Refactor súborovej štruktúry v rozhraní ExpressJS

Peter Kúdela

Fakulta informatiky a informačných technológií
Slovenská technická univerzita v Bratislave

October 05, 2022

Zámer

Písanie kódu v jazyku JavaScript predstavuje veľkú voľnosť týkajúcu sa organizácie súborov v projektoch, či už ide o exportovanie a importovanie objektov medzi súbormi, alebo syntaxu, kde sa dá funkcia definovať minimálne tromi rôznymi spôsobmi. V našej práci budeme analyzovať, pozorovať a upravovať rôzne softvérové projekty napísané v tomto jazyku. Budeme sledovať, či zmeny vykonané na súborovej štruktúre projektu spôsobujú pozorovateľnú zmenu vo výkone bežiaceho projektu, ale aj ich impakt na čitateľnosť samotného kódu. Zameriame sa na existujúce Javascript frameworky, ktoré už majú z časti prednastavenú štruktúru súborov. Pozrieme sa na front-end frameworky, ako Vue, NextJS, React, Svelte atď., ale aj back-end štruktúru súborov používajúc balík expressJS. Predstavíme názory a nápady, ako by mala dobrá softvérová štruktúra vypadať, berúc ohľad na moduláciu, čitateľnosť a efektívnosť. Budeme diskutovať o otázkach, ako: Mala by webová aplikácia v JavaScripte obsahovať iba jeden globálny stylesheet súbor, mala by obsahovať striktne modulárne štýlové súbory, alebo by mala nejak implementovať kombináciu týchto dvoch prístupov. Je lepšie, ak sa back-end a front-end kód nachádza v jednom mono-repozitáry? Alebo by mal byť rozdelený do dvoch reprezentujúcich back-end a front-end. Ktoré abstrakcie v kóde sa oplatí zoskupovať do jedného súboru ak nejaké? Ako predstavenie TypeScript-u v tejto problematike ovplyvňuje naše pozorovania.

Úvod

V tejto eseji sa preto budeme zaoberať tým, ako by mal vyzerať správny, modulárny program napísaný vo frameworku ExpressJS. ExpressJS je „unopinionated“ minimalistický webový framework pre nodeJS. Unopinionated, čo znamená, že nemá pevne danú štruktúru kódu a jeho rozdelenie, čo nám umožňuje v práci navrhnúť súborovú štruktúru a uviesť, prečo je práve nami navrhnutá štruktúra vhodná.

1 Modulácia

Všetky projekty napísané v javascripte by sa teoreticky dali napísať v jednom súbore, to však samozrejme vo väčšine prípadov použitia nieje vhodné. Preto

Použitie individuálnych funkčných jednotiek na zostavenie systému. [3] Okrem žiakov softvérového inžinierstva alebo iných odborov, od ktorých sa vyžaduje v nejakej miere tvoriť programy, navrhovať ich a programovať ich, moduláciu v nejakom množstve používame v praxi všetci. Bez nej by bolo veľmi pravdepodobné, že všetok náš kód by pozostával z jedného monolitického súboru, obsahujúceho celú logiku programu. Čo samozrejme nie je ideálne. [1] S narastajúcou výškou komplexnosti a veľkosťou programu sa potreba modulácie zvyšuje. Nechceme sa ocitnúť v situácií, kedy potrebujeme nájsť veľmi špecifickú čast kódu, a uvedomíme si, že ju musíme hľadať v súbore, ktorý obsahuje viac, ako dvetisíc riadkov. Preto sa reálne snažíme program rozdeliť na akési logické časti alebo aj moduly. Tieto moduly ďalej exportujeme a importujeme do súborov, kde ich aj reálne používame.

Niektoré Javascript framework štruktúru takéhoto rozdelenia priamo prikazujú. Takto pevne stanovená štruktúra kódu so sebou prináša aj výhody aj nevýhody. Uľahčuje prácu developerom, ktorý prišli už do rozrobeného projektu. Nevýhodou je však odobranie možnosti navrhnúť a aplikovať svoje vlastné štýly štruktúrovaní projektu.

