MCTG Protocol

The project implements a REST Server application with HTTP Protocol Request- and Response-Parsing using Routers and API Endpoints. Model classes for users and cards were designed to then store actual users and cards in a postgres database to be fetched and used in the application. The application simulates a battle between two users over multiple round using decks of cards. The cards are being created in packs by the admin and put into packages, which the users, after registering and logging in can buy with their coins. The users can then create their own decks based on which cards they want to be "battle-ready". A deck contains 4 cards. Users can also edit bits of their data and their profile-page, which, in this project, is stored in the database aswell.

The actual battle process consists of each card being picked at random from the 2 opponent's decks. Whichever card has the higher damage, wins the individual round, and the winning player takes the card into their deck. Whichever player runs out of cards first, loses. Cards have interactions with each other, mostly just depending on whether or not they are a "Spell"-type card or a "Monster"-type card or which element they have.

The entire battle can be observed by reading the battlelog once the battle is concluded. A maximum of 100 rounds can be played. If a winner has not been decided by then, the battle ends in a draw.

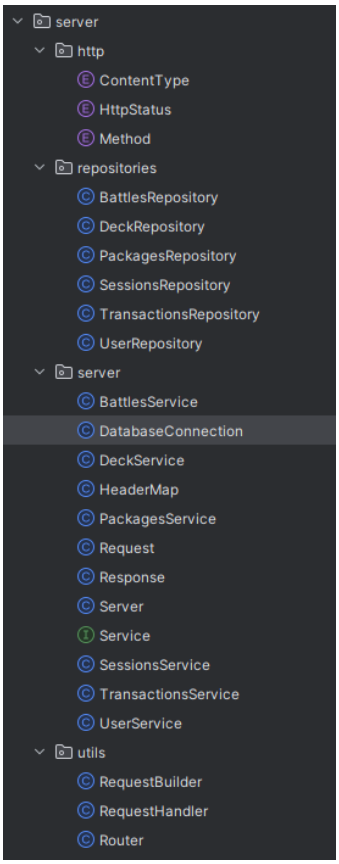
The battle makes use of a unique feature called "**Chaos Burst**"

```
Random random = new Random(); // to determine random cards and chaos burst chance

// UNIQUE FEATURE (CHAOS BURST)
double chaosChance = 0.10 + (random.nextDouble() * 0.20); // random CHAOS BURST chance for this battle between 10% and 30%
battlelog.append("CHAOS BURST chance for this battle: ").append(String.format("%.2f", chaosChance * 100)).append("% ( 10% - 30% )\n\n"); // show this battle's chaos burst chance
// ---------------------------
battlelog.append("Battle begins between ").append(player1).append(" and ").append(player2).append("\n\n");
```

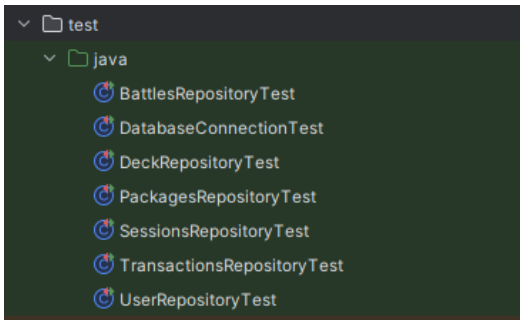
Every battle has a different chance of "chaos". Every round, a player's card has a chance to initiate "Chaos Burst", which adds (also at random) between 10-30 additional points to their damage in that particular round, which could make rounds a bit more chaotic and turn rounds which would've been lost for some cards into unexpected wins.

```
// UNIQUE FEATURE (CHAOS BURST)
if (random.nextDouble() < chaosChance) {
    int chaosBurstPlayer1 = random.nextInt( bound: 21) + 10; // random damage boost between 10-30
    currentRoundDamagePlayer1 += chaosBurstPlayer1;

    battlelog.append("CHAOS BURST! ").append(player1).append("'s ").append(cardPlayer1.getCardName()).append(" gains an additional ").append(chaosBurstPlayer1).append(" damage this round!\n\n");
}
```

