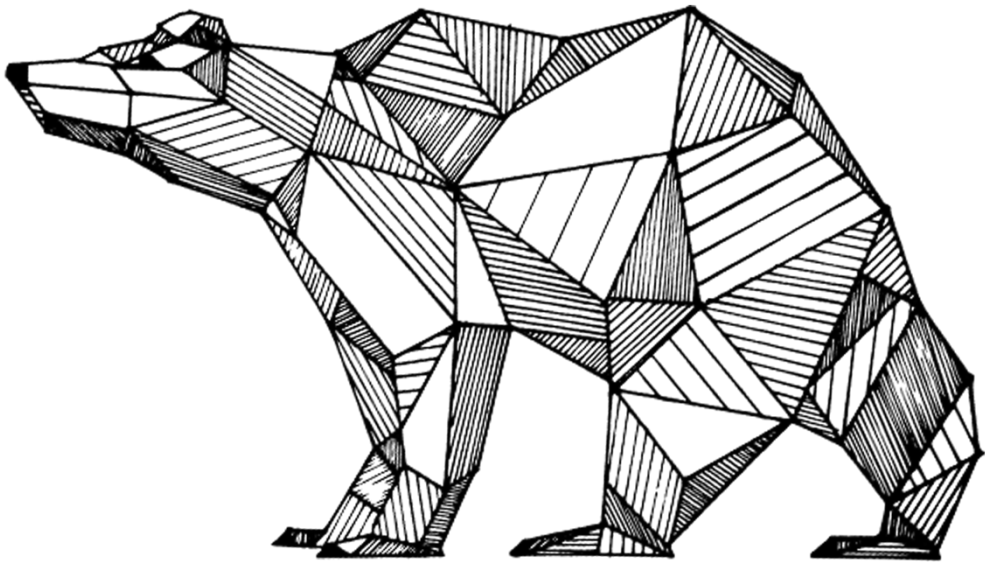
MTCG PROTOCOL



Raphael DOHNALEK
if20b075

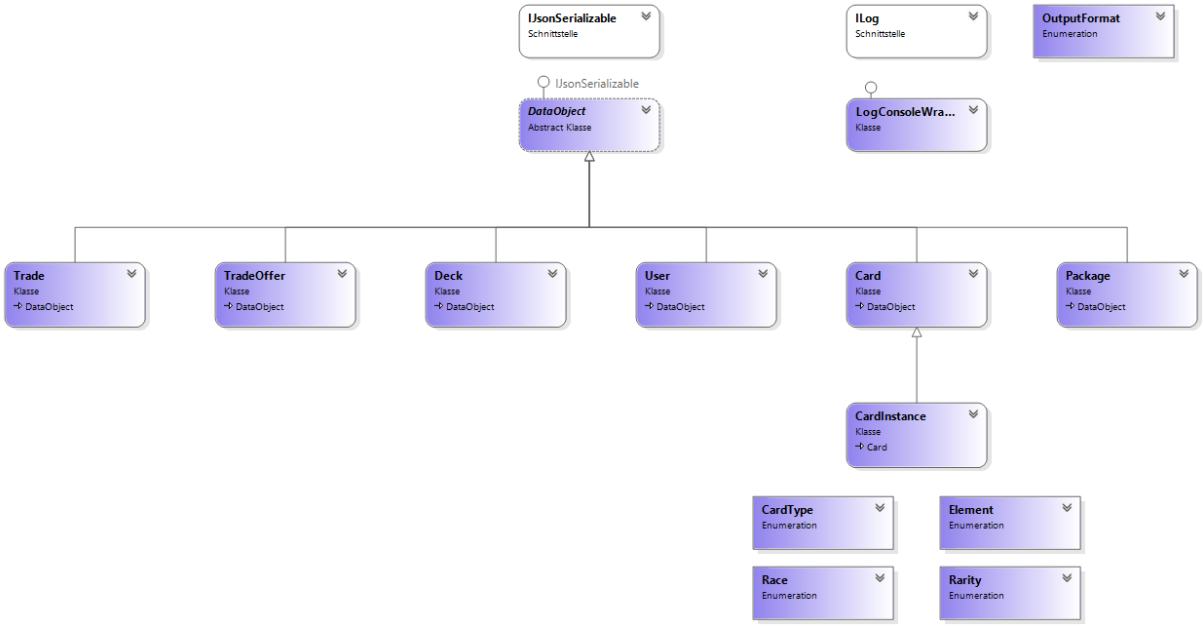
02.01.2022, Vienna

TABLE OF CONTENTS

1. ARCHITECTURE

1.1 MODELS

Class Diagram of MTCG.Models:



1.1.1 DATA OBJECTS

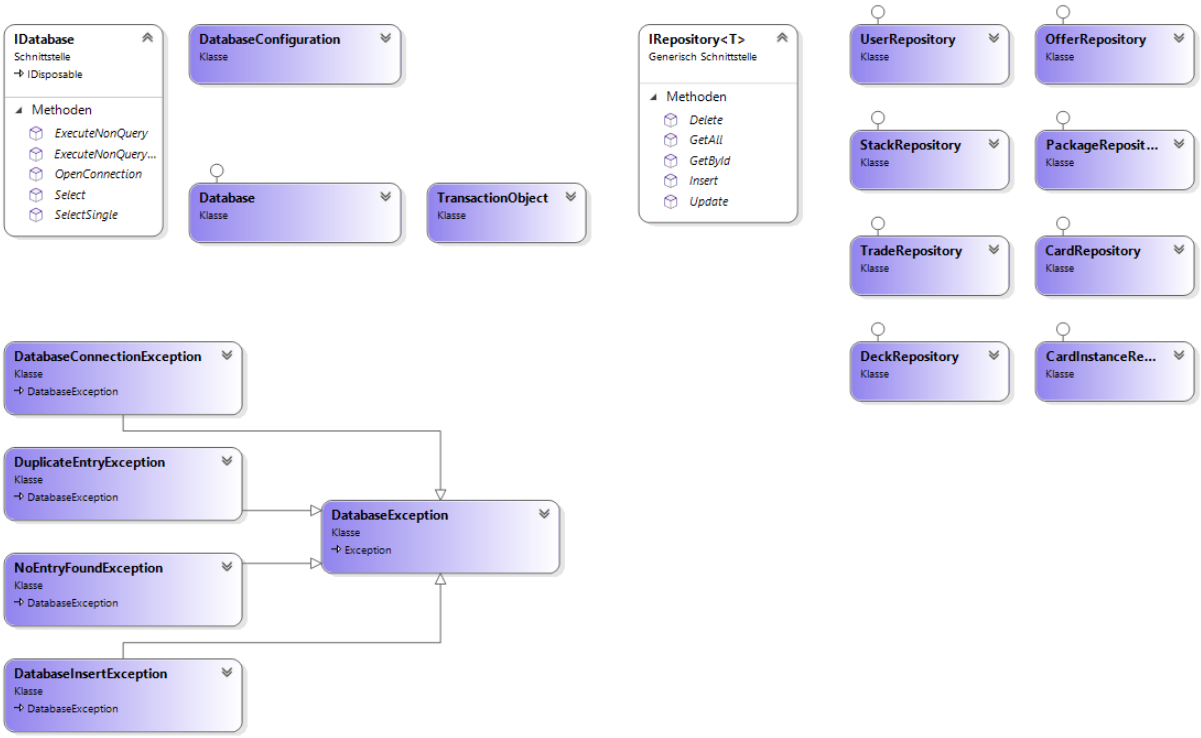
- IJsonSerializable
  - Provides the ToJson() method
- **DataObject**
  - Abstract base class of models (Implements generic ToJson() and provides the basic unique Guid of the model)
- User
- Card
- CardInstance
- Package
- Deck
- TradeOffer
- Trade

1.1.2 LOGGING

- ILog
  - Common log object for logging actions
- LogConsoleWrapper
  - Wraps the C# Console object into the ILog interface

1.2 DATA-ACCESS-LAYER

Class Diagram of MTCG.DAL:



1.2.1 DATABASE

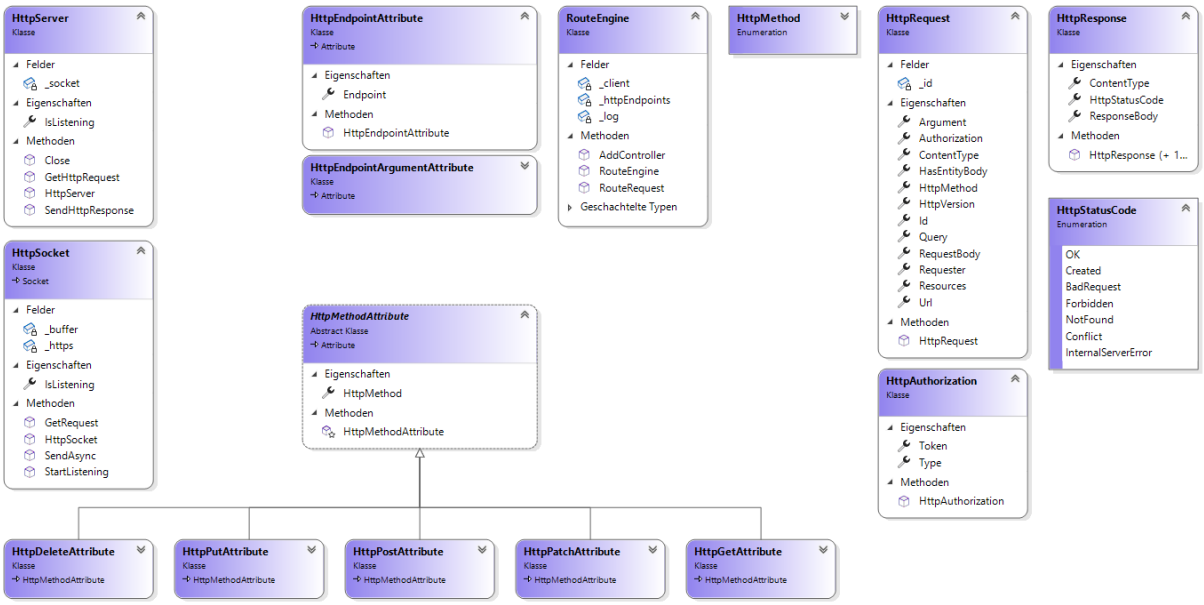
- IDatabase
  - Interface for simpler SQL command execution
- Database
  - Npgsql IDatabase implementation
- Database Configuration
  - Holds the databases connection string
- Transaction Objects
  - Holds all commands that are needed in a transaction

1.2.2 REPOSITORIES

- IRepository<T>
  - Generic repository with all crud operations
  - T should be a Model
- UserRepository
- StackRepository
- CardRepository
- CardInstanceRepository
- OfferRepository (for trade offers)
- TradeRepository
- DeckRepository
- PackageRepository

1.3 BUSINESS-LAYER

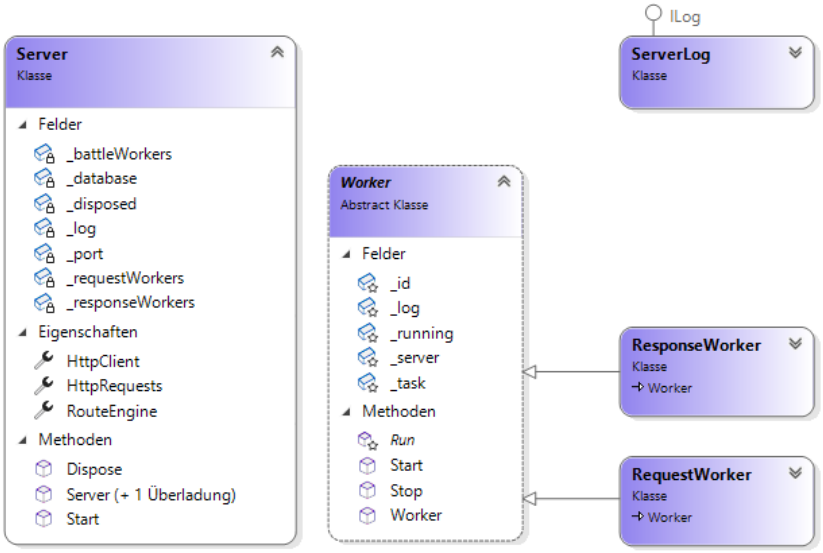
1.3.1 HTTP



- HttpSocket
  - Receives and sends http requests and responses
- HttpServer
  - Wraps the socket into a lightweight send and receive class
- HttpRequest
- HttpAuthorization
  - Responsible for HttpAuthorization via token or other auth-types
- HttpResponse
  - Answer to an http request (needs context reference to request it answers)

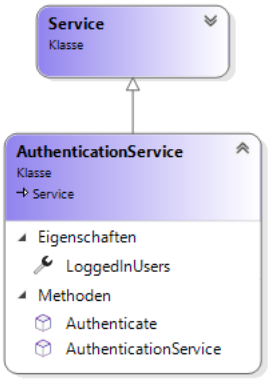
- RouteEngine
  - Gets all methods of controller and routes the http requests to the right endpoints
- HttpMethodAttribute
  - Describes a http endpoint method of a controller class