2 Javascript modul

Pojem JavaScript modul označuje malé jednotky nezávislého, opakovane použiteľného kódu. Majú odlišné funkcie, čím umožňujú ich pridávanie, odstraňovanie bez narušenia systému. Princípom sú veľmi podobné triedam v jazykoch Java alebo Python. [7] Počas vývoja javascriptu vzniklo viacero štandardov pre tieto moduly, existuje ich množstvo, ako napríklad: CommonJS, UMD, AMD, System, ES2020, ES2015 tiež známy ako ES6.

3 Analýza

3.1 Zlý príklad

Na predmete dbs sme mali za úlohu napísať REST api, bol to náš prvý styk s tvorbou takejto aplikácie, a preto jeho čitateľnosť a efektívnosť utrpela. Nasledujú zvýraznené ukážky kódu. Logika express rozhrania je označená modrou farbou. Logika komunikácie s databázou je označená zelenou farbou, a logika prístupových bodov je označená červenou farbou.

```
1  //require('dotenv').config()
2  const express = require('express');
3  const { Sequelize, DataTypes, Model, Op, or } = require('sequelize');
4  //const sequelize = new Sequelize(process.env.DATABASE_URL);
5  const { sequelize} = require('./models')
6  const db = require('./models')
7  const app = express();
8  const port=process.env.PORT || 3000;
9  const dateFormat = require("dateformat");
10 const { createSolutionBuilderHost } = require('typescript');
11 //const raw_issues = require('./models/raw_issues')(Sequelize, DataTypes);
12
13 app.use(require('sanitize').middleware);
14
15 app.get('/', (req, res) => {
16     res.redirect("https://fiit-dbs-xkudela-app.azurewebsites.net/v1/health")
17 })
18
19 app.get('/v1/health', async (req, res) => {
20   const [results, metadata] = await sequelize.query("SELECT date_trunc('second',
21   let resu = {
22     'pgsql':
23     {'uptime': `${results[0].uptime.days} days ${results[0].uptime.hours}:${resu
24   };
25   res.send(resu)
26 })
27
```

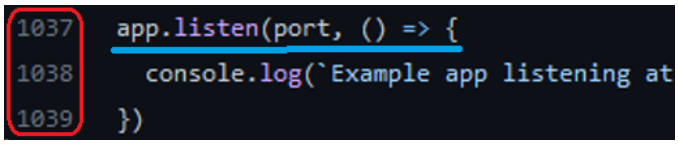
Obr. 1: Poradie importov

Na obrázku 1 je vidieť, že v poradí importov nie je žiadny systém.

```
66
67      toSend ={
68        "items": results,
69        "metadata": {
70          "page": Number(page),
71          "per_page": Number(per_page),
72          "pages": Math.ceil(totalRes[0].count / per_page),
73          "total": Number(totalRes[0].count)
74        }
75    }
76
77
78    res.send(toSend);
79
80  })
81
82  app.use(express.json())
83  app.post('/v1/ov/submissions', async(req, res) =>{
84    //Getting todays date
85    let today = dateFormat(new Date(), "isoDateTime");
86
87    //Validations
88    errors = {
89      "errors": []
90    }
91    if(!req.body.br_court_name) {
92      errors.errors.push({
93        "field": "br_court_name",
94        "reasons":[
95          "required"
96        ]
97      })
98    }
```

Obr. 2: Použitie middleware

Na obrázku 2 sme začali používať middleware uprostred logiky dvoch prístupových bodov.



```
1037  app.listen(port, () => {
1038    console.log(`Example app listening at
1039  })
```

