l'école d'ingénierie informatique

EPSI BORDEAUX - I1 EISI

Module Atelier - Architecture applicative (5/9)

Code Module	Durée	Titre Diplôme	Bloc de Compétences	Promotion	Auteur
ARCE842 - DEVE702	20h	EISI / RNCP 35584	Concevoir & Développer des solutions applicatives métiers	2023/2024	Julien COURAUD

10. Le timer

Le timer sera une donnée importante de notre moteur de jeu. C'est sa valeur qui dirigera la partie.

10.1. Partie WebSocket Server

Dans un premier temps, nous allons nous focus sur la partie Serveur.

C'est donc l'entité en charge de la synchronisation des deux clients concernés par le timer d'une partie avec la valeur disponible dans l'objet lié à la partie en cours.

Si sa valeur tombe à zéro, nous le réinitialisation à sa valeur par défaut.

Initialisation et durée d'un tour

Nous allons d'or et déjà prévoir une constante globale afin que la durée par défaut d'un tour soit configurable à un seul endroit dans notre code.

Nous pouvons imaginer de multiples scénarios où cette valeur pourrait être configurable. Par exemple: alimenter plusieurs types de parties (rapide et plus longues), ou implémenter des modes de jeu fun où certaines combinaisons ferait perdre du temps à l'adversaire.

Bref, il est nécessaire d'avoir cette donnée disponible dans le service d'initialisation pour que les méthodes du service puissent manipuler les objets de parties (`games[gameIndex]`).

```
// Durée d'un tour en secondes
const TURN_DURATION = 30;

const GAME_INIT = {
    gameState: {
        currentTurn: 'player:1',
        timer: TURN_DURATION,
        player1Score: 0,
        player2Score: 0,
        grid: [],
        choices: {},
```

```
        deck: {}
    }
}
```

Lancement de l'horloge à la création d'une partie

L'horloge est lancée à la création d'une partie suite à l'initialisation du timer (`socket.emit('game.timer')` toutes les secondes) aux deux joueurs.

Nous utiliserons la méthode javascript `setInterval` pour ticker toutes les secondes et envoyer l'état de l'horloge aux composants Frontend `PlayerTimer` et `OpponentTimer`.

```
// websocket-game-server/index.js

const createGame = (player1Socket, player2Socket) => {

  // ...

  // On execute une fonction toutes les secondes (1000 ms)
  const gameInterval = setInterval(() => {

    games[gameIndex].gameState.timer--;

  }, 1000);

  // On prévoit de couper l'horloge
  // pour le moment uniquement quand le socket se déconnecte
  player1Socket.on('disconnect', () => {
    clearInterval(gameInterval);
  });

  player2Socket.on('disconnect', () => {
    clearInterval(gameInterval);
  });

};

// ...
```

Réinitialisation du timer et fin de tour

Lorsque le timer tombe à zéro, on finit le tour.

- On reset le timer à sa valeur de tour par défaut

- On change l'attribut `gameState.currentTurn`

```
// websocket-game-server/index.js

const gameInterval = setInterval(() => {

  games[gameIndex].gameState.timer--;

  // Si le timer tombe à zéro
  if (games[gameIndex].gameState.timer === 0) {

    // On change de tour en inversant le clé dans 'currentTurn'
    games[gameIndex].gameState.currentTurn = games[gameIndex].gameState.currentTurn === 'player:1' ? 'player:

    // Méthode du service qui renvoie la constante 'TURN_DURATION'
    games[gameIndex].gameState.timer = GameService.timer.getTurnDuration();
  }
}, 1000);
```

Préparation et envoi de l'objet aux deux clients

Notre manipulation de l'objet prévoit toutes les évolutions possibles du timer actuellement. La valeur pourra évidemment changer à d'autres occasions comme des interactions avec des composants du jeu ou par exemple si le joueur n'a pas fait de combinaisons de dés à son dernier lancer. Mais on verra cela plus tard.

