l'école d'ingénierie
informatique

EPSI BORDEAUX - I1 EISI

Module Atelier - Architecture applicative (3/9)

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

5. Notions de routing mobile React Native

Création des interfaces

Les entités 'screen' côté Client sont responsables de vérifier la connexion au WebSocket Server, tandis que les actions de jeux et le périmètre purement de l'interface graphique sera à terme géré dans des sous composants.

```
// app/screens/online-game.screen.js

import React, { useContext } from "react";
import { StyleSheet, View, Button, Text } from "react-native";
import { SocketContext } from '../contexts/socket.context';

export default function OnlineGameScreen({ navigation }) {

    const socket = useContext(SocketContext);

    return (
        <View style={styles.container}>
            {!socket && (
                <>
                    <Text style={styles.paragraph}>
                        No connection with server...
                    </Text>
                    <Text style={styles.footnote}>
                        Restart the app and wait for the server to be back again.
                    </Text>
                </>
            )}

            {socket && (
                <>
                    <Text style={styles.paragraph}>
                        Online Game Interface
                    </Text>
                    <Text style={styles.footnote}>
                        My socket id is: {socket.id}
                    </Text>
                    <Button
                        title="Revenir au menu"
                        onPress={() => navigation.navigate('HomeScreen')}
```

```
                    />
                </>
            )}
        </View>
    );
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: "#fff",
        alignItems: "center",
        justifyContent: "center",
    }
});
```

```
// app/screens/vs-bot-game.screen.js

import React, { useContext } from "react";
import { StyleSheet, View, Button, Text } from "react-native";
import { SocketContext } from '../contexts/socket.context';

export default function VsBotGameScreen({ navigation }) {

    const socket = useContext(SocketContext);

    return (
        <View style={styles.container}>
            {!socket && (
                <>
                    <Text style={styles.paragraph}>
                        No connection with server...
                    </Text>
                    <Text style={styles.footnote}>
                        Restart the app and wait for the server to be back again.
                    </Text>
                </>
            )}

            {socket && (
                <>
                    <Text style={styles.paragraph}>
                        VsBot Game Interface
                    </Text>
                    <Text style={styles.footnote}>
                        My socket id is: {socket.id}
                    </Text>
                    <Button
                        title="Revenir au menu"
                        onPress={() => navigation.navigate('HomeScreen')}
                    />
                </>
            )}
        </View>
    );
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: "#fff",
        alignItems: "center",
        justifyContent: "center",
    }
});
```

Modification du HomeScreen

```
// app/screens/home.screen.js

import { StyleSheet, View, Button } from "react-native";

export default function HomeScreen({ navigation }) {

  return (
    <View style={styles.container}>
      <View>
        <Button
          title="Jouer en ligne"
          onPress={() => navigation.navigate('OnlineGameScreen')}
        />
      </View>
      <View>
        <Button
          title="Jouer contre le bot"
          onPress={() => navigation.navigate('VsBotGameScreen')}
        />
      </View>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: "#fff",
    alignItems: "center",
    justifyContent: "center",
  }
});
```

Intégration des routes

```
// App.js

// ...
import HomeScreen from './app/screens/home.screen';
import OnlineGameScreen from './app/screens/online-game.screen';
import VsBotGameScreen from './app/screens/vs-bot-game.screen';

// ...

function App() {
  return (
    <SocketContext.Provider value={socket}>
      <NavigationContainer>
        <Stack.Navigator initialRouteName="HomeScreen">
          <Stack.Screen name="HomeScreen" component={HomeScreen} />
          <Stack.Screen name="OnlineGameScreen" component={OnlineGameScreen} />
          <Stack.Screen name="VsBotGameScreen" component={VsBotGameScreen} />
        </Stack.Navigator>
      </NavigationContainer>
    </SocketContext.Provider>
  );
}

export default App;
```

6. Mise en place du Matchmaking

Commençons par mettre en place la file d'attente et la mise en relation de deux clients autour d'une partie.

Templating conditionnel

Comme vu en cours, nous séparons volontairement le composant `OnlineGameScreen` du composant `OnlineGameController` afin de répondre à l'architecture voulue et anticiper une évolution de l'application sereinement.

