FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Věnná města českých královen - Backend administrační části
Student:	Bc. Dominik Sivák
Vedoucí:	Ing. Jiří Chludil
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

Věnná města českých královen (VMČK) je projekt zabývající se zobrazením historického prostředí ve virtuální a rozšířené realitě.

1. Analyzujte dříve vytvořené administrační rozhraní v rámci projektu VMČK
2. Analyzujte nové funkční a nefunkční požadavky na administrační rozhraní
3. Revidujte a doplňte o nové požadavky existující návrh schvalovacího procesu
4. Implementujte moduly rozhraní (schvalovací proces, statistiky, správa uživatelů v schvalovacím procesu) za použití technologie Node.js
5. Výsledek podrobte integračním testům a realizujte techniky CI/CD

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 19. února 2021 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práca

Věnná města českých královen - Backend administrační části

Bc. Dominik Sivák

Katedra softwarového inženýrství
Vedúci práce: Ing. Jiří Chludil

6. mája 2021

Poďakovanie

Chcel by som sa poďakovať vedúcemu mojej práce, Ing. Jiřímu Chludilovi, za priateľský prístup, skvelé vedenie práce a možnosť zúčastniť sa na projekte. Veľké poďakovanie patrí mojim priateľom, ktorí aj v čiernobielych časoch udržali svet farebným. Na záver sa najviac chcem poďakovať svojej rodine, pretože bez ich neustálej podpory by som sa k tejto, záverečnej, časti štúdia nikdy nedostal.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Prahe 6. mája 2021 . . . . . .

Odkaz na túto prácu

Sivák, Dominik. *Věnná města českých královen - Backend administrační části.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Táto diplomová práca sa zaoberá analýzou, návrhom a implementáciou nového schvaľovacieho procesu v rámci backendu projektu VMCK. Schvaľovací proces určuje postup práce pri vytváraní 3D objektov a vystupujú v ňom užívatelia v rôznych roliach, ktorí môžu navrhnuté modely schvaľovať, zamietnuť či komentovať. V práci sa budem venovať preberaniu, spojazdneniu a zhodnoteniu nedostatkov zdrojového kódu vytvoreného v predchádzajúcich iteráciách projektu v kapitole Analýza. Po nej, v kapitole Návrh, rozoberiem možné prípady implementácie niektorých vylepšení spolu s novými časťami projektu. Kapitola Realizácia poskytne náhľad do výsledného stavu kódu, spojazdnenia CI a obsahuje aj príručku pre ďalšie iterácie projektu.

Klíčová slova schvaľovací proces, backend, node.js, javascript, continuous integration

Abstract

This master's thesis captures analysis, design, and implementation of the new part of the project VMCK – the approval process. This process defines a workflow for the creation of 3D objects. The members of the process, which are assigned a role, can approve, decline, or comment on the created objects. The

thesis begins with an analysis chapter, which focuses on taking over the code base, running the code, and evaluating its shortcomings. I will propose several ways to implement the improvements and the new parts of this project in the next chapter, design. The thesis ends with chapter implementation, which provides an insight into the final state of the code, the process of CI along with a manual for the future iterations of this project.

Keywords approval process, backend, node.js, javascript, continuous integration

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Jedným z výstupov projektu *Věnná města českých královen* (VMCK) je systém poskytujúci zážitok prechádzky po centre mesta v podobe, ako vyzeralo pred niekoľkými storočiami. Užívateľ sa môže pomocou mobilných technológií, virtuálnej a rozšírenej reality (VR, resp. AR) nechať sprevádzať po rôznych trasách a navodiť si tým atmosféru renesancie či romantizmu.

Na tomto projekte už pracovalo niekoľko skupín a jednotlivcov, či už v rámci predmetu *Softwarový tímový projekt*[1], alebo bakalárskej a diplomovej práce, čoho výsledkom boli klientske aplikácie podporujúce rôzne platformy a pokračujúca tvorba backendu. Okrem implementačných prác sa mnoho ľudí podieľalo aj na rešeršných a návrhových prácach, ktoré pre ne položili základ.

Rovnako je to aj v prípade tejto práce, ktorá priamo naväzuje na prácu [1] Dana Vančuru. Primárne zameranie bude na schvaľovací proces v backendovej časti, ktorý navrhol Dan spolu s Ing. Michalom Martínkom. Tento proces špecifikuje istý *workflow*, ktorý musí objekt absolvovať, aby sa dostal z modelovacieho programu tvorcu až priamo pred oči používateľa. Michal navrhoval a implementoval klientskú aplikáciu pre správu schvaľovacieho procesu v práci [2]. Tento návrh schvaľovacieho procesu podrobím analýze, určím nedostatky a zakomponujem nové požiadavky, ktoré následne prejdú návrhovou a implementačnou fázou.

Ako primárne využitie som spomínal zobrazovanie modelov, čo ale nie je jediným cieľom tohoto projektu. Po dokončení implementácie schvaľovacieho procesu sa počíta aj s využitím vo výuke na odovzdávanie úloh.

Táto téma ma zaujala hlavne kvôli môjmu kladnému vzťahu k vývoju backendu. Taktiež ma lákali aj technológie, s ktorými som sa síce ešte nestretol, ale rád túto skutočnosť zmením pri práci na tomto projekte. Pevne dúfam, že táto práca pomôže posunúť projekt VMCK o krok alebo dva bližšie k finálnej verzii.

[1] V slovenčine „tímový“, v češtine „týmový“

Kapitola 1

Cieľ práce

Hlavným cieľom tejto práce je tvorba schvaľovacieho procesu pre backend projektu VMCK. V tomto procese vystupuje niekoľko rolí, ako napríklad modelár, grafik a historik a slúži primárne k stanoveniu určitého postupu práce medzi vytvorením 3D objektu a jeho zavedením do prevádzky – teda zobrazením koncovému užívateľovi. Schvaľovací proces sa skladá z iterácií rôznych druhov, podľa ktorých sú určené požiadavky na schválenie alebo zamietnutie. Tento proces už bol navrhnutý a z časti implementovaný v predchádzajúcich iteráciách tohoto projektu. Odvtedy však pribudli nové požiadavky na proces, ktoré bude nutné do týchto návrhov zakomponovať.

Sekundárnym cieľom práce je posunúť projekt dopredu (s ohľadom na časový rozpočet) aj v iných oblastiach, ako je napríklad prihlasovanie, či zavedenie Continuous Integration (CI) a Continuous Delivery/Deployment (CD).

Táto práca začne kapitolou Analýza, v ktorej rozoberiem existujúci návrh schvaľovacieho procesu a vytýčim nájdené nedostatky. Okrem toho budem analyzovať aktuálny stav kódu, ale aj nové požiadavky, jednak na schvaľovací proces a jednak na backend ako celok.

V kapitole Návrh predstavím rôzne možnosti, ako je možné výstup z predchádzajúcej časti implementovať, spolu s ich kladmi a zápormi. Rovnako uvediem aj časti aktuálneho kódu, ktoré budú musieť byť kvôli novým zmenám upravené či odstránené.

Kapitola Realizácia bude slúžiť na zhrnutie implementovanej časti, popísanie testovania, ukážky rozbehnutia CI & CD a spísanie krátkeho manuálu pre ďalšie iterácie tohoto projektu.

Kapitola 2

Analýza

V tejto kapitole sa zameriam na proces prebratia kódu a naviazanie na práce mojich predchodcov v rámci tohoto projektu. Zároveň podrobím príručku obsiahnutú v práci [1] Bc. Daniela Vančuru „*skúške ohňom*“, keď sa pokúsim rozbehnúť backend Private API VMČK v lokálnom prostredí bez akýchkoľvek predchádzajúcich skúseností s týmto projektom. Na základe tejto skúsenosti vyvodím pozorovania, v ktorých uvediem, ktoré časti backendu obsahujú nedostatky, alebo majú priestor na zlepšenie.

Po úspešnom spustení backendu budem analyzovať návrh serverovej časti pre schvaľovací proces, ktorý vytvoril vo svojej práci [2] Ing. Michal Martínek. Zadaním jeho práce bola frontend časť pre schvaľovací proces ale okrajovo sa venoval aj serverovej časti a poskytol základný návrh zmien, ktoré budú pre tento proces potrebné. Tento návrh taktiež podrobím analýze, sformulujem funkčné a nefunkčné požiadavky pre schvaľovací proces a vyhodnotím nedostatky, ktoré tento návrh obsahuje.

2.1 Prvotné spustenie a použité technológie

Kód je obsiahnutý v privátnom git repozitári na školskom *Gitlabe*. Po získaní prístupu od vedúceho práce som si tento repozitár naklonoval a analýza kódu mohla začať. Projekt používa množstvo technológií a knižníc, ale aspoň v skratke uvediem tie, ktoré sú pre túto prácu dôležité. Tieto technológie už majú svoje pevné a nemenné miesto v projekte, pretože je na nich postavený celý projekt, na ktorom sa už podieľal nemalý počet ľudí. To znamená, že tento zoznam je prakticky nezmenený od práce [1], ale pre úplnosť poskytnem ich krátky popis aj v tejto časti.

Docker je open–source projekt s cieľom *kontajnerizovať* technológie, teda vytvoriť a spustiť kontajner s okresaným operačným systémom (zameraný len na jednoúčelové použitie), čo poskytuje výhodu oproti virtualizovanému OS v podobe nižších pamäťových nárokov. Vývojár tým pá-

dom nemá na starosti stiahnutie tej správnej verzie služby či technológie a taktiež je oslobodený od závislosti na platforme. Všetky informácie pre Docker sú uložené v súbore *docker-compose.yaml* a sú to napríklad *Docker image*, ktorý sa má použiť, konfigurácia portov a vzájomné závislosti jednotlivých komponentov.

Node.js je taktiež open–source softwarový systém navrhnutý pre písanie vysoko škálovateľných internetových aplikácií, hlavne serverov. Za svoju vysokú rýchlosť vďačí asynchrónnym I/O operáciám a taktiež V8 Javascript (ďalej len JS) enginu s JIT (just-in-time) compilerom. Niektorými médiami označený ako revolučný[3], zamiešal kartami na poli backend enginov, kde medzi populárne voľby patrili (a stále patria) Java či PHP.

Express.js je najrozšírenejším frameworkom pre Node.js, ktorý umožňuje jednoduché písanie webových aplikácii pre túto platformu. Teší sa veľkej komunitnej podpore a mnohým rozšíreniam.

Typescript je open–source typová nadstavba pre JS od spoločnosti Microsoft. Uľahčuje orientáciu v kóde, zvyšuje udržateľnosť a zlepšuje ranú detekciu chýb pridaním typovej informácie. Kód sa následne kompiluje do JS.

TypeORM poskytuje relačné mapovanie medzi objektami a tabuľkami v databáze. Má dobrú podporu Typescriptu, podporuje migrácie a necháva výber vzoru medzi *Active Record*[2] a *Data Mapper*[3] na vývojárovi.

JSON Web Token alebo JWT je štandard, ktorý rieši podpisovanie a šifrovanie dát uložených vo formáte JSON. V tomto projekte je použitý pre autentifikáciu užívateľov. Obsahuje Header (hlavičku) s informáciou o použitých algoritmoch, Payload (dáta) – samotné dáta a Signature (podpis), teda kryptografický podpis, ktorým sa overí integrita dát.

PostgreSQL je populárny open–source objektovo–relačný databázový systém v tomto projekte použitý na uloženie väčšiny informácií.

MongoDB je taktiež populárna open–source databázová technológia, tentokrát však NoSQL typu *document store*. V tomto ohľade mi nedá neopraviť kolegu Vančuru v práci [1], kde uviedol, že táto technológia nemá vlastný dotazovací jazyk. Opak je však pravdou a keďže sa tento systém radí medzi *document store*, má jazyk, ktorý umožňuje pristupovať aj ku

[2] Active Record je architektonický návrhový vzor, ktorý je definovaný ako „*Objekt, ktorý obsahuje riadok v databázovej tabuľke alebo pohľade, zapuzdruje prístup k databáze a pridáva doménovú logiku.*“[4]

[3] Data Mapper je architektonická návrhový vzor, ktorý oddeľuje objekt držiaci informácie z DB a logiku, ktorá tieto informácie získava, alebo ich manipuluje.[4]

hodnotám a položkám v nich. Pre všeobecné použitie je však jeho vyjadrovacia sila menšia ako pri tradičných relačných DB, preto je nutné dôraznejšie premyslenie štruktúry dokumentov.

Jest je testovací framework pre JS udržovaný spoločnosťou Facebook. Spolu s nadstavbou *Supertest* je populárnou voľbou vývojárov na testovanie endpointov webových aplikácií.

Všetky tieto a ďalšie technológie sú obsiahnuté v projekte. Už spomenutý Docker uľahčuje spustenie celého prostredia konfiguráciou, ktorá obsahuje kontajnery pre:

- Postgres
- Adminer (správa Postgres DB)
- MongoDB
- Mongo-Client (správa MongoDB)
- Samotná webová aplikácia

Nastavenie prostredia je tak záležitosťou nastavenia Dockeru, úpravy súboru s premennými hodnotami prostredia a spustenia kontajnerov. Môj host operačný systém je Windows 10 v edícii Home. V minulosti Docker nebol kompatibilný s touto edíciou a jediný spôsob, ako rozbehnúť Docker bolo vo prostredníctvom virtuálneho stroja, teda kontajner vo VM v hosťovskom OS.

To sa však zmenilo príchodom WSL (Windows Subsystem for Linux[4]) verzie 2, ktorá umožnila fungovanie Dockeru aj v tomto prostredí. Musím však podotknúť, že upgrade z WSL 1 na WSL 2 priniesol výrazné spomalenie v prístupe k súborom v hosťovskom súborovom systéme z virtualizovaného OS[5]. Táto chyba je už dlhšie nahlásená a oprava zo strany Microsoftu je tiež dlho prisľúbená, ale doposiaľ sa tak nestalo.

Ešte pred zapnutím kontajnerov je nutné vytvoriť súbory *.env* a *.env.test* zo súboru *.env.example*. Example súbor neobsahoval placeholdery jednak na nastavenie pripojenia k Postgres DB a jednak na vytvorenie DB v Postgres kontajneri. Súbor neobsahuje ani informácie, aké kľúče sa používajú a ktoré je vhodné zmeniť. Kontajner s Postgres DB si s tým poradí a vytvorí default užívateľa *postgres@postgres* a default databázu *postgres*, čo ale nie je nikde spomenuté a môže to spôsobiť mierny chaos, pretože toto *connection info* sa používa aj pre konfiguráciu pripojenia *TypeORM*. Navyše takto nespomenutá informácia môže spôsobiť, že produkčný server nebude dostatočne zabezpečený. Po spojazdnení Dockeru som postupoval podľa príručky a všetky kontajnery sa mi podarilo (až na Postgres) bez ťažkostí rozbehnúť.

[4]WSL je vrstva kompatibility umožňujúca natívny beh linuxových spustiteľných súborov vo Windows 10.

2.2 Analýza stavu Private API

V tejto časti sa zameriam na analýzu backendu Private API v stave, v ktorom som ho dostal, čo je výstupom práce D. Vančuru.

2.2.1 Testy

2.2.1.1 Spojazdnenie

Ako prvé som sa zameral na spojazdnenie testov, jednak aby som si overil, či systémy v kontajneroch spolu komunikujú a jednak by to mohlo mať nejakú výpovednú hodnotu o stave projektu. Tento proces však nebol taký jednoduchý, ako sa môže zdať, a hneď na začiatku som narazil na problém, kde neprešiel úspešne ani jeden test. Kolega Vančura síce v *README* spomína, že spustenie všetkých testov naraz spôsobí pád všetkých testov, ale testy neboli úspešné ani keď boli spustené osobitne.

Po dlhom zisťovaní sa ako príčina ukázala metóda *beforeAll*, ktorá zabezpečuje prístup k DB vytvorením pripojenia(*connection*) a zároveň vytvára užívateľa, ktorého token sa potom použije pre autentifikáciu v dotazoch. Problémom bolo, že *test cases*[5] používali v ich dobe neinicializované hodnoty connection a tokenu, čo spôsobilo ich neúspech. Kód však obsahoval JS konštrukty, ktoré majú na starosti synchronizáciu asynchrónnych volaní. Po pátraní na internete som zistil, že je to už nahlásený problém, ktorý obsahuje testovací framework *Jest* a ako obídenie chyby sa uvádzalo použiť iný *runner*, ktorý bude testy spúšťať. Po nasadení *runneru jest-circus* bola chyba odstránená a pripojenie aj token boli v dobe behu testu už inicializované.

Nasledujúce spustenie testov však stále neprinieslo úspech, naopak odhalilo dva nedostatky, ktoré tieto testy majú:

1. Testy pracujú s absolútnym počtom záznamov v DB.
2. Testy vytvárajú nové záznamy, ale nemažú ich.

Tieto nedostatky majú implikácie na nasledujúce dôsledky:

1. Testy je možné spúšťať len na DB, ktorá neobsahuje žiadne záznamy.
2. Testy je možné spustiť len raz, potom musíme vyprázdniť DB.
3. Testy (nenadväzujúce na seba) nie je možné spúšťať paralelne.

Prvý dôsledok priamo súvisí s prvým nedostatkom, keďže v mnohých testoch sa vytvárajú napríklad štyri záznamy a na konci sa kontroluje, či je ich počet v DB rovný štyrom. Ak už predtým existoval minimálne jeden záznam tohoto druhu v DB, test neprejde.

[5] v slovenčine „testovacie situácie“, ale kvôli frekvencii anglického výrazu použijem anglický variant

Tabuľka 2.1: Neúspešne testy po prebratí kódu

Test suite	Test
TextureController	Update texture test–all ok
StructureController	Get list of all structures
Model Controller	list all test
Comment Controller	list all test

Druhý dôsledok súvisí s druhým nedostatkom. Test síce vytvorí štyri záznamy, ale nezmaže ani jeden, čo spôsobí, že v rovnakom príklade bude na konci druhého behu v DB osem záznamov.

Tretí dôsledok súvisí s oboma nedostatkami.

Taktiež poradie vykonávania **súborov** (neplatí pre testy v rámci súboru) s testami nie je deterministické a pokiaľ sa najprv testuje súbor pre entitu *Texture*, ktorá je existenčne závislá na entite *TDObject* a pre testy *Texture* sa vyrobí záznam *TDObject* a v testoch pre *TDObject* sa s týmto záznamom (pochopiteľne; absolútny počet) nepočíta, tak test pre *TDObject* neprejde.

Z dôsledkov sa stávajú *nepríjemnosti*, na ktoré je potrebné myslieť pred každým spustením testov. Odstránenie paralelizácie (*Jest* od základu vykonáva testy paralelne) je triviálne, stačí použiť parameter `--runInBand`. Zvyšok však vyžaduje kompletnú revíziu testov, prípadne iným spôsobom vyriešiť „sviežosť“ databázy pred každým testovaním.

Jedným často aplikovaným riešením je použitie tzv. *in-memory* databázy. Jej výhoda spočíva v jednoduchom nastavení a znovupoužití a hlavne v tom, že záznamy nie sú perzistované na pevný disk, takže testovanie neovplyvní napr. produkčnú DB. Ich nevýhodou je, že nie každý databázový stroj podporuje takýto mód prevádzky a to sa týka práve aj *Postgresu*, ktorý je použitý ako hlavný DB stroj pre tento projekt.

