Documentation for the leaderboard system project

Project Overview

This project is a leaderboard system built using Golang, PostgreSQL, and Redis. The application provides user registration, authentication using JWT, score submission, and fetching the leaderboard.

Project Structure

go

```
leaderboard-system/
├── cmd/
│   └── main.go
├── internal/
│   ├── auth/
│   │   └── auth.go
│   ├── database/
│   │   └── postgres.go
│   ├── leaderboard/
│   │   └── redis.go
│   ├── handlers/
│   │   └── handlers.go
│   ├── middleware/
│   └── utils/
├── pkg/
│   └── models/
│       └── user.go
├── go.mod
└── go.sum
```

Flow and Working of the Project

Main Function (`cmd/main.go`)

- Initializes PostgreSQL and Redis connections.
- Sets up HTTP handlers for various endpoints.
- Starts the HTTP server.

Functions Used:

- `database.InitPostgres()`
- `leaderboard.InitRedis()`

- `http.HandleFunc("/register", handlers.RegisterHandler)`
- `http.HandleFunc("/login", handlers.LoginHandler)`
- `http.HandleFunc("/submit-score", handlers.SubmitScoreHandler)`
- `http.HandleFunc("/leaderboard", handlers.FetchLeaderboardHandler)`
- `http.ListenAndServe(":8080", nil)`

1.PostgreSQL Initialization (`internal/database/postgres.go`)

- Establishes a connection to the PostgreSQL database.
- Sets the global `DB` variable to the database connection.

Functions Used:

- `sql.Open("postgres", connStr)`
- `DB.Ping()`

2.Redis Initialization (`internal/leaderboard/redis.go`)

- Establishes a connection to the Redis server.
- Sets the global `RDB` variable to the Redis client.

Functions Used:

- `redis.NewClient(&redis.Options{})`
- `RDB.Ping(ctx)`

3.User Registration Handler (`internal/handlers/handlers.go`)

- Decodes the incoming request to extract user details.
- Inserts the new user into the PostgreSQL database.
- Sends a success response.

Functions Used:

- `json.NewDecoder(r.Body).Decode(&user)`
- `database.DB.Exec(query, user.Username, user.Email, user.Password)`
- `json.NewEncoder(w).Encode(map[string]string{"message": "User registered successfully"})`

4.User Login Handler (`internal/handlers/handlers.go`)

- Decodes the incoming request to extract login details.
- Authenticates the user by verifying email and password from the PostgreSQL database.

- Generates a JWT token for the authenticated user.
- Sends the JWT token in the response.

Functions Used:

- `json.NewDecoder(r.Body).Decode(&user)`
- `database.DB.QueryRow(query, user.Email, user.Password).Scan(&username)`
- `auth.GenerateJWT(username)`
- `json.NewEncoder(w).Encode(map[string]string{"token": token})`

5.JWT Generation (`internal/auth/auth.go`)

- Creates a JWT token with the username and expiration time as claims.
- Signs the token using a secret key.

Functions Used:

- `jwt.NewWithClaims(jwt.SigningMethodHS256, claims)`
- `token.SignedString(jwtKey)`

6.JWT Validation (`internal/auth/auth.go`)

- Parses the JWT token to extract the claims.
- Validates the token signature and expiration.

Functions Used:

- `jwt.ParseWithClaims(tokenString, claims, func(token *jwt.Token) (interface{}, error) { return jwtKey, nil })`

7.Submit Score Handler (`internal/handlers/handlers.go`)

- Validates the JWT token from the Authorization header.
- Decodes the incoming request to extract the score.
- Fetches the user ID from the PostgreSQL database using the username from the JWT claims.
- Adds the score to the Redis sorted set for the leaderboard.
- Inserts the score into the PostgreSQL database.
- Sends a success response.

Functions Used:

- `auth.ValidateJWT(tokenString)`
- `json.NewDecoder(r.Body).Decode(&scoreReq)`
- `database.DB.QueryRow(query, claims.Username).Scan(&userID)`

- `leaderboard.RDB.ZAdd(ctx, "leaderboard", &redis.Z{Score: float64(scoreReq.Score), Member: strconv.Itoa(userID)})`
- `database.DB.Exec(query, userID, scoreReq.Score)`
- `json.NewEncoder(w).Encode(map[string]string{"message": "Score submitted successfully"})`

8.Fetch Leaderboard Handler (`internal/handlers/handlers.go`)

- Retrieves the top N scores from the Redis sorted set.
- Constructs a response with user IDs, scores, and ranks.
- Sends the leaderboard data in the response.

Functions Used:

- `leaderboard.RDB.ZRevRangeWithScores(ctx, "leaderboard", 0, 9).Result()`
- `strconv.Atoi(entry.Member.(string))`
- `json.NewEncoder(w).Encode(leaderboardEntries)`

Detailed Explanation of Each Function

1. **`database.InitPostgres()`**
   - **Purpose:** Initializes the PostgreSQL connection.
   - **How it works:** Uses `sql.Open` to open a connection and `DB.Ping` to verify it.
   - **Connection String:** Uses parameters like user, password, dbname, and sslmode.
   - **Global Variable:** Sets `DB` to the opened connection.
2. **`leaderboard.InitRedis()`**
   - **Purpose:** Initializes the Redis connection.
   - **How it works:** Uses `redis.NewClient` to create a new client and `RDB.Ping` to verify the connection.
   - **Redis Options:** Includes address, password, and database index.
   - **Global Variable:** Sets `RDB` to the created Redis client.
3. **`handlers.RegisterHandler(w http.ResponseWriter, r *http.Request)`**
   - **Purpose:** Handles user registration requests.
   - **How it works:** Decodes the request body to extract user details, executes an SQL query to insert the user into the PostgreSQL database, and sends a JSON response.
   - **Response:** Sends a success message upon successful registration.
4. **`handlers.LoginHandler(w http.ResponseWriter, r *http.Request)`**
   - **Purpose:** Handles user login requests.

