САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: “Web программирование”

Отчет

Лабораторная работа №1

Выполнил:

Митурский Богдан

K33392

Санкт-Петербург

2023 г.

Цель:

Овладеть практическими навыками и умениями реализации web-серверов и использования сокетов.

Программное обеспечение: TypeScript, Node Typescript, MongoDB, React, socket.io

План работы:

В рамках работы реализуем механику подключения пользователей к игровому лобби для сражений 1 на 1, с возможностью подключения как игрока, так и наблюдателя, а также реализуем отправку состояния игрового поля на момент подключения для обоих игроков и возможность создавать юнитов находясь не в режиме наблюдателя на примере моей многопользовательской онлайн игры (https://vk.com/sheeproyale)

Задание 1

Развернем сервер socket.io на Node TS с применением авторизации с помощью секретного ключа мини-приложения ВКонтакте.

Сервер, Index.ts:

```
import express from "express";

import cookieParser from "cookie-parser";

import cors from "cors";
import http from "http";
import { Server } from "socket.io";
import { checkSignMiddleware } from "./src/middlewares/checkSign";
import registerSocketHandler from "./src/routes";
import { mongooseConnect } from "@src/utils/mongooseConnect";
import { createAdapter } from "@socket.io/redis-adapter";

import { Redis } from "@/src/classes";

const app = express();

// Используем express для дальнейшего создания socket io сервера
app.use(express.json());
app.use(
  cors({
    origin: "*",
    credentials: true,
  })
);
app.use(cookieParser());

// Даём возможность установить порт сервера из env
const PORT = Number(process.env.PORT) || 6000;

// Выносим доступ к ключевому io для других файлов
export let io: Server;

// Асинхронная функция запуска необходимых баз данных и socket io сервера
async function start() {
  try {
    // Подключаемся к бд redis
    const redis = new Redis();
    const subClient = redis._redis.duplicate();
    await Promise.all([redis.init(), subClient.connect()]);

    // Подключаемся к mongoDB
    await mongooseConnect();
```

```
    // Создаем http socket.io сервер
    let server = http.createServer(app);
    let options = {
      cors: {
        origin: "*",
        methods: ["GET", "POST"],
        credentials: true,
      },
    } as {};

    // Запускаем socket.io сервер
    io = new Server(server, options);
    io.adapter(createAdapter(redis._redis, subClient));

    // Назначаем первым роутом по умолчанию авторизацию
    io.use(checkSignMiddleware);

    // Назначаем различные socket.on каждому подключившемуся сокеты прошедшему
авторизацию
    io.on("connection", (socket) => {
      registerSocketHandler(io, socket);
    });

    // Запуск прослушивание порта и выводим логи
    server.listen(PORT, () => {
      console.log(`[Lobby] has been started on port ${PORT}`);
    });
  } catch (e: any) {
    console.log("Lobby Error", e.message);
    process.exit(1);
  }
}

start();
```

Сервер, middlewares/checkSign.ts:

```
import { ExtendedError } from "socket.io/dist/namespace";
import { Socket } from "socket.io";
import { checkSignWebSocket } from "@src/utils/checkSignWebSocket";

type NextFunction = (err?: ExtendedError) => void;

export const checkSignMiddleware = (req: Socket, next?: NextFunction) => {
  try {
    if (!req.handshake.auth?.params) return;
```

```
    // Проверяем наличие необходимых частей ключа для проверки подлинности
подписи
    if (
      !checkSignWebSocket(decodeURIComponent(req.handshake.auth?.params), [
        process.env.VK_SECRET_KEY || "",
        process.env.VK_SECRET_GAME_KEY || "",
      ])
    ) {
      return;
    }

    // Парсим id пользователя в зависимости от его точки входа в приложение
    req.data.userID = Number(
      req.handshake.auth?.params?.split("vk_user_id=")[1]?.split("&")[0] ||
        req.handshake.auth?.params?.split("viewer_id=")[1]?.split("&")[0]
    );

    // Парсим id игрового лобби, которое пришел посмотреть пользователь (если
он зашел в режиме наблюдателя)
    req.data.watchID = req.handshake.auth?.watchId;

    if (next) {
      next();
    }
  } catch (err) {
    return;
  }
};
```