Častou alternatívou v takomto prípade je použitie DB stroja *SQLite*. V tomto prípade však fungovať nebude, keďže *migrácie* (prípadne synchronizovanie entít)[6] používajú konštrukty a prvky SQL dialektu *Postgresu*, ktoré nie sú kompatibilné s *SQLite*. Jedným takýmto príkladom je prvý SQL príkaz v prvej *migrácii* `CreateDatabase`, kde sa definuje *typ* príkazom `CREATE TYPE`, ale *SQLite* typy nepodporuje.

2.2.1.2 Výsledky

Po prispôsobení testovacieho prostredia pripomienkam spomenutých vyššie, sa podarilo spustiť testy s prevažným úspechom. Úspešných bolo **215** z **219** testov, čo znamená, že **štyri** testy neboli úspešné, menovite sú obsiahnuté v tabuľke 2.1.

[6]Princíp migrácií a synchronizácie popíšem v kapitole Návrh.

Po dôkladnejšej analýze som prišiel na to, že tri (v tabuľke 2.1 riadky 1, 3 a 4) z týchto štyroch testov sú neúspešné, kvôli chýbajúcej implementácii endpointov. Endpointy vracajú HTTP odpoveď s kódom 501, ale test očakáva hodnotu 200. Tieto testy nemajú žiadnu výpovednú hodnotu a dá sa polemizovať dokonca aj o ich negatívnom prínose v zmysle, že vývojový tím si na padajúce testy „zvykne“. Zároveň po implementácii endpointu nebudú klásť klamlivý dojem, že testy na tento endpoint už existujú (nič iné, ako HTTP kód 200 sa v týchto testoch nekontroluje). Tieto testy nateraz zakážem.

Test, ktorý „netrpí“ na prvý pohľad nedokončenou implementáciou (riadok 2 v tabuľke 2.1) očakáva, že sú v DB vytvorené 4 záznamy, ale reálna hodnota je 5. Po analýze testov môžem potvrdiť, že by mali byť vytvorené práve štyri záznamy, čo znamená, že chyba bude inde a konkrétne pri už spomenutej absolútnej početnosti, teda pred týmto *Test suite* sa vyrobí pomocný záznam *Structure*, s ktorým tento *Test suite* nepočíta. Jednoduchou opravou je teda buď vymazať všetky záznamy v tabuľke pred začatím tohoto *suite*, alebo si poznamenať počet záznamov a kontrolovať len relatívny prírastok. Vzhľadom na deštruktívnu podobu prvej alternatívy sa prikloním k druhej možnosti.

Po spomínaných opravách sú všetky splnené (tri preskočené) a to je dobrá východisková situácia pred akoukoľvek úpravou funkcionality.

Pre zvýšenie prehľadnosti som použil rozšírenie pre *Jest* s názvom *jest-html-reporter*, ktoré, ako je z názvu zrejmé, vygeneruje ľudsky dobre čitateľný HTML súbor s prehľadom a výsledkami testov. Na priloženom médiu pripájam pod názvom `test-report-before.html` a `test-report-before-fixed.html` v zložke `tests` výsledky po prevzatí kódu resp. po vykonaní úprav testov.

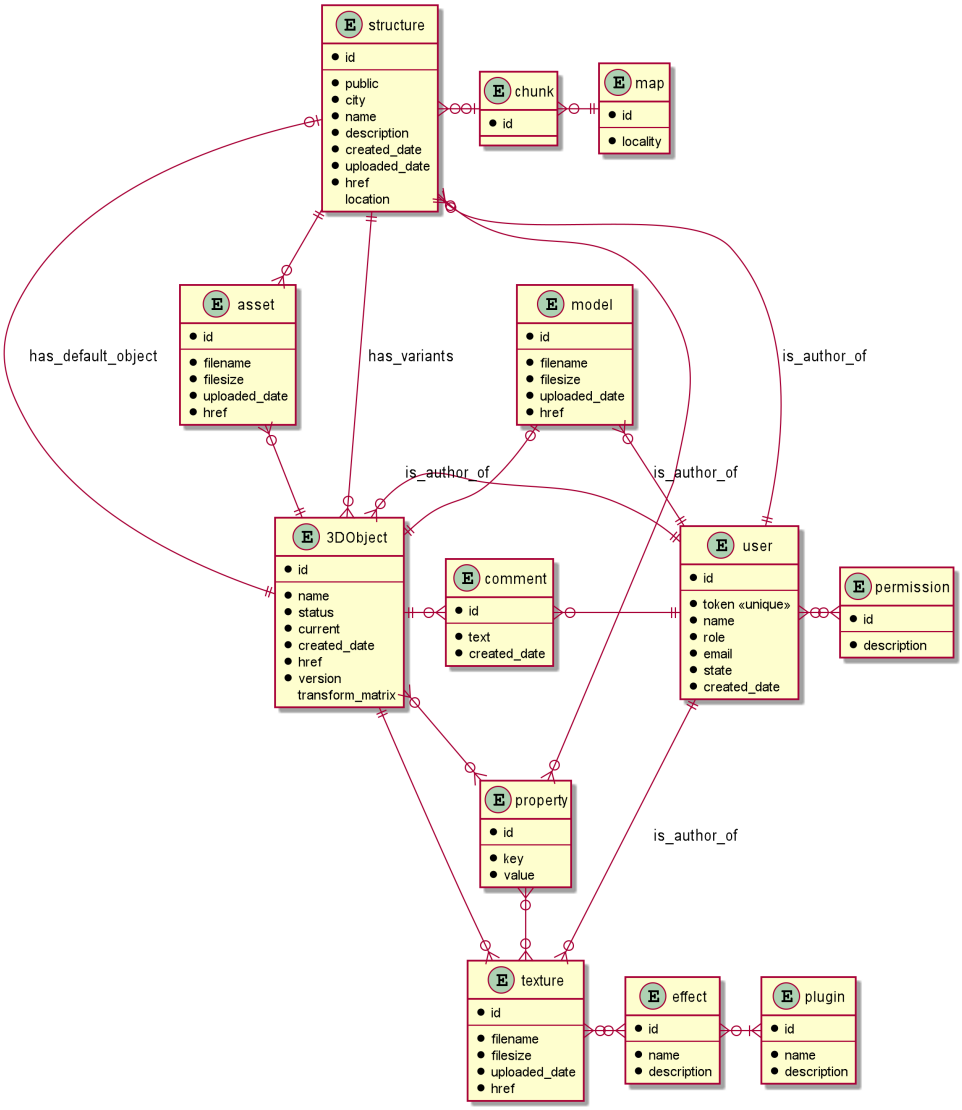
2.2.2 Databáza a dátový model

V tejto časti budem analyzovať model uloženia dát, ktorý revidoval vo svojej práci kolega Vančura. Výstupy jeho predchodcov analyzoval a vykonal potrebné úpravy jednak pre schvaľovací proces, ku ktorému sa dostanem neskôr, a jednak upravil viaceré nezrovnalosti, ktoré sa preniesli do projektu z návrhov tímov z predmetu *Softwarové inžinierstvo*. Tento model je znázornený grafom na obrázku 2.1.

Primárny kľúč entity *User* Ako primárny kľúč tejto entity slúži atribút *username*. Tento atribút je možné v rámci API meniť a zároveň slúži ako cudzí kľúč v tabuľkách, ktoré majú väzbu práve na túto tabuľku. Takéto použitie je *krajne nevhodné* a pre správne fungovanie cudzích kľúčov bude musieť byť vytvorený nový nemenný, identifikujúci atribút v tejto tabuľke.

2.3 Schvaľovací proces

Táto sekcia bude zameraná hlavne na časť, ktorá je témou tejto diplomovej práce a tou je už spomínaný schvaľovací proces. Na jeho návrhu sa podieľali



Obr. 2.1: Navrhnutý dátový model z práce [1] v notácii „Information Engineering“ (vygenerované nástrojom plantuml)

spoločne kolega Vančura a taktiež kolega Martínek, ktorý, ako už bolo spomenuté, mal vo svojej diplomovej práci [2] na starosti administračné rozhranie, teda webový frontend nielen (ale v značnej miere) ku schvaľovaciemu procesu.

Pre úplnosť popíšem, ako tento proces navrhol kolega Martínek z hľadiska persón. Najprv predstavím užívateľské role, ktoré v tomto procese budú vystupovať:

Historik má na starosti vytváranie záznamov, prípravu a predanie materiálov ku modelovaniu a taktiež z hľadiska historickej správnosti posudzuje modely v schvaľovacom procese.

Grafik schvaľuje modely z hľadiska grafickej správnosti, generovania a schvaľovania variantov základného modelu.

Modelár vytvára podľa historickej predlohy model a má možnosť vyjadriť sa k zadaniu.

2.3.1 Fáza modelu

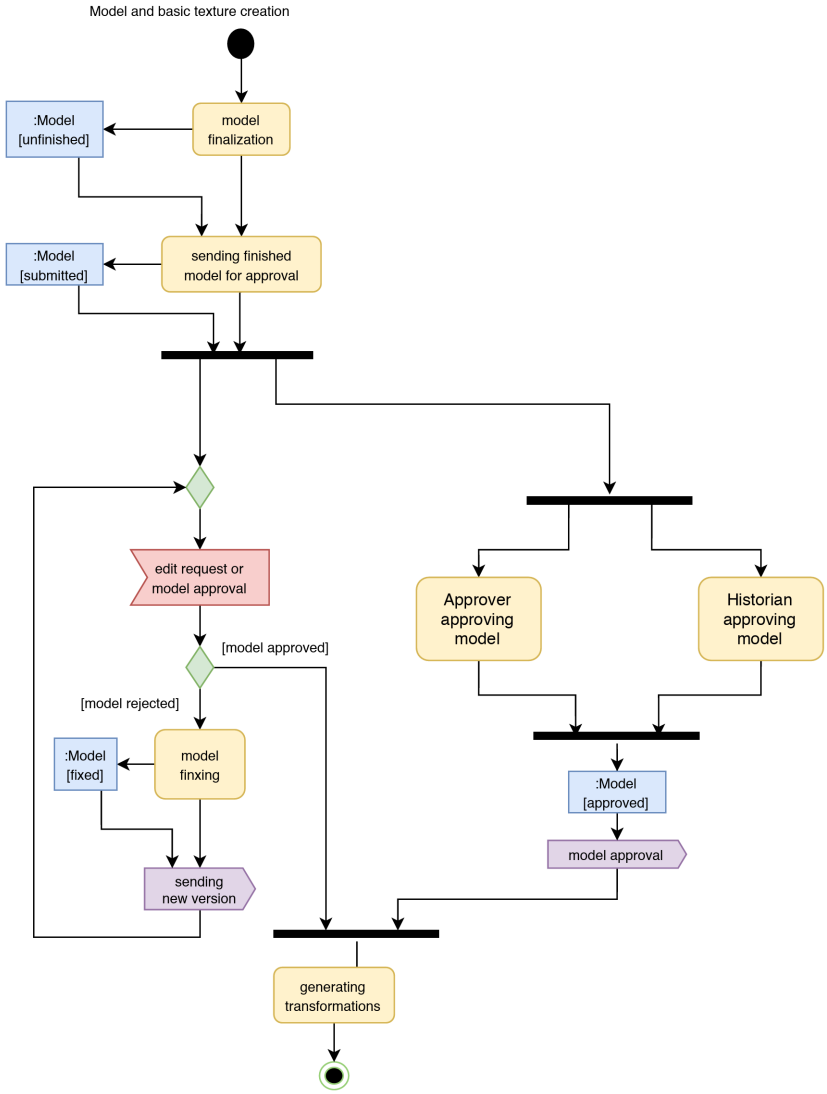
V prvom kroku vytvorí *Historik* entitu *Structure* so všetkými informáciami pre vytvorenie modelu. Pridá rôzne metadata, napr. kde sa artefakt nachádza a dodá všetky potrebné relevantné podklady ku modelovaniu. Týmto vytvára *zadanie*, ktoré spúšťa celý proces. Akonáhle je spokojný so zadaním, prejde sa do ďalšieho kroku a tým je modelovanie. *Modelár* má teraz za úlohu podľa zadania vytvoriť 3D model. Do systému ma možnosť nahrať niekoľko verzií a keď už je s výsledkom spokojný, odošle konkrétnu verziu na schválenie. Tento model musí schváliť jednak *Historik*, ktorý kontroluje historickú vernosť modelu a jednak *Grafik*, ktorý kontroluje technickú správnosť. Teda na postup modelu do ďalšej fáze procesu je potrebný súhlas oboch rolí. Ak majú jeden či druhý zo svojej strany výhrady, môžu ho vrátiť *Modelárovi* spolu s pripomienkami. V tejto chvíli sa čaká na *Modelára*, aby upravil model a znovu ho poslal na schválenie. Pre redukovanie pracnosti procesu nie je nutné, aby bolo schválenie znovu vyžadované od role, ktorá už tento proces schválila.

Táto časť procesu je znázornená v diagramoch na obrázkoch 2.2 a 2.3, ktoré pripravil kolega Vančura.

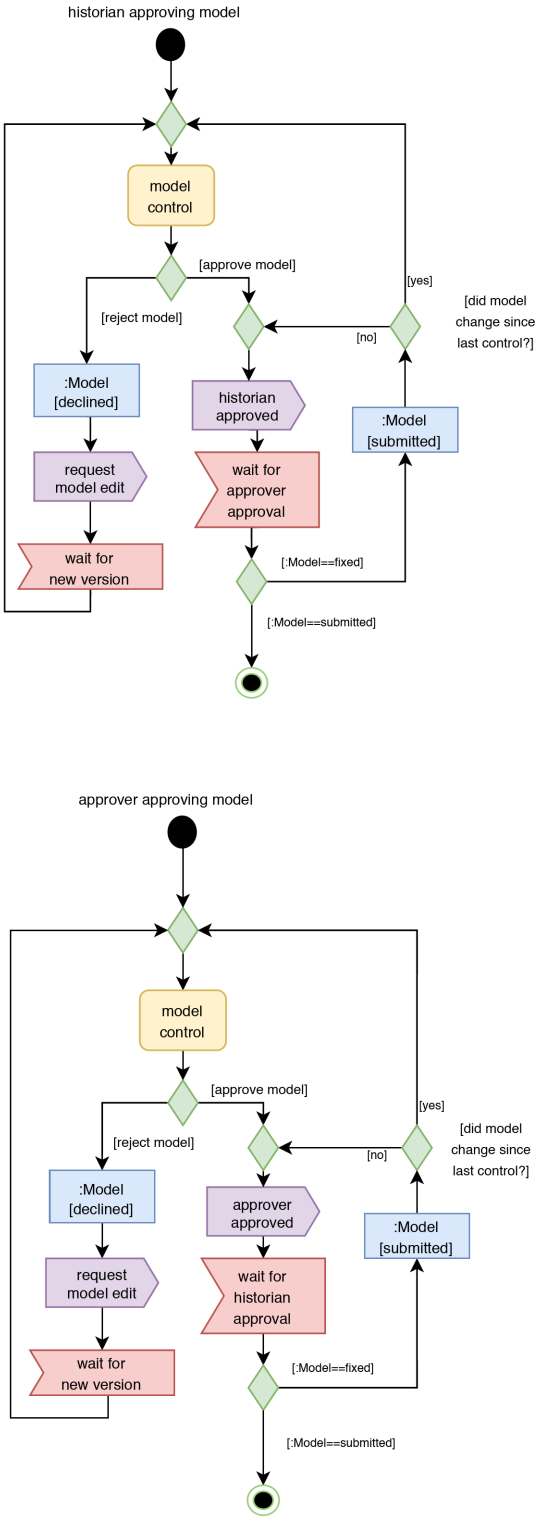
Týmto sa končí prvá fáza schválenia a model prechádza do stavu *zverejnený*.

2.3.2 Fáza variantov

Druhá fáza sa týka generovania variantov. Medzi požiadavky na projekt práve patrí aj zobrazovanie rôznych obmien jedného objektu. Tieto obmeny zahŕňajú rôzne poveternostné podmienky, ročné obdobia, úrovne zostarnutia v priebehu času a podobne. Všetky tieto varianty majú prispieť k vizuálnej vernosti, vďaka ktorej sa užívateľovi v zime nezobrazí dom, ktorého parapetné dosky sú plné



Obr. 2.2: Základný schvaľovací proces I (z práce [1])



Obr. 2.3: Základný schvaľovací proces II (z práce [1])

rôznofarebných kvetín. Tvorba týchto variantov je zamýšľaná ako poloautomatická za pomoci transformačného systému využívajúceho vyhotovené pluginy do programu *Blender*. Touto časťou sa zaoberali Denisa Sůvová a Michal Zajíc v bakalárskych prácach [6], respektíve [7]. *Grafik* skúsi vygenerovať pomocou týchto transformácií variant podľa jeho predstáv. Po dokončení transformácie, ktorá môže trvať aj niekoľko hodín, si *Grafik* zobrazí výsledok a v prípade, že je spokojný, je tento variant pripravený pre užívateľov a proces je ukončený. V opačnom prípade môže parametre transformácie (ktoré majú svoje základné hodnoty, ale je možné ich meniť) upraviť a ak by ani toto neviedlo ku požadovanému výsledku, môže zadať *Modelárovi* zadanie na manuálne vyhotovenie variantu. V tomto prípade sa znovu začína schvaľovací proces podobný k tomu, ako bolo popísané vyššie s tým rozdielom, že schvaľovateľ je v tomto prípade už len *Grafik*.

Táto fáza procesu je znázornená v diagrame na obrázku 2.4, ktorý pripravil kolega Martínek.

2.3.3 Aktuálny stav schvaľovacieho procesu

Po rozobratí procesu z pohľadu persón sa teraz zameriam na implementáciu, ktorej základný kameň položil vo svojej práci kolega Vančura. Vzhľadom na fakt, že plná implementácia bola mimo rozsah jeho práce, mám v počiatočnej fáze k dispozícii návrh, akým spôsobom sa budú dáta o procese ukladať (spomenutý vyššie a naznačený na obrázku 2.1) a základné úpravy v API realizujúce tento proces.

