**Mini Moodle**

**Se-2224 Pernebek Amina**

**Assignment 4**

**User Management and Authentication Microservice**

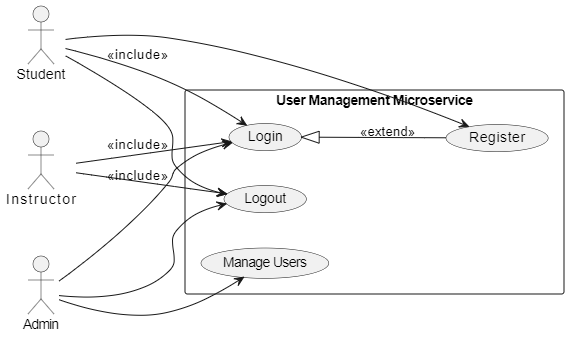
The User Management and Authentication Microservice is responsible for handling all user-related functionalities within the Mini Moodle system. This includes user registration, authentication (login/logout), and user management. This microservice is critical as it provides the necessary authentication and authorization mechanisms for the entire system.

**API Design**

For API design best practices, we follow Google's API Design Guide, which emphasizes consistency, simplicity, and clarity in API design.

**Use Case Diagram**

The following use case diagram outlines the interactions of various actors (Student, Instructor, Admin) with the User Management microservice in Mini Moodle:



**Architecture Choice & Justification**

**Project Structure**

The microservice will adhere to the clean architecture principles as described in [Evrone's Go Clean Template](https://github.com/evrone/go-clean-template/blob/master/README.md#clean-architecture). This ensures that the codebase is maintainable, scalable, and testable. The project will also use the API gateway architecture to handle request routing and service aggregation.

**Directory Structure**

/user-management

├── cmd

│ └── main.go

├── controllers

│ └── user\_controller.go

├── services

│ └── user\_service.go

├── repositories

│ └── user\_repository.go

├── models

│ └── user.go

├── routes

│ └── routes.go

├── utils

│ └── jwt.go

└── config

└── config.go

**Justification**

* **Separation of Concerns**: Each layer (controllers, services, repositories) has a distinct responsibility.
* **Maintainability**: Clean architecture facilitates easier maintenance and scaling.
* **Testability**: With clear boundaries between layers, unit testing and integration testing become simpler.
* **Scalability**: Microservice architecture allows independent scaling of different services.

**Mockup of the Project**

**API Endpoints**

**1. User Registration**

* **Endpoint**: /api/v1/register
* **Method**: POST
* **Request Body**:

{

"username": "string",

"email": "string",

"password": "string"

}

* **Response**:
  + **201 Created**:

{

"message": "User registered successfully",

"user": {

"id": "string",

"username": "string",

"email": "string"

}

}

**400 Bad Request**:

{

"message": "Error message"

}

**2. User Login**

* **Endpoint**: /api/v1/login
* **Method**: POST
* **Request Body**:

{

"email": "string",

"password": "string"

}

* **Response**:

**200 OK**:

{

"message": "Login successful",

"token": "JWT token"

}

**401 Unauthorized**:

{

"message": "Invalid email or password"

}

**3. User Logout**

* **Endpoint**: /api/v1/logout
* **Method**: POST
* **Headers**:

Authorization: Bearer <JWT token>

* **Response**:

**200 OK**:

{

"message": "Logout successful"

}

**4. Manage Users (Admin only)**

* **Endpoint**: /api/v1/users
* **Method**: GET
* **Headers**:
  + Authorization: Bearer <Admin JWT token>
* **Response**:

**200 OK**:

{

"users": [

{

"id": "string",

"username": "string",

"email": "string",

"role": "string"

}

]

}

**403 Forbidden**:

{

"message": "Access denied"

}

**Implementation Details**

**Controller Layer**

Handles incoming HTTP requests and maps them to the appropriate service methods.

* user\_controller.go:
  + RegisterUser(w http.ResponseWriter, r \*http.Request)
  + LoginUser(w http.ResponseWriter, r \*http.Request)
  + LogoutUser(w http.ResponseWriter, r \*http.Request)
  + GetAllUsers(w http.ResponseWriter, r \*http.Request)

**Service Layer**

Contains the business logic and interacts with the repository layer.

* user\_service.go:
  + RegisterUser(user \*models.User) (\*models.User, error)
  + LoginUser(email, password string) (string, error)
  + LogoutUser(token string) error
  + GetAllUsers() ([]models.User, error)

**Repository Layer**

Handles database operations using an ORM or direct SQL queries.

* user\_repository.go:
  + CreateUser(user \*models.User) (\*models.User, error)
  + FindUserByEmail(email string) (\*models.User, error)
  + FindUserByID(id string) (\*models.User, error)
  + GetAllUsers() ([]models.User, error)

**Models**

Defines the data structures

user.go:

type User struct {

ID string `json:"id"`

Username string `json:"username"`

Email string `json:"email"`

Password string `json:"-"`

Role string `json:"role"`

}

**Routes**

Defines the API endpoints and maps them to the controller functions.

* routes.go:

func RegisterRoutes(router \*mux.Router) {

router.HandleFunc("/api/v1/register", userController.RegisterUser).Methods("POST")

router.HandleFunc("/api/v1/login", userController.LoginUser).Methods("POST")

router.HandleFunc("/api/v1/logout", userController.LogoutUser).Methods("POST")

router.HandleFunc("/api/v1/users", userController.GetAllUsers).Methods("GET")

}

**Utils**

Contains utility functions such as JWT token generation and validation.

jwt.go:

func GenerateToken(user \*models.User) (string, error) {

// Implementation for generating JWT token

}

func ValidateToken(token string) (\*models.User, error) {

// Implementation for validating JWT token

}

**Configuration**

Handles application configuration such as database connection details, secret keys, etc.

config.go:

func LoadConfig() (\*Config, error) {

// Implementation for loading configuration

}