

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа № 1
“Typescript: основы языка”

Выполнил:
Ле Хоанг Чыонг

Группа:
К33392

Проверил:
Добряков Д. И.

Санкт-Петербург
2024 г.

Задача

Нужно написать свой boilerplate на express + sequelize / TypeORM + typescript.

Должно быть явное разделение на:

- модели
- контроллеры
- роуты
- сервисы для работы с моделями (реализуем паттерн “репозиторий”)

Ход работы

"Boilerplate" - это шаблонный код или начальная заготовка, которая используется для инициализации нового проекта или компонента. Этот код содержит основные структуры, конфигурации и функциональность, необходимые для начала работы над проектом.

Для начала опишем структуру boilerplate проекта. Он должен содержать в себе:

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.

Сначала я развернул команду npm для сборки, тестирования и запуска проекта (команды dev, test и build).

Реализованные команды приведены на рисунке 1.

```
"scripts": {  
  "dev": "tsnd --respawn index.ts",  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "build" : "npm tsc"  
},
```

Рисунок 1 — Реализованные npm-команды

Проект будет иметь следующую структуру:

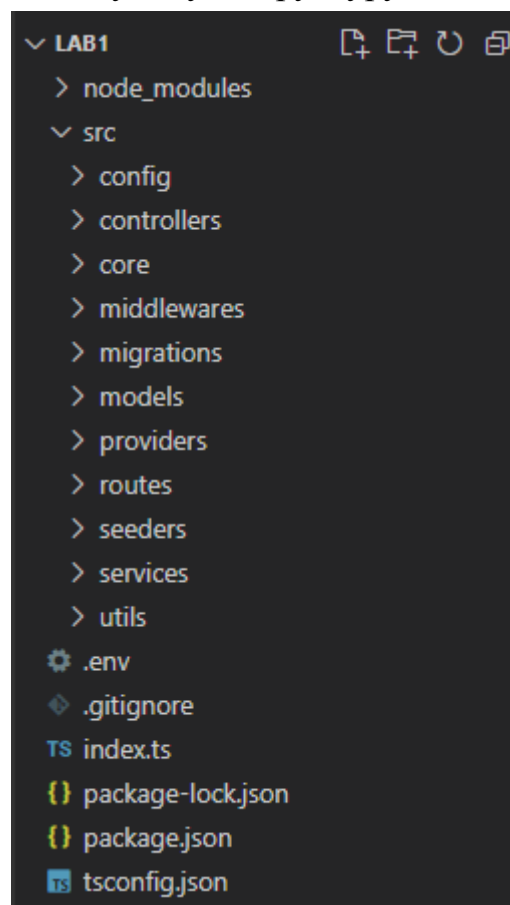


Рисунок 2 — Файловая структура проекта

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

```
import { Sequelize } from 'sequelize-typescript'

import User from '../models/users/user'
require('dotenv').config();

const sequelize = new Sequelize({
  username: process.env.DB_USER,
  password: process.env.DB_PASS,
  database: process.env.DB_NAME,
  host: "127.0.0.1",
  dialect: "postgres"
})
```

Рисунок 3 — Конфигурация ORM Sequelize

Далее я создаю модель для пользователей.

```
import hashPassword from '../../utils/hashPassword'
import { Table, Column, Model, Unique, AllowNull, BeforeCreate, BeforeUpdate, PrimaryKey, DataType } from 'sequelize-typescript'
import { Optional } from "sequelize";

export type UserAttributes = {
  id: string,
  name: string,
  email: string;
  password: string;
  // other attributes...
};

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

@Table
export class User extends Model<UserAttributes, UserCreationAttributes> {
  @PrimaryKey
  @Column({
    type: DataType.UUID,
    defaultValue: DataType.UUIDV4,
  })
  id: string;

  @Column
  name: string;

  @Unique
  @Column
  email: string

  @AllowNull(false)
  @Column
  password: string

  @BeforeCreate
  @BeforeUpdate
  static generatePasswordHash(instance: User) {
    const { password } = instance

    if (instance.changed('password')) {
      instance.password = hashPassword(password)
    }
  }
}

export default User
```