V súbore `TDObjectController` sú implementované endpointy na schvaľovanie rôznymi rolami popísanými vyššie a taktiež *stubs*[7] endpointov pre CRUD operácie s komentármi ku procesu. Ku samotnému schvaľovaniu je napísaných aj niekoľko testov. V dátovom modeli na obrázku 2.1 môžeme označiť ako hlavné body implementácie procesu entity *3DObject*, *Comment*, *Structure* a *Texture*. Kolega Vančura síce navrhol, ako sa schvaľovanie premietne do dátového modelu, ale v samotnom grafe už tieto návrhy nezohľadnil. Zmeny sa týkajú hlavne entity *3DObject*, ktorá v momentálnom stave obsahuje naviac (medzi inými) štyri polia:

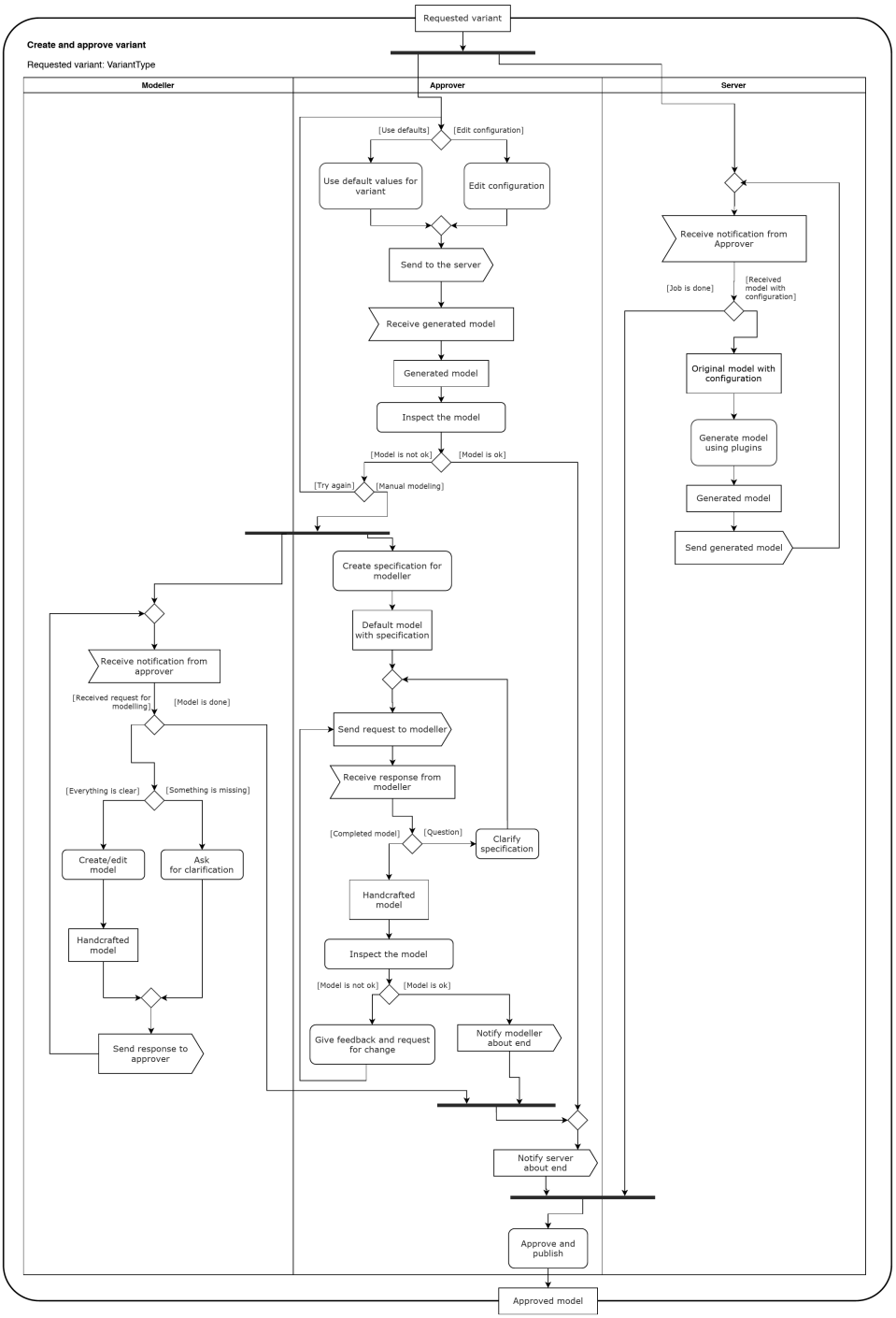
is_approved_by_approver príznak určujúci schválenie modelu *Grafikom*

is_approved_by_historian príznak určujúci schválenie modelu *Historikom*

approver_approving_username užívateľské meno schvaľujúceho *Grafika*

historian_approving_username užívateľské meno schvaľujúceho *Historika*

[7] stub (v preklade „útržok“) je vygenerované telo funkcie, ktoré zväčša na túto skutočnosť aj upozorňuje (v tomto prípade nevykoná žiadnu akciu a vráti HTTP status 501 s textom, ktorý informuje, že daná funkcionalita nie je implementovaná).



Obr. 2.4: Schvaľovací proces jednotlivého variantu (z práce [2])

Nad týmito položkami je postavený aktuálny (**veľmi** obmedzený) schvaľovací proces.

2.3.4 Nedostatky návrhu schvaľovacieho procesu

Túto časť budem venovať pozorovaniam a záverom ku ktorým som došiel pri analýze súčasného stavu.

Oddelenie Ako najväčší problem považujem nedostatočné **oddelenie procesu od entity *3DObject***. Už len to, že záznam obsahuje dáta, ktoré sú relevantné len na začiatku životného cyklu entity (životný cyklus plynie od vytvorenia, cez ostrú prevádzku až po nahradenie novou verziou) značí, že údaje o schvaľovacom procese by sa mohli oddeliť. Zároveň ak by sa v budúcnosti tento proces nejak menil, nemusí to nutne znamenať zásah do entity a všetkých záznamov v tabuľke, čo znamená ľahšiu aplikáciu zmien.

História Ďalším nedostatkom je chýbajúce **ukladanie histórie**. V aktuálnom modeli je priestor len pre jednu hodnotu určujúcu schválenie pre *Grafika* a jednu pre *Historika* a v rovnakom počte to platí aj pre identifikáciu schvaľovateľa. Ak by sme mali záujem zobraziť si históriu procesu, bolo by toto nepostačujúce.

Požiadanie Taktiež chýba implementácia **spätnej väzby** (v rámci schvaľovania) a **požiadania** o premodelovanie objektu. Spätná väzba je myslená ako prijatie upozornenia na zmenu v procese a podobne.

Rozlíšenie Rovnako chýba **rozlíšenie** medzi schvaľovaním v prvej fáze (potrebný súhlas oboch rolí) a vo fáze druhej (potrebný len súhlas *Grafika*).

Komentáre V návrhu sa počíta len s jednoduchým typom komentárov (atribút text). Kolega Vančura však spomínal aj iný typ komentárov ako víziu do budúcnosti, a to napríklad bodové, ktoré sa viažu na určitú súradnicu.

Sémantika Miernym sémantickým nedostatkom je nazvanie *Grafika* v implementácii ako „*approver*“. Toto označenie môže pôsobiť mylne a mätúco.

2.3.5 Možné vylepšenia schvaľovacieho procesu

V tejto časti rozoberiem funkcionalitu, ktorá by mohla byť pridaná (a je súčasťou zadania tejto práce), ale v návrhu a implementácii v prácach kolegov Vančuru a Martinka sa nenachádza.

Správa užívateľov Vítaný by bol systém správy užívateľov s prístupom k schvaľovaniu. V reálnej prevádzke by to *nemusela* byť ideálna situácia, keby

napr. *Historik* z Prahy schvaľoval historickú presnosť objektov na druhom konci republiky. Použitie slov v tejto vete bolo skôr opatrné, aby sa pridaním správy neobmedzila funkcionalita, ale naopak dala užívateľovi užitočný nástroj. Túto správu musí byť možné editovať aj v priebehu spracovania, kedy napríklad môže nastať situácia, že jeden alebo viacerí členovia už nie sú schopní ďalej na projekte pracovať a je nutné do ich rolí nasadiť iných užívateľov.

Štatistiky Rovnako je v zadaní požiadavka na implementáciu modulu štatistík v rámci tohoto procesu. Tieto štatistiky môžu mať rôzne prípady použitia, či už zistenie TOP prispievateľov, alebo modelára, ktorému bolo percentuálne zamietnutých najviac modelov, či objekt, ktorý mal najdlhší schvaľovací proces. Konkrétne prípady už budú záležať hlavne na frontende podľa toho, aké grafy sa budú užívateľom zobrazovať. Keďže, ako som spomínal, s týmito vylepšeniami sa v pôvodnom návrhu backendu a frontendu nepočítalo, reálny produkt pre koncového užívateľa bude cieľom až nasledujúcich bakalárskych, diplomových či tímových prác.

2.4 Analýza zvyšných častí backendu

Po analýze schvaľovacieho procesu, ktorý je hlavnou témou tejto práce, sa v nasledujúcej časti zameriam na aktuálny stav zvyšných častí a modulov backendu.

2.4.1 Plány do budúcna

Na niektoré nedostatky upozornil už kolega Vančura a niektoré z nich opravil. Ku niektorým sa však už nestihol dostať (kvôli tomu, že boli nad rámec jeho práce) a pre úplnosť ich uvediem aj tu. Body, ktoré sú relevantné pre túto prácu doplním o svoj komentár, ostatné len veľmi krátko popíšem, prípadne vymenujem.

Toto sú nedostatky alebo vízie do budúcna, ktoré na konci kapitoly *Návrh* uviedol v práci D. Vančura:

Management užívateľov a prístupových práv V projekte momentálne existuje len veľmi základná správa užívateľov. Užívateľ má v DB tabuľke uložený status (napr. *nový* alebo *schválený*) a priradenú rolu z viacerých možných hodnôt (napr. *Historik*, *Grafik* alebo *Modelár*). Z tohoto už na prvý pohľad plynie nedostatok v podobe možnosti zaujať len jednu rolu. Zároveň pre jemnejšiu (detailnejšiu) správu užívateľov v rámci schvaľovacieho procesu, o ktorej som sa zmienil vyššie, je potrebné robustnejšie riešenie. Nedostatkom v tomto bode je aj fakt, že existuje len jediná možnosť autentifikácie, a to cez JWT token, ktorý je napevno uložený v DB

od vytvorenia užívateľa. Kolega Vančura navrhoval prihlásenie pomocou Google účtu, či SSH kľúča.

Transformácia 3D objektov generovaných na serveri Tento bod priamo súvisí so schvaľovacím procesom, konkrétne s iteráciami v druhej fáze. V nich sa primárne počíta s automatickou transformáciou pomocou Blender pluginov, kde užívateľ zadá v rozhraní konfiguráciu, ktorá spustí generovací proces na pozadí. V schvaľovacom procese potom užívateľ buď schváli alebo neschváli tento výsledok.

Webové rozhranie pre schvaľovací proces Toto rozhranie z veľkej časti navrhol vo svojej práci [2] Ing. Michal Martínek. V čase písania tejto práce v implementácii pokračuje vo svojej práci [8] kolega Pavel Antoš.

Databáza lokalizačných informácií Týka sa upresnenia polohy zariadenia pre AR a VR pomocou iných informácií než je len GPS.

Load balancing a synchronizácia databáz V budúcnosti sa plánuje využiť replikácia databáz s cieľom minimalizácie zaťaženia.

Vyhľadávanie Tvorba optimálneho vyhľadávania na základe filtrov, ako autorstvo či status, bola prenechaná na ďalšie iterácie projektu.

Public API Táto časť API je zamýšľaná ako spojenie podmnožiny funkcionality Private API a častí, ktoré súvisia s užívateľskou časťou, ako napríklad zber spätnej väzby, či navigácia po meste.

Feedback API Je podmnožina Public API venujúca sa spätnej väzbe. Zamýšľa sa prepojenie s trackovacím SW (napríklad Redmine), ktorý by automaticky priraďoval *issues* k 3D objektom.

Navigácia S implementáciou tejto časti sa počíta až v záverečných častiach projektu a zameriava sa na tvorbu a prípravu trás, po ktorých sa užívateľ dostane k budovám, ktoré chce vidieť.

Optimalizácia mobilných zariadení Ako už bolo spomenuté, medzi cieľové platformy patria aj smartfóny, u ktorých je však rozpätie výkonu veľmi veľké. Z tohoto dôvodu sa zamýšľa možnosť ukladať a načítať optimálne nastavenia zobrazenia a vykresľovania pre konkrétne modely smartfónov.

2.4.2 Prihlasovanie

Ako už bolo spomenuté, tento projekt obsahuje len veľmi základné možnosti autentifikácie. Pri vytváraní užívateľa sa vytvorí JWT token, ktorý obsahuje len jeho identifikátor. Tento token je tým pádom počas celej doby existencie užívateľa v systéme nemenný, čo predstavuje bezpečnostné riziko. Taktiež je

nepraktické a potenciálne nebezpečné predanie tohoto tokenu užívateľovi. Kolega Vančura sa vo svojej práci niekoľkými vetami venoval aj alternatívnym možnostiam prihlásenia (príp. registrácie), medzi ktorými uviedol prihlasovanie pomocou Googlu, či SSH kľúča.

Prvý spôsob je použitie autorizačnej autority tretej strany, ako napríklad už spomenutý Google. Pri použití tohoto spôsobu sa prenáša zodpovednosť správy hesiel práve na autoritu, čo znamená, že backend nemusí riešiť ukladanie či zmenu hesiel. Výhodou pre užívateľa je tiež menší počet účtov, a teda aj menší počet hesiel na zapamätanie si (v ideálnom a bezpečnom svete, kde koncový užívateľ používa pre každú službu iné heslo). Oproti použitiu SSH kľúča je táto možnosť vhodná aj pre technicky menej zdatných užívateľov. Často používané autority vo webových službách z vlastného pozorovania sú napríklad Google, Twitter, Microsoft a v technickej sfére GitHub, GitLab či Discord.

Použitie SSH kľúča na prihlasovanie považujem v tomto projekte za zbytočne silný mechanizmus. V praxi by to znamenalo šifrovanie celého, časti alebo hashu payloadu každej požiadavky na server. Oproti kontrole zhody tokenu to prináša zvýšené výpočtové nároky jednak na strane klienta a jednak aj na strane backendu. Taktiež vzniká problém s generovaním takéhoto kľúča aj užívateľmi, ktorí v tejto oblasti nemajú žiadnu prax.

OAuth Pre vyššie uvedené pre a proti oboch variantov sa prikláňam k prvej možnosti, takže použitie autorizačnej autority tretej strany. Všetky vyššie uvedené príklady (zistené z dokumentácií) a mnoho ďalších funguje na protokole OAuth. OAuth je autorizačný protokol, ktorý dáva možnosť delegovať overenie užívateľa na dôveryhodné služby a následne autorizuje webové aplikácie k prístupu k obmedzenému užívateľskému profilu [9]. V praxi to znamená nasledujúci postup:

1. Užívateľ chce pristúpiť k zdroju pomocou prehliadača
2. Užívateľ nie je prihlásený, je presmerovaný na prihlasovaciu stránku poskytovateľa
3. Užívateľ sa autentifikuje a umožní (alebo neumožní) webovej službe získať základné informácie o jeho profile
4. V kladnom prípade sa užívateľovi (prehliadaču) vráti token spolu s profilom, ktoré sú podpísané autoritou a prehliadačom zaslané na webovú službu
5. Webová služba overí podpis a prihlási používateľa

Sekvenčný diagram pre konkrétnu vybranú tretiu stranu bude obsiahnutý v kapitole Návrh. Keďže všetky autority používajú rovnaký protokol a ich použi-

Tabuľka 2.2: Porovnanie počtu užívateľov autorizačných autorít

Služba	Počet aktívnych užívateľov
Google	1,5 mld (2019) [10]
Microsoft (365)	200 mil (2020) [11]
Twitter	353 mil (2020) [12]
GitHub	56 mil (2021) [13]
GitLab	30+ mil (2021) [14]
Discord	140 mil (2021) [15]

tie sa vzájomne nevylučuje, jediná metrika, ktorá mi napadla, určujúca výber služby je počet užívateľov služby. Porovnanie je uvedené v tabuľke 2.2.

Z nej vychádza ako najlepšia voľba podľa stanovenej metriky Google. Počet užívateľov je obrovský a aj z vlastného pozorovania môžem usúdiť, že nepoznám človeka, ktorý by nemal založený Google účet. V ďalších častiach sa teda zameriam na implementáciu nového spôsobu prihlásenia – cez službu Google.

2.4.3 Generovanie dát

Backend taktiež neobsahuje možnosť vygenerovať rôzne data v DB, čo je pre raný vývoj projektu (z mojich skúseností) veľmi dôležité. Medzi prínosy patrí napríklad možné uľahčenie vývoju užívateľských rozhraní, kde je lepšie pracovať s väčším množstvom rôznorodých dát, než napríklad s užívateľmi s menami *Test User 1* až *Test User 50*. Ďalším faktorom je častá prvotná neskúsenosť s projektom, keďže sa zadáva hlavne v rámci bakalárskych či diplomových prác a zoznámenie s projektom trvá nejaký čas. Možnosť zobraziť si a skúmať data v tabuľkách a v odpovediach endpointov je prínosnejšie, než skúmanie prázdnych tabuliek a odpovedí. Generovanie môže tiež pomôcť pri testovaní výkonu pri väčšom počte záznamov v DB (nie je problém vygenerovať tisícky, či milióny záznamov).

2.5 Funkčné a nefunkčné požiadavky

Na základe vyššie spomenutých sekcií, predchádzajúcich prác a konzultácie s vedúcim práce stanovujem nasledujúce funkčné a nefunkčné požiadavky na tento projekt:

2.5.1 Funkčné požiadavky

F1: Management 3D objektov vrátane modelov a textúr: API umožní užívateľovi kompletné CRUD (Crete, Read, Update, Delete) operácie

nad spomenutými tabuľkami.

Splnené pri preberaní projektu.

F1.1: Verzovanie 3D Objektov: API umožní užívateľovi zistiť si celú históriu daného 3D objektu, a to jednak staršie a jednak aj novšie verzie, ďalej vytvárať nové verzie daného objektu, či naopak ich mazať alebo upravovať.

Splnené pri preberaní projektu.

F1.2: Ukladanie rôznych podôb 3D objektov: API musí byť schopné uložiť jeden 3D objekt v mnohých podobách v závislosti na vytvorených textúrach, úrovniach detailov a podobne.

Splnené pri preberaní projektu.

F2: Management užívateľov: Užívateľovi bude umožnené sa cez API registrovať. Po registrácii bude užívateľ čakať na schválenie účtu administrátorom a od toho momentu bude mať užívateľ prístup k dátam podľa jeho pridelených prístupových práv.

Nesplnené pri preberaní projektu: registrácia užívateľa nie je implementovaná.

F2.1: Prístupové práva a užívateľské role: Užívateľovi budú v rámci API pridelené prístupové práva a užívateľské role. Tieto dáta môže užívateľovi zmeniť len administrátor.

Nesplnené pri preberaní projektu: užívateľ môže mať maximálne jednu priradenú rolu.

F3: Properties a ich management: Všetky stĺpce, podľa ktorých sa predpokladá vyhľadávanie nad tabuľkami, z nich budú odobraté a premenené na záznamy v tabuľke *Properties*. Opäť bude vytvorená kompletná obsluha všetkých CRUD požiadavkov nad danou tabuľkou

Nesplnené pri preberaní projektu: endpointy nie sú implementované.

F3.1: Podpora tagovania 3D objektov: API umožní užívateľom pridávať k jednotlivým 3D objektom rôzne tagy. Tie budú pre každý záznam uložené vo forme *properties* v rovnomennej tabuľke.

Nesplnené pri preberaní projektu: endpointy nepracujú s tagmi.

F4: Filtrovanie a vyhľadávanie: API umožní užívateľovi vyhľadávanie 3D objektov, textúr a podobne nad stĺpcami v daných tabuľkách. Pokiaľ nie sú v GET requeste uvedené vyhľadávacie parametre, vráti API užívateľovi zoznam všetkých modelov, ktorých je autorom.

Čiastočne splnené: je možné filtrovať modely podľa autora.

F4.1: Vyhľadávanie modelov podľa geografie, počasia, atď: Všetky tieto informácie budú pre záznamy v DB ukladané vo forme tagov a API bude podľa nich umožňovať vyhľadávanie.

