
 deploy:  
 mode: replicated  
 replicas: 2  
 volumes:  
 - ./logs/product/:/var/log/product  
 networks:  
 - app-network  
  
 fake-smtp-server:  
 image: gessnerfl/fake-smtp-server:latest  
 ports:  
 - "8025:8025" *#expose smtp port* - "5555:8080" *#expose web ui* - "8081:8081" *#expose management api* nginx:  
 image: nginx:latest  
 volumes:  
 - ./nginx.conf:/etc/nginx/nginx.conf:ro  
 ports:  
 - "4000:4000"  
 networks:  
 - app-network  
 depends\_on:  
 - authentication

**Step 2: Run Your Application**

Run the following command in the same directory as your docker-compose.yml file:

docker-compose up -d

This will:

* Download the required images if not available locally.
* Start all services in detached mode (-d runs them in the background).

**Step 3: Verify Running Containers**

Check the running containers:

docker ps

You should see services like postgres-db, authentication, product, and nginx running.

**Step 4: Stopping and Removing Containers**

To stop all running services, use:

docker-compose down

To remove all stopped containers, unused networks, and unused images:

docker system prune -a

**Step 5: Debugging Issues**

* **Check container logs:**

docker logs -f <container\_name>

* **Enter a running container shell:**

docker exec -it <container\_name> sh

* **List networks:**

docker network ls

* **Inspect a network:**

docker network inspect app-network

**5. Conclusion**

Now you can modify your application, rebuild the images, and restart the services with:

docker-compose up --build -d