- Intégration du contrôleur si la connexion est établie.

```
// app/screens/online-game.screen.js

// ...
import OnlineGameController from "../controllers/online-game.controller";

// ...

export default function OnlineGameScreen({ navigation }) {

  // ...

  return (
      <View style={styles.container}>
          {!socket && (
              // ...
          )}

          {socket && (
              <OnlineGameController />
          )}
      </View>
  );
}
```

Emission de sockets côté Client

- Le contrôleur, dans un premier temps, notifie le serveur WebSocket (emit on `queue.join`) qu'il souhaite rejoindre une partie en ligne lorsque le `OnlineGameController` s'instancie.

```
// app/controller/online-game.controller.js

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


export default function OnlineGameController() {

    const socket = useContext(SocketContext);

    const [inQueue, setInQueue] = useState(false);
    const [inGame, setInGame] = useState(false);
    const [idOpponent, setIdOpponent] = useState(null);

    useEffect(() => {
        console.log('[emit][queue.join]:', socket.id);
        socket.emit("queue.join");
        setInQueue(false);
        setInGame(false);
    }, []);

    return (
        <View style={styles.container}>
            {!inQueue && !inGame && (
                <>
                    <Text style={styles.paragraph}>
                        Waiting for server datas...
                    </Text>
                </>
            )}

            {inQueue && (
                <>
```

```
                    <Text style={styles.paragraph}>
                        Waiting for another player...
                    </Text>
                </>
            )}

            {inGame && (
                <>
                    <Text style={styles.paragraph}>
                        Game found !
                    </Text>
                    <Text style={styles.paragraph}>
                        Player {socket.id} vs {idOpponent}
                    </Text>
                </>


            )}
        </View>
    );
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: "#fff",
        alignItems: "center",
        justifyContent: "center",
        width: '100%',
        height: '100%',
    },
    paragraph: {
        fontSize: 16,
    }
});
```

Réception et émission du Socket côté WebSocket Server

Le WebSocket Server est en charge d'animer les parties.

- Nous aurons besoin de deux variables globales `games[]` et `queue[]` pour stocker et accéder aux données de jeu en temps réel.
- Le serveur écoute sur `queue.join` et manipule les objets `games[]` et `queue[]` en fonction de la tâche à réaliser.
    - si la taille de la file est plus petite que 2 alors on enlève les deux joueurs de la file et on les ajoute à une partie dans `games[]`. Le serveur émet alors un socket sur `game.start`.
    - sinon, si la file d'attente est vide, le socket du joueur est ajouté à `queue[]` et le serveur émet un socket sur `queue.added`

Dépendances (à la racine du dossier du serveur):

```
npm install --save uniqid
```

- Mise à jour du 'index.html' côté WebSocket Server

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

const app = require('express')();
const http = require('http').Server(app);
const io = require('socket.io')(http);
var uniqid = require('uniqid');
const GameService = require('./services/game.service');
```

```
// ------------------------------------------------
// -------- CONSTANTS AND GLOBAL VARIABLES --------
// ------------------------------------------------
let games = [];
let queue = [];

// ----------------------------------
// -------- GAME METHODS ----------
// ----------------------------------

const newPlayerInQueue = (socket) => {

  queue.push(socket);

  // Queue management
  if (queue.length >= 2) {
    const player1Socket = queue.shift();
    const player2Socket = queue.shift();
    createGame(player1Socket, player2Socket);
  }
  else {
    socket.emit('queue.added', GameService.send.forPlayer.viewQueueState());
  }
};

const createGame = (player1Socket, player2Socket) => {

  const newGame = GameService.init.gameState();
  newGame['idGame'] = uniqid();
  newGame['player1Socket'] = player1Socket;
  newGame['player2Socket'] = player2Socket;

  games.push(newGame);

  const gameIndex = GameService.utils.findGameIndexById(games, newGame.idGame);

  games[gameIndex].player1Socket.emit('game.start', GameService.send.forPlayer.viewGameState('player:1', game
  games[gameIndex].player2Socket.emit('game.start', GameService.send.forPlayer.viewGameState('player:2', game
};

// ---------------------------------------
// -------- SOCKETS MANAGEMENT ----------
// ---------------------------------------

io.on('connection', socket => {
  console.log(`[${socket.id}] socket connected`);

  socket.on('queue.join', () => {
    console.log(`[${socket.id}] new player in queue `)
    newPlayerInQueue(socket);
  });

  socket.on('disconnect', reason => {
    console.log(`[${socket.id}] socket disconnected - ${reason}`);
  });
});

// ----------------------------------
// -------- SERVER METHODS ----------
// ----------------------------------

app.get('/', (req, res) => res.sendFile('index.html'));

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

- Implémentation d'un service sous la forme d'un objet organisé de fonctions utiles à plusieurs aspects du moteur de jeu (`init.gameState()` , `utils.findGameIndexById()` , `send.forPlayer.viewGameState()`).