Nesplnené pri preberaní projektu: endpointy nepracujú s tagmi.

F5: Schvaľovanie modelov: *(upravené)* Nad API bude vytvorený schvaľovací proces 3D objektov. Ten bude pozostávať z iterácií, ktorej prislúcha jeden objekt a túto iteráciu budú schvaľovať užívatelia v roli *historik* a *grafik*. Pre iteráciu vo fáze *model* je vyžadované schválenie oboch rolí, vo fáze *variant* len grafika.

F5.1: Správa užívateľov v schvaľovacom procese: *(nové)* Pre schvaľovací proces bude možné nastaviť jeho členov, ktorí budú mať prístup k iteráciám a komentárom. Užívateľ môže byť členom v rôznych roliach, na základe čoho sa odvíja aj jeho právomoc k vykonaniu rôznych akcií v rámci procesu.

F5.2: Komentáre v schvaľovacom procese: *(nové)* API umožní členom schvaľovacieho procesu komentovať iterácie za účelom odovzdania spätnej väzby. Komentáre môžu mať rôzny typ. V tejto práci sa počíta s jednoduchými komentármi (len text) a bodovými komentármi (text + súradnice v rámci objektu).

F5.3: Štatistiky schvaľovacieho procesu: *(nové)* Pre každý schvaľovací proces bude možné získať jeho štatistiky.

F6: Alternatívne možnosti prihlásenia: *(nové)* Užívateľom bude umožnené prihlásenie sa do aplikácie pomocou účtov tretích strán. V prípade, že užívateľ v systéme neexistuje, je mu užívateľský účet vytvorený. V tejto práci sa počíta s prihlásením sa cez účet Google.

2.5.2 Nefunkčné požiadavky

NF1: Minimalizácia využitia mobilných dát: API bude vytvorené s ohľadom na veľkú pamäťovú efektivitu z pohľadu koncového užívateľa.

NF2: Optimalizácia ukladaných dát: Ukladanie dát v DB musí byť implementované s ohľadom na minimalizáciu využitia úložného priestoru. Za predpokladu rovnakých dát v rôznych verziách 3D objektov bude šetrené miesto v databáze technikami počítanej referencie a *copy-on-write*.

NF3: Minimalizácia odozvy serveru: Štruktúra dát bude v práci zvolená tak, aby odozva serveru bola čo najmenšia, a užívateľ tak na ich načítanie nečakal dlho.

NF3.1: Rozšíriteľnosť, replikácia a load balancing: Systém musí byť dobre rozšíriteľný, zreplikovateľný a synchronizovateľný, aby umožnil rozdelenie záťaže na niekoľko serverov pomocou technológie load balancing, a tým znížil odozvu serveru na minimum.

NF4: Technológie: Práca bude vypracovaná použitím technológii spomenutých v časti 2.1.

NF5: Dokumentácia: Pre aktuálny stav API bude v štandarde OpenAPI vytvorená dokumentácia a kód bude dokumentovaný.

NF6: Continuous Integration a Continuous Deployment: *(nové)* Pre API bude spojazdnené automatické zostavenie, testovanie a nasadenie funkčného kódu na server.

Kapitola 3

Návrh

V tejto kapitole sa budem venovať návrhu schvaľovacieho procesu, ktorý bude opravovať nedostatky predošlej implementácie spomínané v časti 2.3.4 a implementovať nové požiadavky na túto funkcionalitu analyzované v časti 2.3.5.

Okrem schvaľovacieho procesu navrhnem rôzne vylepšenia backendovej časti a podmnožinu z nich (s ohľadom na časový rozpočet) implementujem.

3.1 Schvaľovací proces

Primárnou témou tejto práce je implementovať schvaľovací proces na serverovej časti. Ako už bolo spomenuté, základná implementácia už existuje, ale tá v praxi nepostačuje (analyzoval som v časti 2.3.3).

Aby bolo možné opraviť všetky nedostatky a implementovať vylepšenia, je nutné zmeniť predošlý návrh, kde schvaľovací proces je súčasťou záznamu objektu (`TDObject`, viď obrázok 2.1). Schvaľovací proces by mala byť osobitná entita s **väzbou** na objekt, ktorý je predmetom procesu. Táto nová entita, ktorú nazvem **ApprovalProcess**, bude ďalej združovať iterácie procesu v prvej fáze (**ApprovalIteration**).

Prvá fáza Iterácie prvej fázy potom budú obsahovať jednotlivé modely, ktoré boli vygenerované *Modelárom*. Túto iteráciu budú následne schvaľovať *Historik* a *Grafik*. Teoreticky by schválenie stačilo na úrovni procesu, ale kontrola, či proces má iteráciu schválenú jednou či druhou schvaľovacou entitou, nie je výpočtovo náročná a považujem to za flexibilnejšie riešenie.

Druhá fáza Druhá fáza je komplikovanejšia vzhľadom k viac heterogénnej povahe jej iterácií. Tentokrát sa na *ApprovalProcess* bude viazať niekoľko záznamov **ApprovalVariant**. Tá zastrešuje jeden variant a obsahuje informácie napr. o ročnom období alebo počasí. V rámci jedného variantu bude znovu niekoľko iterácií, kde každá sa bude viazať na jeden model. Vzhľadom

na zamýšľané poloautomatické generovanie variantov bude táto entita obsahovať informácie o použitej transformácii. Medzi požiadavky patrí aj možnosť požiadania *Modelára* o ručné vyhotovenie variantu v prípade, že *Grafik* nie je s vygenerovaným produktom spokojný. Tu prichádza na rad spomínaná heterogénnosť. Iterácia teda môže pozostávať buď z poloautomatického generovania na strane serveru, alebo požiadania *Modelára* o manuálne vyhotovenie variantu. V oboch prípadoch v čase vytvárania novej iterácie nebude model dostupný a doplní sa až neskôr, čo je ďalší rozdiel oproti iterácii v prvej fáze.

3.1.1 Štatistiky

Aby bolo možné udržiavať štatistiky, ktoré sú súčasťou požiadaviek, je nutné si uchovávať tiež údaje o každej vykonanej akcii v rámci procesu. Tieto akcie sa budú ukladať do novej entity **ApprovalAction**, ktorá bude mať jednak povinnú väzbu na proces *ApprovalProcess* a jednak nepovinnú na užívateľa *User*, ktorá bude v role pôvodcu, ktorým ale nemusí nutne byť užívateľ. Tabuľka okrem toho obsahuje aj *enum* určujúci typ akcie a dátový typ *json*, ktorý slúži na uloženie dát akcie. Záznamy tejto entity môžu obsahovať údaje, ktoré by sa za bežných okolností do DB neuložili. Môže ísť napríklad o dĺžku automatického generovania modelu či záťaž pri takomto generovaní. Takáto automatizácia v čase písania tejto práce v projekte ešte neexistuje, no voľná štruktúra tejto entity poskytuje dobrý základ na implementáciu ukladania týchto údajov v budúcnosti.

Iné štatistiky, ako napríklad top prispievatelia v procese v počte komentárov (ktoré spomeniem za chvíľu) alebo top použité typy komentárov v rámci procesu, je možné implementovať aj bez nutnosti ukladať si záznamy do tabuľky **ApprovalAction**. Z toho vyplýva, že je nutné dopredu stanoviť požiadavky na štatistiky a z toho potom odvodiť rozsah dát, ktoré si musíme uložiť, aby ich bolo možné vypočítať.

3.1.2 Správa užívateľov

Súčasťou nových požiadaviek na schvaľovanie je aj kontrola prístupu a správa užívateľov pre daný proces. V tomto prípade sa vynárajú dve možnosti na implementáciu. V oboch prípadoch je to väzba medzi entitou *User* a *ApprovalProcess* spolu s typom väzby. Tento typ môže byť aktuálne buď *Historik*, *Grafik* alebo *Modelár*. Prvou možnosťou by bolo využitie polí v entite *ApprovalProcess*. Tá plynie buď z natívnej podpory polí v DB engine *Postgres*, prípadne z ich implementácií až na úrovni ORM[16]. Druhá možnosť pozostáva z klasického rozpadu väzby M:N na pomocnú tabuľku a dve väzby 1:N. Prvý variant má výhodu v tom, že nie je potrebné vytvárať ďalšiu tabuľku, avšak jeho veľkou nevýhodou je len jednostranný efektívny prístup k väzbe. Ak by sme chceli získať projekty, ku ktorým má prístup užívateľ, bolo by

to výpočtovo náročné. Takéto riešenie by navyše nebolo konformné s 1NF[8]. Preto sa z týchto dvoch možností prikloním k druhej a v pomocnej tabuľke okrem identifikátoru užívateľa a procesu bude taktiež aj rola, v ktorej užívateľ v tomto procese vystupuje.

Ďalšou časťou bude následná kontrola prístupu na základe tejto tabuľky. To bude mať na starosti osobitná trieda, ktorá bude obsahovať pravidlá, ktoré na základe vstupu určia povolenie alebo zamedzenie požiadavky. Vstup bude v tomto prípade **užívateľ** (subjekt) vykonávajúci požiadavku, **akcia** (predikát), ktorú chce vykonať a nakoniec **záznam entity**, na ktorej chce akciu vykonať (objekt).

3.1.3 Komentáre

Medzi analyzovanými nedostatkami figurovala taktiež podpora komentárov obmedzená len na jednoduchý text. V novom návrhu budú komentáre uložené v entite **ApprovalComment**, ktorá bude viazaná na iteráciu či už prvej alebo druhej fázy. Táto entita bude mať atribút **type**, určujúci typ komentára, a **content**, určujúci obsah, kde sa počíta s uložením objektu s určitými hodnotami, ktoré prislúchajú danému typu. Takéto riešenie síce porušuje už spomínanú 1NF, ale dáva určitú voľnosť nad pridávaním nového typu komentára, kde nie je nutné, aby sa vytvárala nová tabuľka. Dokumentácia a hlavne aj validácia atribútov rôznych typov komentárov je obzvlášť **nutná**.

Kompletný prehľad návrhu dátovej štruktúry schvaľovacieho procesu je znázornený na obrázku 3.1.

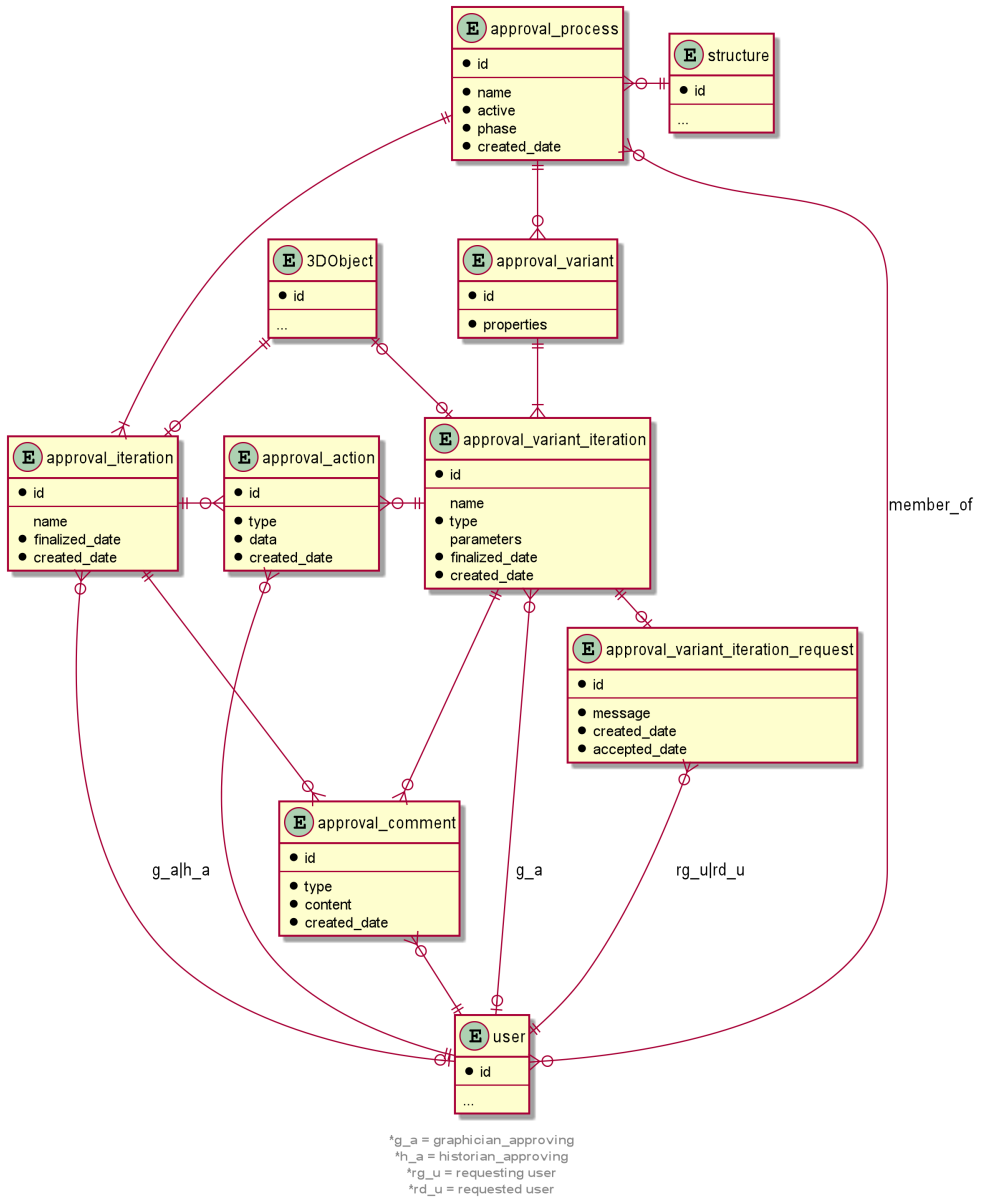
3.1.4 Nutné úpravy existujúceho stavu

Na začiatku tejto sekcie som spomenul nutnosť zmien v aktuálnom dátovom modeli, napríklad oddelenie informácií o schvaľovacom procese od entity **TDObject**. V tejto časti popíšem konkrétne zmeny, ktoré sa vykonajú v už **existujúcej** časti dátového modelu. To môže slúžiť ako príručka pre už existujúce klientské aplikácie, ktoré bude (alebo nebude – to by malo byť jasné z tejto časti) potrebné upraviť.

Odobratie údajov o schvaľovaní z entity TDObject Za prítomnosti osobitnej entity vyjadrujúcej iteráciu v schvaľovacom procese, nie je nutné uchovávať údaje o schválení grafikom, či historikom v zázname 3D objektu. Navyše, na základe typu iterácie v ktorom sa objekt nachádza, je nutné schválenie buď len *grafikom*, alebo *grafikom* a *historikom*.

Úprava možných stavov entity TDObject Rovnako je potrebné zmeniť stavy, v ktorom sa môže objekt nachádzať, čo je vyjadrené v poli **status**.

[8]Relácia je v 1NF, ak všetky jej atribúty sú atomické (nemáme štrukturované a **viachodnotové** atribúty).[17]



Obr. 3.1: Návrh dátového modelu schvaľovacieho procesu

V pôvodnom modeli táto hodnota môže byť jedna z `UNFINISHED`, `SUBMITTED`, `DECLINED`, `FIXED` a `APPROVED`. S novým návrhom nie je konformná hlavne hodnota `FIXED`, čo značí, že aktuálna verzia objektu bola upravená a znovu poslaná na schválenie.

Nový návrh počíta s novou verziou objektu pre každú iteráciu, čo zabezpečí prehľadnosť týkajúcu sa určenia, ktorá verzia bola zamietnutá a prečo. Ak sa bude jedna verzia objektu upravovať, je jednoduché sa rýchlo stratiť v tom, čo už bolo opravené a čo nie, keďže chýba možnosť porovnať si tieto verzie (po úprave verzie sa stratia informácie o starej verzii).

Ďalší problém vzniká pri využití bodových/súradnicových komentárov. Po úprave verzie teoreticky prestáva byť tento komentár relevantný, keďže sa zmenil aj model, ku ktorému súradnice patrili.

Taktiež nie je možné označiť objekt ako dokončený v prípade, že ho autor nechce podrobiť schvaľovaciemu procesu (ten nie je povinný).

Na základe týchto nedostatkov preto navrhujem zmeniť možné hodnoty stavu na: `UNFINISHED`, `FINISHED`, `SUBMITTED`, `DECLINED`, `ACCEPTED` a `FINALIZED` s nasledujúcim popisom ich významu:

UNFINISHED označuje objekt, ktorý je vo fáze tvorby. Je možné ho upravovať k dosiahnutiu spokojnosti autora.

FINISHED označuje objekt, ktorého tvorbu autor dokončil. Od tohoto stavu ďalej nie je možné objekt upravovať. V prípade, že autor nechce objekt podrobiť schvaľovaciemu procesu, je to finálna fáza životného cyklu objektu.

SUBMITTED označuje objekt, ktorý bol určený ako predmet iterácie a výsledok iterácie ešte nie je známy.

DECLINED označuje objekt, ktorý bol odmietnutý jednou zo schvaľovacích autorít v rámci iterácie.

APPROVED označuje objekt, ktorý bol schválený všetkými vyžadovanými schvaľovacími autoritami (záleží na type iterácie).

FINALIZED označuje objekt, ktorý je výstupom schvaľovacieho procesu. Idea za tým je taká, že schválených objektov v rámci procesu môže byť niekoľko, no len jeden z nich sa bude „servírovať“ koncovým užívateľom, a to práve tento.

Z nového návrhu vyplýva, že upravovať verziu objektu bude možné len v počiatočnom stave (`UNFINISHED`). V prípade neskoršej nespokojnosti s objektom sa využije už existujúci verzovací systém, ktorý vytvorí novú verziu a cyklus sa opakuje. Stavový diagram znázorňujúci životný cyklus objektu po nových zmenách sa nachádza na obrázku 3.2.

Zmeny v endpointoch Tieto zmeny spolu s vyňatím schvaľovacieho procesu do osobitných entít viedli aj k úprave rozhrania poskytovaného backendom. Konkrétne ide o nasledujúce adresy:

- `PATCH /3Dobjects/{3DobjectId}/approval` bola odstránená. Po novom sa schvaľuje iterácia, ktorá obsahuje objekt.
- `GET|POST /3Dobjects/{3DobjectId}/comments` boli odstránené. Komentáre sa v novom návrhu viažu na konkrétnu iteráciu.
- `GET /3Dobjects/{3DobjectId}/{version}/comments` bola odstránená z rovnakého dôvodu.
- `PUT /3Dobject/{3DobjectId}/{version}` vyžaduje ako povinný parameter taktiež aj **status**. Hodnota musí byť konformná s popisom vyššie, čo znamená, že jediný možný prechod stavu zásahom užívateľa je z `UNFINISHED` do `FINISHED`. Administrátor môže meniť stav bez obmedzenia (je potrebná **veľká** ostražitosť).

