САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бек-энд разработка

Отчет
Лабораторная работа №1

Выполнил: Митурский Богдан Антонович

Группа: K33392

Проверил:
Добряков Д. И.

Санкт-Петербург

2024 г.

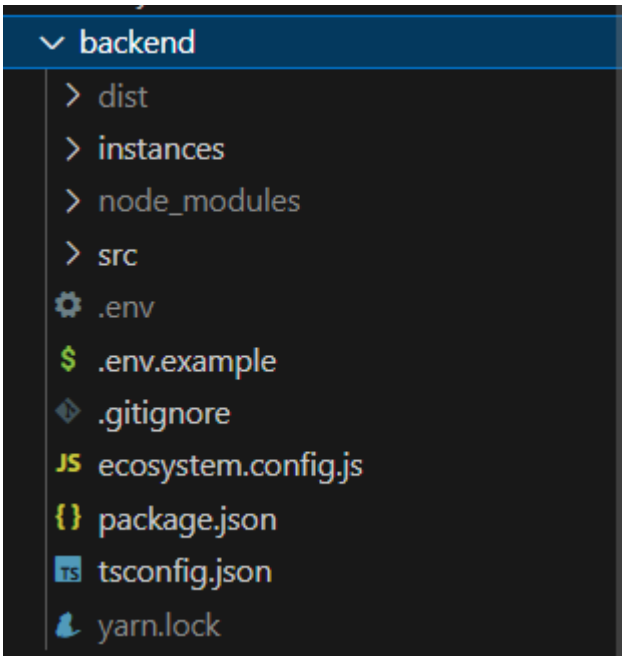
Задача
Нужно написать свой boilerplate на express + typescript.
Должно быть явное разделение на: модели; контроллеры; роуты; сервисы для работы с моделями.

Ход работы

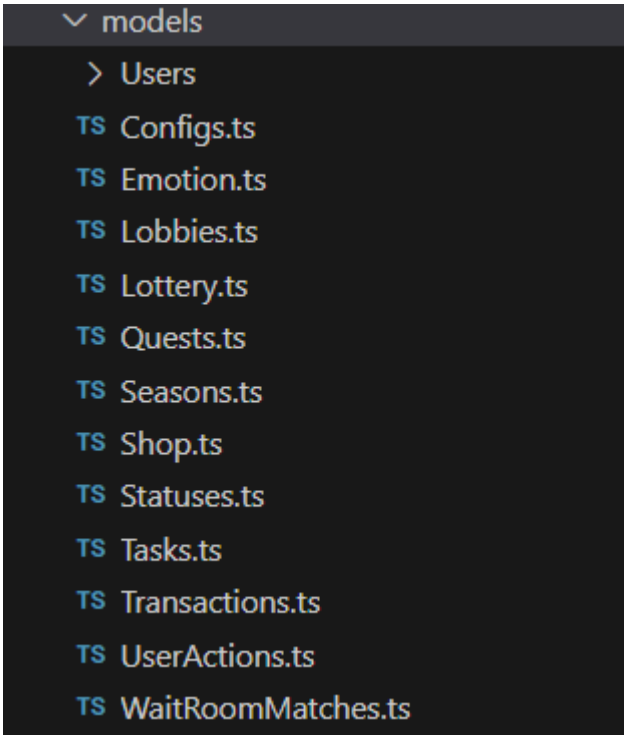
Будущий проект будет поделен на микросервисы, заложим это в структуре проекта сразу.

1. Создадим в папке backend-а:

- Instances (Тут будут храниться все микросервисы и их код)
- Src (Общая для всех микросервисов папка с полезными утилитами, типами и так далее)
- Ecosystem.config.js (Конфигурирующий файл для запуска проекта через PM2 в микросервисном режиме)



2. Работать будем с MongoDB, поэтому, начнем с создания необходмых для проекта коллекций в src/models:



Отразим в коллекциях пользователей, игровые лобби, задания, сезоны, магазин, систему квестов, транзакции и действия пользователей.

