Projekt: Trach Game
Data modyfikacji: 19.05.2019 r.

Zespół:
Mateusz Kobak,
Krzysztof Piesiewicz,
Tomasz Grześkiewicz,
Karolina Gabara

Modele i komunikacja

Modele gry

- **Karta**
  Posiada:
  - identyfikator,
  - typ wyrażony kodem (napisem).

  Karty dzielą się na
  1. Zagrywalne - można zacząć od nich swoją turę
  2. Karty odpowiedzi - karta, którą można odpowiedzieć na akcję innego gracza, najczęściej pewien sposób na uchronienie się przed atakiem
  3. Karty modyfikacji - karty modyfikujące inne karty, rzucane jedynie w połączeniu z innymi

  Rodzaje kart: *Atak, Obrona, Odbicie, Przebicie, Przerzut, Pustak, Uzdrowienie, Wskrzeszenie, Zamrożenie, Atomowy Guzik, Schron, Super Trach, Zmasowany Atak*

- **Gracz**
  Posiada:
  - identyfikator,
  - punkty życia (liczba całkowita z przedziału [0, 5])
  - lista kart należących do gracza, które może zagrywać (zawsze jest ich 5)
- **Stół**
  Posiada:
  - Stos kart do dobierania
  - Stos kart zgranych
  - Drzewo kart zagranych w aktualnie trwającej turze.

Modele systemu

- **Użytkownik**
  Posiada:
  - identyfikator,
  - nazwę,
  - adres e-mail,
  - hasło.
- **Rozgrywka**
  Posiada:
  - identyfikator,
  - mapę przyporządkowującą użytkownikom graczy, którymi sterują.

Projekt: Trach Game
Data modyfikacji: 19.05.2019 r.

Oznaczenia używane w definicjach zapytań i odpowiedzi HTTPS, websocketowych wiadomości i typów w nich występujących

Wszystkie wiadomości są obiektami w formacie JSON w standardzie ECMA-404 The JSON Data Interchange Standard (http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf).

Na potrzeby systemu zdefiniowane są następujące typy:

- `Int` - 32-bitowa liczba całkowita ze znakiem (od -2147483648 do 2147483647 wł.) reprezentowana w formacie JSON przez typ ***number*** z dozwolonymi znakami: '0'-'9' i '-';
- `Long` - 64-bitowa liczba całkowita ze znakiem (od -9223372036854775808 do 9223372036854775807 wł.) reprezentowana w formacie JSON przez typ ***number*** z dozwolonymi znakami: '0'-'9' i '-';
- `String` - napis reprezentowany w formacie JSON przez typ ***string***;
- `Datetime` - ***string*** pasujący do szablonu: `"YYYY-MM-DD hh:mm:ss"`, gdzie `YYYY` to rok, `MM` miesiąc, `DD` - dzień, `hh` - godzina, `mm` - minuty, `ss` - sekundy;
  np. `"2019-01-01 02:30:00"`;
  Uwaga: daty przedstawiamy w UTC.

Tablice, reprezentowane przez JSON ***array***. Napis `[<Type>, <Type>, ...]` oznacza tablicę o dowolnej skończonej liczbie elementów (w szczególności tablica może być pusta), z których każdy jest typu `Type`, gdzie `Type` to jeden ze zdefiniowanych wyżej typów: `Int`, `Long`, `String`, `Datetime` lub inny typ/alias opisany w tym dokumencie.

Rejestracja, uwierzytelnienie, otwarcie websocketu

Rejestracja i uwierzytelnienie użytkownika za pomocą aplikacji klienta prowadzone są przez HTTPS z obsługą sesji i CSRF.

- Żądanie otrzymania tokena CSRF wysyłane do serwera
  - protokół HTTPS, endpoint: /token, metoda: GET, pusta zawartość.

  Odpowiedzi serwera na żądanie tokena CSRF:
  - gdy brak błędów- status: 200

    ```
    {
      "csrfToken": <String>
    };
    ```

  - w przypadku innych błędów z odpowiednim statusem.