3.2 Spôsob prihlasovania

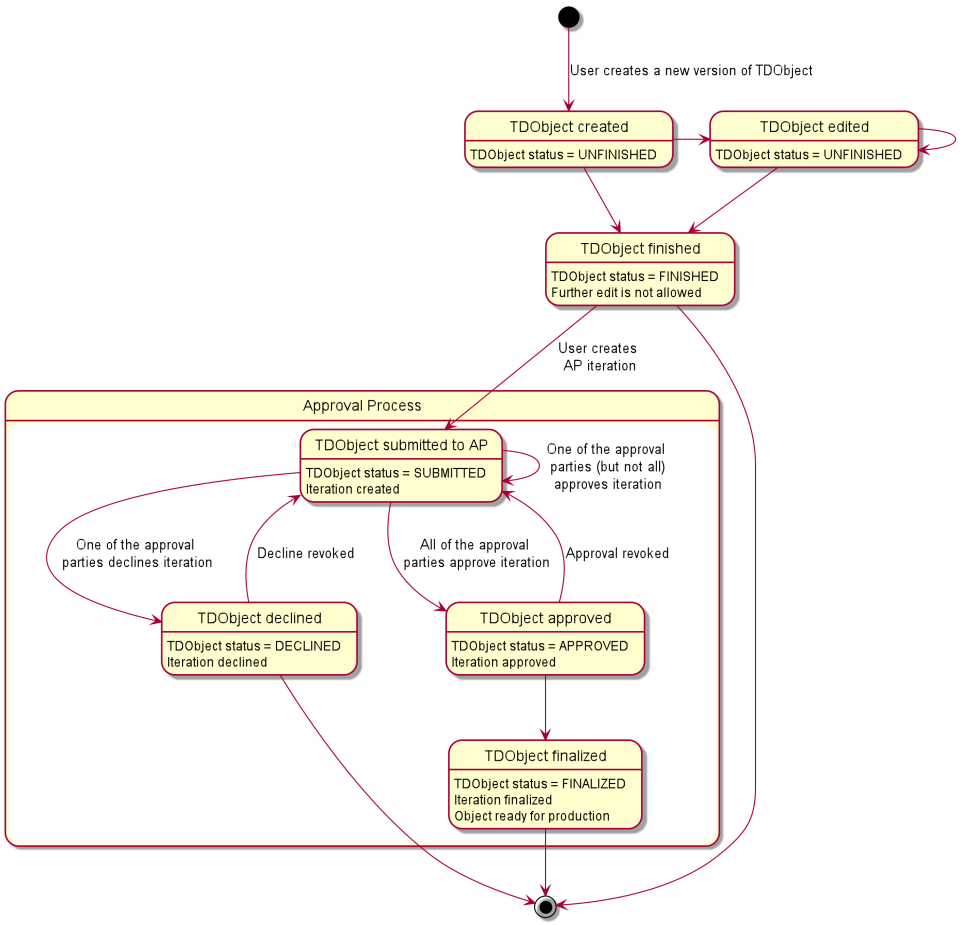
V kapitole Analýza, v časti 2.4.2 som krátko zhrnul návrhy na alternatívne spôsoby prihlasovania z predchádzajúcich prác a doplnil som ich o svoju analýzu. V nej vyšla spomedzi rôznych OAuth autorít ako najlepšia služba Google.

Návrh je pomerne jednoduchý. Pre prihlasovanie vznikne nový *Controller*, ktorý bude prijímať požiadavky na prihlásenie. Tie budú zasielané frontend klientskou aplikáciou, ktorá bude prihlasovanie obsahovať. V praxi pre webové stránky to znamená umiestnenie tlačítka na stránku a minimum kódu vďaka využitiu knižnice priamo od Google. Po úspešnom prihlásení do Google a udelení povolenia aplikácii VMCK pristupovať k základným údajom o profile klientská aplikácia obdrží profil s tokenom. Tieto údaje odošle na backend, kde sa overí ich podpis a v kladnom prípade je užívateľ prihlásený, teda prehliadaču sa odošle VMCK token, pomocou ktorého bude autorizovaný pri nasledujúcich požiadavkách. V prípade, že užívateľ s daným emailom v DB ešte neexistuje, je mu vytvorený nový profil. Meno a email sú známe z prichádzajúceho profilu. Implementačnú príručku pre klientské aplikácie uvediem v kapitole Realizácia.

Sekvenčný diagram procesu prihlasovania sa nachádza na obrázku 3.3.

3.3 Generovanie dát

V kapitole Analýza, v časti 2.4.3 som sa venoval nápadu o možnosti jednoduchého generovania dát v databáze. Ako som uviedol, vidím v tom prínos



Obr. 3.2: Stavový diagram stavu 3D objektu v závislosti na akciách schvaľovacieho procesu

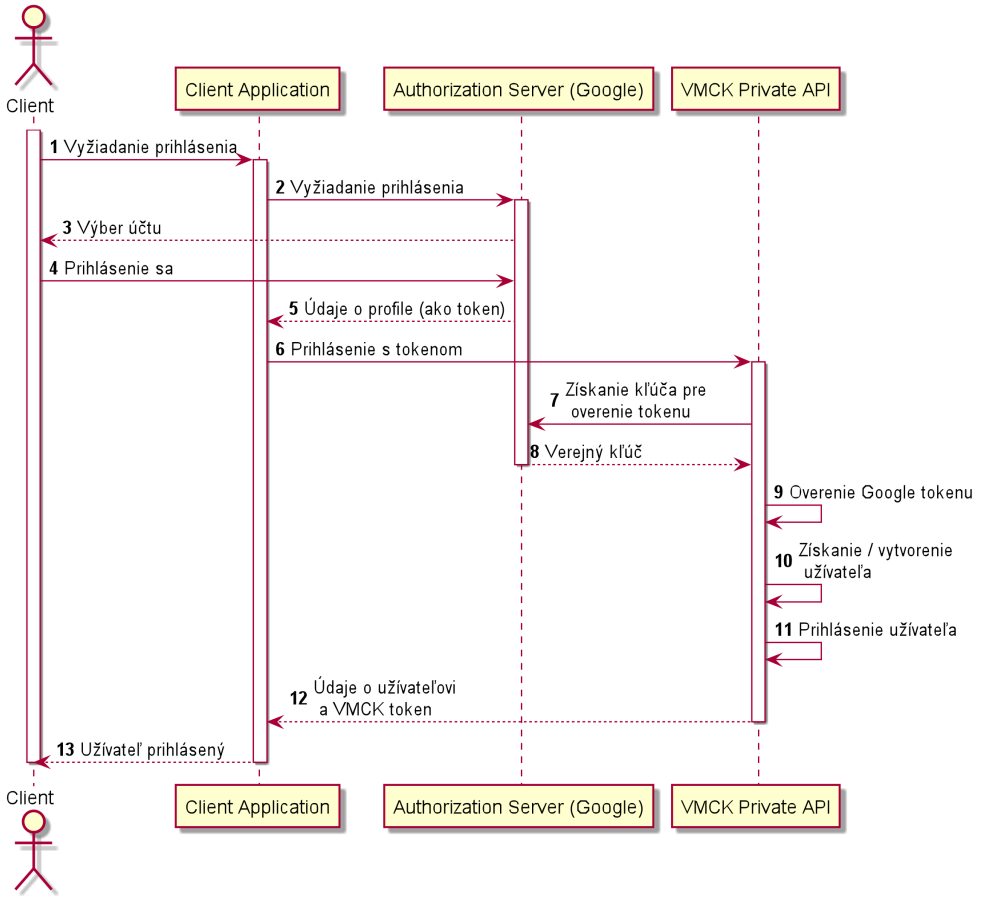
hlavne v počiatočných fázach vývoju či už projektu, alebo novej funkcionality, kde sa práve generovaním dosiahne veľká rôznorodosť a objem dát.

Funkcionalita by mohla byť realizovaná pomocou dvoch variantov:

- Migrácie
- Seedovanie[9]

Migrácie sa v projekte už využívajú na vytvorenie tabuliek a stĺpcov a na zabezpečenie hladkého priebehu zmeny štruktúry DB s ohľadom na dáta. Zvyčajne sa v nich používa priamo SQL jazyk v konkrétnom dialekte, prípadne sa využije vrstva, ktorá umožňuje programátorovi vyjadriť zámer v inom jazyku

[9]Z anglického *seeding* – *siatie*.



Obr. 3.3: Sekvenčný diagram vyjadrujúci proces prihlasovania cez Google (OAuth2)

(napríklad JS/Typescript) a ten sa následne preloží do konkrétneho dialektu v závislosti od aktuálneho pripojenia k DB.

TypeORM poskytuje obe možnosti.

Seedovanie ešte v projekte použité nebolo. V princípe ide o definovanie **Factories** (tovární) a **Seeders**[10]. Factory v sebe obsahuje predpis, ako vyrobiť nový záznam entity. Každej entite teda prislúcha Factory a vygenerované dáta by mali byť ideálne náhodné. Tieto továrne zväčša poskytujú jednoduché rozhranie typu **make**/**create** (vytvor/vytvor a ulož) 1 až n záznamov entity. Toto rozhranie je použité v Seederoch, ktoré obsahujú *logiku* generovania dát. Tu musí programátor zvážiť poradie generovania dát a správne naplnenie väzieb. Od migrácie sa tento spôsob odlišuje primárnym použitím ORM modelov miesto SQL kódu.

[10]Kvôli prehľadnosti nebudem prekladať.

TypeORM takýto spôsob nepodporuje, ale knižnica **TypeORM Seeding** túto funkcionalitu implementuje.

Výber Vo využití migrácií pre generovanie vidím málo výhod a mnoho nevýhod. Jednou takouto výhodou je fakt, že migrácie sú podporované knižnicami, ktoré sa už v projekte nachádzajú. Medzi nevýhody však patrí mixovanie tvorby a úpravy štruktúry DB a vytváranie náhodných dát v DB. Hlavný prínos migrácií oproti synchronizácii ORM modelov (ako som už spomínal napríklad v časti 2.2.1) je bezpečné použitie v produkčnom prostredí vďaka možnosti definície úpravy dát na základe zmien v štruktúre. Na druhú stranu generovanie je niečo, s čím by sa v produkčnom prostredí pracovať nemalo. Určovať pri každom migrovaní, ktoré migrácie sa spúšťať majú a ktoré nie, je veľmi náchylné na ľudskú chybu. Navyše použitie ORM modelov namiesto SQL kódu je bližšie tomu, ako sa v aplikácii reálne nakladá s dátami.

Všetky tieto okolnosti ma priviedli k záveru, že na generovanie dát použijem spôsob **Seedovania** za pomoci knižnice **TypeORM Seeding**.

Príručka k vytváraniu *Factories* a *Seeders* sa bude nachádzať v kapitole Realizácia.

Tvorbu náhodných dát bude mať na starosti knižnica **faker.js**, ktorá podporuje aj český jazyk, čo zvýši hodnovernosť vytvorených mien, či miest.

Kapitola 4

Realizácia

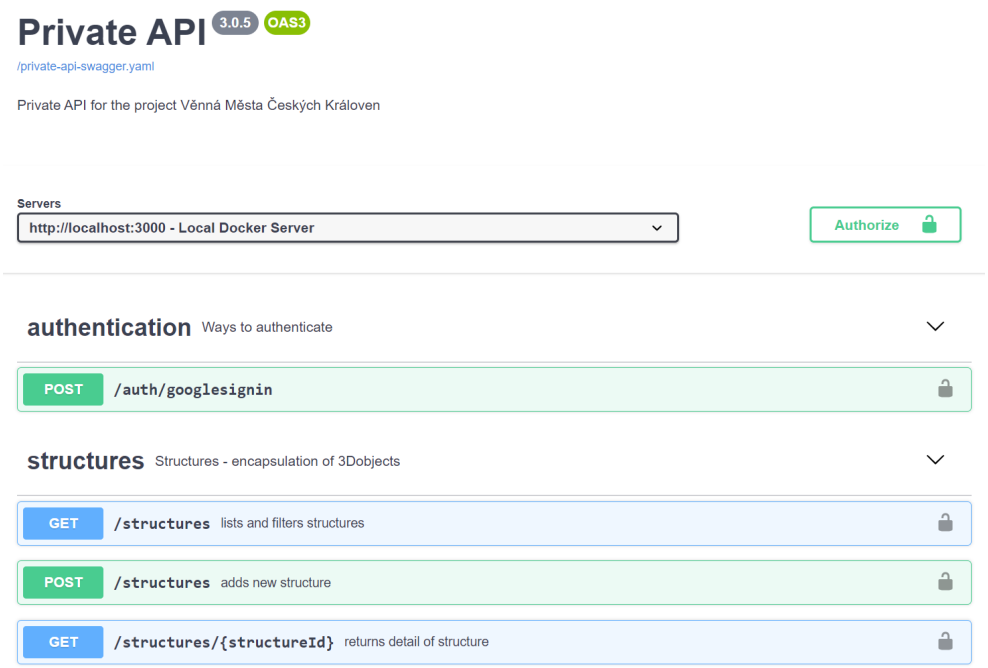
V tejto kapitole sa budem spočiatku venovať popisu vykonanej implementácie s prípadnými návodmi, ako na implementované zmeny napojiť klientské aplikácie. Následne popíšem stav testov a ich automatizáciu, čo vedie k ďalšej časti – Continuous Integration a Continuous Delivery/Deployment. V nej predstavím riešenie, ktoré sa postará o automatické zostavenie, testovanie a nasadenie kódu. Na záver uvediem inštalačnú a programátorskú príručku, ktorá pomôže s orientovaním sa v projekte.

4.1 Schvaľovací proces

Schvaľovací proces bol implementovaný v celom rozsahu tak, ako som popísal v kapitole Návrh. Pre novo vytvorené enpointy boli napísané nové integračné testy a pre upravené endpointy som testy modifikoval, čoho výsledkom je úspech všetkých testov vo verzii, ktorá sa nachádza na priloženom médiu. Počas implementácie som taktiež, vďaka čím ďalej, tým väčšiemu pochopeniu kódu, odstránil aj niekoľko chýb, ktoré boli v kóde spomenuté v komentároch.

4.1.1 Swagger

Pri prevzatí projektu som sa dostal aj k dokumentácii endpointov na portále *Swagger*, ktorý umožňuje uložiť, vizualizovať a testovať API uložené vo formáte *OpenAPI* priamo vo webovom prehliadači. Už počas prvotných fáz návrhu sme spolu s vedúcim práce prišli na fakt, že cenová politika tejto služby neumožňuje ostatným osobám upravovať zdieľaný dokument v bezplatnej verzii. Bol som tým pádom nútený vytvoriť kópiu špecifikácie, ktorá sa v najnovšej verzii nachádza na [18] a zároveň aj na priloženom médiu, v zdrojom kóde backendu v zložke `docs/swagger`. Mojim nástupcom odporúčam rovnaký postup, teda založiť nový Swagger dokument z poslednej verzie. Ako spojazdniť Swagger v lokálnom prostredí, čo je nutné pre testovanie lokálnej inštancie



Obr. 4.1: Ukážka Private API v nástroji Swagger.

backendu, uvediem v programátorskej príručke. Pre krátku predstavu, ako tento nástroj vyzerá, prikladám obrázok 4.1.

4.1.2 Linky/HATEOAS

HATEOAS (*Hypermedia as the Engine for Application State*) je jedným zo základných princípov REST architektúry, ktorý propaguje nízku previazanosť medzi serverom a klientskou aplikáciou. Hypertext označuje akýkoľvek obsah, ktorý obsahuje linky na ďalšie zdroje či akcie.[19] Keďže backend už implementuje REST rozhranie, rozhodol som sa ho skúšobne (mimo funkčných a nefunkčných požiadaviek) rozšíriť o HATEOAS na dvoch miestach, na ktorých by bolo vyhodnocovanie logiky týkajúcej sa týchto linkov v klientskej aplikácii náročné. Konkrétne sa jedná o operáciu získania detailu pre iteráciu vo fáze *model* resp. *variant*. Okrem jednoduchých linkov, ako napríklad *comments* a *tdObject* (odkaz na komentáre, resp. 3D objekt), ktoré v sebe neskrývajú žiadnu logiku a sú prítomné v každej iterácii, sa v detaile môžu vyskytovať aj odkazy na schválenie, odmietnutie či finalizáciu. Tieto akcie sú však podmienené či už stavom 3D objektu, alebo obmedzené len pre určitú rolu v rámci procesu. Všetky tieto okolnosti je možné zo zdrojov vyhodnotiť aj v klientskej aplikácii, no to so sebou prináša správu tejto logiky naprieč všetkými aplikáciami, ktoré implementujú schvaľovací proces. Práve využitie princípu HATEOAS v tomto prípade umožní vyňatie tejto logiky z ostatných aplikácií

a jediné miesto, kde sa bude spravovať je backend. Klientská aplikácia tak bude len kontrolovať prítomnosť linku na vyššie spomínané akcie a v kladnom prípade môže očakávať, že akciu je možné bezprostredne vykonať.

Príklad použitia tohoto princípu je uvedený vo výpise 4.1. Framework *Express.js* však umožňuje len základnú podporu pre linky vo formáte kľúč – hodnota, kde kľúč je názov linku a hodnota je URI adresa, čo neumožňuje špecifikovať napríklad metódu HTTP požiadavky. Vyhľadanie a implementácia robustnejšieho riešenia je mimo rozsah tejto práce, ale subjektívne považujem túto základnú implementáciu za vhodnú na overenie prínosu HATEOAS princípu v projekte.

```
// request
GET http://localhost:3000/approvals/iterations/model/27
accept: application/json
Authorization: Bearer /* token */

// response
HTTP/1.1 200 OK
/* omitted */
Link: </comments/model/27>; rel="comments", \
</3Dobjects/cb3cf416-da79-4ae7-a88b-09a242588f6d>;
    rel="tdObject", \
</approvals/iterations/model/27/approve?as=graphician>;
    rel="approveAsGraphician", \
</approvals/iterations/model/27/approve?as=historian>;
    rel="approveAsHistorian", \
</approvals/iterations/model/27/rejection?as=graphician>;
    rel="rejectAsGraphician", \
</approvals/iterations/model/27/rejection?as=historian>;
    rel="rejectAsHistorian"
Content-Type: application/json; charset=utf-8
Content-Length: 284
/* omitted */

{
  "id": 27,
  /* omitted */
}
```

Výpis 4.1: Ukážka použitia HATEOAS v detaile iterácie vo fáze *model*

4.1.3 Správa užívateľov

Správu užívateľov som implementoval podľa návrhu v časti 3.1.2. Endpointy zabezpečujúce pridanie, či odoberanie užívateľov z procesu sa nachádzajú pod metódami `POST` a `DELETE` na adrese `/approvals/{approvalId}/users/{approvalRole}`. Bližšia špecifikácia je uvedená v Swaggeri popísanom vyššie.

4.1.4 Štatistiky

Základ pre implementáciu štatistík som implementoval podľa návrhu. Medzi implementované výpočty patrí získanie užívateľov s najväčším počtom komentárov v rámci procesu a taktiež typ komentárov, ktorý má najväčšie zastúpenie. Predpokladám, že neskoršie fázy projektu spolu s ostrou prevádzkou prinesú ďalšie nápady na relevantné štatistické informácie, ktorých výpočty budú prebiehať na implementovanom základe.

4.2 Autentifikácia cez Google

V kapitole Návrh, v časti 3.2 som sa venoval možnosti realizovať prihlasovanie cez externú autorizačnú autoritu a výber, po vyhodnotení metrík a dohode s tímom, padol na Google. V tejto časti popíšem nutné kroky, na strane jednak klientskej aplikácie a jednak backendu, vedúce k spojazdneniu takéhoto prihlasovania.

