# MTCG Protocol

## Development Process & Design

The project was initialized according to the best practices of building server backend applications with RESTful APIs. First, classes and game mechanics were defined and created according to the features described above. Afterwards, a RESTful API was built with various endpoints and respective request requirements to enable numerous features. The next step included creating a PostgreSQL database and defining respective schemas: User, Card and TradingDeal (specifications in file "db_conf.txt"). Upon establishing the database connection several layers according to the Controller-Service-Repository-Models were defined and implemented:

- Controllers are responsible for directly handling HTTP requests and sending responses back to clients
- Services handle application logic and make calls to repositories if necessary (services are also the only ones with access to the repositories, otherwise it violates the Dependency Inversion Principle)
- Repositories directly access the database and perform the operations
- Models are annotated with @JsonProperty and each field has a unique value, so that they can be used for Json serialization when parsing the requestBody to create respective objects, and Json deserialization when turning Java objects into Json and sending them over to the client in responseBody.

Eventually, multithreading was integrated to handle multiple clients at once, as well as unit tests were written for core repository operations.

## Lessons Learned

This project gave a solid insight into building robust, reliable backend servers for web applications. One of the most challenging aspects of it was piecing all things together and developing a whole system whose parts would work cohesively. While handling database queries and HTTP requests is relatively simple, defining the business logic for the card game and carefully monitoring the battle outcomes was much more sophisticated.

In retrospective, the things I would do differently are:

- start as early as possible
- define a clear structure for the whole project from the very beginning
- don't rush to implement the business logic: start off with client-server connection & communication

## Unit Testing Decisions

The implemented unit tests mostly check the correct functionality of the repository implementations - UserRepositoryImplTests, CardRepositoryImplTests, TradingRepositoryImplTests, BattleRepositoryImplTests. Using annotations @Mock for DatabaseUtil, Connection, PreparedStatement, ResultSet, and annotation @InjectMocks for the respective repository, I mimic the work of the real database, so that all queries and tests do not interfere with the work of the actual application.
Besides, several tests were implemented to ensure the correct functionality of the business logic under "gameElements" & "gameManager" packages.

## Unique Feature

The unique feature "Gamble" can be accessed by sending a POST request to the server with path "/gamble", attaching user's authToken under the "Authorization" header. After the request has been sent, a random card from the user's deck is chosen and the following logic is applied to it: there is a 50% chance that the card's damage will be doubled, and also a 50% chance that the card's damage will be halved. The updated damage persists in the next battle between users.

## Tracked Time

| Feature | Time |
|---|---|
| HTTP | 10 h |
| Battle Management | 20 h |
| Controller Logic | 15 h |
| Repository Logic | 30 h |
| Unit Tests | 7 h |
| Multithreading | 2 h |

**Total: ~84 hours**

[Link to GitHub]