САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа № 1
“Typescript: основы языка”

Выполнил:

Чан Дык Минь

К33392

Проверил:
Добряков Д. И.

Санкт-Петербург

2024 г.

Задача

Реализовать boilerplate на express + sequelize + typescript. Должно быть явное разделение на:

— модели,

— контроллеры

— роуты,

— сервисы для работы с моделями (реализуем паттерн “репозиторий”).

Ход работы

Понятие **boilerplate** относится к секциям кода, которые должны быть написаны во многих местах с минимальными изменениями. Часто используется по отношению к языкам, в которых программист должен писать много кода, чтобы выполнить минимальную задачу.

Сначала я организовал свой проект:

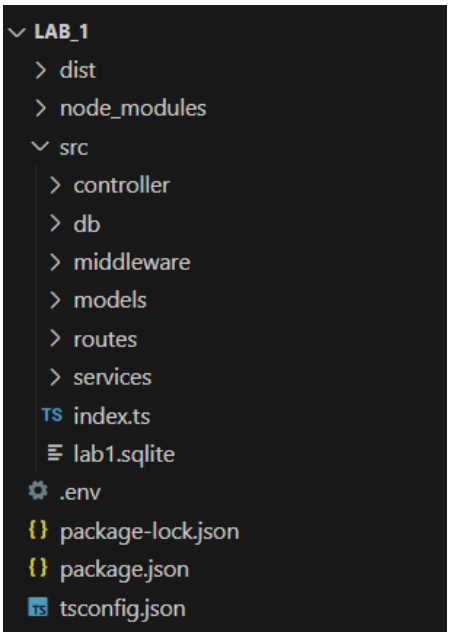
1) настройки npm-пакета,

2) настройки typescript,

3) настройки окружения (файл .env для хранения конфигурационных параметров),

4) базовую настройку express и sequelize, включая модели, контроллеры, роуты и сервисы.

Проект будет иметь следующую структуру:



Далее произведем настройку ORM Sequelize. Параметры конфигурации будем хранить в файле .env.

src > db > TS config.ts > [connection] connection

```ts
import { Sequelize } from "sequelize-typescript";
import { Users } from "../models/users";
import dotenv from 'dotenv';
dotenv.config();

const connection = new Sequelize({
    dialect: "sqlite",
    host: process.env.HOST,
    username: process.env.USERNAME,
    password: process.env.PASSWORD,
    database: process.env.DATABASE,
    storage: "./src/lab1.sqlite",
    models:[Users]
});

export default connection;

```

Далее я создаю модель для пользователей.

```ts
export type UserAttributes = {
    id: number,
    username: string,
    email: string,
    password: string
}

export type UserCreationAttributes = Optional<UserAttributes, 'id'>;

@Table({
    timestamps: false,
    tableName: "users"
})

export class Users extends Model<UserAttributes, UserCreationAttributes> {

    @AllowNull(false)
    @Column(DataType.STRING)
    username!: string;

    @Unique
    @AllowNull(false)
    @Column(DataType.STRING)
    email!: string;

    @AllowNull(false)
    @Column(DataType.STRING)
    password!: string;

    @BeforeUpdate
    @BeforeCreate
    static hashPassword(instance: Users): void {
        const {password} = instance;

        if (instance.changed('password')) {
            instance.password = bcrypt.hashSync(password, bcrypt.genSaltSync(8));
        };
    }
}
```

Services: — это модули, содержащие логику приложения. Они обычно используются для выполнения таких задач, как аутентификация пользователей, взаимодействие с базой данных и других.

```
export const createUser = async (userData: UserAttributes): Promise<Users> => {
    return await Users.create(userData);
};

export const login = async (userData: UserLogin): Promise<AuthResult | null> => {
    try {
        const user = await Users.findOne({ where: { username: userData.username } });
        if (!user) {
            throw new Error("User is not exist");
        }
        const isPasswordValid = await bcrypt.compare(userData.password, user.password);
        if (!isPasswordValid) {
            throw new Error("Password is not correct");
        }
        else {
            const token = jwt.sign({ username: userData.username }, process.env.SECRET_KEY || '');
            return { username: user.username, token };
        }

    } catch (error) {
        console.error("Error occurred while logging in:", error);
        throw error;
    }
};
```

```
export const deleteUser = async (id: number): Promise<Users | null> => {
    const userToDelete = await Users.findByPk(id);
    if (!userToDelete) throw new Error("User not found");

    await Users.destroy({ where: { id } });
    return userToDelete;
};

export const getUserById = async (id: number): Promise<Users | null> => {
    return await Users.findByPk(id);
};

export const getAllUsers = async (): Promise<Users[]> => {
    return await Users.findAll();
};

export const updateUsers = async (id: number, userData: Partial<UserAttributes>): Promise<Users | null> =>
    await Users.update(userData, {where:{id}});
    return await Users.findByPk(id);
}
```