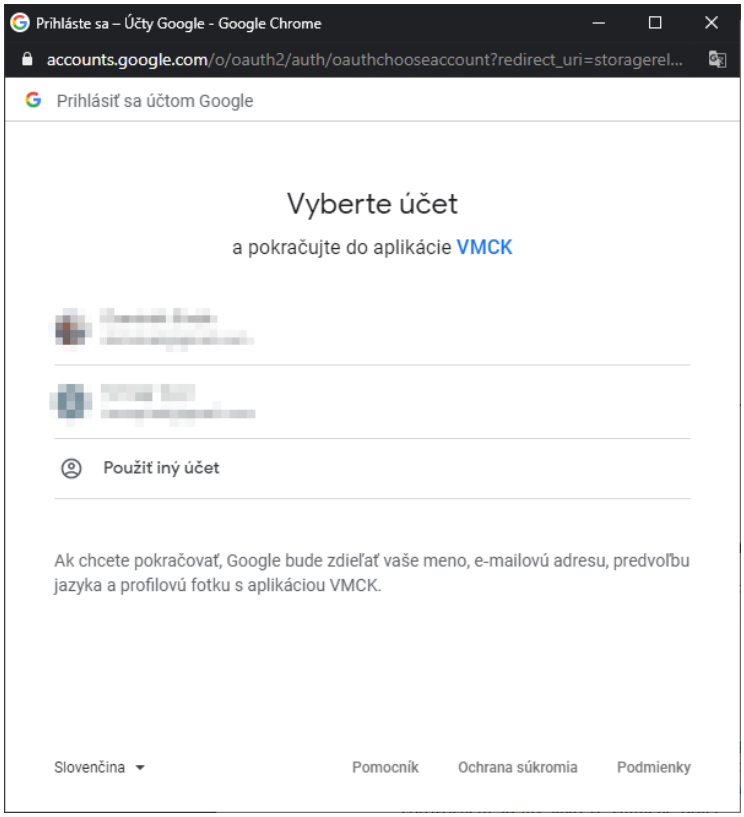
4.2.1 Klientská aplikácia

V tomto projekte sa už od samého začiatku počíta s klientskými aplikáciami postavenými na viacerých platformách. V tejto príručke sa budem sústrediť hlavne na webovú klientskú aplikáciu, keďže mnoho platforiem má možnosť renderovať HTML stránky. Kompletnú príručku je možné nájsť na [20], ja však zhrniem niekoľko dôležitých bodov a postrehov.

Consent screen Ako prvé je nutné vytvoriť tzv. *OAuth consent screen* na adrese `https://console.developers.google.com/apis/credentials/consent`. Jedná sa o dialóg, ktorý sa užívateľovi zobrazí po kliknutí na tlačítko prihlásenia cez Google. V tomto dialógu je užívateľ poučený o tom, do akej aplikácie sa prihlasuje a aké dáta o ňom aplikácia dostane. Pre predstavu prikladám obrázok 4.2.

Client ID Následne je možné vytvoriť OAuth Client ID, čiže identifikátor klientskej aplikácie. Do *Authorized JavaScript origins* je nutné pridať URI, z ktorého sa bude pristupovať k prihlasovaniu. V prípade lokálneho testovania, keď som si vyrobil skúšobnú klientskú aplikáciu to bolo `http://localhost:4000`. Testovaciu aplikáciu popíšem nižšie. Pri ostrej prevádzke to bude pochopiteľne URL z registrovanej domény.

Prihlásenie do backendu Po vytvorení Client ID a nastavení stránky podľa návodu sa postupuje podľa príručky [21]. Z odpovede po prihlásení cez Google účet je nutné odoslať hodnotu `id_token` spolu s Client ID na endpoint `/auth/googlesignin`. Kladná odpoveď má status kód `200` (užívateľ už existuje a bol prihlásený) alebo `201` (užívateľ neexistoval, bol vytvorený



Obr. 4.2: Dialóg s výberom účtu a súhlasom pri prihlasovaní cez Google

a prihlásený) spolu s telom, kde sa nachádza token (tentokrát VMCK token) a údaje o prihlásenom užívateľovi. Konkrétnu podobu requestu a odpovede je možné nájsť v priloženej dokumentácii vo formáte Swagger.

4.2.2 Backend

Po prijatí spomínaného `id_token` a Client ID sa overí ich platnosť verejným kľúčom Google a užívateľ buď existuje, alebo je vytvorený, a následne je prihlásený. Jediná úprava nutná pre spojazdnenie prihlasovania na tejto strane je uvedenie Client IDs, ktoré server prijme (popis generovania Client ID je uvedený vyššie). Tá sa realizuje nastavením hodnoty `GOOGLE_CLIENT_IDS` v súbore `.env`. Backend podporuje aj viacero identifikátorov oddelených čiarkou.

4.2.3 Ukážka

Pre predstavu a ukážku pripájam na priloženom médiu jednoduchú webovú aplikáciu, ktorá ukazuje implementáciu prihlasovania. Aplikácia je jednoduchý Node.js server. Na začiatku je potrebné nainštalovať závislosti príkazom

`npm install`. Následne je nutné nastaviť env hodnotu `CLIENT_ID` podľa popisu vyššie a `SERVER_HOST` v závislosti na adrese bežiacej inštancie backendu. Port je v základe nastavený na hodnotu 4000, ale env hodnotou `PORT` je možné ju zmeniť. Po spustení aplikácie príkazom `npm run start` sa stránka nachádza na `http://$HOST$:$PORT$/` a relevantný kód v súbore `/public/javascripts/google.js`.

4.2.4 Auth0

V neskorej fáze implementácie tejto práce kolega Marek Klofáč narazil vo svojej bakalárskej práci [22] na problém s implementovaným riešením. Ním použitý framework pre VR aplikáciu totiž nebolo možné napojiť na prihlasovanie cez Google. Z tohoto dôvodu implementoval osobitný launcher pre PC a autentifikáciu cez Google, ale pomocou služby Auth0, ktorej úlohou je zjednotiť takéto druhy prihlasovania [23].

V nasledujúcich riadkoch krátko zhrniem proces implementácie a spojazdnenia v backendovej časti. Dôvod výberu tejto služby a kroky pre implementáciu v klientských aplikáciách očakávam v Marekovej práci.

Implementácia a správa Z hľadiska kódu bolo nutné len vyňať proces získavania verejných kľúčov, ktoré podpisujú JWT token a špecifikovať emitentov (*issuers*) týchto kľúčov. Tým bolo možné dodať do overovacieho procesu (ktorý sa stále vykonáva v knižnici od Googlu) všetky potrebné informácie aj pre prihlásenie cez Auth0. Samotné informácie o klientských aplikáciach sa aj v tomto prípade predávajú cez premenné prostredia (súbor `.env`). Konkrétne je to kľúč `AUTH0_PEM_URLS`, ktorý očakáva pole trojíc oddelených čiarkou. Trojica pozostáva z identifikátora kľúča, URL adresy emitenta (*issuer*) a URL adresy verejného kľúča, v takom poradí a oddeľovač je v tomto prípade zvislá čiara (`|`).

Medzi `GOOGLE_CLIENT_IDS` je nutné pridať aj Auth0 identifikátor klienta.

4.3 Generovanie dát

V kapitole Návrh, v časti 3.3 som hodnotil dva spôsoby generovania dát a následne som sa priklonil k variantu použiť knižnicu **TypeORM Seeding**. Generovanie pozostáva z výroby **tovární** (Factories) a **seederov** (Seeders).

Factories sa nachádzajú v zložke definovanej v súbore `.env` pod kľúčom `TYPEORM_SEEDING_FACTORIES`. V základnom nastavení je to zložka `src/database/factories` a v nej súbory s koncovkou `.ts` a `.js`. Takéto súbory obsahujú volanie funkcie `define` z modulu `typeorm-seeding` s prvým parametrom určujúci ORM model, ktorý továreň vyrába a druhým parametrom, ktorým je funkcia prijímajúca inštanciu generátora náhodných dát **faker**

vracajúca inštanciu spomínaného ORM modelu. Pre predstavu odporúčam preštudovať už existujúce továrne v projekte. V tomto texte uvediem príklad jednoduchej továrne na užívateľov vo výpise 4.2.

```
import Faker from 'faker';
import { sign } from 'jsonwebtoken';
import { define } from 'typeorm-seeding';
import { User, UserRole, UserStatus } from '../../entity/User';

define(User, (faker: typeof Faker, context: any | null = null) => {
faker.locale = 'cz';

if (context?.seed) {
    faker.seed(context.seed);
}

const gender = faker.random.boolean() ? 1 : 0;
const user = new User();
const firstName = faker.name.firstName(gender);
const lastName = faker.name.lastName(gender);
user.id = faker.random.uuid();
user.name = faker.name.findName(firstName, lastName, gender);
user.username = faker.internet.userName(firstName, lastName);
user.role = UserRole.COMMON;
user.status = UserStatus.ACTIVE;
user.createdDate = faker.date.recent();
user.email = faker.internet.email(firstName, lastName);

const secret = process.env.JWT_SECRET!;
user.token = sign({ userId: user.id }, secret);

return user;
});
```

Výpis 4.2: Ukážka factory pre generovanie entity User

Taktiež, ak ste sa s tým nestretli, odporúčam prezrieť dokumentáciu [24] knižnice **faker.js**, kde sú rozpísané položky, ktoré dokáže generovať.

Seeders sa nachádzajú v zložke definovanej v súbore `.env` pod kľúčom `TYPEORM_SEEDING_SEEDS`. V základnom nastavení je to zložka `src/database/seeds` a v nej súbory s koncovkou `.ts` a `.js`. Tieto súbory exportujú triedu implementujúcu interface `Seeder`. V metóde `run` s parametrom `factory` sa vykonáva seedovanie. Pomocou tohoto parametru sa dá jednoducho dostať k továrni prislúchajúcej zvolenej entite a využiť make/create metódy popísané v návrhu. Taktiež je nutné dbať na splnenie povinných väzieb. Pre inšpiráciu odporúčam preštudovať už existujúce seedery v projekte. V tomto texte uvediem príklad seederu užívateľov vo výpise 4.3.

```
import { Factory, Seeder } from 'typeorm-seeding';
import { User, UserStatus } from '../../entity/User';
```

```
3
4  export default class CreateUsers implements Seeder {
5  /**
6   * so that token stays the same
7   */
8  private seed = 123456789;
9
10 public async run(factory: Factory): Promise<void> {
11     for (let i = 0; i < 10; i++) {
12         await factory(User)({ seed: this.seed++ }).create({
13             username: `user${i}`,
14             email: `user${i}@example.org`,
15             name: `User ${i}`,
16             status: UserStatus.ACTIVE,
17         });
18     }
19 }
20 }
```

Výpis 4.3: Ukážka seederu pre generovanie entity User

Tento seeder spolu s ukázanou továrňou vznikli na požiadanie vedúceho práce a slúžia na vytvorenie užívateľov s rovnakým ID (a teda aj tokenom) naprieč rôznymi generovaniami dát. Z tohoto dôvodu sa v oboch súboroch pracuje so seedom (*semienkom*) zaručujúcim rovnaké generovanie dát. V iných prípadoch (teda pri iných seederoch) jeho použitie nutné nie je.

Spustenie Po vytvorení príslušných súborov je možné použiť nasledujúce príkazy:

- `yarn seed:run` spustenie seedovania
- `yarn seed:clear` **vymazanie** všetkých záznamov v DB a spustenie seedovania
- `yarn seed:config` zobrazenie konfigurácie použitej pri seedovaní

Seeders sa spúšťajú v abecednom poradí.

4.4 Validácia dát

Pri implementovaní zmien som si všimol veľkú premiešanosť kódu určeného k validácii vstupu, kódu k získaniu záznamov z DB a samotnej logiky endpointov. Vybratím prvých dvoch častí z tela metódy sa výrazne zlepší ich prehľadnosť a pochopenie.

4.4.1 Anotácia @Route

Ešte pred návodom, ako túto validáciu využiť, spomeniem štruktúru anotácie `@Route`. Táto anotácia slúži na označenie metódy ako handler pre určitú URL a HTTP metódu, má päť argumentov a ich význam je nasledovný:

- **route** URL adresa, na ktorej je metóda vystavená
- **validators** pole *validátorov*
- **methods** HTTP metódy, pod ktorými je metóda vystavená
- **fillers** pole *fillerov* (*plničov*)
- **gates** pole *brán*

4.4.2 Validátory

Hlavnou úlohou validátorov je kontrola a sanácia užívateľského vstupu. V projekte pred mojim zásahom sa používali len zriedkavo, a to zväčša len na validáciu *path* alebo *query* parametrov. Po vykonaní malej úpravy v kóde je odteraz možné validovať aj položky v tele požiadavky, dokonca aj vnorené položky. Validátor môže byť typu (medzi inými) *chain*[25] alebo *schema*[26]. Použitie oboch ukážem na príklade. Súbor `UserController` obsahuje metódu `createUser`, ktorá, ako názov napovedá, slúži k vytvoreniu užívateľa. V jej tele sa nachádza nasledujúci kód:

```
1  const name = validateField(this.request, 'name', [
2      (value) => !validator.isEmpty(value)
3  ]);
4  const username = validateField(this.request, 'username', [
5      (value) => !validator.isEmpty(value)
6      && validator.isAlphanumeric(value)
7  ]);
8  const email = validateField(this.request, 'email', [
9      (value) => !validator.isEmpty(value)
10     && validator.isEmail(value)
11 ]);
12
13 if (!name || !username || !email) {
14     return this.response.status(400).json('Name, username and
           email are required');
15 }
```

Výpis 4.4: Validačný kód v tele funkcie

Namiesto toho je však možné v poli **validators** uviesť validátory, a to buď typu *chain*:

```
1  @Route(..., /* validators */ [
2      body('name').notEmpty(),
3      body('username').notEmpty().isAlphanumeric(),
```

```
    body('email').notEmpty().isEmail(),
], ...)
```

Výpis 4.5: Validátory typu chain

alebo typu *schema*:

```
@Route(..., /* validators */ [
    ...checkSchema({
        name: {
            in: 'body',
            notEmpty: true
        },
        username: {
            in: 'body',
            notEmpty: true,
            isAlphanumeric: true
        },
        email: {
            in: 'body',
            notEmpty: true,
            isEmail: true
        }
    })
], ...)
```

Výpis 4.6: Validačný kód v tele funkcie

Pri použití validátorov sa chybová hláška spolu s kódom *400* odošlú automaticky a tým pádom nie je nutné ich neustále opisovať. Chain metóda je kompaktnejšia, no z môjho pohľadu menej prehľadná ako schema metóda. Ich efektivita je však ekvivalentná. `express-validator` obsahuje okrem validácie aj sanáciu vstupu. Teda napríklad, ak čakáme na vstupe číslo či dátum, je možné využiť *sanitizer* (voľný preklad „*očisťovač*“) `toInt` resp. `toDate` na automatické prevedenie do cieľového typu. Použitie validátorov `isNumeric` a `isISO8601` je v tomto prípade vysoko odporúčané. Ku všetkým validovaným a očisteným dátam sa v metóde controlleru dostaneme (v prípade, že boli oba procesy úspešné) cez `this.matched`.

4.4.3 ID validátory

Špeciálne by som chcel spomenúť aj koncept, ktorý som pomenoval ID validátory. Z mojej pracovnej činnosti (vývoj vo frameworku Laravel) som bol zvyknutý na pohodlné validovanie prítomnosti identifikátoru zo vstupu v DB, avšak `express-validator` takú možnosť neposkytoval. Podobne, ako pri bežných validátoroch, som chcel deduplikovať kód, ktorý kontroluje, či sa záznam s určitým identifikátorom v DB nachádza a v prípade negatívneho výsledku vráti chybovú hlášku. Využil som na to validátor `custom`, ktorý umožňuje vlastnú špecifikáciu kontrolnej metódy. ID validátor pre entitu User vyzerá takto:

```
@Route(..., /* validators */ [
    ...checkSchema({
        userId: UserIdValidator(),
    }),
], ...)

//UserIdValidator
export const UserIdValidator: IDValidator = (location) => {
    return {
        in: location ?? 'params',
        isString: true,
        isUUID: {
            options: '4',
        },
        custom: {
            options: existsDB(() => injector.get(UserModel.name)
                as UserModel),
        },
    };
};

//existsDB
export function existsDB<T extends ExistsDBModel>(model: () => T)
    : CustomValidator {
    return (input, meta) => model().existsDB(input);
}

//UserModel
public existsDB(input: any): Promise<any> {
    return this.userRepository.findOneOrFail(input);
}
```

Výpis 4.7: ID validátor entity User

Validácia ID užívateľa sa teda v kóde nachádza len raz a v prípade nutnosti komplexnejšej validácie (napr. užívateľ musí mať rolu `active`) sa upraví len metóda `existsDB` v `UserModel`.