Т.к. игра будет кроссплатформенной (с поддержкой входа через ВК, ТГ и ОК), нам понадобится три различные коллекции пользователей, с единым доступом и набором параметров. Для этого, создадим основную коллекцию Users:

```
import mongoose, {
  HydratedDocument,
  ObjectId,
  PopulatedDoc,
  Schema,
  Types,
  Document,
} from "mongoose";
import { IEntity } from "../../data/cards";
import { ILobby } from "../Lobbies";
import { IChestTypes } from "../../data/chests";
import { StoreProducts } from "../../data/shop";
import { MIN_RATING } from "@src/data/rating";
import { Emotions, IEmotion, PopulatedIEmotion } from "../Emotion";
import { Platforms } from "@/src/types/platforms";
import { Lang } from "@/src/types/language";
export type Currency = "coins" | "voices" | "ok" | "ton";

export type ICard = {
  entity: IEntity;
  parts?: number;
```

```
  position?: number;
};

const CardSchema = new Schema<ICard>({
  entity: {
    type: String,
    required: true,
  },
  parts: Number,
  position: Number,
});

export type IUserEmotionCard = {
  emotion: IEmotion["_id"];
  position: number | null;
};

export type IUserWithEmotions = Document<any, unknown, IUser> &
  Omit<IUser, "emotions"> & {
    emotions: mongoose.Types.Array<
      IUserEmotionCard & { emotion: PopulatedIEmotion }
    >;
  };

type Bonuses = {
  daily: number;
  subGroup: boolean;
  joinChat: boolean;
  notificationApp: boolean;
  notificationBot: boolean;
  addToFavorites: boolean;
  addToHomeScreen: boolean;
  rewardedGift: {
    lastUpdate: number;
    adWatchedTimes: number;
  };
};

type Settings = {
  isMuted: boolean;
};

export type IChest =
  | { empty: true; _id?: ObjectId }
  | ({
      empty?: false;
      _id?: ObjectId;
      type: IChestTypes;
      adWatchedTimes: number;
```

```
    } & (
      | {
          status: "idle";
        }
      | {
          status: "opening";
          willOpenAt: number;
        }
    ));

export type IChestDocs = [
  HydratedDocument<IChest>,
  HydratedDocument<IChest>,
  HydratedDocument<IChest>,
  HydratedDocument<IChest>
];

const ChestSchema = new Schema<IChest>({
  type: String,
  status: String,
  willOpenAt: Number,
  empty: Boolean,
  adWatchedTimes: {
    type: Number,
    default: 0,
    required: true,
  },
});

export interface UserStatus {
  status: mongoose.Types.ObjectId;
  isSelected: boolean;
  isGiftReceived: boolean;
}

export interface IUser {
  markModified(arg0: string): unknown;
  _id: number;
  platform: Platforms;
  name: string;
  photo_100: string;
  photo_200: string;
  registrationDate: Date;
  ban?: boolean;
  cards: ICard[];
  emotions: IUserEmotionCard[];
  balance: number;
  language: Lang;
  history: any[];
```

```
  chests: [IChest, IChest, IChest, IChest];
  rating: number;

  stats: {
    draws: number;
    wins: number;
    losses: number;
  };
  lobby: PopulatedDoc<ILobby>;

  bonuses: Bonuses;
  isTutorialPassed: boolean;
  shop: {
    lastUpdate: number;
    boughtProducts: StoreProducts[];
    chest: {
      currency: Currency;
      chestType: IChestTypes | "";
    };

    cardFragment: {
      currency: Currency;
      card: IEntity | "";
    };
  };

  referrals: {
    refId: string;
    referralsCount: number;
  };

  statuses: UserStatus[];

  tasks: {
    current: string[];
    completed: string[];
    progress: Record<string, number>;
    nextUpdateAt: number;
    lastTasksMessageId?: number;
    madeProgress?: boolean;
  };

  settings: Settings;
}

const defaultCards: ICard[] = [
  {
    entity: "sheep",
    position: 1,
```

```
  },
  {
    entity: "farm",
    position: 2,
  },
  {
    entity: "pasture",
    position: 3,
  },
  {
    entity: "teslaTower",
    position: 4,
  },
  {
    entity: "bigSheep",
  },
  {
    entity: "bee",
  },
  {
    entity: "doubleFarm",
    parts: 1,
  },
];

const defaultEmotions: IUserEmotionCard[] = [
  { emotion: "sheepyHello", position: 1 },
  { emotion: "sheepSorry", position: 2 },
  { emotion: "sheepAngry", position: 3 },