Ici se pose tout de même une question de conception. Doit-on dupliquer les sockets à envoyer à la vue pour chacun des deux composants `PlayerTimer` et `OpponentTimer` côté Client, donc sur deux sockets différents ? Ou préfèrons-nous émettre sur un seul socket, adapté évidemment au client, mais qui sera écouté par les deux composants ?

La réponse varie selon les cas d'usages (comme souvent) et ici nos objets sont très petits et nous allons essayer d'optimiser le nombre de sockets envoyés. Les deux protagonistes d'une partie ne recevront donc qu'un seul socket pour avoir les données du timer.

```
// websocket-game-server/index.js

const gameInterval = setInterval(() => {

  games[gameIndex].gameState.timer--;

  // Si le timer tombe à zéro
  if (games[gameIndex].gameState.timer === 0) {

    // On change de tour en inversant le clé dans 'currentTurn'
    games[gameIndex].gameState.currentTurn = games[gameIndex].gameState.currentTurn === 'player:1' ? 'player:

    // Méthode du service qui renvoie la constante 'TURN_DURATION'
    games[gameIndex].gameState.timer = GameService.timer.getTurnDuration();
  }

  // On notifie finalement les clients que les données sont mises à jour.
  games[gameIndex].player1Socket.emit('game.timer', GameService.send.forPlayer.gameTimer('player:1', games[ga
  games[gameIndex].player2Socket.emit('game.timer', GameService.send.forPlayer.gameTimer('player:2', games[ga

}, 1000);
```

Nous devons donc retourner un objet qui donne les informations pour les deux composants `PlayerTimer` et `OpponentTimer`.

```
send: {
    forPlayer: {
        viewGameState: (playerKey, game) => {
          // ...
        },
        viewQueueState: () => {
          //...
        },
        gameTimer: (playerKey, gameState) => {
            // Selon la clé du joueur on adapte la réponse (player / opponent)
            const playerTimer = gameState.currentTurn === playerKey ? gameState.timer : 0;
            const opponentTimer = gameState.currentTurn === playerKey ? 0 : gameState.timer;
            return { playerTimer: playerTimer, opponentTimer: opponentTimer };
        },
    }
},
```

10.2. Partie Client

```
// app/components/board/timers/player-timer.component.js

const PlayerTimer = () => {
```

```
  const socket = useContext(SocketContext);
  const [playerTimer, setPlayerTimer] = useState(0);

  useEffect(() => {
    socket.on("game.timer", (data) => {
      setPlayerTimer(data['playerTimer'])
    });
  }, []);

  return (
    <View style={styles.playerTimerContainer}>
      <Text>Timer: {playerTimer}</Text>
    </View>
  );
};
```

```
// app/components/board/timers/opponent-timer.component.js

const OpponentTimer = () => {
  const socket = useContext(SocketContext);
  const [opponentTimer, setOpponentTimer] = useState(0);

  useEffect(() => {

    socket.on("game.timer", (data) => {
      setOpponentTimer(data['opponentTimer'])
    });

  }, []);
  return (
    <View style={styles.opponentTimerContainer}>
      <Text>Timer: {opponentTimer}</Text>
    </View>
  );
};
```

/ ! \ Attention aux 'import .. from …' et au style CSS à également déporter / ! \

11. Les decks, les dés et le bouton 'Roll'

Cette section sera présentée sous forme d'un exercice. Via le code des composants `PlayerDeck` et `OpponentDeck` côté Client, vous devrez implémenter la logique côté WebSocket Server des mécanismes liés aux dés.

Cela consistera donc à manipuler des sockets, effectuer des actions sur le `gameState` de l'objet de la partie puis notifier les deux clients que l'affichage des decks doit être mis à jour.