Controller: где обрабатываются запросы со стороны клиента. Они получают запросы от маршрутизаторов, вызывают **services** для выполнения необходимой логики, а затем возвращают результаты клиенту через HTTP-ответ.

```ts
export const registerUser: RequestHandler = async(req, res, next) => {
    try{
        const user = await userService.createUser(req.body);
        return res.status(201).json({ message: "User registered successfully!", data: user });
    } catch (error: any) {
        return res.status(500).json({ message: "Error creating user", error: error.message });
    }
};

export const loginUser: RequestHandler = async(req, res, next) => {
    try{
        const user = await userService.login(req.body);
        return res.status(201).json({ message: "User logged in successfully!", data: user });
    } catch (error: any) {
        return res.status(500).json({ message: "Error login user", error: error.message });
    }
};

export const deleteUser: RequestHandler = async(req, res, next) => {
    try{
        const {id} = req.params;
        const user = await userService.deleteUser(parseInt(id));
        if (!user) {
            return res.status(404).json({ message: "User not found" });
        }
        return res.status(200).json({ message: "User deleted successfully!", data: user });
    } catch (error: any) {
        return res.status(500).json({ message: "Error deleting user", error: error.message });
    }
};

```

```ts
export const getAllUser: RequestHandler = async(req, res, next) => {
    try{
        const allUsers = await userService.getAllUsers();
        return res.status(200).json({ message: "User fetched using userService successfully!", data: allUsers
    } catch (error: any) {
        return res.status(500).json({ message: "Error fetching user", error: error.message });
    }
};

export const getUserById: RequestHandler = async(req, res, next) => {
    try{
        const {id} = req.params;
        const user = await userService.getUserById(parseInt(id));
        if (!user) {
            return res.status(404).json({ message: "User not found" });
        }
        return res.status(200).json({ message: "User fetched using userService successfully!", data: user });
    } catch (error: any) {
        return res.status(500).json({ message: "Error fetching user", error: error.message });
    }
};

export const updateUser: RequestHandler = async(req, res, next) => {
    try{
        const {id} = req.params;
        const user = await userService.updateUsers(parseInt(id), req.body);
        if (!user) {
            return res.status(404).json({ message: "User not found" });
        }
        return res.status(200).json({ message: "User fetched using userService successfully!", data: user });
    } catch (error: any) {
        return res.status(500).json({ message: "Error fetching user", error: error.message });
    }
};
```

Для маппинга входящих HTTP-запросов к соответствующим методам контроллера был создан роутер

```ts
const router = Router();

router.post("/register", registerUser);
router.post("/login", loginUser);
router.get("/", auth, getAllUser);
router.get("/:id", auth, getUserById);
router.put("/:id", auth, updateUser);
router.delete("/:id", auth, deleteUser);
```

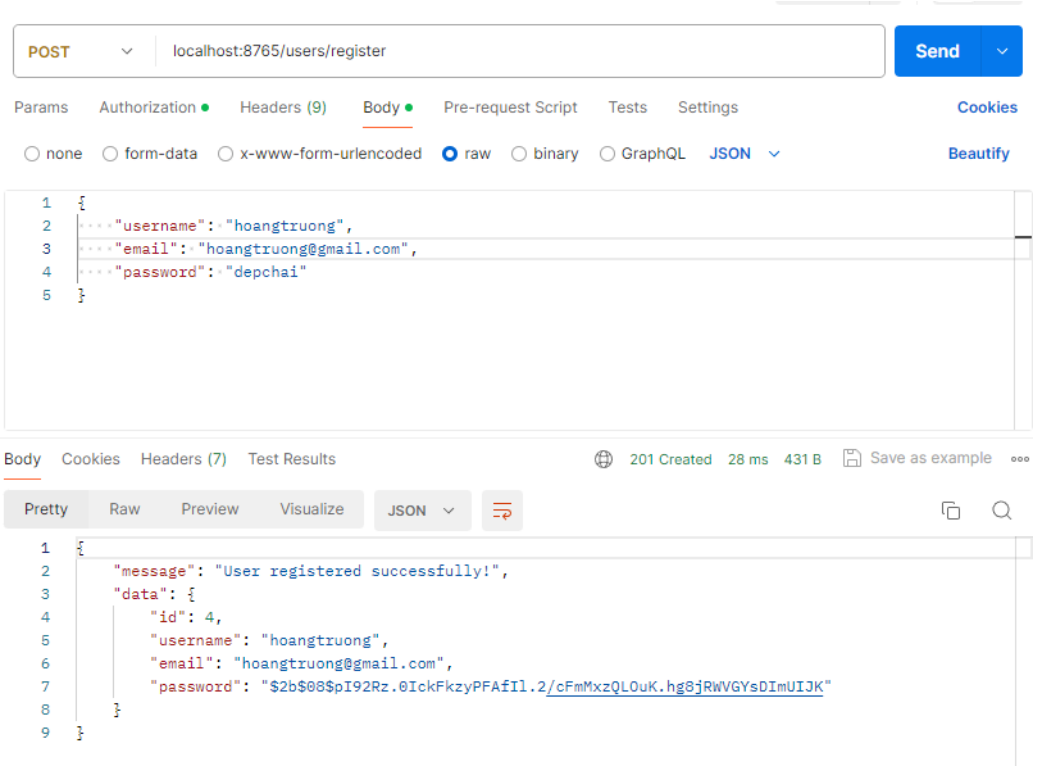
После был создан общий роутер, агрегирующий маршруты от всех роутеров низшего порядка. Роутер для модели User использовался с префиксом /users