Рисунок 4 — Модель User

Далее я реализовал 2 сервиса для работы с моделью: AuthService и UserService.

```
import User from '../../models/users/user'
class UserService {
  async getById(id: number): Promise<any> {
    try {
      const user = await User.findByPk(id)
      if (user) return user
    }
    catch (error) {
      throw new Error('Not Found');
    }
  }

  async getAll(): Promise<any> {
    try {
      const user = await User.findAll()
      return user
    } catch (error) {
      throw error;
    }
  }

  async update(id: number, userData: any): Promise<any> {
    try {
      const [updatedRowCount, updatedUser] = await User.update(userData, {
        where: { id },
        returning: true,
      });

      if (updatedRowCount === 0) {
        throw new Error('User not found');
      }
      return updatedUser[0];
    } catch (error) {
      throw error;
    }
  }

  async delete(id: number): Promise<number> {
    try {
      const deletedRowCount = await User.destroy({
        where: { id },
      });
      if (deletedRowCount === 0) {
        throw new Error('User not found');
      }
      return deletedRowCount;
    } catch (error) {
      throw error;
    }
  }
}
```

Рисунок 5 — UserService

```

import User from '../../models/users/user'
import jwt, { Secret, JwtPayload } from 'jsonwebtoken';
import checkPassword from '../../utils/checkPassword'
require('dotenv').config()

if (!process.env.SECRET_KEY) {
  throw new Error('Missing SECRET_KEY in environment variables');
}

export const SECRET_KEY: Secret = process.env.SECRET_KEY;

class AuthService {

  async register(userData: any): Promise<User> {
    try {
      const user = await User.create(userData)
      return user
    } catch (error) {
      throw error;
    }
  }

  async login(email: string, password: string): Promise<any> {
    try {
      const user = await User.findOne({ where: { email } })

      if (!user || !checkPassword(user, password))
        throw new Error('Email or password is not correct');

      const token = jwt.sign({ id: user.id?.toString() }, SECRET_KEY, {
        expiresIn: '2 days',
      });
      return { user, token: token };
    } catch (error) {
      throw error;
    }
  }
}

export default AuthService

```

Рисунок 5 — AuthService

Продолжая, я развернул контроллер и вызвал ранее развернутый сервис для обработки необходимой логики. Здесь я создал 2 контроллера, User и Auth.

```
import UserService from '../..services/users/user'
import { getErrorMessage } from '../..utils/getErrorMessage';
export default class UserController {
  private userService: UserService

  constructor() {
    this.userService = new UserService()
  }

  get = async (request: any, response: any) => {
    try {
      const user: any = await this.userService.getAll()
      response.status(201).send(user)
    } catch (error: any) {
      response.status(404).send(getErrorMessage(error))
    }
  }

  update = async (request: any, response: any) => {
    const { body } = request
    const userId = request.user.id;
    try {
      const user : any = await this.userService.update(userId, body)
      response.status(201).send(user)
    } catch (error: any) {
      response.status(400).send(getErrorMessage(error))
    }
  }

  delete = async (request: any, response: any) => {
    const userId = request.user.id;
    try {
      const number : any = await this.userService.delete(userId)

      response.status(201).send('User have successful deleted')
    } catch (error: any) {
      response.status(400).send(getErrorMessage(error))
    }
  }

  me = async (request: any, response: any) => {

    response.send(request.user)
  }
}
```

Рисунок 5 — UserController


```

import AuthService from '../../../services/auth/auth'
import { getErrorMessage } from '../../../utils/getErrorMessage';
export default class AuthController {
  private authService: AuthService

  constructor() {
    this.authService = new AuthService()
  }

  login = async (request: any, response: any) => {
    try {
      const { email, password } = request.body;
      const data: any = await this.authService.login(email, password)
      response.status(201).send(data)
    } catch (error: any) {
      response.status(404).send(getErrorMessage(error))
    }
  }

  register = async (request: any, response: any) => {
    const { body } = request
    try {
      const data : any = await this.authService.register(body)
      response.status(201).send(data)
    } catch (error: any) {
      response.status(400).send(getErrorMessage(error))
    }
  }
}