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

const GAME_INIT = {
    gameState: {
        currentTurn: 'player:1',
        timer: 60,
        player1Score: 0,
        player2Score: 0,
        grid: [],
        choices: {},
        deck: {}
    }
}

const GameService = {

    init: {
        // Init first level of structure of 'gameState' object
        gameState: () => {
            return GAME_INIT;
        },
    },
    send: {
        forPlayer: {
            // Return conditionnaly gameState custom objet for player views
            viewGameState: (playerKey, game) => {
                return {
                    inQueue: false,
                    inGame: true,
                    idPlayer:
                        (playerKey === 'player:1')
                            ? game.player1Socket.id
                            : game.player2Socket.id,
                    idOpponent:
                        (playerKey === 'player:1')
                            ? game.player2Socket.id
                            : game.player1Socket.id
                };
            },

            viewQueueState: () => {
                return {
                    inQueue: true,
                    inGame: false,
                };
            }
        }
    },
    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;
        },
    }
}

module.exports = GameService;
```

Réception du socket côté Client

Le `OnlineGameController` doit donc écouter les sockets `queue.added` et `game.start` puis mettre à jour l'état de ses `useState()` React.

```
// app/controller/online-game.controller.js

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


export default function OnlineGameController() {

    const socket = useContext(SocketContext);

    const [inQueue, setInQueue] = useState(false);
    const [inGame, setInGame] = useState(false);
    const [idOpponent, setIdOpponent] = useState(null);

    useEffect(() => {
        console.log('[emit][queue.join]:', socket.id);
        socket.emit("queue.join");
        setInQueue(false);
        setInGame(false);

        socket.on('queue.added', (data) => {
            console.log('[listen][queue.added]:', data);
            setInQueue(data['inQueue']);
            setInGame(data['inGame']);
        });

        socket.on('game.start', (data) => {
            console.log('[listen][game.start]:', data);
            setInQueue(data['inQueue']);
            setInGame(data['inGame']);
            setIdOpponent(data['idOpponent']);
        });

    }, []);

    return (
        <View style={styles.container}>
            {!inQueue && !inGame && (
                <>
                    <Text style={styles.paragraph}>
                        Waiting for server datas...
                    </Text>
                </>
            )}

            {inQueue && (
                <>
                    <Text style={styles.paragraph}>
                        Waiting for another player...
                    </Text>
                </>
            )}

            {inGame && (
                <>
                    <Text style={styles.paragraph}>
                        Game found !
                    </Text>
                    <Text style={styles.paragraph}>
                        Player - {socket.id} -
                    </Text>
                    <Text style={styles.paragraph}>
                        - vs -
                    </Text>
                    <Text style={styles.paragraph}>
                        Player - {idOpponent} -
                    </Text>
                </>
            )}
        </View>
    );
}
```

```
const styles = StyleSheet.create({
    container: {
        flex: 1,
        backgroundColor: "#fff",
        alignItems: "center",
        justifyContent: "center",
        width: '100%',
        height: '100%',
    },
    paragraph: {
        fontSize: 16,
    }
});
```

7. Exercice - Manipulation de sockets Client / Server

Gérer le cas où un utilisateur quitte la file ou quitte l'écran d'accueil:

- Le serveur doit effectuer une modification de ses variables globales `games[]` et `queue[]` lorsqu'un socket Client est déconnecté.
- Lorsque l'utilisateur navigue sur l'application, il doit pouvoir retrouver son état de partie/queue en cours.
- Lorsqu'un joueur se déconnecte, l'adversaire doit être informé et une action peut être faite ("en attente de l'adversaire", "voulez-vous rejoindre une nouvelle partie", etc…)