1.3.2 SERVER



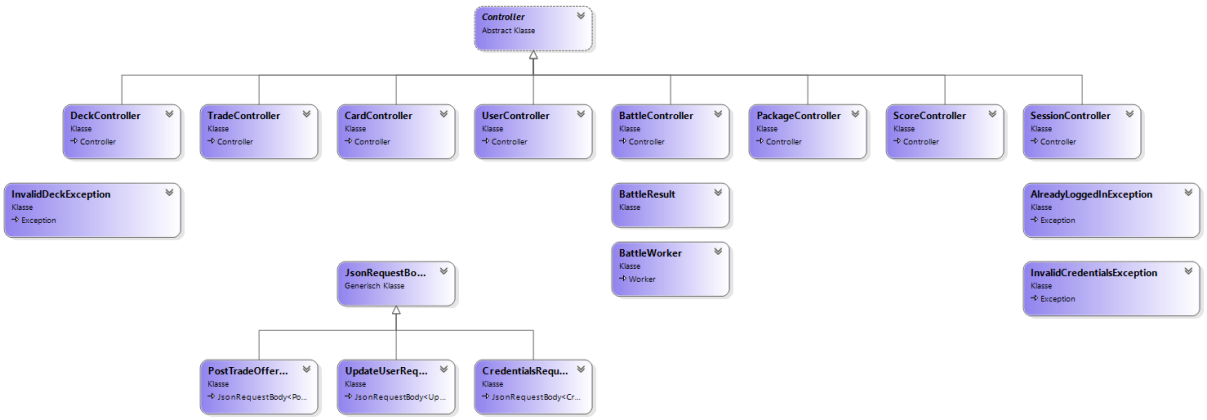
- Server
  - Main server that loads all controllers, instantiates the database connection and starts listening on the given TCP port
- ServerLog
  - Log implementation for server output
- Worker
  - Waits for a specific input to process this input and return to waiting state
- RequestWorker
  - Processes the http requests, which the server receives
- ResponseWorker
  - Sends the http responses from the routing engine

1.3.3 SERVICES



- Service
  - Base class for all services
- AuthenticationService
  - Service that handles the http authorization and session control on the server

1.3.4 CONTROLLER



- Controller
  - Base class for the routing engine to map a specific http endpoint controller to the given endpoint string
  - Can implement multiple http methods
- All other controllers
  - Business logic for their specific business task (e.g., the BattleController is responsible for a battle between users and dispatches the battle request to a battle worker, which will send the result back to the clients)

2. LESSONS LEARNED

2.1 REPOSITORY PATTERN

In this semester of software engineering, I have learned how to use and implement the repository pattern for my project. I made the mistake of starting the project at the start of the semester and back then, I did not know how to properly separate the data access layer and the business layer. Therefore, my business logic was entangled withing my data access logic (e.g., SQL statements). I reworked the complete business layer and separated the data access logic from the business logic. This made the code much more readable and maintainable. It is also easier now, to implement new repositories and controllers even though the project is in its finished state.

2.2 TEST-DRIVEN-DEVELOPMENT

Test driven development (or TDD in short) is a software development approach, where the test for a functionality is written beforehand. Then the developer writes the functionality for as long as the test fails. If the test passes, the function is finished, and the developer is able to move to the next task. This approach helped me a lot on the data access side of the project. I ran integration and unit tests of the repositories very early on to determine whether I made errors in my SQL statements or prepared function calls. After a repository was successfully tested, it was safe to concentrate on the business logic and debug it without the need to check the data access layer on each business logic test.

3. UNIT TESTS

3.1 REPOSITORIES

For my repositories I made several unit and integration tests. The unit tests mock the IDatabase interface and check how many calls were made and if the right calls were made, as well as the parsing functionality with a given data result set. The integration tests were done using a specific standalone PostgreSQL database where all tests could be done without worrying about the effect on any important data. (Never test in production)

3.2 HTTP SERVER

The http server is tested for its core functionality (receiving requests and sending responses with multiple http methods)

4. UNIQUE FEATURES

4.1 CARD RARITY

My first unique feature is a rarity attribute for cards. For itself it is only cosmetic and will be shown as an attribute on the card model. There are several rarities (Common, Uncommon, Rare, Epic, Legendary).

It acts as a base for future luck driven or balancing functionalities.

4.2 RANDOM PACK DRAWING

This feature uses the card rarity feature and maps draw percentages to the rarities. In my version of the MTCG, the packages hold a specific number of cards and whenever a user buys a package, X of the cards are drawn from the pack. The possibilities of which cards are drawn are determined by their rarity, where legendary is the rarest. In order to reward the player for his luck, legendary cards are typically stronger than the less rarer cards.

5. GIT

GitHub repository link (Currently private due to sensible data):

https://github.com/rasebdon/swen_project_2021/