11.1 Partie Client et besoins d'interface

- `PlayerDeck` :
  - useState :
  - `displayPlayerDeck` : On affiche ou non le deck (le serveur renvoie true si c'est le tour du joueur et false si c'est le tour de l'adversaire).
  - `dices` : Objet contenant la valeur des dés et leur état (locked).
  - `displayRollButton` : On affiche ou non le bouton 'Roll' (false à partir du dernier lancé).
  - `rollsCounter` : Décompte des lancers (commence à 1).
  - `rollsMaximum` : Nombre maximum de lancers.
  - socket.on :
  - `game.deck.view-state` : Met à jour tous ses 'useState'.
  - socket.on :
  - `game.dices.roll` : Clic sur bouton. 'Roll'
  - `game.dices.lock` : Clic sur un dé.

```
// app/components/board/decks/player-deck.component.js

import React, { useState, useContext, useEffect } from "react";
import { View, TouchableOpacity, Text, StyleSheet } from "react-native";
import { SocketContext } from "../../../contexts/socket.context";
import Dice from "./dice.component";

const PlayerDeck = () => {

  const socket = useContext(SocketContext);
  const [displayPlayerDeck, setDisplayPlayerDeck] = useState(false);
  const [dices, setDices] = useState(Array(5).fill(false));
  const [displayRollButton, setDisplayRollButton] = useState(false);
  const [rollsCounter, setRollsCounter] = useState(0);
  const [rollsMaximum, setRollsMaximum] = useState(3);

  useEffect(() => {

    socket.on("game.deck.view-state", (data) => {
      setDisplayPlayerDeck(data['displayPlayerDeck']);
      if (data['displayPlayerDeck']) {
        setDisplayRollButton(data['displayRollButton']);
        setRollsCounter(data['rollsCounter']);
        setRollsMaximum(data['rollsMaximum']);
        setDices(data['dices']);
      }
    });
  }, []);

  const toggleDiceLock = (index) => {
    const newDices = [...dices];
    if (newDices[index].value !== '' && displayRollButton) {
      socket.emit("game.dices.lock", newDices[index].id);
    }
  };

  const rollDices = () => {
    if (rollsCounter <= rollsMaximum) {
      socket.emit("game.dices.roll");
    }
  };

  return (

    <View style={styles.deckPlayerContainer}>

      {displayPlayerDeck && (

        <>
          {displayRollButton && (

            <>
              <View style={styles.rollInfoContainer}>
                <Text style={styles.rollInfoText}>
                  Lancer {rollsCounter} / {rollsMaximum}
                </Text>
              </View>
            </>

          )}

          <View style={styles.diceContainer}>
            {dices.map((diceData, index) => (
              <Dice
                key={diceData.id}
                index={index}
                locked={diceData.locked}
                value={diceData.value}
                onPress={toggleDiceLock}
              />
            ))}
          </View>
```

```
          {displayRollButton && (

            <>
              <TouchableOpacity style={styles.rollButton} onPress={rollDices}>
                <Text style={styles.rollButtonText}>Roll</Text>
              </TouchableOpacity>
            </>

          )}
        </>
      )}

    </View>
  );
};

const styles = StyleSheet.create({
  deckPlayerContainer: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    borderBottomWidth: 1,
    borderColor: "black"
  },
  rollInfoContainer: {
    marginBottom: 10,
  },
  rollInfoText: {
    fontSize: 14,
    fontStyle: "italic",
  },
  diceContainer: {
    flexDirection: "row",
    width: "70%",
    justifyContent: "space-between",
    marginBottom: 10,
  },
  rollButton: {
    width: "30%",
    backgroundColor: "green",
    paddingVertical: 10,
    borderRadius: 5,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "black"
  },
  rollButtonText: {
    fontSize: 18,
    color: "white",
    fontWeight: "bold",
  },
});

export default PlayerDeck;
```

