That's a fantastic next step! Moving from basic CRUD operations to user authentication is essential for any real-world application.

To simulate user authentication with a token, you'll typically use the **JSON Web Token (JWT)** standard. The flow involves adding two new handlers: one for **logging in** (generating a token) and one for **authenticating** subsequent requests (checking the token).

## 🔐 Authentication Flow with JWT

Here is the plan to simulate token-based authentication:

1. **Login Handler (/login):** Accepts a username/password. If valid (simulated), it generates a signed JWT.
2. **Middleware/Auth Function:** A reusable function that verifies the JWT token sent in the Authorization header on protected routes.
3. **Protected Handler (/profile):** Uses the auth function to check the token before executing its logic.

We'll use the standard Go package github.com/golang-jwt/jwt/v5 for this, but for simplicity, we'll keep the actual logic within your current handler.go structure.

## 1. ⚙️ Setup: Secret Key and JWT Package

First, define a secret key to sign your tokens and ensure you have the necessary package.

### A. Secret Key

Add this constant near the top of your handler.go file:

Go

// handler.go  
const jwtSecret = "your\_super\_secret\_and\_long\_key\_here"

### B. Install JWT Package

In your terminal, run:

Bash

go get github.com/golang-jwt/jwt/v5

## 2. 🔑 The Login Handler (LoginHandler)

This handler simulates checking credentials and, if successful, generates and returns a JWT.

Go

// handler.go  
  
// A simple map to simulate a user database for this example  
var users = map[string]string{  
 "testuser": "password123",  
}  
  
func LoginHandler(w http.ResponseWriter, r \*http.Request) {  
 if r.Method != "POST" {  
 http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)  
 return  
 }  
  
 // 1. Decode credentials from request body  
 var creds struct {  
 Username string `json:"username"`  
 Password string `json:"password"`  
 }  
 if err := json.NewDecoder(r.Body).Decode(&creds); err != nil {  
 http.Error(w, "Invalid request body", http.StatusBadRequest)  
 return  
 }  
  
 // 2. Simulate User Authentication  
 expectedPassword, ok := users[creds.Username]  
 if !ok || expectedPassword != creds.Password {  
 http.Error(w, "Invalid credentials", http.StatusUnauthorized)  
 return  
 }  
  
 // 3. Create the JWT Token  
 // Set token claims (data inside the token)  
 expirationTime := time.Now().Add(5 \* time.Minute)  
 claims := jwt.MapClaims{  
 "user": creds.Username,  
 "exp": expirationTime.Unix(),  
 }  
   
 // Create the token instance  
 token := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)  
   
 // Sign the token using the secret key  
 tokenString, err := token.SignedString([]byte(jwtSecret))  
 if err != nil {  
 log.Printf("Error signing token: %v", err)  
 http.Error(w, "Could not generate token", http.StatusInternalServerError)  
 return  
 }  
  
 // 4. Send the token back to the client  
 w.Header().Set("Content-Type", "application/json")  
 json.NewEncoder(w).Encode(map[string]string{"token": tokenString})  
}

## 3. 🛡️ The Authentication Middleware/Wrapper

This reusable function will wrap any handler you want to protect.

Go

// handler.go  
  
// AuthMiddleware wraps an http.HandlerFunc to enforce JWT authentication.  
func AuthMiddleware(next http.HandlerFunc) http.HandlerFunc {  
 return func(w http.ResponseWriter, r \*http.Request) {  
 // 1. Get the token from the Authorization header  
 authHeader := r.Header.Get("Authorization")  
 if authHeader == "" {  
 http.Error(w, "Missing token", http.StatusUnauthorized)  
 return  
 }  
   
 // Header format is typically "Bearer TOKEN\_STRING"  
 tokenString := ""  
 if len(authHeader) > 7 && authHeader[:7] == "Bearer " {  
 tokenString = authHeader[7:]  
 } else {  
 http.Error(w, "Invalid token format", http.StatusUnauthorized)  
 return  
 }  
  
 // 2. Parse and Validate the Token  
 token, err := jwt.Parse(tokenString, func(token \*jwt.Token) (interface{}, error) {  
 // Check the signing method  
 if \_, ok := token.Method.(\*jwt.SigningMethodHMAC); !ok {  
 return nil, fmt.Errorf("unexpected signing method: %v", token.Header["alg"])  
 }  
 return []byte(jwtSecret), nil  
 })  
   
 if err != nil || !token.Valid {  
 log.Printf("Token validation failed: %v", err)  
 http.Error(w, "Invalid or expired token", http.StatusUnauthorized)  
 return  
 }  
  
 // 3. If token is valid, execute the original handler (the 'next' function)  
 next(w, r)  
 }  
}  
  
// A simple handler that requires authentication  
func ProfileHandler(w http.ResponseWriter, r \*http.Request) {  
 w.Header().Set("Content-Type", "application/json")  
 io.WriteString(w, `{"message": "Welcome to your protected profile!"}`)  
}

## 4. 🔗 Update Server() Function

Finally, update Server() to add the login route and wrap the protected route (/profile) with the middleware.

Go

// handler.go - inside Server() function  
  
func Server() {  
 mux := http.NewServeMux()  
 // ... existing handlers ...  
 mux.HandleFunc("/movie", FindMovieHandler)  
 mux.HandleFunc("/insert", InsertMovieHandler)   
  
 // 🎯 NEW: Public route for obtaining the token  
 mux.HandleFunc("/login", LoginHandler)   
  
 // 🎯 NEW: Protected route using the AuthMiddleware  
 mux.HandleFunc("/profile", AuthMiddleware(ProfileHandler))   
   
 // ... rest of the server setup ...  
}

## 🧪 How to Test the Flow

1. **Login (Get Token):**  
   Bash  
   # CMD/PowerShell: Escape quotes for JSON body  
   curl -X POST -H "Content-Type: application/json" -d "{\"username\": \"testuser\", \"password\": \"password123\"}" http://localhost:3333/login  
     
   *Output will be a JSON object containing your JWT.*
2. Access Protected Route (Use Token):  
   (Replace [YOUR\_TOKEN] with the string you received from the login step.)  
   Bash  
   curl -X GET -H "Authorization: Bearer [YOUR\_TOKEN]" http://localhost:3333/profile

This structure is the standard blueprint for implementing token-based authentication in Go web services.