FH University of Applied Sciences TECHNIKUM WIEN

Monster Trading Cards Game (MCTG) - Specification

Create an application in Java or C# to spawn a REST-based (HTTP) server that acts as an API for possible frontends (WPF, JavaFX, Web, console). **The frontend is not part of this project!**

Features

This HTTP/REST-based server is built to be a platform for trading and battling with and against each other in a magical card-game world:

- a **user** is a registered player with **credentials** (unique username, password).
- a **user** can manage his **cards**.
- a **card** consists of: a name and multiple attributes (damage, element type).
- a **card** is either a **spell-card** or a **monster-card**.
- the damage of a **card** is constant and does not change.
- a **user** has multiple **cards** in his **stack**.
- a **stack** is the collection of all his current **cards** (hint: cards can be removed by **trading**).
- a **user** can buy **cards** by acquiring **packages**.
- a **package** consists of 5 **cards** and can be acquired from the server by paying 5 virtual **coins**.
- every **user** has 20 **coins** to buy 4 **packages**.
- the best 4 **cards** are selected by the **user** to be used in the **deck**.
- the **deck** is used in the **battles** against other players.
- a **battle** is a request to the server to compete against another user with your currently defined **deck** (see detail description below).

Users can:

- *register* and *login* to the server,
- *acquire* some **cards**,
- *define* a **deck** of monsters/spells,
- *battle* against another **user**
- *compare* their **stats** in the **score-board**.
- *trade* **cards** to have better chances to win (see detail description below).

Further Features:

- *display a* **scoreboard** (= sorted list of ELO values)
- *editable* **profile page**
- user **stats** – especially ELO *calculation* (+3 points for win, -5 for loss, starting value: 100; higher sophisticated ELO system welcome)
- *security check* (using the **token** that is retrieved at login on the server-side, so that user actions can only be performed by the corresponding user itself)

FH University of Applied Sciences TECHNIKUM WIEN

The Battle Logic

Your cards are split into 2 categories:

1. **monster-cards**: cards with active attacks and damage based on an element type (fire, water, normal). The element type does not effect pure monster fights.
2. **spell-cards**: a spell-card can attack with an element based spell (again fire, water, normal) which is:
    - effective (eg: water is effective against fire, so damage is doubled)
    - not effective (eg: fire is not effective against water, so damage is halved)
    - no effect (eg: normal monster vs normal spell, no change of damage, direct comparison between damages).

    Effectiveness:

    - water -> fire
    - fire -> normal
    - normal -> water

Battle-Rounds:

- Cards are chosen randomly each round from the deck to compete (this means 1 round is a battle of 2 cards = 1 of each player).
- There is no attacker or defender. All parties are equal in each round.
- Pure monster fights are not affected by the element type.
- As soon as 1 spell cards is played the element type has an effect on the damage calculation of this single round.
- Each round the card with higher calculated damage wins.
- Defeated monsters/spells of the competitor are removed from the competitor's deck and are taken over in the deck of the current player (vice versa).
- In case of a draw of a round no action takes place (no cards are moved).
- Because endless loops are possible we limit the count of rounds to 100 (ELO stays unchanged in case of a draw of the full game).

The following specialties are to consider:

- **Goblins** are too afraid of **Dragons** to attack.
- **Wizzard** can control **Orks** so they are not able to damage them.
- The armor of **Knights** is so heavy that **WaterSpells** make them drown them instantly.
- The **Kraken** is immune against **spells**.
- The **FireElves** know **Dragons** since they were little and can evade their attacks.

As a result of the battle we want to return a log which describes the battle in great detail. Afterwards the player stats (see scoreboard) need to be updated (count of games played and ELO calculation).

Add a unique feature (mandatory) to your solution e.g.: additional booster for 1 round to 1 card, spells… (be creative)

Attention: Everything which is not clearly specified is up to your choice! Document your decisions in the protocol document (see below)!

Trading Deals

You can request a trading deal by pushing a card (concrete instance, not card type) into the store (MUST NOT BE IN THE DECK and is locked for the deck in further usage) and add a requirement (Spell or Monster and additionally a type requirement or a minimum damage) for the card to trade with (eg: “spell or monster” and “min-damage: 50”).

Example:

```
Player A: adds WaterGoblin (50 damage) in the store and wants “Spell with
min 70 damage”.
Player B: accepts Trade with “RegularSpell (80 damage)”.
```

Implementation Requirements

- Build a REST-Server and implement the HTTP protocol stack on your own.
- You are not allowed to use any helper framework for the HTTP communication, but
- you are allowed to use nuget/mvn-packages (e.g. Jackson) for serialization of objects into strings (and vice versa)
- The data should be persisted in a PostgreSQL database.
- Test your application with the provided curl script (integration test) and
- add unit tests (~20+) to verify your application code.

Use the provided CURL script (integration test) to fetch the implementation details of the REST-API, e.g. HTTP-Method, Endpoint-Path, Parameters, Headers and Payload., e.g.

```
echo "2) Login Users"
curl -X POST http://localhost:10001/sessions \
    --header "Content-Type: application/json" \
    -d "{\"Username\":\"kienboec\", \"Password\":\"daniel\"}"
echo "should return generated token for the user, here: kienboec-mtcgToken"
```

This code snipped defines the REST-API as follows:

- `POST` - the HTTP Verb
- `/sessions` - the endpoint of the REST-API (HTTP Path)
- `--header "Content-Type: application/json"` - a HTTP-Header Parameter "Content-Type" with the value "application/json"
- `-d "{\"Username\":\"kienboec\", \"Password\":\"daniel\"}"` - defines the payload (HTTP Body) as a JSON formatted string.
- in this code-sample the token for the user should be returned in the HTTP-Response body (e.g. "kienboec-mtcgToken")

The HTTP-Response in general should return the corresponding HTTP-Response-Code (details see HTTP-specs.):

- if the request was fulfilled successfully, then the corresponding 2XX code should be returned and, if available, the result-content in the HTTP-Response-Body
- if the request failed due to missing parameters, authentication errors,... --> then the corresponding 4XX code should be returned
- if the request failed due to a server-error, e.g. PostgreSQL not connected --> then the corresponding 5XX code should be returned.

Hand-Ins

The submission is done in two steps:

1. **Intermediate-Submission** (class 10): covers the HTTP-Server with the User-Registration REST-API endpoints.
2. **Final Submission** (class 22): added business-logic and all other features.

Hand in the latest version of your source code as a zip in Moodle (legal issue) with a README.txt (or md)-file pointing to your git-repository.

Add a protocol document with the following content:

- protocol about the technical steps you made (designs, failures and selected solutions)
- explain why these unit tests are chosen and why the tested code is critical
- track the time spent with the project
- consider that the git-history is part of the documentation (no need to copy it into the protocol)

See the corresponding Checklist-Excel sheets for the MUST-HAVES, Grading-Items and Grading-Points.. Late submissions are not accepted! Missing hand-ins or missing MUST-HAVEs will automatically grade the submission with 0 points!

Final Presentation

For the final presentation in class 22-25, be prepared with your

- working solution already started on your machine
- setup your environment so you can start the curl tests directly.
- open your design (see: protocol) to show your architecture / approach.

Optional Features

With optional features implemented you can compensate possible errors in the implementation above. Nevertheless, it is not possible to exceed the maximum number of points (= 100%).