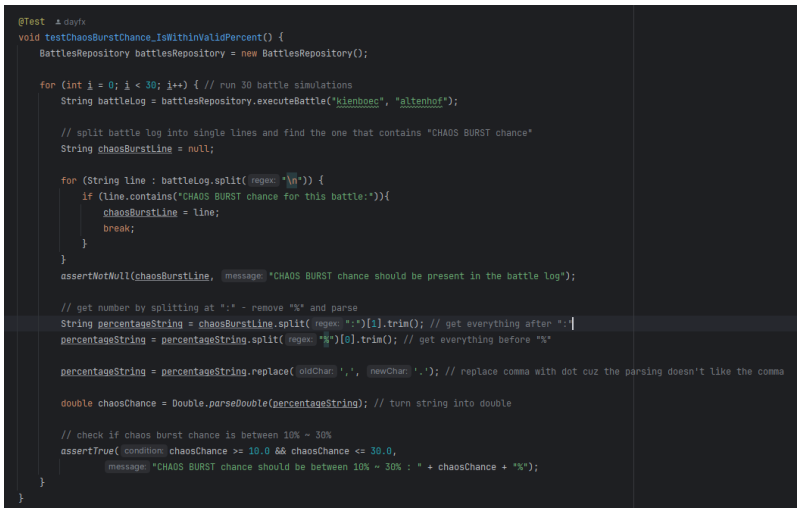
As for the general architecture of the project, it mainly makes use of individual Services Classes for different functionalities depending on which API Endpoints need to be handled and which type of requests and methods. Those individual Services Classes are then connected to the corresponding Repository, where most of the functionality between the application and the database interactions happens. Depending on that, the Service Classes then create different responses which are then sent back to the client to be displayed.



The application also makes use of unit tests to make sure that everything works as intended.



The unit tests mainly cover the functionality of the repositories, since this is where most of the important functionality lies.



```
@Test  ± dayfx
void testChaosBurstChance_IsWithinValidPercent() {
    BattlesRepository battlesRepository = new BattlesRepository();

    for (int i = 0; i < 30; i++) { // run 30 battle simulations
        String battleLog = battlesRepository.executeBattle("kienboec", "altenhof");

        // split battle log into single lines and find the one that contains "CHAOS BURST chance"
        String chaosBurstLine = null;

        for (String line : battleLog.split( regex: "\n")) {
            if (line.contains("CHAOS BURST chance for this battle:")){
                chaosBurstLine = line;
                break;
            }
        }
        assertNotNull(chaosBurstLine,  message: "CHAOS BURST chance should be present in the battle log");

        // get number by splitting at ":" - remove "%" and parse
        String percentageString = chaosBurstLine.split( regex: ":")[1].trim(); // get everything after ":"
        percentageString = percentageString.split( regex: "%")[0].trim(); // get everything before "%"

        percentageString = percentageString.replace( oldChar: ',',  newChar: '.'); // replace comma with dot cuz the parsing doesn't like the comma

        double chaosChance = Double.parseDouble(percentageString); // turn string into double

        // check if chaos burst chance is between 10% - 30%
        assertTrue( condition: chaosChance >= 10.0 && chaosChance <= 30.0,
                message: "CHAOS BURST chance should be between 10% - 30% : " + chaosChance + "%");
    }
}
```

Such unit tests include the testing of battle logic, whether or not exceptions are being thrown where they need to happen or the opposite and so on.

Overall the project took about ~25 hours to create in total. Time was first spent with researching and familiarizing with the general concepts of the HTTP Protocol handling and Server implementation and making sure simple requests like Registration and Login could even be handled in the first place, which took roughly 7 hours and was done over the beginning of to halfway point of the SWEN1 course.

Most of the other functionality was implemented later. The database connection and Docker container functionality took roughly 2 hours, but the database was continuously adapted to fit the project's functionality. Creation of card packages, buying them and then letting them be configured individually by users took roughly 6 hours. Letting users edit their data also took roughly 2 extra hours. Implementing the actual battle logic and handling of the battle itself, on top of creating the unique feature and making sure the battle log looks smooth took an additional 5-7 hours.

The biggest takeaways for the implementation of this application was definitely the knowledge of how to work with API Endpoints, HTTP Protocol handling and making use of a Rest Server and the handling of clients. Most of the app design's structure, like the splitting of Services and Repositories was also very new, as well as the creation of many different useful unit tests. Working so closely with a database was also very refreshing to work with and a big takeaway. Overall, for me personally, this was actually one of my favorite applications to implement so far over the course of my studies for me.