];

export const UserSchema = new Schema<IUser>({
  _id: { index: true, required: true, type: Number },
  platform: {
    type: String,
    required: true,
    enum: Platforms,
    default: Platforms.vk,
  },
  ban: {
    type: Boolean,
    required: false,
  },
  name: {
    type: "string",
    required: true,
  },
  photo_100: {
    type: "string",
```

```
    required: true,
  },
  photo_200: {
    type: "string",
    required: true,
  },
  registrationDate: {
    type: Date,
  },
  cards: {
    type: [CardSchema],
    required: true,
    default: defaultCards,
  },
  emotions: {
    type: [
      {
        emotion: { type: String, ref: Emotions },
        position: { type: Number, default: null },
      },
    ],
    required: true,
    default: defaultEmotions,
    _id: false,
  },
  balance: {
    type: Number,
    required: true,
    default: 0,
  },
  language: {
    type: String,
    required: true,
    default: "ru",
  },
  history: {
    type: [],
    required: true,
    default: [],
  },
  stats: {
    wins: {
      type: Number,
      default: 0,
      required: true,
    },
    losses: {
      type: Number,
      default: 0,
```

```
    required: true,
  },
  draws: {
    type: Number,
    default: 0,
    required: true,
  },
},

chests: {
  type: [ChestSchema],
  default: [
    { empty: true },
    { empty: true },
    { empty: true },
    { empty: true },
  ],
  required: true,
},
rating: {
  type: Number,
  default: MIN_RATING,
  required: true,
  index: -1,
},

lobby: { type: Schema.Types.ObjectId, ref: "lobbies" },

bonuses: {
  daily: {
    type: Number,
    default: 0,
    required: true,
  },
  subGroup: {
    type: Boolean,
    default: false,
    required: true,
  },
  joinChat: {
    type: Boolean,
    default: false,
    required: true,
  },
  notificationApp: {
    type: Boolean,
    default: false,
    required: true,
  },
```

```
    notificationBot: {
      type: Boolean,
      default: false,
      required: true,
    },
    addToFavorites: {
      type: Boolean,
      default: false,
      required: true,
    },
    addToHomeScreen: {
      type: Boolean,
      default: false,
      required: true,
    },
    rewardedGift: {
      lastUpdate: {
        type: Number,
        default: Date.now(),
        required: true,
      },
      adWatchedTimes: {
        type: Number,
        default: 0,
        required: true,
      },
    },
  },
  isTutorialPassed: {
    type: Boolean,
    default: false,
    required: true,
  },
  shop: {
    lastUpdate: {
      type: Number,
      default: 0,
      required: true,
    },
    boughtProducts: {
      type: Object,
      default: [],
      required: true,
    },
    chest: {
      chestType: {
        type: String,
        default: "",
      },
```

```
    currency: {
      type: String,
      default: "coins",
    },
  },
  cardFragment: {
    card: {
      type: String,
      default: "",
    },
    currency: {
      type: String,
      default: "coins",
    },
  },
},
tasks: {
  completed: {
    type: [String],
    required: true,
    default: [],
  },
  current: {
    type: [String],
    required: true,
    default: [],
  },
  progress: {
    type: {},
    default: {},
    required: true,
  },
  nextUpdateAt: {
    type: Number,
    required: true,
    default: 0,
  },
  lastTasksMessageId: {
    type: Number,
  },
  madeProgress: { type: Boolean },
},

referrals: {
  refId: {
    type: String,
    default: "",
  },
  referralsCount: {
```

```
      type: Number,
      default: 0,
    },
  },

  statuses: [
    {
      status: {
        type: Types.ObjectId,
        required: true,
        unique: false,
        ref: "emoji_status",
      },
      isGiftReceived: { type: Boolean, default: false },
      isSelected: { type: Boolean, default: false },
    },
  ],

  settings: { isMuted: { default: false, type: Boolean, required: true } },
});

export const Users = mongoose.model("users", UserSchema);
```