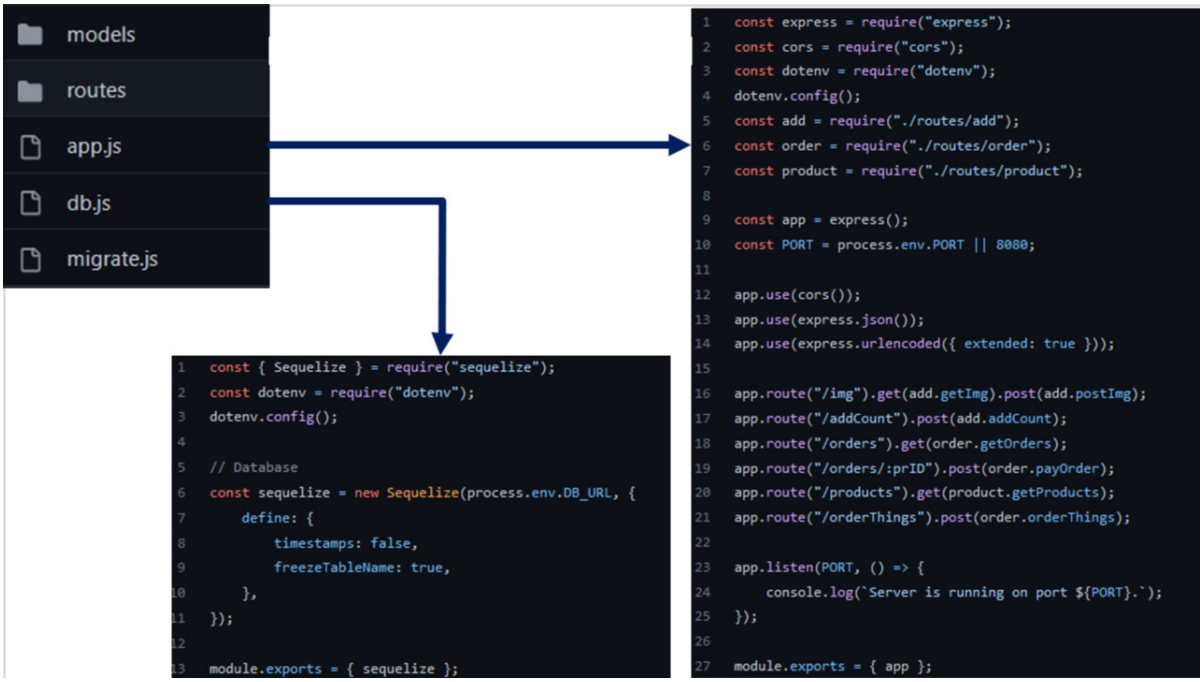
Obr. 3: Nekončiace riadky

A keďže logika celého programu je napísaná v jednom súbore, aplikácia začína počúvať až na riadku 1000+.

3.2 Lepší príklad

Semestre prichádzali a odchádzali, skúsenosti sa nadobúdali a s tým sa zvyšovala aj kvalita nášho kódu, preto, ako ukážku lepšieho príkladu sme sa rozhodli použiť projekt z predmetu VAVJS (Vývoj aplikácií v jazyku javascript). [2] Je vhodné porovnávať ho s uvedeným zlým príkladom skrz to,

že v obe aplikácie sú napísané v rozhraní ExpressJS a ide o REST API. Je tu však vidieť pokrok v skúsenostiach pri písaní kódu.



Obr. 4: Lepšia štruktúra

Už len skrz to, že sa celá logika aplikácie a jej koncových bodov zmestila na jeden obrázok je jasné, že prehľadnosť a modulácia tohto kódu je na úplne inej úrovni, ako predchádzajúceho. Poďme analyzovať, čo sú kľúčové vlastnosti, alebo aj vzory tohto kódu, ktoré bude vhodné zapamätať si a použiť v budúcnosti.

