MTCG Protocol

**Use of Dependency Injection:**

The application utilizes dependency injection (DI) to manage service lifetimes and dependencies, enabling modularity and easier testing. The IHostBuilder is used to configure and inject dependencies for services such as UserService, CardService, and TcpServer. This ensures that the application components are decoupled, allowing for better maintainability and scalability.

**TCP Server with HTTP-like Handling:**

The server implements a custom TCP server that handles HTTP-like requests over raw sockets. This design decision avoids relying on a full HTTP server, giving you more control over request handling, stream management, and custom error handling. Each request is processed asynchronously using a NetworkStream, which allows for efficient handling of multiple connections.

**Prepared Statements for Security:**

To prevent SQL injection, prepared statements are used in all database queries. This ensures that user inputs are correctly parameterized, avoiding the risk of malicious data manipulation in SQL queries. For instance, when interacting with the PostgreSQL database via Npgsql, the queries are constructed with parameter placeholders, which are then filled with user input values. This approach improves security and minimizes the risk of SQL injection attacks.

**Routing Based on HTTP Method and Path:**

The routing system is built to differentiate between different HTTP methods (GET, POST, PUT) and paths (e.g., /sessions, /users, /cards). Each path corresponds to a specific action, such as logging in a user, creating a package, or listing cards. The method ensures that requests are correctly routed to the appropriate handler, simplifying the request processing logic.

**Handling User Authentication and Authorization:**

Authentication is handled via bearer tokens passed in the Authorization header. These tokens are validated before processing protected routes. If a request does not contain a valid token, an UnauthorizedException is thrown, ensuring that only authorized users can access certain resources (e.g., user profiles, deck updates, etc.).

**Transaction Handling:**

The server supports transaction-like behavior when acquiring packages or processing other sensitive data. By using prepared statements in conjunction with the connection pooling provided by Npgsql, the application ensures that database transactions are handled efficiently and safely.

This approach ensures that the application is robust, secure, and easily extensible for future features, such as adding new API endpoints or supporting additional game mechanics.

**Important Classes**

**CardService Class**

The CardService class is responsible for managing the collection of cards. It provides methods to add, retrieve, update, and delete cards from an in-memory list. The class also includes an asynchronous method (SaveCardsAsync) to save multiple cards at once, which simulates saving the cards to a persistent storage. This class operates on a collection of Card objects and ensures that each card has a unique ID before adding it to the list.

**Card Class**

The Card class represents a card in the game. Each card has properties such as id, name, damage, type, and element, with logic to automatically determine the card's type (e.g., Monster or Spell) and elemental type (e.g., Fire or Water) based on its name. The class also contains methods for interacting with the database, such as AddCardAsync, which inserts a new card into the database, and AddCardsToUserInventoryAsync, which adds cards to a user's inventory.

**TokenService Class**

The TokenService class is used for validating tokens. The ValidateToken method compares two tokens and returns a boolean value indicating whether they match. This service is likely used for authentication purposes in the application.

**AuthenticationResult Class**

The AuthenticationResult class encapsulates the result of an authentication attempt. It includes a boolean property IsAuthenticated to indicate if the authentication was successful, a User object that represents the authenticated user, and a Token property that holds the authentication token.

**Deck Class**

The Deck class represents a player's deck of cards. It contains properties such as Id, Name, and a list of Card objects that make up the deck. The GetDeckForUser method retrieves the deck associated with a specific user from the database, including the cards in that deck. It queries the user\_deck and cards tables in the PostgreSQL database and returns the deck with all the cards for the user.

**Package Class**

The Package class handles the creation of card packages, inserting them into the database, and associating them with specific card details. It uses dependency injection to manage card-related and user-related operations. The primary purpose of this class is to allow administrators to create packages containing multiple cards, which can then be used in the game or for other purposes.

**UserService Class**

The UserService class manages user operations, such as registration, authentication, retrieving user data, and handling user-specific operations like updating statistics or fetching decks. It interacts with the PostgreSQL database to store and retrieve user information, ensuring secure password management and facilitating user-related actions.

These classes work together to manage the card and user data, interact with the database, and handle user authentication and deck management in the application. The CardService and Card classes handle core functionality, while the TokenService, AuthenticationResult, and Deck classes focus on user authentication and deck management.