Далее, создадим коллекции для не основных соц сетей (ТГ и ОК):

```
import mongoose from "mongoose";

import { UserSchema } from "./Users";

export const UsersOk = mongoose.model("usersok", UserSchema);
```

```
import mongoose from "mongoose";

import { UserSchema } from "./Users";

export const UsersTg = mongoose.model("userstg", UserSchema);
```

И, для удобства, разработаем функцию, которая будет, опираясь на платформу возвращать нужную нам коллекцию:

```
import { Platforms } from "@/src/types/platforms";
import { Users } from "../Users";
import { UsersOk } from "../UsersOk";
import { UsersTg } from "../UsersTg";

function UsersModel(platform: Platforms) {
  switch (platform) {
```

```
    case Platforms.vk:
      return Users;
    case Platforms.tg:
      return UsersTg;
    case Platforms.ok:
      return UsersOk;
  }
}
export default UsersModel;
```

Остальные коллекции заполним исходя из требований к самому проекту.

3. Создадим базовые утилиты, которые помогут в дальнейшем при написании бекенда

- findUser (Позволяет найти пользователя в бд. Создаёт его при отсутствии, также, обрабатывает базовую логику обновления необходимых данных пользователя при входе)

```
import "../models/Lobbies";
import { IUserWithEmotions, Users } from "../models/Users/Users";
import { Platforms } from "../types/platforms";
import UsersModel from "../models/Users/utils/getUsersModel";
import { newUserMessage } from "@/instances/hub/src/utils/tasks/checkTasks";
import { ONLINE_SECONDS_TIMEOUT } from "../configs/online";

export type findUserParams = IGetUserIDAuthInfo & {
  userRegistrationData?: {
    firstName: string;
    lastName?: string;
    languageCode?: string;
  };
};

export const findUser = async (req: findUserParams) => {
  try {
    let user;
    req.userId = Number(req.userId);

    user = await UsersModel(req.platform)
      .findOne({ _id: req.userId })
      .populate("lobby")
      .populate<Pick<IUserWithEmotions, "emotions">>("emotions.emotion");

    if (!user) {
      let usersInfo;

      usersInfo = await userInfoPlatform({
        userId: req.userId,
        platform: req.platform,
```

```
        authorization: req.authorization,
        userRegistrationData: req.userRegistrationData,
      });

      if (!usersInfo) {
        throw new Error("failed to get user info for create");
      }

      const userInfo = usersInfo[0];

      const date = new Date();

      let userData = {
        _id: req.userId,
        platform: userInfo.platform,
        name: userInfo.name,
        photo_100: userInfo.photo_100,
        photo_200: userInfo.photo,
        registrationDate: date.toISOString(),
        referrals: {},
        bonuses: { notificationBot: false },
        language: "ru"
      };

      // Отмечаем для телеграмма, если пользователь разрешил ему писать
      if (userInfo.platform === Platforms.tg && req?.allowsWriteToPmTg) {
        userData.bonuses.notificationBot = true;
      }
      if (userInfo?.languageCode) {
        userData.language = userInfo?.languageCode;
      }

      const referrerID = Number(req?.userRefferal);

      // Если передан ref, проверяем валидность всех данных и обновляем
модели
      if (referrerID) {
        console.info("ref", referrerID, req.platform);
        const referrer = await UsersModel(req.platform).findOne({
          _id: referrerID,
        });
        /**
         * В каких случаях выбрасываем ошибку:
         * 1. Если referrer не найден (тот, кто пригласил в игру)
         * 2. Если айди приглашающего совпадает с приглашенным (пригласил
сам себя)
         */

        if (!referrer || req.userId === referrerID) {
```

```
        } else {
          // Сохраняем айди, чтобы чуть позже выдать бонус
          userData = { ...userData, referrals: { refId: String(referrerID) }
};
        }
      }
      let user;

      user = await UsersModel(req.platform).create(userData);

      if (!user) throw new Error("failed to create user");

      user = await user.populate<Pick<IUserWithEmotions, "emotions">>(
        "emotions.emotion"
      );

      req.platform === Platforms.vk &&
        recordUserAction(user._id, "registration", {
          vkRef: req?.userRef,
          vkPlatform: req?.userVkPlatform,
          vkInitialParams: req?.authorization,
          hash: req?.hash,
        });

      return user;
    }

    redis.setex(
      `online_list:${user._id}`,
      ONLINE_SECONDS_TIMEOUT,
      user._id + ""
    );

    return user;
  } catch (e) {
    console.log("Find User Error", e);
  }
};
```