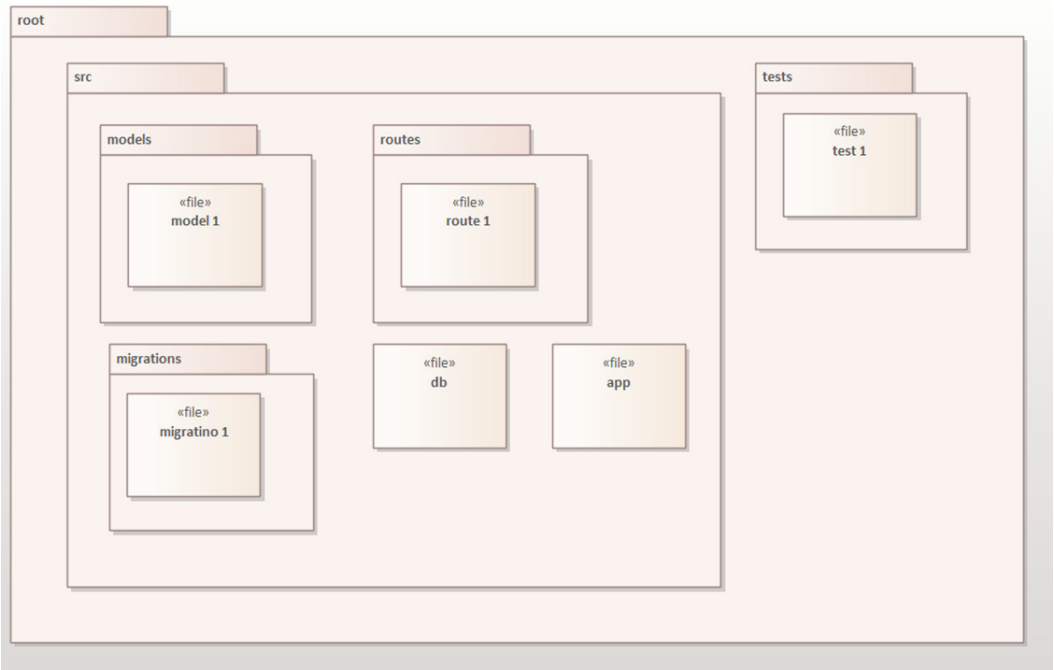
Pravdepodobne prvé, čo si každý na tejto ukážke všimne je modulácia. Celá aplikácia nie je napísaná v jednom súbore. Jej logika je rozdelená do samostatných modulov [4], ktoré naďalej importujeme a exportujeme kde potrebujeme. Týmto sa nám podarilo aj zapuzdrovať niektorú logiku, napríklad logika komunikácie s databázou. Express aplikácia s databázou vôbec nekomunikuje, no importuje v sebe logiku koncových bodov, ktoré s ňou komunikujú.

4 Návrh ideálnej štruktúry

Cieľom tejto práce je analyzovať štruktúru už napísaných kódov, identifikovať v nich dobré a zlé vzory, a po tejto analýze vytvoriť návrh ideálnej štruktúry aplikácie. Poďme sa preto pozrieť na už existujúce návrhy ideálnej

štruktúry express aplikácie.

Pri návrhu štruktúry potrebujeme najprv urobiť jedno závažné rozhodnutie, podľa čoho budeme zoskupovať logiku? Pri čítaní už existujúcich projektov v expresse sme pozorovali dva prístupy, a to: Zoskupenie na moduly alebo zoskupenie podľa typu. Príklad z predchádzajúcej kapitoly využíva práve združovanie podľa typu, skrz to, že všetka logika koncových bodov sa nachádza v zložke routes atď. Jej grafické znázornenie vyzerá nasledovne:

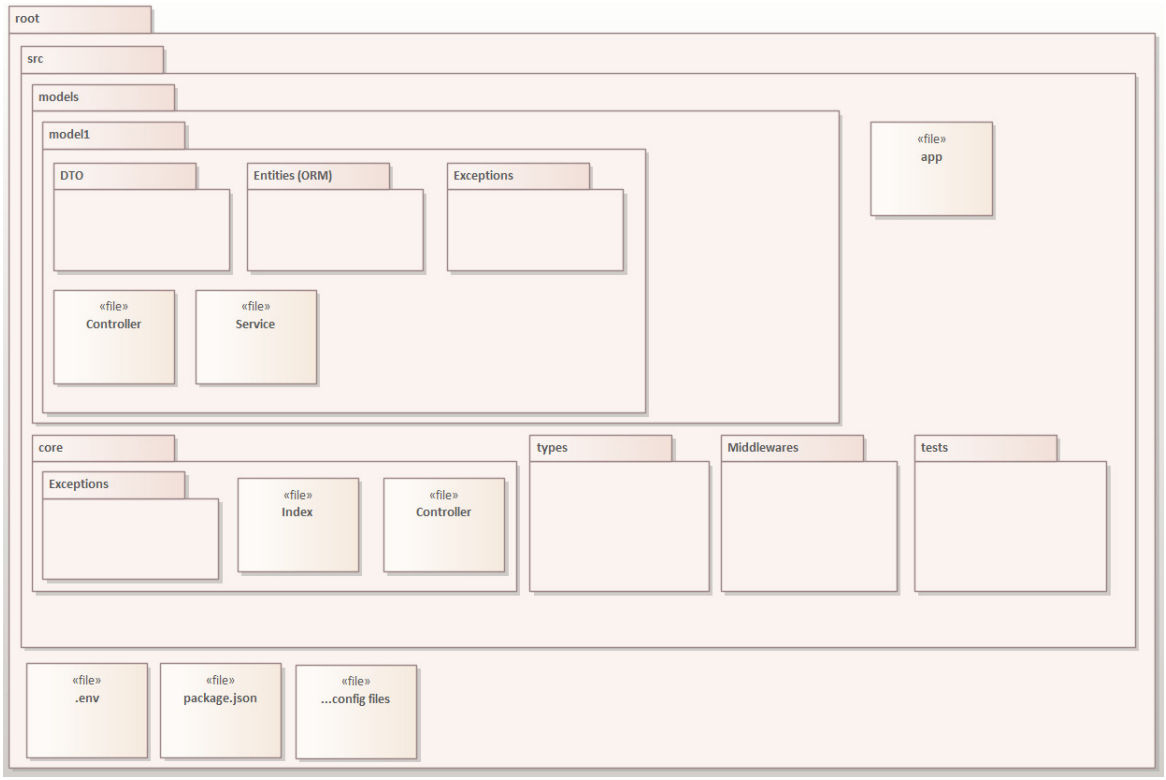


Obr. 5: Schéme lepšieho príkladu

Takéto rozdelenie je vhodné pre projekty, ktoré sú rozsahom malé. Neobsahujú veľa koncových bodov, nekomunikujú s veľkým množstvom entít v databáze. Pri hľadaní konkrétnej logiky sa stále nestrácame, pretože v žiadna zložka na to neobsahuje dostatočne veľké množstvo súborov. Dalo by sa povedať, že podľa nás toto je ideálna štruktúra no len pre rozsahovo menšie projekty.

Teraz je už ale čas navrhnúť ideálnu súborovú štruktúru pre veľké a stredne veľké projekty. Tu by sme sa mohli inšpirovať "passion"projektom, ktorý sme začali v minulosti spolu s vtedy skúsenejším kolegom. [5] Projekt samotný sa veľmi ďaleko nedostal, no mal vytvorený solídny základ pre súborovú štruktúru kde sa logika delila podľa modulov. Išlo o online chat aplikáciu, v príklade sa preto budú vyskytovať entity ako "miestnosťä "správa". V projekte bol využitý framework "NestJS". [6]

NestJS za nás rieši veľké množstvo otázok skrz to, že tento framework má silné názory (strongly opinionated). Podporuje MVC model. Štruktúra, ktorú NestJS zvolil sa nám dosť pozdáva, preto sa ňou budeme ďalej inšpirovať pri návrhu našej štruktúry.



Obr. 6: Graf návrhu ideálnej štruktúry

Pri tejto schéme už používame rozdelenie do zložiek podľa modulov. Poďme si teraz vysvetliť túto štruktúru.

4.1 Modules

Obsahujú zložky všetkých modulov projekty.

Model obsahuje *controller* a *service* a mal by obsahovať *DTO*, *entity* a *exeptions*.

Entities obsahujú súbory s *ORM* definíciou entít z databázy.

Exceptions obsahujú definície errorov, ktoré aplikácia vyhodí pri nami zvolených scenároch.

DTO (Data Transfer Object) v tejto súborovej štruktúre funguje na definovanie validácie pre konkrétne úkony, napríklad vytvorenie entity bude prihliadať na to, či všetky parametre pre ňu potrebné sú zadané správne.

Napríklad, či parameter "meno"má počet písmen v rozmedzí ktoré si v DTO určíme.

Controller validuje vstup, ktorý aplikácia príma pomocou vyššie spomenutých DTO, ak validácia prebehne neúspešne, controller vráti error, ktorý sme si pre danú situáciu vytvorili vo vyššie spomenutých exeptions. Ak validácia prebehne úspešne, controller vráti návratovú hodnotu z daného servisu.