    - **How it works:** Decodes the request body to extract login details, authenticates the user by querying the PostgreSQL database, generates a JWT token, and sends the token in the response.
    - **Response:** Sends the JWT token upon successful authentication.
5. `auth.GenerateJWT(username string) (string, error)`
    - **Purpose:** Generates a JWT token for a given username.
    - **How it works:** Creates a token with username and expiration claims, signs it using a secret key, and returns the token string.
    - **Claims:** Includes username and expiration time.
6. `auth.ValidateJWT(tokenString string) (*Claims, error)`
    - **Purpose:** Validates a JWT token and extracts the claims.
    - **How it works:** Parses the token, verifies the signature and expiration, and returns the claims.
    - **Claims:** Includes username and expiration time.
7. `handlers.SubmitScoreHandler(w http.ResponseWriter, r *http.Request)`
    - **Purpose:** Handles score submission requests.
    - **How it works:** Validates the JWT token, decodes the request body to extract the score, fetches the user ID from the PostgreSQL database, adds the score to the Redis sorted set, inserts the score into the PostgreSQL database, and sends a success response.
    - **Response:** Sends a success message upon successful score submission.
8. `handlers.FetchLeaderboardHandler(w http.ResponseWriter, r *http.Request)`
    - **Purpose:** Handles requests to fetch the leaderboard.
    - **How it works:** Retrieves the top scores from the Redis sorted set, constructs a response with user IDs, scores, and ranks, and sends the leaderboard data in the response.
    - **Response:** Sends the leaderboard data.

Testing the Leaderboard System

Using `curl`

1. User Registration

Command:

```
curl -X POST -d
'{"username":"testuser","email":"testuser@example.com","p
assword":"password"}' \
```

```
-H "Content-Type: application/json"
http://localhost:8080/register
```

Expected Response:

json

```
{
    "message": "User registered successfully"
}
```

2. User Login

Command:

```
curl -X POST -d
'{"email":"testuser@example.com","password":"password"}'
\
-H "Content-Type: application/json"
http://localhost:8080/login
```

Expected Response:

json

```
{
    "token": "your_jwt_token_here"
}
```

Replace `your_jwt_token_here` with the actual token received in the response.

3. Submit Score

Command:

```
curl -X POST -d '{"score": 100}' \

-H "Content-Type: application/json" \

-H "Authorization: Bearer your_jwt_token_here" \

http://localhost:8080/submit-score
```

Expected Response:

json

```
{

    "message": "Score submitted successfully"

}
```

Replace `your_jwt_token_here` with the actual token received from the login response.

4. Fetch Leaderboard

Command:

```
curl -X GET http://localhost:8080/leaderboard
```

Expected Response:

json

```
[
    {
        "user_id": 1,
        "score": 100,
        "rank": 1
    }
]
```

Using Postman

1. **User Registration**
    - **Open Postman.**
    - **Create a new `POST` request.**
    - **URL: `http://localhost:8080/register`**
    - **Go to the `Body` tab, select `raw` and `JSON` format.**

Enter the following JSON:
json

```
{
    "username": "testuser",
    "email": "testuser@example.com",
    "password": "password"
}
```

-
- Click `Send`.

Expected Response:
json

```
{
    "message": "User registered successfully"
}
```

-

2. **User Login**
    - Create a new `POST` request.
    - URL: `http://localhost:8080/login`
    - Go to the `Body` tab, select `raw` and `JSON` format.

Enter the following JSON:
json

```
{
    "email": "testuser@example.com",
    "password": "password"
}
```

-
- Click `Send`.

Expected Response:
json

```
{
    "token": "your_jwt_token_here"
}
```

-
- Copy the `token` from the response.

3. Submit Score
   - Create a new `POST` request.
   - URL: `http://localhost:8080/submit-score`
   - Go to the `Headers` tab and add:
     - Key: `Content-Type`, Value: `application/json`
     - Key: `Authorization`, Value: `Bearer your_jwt_token_here` (replace with the actual token)
   - Go to the `Body` tab, select `raw` and `JSON` format.

Enter the following JSON:
json

```
{
    "score": 100
}
```

-
- Click `Send`.

Expected Response:
json

```
{
    "message": "Score submitted successfully"
}
```

-

4. Fetch Leaderboard
   - Create a new `GET` request.
   - URL: `http://localhost:8080/leaderboard`
   - Click `Send`.

Expected Response:
json

```
[
    {
        "user_id": 1,
        "score": 100,
        "rank": 1
    }
    ]
```

Conclusion

This leaderboard system provides a robust and efficient way to manage user scores and display a leaderboard. By using PostgreSQL for persistent data storage and Redis for fast, in-memory data retrieval, the system ensures both reliability and performance. The use of JWT for authentication ensures secure access to the API endpoints.