**ByBud Microservices Documentation**https://github.com/nisjety/Bybud2

This guide provides a concise overview of how to set up and run the ByBud microservices architecture, and some insight into the core system components, communication patterns, and best practices.

**Getting Started**

1. To build all backend services, open a terminal in the project's root directory and run:

* mvn clean install

1. Once the build is complete, start the Kafka, Redis, and MongoDB containers:

* docker compose down -v && docker compose up -d
* It may take a moment to initialize all requirements.

1. Next, open separate terminals to run each microservice. For the Auth Gateway (port 8080):

* cd backend/services/auth-gateway
* mvn spring-boot:run

1. For the User Service (port 8083):

* cd backend/services/user-service
* mvn spring-boot:run

1. For the Delivery Service (port 8082):

* cd backend/services/delivery-service
* mvn spring-boot:run

1. And for the Eureka Server (port 8761):

* cd backend/services/eureka-server
* mvn spring-boot:run

1. To run the frontend, navigate to the frontend folder:

* cd frontend
* npm install
* npm run dev

It may take a few minutes before all services are fully operational. (up to 3 min)

**System Overview**

ByBud’s backend is built on a microservices approach using both synchronous REST calls and asynchronous messaging with Kafka. The core services include:

* **Authentication Gateway**: Validates and manages JWT tokens, serving as the main entry point for incoming requests.
* **User Service**: Handles user data and profiles, stores credentials in MongoDB, and publishes events like registration and updates.
* **Delivery Service**: Handles the creation and management of delivery orders, tracks status changes, and coordinates between customers and couriers.

Support for service discovery is provided by a **Eureka Server** this enables the gateway to find each microservice.

**Key Technologies**

The system uses Spring Boot Reactive (WebFlux) for non-blocking APIs, reactive MongoDB for data storage, reactive Redis for token sessions, custom reactive Kafka setup for event-driven communication, and JWT for stateless authentication. This allows for efficient, scalable, and resilient service operation.

**Authentication Flow**

When a user logs in at /api/auth/login, the Auth Gateway checks credentials against the User Service. If they're valid, it generates access and refresh tokens and stores token signatures in Redis. Then it returns the tokens to the client and sends an authentication event to Kafka. For protected endpoints, a JWT filter verifies the token's signature, ensuring it hasn't expired or been revoked. User details and roles are then propagated to downstream services via request headers.

**Delivery Management**

A new delivery request is sent by an authenticated client. The Auth Gateway confirms the user token, then forwards the request with user details to the Delivery Service, which creates a record in MongoDB and publishes a creation event to Kafka. When a courier accepts the delivery, the status changes to ASSIGNED, triggers another Kafka event, and updates relevant records accordingly.

**Token Management**

JWT signatures are kept in Redis and use a time-to-live (TTL) value for each expiration. Revoked tokens are blacklisted. Users can refresh their tokens with all valid refresh tokens. Multiple validation layers guarantee token security, and the separate storage of token signatures deters forbidden access.

**Data Propagation**

User information, including all names as well as roles, goes through request headers added by the Auth Gateway, asynchronous Kafka events, or direct service-to-service calls, relative to a specific context. It guarantees little duplication. Also, it keeps the system quite loosely coupled.

**Security and Resilience**

To guarantee secure communication among services, all internal services use secrets. All other resources have JWT checks, but only a small number of endpoints are publicly accessible. Kafka operations can be retried and extensively logged by each microservice. Circuit breakers help applications handle each timeout or unresponsive dependency gracefully, and Redis guarantees automatic removal of each expired token.

**Development Workflow**

A typical local workflow involves running MongoDB, Redis, and Kafka in containers, starting services with Docker Compose, and setting up environment variables in .env files. Complete end-to-end tests thoroughly verify the entire flow, while unit and integration tests carefully check individual services. Tools like Postman can authenticate and test routes using JWT tokens in request headers. The tokens make this possible. Kafka topics can be examined to confirm events are published and consumed correctly.

**Known Issues and Future Improvements**

Some methods are not fully operational yet, but as an MVP, the current state is sufficient for demonstration. Certain buttons, such as **Cancel Delivery** and **Unassign Delivery**, are placeholders for future functionality. Additionally, not all status updates are available in the dropdown menu.

The **customer name** display is currently undecided, as they are being evaluating whether to show the full name or username due to privacy concerns. A future update may include a **phone number field**, allowing couriers to contact customers directly.

For future enhancements, the plan is to introduce **profile pictures**, a **notification service**, and an improved authentication system, including **two-step verification** and **password reset functionality via email**. The primary goal of this project is to serve as a proof of concept, illustrating how the application would function in a real-world scenario. Feel free to test the system as you like.

This microservices architecture is designed to be flexible, with synchronous REST combined with event-driven messaging to maintain the system as scalable and reliable. Every service runs in isolation yet remains tightly integrated with common authentication, data, and monitoring patterns. There Is also 3 diagrams added to the project, created using mermaid. These diagrams include:

1. **System Architecture Diagram**: Shows the high-level architecture with all major components and their relationships. You can see how the client applications connect through the Auth Gateway to access backend services, and how data flows between services, databases, and the Kafka event.
2. **Authentication and Delivery Flow Diagram**: This sequence diagram illustrates the detailed process of:
   * User authentication and token generation
   * Creating a new delivery as an authenticated customer
   * A courier accepting a delivery assignment
   * It shows the exact sequence of operations and how information travels between services.
3. **System Data Flow Diagram**: This visualization focuses on how different types of data move through the system:
   * Request/Response data (HTTP headers, user information, response payloads)
   * Event data that flows through Kafka (authentication events, user events, delivery events)
   * Stored data in the various databases (user records, token information, delivery details)

These diagrams together provide a comprehensive understanding of the backend architecture from different views: component structure, process flow, and data movement. This multi-faceted view helps in understanding the system's behavior.

There is also a postman import to test the whole system using postman, it’s not fully automatic. It still needs some dynamic implementation of the tokens and userId and so on.