Сервер, Routes/index.ts:

```
import { ServerType, SocketType } from "@src/types/socketData";

import connectRoute from "./connect";
import disconnectRoute from "./disconnect";
import spawnUnit from "./spawnUnit";
import checkEqual from "./checkEqual";
import lobbyJoin from "./lobbyJoin";
import extraSync from "./extraSync";
import updateOnline from "./updateOnline";

// Регистрируем всевозможные socket события в одном файле
export default async (io: ServerType, socket: SocketType) => {
  connectRoute(io, socket);
  spawnUnit(io, socket);
  disconnectRoute(io, socket);
  lobbyJoin(io, socket);
  if (process.env.CHECK_EGINE_EQUAL === "true") {
```

```
    checkEqual(io, socket);
  }
  extraSync(io, socket);
  updateOnline(io, socket);
};
```

Задание 2:

Реализуем подключение пользователя к конкретной игре при его успешной авторизации и при условии наличия игры в базе данных. Для реализации будем использовать комнаты из socket.io и такие методы как socket.join.

По хранящемуся в бд id игры будем создавать комнату и подключать туда игроков с помощью socket.join(“game”+gameId), далее, все запросы к комнате будем отправлять с помощью io.to(“game”+gameId)

Сервер, Routes/connect.ts:

```
import { SocketDataType } from "@src/types/socketData";

import { Server, Socket } from "socket.io";
import { findUser } from "@src/utils/findUser";
import { Games } from "../controllers";
import { CardsData, CardUnits } from "@src/data/cards";
import { ObjectID } from "mongodb";
import { BEFORE_GAME_DURATION, DURATION_OF_GAME } from "@/../engine/config";
import { ICard } from "@/src/models/Users";

const isUnitType = (str: string): str is CardUnits => {
  return str in CardUnits;
};

export default async (_: Server, socket: Socket) => {

  /* Достаем информацию о пользователе полученную при прохождении авторизации,
   а также как часть запроса при подключении **/
  const socketData = socket.data as SocketDataType;
  const userId = socketData.userID;
  const watchID = socketData.watchID;

  // Находим данные пользователя в mongoDB
  const user = await findUser(userId);

  if (user) {
    try {
      // Проверяем существует ли игра к которой хочет подключиться
пользователь
      const gameID =
        user?.lobby?._id.toString() || String(new ObjectID(watchID));
      const game = await Games.findOne({
        _id: gameID,
      });

      /* Если игры не существует, отдаем сообщение о завершении игры, чтобы
```

```
      не вызвать визуальных багов на фронтенде **/
      if (!game) {
        if (watchID) {
          socket.emit("gameFinished", undefined, undefined);
        }
        else {
          socket.emit("gameFinished", user.history[0].userResponse,
user.history[0].userInfo);
        }
        return;
      };

      // Находим в базе данных те карты пользователя, которые он выбрал для
текущей игры
      let userCards = user.cards
        .filter((item) => item.position)
        .sort(function (a, b) {
          if (!a.position || !b.position) return 0;
          return a.position - b.position;
        })
        .map((item) => {
          if (isUnitType(item.entity))
            return {
              id: CardsData[item.entity].id,
              price: CardsData[item.entity].price,
              unitClass: CardsData[item.entity].unitClass,
              unitType: CardsData[item.entity].unitType,
            };
        });

      // Заносим в socket информацию об игре и колоде для удобного
взаимодействия с ними дальше по коду
      socket.data.gameID = gameID;
      socket.data.deck = userCards;

      // Подключаем пользователя к нужной нам игре
      socket.join("game" + gameID);

      // Вызываем обновление игрового движка
      game.update();

      // Отправляе пользователю данные об игре, необходимые для её корректной
отрисовки
      socket.emit(
        "setGame",
        game.getInitialState(),
        userCards,
        gameID,
```

```
        game.gameCompletionTime - DURATION_OF_GAME + BEFORE_GAME_DURATION,
        Date.now()
      );
    } catch (err) {
      console.log(err);
    }
  }
};
```

Задание 3:

Реализуем подключение к серверу на клиенте, а также повесим обработчики на ключевые события игры.

