FH University of Applied Sciences TECHNIKUM WIEN

Monster Trading Cards Game (MTCG)

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).

Game mechanics

Users can

- register and login to the server,
- acquire some cards,
- define a deck of monsters/spells,
- battle against each other and
- compare their stats in the score-board.

Hereby you can trade cards to have better chances to win (see detail description below). The data should be persisted in a postgreSQL database.

Further features:

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

The Battle Logic

Your cards are split into 2 categories:

- **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.

- **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

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).

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).

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.

Example

Monster Fights (= round with only monster cards involved):

- PlayerA: WaterGoblin (10 Damage) vs PlayerB: FireTroll (15 Damage) => Troll defeats Goblin
- PlayerB: FireTroll (15 Damage) vs PlayerA: WaterGoblin (10 Damage) => Troll defeats Goblin

Spell Fights (= round with only spell cards involved):

- PlayerA: FireSpell (10 Damage) vs PlayerB: WaterSpell (20 Damage) => 10 VS 20 -> 05 VS 40 => WaterSpell wins
- PlayerA: FireSpell (20 Damage) vs PlayerB: WaterSpell (05 Damage) => 20 VS 05 -> 10 VS 10 => Draw (no action)
- PlayerA: FireSpell (90 Damage) vs PlayerB: WaterSpell (05 Damage) => 90 VS 05 -> 45 VS 10 => FireSpell wins

Mixed Fights (= round with a spell card vs a monster card):

- PlayerA: FireSpell (10 Damage) vs PlayerB: WaterGoblin (10 Damage) => 10 vs 10 -> 05 vs 20 => WaterGoblin wins
- PlayerA: WaterSpell (10 Damage) vs PlayerB: WaterGoblin (10 Damage) => 10 vs 10 -> 10 vs 10 => Draw
- PlayerA: RegularSpell (10 Damage) vs PlayerB: WaterGoblin (10 Damage) => 10 vs 10 > 20 vs 05 => Knight wins
- PlayerA: RegularSpell (10 Damage) vs PlayerB: Knight (15 Damage) => 10 vs 15 -> 10 vs 15 => Knight wins

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)”.

HandIns

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! You are not allowed to use any helper framework for the HTTP communication, but you are allowed to use nuget (JSON.NET) /mvn-packages (Jackson) for serialization of objects into strings (vice versa). Test your application with the provided curl script (integration test) and add unit tests (~20+) to verify your application code.

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

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 as plain-text file (txt or md) 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)

For the final presentation 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.

Please be aware that the curl-tests maybe altered throughout the course (one week before final presentation). In case of custom adjustments (depending on your implementation) in the curl scripts please adapt these final versions as well.

FH University of Applied Sciences TECHNIKUM WIEN

Intermediate Submissions

Be able to present your solution during the course (in the exercise lessons (de: Übungen)) at the beamer.

Goals:

- after theory lesson 1 (OOP):
  - Choose a language (C# or Java) and get your environment up and running.
  - Start creating the business layer and design/develop the corresponding classes.
  - Test with command line.
- after theory lesson 2 (Networking / Parallel Execution):
  - Implement the TCP-Communication over HTTP and be able to communicate with the testing script provided.
  - The server should be able to read the most important headers and read the body correctly.
  - Consider parallel execution here and prepare a method to test this (e.g.: console.log and parallel running curl scripts).
- after theory lesson 3 (Testing):
  - Check your code and add unit tests to critical parts of your implementation.
  - Explain the difference between unit testing and the usage of the curl script.
  - Implement register / login and the user profile methods with focus on security (see the "Authorization" headers in the scripts) and test these aspects using unit tests.
- after theory lesson 4 (Collections / Files / Streams):
  - Explain which collection type is chosen for each purpose and why (add this to your protocol to hand-in)
  - Implement the package and card (+deck) handling
- after theory lesson 5 (Json, Databases):
  - Implement the trading possibility.
  - store all relevant data in the database (use SQL only; no O/R-Mappers allowed).
- after theory lesson 6 (Recap):
  - implement further requirements (90% of the project should be finished)

Consider that theory questions are asked and that these intermediate presentations are graded as well. Always be prepared to test edge-cases and to extend your curl script for the

given purpose, BUT be prepared that the curl script can be extended by the lecturers as well (new versions might test further error cases eg: check for invalid decks).

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%).

- Trading system: trade cards vs coins
- Further card classes (eg: trap cards ~ passive spells)
- Further element types (eg: ice, wind,…) including additional dependencies
- Friends-List (play against friends from list, manage friends by username)
- Card Description
- Extended Scoreboard: see ELO or WHR
- be able to add virtual coins to your system (with stats on spent coins)
- transaction history
- win/lose ratio in user stats

Grading

- 30: functional requirements
  - Battle
    - play rounds,
    - draw possible,
    - clean log of a battle,
    - take-over cards after loss of a round,
    - consider specialties in battle-rounds between cards
  - User Management (Register, ProfilePage)
  - ScoreBoard and User Stats Management
  - Trading
  - mandatory unique feature
- 20: non-functional requirements
  - Token-based security
  - Persistence (DB)
  - Unit Tests

  - Integration Tests (curl or alternatively a custom app that works in an automated way)
- 05: protocol
  - design / lessons learned
  - unit test design
  - time spent
  - link to git