- Żądanie rejestracji użytkownika wysyłane do serwera
  - protokół HTTPS, endpoint: /register, metoda: POST, zawartość:

  ```
  {
    "username": <String>,
    "email": <String>,
    "password": <String>,
  ```

```
  "csrfToken": <String>
}.
```

Odpowiedź serwera na żądanie rejestracji

- w przypadku pomyślnej rejestracji - status: 201;
- gdy adres email jest w użyciu - status: 400

```
{
  "errorType": "email",
  "msg": <String>
};
```

- gdy hasło nie spełnia wymogów - status: 400

```
{
  "errorType": "password",
  "msg": <String>
};
```

- gdy nazwa użytkownika nie spełnia wymogów - status: 400

```
{
  "errorType": "username",
  "msg": <String>
};
```

- w przypadku innych błędów z odpowiednim statusem.

- Żądanie uwierzytelnienia użytkownika wysyłane do serwera
  - protokół HTTPS, endpoint: /login, method: POST, zawartość:

```
{
  "email": <String>,
  "password": <String>,
  "csrfToken": <String>
}.
```

Odpowiedzi serwera na żądanie uwierzytelnienia

- w przypadku pomyślnego uwierzytelnienia - status: 200;
- gdy dane uwierzytelniające są błędne - status: 400

```
{
  "errorType": "login",
  "msg": <String>
};
```

- w przypadku innych błędów z odpowiednim statusem.

- Żądanie otwarcia websocketu wysyłane do serwera
  - protokół: WSS, endpoint: /ws.

Odpowiedź serwera na żądanie otwarcia websocketu

- w przypadku braku uwierzytelnienia - odrzucenie połączenia.

- Żądanie otwarcia websocketu wysyłane do serwera
  - protokół: WS, endpoint: /ws. (czasowo zamiast WSS)

Projekt: Trach Game
Data modyfikacji: 19.05.2019 r.

Definicje typów, związanych z obiektami gry, występujących w wiadomościach (webclient + desktop client + server + bot + game)

- typ `GameState`

```
{
  "players": [<Player>, <Player>, ...],
  "coveredCardsStack": [<Card or CoveredCard>, <Card or CoveredCard>, ...],
  "usedCardsStack": [<Card or CoveredCard>, <Card or CoveredCard>, ...],
  "tableActiveCards": [<Card>, <Card>, ...],
  "cardTrees": [<CardTree>, <CardTree>, ⋯],
  "roundId": <Int>,
  "playerIdOnMove": <Int>
};
```

  - ~~➢ pole "cardTree" jest opcjonalne!~~
  - ➢ W zwykłym trybie gry karty na stosie kart zakrytych (`"coveredCardsStack"`) są zasłonięte (`<CoveredCard>`), a na stosie kart użytych (`"usedCardsStack"`) są odsłonięte.
  - ➢ pole `"playerIdOnMove"` to identyfikator gracza, którego tura trwa. Jeśli pole ~~`"cardTree"` nie występuje~~ `"cardTrees"` ma pustą listę, to gra oczekuje na akcję rozpoczęcia tury przez tego gracza (na zagranie drzewa kart `PlayedCardsRequest` lub wymianę kart z ręki `HandExchangeRequest`).

- typ `Player`

```
{
  "id": <Int>,
  "name": <String>,
  "health": <Int>,
  "hand": [<Card or CoveredCard>, <Card or CoveredCard>, ...],
  "activeCards": [<Card>, <Card>, ...]
};
```

  Pole `"hand"`, czyli ręka gracza, zawiera listę kart. Klient otrzymuje stan gry, w którym na ręce gracza użytkownika są karty odsłonięte `<Card>`, a na rękach pozostałych graczy karty zasłonięte `<CoveredCard>`.

- typ `Card`

```
{
  "id": <Int>,
  "type": <String>
},
```

  gdzie pod kluczem `"type"` znajduje się jeden z kodów:

kod:	karta:
`"attack"`	*Atak*
`"mass_attack"`	*Zmasowany atak*

`"defence"`	*Obrona*
`"reflection"`	*Odbicie*
`"transfer"`	*Przerzut*
`"break_through"`	*Przebicie*
`"priority_inc"`	*Podniesienie priorytetu*
`"airbrick"`	*Pustak*
`"healing"`	*Uzdrowienie*
`"resurrection"`	*Wskrzeszenie*
`"freeze"`	*Zamrożenie*
`"atomic_bomb"`	*Atomowy guzik*
`"shelter"`	*Schron*
`"super_trach"`	*Super Trach*

- typ `CoveredCard`

```
{
  "id": -1,
  "type": "covered_card"
},
```

- typ `VirtualCard`

```
{
  "id": <Int>,
  "type": <String>,
  "virtual": true
},
```

- typ `CardTree`

```
{
  "id": <Int>,
  "playedCard": <PlayedStartingCard>,
  "childrenNodes": [<CardNode>, <CardNode>, ...]
};
```

- typ `CardNode`

```
{
  "playedCard": <PlayedCardInTree>,
  "childrenNodes": [<CardNode>, <CardNode>, ...]
};
```

- typ `PlayedCard` dzieli się na trzy główne rodzaje: `PlayedStartingCard`, `PlayedCardInTree`, `PlayedSuperTrachCard`.