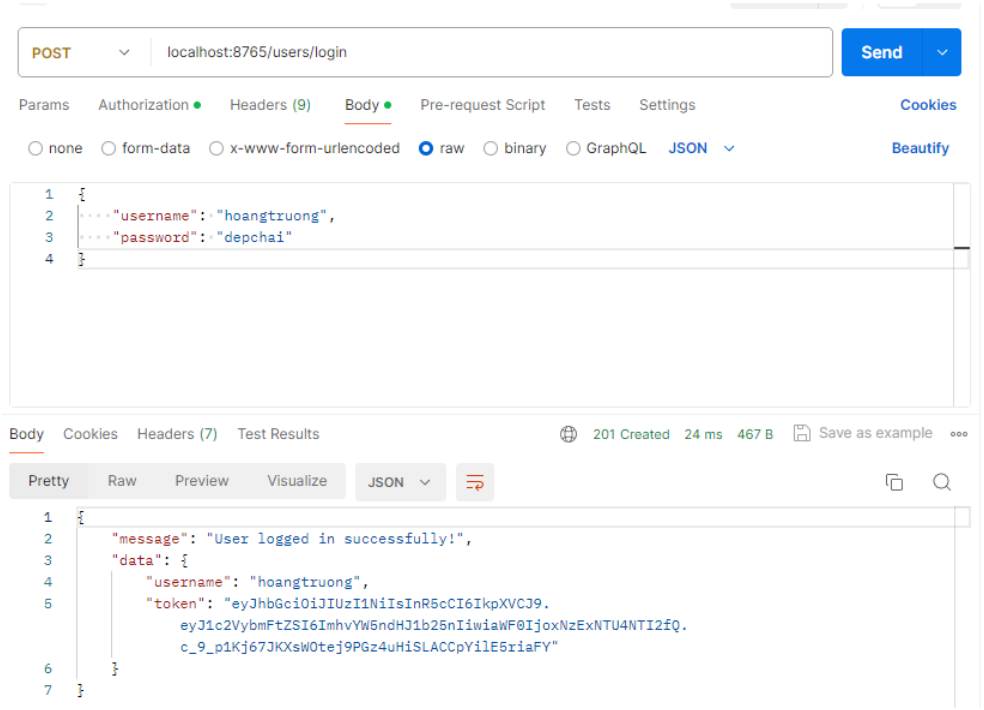
```
import userRouters from "./routes/users";

const app: Express = express()
app.use("/users", userRouters)
```

Затем есть другие конфигурации для запуска приложения.

```
app.use((
    err: Error,
    req: express.Request,
    res: express.Response,
    next: express.NextFunction,

) => {
    res.status(500).json({message: err.message});
});

connection.sync().then(() => {
    console.log("Database synced successfully!", __dirname);
}).catch((err)=>{
    console.log("Error", err);
});


app.listen('8765')
```

Проверим работоспособность приложения в среде Postman. Попробуем зарегистрироваться и войти как новый User.





Мы видим, что операции завершились успешно, поэтому развернутую сводку можно считать живой.

Вывод

В ходе лабораторной работы был изучен язык TypeScript, а также способы использования TypeScript для работы с Express и Sequelize. В результате был разработан Boilerplate, который может использоваться другими лабораториями.