Sami Kärkkäinen (2214482 POKT22SP)

RPG ACCOUNT MANAGEMENT

Course Final
Backend Programming

2025

Kaakkois-Suomen
ammattikorkeakoulu

TABLE OF CONTENTS

1 INTRODUCTION 3

2 TECHNOLOGY STACK 3

3 FEATURES 4

3.1 Session management 4

3.2 Database structure explained 5

4 REST API ENDPOINTS 5

4.1 db.php 6

4.2 Account creation / registration 6

4.3 Login using username and password 6

4.4 Logout 7

4.5 Fetching all characters of an account 7

4.6 Delete character 8

4.7 Change password 9

5 WORK ON THE PROJECT 9

1 INTRODUCTION

As the final project for the Backend Programming course, a prototype account management system was created for a role-playing game (RPG). The premise was that these types of games always require the player to have their own user accounts for persistence (character progress, unlocks, achievements, etc.), security and social features. It is also common for RPGs to allow the player to create multiple characters under the same account.

2 TECHNOLOGY STACK

The backend was built using XAMPP version 3.3.0 on a Windows machine. The web server is just the built in, preconfigured Apache HTTP server. The database is a MySQL database with only two tables: accounts and characters. The characters are related to their owner accounts by the account_id foreign key in the characters table. PHP was used to write the REST API to handle requests between the client software and the database. This makes the backend a WAMP stack (Windows, Apache, MySQL, PHP).

The frontend was built in Unity and consists of merely a user interface with various buttons and some input fields. Their logic is written in C# and the API requests using Unity's own UnityWebRequest class. UnityWebRequest simplifies the process of sending requests and receiving responses from the server. All requests are executed asynchronously using coroutines, allowing the app to remain responsive while waiting for data from the backend.