- findShop (Возвращает данные магазина, создаёт его при отсутствии)

```
import { Shop } from "@/src/models/Shop";

export const findShop = async () => {
  try {
    let shop = await Shop.findOne();

    if (!shop) {
      console.log("CREATE_SHOP");
```

```
      let shop = await Shop.create({});
      return shop;
    }

    return shop;
  } catch (e) {
    console.log("Find Shop Error", e);
  }
};
```

- mongooseConnect (Подключает mongoose к mongoDb)

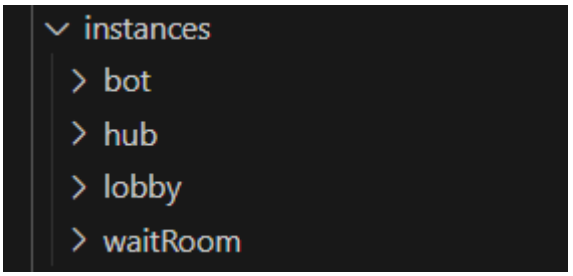
```
import mongoose from "mongoose";

export const mongooseConnect = async () => {
  mongoose.set("strictQuery", true);
  console.info("MONGOOSE", process?.env?.MONGODB_URL);
  await mongoose.connect(process?.env?.MONGODB_URL || "");

  return true;
};
```

- userInfoPlatform (Позволяет получить аватар, имя пользователя и прочие его данные исходя из платформы через которую он авторизовался)

```
import { initData } from "@twa.js/init-data-node";

import { IGetUserIDAuthInfo } from "@/instances/hub/src/types/request";
import { getOkUsersInfo } from "../libs/ok";
import { getUsersInfo } from "./vkapi";
import { Platforms } from "../types/platforms";
import { findUserParams } from "./findUser";

export const userInfoPlatform = async ({
  userId,
  platform,
  authorization,
  userRegistrationData,
}: IGetUserIDAuthInfo & {
  userRegistrationData?: findUserParams["userRegistrationData"];
 }): Promise<any> => {
  if (platform === Platforms.vk) {
    return await getUsersInfo(Number(userId));
  }
  if (platform === Platforms.ok) {
    return await getOkUsersInfo(Number(userId));
  }
  if (platform === Platforms.tg) {
    if (!userRegistrationData && !authorization) {
```

```
      console.error("Нет данных о пользователе для его регистрации");
      return;
    }

    const data = authorization
      ? initData.parse(authorization)?.user
      : userRegistrationData;

    if (!data) {
      return;
    }

    return [
      {
        platform: "tg",
        name: `${data.firstName} ${data?.lastName}`,
        photo_100: "https://i.ibb.co/4NBk2bp/Frame-1321313568.png",
        photo: "https://i.ibb.co/4NBk2bp/Frame-1321313568.png",
        languageCode: data.languageCode,
      },
    ];
  }
};
```