Service obsahuje logiku, ktorá vracia dáta, alebo manipuluje s dátami priamo z databázy cez ORM.

4.2 Core

Týmto sme si predstavili celú štruktúru pre jednotlivé moduly, no okrem toho sa v kóde ďalej nachádza logika aplikácie. Zložka *core* obsahuje práve túto logiku kde:

Controller v zložke core, je abstraktnou triedou, ktorú ďalej rozširujú všetky ostatné controllery v projekte. Týmto zaručujeme enkapsulovanosť premennej ëxpress.routerä dávame prístup k nej len týmto triedam.

Index je trieda, ktorá všetko spája dokopy. Inicializuje všetky middlewares, všetky súbory s logikou koncových bodov a obsahuje funkciu, na štart servera.

4.3 Ostatné

Súbor **App** v zložke **Src** len vytvára inštanciu triedy z minulého paragrafu a nasledovne volá jej štart funkciu. Týmto princípom sme schopný spustiť viac inštancií serveru, ak to v budúcnosti budeme potrebovať.

Types obsahuje definície typov pre typescript. (možné preskočiť pokiaľ ide o projekt v javascripte a nie typescripte)

Middlewares obsahuje všetky nami definované middlewares.

Tests obsahuje automatizované testy.

Root obsahuje zložku **src** ktorú sme teraz podrobne opísali, no okrem toho sa v nej nachádzajú ďalšie súbory. Toto miesto je vyhradené len pre konfiguračné súbory pre nástroje/frameworky, ktoré používame ako *tsconfig*, *package.json*, *.env* a ďalšie.

4.4 Objasnenie

Kombináciou použitia kľúčových slov `public` a `private`, a využitím modelu Model View Controller prirodzene enkapsulujeme kód. Každá časť kódu má preto prístup len ku logike, s ktorou reálne pracuje. Okrem toho nám takáto štruktúra umožňuje ľahko a intuitívne vyhľadať, kde sa konkrétne časti kódu nachádzajú vďaka princípu rozdelenia podľa modulov a nie typu.

Záver

V práci sme objasnili dôležitosť modulácie a prečo by sme ju mali aplikovať. Ďalej sme analyzovali už existujúce riešenia štrukturovaní ExpressJS aplikácií, pri analýze sme hľadali kladné a záporné vlastnosti daných rozložení ktoré nám ďalej pomohli pri vytvorení návrhu ideálnej štruktúry aplikácie. Pri vypracovaní práce sme išli viac do hĺbky ako do šírky, čo viedlo k zúženi témy z refaktoru súborových systémov v jazyku javascript an refaktor súborových systémov v rozhraní ExpressJS. Toto zúženie nám dovolilo hlbšie sa pozrieť na už existujúce riešenia a inšpirovať sa ďalšími rozhraniami ako je NestJS.

Referencie

[1] J. Hughes. "Why Functional Programming Matters". In: *The Computer Journal* 32.2 (jan. 1989), s. 98–107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98. eprint: https://academic.oup.com/comjnl/article-pdf/32/2/98/1445644/320098.pdf. URL: https://doi.org/10.1093/comjnl/32.2.98.

[2] Peter Kúdela. *E-shop*. 2021. URL: https://github.com/Drejky/VAVJS-Eshop (cit. 30. 11. 2022).

[3] MultiMedia LLC. *Modularity*. 2022. URL: https://www.dictionary.com/browse/modularity (cit. 12. 11. 2022).

[4] MDN. *JavaScript modules*. 2022. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules (cit. 23. 11. 2022).

[5] Mishyalt. *Lunar Chat*. 2021. URL: https://github.com/mishyalt/lunar-chat (cit. 30. 11. 2022).

[6] Kamil Mysliwiec. *NestJS*. 2022. URL: https://nestjs.com/ (cit. 01. 12. 2022).

[7] Mohanesh Sridharan. *CommonJS vs AMD vs RequireJS vs ES6 Modules*. 2017. URL: https://medium.com/computed-comparisons/commonjs-vs-amd-vs-requirejs-vs-es6-modules-2e814b114a0b (cit. 12. 11. 2022).