```

Рисунок 6 — AuthController

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

```
import express from "express"
import AuthController from "../../controllers/auth/auth"

const router: express.Router = express.Router()

const controller: AuthController = new AuthController()

router.post('/login', controller.login);
router.post('/register', controller.register);

export default router
```

Рисунок 7 — User Route

```
import express from "express"
import UserController from "../../controllers/users/user"

const router: express.Router = express.Router()

const controller: UserController = new UserController()

router.get('/me', controller.me)
router.route('/')
  .get(controller.get)
  .patch(controller.update)
  .delete(controller.delete)
export default router
```

Рисунок 8 — Auth Route

Далее, был создан класс, ответственный за запуск express приложения. Фрагмент приведен на рисунке 9

```
import express from "express"
import { createServer, Server } from "http"
import sequelize from "../providers/db"
import { Sequelize } from 'sequelize-typescript'
import bodyParser from "body-parser"
import cors from "cors";
import routes from "../routes/v1/index"
require('dotenv').config()
export default class App {
  public port: number
  public host: string

  private app: express.Application
  private server: Server
  private sequelize: Sequelize
  constructor(port = 8000, host = "localhost") {
    this.port = Number(process.env.PORT) || port
    this.host = process.env.HOST || host


    this.app = this.createApp()
    this.server = this.createServer()
    this.sequelize = sequelize
  }
  private createApp(): express.Application {
    const app = express()
    app.use(cors())
    app.use(bodyParser.json())
    app.use('/v1', routes)

    return app
  }
  private createServer(): Server {
    const server = createServer(this.app)

    return server
  }
  public start(): void {
    this.server.listen(this.port, () => {
      console.log(`Running server on port ${this.port}`)
    })
  }
}
```

Рисунок 9 — Класс App

И, наконец, в файле `index.ts` создается и запускается express-приложение (рисунок 10)

A screenshot of a code editor showing three lines of TypeScript code. The first line is an import statement for 'App' from './src/core'. The second line creates a new instance of 'App' and assigns it to a constant 'app'. The third line calls the 'start()' method on the 'app' object. The code is color-coded: 'import' is purple, 'App' is blue, 'from' is pink, and the string './src/core' is orange. 'const' is blue, 'app' is blue, 'new' is green, and 'App()' is blue. 'app.start()' is blue, and 'start()' is yellow.

```
import App from "./src/core";  
  
const app = new App();  
  
app.start();
```

Рисунок 10 — Создание и запуск приложения

Проверим работоспособность приложения в среде Postman.