4. Теперь, создадим папки под будущие микросервисы



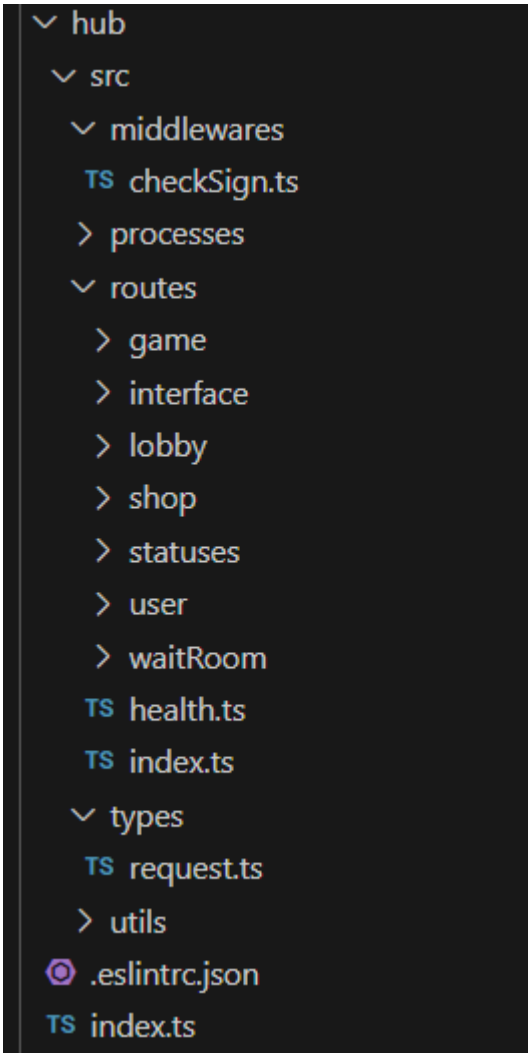
- Bot (Микросервис чат-бота работающего отдельно от проекта)
- Hub (Микросервис для основного Rest API возвращающего данные пользователя по запросу)
- Lobby (Микросервис для обработки игрового процесса в реальном времени посредством socket.io)
- WaitRoom (Микросервис ожидающий подключения двух игроков и перенаправляющий их в конкретное игровое лобби)

5. Наконец, подготовим корневой файл запускающий сервер Hub на express.

```
import express from "express";
import { mongooseConnect } from "../../src/utils/mongooseConnect";
import cookieParser from "cookie-parser";
import cors from "cors";

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

const app = express();

app.use(express.json());
app.use(
  cors({
    origin: "*",
  })
);
app.use(cookieParser());

// Логирование поступающих запросов
app.use("/", (req, res, next) => {
  if (process.env.NODE_ENV === "development")
    console.log(
      `[REQUEST] [${new Date(Date.now()).toLocaleString()}] ${req.path}`
    );
  next();
});

app.use("/", require("./src/routes/index"));

// Запуск сервера
async function start() {
  try {
    // Инициализация редиса для подсчета онлайна игроков
    await new Redis().init();
    new CountOnline();
    // Подключение к mongoose
    await mongooseConnect();

    app.listen(process.env.PORT, () => {
      console.log(
        `[HUB] Server has been started on port ${process.env.PORT} in
${process.env.NODE_ENV} mode`
      );
    });

  } catch (e: any) {
    console.log("Server Error", e.message);
    process.exit(1);
  }
}
```

```
start();
```

6. Также, подготовим основную структуру для сервиса Hub.



В middlewares предусмотрим файл авторизации поддерживающий все 3 платформы на которых расположена игра, в роутах предусмотрим основные эндпоинты для данного микросервиса. Дополнительно, предусмотрим эндпоинт health, чтобы отслеживать аптайм.

Вывод

В рамках работы был разработан boilerplate на express + typescript + mongodb для будущей игры. Также, была заложена основа для

микросервисной архитектуры и разработаны все основные модели проекта.