**Lessons Learned**

Throughout the development of this project, I have gained valuable insights and hands-on experience in several key areas:

1. **HTTP Routing**: I learned how to implement routing for HTTP-like requests in a TCP server. This involved defining custom routes for various HTTP methods (GET, POST, PUT) and creating efficient, structured handling of requests based on the HTTP path and method. I became familiar with handling headers, parsing query parameters, and managing request and response formatting in a custom-built server environment without relying on high-level frameworks. This approach helped me better understand the underlying mechanics of web servers and APIs.
2. **Working with PostgreSQL in C#**: In this project, I worked directly with PostgreSQL using Npgsql, bypassing higher-level ORM frameworks. This allowed me to get hands-on experience in constructing SQL queries, managing database connections, and executing transactions. I focused on writing optimized SQL queries for tasks such as inserting, selecting, and updating user data, as well as ensuring that prepared statements were used to prevent SQL injection vulnerabilities. This experience strengthened my understanding of SQL and how to interact with relational databases from C#.
3. **Deepening Know-How on Threading**: As the project involved managing multiple concurrent operations (such as handling incoming requests, database queries, and updates), I deepened my knowledge of threading in C#. I implemented solutions for managing database connections with a connection pool to ensure efficient, concurrent access to the database. I also worked with asynchronous programming to prevent blocking operations, which improved performance and scalability.

These lessons not only enhanced my technical skills but also gave me a deeper understanding of how to build scalable and secure applications with optimized performance.

**NUnit Test Decisions**

**UserService Tests:**

Verify that user authentication and retrieval functions are working correctly.

* GetUserByUsernameAsync\_ShouldReturnUser\_WhenUserExists: Checks whether the correct user is returned when a valid username is queried. It tests the database interaction and the mapping of user data.
* AuthenticateUserAsync\_ShouldAuthenticate\_WhenCorrectCredentials: Ensures that the authentication process works as expected when the correct username and password are provided.
* GetUserByUsernameAsync\_ReturnsNull\_WhenUsernameDoesNotExist: Tests the system's response when querying a non-existent user, ensuring that the method handles this gracefully by returning null.

TokenService Tests:

Test the functionality of token generation and validation.

* GenerateToken\_ReturnsToken\_WhenValidUser: Ensures that a token is generated for valid users and checks that it’s not null or empty.
* ValidateToken\_ReturnsTrue\_WhenTokensAreEqual: Verifies the token validation mechanism works when both tokens match.
* ValidateToken\_ReturnsFalse\_WhenTokensAreNotEqual: Ensures that the token validation correctly returns false for mismatched tokens.

Package Tests:

Ensure that package creation and card management functions work correctly.

* CreatePackage\_ShouldReturnUnauthorized\_WhenInvalidToken: Verifies that unauthorized users (those with invalid tokens) are prevented from creating packages, enforcing secure access.
* GetCardType\_ShouldReturnCorrectCardType\_WhenCardNameIsProvided: Ensures the system can correctly identify the card type based on the card’s name.
* GetElementalType\_ShouldReturnCorrectElementType\_WhenCardNameIsProvided: Verifies the correct identification of elemental types based on card names.

Deck Tests:

Ensure deck management functions correctly.

* GetDeckForUser\_ShouldReturnEmptyDeckForUserWithNoDeck: Verifies that a user with no cards in their deck gets an empty deck, ensuring proper handling of edge cases.

Card Tests:

Test card-related functionalities such as type and element assignment.

* SetCardTypeAndElement\_ShouldSetCardTypeAndElementCorrectly: Verifies that the correct card type and elemental type are assigned to cards based on their names.
* Card\_ShouldHaveCorrectName\_WhenCreated: Tests that cards are correctly instantiated with the expected name.
* CreatePackage\_ShouldCreatePackage\_WhenAuthorized: Verifies that authorized users can create packages successfully, testing both the authorization and the package creation logic.

General Purpose of the Tests

* Authentication and Authorization: Several tests check the login and authentication flow, ensuring that only valid credentials lead to access and that unauthorized users are blocked (e.g., in CreatePackage\_ShouldReturnUnauthorized\_WhenInvalidToken).
* Data Integrity: Tests like GetUserByUsernameAsync\_ShouldReturnUser\_WhenUserExists and GetDeckForUser\_ShouldReturnEmptyDeckForUserWithNoDeck ensure that data in the database is correctly retrieved, and edge cases like missing data are handled gracefully.
* Security: By including token validation tests (ValidateToken\_ReturnsTrue\_WhenTokensAreEqual), we ensure that security mechanisms such as token generation and validation are robust.
* Database Interaction: Most of the tests interact with the database to verify that the application correctly handles and manipulates user, card, and package data stored in PostgreSQL.
* Edge Cases and Error Handling: Many tests also focus on how the system behaves under non-ideal conditions, such as invalid tokens or non-existent users, helping to ensure graceful error handling and avoid crashes or unintended behaviors.

The overall goal of these tests is to confirm that the system functions correctly, is secure, and handles errors appropriately, thus ensuring the reliability and stability of the application.

**Time spent on the Project:**

|  |  |
| --- | --- |
| Date | Hours spent |
| Prior to silvester: | ~35h |
| 01.01.2025 | 5h |
| 02.01.2025 | 7h |
| 03.01.2025 | 6h |
| 04.01.2025 | 8h |
| 05.01.2025 | 12h |
| 06.01.2025 | 12h |
| **TOTAL** | **~85h** |

Link to Github Repository:

https://github.com/LeafOfBread/MTCG\_FINAL