Попробуем зарегистрироваться и войти как новый User

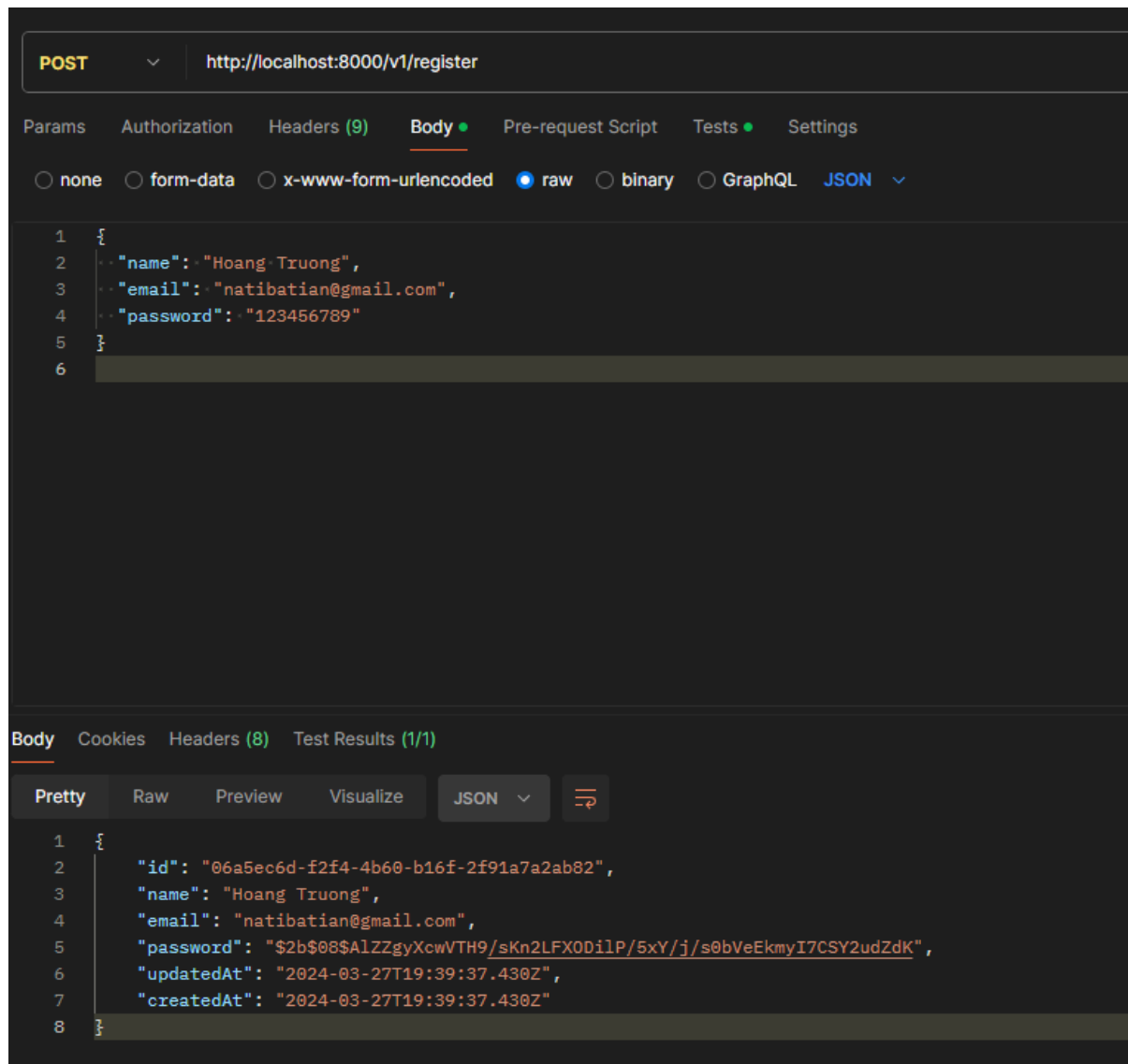


Рисунок 11 — Register

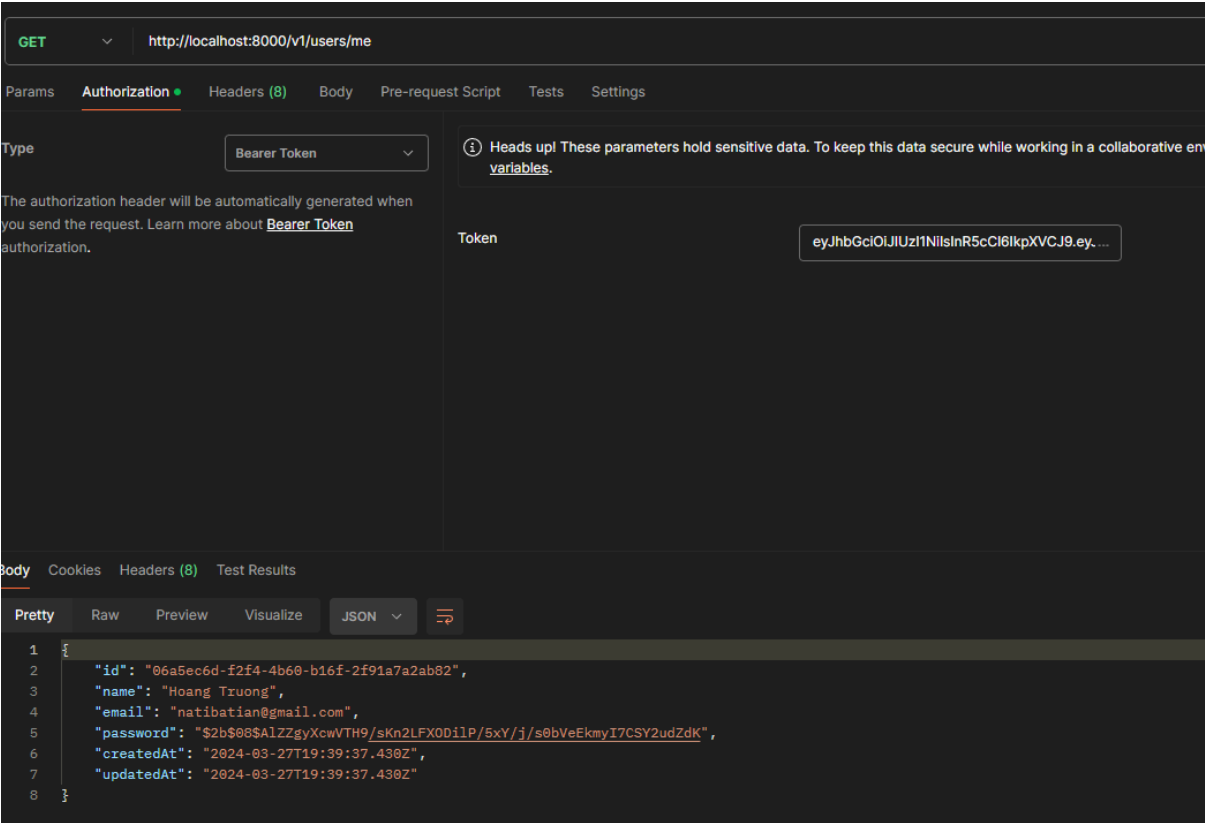


Рисунок 13 — Get Me

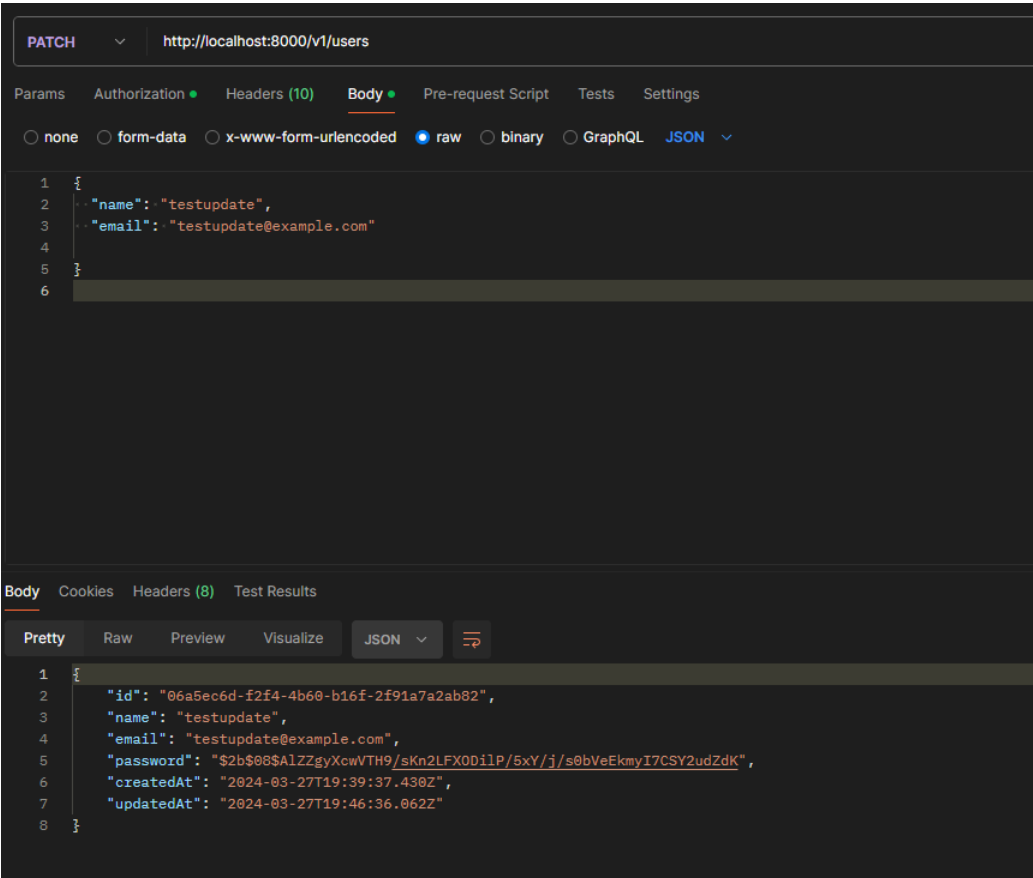