Клиент:

```
/* Подключаемся по полученному с сервера lobby url, передавая параметры
авторизации и watchId,
     если хотим зайти в режим наблюдателя **/
    const socket = io(lobby?.url || meta?.url, {
      transports: ["websocket"],
      auth: { params: getAuthParams(), watchId: meta?.watchId },
    });

    // Добавляем логирование для отслеживание состояния подключения
    socket.on("reconnect", () => {
      console.log("reconnect");
    });

    socket.on("connect", () => {
      console.log("connect");
    });

      // Подписываем на событие завершения игры
    socket.on("gameFinished", (data, userUpdated) => {

      // Записываем в redux обновленные данные пользователя
      if (userUpdated) {
        dispatch(setUser(userUpdated));
      }
      // Разблокируем навигацию
      unblock();

      // Воспроизводим звук завршения игры
      gameEndSound.play();
      gameEndSound.on("end", () => {
        backgroundSound?.current?.play();
      });

      // Открываем страницу завершения игры, передавая данные с результатами
      replace("/game?blackScreenPopout=endGame", { ...data });
    });

      // Подписываемся на событие инициализации игры
    socket.on("setGame", (data, deck, gameID, gameStartedAt, timeNow) => {
      console.log("setGame");
```

```
      /* ...Код для работы с визуальной составляющей (отрисовки игры)... */

      // Подписка на события внутриигрового цикла с их передачей внутрь движка
игры
      socket.on("actions", (data: actionType[]) => {
        AppEngine?.insertActions(data);
      });

      /** Калибруем игровой движок в зависимости от пинга */
      socket.on("ping", (time) => {
        if (!AppEngine) return;
        AppEngine.setTimeNow(time);
      });
    });
```

Задание 4:

Реализуем функционал для взаимодействия с игровым полем. Дадим возможность игрокам отправлять запрос для создания игрового юнита и будем передавать информацию с подтверждением для всех подключенных к лобби игроков.

Сервер, Routes/spawnUnit.ts:

```
import { SocketDataType } from "@src/types/socketData";
import { Games } from "../controllers";
import { Server, Socket } from "socket.io";
import { UnitsSpawnFunctions } from "@engine/units";
import { Lobbies } from "@/src/models/Lobbies";
import { BEFORE_GAME_DURATION, DURATION_OF_GAME } from "@/../engine/config";

export default (_: Server, socket: Socket) => {
  const socketData = socket.data as SocketDataType;
  let isGameStarted = false;

  // Обрабатываем запрос spawnUnit
  socket.on(
    "spawnUnit",
    async (
      unitType: keyof typeof UnitsSpawnFunctions,
      coords?: { x?: number; y?: number }
    ) => {

      // Выполняем базовые проверки на существование игрового лобби и находим
его в базе
      const userId = socketData.userID;
      const gameID = socketData.gameID;

      const lobby = await Lobbies.findOne({ _id: gameID });
      if (!lobby) return;

      if (!userId || !gameID) return;

      const gameManager = await Games.findOne({ _id: gameID });
      if (!gameManager) return;

      const gameEndedAt = gameManager.gameCompletionTime;
      if (
        DURATION_OF_GAME - (gameEndedAt - Date.now()) > BEFORE_GAME_DURATION
||
        process.env.NODE_ENV === "development"
      ) {
        isGameStarted = true;
```

```
      }
      if (process.env.INFINITY_GAME === "true") {
        isGameStarted = true;
      }

      if (isGameStarted) {
        // Передаем игровому менеджеру задачу на создание юнита
        gameManager.spawnUnit({ unitType, coords, data: socketData, socket });
      }
    }
  );
};
```