- `OpponentDeck` :
  - useState :
  - `displayOpponentDeck` : On affiche ou non le deck (le serveur renvoie false si c'est le tour du joueur et true si c'est le tour de l'adversaire).
  - `dices` : Objet contenant la valeur des dés et leur état (locked).
  - socket.on :
  - `game.deck.view-state` : Met à jour tous ses 'useState'.

```
// app/components/board/decks/opponent-deck.component.js

import React, { useState, useContext, useEffect } from "react";
import { View, Text, StyleSheet } from "react-native";
import { SocketContext } from "../../../contexts/socket.context";
import Dice from "./dice.component";
```

```
const OpponentDeck = () => {
  const socket = useContext(SocketContext);
  const [displayOpponentDeck, setDisplayOpponentDeck] = useState(false);
  const [opponentDices, setOpponentDices] = useState(Array(5).fill({ value: "", locked: false }));

  useEffect(() => {
    socket.on("game.deck.view-state", (data) => {
      setDisplayOpponentDeck(data['displayOpponentDeck']);
      if (data['displayOpponentDeck']) {
        setOpponentDices(data['dices']);
      }
    });
  }, []);

  return (
    <View style={styles.deckOpponentContainer}>
      {displayOpponentDeck && (
        <View style={styles.diceContainer}>
          {opponentDices.map((diceData, index) => (
            <Dice
              key={index}
              locked={diceData.locked}
              value={diceData.value}
              opponent={true}
            />
          ))}
        </View>
      )}
    </View>
  );
};

const styles = StyleSheet.create({
  deckOpponentContainer: {
    flex: 1,
    justifyContent: "center",
    alignItems: "center",
    borderBottomWidth: 1,
    borderColor: "black"
  },
  diceContainer: {
    flexDirection: "row",
    width: "70%",
    justifyContent: "space-between",
    marginBottom: 10,
  },
});

export default OpponentDeck;
```

- `Dice`
  - Method `handlePress` : Remonte le call back du paramètre `onPress` du composant.

```
// app/components/board/decks/dice.component.js

import React from "react";
import { View, Text, TouchableOpacity, StyleSheet } from "react-native";

const Dice = ({ index, locked, value, onPress, opponent }) => {

  const handlePress = () => {
    if (!opponent) {
      onPress(index);
    }
  };

  return (
    <TouchableOpacity
      style={[styles.dice, locked && styles.lockedDice]}
      onPress={handlePress}
```

```
      disabled={opponent}
    >
      <Text style={styles.diceText}>{value}</Text>
    </TouchableOpacity>
  );
};

const styles = StyleSheet.create({
  dice: {
    width: 40,
    height: 40,
    backgroundColor: "lightblue",
    borderRadius: 5,
    justifyContent: "center",
    alignItems: "center",
  },
  lockedDice: {
    backgroundColor: "gray",
  },
  diceText: {
    fontSize: 20,
    fontWeight: "bold",
  },
  opponentText: {
    fontSize: 12,
    color: "red",
  },
});

export default Dice;
```