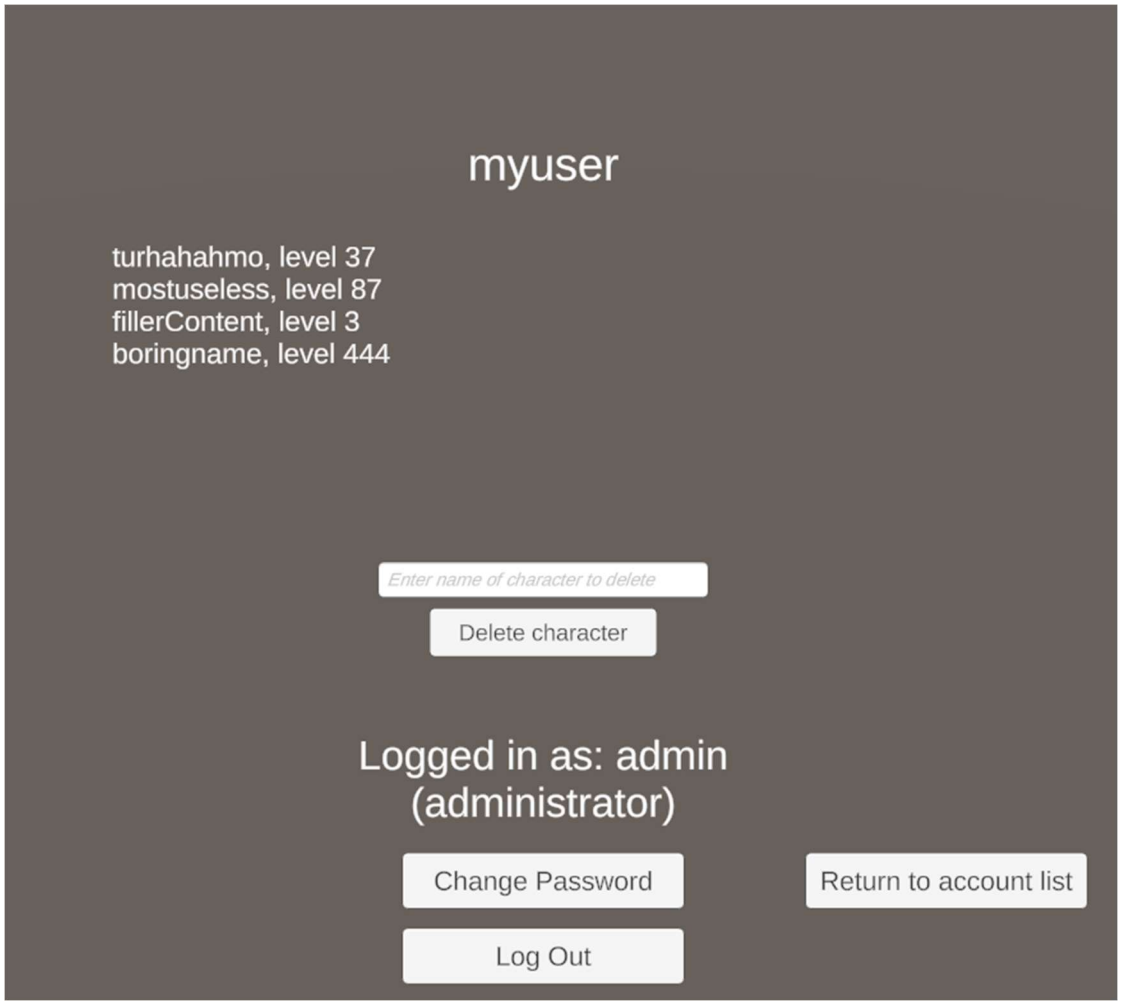

Figure 1 Client application view. The Unity-made frontend is very simplistic, but has quite a few features. Here an administrator is logged in and is viewing the characters of "myuser".

3 FEATURES

The features chosen to be implemented are:

- Account creation / registration
- Login using username and password
- Logout
- View list of characters associated with an account
- Delete chosen character
- Admin privileges to view and delete characters of any account
- Change password

3.1 Session management

For session management, PHP sessions are used. They are used to maintain state across HTTP requests. When a session is created, login data (such as username) is stored in session data and a unique session id is sent to the client as a cookie, which is then saved locally on the client's device. This id is

then sent back to the server with future requests, so the server can verify login status. In production, HTTPS should be used instead to prevent session id hijacking.

3.2 Database structure explained

The database is a very simple MySQL (relational) database. It consists of only two tables: accounts and characters.

Each user must create an account, and each account has the following:

- id
  - a unique auto-incremented integer value
- username
  - varchar string used as both login and display name. Must be unique
- password_hash
  - user passwords are hashed when created and stored in the database. These password hashes are basically impossible to reverse-engineer, providing more than sufficient safety.
- account_type
  - integer value that is used to define user privileges. 0 for regular users and 1 for admin rights.

All characters are stored in the characters table and have the following:

- id
  - a unique auto-incremented integer value
- account_id
  - an integer value that is used to associate the character with its owner account. An account can be associated with multiple characters.
- name
  - character name with a maximum length of 16 characters
- level
  - integer value to denote the character's level. Default 1

4 REST API ENDPOINTS

This chapter describes each of the various REST API endpoints and their functionalities in short. For each, a small code snippet is added containing the most important lines of code.

4.1 db.php

The db.php file is simply a helper script that is included in all other PHP-scripts to handle connecting to the database using PHP Data Objects (PDO).

```php
<?php
// handles connecting to MySQL database using PHP Data Objects (PDO)
$pdo = new PDO("mysql:host=localhost;dbname=testdb", "root", "");
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
?>
```

Figure 2 db.php helper script

4.2 Account creation / registration

The logic for the API endpoint for registration is the found in the register.php script file. It expects a POST request with JSON formatted input data as the body. The input data should contain a username and a password. If the input data is applicable, the accounts table is then checked for an identical username. If no conflicts are found, the script proceeds in account creation. The given password is hashed using PHP's password_hash function. A prepared statement is then created and executed, which queries the database using SQL creates a new row in the accounts table, with the given username and the hashed password.

```php
// Hash the password before storing it
$hash = password_hash($password, PASSWORD_DEFAULT);

// Insert new user into database
$stmt = $pdo->prepare("INSERT INTO accounts (username, password_hash) VALUES (?, ?)");
$stmt->execute([$username, $hash]);
```

Figure 3 These lines create the new account into the database.

4.3 Login using username and password

The login.php script contains the logic for login (sign in) using PHP sessions. Each script that uses PHP sessions must have the line session_start(); at the top, as it starts a new session or resumes an existing one. As with the registration API, the login endpoint also expects a POST request with the username and password as its body in JSON format.

A matching username is being searched for using a prepared statement. If found, the password given as input is compared to the hashed password in the database using the password_verify function. If the password is correct, the session variables user_id, username and accountType are being updated. These variables can be used by the frontend as well through a custom C# class SessionManager and its public methods.

```php
if ($user && password_verify($password, $user['password_hash'])) {
    $_SESSION['user_id'] = $user['id'];
    $_SESSION['username'] = $user['username'];
    $_SESSION['accountType'] = $user['accountType'];
    echo json_encode(["message" => "Login successful"]);
```

Figure 4 The most important part of login.php, where the password is verified and session variables are set.

4.4 Logout

The logout endpoint does not need to any additional data in a request body, so a GET request is used to call it. It simply clears the current session variables and terminates the session, requiring the user to log in again the next time they wish to access their account.

```php
<?php
header("Cache-Control: no-cache, no-store, must-revalidate");

session_start();
session_unset(); // clears session variables like username, id...
session_destroy(); // destroys the complete session

echo json_encode(["success" => true, "message" => "Logged out successfully."]);
?>
```

Figure 5 Logout.php is a straightforward script.