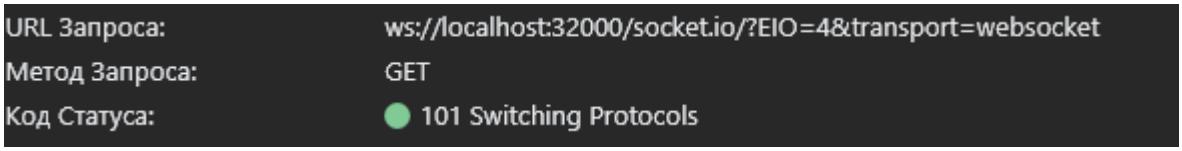
Сервер, controllers/GameManager/index.ts:

```
spawnUnit({
    unitType,
    coords,
    data,
    socket,
  }: {
    socket: Socket;
    unitType: keyof typeof UnitsSpawnFunctions;
    coords?: { x?: number; y?: number };
    data: SocketDataType;
  }) {
    // Проверяем что пользователь не создавал юнита в последние 50мс
    if (
      this.lastSpawn[data.userID] &&
      Date.now() - this.lastSpawn[data.userID] < 50
    )
      return;

    // Проверяем наличие у пользователя карты юнита, которого он хочет создать
    const card = data.deck.find((item) => item.unitType === unitType);
    if (!card) return;

    // Обновляем игровой движок
    this.update();
    // Получаем пользователя из движка
    const user = this.getUser(data.userID);
    if (!user) return;

    // Создаем новое событие в движке
    const newActions = this.buyUnit({
      userID: data.userID,
      coords,
      unitType,
```

```
    }) as any;

    if (!newActions) return;
    this.lastSpawn[data.userID] = Date.now();

    // Отправляем всем пользователям в лобби информацию о произошедшем
действии
    io.to("game" + this._id).emit("actions", newActions);

  }
```

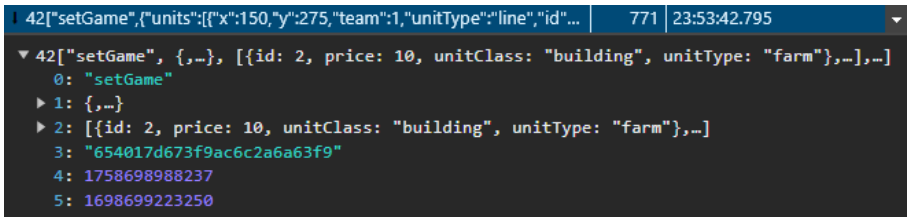
Результат:

Оба клиента подключились к lobbyUrl при входе в игру:

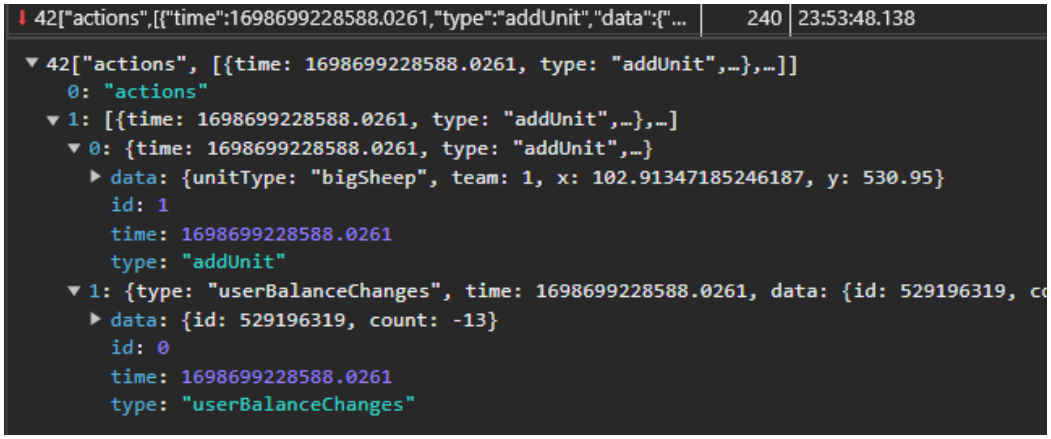


Клиент А:

- Получил информацию при подключении к лобби

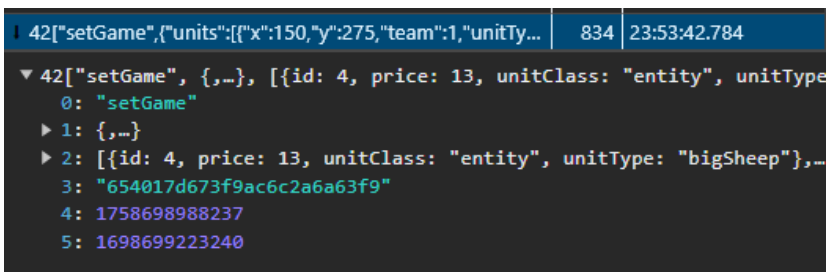


- Получил информацию о совершенном Клиент Б действии (создании игрового юнита)

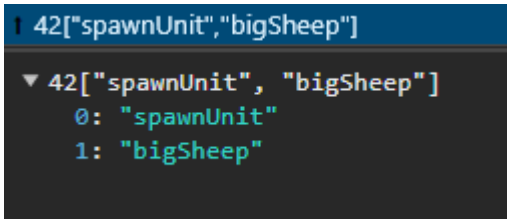


Клиент Б:

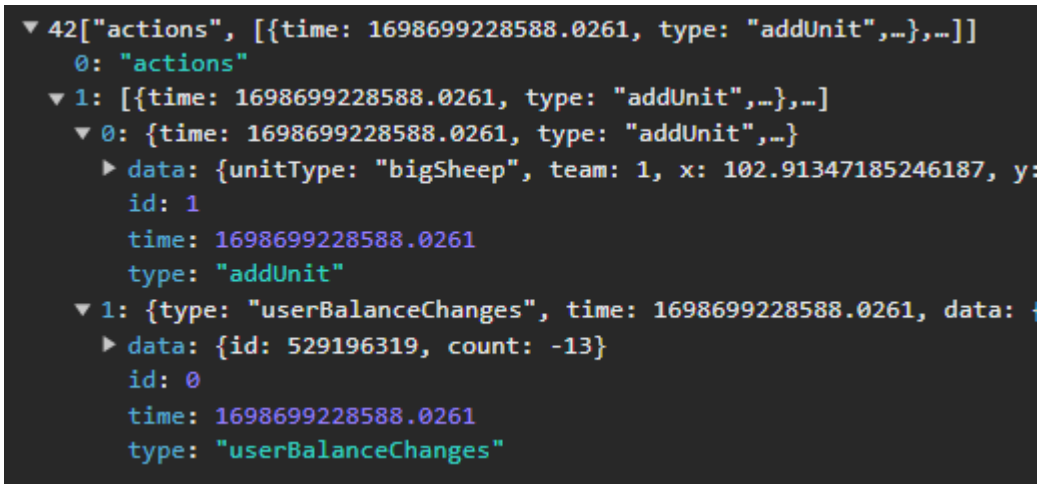
- Получил информацию при подключении к лобби



- Отправил запрос на создание игрового юнита



- Получил информацию о совершенном Клиент Б действии (создании игрового юнита)



Вывод:

В рамках реализации работы я овладел практическими навыками и умениями реализации web-серверов и использования сокетов. Сокеты, крайне удобный для реализации игр способ клиент-серверного взаимодействия, т.к. позволяет отправлять данные не прерывая соединение. Что в свою очередь сильно упрощает разработку многопользовательских игр, чатов и приложений.