11.2 Partie WebSocket Server

Nouvelles constantes et méthodes du GameService

```
// websocket-server/services/game.service.js

// Durée d'un tour en secondes
const TURN_DURATION = 30;

const DECK_INIT = {
    dices: [
        { id: 1, value: '', locked: true },
        { id: 2, value: '', locked: true },
        { id: 3, value: '', locked: true },
        { id: 4, value: '', locked: true },
        { id: 5, value: '', locked: true },
    ],
    rollsCounter: 1,
    rollsMaximum: 3
};

const GAME_INIT = {
    gameState: {
        currentTurn: 'player:1',
        timer: null,
        player1Score: 0,
        player2Score: 0,
        grid: [],
        choices: {},
        deck: {}
    }
}

const GameService = {

    init: {
        gameState: () => {
            const game = { ...GAME_INIT };
            game['gameState']['timer'] = TURN_DURATION;
```

```
            game['gameState']['deck'] = { ...DECK_INIT };
            return game;
        },

        deck: () => {
            return { ...DECK_INIT };
        },
    },

    send: {
        forPlayer: {

            // ...

            deckViewState: (playerKey, gameState) => {
                const deckViewState = {
                    displayPlayerDeck: gameState.currentTurn === playerKey,
                    displayOpponentDeck: gameState.currentTurn !== playerKey,
                    displayRollButton: gameState.deck.rollsCounter <= gameState.deck.rollsMaximum,
                    rollsCounter: gameState.deck.rollsCounter,
                    rollsMaximum: gameState.deck.rollsMaximum,
                    dices: gameState.deck.dices
                };
                return deckViewState;
            }
        }
    },

    timer: {
        getTurnDuration: () => {
            return TURN_DURATION;
        }
    },

    dices: {
        roll: (dicesToRoll) => {
            const rolledDices = dicesToRoll.map(dice => {
                if (dice.value === "") {
                    // Si la valeur du dé est vide, alors on le lance en mettant le flag locked à false
                    const newValue = String(Math.floor(Math.random() * 6) + 1);
                    return {
                        id: dice.id,
                        value: newValue,
                        locked: false
                    };
                } else if (!dice.locked) {
                    // Si le dé n'est pas verrouillé et possède déjà une valeur, alors on le relance
                    const newValue = String(Math.floor(Math.random() * 6) + 1);
                    return {
                        ...dice,
                        value: newValue
                    };
                } else {
                    // Si le dé est verrouillé ou a déjà une valeur mais le flag locked est true, on le laiss
                    return dice;
                }
            });
            return rolledDices;
        },

        lockEveryDice: (dicesToLock) => {
            const lockedDices = dicesToLock.map(dice => ({
                ...dice,
                locked: true
            }));
            return lockedDices;
        }
    },

    utils: {
        // Return game index in global games array by id
        findGameIndexById: (games, idGame) => {
            for (let i = 0; i < games.length; i++) {
                if (games[i].idGame === idGame) {
```

```
                    return i; // Retourne l'index du jeu si le socket est trouvé
                }
            }
            return -1;
        },

        findGameIndexBySocketId: (games, socketId) => {
            for (let i = 0; i < games.length; i++) {
                if (games[i].player1Socket.id === socketId || games[i].player2Socket.id === socketId) {
                    return i; // Retourne l'index du jeu si le socket est trouvé
                }
            }
            return -1;
        },

        findDiceIndexByDiceId: (dices, idDice) => {
            for (let i = 0; i < dices.length; i++) {
                if (dices[i].id === idDice) {
                    return i; // Retourne l'index du jeu si le socket est trouvé
                }
            }
            return -1;
        }
    }
}

module.exports = GameService;
```

Exercice: Implémentation de la logique de jeu

Finaliser les fonctionnalités autour des decks au sein du 'index.js':

- Creation de partie (`createGame`)
    - Après initialisation du `gameState` , on envoie les données nécessaires aux deux composants de decks: `GameService.send.forPlayer.deckViewState(playerKey, gameState)` .
    - Chaque seconde au sein du `setInterval()` du timer, nous avons une condition de fin de tour (si timer égal 0), à ce moment là les dés sont réinitialisés: `GameService.init.deck();` , puis les deux clients sont notifiés du changement d'état: `GameService.send.forPlayer.deckViewState(playerKey, gameState)` .
- Lorsque que le `socket.on('game.dices.roll')` est appelé:
    - Pour les deux premiers lancés (condition calculée avec `rollsCounter` et `rollsMaximum`):
    - on roll seulement les dés non-lockés (`GameService.dices.roll(dicesToRoll)`).
    - on incrémente le counter `rollsCounter` .
    - on met à jour la vue des clients en envoyant le socket nécessaires aux vues.
    - Pour le dernier jet de dés:
    - on roll seulement les dés non-lockés (`GameService.dices.roll(dicesToRoll)`).
    - on incrémente le counter `rollsCounter` .
    - on lock tous les dés `GameService.dices.lockEveryDice(dicesToLock)`
    - (temporaire) on met le timer à 5 secondes, pour que le changement de tour s'opère plus vite et que le player ait le temps de voir les dés de son dernier lancer.
    - on met à jour la vue des clients en envoyant le socket nécessaires aux vues.

- Lorsque que le `socket.on('game.dices.lock', (idDice))` est appelé:
    - On inverse l'attribut `games[gameIndex].gameState.deck.dices[indexDice].locked` du dé lié à l'id passé en paramètre du socket.
    - On trouve la partie: `GameService.utils.findGameIndexBySocketId()`
    - On trouve l'index du dé `GameService.utils.findDiceIndexByDiceId()`
    - on met à jour la vue des clients en envoyant le socket nécessaires aux vues.