4.4.4 Plniče

Koncept plničov, alebo *fillers*, som vymyslel za účelom odstránenia duplicitného kódu na získanie záznamu z DB. Aj pri tejto operácii sa často kontrolovala `not-null` hodnota premennej a v negatívnom prípade sa vracala stále pomerne rovnaká chybová hláška. Koncept pozostáva z definovania objektu *fillers* v anotácii `@Route`. Kľúče tohoto objektu sú názvy premenných, podľa ktorých sa k ním bude pristupovať a hodnoty sú objekty typu `Promise`, ktoré obsahujú logiku získania záznamu z DB. V nej môžu pristupovať k už validovaným a vyčisteným dátam z predchádzajúceho kroku. Tieto `Promises` sa asynchrónne vykonajú a výsledkom je prístup k záznamom v tele metódy controlleru cez objekt `this.filled`. V prípade, že sa niektorý z `Promises` ne-

podarí splniť, je vrátená chybová hláška s kódom `404`. Príklad použitia plniču pre entitu User:

```
@Route(..., /* fillers */ (matched, _this) => ({
    user: _this.userModel.getById(matched.userId),
}), ...)
public async example() {
    const user: User = this.filled.user;
    // nemusime kontrolovat not-null
}
```

Výpis 4.8: Plnič pre entitu User

4.4.5 Brány

Brány slúžia ako akýsi pracovník bezpečnostnej služby, ktorý kontroluje právo užívateľa pristúpiť k zdroju. Logika kontroly prístupu sa tak môže oddeliť a nemusí byť súčasťou každej metódy obsluhujúcej endpoint. Princíp spočíva vo funkcii, ktorá vráti `boolean` odpovedajúci pusteniu cez bránu, prípadne môže byť doplnený o správu, ktorá sa odošle v odpovedi v negatívnom prípade. Kód tejto odpovedi je `403`, teda `Unauthorized`. Ich fungovanie je síce podobné ako použitie middleware, ktorý môže taktiež prerušiť spracovávanie požiadavky a vrátiť chybovú hlášku, a ich podpora už existovala pred mojou prácou, no *brány* môžu využiť už spomenuté objekty **matched** a **filled**, teda validované, vyčistené dáta resp. doplnené záznamy z DB. To výrazne odľahčí nutnú komplexitu brán (nemusí sa už kontrolovať vstup a komunikovať s DB). Príklad brány a príklad jej použitia pre entitu ApprovalProcess:

```
@Route(..., /* gates */ (matched, filled, _this, req) => [
    _this.accessManager.can(req.user, 'delete', filled.process),
], ...)

// accessManager.can
public async can(user: User, action: ApprovalProcessCan, process:
     ApprovalProcess) {
    if (user.role === UserRole.ADMIN) {
        return true;
    }
    const where: FindConditions<ApprovalProcessToUser> = {
        process, user };
    switch (action) {
        case 'detail':
        // omitted
        case 'createVariantIteration': {
            // any role can view
            break;
        }
        case 'update':
        case 'delete':
        case 'owner': {
            where.role = ApprovalProcessToUserRole.CREATOR;
```

```
22                 break;
23             }
24             // omitted
25         }
26         // EXISTS would be more efficient, but it's not supported by
               TypeORM
27         const results = await this.membersRepository.count({ where })
               ;
28         return results > 0;
29     }
```

Výpis 4.9: Brána pre entitu ApprovalProcess

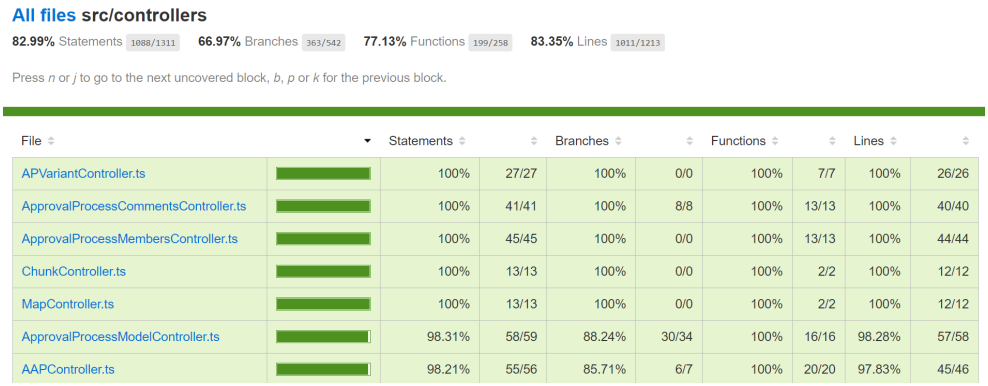
4.5 Testy

Testom som sa venoval už v kapitole Analýza, v časti 2.2.1 a popisoval som ich stav a malé úpravy, ktoré tento stav vylepšili. Stále sa však používala rovnaká DB pre testy aj server, čo spôsobovalo problémové spúšťanie testov vyžadujúce si zmazanie všetkých údajov v DB po testovaní. To som vyriešil využitím osobitnej databázy pre testovanie. Databáza sa vytvorí pri prvotnom spustení *Postgres* Docker kontajneru vykonaním skriptu uloženého v `/docker/postgres/db_init.sql`. Po úprave tvorby pripojenia k DB pre testy sa všetky testy spúšťajú oproti novovytvorenej databáze **test**. Obsah DB sa pred vykonávaním každého *Test suite*, alebo súboru s testami, zmaže a pomocou migrácií nastaví do počiatočnej podoby. To zabezpečí bezproblémový priebeh testov a nezávislosť na vzájomnom poradí spúšťania. Testy, oproti stavu pri preberaní kódu, je teraz možné spúšťať všetky a žiaden z nich nie je neúspešný, čo otvára dvere pre automatické testovanie ako súčasť Continuous Integration a Continuous Deployment. Tomu sa budem venovať v ďalšej časti.

Pre všetky časti schvaľovacieho procesu som vytvoril príslušné integračné testy, ktoré sa nachádzajú štandardne v zložke `/tests`. Celkový počet nových testov je 55.

Testy sa spúšťajú príkazom `yarn test`. Týmto sa spustia všetky testy uložené v spomínanej zložke. Pre spustenie len vybranej sady testov (napríklad súbor `UserController.test.ts`) môžeme tento výber špecifikovať (`yarn test UserController`).

Vhodný môže byť tiež príkaz `yarn test:coverage`, ktorý spúšťa testy spolu s kontrolou pokrytia kódu. Výstupom, okrem splnených a nesplnených testov, je aj percento pokrytia riadkov kódu, príkazov, funkcií a vetiev pre každý Typescript súbor v projekte. Vzhľadom na to, že to pokrýva aj migrácie (časť vykonávajúca migráciu sa vykoná, časť vracajúca migráciu sa nevykoná), seedery a továrne, môže to znižovať výsledné percento. Našťastie existuje možnosť vylúčiť tieto súbory zo zberu pokrytia. Výstupom je okrem výpisu do konzoly aj XML, JSON, či ľudsky čitateľný HTML súbor (ukážka



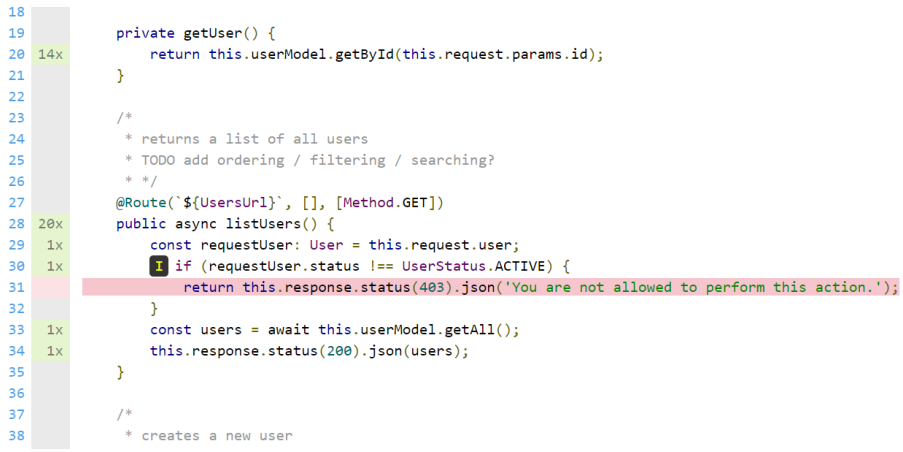
All files src/controllers

82.99% Statements 1088/1311 66.97% Branches 363/542 77.13% Functions 199/258 83.35% Lines 1011/1213

Press n or j to go to the next uncovered block, b, p or k for the previous block.

File	Statements		Branches		Functions		Lines	
APVariantController.ts	100%	27/27	100%	0/0	100%	7/7	100%	26/26
ApprovalProcessCommentsController.ts	100%	41/41	100%	8/8	100%	13/13	100%	40/40
ApprovalProcessMembersController.ts	100%	45/45	100%	0/0	100%	13/13	100%	44/44
ChunkController.ts	100%	13/13	100%	0/0	100%	2/2	100%	12/12
MapController.ts	100%	13/13	100%	0/0	100%	2/2	100%	12/12
ApprovalProcessModelController.ts	98.31%	58/59	88.24%	30/34	100%	16/16	98.28%	57/58
AAPController.ts	98.21%	55/56	85.71%	6/7	100%	20/20	97.83%	45/46

Obr. 4.3: Ukážka HTML výstupu pokrytia testov



```
18
19          private getUser() {
20    14x       return this.userModel.getById(this.request.params.id);
21          }
22
23          /*
24           * returns a list of all users
25           * TODO add ordering / filtering / searching?
26           * */
27          @Route(`${UsersUrl}`, [], [Method.GET])
28    20x   public async listUsers() {
29     1x       const requestUser: User = this.request.user;
30     1x       if (requestUser.status !== UserStatus.ACTIVE) {
31                 return this.response.status(403).json('You are not allowed to perform this action.');
32           }
33     1x       const users = await this.userModel.getAll();
34     1x       this.response.status(200).json(users);
35          }
36
37          /*
38           * creates a new user
```

Obr. 4.4: Ukážka HTML výstupu pokrytia testov – zobrazenie nepokrytých riadkov

na obrázkoch 4.3 a 4.4). Pokrytie kódu testami vo výstupnej fáze kódu tejto práce je 91%.

4.6 Continuous Integration a Continuous Deployment

Continuous Integration (CI) a Continuous Deployment (CD) sa pri tomto projekte neskloňujú po prvý krát.

„*CI je postup vývoja SW, pri ktorom členovia tímu často integrujú svoju prácu - často niekoľkokrát denne. Každá integrácia je overená automatickým zostavením (buildom) a automatickým testovaním, aby sa chyby integrácie zistili čo najrýchlejšie. Mnoho tímov vykazuje, že tento prístup vedie k výrazne zníženým problémom s integráciou a umožňuje tímu rýchlejšie vyvíjať súdržný*

SW.“[27] Kolega Vančura vo svojej práci [1] krátko analyzoval možnosti spojazdnenia CI pre tento projekt, no práve nespoľahlivá testovacia infraštruktúra ho obmedzila v realizácii tejto funkcionality. „*Jedným z hlavných dôvodov nevytvorenia fungujúceho CI v mojej práci je práve v predchádzajúcej kapitole spomenutá chyba, kedy je nutné testy spúšťať jednotlivo, inak dôjde k neúspechu všetkých, ktorú sa mi behom vývoja nepodarilo odstrániť. Vo chvíli, kedy testovanie nefunguje bez chýb ani pri lokálnom spustení, mi prišlo spúšťanie testov na vzdialenom serveri zbytočné.*“ napísal. Keďže testy, ako som v predchádzajúcej časti tejto práce spomenul, už fungujú bez problémov, a to aj všetky naraz, znamená to možnosť využiť plný potenciál, ktorý CI poskytuje. Na tomto projekte síce nepracuje veľké množstvo vývojárov (ak vôbec nejakí paralelne pracujú alebo budú pracovať), stále však vidím prínos v realizácii CI práve vďaka rýchlemu zisťovaniu prvotných chýb v kóde.

Nástrojov a služieb na realizáciu CI existuje niekoľko, no ja sa zameriam na, pre tento prípad, najjednoduchšiu možnosť – využitie CI/CD poskytovaného GitLabom. Ako už bolo spomenuté, zdrojový kód backendu je uložený práve na tejto službe a tá už niekoľko rokov poskytuje CI/CD zadarmo pre každý projekt[11]. CI sa konfiguruje pomocou súboru *.gitlab-ci.yml*, na základe ktorého sú vykonávané fázy (stages). V rámci jednej fázy môže byť niekoľko úloh (jobs), ktoré sú na sebe zväčša nezávislé. Tento súbor zároveň špecifikuje Docker image, ktorý sa použije pre vykonávanie fáz, artefakty (stiahnuteľný výstup úloh), či napríklad použitie cache. Kompletná príručka ku konfigurácii GitLab CI je dostupná na [29].

Pred vysvetlením štruktúry celého procesu by som chcel spomenúť aj pojmy Continuous Delivery a Continuous Deployment. „*Continuous Delivery je disciplína vývoja SW, v ktorej sa SW zostavuje takým spôsobom, aby mohol byť kedykoľvek uvedený do produkcie.*“[30] V praxi to znamená možnosť nasadiť zostavený kód na bežiaci server jednoducho, napríklad jedným klikom. Tento proces priamo naväzuje na CI, ktorý build pripraví a otestuje.

„*Continuous Deployment znamená, že každá zmena, ktorá prejde pipeline, je automaticky nasadená, čo vedia k mnohonásobným nasadeniam denne.*“[30] Tieto pojmy sa často zamieňajú, prípadne sa preferuje len jeden termín a rozdiel je práve v automatizácii nasadenia. Continuous Delivery je nutná podmienka pre Continuous Deployment.

4.6.1 CI & CD v tomto projekte

Pre tento projekt som vytvoril tri fázy: **build**, **test** a **deploy**. Všetky využívajú rovnaký Docker image – `docker/compose`, teda oficiálny image od Dockeru, ktorý je stavaný na spúšťanie ďalších kontajnerov v sebe cez službu `docker-compose`. Prostredie použité v CI je tým pádom identické s prostre-

[11] Obmedzený počet minút behu CI/CD. V prípade vyšších požiadaviek je možné na build a testing použiť vlastný server, ktorý bude s GitLabom komunikovať.[28]

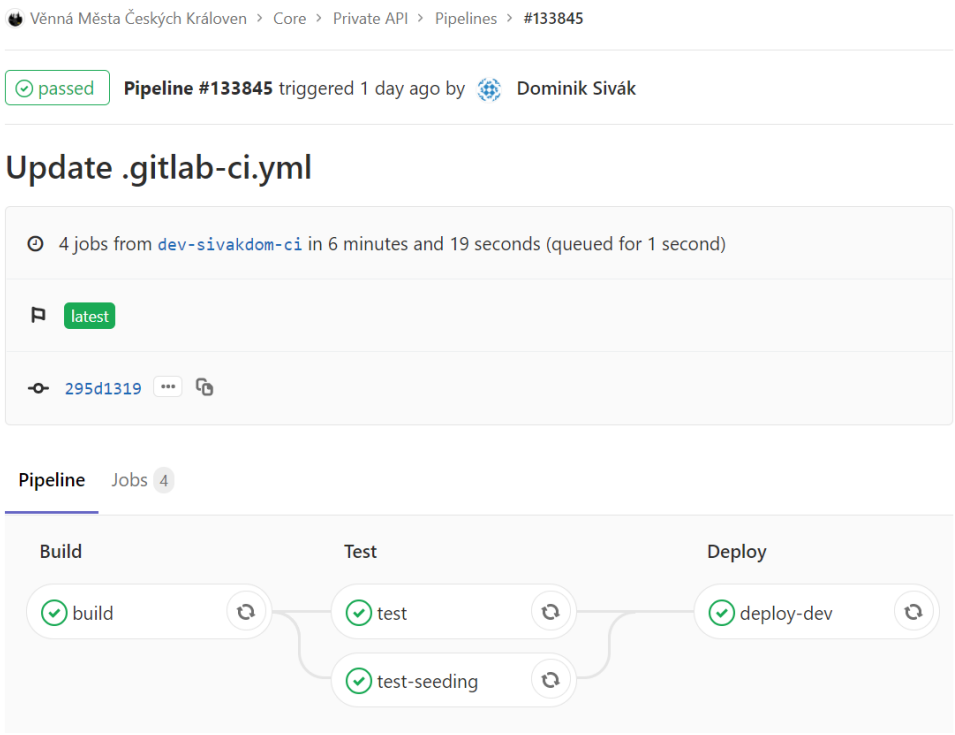
dím pre vývoj a prevádzku. Postupnosť týchto fáz a úloh v nich sa nazýva *pipeline*.

build Fáza build je zodpovedná za inštalovanie závislostí a zostavenie kódu. Zostavením sa môže prísť aj na syntaktické chyby, ktoré by však kód nemal obsahovať vďaka lokálnym kontrolám pred každým commitom. Výsledok zostavenia je JavaScript kód (zdrojový je pre pripomenutie v TypeScript). Pre ďalšie fázy a beh na serveri momentálne tento výsledný kód nemá žiadnu hodnotu, pretože všetky tieto činnosti pracujú s originálnym TypeScript kódom. Porovnanie výkonnosti serveru spusteného pomocou `yarn dev` (používa TypeScript) a `yarn start` (používa zostavený JavaScript v zložke `build`) je mimo rozsah tejto práce, preto to nechávam na nasledujúce iterácie. Táto fáza taktiež využíva **cache**, teda závislosti sa nemusia pri každom builde sťahovať (linkovať a prípadne zostavovať), ale na konci úlohy sa zložka `node_modules` zabalí do archívu a v ďalšej iterácii sa naspäť rozbalí.

test Fáza test sa spúšťa za účelom testovať bežiaci kód. Vo fáze bežia dve úlohy paralelne – prvá, dôležitejšia, spúšťa už v predchádzajúcej kapitole spomenuté testy spolu s pokrytím kódu. V druhej sa testuje naplnenie databázy seedermi, čo z hľadiska funkcionality nemá veľký prínos, ale má potenciál odhaliť možné chyby. Obe úlohy využívajú cache z prvej fázy, čo znamená skrátenie času behu. V tomto prípade ale cache funguje len jednosmerne a po ukončení úlohy sa nevytvára nová verzia.

deploy Vstup do poslednej fázy znamená úspech predchádzajúcich, čo značí, že (s najväčšou pravdepodobnosťou) máme nový funkčný build. Tento build ostáva už len nasadiť na server (prípadne viacero). K tomu som použil program **rclone**[31], ktorý slúži na manažment súborov v cloudových alebo inak vzdialených úložiskách. Pre tento projekt je relevantná podpora SFTP, keďže k serverom existuje SSH pripojenie. V rclone sa najprv vytvorí konfigurácia pre pripojenie za použitia premenných prostredia, ktoré je možné nastaviť v možnostiach pre CI v správe projektu na GitLabe. Vďaka tomu je možné oddeliť citlivé prihlasovacie údaje od verzovaného kódu. Následne sa príkazom `copy` prekopírujú zmenené súbory na server. Zmena sa vyhodnocuje na základe veľkosti súboru a kontrolného súčtu. Opäť sa využíva zostavenie z prvej fázy. Po úspešnom kopírovaní sú zmeny nasadené[12]. Táto fáza nasleduje princíp Continuous Delivery vďaka použitiu pravidla `when` s hodnotou `manual` v konfigurácii CI. Úloha sa tak vykoná len v prípade explicitného vyžiadania, ktoré je v GitLabe dosiahnuté dvoma klikmi myšou. V budúcnosti je možné hocikedy prejsť na Continuous Deployment (automatické nasadenie) zmazaním spomínaného riadku. Taktiež je možné obmedziť úlohu len na konkrétnu

[12] V aktuálnej konfigurácii serverového kontajneru sa spúšťa už spomínaný príkaz `yarn dev`, ktorý sleduje zmeny v súboroch a v prípade potreby sa automaticky reštartuje.



Obr. 4.5: Detail vykonávania pipeline po commite.

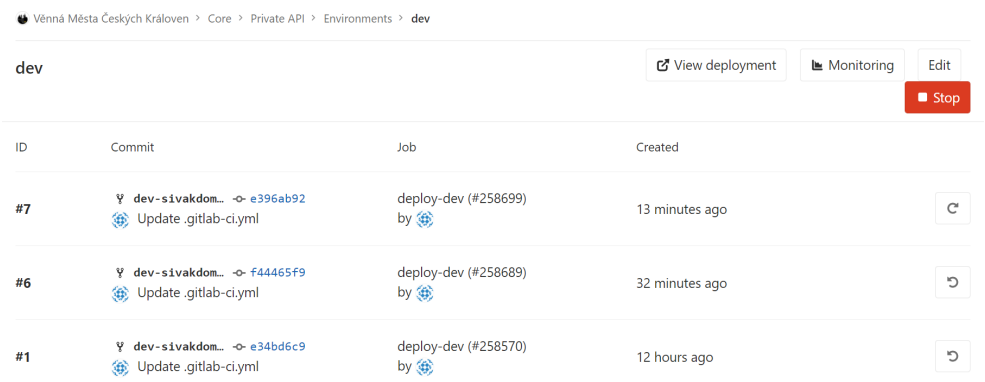
git vetvu, a/alebo prípadne na commity s tagom, nastavením pravidla `only` s hodnotami `master`, resp. `tags`. Toto je možné využiť napríklad pre automatický deploy z **dev** vetvy na dev server (v prípade vymazania pravidla `when`). Ďalším využitím môže byť zamedzenie nasadenia na produkčný server z inej, ako **master** vetvy.