Рисунок 14 — Update User

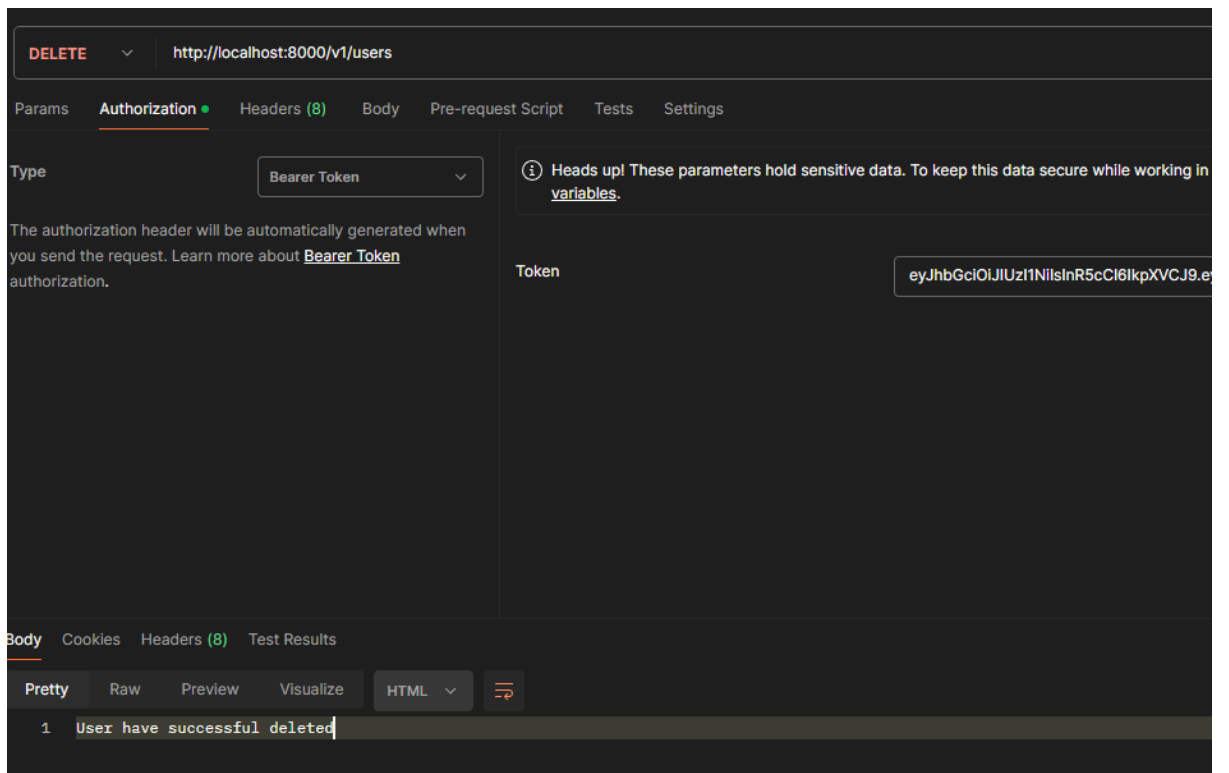


Рисунок 15 — Delete User

Можем видеть, что операции были выполнены успешно, поэтому реализованный boilerplate можно считать рабочим.

Вывод

Во время работы в лаборатории я изучил основы языка TypeScript, а также Sequelize-TypeScript и экспресс-библиотек. В результате был реализован шаблон, который можно использовать в проектах.