- typ `PlayedStartingCard` występujący jako:
  - typ `PlayedStartingCard`

    ```
    {
      "type": "PlayedStartingCard",
      "card": <Card or VirtualCard>,
      "whoPlayedId": <Int>
    },
    ```

    można tak zagrywać karty: *Zmasowany atak, Pustak, Zamrożenie, Atomowy guzik;*
  - typ `PlayedStartingCardAtPlayer`

    ```
    {
      "type": "PlayedStartingCardAtPlayer",
      "card": <Card or VirtualCard>,
      "whoPlayedId": <Int>,
      "targetPlayerId": <Int>
    },
    ```

    można tak zagrywać karty: *Atak, Uzdrowienie;*
  - typ `PlayedStartingCardAtCard`

    ```
    {
      "type": "PlayedStartingCardAtCard",
      "card": <Card or VirtualCard>,
      "whoPlayedId": <Int>,
      "targetCardId": <Int>
    };
    ```
- typ `PlayedCardInTree`
  - typ `PlayedCardInTree`

    ```
    {
      "type": "PlayedCardInTree",
      "card": <Card or VirtualCard>,
      "whoPlayedId": <Int>,
      "parentCardId": <Int>
    },
    ```

    można tak zagrywać karty: *Obrona, Odbicie, Przebicie, Podniesienie priorytetu, Schron;*
  - typ `PlayedCardInTreeAtPlayer`

    ```
    {
      "type": "PlayedCardInTreeAtPlayer",
      "card": <Card or VirtualCard>,
      "whoPlayedId": <Int>,
      "parentCardId": <Int>,
      "targetPlayerId": <Int>
    },
    ```

    można tak zagrywać karty: *Przerzut;*
- typ `PlayedSuperTrachCard`

```
{
  "type": "PlayedSuperTrachCard",
  "card": <Card or VirtualCard>,
  "whoPlayedId": <Int>,
  "playedCard": <PlayedCard>
},
```

gdzie w polu `"card"` jest karta *Super Trach*. Obiekt `PlayedSuperTrachCard` może być użyty wszędzie tam, gdzie obiekt z pola `"playedCard"` (innymi słowy, obiekt `PlayedSuperTrachCard` jest traktowany jak obiekt `PlayedCard` o typie `"playedCard"/"type"`).

Jeśli obiekt `PlayedSuperTrachCard` jest wysyłany w żądaniu zagrania od klienta, to obiekt z pola `"playedCard"/"card"` musi być wirtualną kartą o id równym -1. Z kolei `"playedCard"/"whoPlayedId"` musi być równe `"whoPlayedId"`; np. żądanie zagrania super tracha jako ataku:

```
{
  "type": "PlayedSuperTrachCard",
  "card": {
    "id": 10,
    "type": "super_trach"
  },
  "whoPlayedId": 1,
  "playedCard": {
    "type": "PlayedStartingCardAtPlayer",
    "card": {
      "id": -1,
      "type": "attack",
      "virtual": true
    },
    "whoPlayedId": 1,
    "targetPlayerId": 2
  }
}
```

Definicje typów, związanych z elementami systemu, występujących w wiadomościach (webclient + desktop client + server)

- typ `GamePlayResult`

```
{
  "gamePlayId": <Long>,
  "winnerId": <Int>
}
```

- typ `GamePlayInfo` występujący jako:
  - typ `MultiPlayerGamePlayInfo`

```
{
```

```
  "type": "MultiPlayerGamePlay",
  "gamePlayId": <Long>,
  "created": <Datetime>,
};
```

- typ SinglePlayerGamePlayInfo

```
{
  "type": "SinglePlayerGamePlay",
  "gamePlayId": <Long>,
  "created": <Datetime>
}.
```

Wiadomości dot. rozgrywek (webclient + desktop client + server)

- Wiadomość dla serwera od klienta z żądaniem informacji o stanie gry

```
{
  "msgType": "GameStateRequest",
  "gamePlayId": <Long>
}.
```

- Wiadomość dla klienta od serwera informująca o stanie gry

```
{
  "msgType": "GameStateUpdate",
  "gamePlayId": <Long>,
  "updateId": <Long>,
  "gameState": <GameState>,
  "timeOfComingEvaluation": <Datetime>
},
```

gdzie pole `"timeOfComingEvaluation"` jest opcjonalne; jego wartość to czas, podany w UTC, w którym nastąpi najbliższa ewaluacja stołu z drzewem kart i aplikacja zmieniająca stan. Gdy pole jest nieobecne, to nie jest planowana żadna ewaluacja (np. na początku tury).