4.5 Fetching all characters of an account

To get the list of all characters associated with a certain account, the ‘characters’ endpoint must be used. Its code is found together with the ‘accounts’ and ‘delete’ endpoints in the index.php file.

To use the ‘characters’ endpoint, a simple GET request should be used and the selected account id given as a path parameter. E.g.: …path/testapi/characters/3/ tries to find all characters associated with the user account with id = 3. An SQL query is performed on the database to select all characters where account id matches with the given parameter. The results are then added to an array, encoded in JSON format and echoed as a HTTP response. The frontend can then easily deserialize the data and use it to display the list of characters on the UI.

4.6 Delete character

The ‘delete’ endpoint allows the deletion of selected character from the database. Its code is also found in the index.php file. To use the endpoint, a DELETE HTTP request is used. The account id and character name associated with the character to be deleted are given as path parameters in the URL. The API code then checks whether the session account type is regular user or admin to determine whether the user has rights to delete characters from other accounts or only their own. While logged in as a regular user, the frontend automatically sends the logged-in user’s account id as parameter, whereas on admin accounts, it depends on the account currently selected. A prepared statement is created to form and execute the SQL query on the database. To create the SQL query string the bind_param function is used to prevent SQL injections, such as “DELETE FROM characters WHERE account_id = 3 OR 1=1;” which would always evaluate to true and delete all characters from the database.

```php
// Authorization check
$currentUser = $_SESSION['user_id'];
$accountType = $_SESSION['accountType'];
// if not logged in as admin and trying to delete a character that does not belong to you
if ($accountType != 1 && $currentUser != $accID) {
    http_response_code(403);
    echo json_encode(["error" => "Forbidden: You cannot delete characters from this account."]);
    exit;
}

$query = "DELETE FROM characters WHERE account_id = ? AND name = ?";
$stmt = $conn->prepare($query);
$stmt->bind_param("is", $accID, $charName);
$stmt->execute();
```

Figure 6 The 'delete' endpoint checks for both account type and ownership of the character in question to allow or disallow deletion.

4.7 Change password

The endpoint used to change the password or current account in found in its own file changepassword.php. Similarly to the ‘register’ and ‘login’ endpoints, this one also requires a POST request with the new password in JSON format as its body. The username is read from the session variable and does not need to be sent in the request body.

The API first checks that a password is given and that its length is sufficient. Then, the new password is hashed using password_hash and updated on the database using an SQL query created and executed using prepared statements.

```php
$username = $_SESSION['username'];
$new_password = $data['new_password'] ?? null;
$hashed_password = password_hash($new_password, PASSWORD_DEFAULT);

try {
    $stmt = $pdo->prepare("UPDATE accounts SET password_hash = ? WHERE username = ?");
    $stmt->execute([$hashed_password, $username]);

    echo json_encode(["success" => true, "message" => "Password changed successfully."]);
} catch (PDOException $e) {
    http_response_code(500);
    echo json_encode(["error" => "Database error", "details" => $e->getMessage()]);
}
```

Figure 7 The API notifies the client application of either a successful password change or an error, if something goes wrong.

5 WORK ON THE PROJECT

This project was done as a solo project. The PHP REST API scripts would have been quite difficult and/or time consuming to figure out completely on my own, so generative AI was used to help in the process. I did, however, spend a lot of time in learning how the code actually works, and I hope it shows in how I explained everything above.

A large portion of my hours in this project was spent on the Unity side, because developing even a basic user interface takes a surprising amount of time. Using UnityWebRequests turned out to be relatively straightforward, and I could mostly just use the same formula found in an example in Unity’s documentation.

```
IEnumerator Login(string username, string password)
{
    LoginData credentials = new LoginData { username = username, password = password };
    string jsonData = JsonUtility.ToJson(credentials);

    UnityWebRequest wr = new UnityWebRequest(baseUrl + "login.php", "POST");
    byte[] bodyRaw = System.Text.Encoding.UTF8.GetBytes(jsonData);
    wr.uploadHandler = new UploadHandlerRaw(bodyRaw);
    wr.downloadHandler = new DownloadHandlerBuffer();
    wr.SetRequestHeader("Content-Type", "application/json");

    yield return wr.SendWebRequest();

    if (wr.result != UnityWebRequest.Result.Success)...
    else...
}
```

Figure 8 The different client methods for creating HTTP requests are mostly similar. The ones like Login here are slightly more complicated due to requiring JSON formatted data to be sent as POST body.

Overall, I learned a lot during this project and course, especially about REST APIs, what they are, how to use them and how to implement them. PHP was also completely new for me and while it does not always have the easiest syntax to read, does seem like quite a powerful language for backend applications.