Cards Against Humanity

# Group

Mateusz Łąpieś

# Project description

Project will try to recreate gameplay of popular game Cards Against Humanity. It will be a multiplayer game dedicated to be played on a LAN network over Wi-Fi of laptop hotspot. Implementation will be performed in form of a web application using React.js for frontend and Spring Boot application as a backend. Targeted will be mobile devices.

# Additional libraries

## Frontend

* MDB Bootstrap
  + <https://mdbootstrap.com/>

## Backend

* Spring Web
* WebSocket
* Spring Data Reactive MongoDB
* Embedded MongoDB Database

# Functionalities

* Account registration
  + Form allowing for user to create an account using login and password.
  + Every account will have their amount of won games recorded in the database.
* Logging in
  + Anonymously by providing just a user name (user not saved to the database).
  + Using created account by providing login and password.
* Cards
  + Cards will be stored in a MongoDB database.
  + Base class of the card will contain following fields:
    - id (UUID)
    - text (String)
  + Cards split between to types: black and white, both types will be stored in separate collections. Class of the black cards will extend will extend the base card class as it requires additional field:
    - blanks (int)
  + During the game, web socket will only operate using ids of the cards. Frontend will have to pull the rest of the information using the regular http requests.
* Creating the game
  + Application will allow registered user to create a game and allow them to select time limit to make selection and required amount of won rounds to be declared a winner.
  + Game will be identified by a code which will be used by other players to join the game. To allow faster way of sharing the code application will offer a QR code to be displayed on the phone of the player which then can be scanned by other players to join.
* Joining the game
  + Any type of player, that has been already authenticated will be able to join the game by providing a code on the website using form or by scanning QR code.
* Gameplay
  + Game is built on the base of two types of cards, black (questions / statements with blanks to fill) and white (answers / words). In the beginning of each round, every player should have 11 white cards, unless there are no more cards left. They pick their white card(s) (amount specific for each black card) and confirm their selection. Every round new player is selected to be a cezar, a person that will be picking the best answer (provided by other players) to a question that has been drawn. Game ends when the limit of won rounds is reached.
  + On the start of each round, cezar’s name will be announced together with a black card that has been drawn.

# Implementation

## Github

### Repository

All the files are being held in a public github repository as it makes it easier to deploy the solution. <https://github.com/mateuszlapies/studia/tree/I_Semestr/ztp/projekt>

### Folders

Repository is split into three categories:

* backend
* frontend
* docker

## Backend

This part of the project can be easily split into three parts: database, rest interface and socket endpoint.

### Base classes

* GameInstance – contains all the information about the current state of the game, its players, their cards, current black card, current cezar and parameters of the game defined by the first player.
* GameParameters – structures parameters of the game (time, round limit)
* Requests
* Responses
* UserService – provides source of user data in form of class which implements UserDetails interface
* UserServiceImpl – utilizes UserService and implements UserDetailsService

### Database

For each of the classes there is a mongodb repository interface that uses them as structure for the data stored in form of a json in the database.

#### User

This class apart from being used for database it also implements UserDetails interface which allows to use this class for basic authentication purposes.

The structure of the class is following:

* String – id (automatically generated by the database)
* String – user
* String – pass
* Role - role (enum which represents role of the user, it is also used for basic authentication)
* int – history (amount of wins)

#### Card

Structure of the class:

* String – id (automatically generated by the database)
* String – text
* int – blanks (represents amount of white cards that should be selected – only applies to black cards)
* Color – color (enum that represents type / color of the card with a char – b or w)

### Rest interface

#### Configuration

Class WebConfig implements WebSecurityConfigurerAdapter. Using exposed properties way of authentication is being defined as well as what endpoints are secured and in what level.

#### Endpoints

##### (built-in) GET /actuator/health

Responds with a json containing status of the backend application. Status UP represents correctly initialized and working application.

##### (admin) GET /restart

Restarts application – to start a new game

##### (authenticated) GET /me

Responds with a json containing authenticated user data excluding its password.

##### POST /users

Allows anyone to create their account using username and password. Responds with json containing code 200 and message OK.

##### (authenticated) GET /users/{id}

This path allows authenticated users to retrieve other players names using id.

##### (admin) DELETE /users/{id}

Providing id as a parameter of this endpoint admin can remove user profile from the database.

##### (authenticated) GET /cards/{id}

Endpoint allowing to retrieve details of a specific card, defined by an id.

### Socket endpoint

#### Configuration

Using ChannelInterceptor interface in AuthChannelInterceptorAdapter class, method of basic authentication is being implemented.

#### Listeners

##### In – Out /sock/info

Players who have just authenticated or expect an update call this path to receive current object of GameInstance in form of a json. In case of fresh players it also registers them in the object of the game. In case of the first player, it is being set as an owner of the lobby and as a first cezar.

##### In /sock/start – Out /sock/info

First player can start the game by calling this path. If the identity is confirmed the game is being set as started and object of the GameInstance is sent to all the players.

##### In /sock/cards – Out /sock/cards

When the game is started, every player calls for the set of ids of cards that are assigned to them.

##### In /sock/submitted – Out /sock/submitted

During the round, players are asked to select limited amount of white cards, they can use this path to send ids of cards that they have selected.

##### In /sock/select – Out /sock/win

When all players have selected their cards or the time limit is reached, cezar selects the winner. If the identity of the cezar is confirmed winner is announced to all the players.

##### In /sock/timeout – Out /sock/info

As cezar is not involved in selection of cards timeout is called by cezar. If the time has actually ran out, faze of selection of white cards is ended.

## Frontend

### Registration

### Authentication

### Gameplay

## Docker

### Dockerfile

To define procedure to correctly build both solutions: frontend and backend. Two separate Dockerfiles were used. The naming is as follows: Dockerfile.spring, Dockerfile.react.

To run both of the builds and perform configuration powershell script was created - build.ps1. This way both images will be available in local docker repository. Their names are equal to extensions of Dockerfiles: spring, react.

### docker-compose

Having images build we can use docker-compose to create set of containers defined in   
docker-compose.yml file. To make it easier to use the tool cmd script was also created – compose.cmd.

Obraz zawierający tekst

Opis wygenerowany automatycznie

# Schema

# 