- Wiadomość dla serwera od klienta z żądaniem zagrania karty

```
{
  "msgType": "PlayedCardsRequest",
  "gamePlayId": <Long>,
  "updateId": <Long>,
  "playerId": <Int>,
  "played": <CardTreeOrCardNode>
},
```

gdzie `<CardTreeOrCardNode>` przyjmuje wartość: `<CardTree>` lub `<CardNode>`. Jeśli w polu `"played"` znajduje się `<CardTree>`, to wiadomość jest żądaniem utworzenia drzewa kart. Natomiast jeśli jest tam `<CardNode>`, to jest to żądanie podczepienia poddrzewa.

- Wiadomość dla serwera od klienta z żądaniem wymiany wybranych kart z ręki na karty ze stosu kart zakrytych. Pole powinno zawierać listę (o długości od 0 do 3 włącznie) różnych identyfikatorów kart z ręki gracza.

```
{
  "msgType": "HandExchangeRequest",
  "gamePlayId": <Long>,
  "updateId": <Long>,
  "playerId": <Int>,
  "cardsIdsToExchange": [<Int>, <Int>, ...]
}.
```

- Wiadomość dla serwera od klienta informująca, że gracz o identyfikatorze `playerId` nie chce wykonywać żadnych akcji przy obecnym stanie gry (otrzymanym przy aktualizacji z danym `updateId`). Wiadomość, odebrana na początku tury gracza, nie jest brana pod uwagę.

```
{
  "msgType": "NoActionRequest",
  "gamePlayId": <Long>,
  "updateId": <Long>,
  "playerId": <Int>
}.
```

- Wiadomość dla serwera od klienta z żądaniem informacji o stanie rozgrywki

```
{
  "msgType": "GamePlayInfoRequest",
  "gamePlayId": <Long>
}.
```

- Wiadomość dla klienta od serwera informująca o zmianie stanu rozgrywki

```
{
  "msgType": "GamePlayInfoUpdate",
  "gamePlayId": <Long>,
  "playerId": <Int>,
  "gamePlayState": <GamePlayState>
},
```

gdzie:

  - `"playerId"` - identyfikator gracza, którym steruje użytkownik,
  - `<GamePlayState>` przyjmuje wartość ze zbioru: `{"running", "stopped", "finished"},`

- Wiadomość dla serwera od klienta z żądaniem podania wyniku rozgrywki

```
{
  "msgType": "GamePlayResultRequest",
  "gamePlayId": <Long>
}.
```

- Wiadomość dla klienta od serwera informująca o wyniku rozgrywki

```
{
  "msgType": "GamePlayResult",
  "gamePlayId": <Long>,
  "winnerId": <Int>
},
```

gdzie pod kluczem `"winnerId"` znajduje się identyfikator zwycięzcy lub wartość `-1`, gdy takiego

nie ma.

- Wiadomość dla serwera od klienta z żądaniem rozpoczęcia szybkiej rozgrywki wieloosobowej

```
{
  "msgType": "QuickMultiplayerGameRequest"
},
```

- Wiadomość dla serwera od klienta z żądaniem wejścia użytkownika do danej rozgrywki

```
{
  "msgType": "EnterGamePlayRequest",
  "gamePlayId": <Long>
}.
```

- Wiadomość dla serwera od klienta z żądaniem listy rozgrywek (jednoosobowych i wieloosobowych), do których użytkownik może wejść

```
{
  "msgType": "AvailableGamePlaysRequest"
}.
```

- Wiadomość dla klienta od serwera z informacją o liście rozgrywek (jednoosobowych i wieloosobowych), do których użytkownik może wejść

```
{
  "msgType": "AvailableGamePlays",
  "gamePlays": [<GamePlayInfo>, <GamePlayInfo>, ...]
}.
```

Wiadomości dot. rozgrywek (server + game)