# Kemal Yılmaz 20239189

# Microservice Architecture Report

## Introduction

This report presents the microservice decomposition for my application. The decomposition ensures modularity, scalability, and resilience to failures. All communications between microservices are mediated through the GUI API service. This architecture allows for independent development, deployment, and scaling of each service. All four microservices that I created have their own databases in separate containers. In the data storage parts of the microservices, I'll explain which data the databases hold.

## Microservice Decomposition

### 1. Authentication Service

**Features:**

- User registration
- User login

**Data Storage:**

- Username
- Password

**Connections:**

- Communicates with the GUI API for user registration and login requests.

### 2. Event Service

**Features:**

- Event creation
- Public event listing
- Displaying details for a single event given the ID of the event
- Displaying details for a list of events given a list of event IDs

**Data Storage:**

- Unique event ID
- Event details (date, organizer, title, description, public/private info)

**Connections:**

- Communicates with the GUI API for:
    - Listing public events on the homepage
    - Showing the event details when the user clicks on the event title
    - Returning the details of events when another microservice needs it through the GUI

**3. Invite Service**

**Features:**

- Invite creation for a given event ID and list of usernames
- Handling user responses to invites
- Displaying invites that the user has
- Displaying the response status of users attending a certain event
- Displaying the response status of a single user for a certain event

**Data Storage:**

- Event ID
- Username
- User response

**Connections:**

- Communicates with the GUI API for:
    - Displaying the invites for the user
    - Updating the response of the user in the database when they respond to the invitation
    - Providing user attendance data when event details are displayed
    - Setting the event owner's status to 'Participating' and adding the event to the user’s calendar when an event is created
    - Adding the event ID to the user's calendar when a user decides to participate in an event

**4. Calendar Service**

**Features:**

- Displaying personal user calendars
- Managing calendar sharing
- Creating a calendar
- Adding an event to a user’s calendar
- Getting a list of users who can access a certain user’s calendar

**Data Storage:**

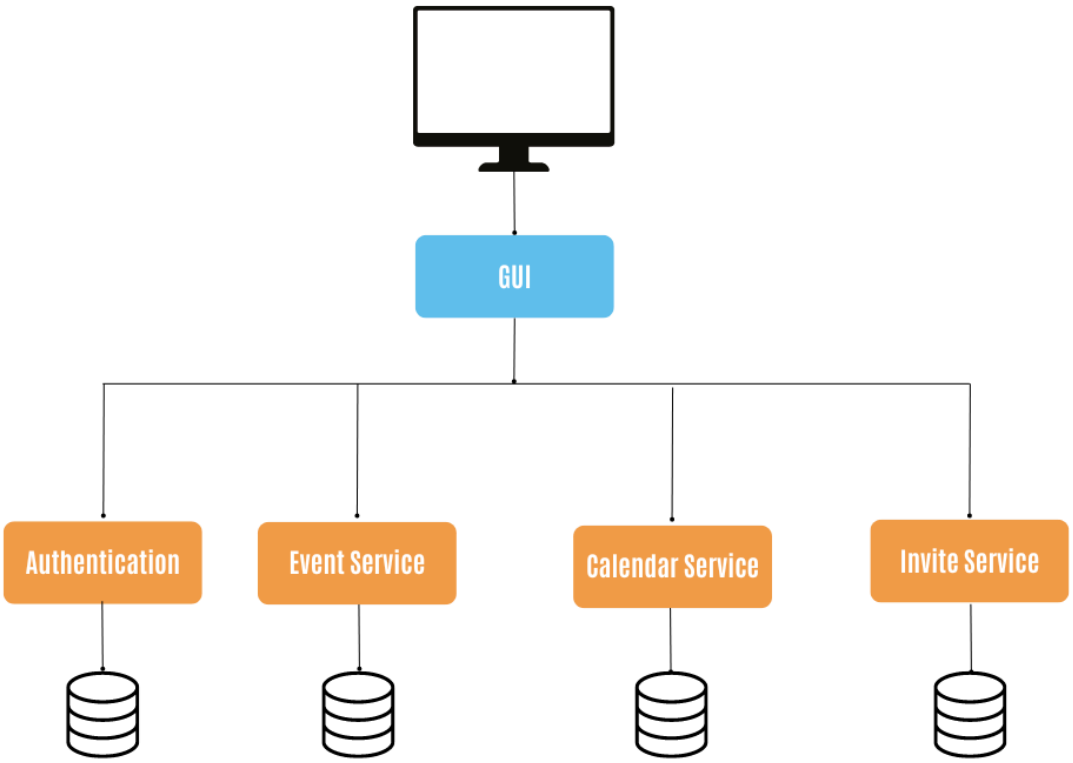
- Username
- List of event IDs a user is participating in
- Shared calendar permissions

**Connections:**

- Communicates with the GUI API for calendar viewing and sharing operations.