4.6.2 Prostredia

Služba GitLab ponúka funkciu **Environments**, teda prostredí, ktoré má za úlohu sledovať nasadenia na serveri. Nastavenie je jednoduché:

- V prvom rade je potrebné vytvoriť prostredie v UI GitLabu v *Operations* →*Environments* →*New environment*
- Následne do úlohy vo fáze deploy je potrebné špecifikovať objekt `environment` s hodnotami `name` rovnajúcej sa názvu z predchádzajúceho kroku, prípadne `url` s adresou serveru (hodí sa skôr pre frontend projekty).

Každá splnená CI úloha s takýmto nastavením pridá záznam o nasadení na konkrétny server spolu s časom a commitom, ktorý spustil CI. Prehľad nasadení na server s pomenovaním dev je ukázaný na obrázku 4.6.



Obr. 4.6: Prehľad nasadení na server **dev** v GitLab Environments.

4.6.3 Zhrnutie CI & CD

Celý konfiguračný súbor je k dispozícii na priloženom médiu v zložke so zdrojovým kódom v súbore `.gitlab-ci.yml`.

Na záver tejto časti by som rád vyzdvihol veľmi pozitívne skúsenosti s CI v GitLabe. Okrem použitia v zamestnaní som ho zaviedol aj na zostavenie PDF súboru mojej bakalárskej a taktiež aj tejto práce (môjmu programátorskému ja sa priečilo verzovanie „buildovaného“ PDF súboru). S touto jednoduchou, párriadkovou konfiguráciou sa na priloženom médiu veľmi rád podelím s budúcimi pisateľmi záverečných prác.

4.7 Návrhy do budúcna

V tejto sekcii by som sa rád podelil o pozorovania a nápady, ktoré vznikli počas písania práce a implementácie kódu. Ešte predtým však chcem upozorniť aj na vízie, ktoré vo svojej práci uviedol Dan Vančura na konci kapitoly návrh a krátko som ich vymenoval a zhrnul v tejto práci v časti 2.4.1. Táto sekcia bude obsahovať konkrétnejšie pozorovania, niekedy aj s návrhmi na implementáciu, ktorých realizácia sa ale do časového rozpočtu tejto práce nezmestila.

4.7.1 Získavanie verzií entity TDObject

Verzie entity TDObject fungujú ako spojový zoznam. Každá verzia má identifikátor a (nepovinnú) väzbu na predchádzajúcu a nasledujúcu verziu. Pri získavaní všetkých verzií sa vykonáva rekurzívny SQL dotaz (súbor `TDObjectModel`), ktorý postupuje podľa týchto väzieb. Skúsenosti a predstavu a výkonnosti takéhoto dotazu nemám, ale odhadujem, že pri väčšom počte verzií môže byť spomaľujúca.

Môj návrh je zakomponovať nový identifikátor skupiny (`gid`), ktorý by všetky verzie zdieľali a zároveň použiť identifikátor *v rámci* skupiny, ktorý by určoval poradie. To sa dá implementovať pomocou vlastných sekvencií v Postgres alebo získaním maximálnej hodnoty v príslušných stĺpcoch.

Pred zvažovaním tohoto variantu je však nutné zobrať do úvahy frekvenciu tohoto typu dotazu a predpokladanú početnosť verzií na jeden reálny objekt a na základe toho vyhodnotiť výhodnosť takejto úpravy.

4.7.2 Zavedenie pokročilejšej autorizácie

Ako som už spomínal v časti 2.4.1, token pre autorizáciu užívateľa je od jeho vytvorenia v DB nemenný, keďže sa do neho ukladá a podpisuje len jeho identifikátor. To vyhodnocujem ako bezpečnostné riziko, ktoré musí byť pred ostrou prevádzkou odstránené.

4.7.3 Správa rolí užívateľa

Je vhodné v budúcnosti prehodnotiť roly užívateľa. Momentálne je možné, aby užívateľ mal priradenú len jednu rolu, čo ako nedostatok uviedol aj kolega Vančura. V budúcnosti, hlavne keď sa použije tento projekt aj v školskom prostredí, hodnotím toto riešenie ako nedostatočné. Zároveň vidím potrebu implementácie lepšej formy kontroly prístupu k endpointom, než je to aktuálne.

4.7.4 Notifikačný systém

O vytvorení novej iterácie, schválení, zamietnutí, či požiadavky o modelovanie je vhodné informovať zainteresované osoby. To môže byť vykonané minimálne emailom, ale aj inými typmi notifikácií. Odporúčam pre to využiť eventy a event bus (príklad v súbore `src/utils/ApprovalEventBus.ts`), aby sa predišlo zbytočnej previazanosti.

4.8 Inštalačná príručka

Od stavu, v ktorom sa kód preberal, nastalo zopár zmien týkajúcich sa inštalácie a nasadenia.

4.8.1 Testovacia DB v Postgres

Prvou z nich je zavedenie testovacej DB v Postgres. V prípade prvého spúšťania Docker kontajnerov nie je potrebná žiadna akcia naviac.

Ak však budeme chcieť vykonávať testy na kontajneri, ktorý bol vytvorený ešte pred touto zmenou, je potrebné DB vytvoriť. To je možné cez prostredie Adminer (jeden z Docker kontajnerov). Po prihlásení je potrebné vybrať

v hornom menu položku `postgres:5432` a zvoliť `Create database`. Názov DB musí byť `test`.

4.8.2 Swagger

Použitie a online lokalitu uloženia dokumentácie backendu v Swagger som už spomenul v časti 4.1.1. Vyjadril som sa aj k platobným podmienkam a nemožnosti použiť online editor na kolaboráciu v bezplatnej verzii. Vlastné verzovanie dokumentu s OpenAPI špecifikáciou spolu so zdrojovým kódom preto vidím ako jedinú reálnu možnosť spolupráce. Konkrétne bude možné tento súbor nájsť pod menom `private-api-swagger.yaml` v zložke `docs/swagger`.

4.8.2.1 Prístup k localhost

Je zrejmé, že online editor môže pristupovať iba na verejne dostupné IP adresy, čo v prípade lokálneho nasadenia spôsobuje problém. Je potrebné preto Swagger spúšťať lokálne. Jedným zo spôsobov, ako to dosiahnuť je použitie oficiálneho Docker image `swaggerapi/swagger-editor`. Ten som zakomponoval do už existujúceho `docker-compose.yaml`, čoho výsledkom je plne funkčný editor Swagger dokumentácie, ktorý beží lokálne, a to na adrese `localhost:3003`.

4.8.3 Alternatívne lokálne prostredie

V tejto časti by som rád uviedol niekoľko postrehov pri rozbiehavaní backendu v lokálnom prostredí. Niektoré body som uviedol na začiatku analýzy v sekcii 2.1. Zo začiatku vývoja som Docker kontajnery používal tak, ako boli navrhnuté a popísané v práci kolegu Vančuru. Po výrazných nevýhodách v používaní WSL 2, ktoré som popísal v spomínanej sekcii (pre pripomenutie hlavne extrémne pomalá práca so súbormi a chýbajúca propagácia zmien), som sa rozhodol server spúšťať lokálne vo Windows a zvyšné kontajnery pôvodným spôsobom. Dĺžka vykonávania operácií sa znížila rádovo a pre predstavu pripájam tabuľku 4.1. Najväčším prekvapením pre mňa bol fakt, že pridanie jednej závislosti trvalo v Docker + WSL2 prostredí dlhšie, ako inštalácia všetkých závislostí v projekte. Medzi kvalitatívne vylepšenia patrí napríklad aj automatický reštart serveru po vykonaní zmien (cez WSL nefunguje, pretože vykonané zmeny v súboroch sa nepropagujú). Preto odporúčam vývojárom pracujúcich na tomto projekte na operačnom systéme Windows použiť práve tento spôsob.

Pri prebratí kódu taktiež nefungovali vo Windows `npm` príkazy `code:check` a `code:fix`, ktoré sú nastavené ako pre-commit hooks. Po oprave úvodzoviek v príkaze bol problém odstránený.

4.8.3.1 Potrebné zmeny v prípade alternatívnej konfigurácie

Aby to však fungovalo, je potrebné vykonať pár úprav.

Tabuľka 4.1: Porovnanie dĺžky vykonávania príkazov v konfigurácii Docker + WSL2 oproti Windows. Pre operáciu **úprava súboru** v prvej konfigurácii čas znamená reštartovanie celého kontajneru (zmeny sa nepropagujú). Testy boli vykonané na stroji s CPU Intel i7 6700HQ a 16 GB RAM a výsledky sú priemerom viacnásobných meraní.

Operácia	Čas Docker + WSL2	Čas Windows
yarn install (clean)	207,7 s	95,2 s
yarn add (pridanie 1 package)	279,3 s	15,8 s
yarn dev	32,3 s	9,2 s
úprava súboru	44,8 s	3,3 s

Docker V súbore `docker-compose.yaml` je potrebné nastaviť presmerovanie/vystavenie portov. Do objektu `postgres` sa pridá pole pod kľúčom `ports` s položkou `'5432:5432'`. Touto zmenou bude možné pristúpiť k DB cez `localhost:5432`. **!POZOR!** je však dôležité aby táto konfigurácia nebežala na vzdialenom serveri, čo by spôsobilo, že prístup k DB by bol možný aj z vonku. Ja som to vyriešil použitím alternatívneho changelistu vo WebStorm.

Env Server už nebude brať premenné prostredia zo súboru `.env`. Je teda nutné nastaviť tieto premenné pri spúšťaní serveru (Vo WebStorme jednoduchá úprava npm run konfigurácie). Taktiež je potrebné zmeniť URL databázy z `postgres` na `localhost`. Kontajner spúšťajúci server taktiež kontroluje hodnotu premennej `NO_SERVER`, ktorá v prípade rovnosti s hodnotou `true` zabezpečí vynechanie inštalovania závislostí a spúšťanie serveru (kvôli možnej nekompatibilite „*build*“-ovaných závislostí resp. obsadenosti portu).

MongoDB Pre konfiguráciu MongoDB je postup úprav obdobný.

4.8.3.2 Zadávanie príkazov

Pre všetky nasledujúce príkazy, ktoré súvisia so serverom alebo `yarn` platí nasledovné:

- Ak server beží v Docker kontajneri, príkazy sa spúšťajú zadaním `docker exec -it vmck-core-private-api {PRÍKAZ}` z hostiteľského stroja.
- Ak server beží lokálne vo Windows, príkazy sa zadávajú priamo v zložke s projektom.

4.9 Programátorská príručka

Z praxe je veľmi dobre známe, že orientácia v novom projekte je neľahká úloha. Samému mi trvalo niekoľko hodín, kým som získal predstavu, ako kód funguje a kde mám čo hľadať (spôsobené aj neexistujúcou predchádzajúcou skúsenosťou s Node.js). Rád by som aspoň pár riadkami ušetril cenný čas budúcim iteráciám pri orientovaní sa v projekte. Tento manuál neslúži ako kompletné, podrobné a formálne vysvetlenie kódu.

Vstupným bodom je súbor `server.ts`, ktorý iniciuje pripojenia k DB a následne spúšťa server v `app.ts`. V `app.ts` sa definuje port a middleware, ktorý bude použitý pre **každý** prichádzajúci request. Teraz prejdem postupom práce pri jednoduchom scenári – pridať novú entitu (s názvom `Manual`) spolu s endpointmi. Entity sú uložené v zložke `src/entity`. Vytvoríme novú triedu `Manual` s anotáciou `@Entity()` podľa manuálu TypeORM na [32]. Inšpirovať sa je možné aj už existujúcimi entitami. Pri vytváraní väzieb odporúčam definovať aj tzv. *join column*, ktorá obsahuje cudzí kľúč (často stačí len id cudzej entity, nemusíme využívať eager loading na načítanie a spájanie). Konvenciou v tomto projekte je využitie vzoru `Data Mapper`, čo znamená, že entita neobsahuje žiadnu logiku. Tá je obsiahnutá v triede `ManualModel` v zložke `src/models`. Následne vytvoríme v zložke `src/controllers` `ManualController`, ktorý bude špecifikovať endpointy definované pomocou anotácie `@Route`. Použitie tejto anotácie a jej častí som popísal v sekcii 4.4. Tento Controller by mal záznamy z DB získavať a upravovať pomocou `ManualModel`.

Následne už len ostáva vytvoriť testy v zložke `tests` a upraviť Swagger špecifikáciu v súbore `docs/swagger/private-api-swagger.yaml`.

Záver

V práci som sa zaoberal analýzou, návrhom, implementáciou a testovaním vylepšení časti *Private API* backendu v projekte VMCK.

Na začiatku práce, v kapitole Analýza, som zhodnotil aktuálny stav projektu, príručku na rozbehnutie serveru v lokálnom prostredí, existujúci návrh schvaľovacieho procesu a jeho čiastočnú implementáciu. Výstupom tohto zhodnotenia boli vytýčené nedostatky. Okrem spomenutých častí som sa rovnakým spôsobom venoval aj niektorým plánom do budúcna, ktoré boli uvedené v predchádzajúcich iteráciách. Na záver som tieto poznatky pretavil do funkčných a nefunkčných požiadavkov.

Tie tvorili vstup do nasledujúcej kapitoly, Návrh. V nej som uviedol niekoľko možných spôsobov realizácie opráv nedostatkov a novej funkcionality. K týmto spôsobom som uviedol klady a zápory, na základe ktorých som zvolil finálne riešenie vhodné na implementáciu.

V kapitole Realizácia som popísal, ako som niektoré navrhnuté časti implementoval. K tým, na ktoré sa viaže klientská aplikácia som popísal návod, ako takúto funkcionalitu zaviesť a používať, a taktiež som vytvoril aj osobitnú aplikáciu s fungujúcou ukážkou, ako by malo fungovať prihlasovanie cez Google. Potom som sa venoval aktuálnemu stavu testov, rozbehnutiu automatických testov a zavedeniu Continuous Integration a Continuous Delivery/Deployment. Na záver som spísal príručku pre programátorov, v ktorej som popísal, ako sa v aplikácii zorientovať, ako spojazdniť jednoduchý endpoint a pridal som aj niekoľko postrehov a odporúčaní z vývoja na operačnom systéme Windows 10 Home.

Ciele práce považujem za splnené. Na schvaľovací proces je v tejto chvíli možné napojiť klientské aplikácie a zaviesť ho do testovacej prevádzky.

Na záver, som rád, že môžem povedať, že pri písaní práce som si splnil aj svoj osobný cieľ, a to vyskúšať si vývoj backendu v technológii Node.js. Táto práca mi poskytla veľa cenných skúseností a takýto prínos pre obe strany považujem za obzvlášť hodnotný.

Literatúra

[1] Vančura, D.: *Věnná města českých královen - jádro*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

[2] Martinek, M.: *Administrační rozhraní pro projekt Věnná města českých královen*. Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

[3] Rajput, M.: Node.js Brings Revolution in App Development. Dostupné z: `https://www.mindinventory.com/blog/node-js-brings-revolution-in-app-development/`

[4] Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[5] Loewen, C.: Comparing WSL 1 and WSL 2. 2020. Dostupné z: `https://docs.microsoft.com/en-us/windows/wsl/compare-versions`

[6] Sůvová, D.: *Věnná města českých královen I. – úprava textur*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

[7] Zajíc, M.: *Věnná města českých královen I. - úprava textur*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

[8] Antoš, P.: *Věnná města českých královen – Webová aplikace pro schvalovací proces 3D modelů*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

[9] Anicas, M.: An Introduction to OAuth 2. 2014. Dostupné z: `https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2`

[10] Petrova, M.: Google's rocky path to email domination. 2019. Dostupné z: https://www.cnbc.com/2019/10/26/gmail-dominates-consumer-email-with-1point5-billion-users.html

[11] Redmond, T.: Half of Active Office 365 Users Now Use Teams. 2020. Dostupné z: https://office365itpros.com/2020/10/28/teams-115-million-users/

[12] Iqbal, M.: Twitter Revenue and Usage Statistics (2020). 2021. Dostupné z: https://www.businessofapps.com/data/twitter-statistics/

[13] GitHub: User search. 2021. Dostupné z: https://github.com/search?q=type:user&type=Users

[14] GitLab: Is it any good? 2021. Dostupné z: https://about.gitlab.com/is-it-any-good/

[15] Curry, D.: Discord Revenue and Usage Statistics (2021). 2021. Dostupné z: https://www.businessofapps.com/data/discord-statistics/

[16] Alikhanov, O.: Entities. Dostupné z: https://orkhan.gitbook.io/typeorm/docs/entities#simple-array-column-type

[17] Valenta, M.: Normalizace a normální formy. 2020. Dostupné z: https://courses.fit.cvut.cz/BI-DBS/materials/slides/hand-les09-normalizace.pdf

[18] Sivák, D.: Private API. 2021. Dostupné z: https://app.swaggerhub.com/apis/sivakdom/private-api/3.0.5#/

[19] Karanam, R.: REST API — What Is HATEOAS? 2019. Dostupné z: https://dzone.com/articles/rest-api-what-is-hateoas

[20] Google: Integrating Google Sign-In into your web app. 2020. Dostupné z: https://developers.google.com/identity/sign-in/web/sign-in

[21] Google: Authenticate with a backend server. 2020. Dostupné z: https://developers.google.com/identity/sign-in/web/backend-auth

[22] Klofáč, M.: *Věnná města českých královen - Schvalovací systém pro 3D modelu ve virtuální realitě*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

[23] Auth0: node-auth0. Dostupné z: https://www.npmjs.com/package/auth0

[24] Marak: faker.js - generate massive amounts of fake data in the browser and node.js. Dostupné z: http://marak.github.io/faker.js/

[25] express validator: Validation Chain API. 2021. Dostupné z: https://express-validator.github.io/docs/validation-chain-api.html

[26] express validator: Schema Validation. 2021. Dostupné z: https://express-validator.github.io/docs/schema-validation.html

[27] Fowler, M.: Continuous Integration. 2006. Dostupné z: https://www.martinfowler.com/articles/continuousIntegration.html

[28] Sijbrandij, S.: Upcoming changes to CI/CD Minutes for free tier users on GitLab.com. 2020. Dostupné z: https://about.gitlab.com/blog/2020/09/01/ci-minutes-update-free-users/

[29] GitLab: Configuration of your jobs with .gitlab-ci.yml. Dostupné z: https://gitlab.fit.cvut.cz/help/ci/yaml/README.md

[30] Fowler, M.: ContinuousDelivery. 2013. Dostupné z: https://martinfowler.com/bliki/ContinuousDelivery.html

[31] rclone: Rclone syncs your files to cloud storage. Dostupné z: https://rclone.org/

[32] TypeORM: TypeORM. Dostupné z: https://typeorm.io/

Dodatok **A**

Zoznam použitých skratiek

GUI Graphical user interface

XML Extensible markup language

JSON Javascript Object Notation

REST Representational state transfer

Dodatok **B**

Obsah priloženého média

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `impl` ..... zdrojové kódy implementácie
  - `thesis` ..... zdrojová forma práce vo formáte LaTeX
  - `google_test_app` ... zdrojový kód testovacej aplikácie pre prihlásenie cez Google
  - `latex_gitlab_ci` ..... konfigurácia pre automatické zostavenie LaTeX dokumentu cez GitLab CI
- `tests` ..... výsledky testov vo formáte HTML
  - `test-report-before.html` ..... výsledky testov po prevziatí kódu
  - `test-report-before-fixed.html` ..... výsledky testov po prevziatí kódu a úprave testov
- `text` ..... text práce
  - `thesis.pdf` ..... text práce vo formáte PDF