## Overview of Microservice Architecture

Below is an overview diagram of the microservice architecture:




## Explanation of Decomposition

### Grouping of Features

1. **Authentication Service**:
    - **Reasoning**: User registration and login are fundamental to all other services but are independent of other features. By isolating authentication, we enhance security and simplify user management.
    - **Consequences of Failure**: If this service fails, new users cannot register and existing users cannot log in. However, users already logged in can continue using other services.
2. **Event Service**:
    - **Reasoning**: Event creation and management are core functionalities. Grouping these together allows for centralized handling of event data and public event listings.

    - **Consequences of Failure**: If this service fails, users cannot create or view events. However, existing invites and calendar views remain unaffected.
3. **Invite Service**:
    - **Reasoning**: Invitations are closely tied to events but involve separate functionality (sending and responding to invites). Keeping this separate ensures event management remains uncluttered.
    - **Consequences of Failure**: If this service fails, users cannot send or respond to invites, but they can still view events and their calendars.
4. **Calendar Service**:
    - **Reasoning**: User calendars and sharing functionalities are naturally cohesive. Grouping these features simplifies calendar-related operations and permissions management.
    - **Consequences of Failure**: If this service fails, users cannot view their calendars or shared calendars, but event and invite operations remain functional.

## Communication and Data Flow

- All microservices communicate exclusively through the GUI API. This centralizes interactions and ensures a uniform interface for external clients.
- Data transfer between microservices also occurs through the GUI API, which handles requests and responses, thereby decoupling the services from direct interdependencies.

## Scalability

- **Modular Scaling**: Each microservice can be scaled independently based on its load. For example, the Authentication Service can be scaled to handle peak login times, while the Event Service can be scaled for heavy event creation and viewing loads.
- **Horizontal Scaling**: Services can be replicated across multiple instances to handle increased demand. Load balancing ensures efficient distribution of requests.

## Implementation

The implementation of the microservices includes defining the necessary API endpoints for each service. Below are the endpoints used for each microservice along with their documentation:

#### Authentication Service

1. **Register**
    - **Endpoint**: `/register`
    - **Description**: Registers a new user.
2. **Login**
    - **Endpoint**: `/login`
    - **Description**: Authenticates a user and provides a token.

#### Calendar Service

1. **Create Calendar**
    - **Endpoint**: `/calendar`
    - **Description**: Creates a new calendar for the user.
2. **Share Calendar**
    - **Endpoint**: `/calendar/<string:owner>/share`
    - **Description**: Shares a user's calendar with another user.
3. **View Calendar**
    - **Endpoint**: `/calendar/<string:owner>`
    - **Description**: Returns the ids of the events that are in the user's calendar.
4. **Add to Calendar**
    - **Endpoint**: `/calendar/<string:owner>/add`
    - **Description**: Adds an event to a user's calendar.
5. **Get Allowed Users**
    - **Endpoint**: `/calendar/shared/<string:owner>`
    - **Description**: Retrieves a list of users who can access a certain user's calendar.

**Event Service**

1. **Create Event**
    - **Endpoint**: `/create-event`
    - **Description**: Creates a new event.
2. **Public Events**
    - **Endpoint**: `/public-events`
    - **Description**: Lists all public events.
3. **Event Detail**
    - **Endpoint**: `/event/<string:event_id>`
    - **Description**: Displays details of a single event.
4. **Multiple Event Details**
    - **Endpoint**: `/events`
    - **Description**: Displays details of multiple events.

**Invite Service**

1. **Single Invite Status**
    - **Endpoint**: `/invites/status`
    - **Description**: Displays the response status of a single invite.
2. **Get Invitation Status**
    - **Endpoint**: `/get-invitation-status/<string:event_id>`
    - **Description**: Retrieves the invitation status for a specific event.
3. **Create Invite**
    - **Endpoint**: `/create-invite`
    - **Description**: Creates invites for a given event ID and list of usernames.
4. **Change Status**
    - **Endpoint**: `/change-status`
    - **Description**: Updates the response status of a user for an invite.

5. **Show Pending**
    - **Endpoint**: `/show-pending/<string:username>`
    - **Description**: Displays pending invites for a user.

## Conclusion

The microservice decomposition presented here enhances the modularity, scalability, and resilience of the application. By grouping related functionalities and managing inter-service communication through the GUI API, we ensure that each service remains focused and manageable. This architecture supports graceful degradation and efficient scaling, making it suitable for large user bases and high-traffic scenarios.