Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Continuous Integration pro JS stack v GitLabu

Plzeň 2023 Martin Forejt

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Martin FOREJT**
Osobní číslo:	**A20N0079P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Continuous Integration pro JS stack v GitLabu**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s pojmem Continuous Integration (CI) v prostředí GitLab.
2. Proveďte analýzu nástrojů CI pro JavaScript stack tak, aby bylo možno pro každý merge request ověřit, že:
   - proběhl úspěšný build pomocí nástroje Webpack,
   - nejsou detekovány žádné chyby a varování nástrojem ESLint nebo jiným (existuje-li lepší),
   - všechny jednotkové a integrační testy proběhly úspěšně pomocí frameworku Mocha nebo jiným (vyhovuje-li více).
3. Prozkoumejte existenci dalších potenciálně užitečných nástrojů pro CI pipeline. Porovnejte a zhodnoťte klady a zápory jednotlivých řešení.
4. Navrhněte demonstrační aplikaci pro evidenci pracovní doby zaměstnanců na projektech.
5. Implementujte navrženou aplikaci s využitím vybraných nástrojů CI.
6. Zhodnoťte navržené řešení a uveďte jeho klady, zápory a možnosti rozšíření.

Rozsah diplomové práce: **doporuč. 50 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce: **Ing. Martin Dostal, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání diplomové práce: **9. září 2022**
Termín odevzdání diplomové práce: **18. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2022

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V diplomové práci jsou použity názvy programových produktů, firem apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

V Plzni dne 15. května 2023

Martin Forejt

Abstract

This thesis focuses on the topic of continuous integration and continuous deployment in the GitLab CI/CD tool for automating testing, deployment, and other processes in the development of JavaScript applications. The goal was to analyze and select suitable tools for use in continuous integration, not only for automatic application building and testing. Using the selected tools, automatic processes were implemented and configured for use in GitLab CI/CD. In addition, a sample JavaScript application was designed and implemented, on which the use of selected tools was verified to streamline the software development process.

Abstrakt

Tato diplomová práce se zaměřuje na téma kontinuální integrace a kontinuálního nasazení v prostředí GitLab CI/CD pro automatizaci testování, nasazení a dalších procesů při vývoji JavaScript aplikací. Cílem bylo provést analýzu a výběr vhodných nástrojů pro využití v kontinuální integraci a to nejen pro automatické sestavení a testování aplikace. S využitím vybraných nástrojů byly implementovány a konfigurovány automatické procesy pro použití v GitLab CI/CD a také byla navržena a implementována ukázková JavaScript aplikace, na které bylo ověřeno využití vybraných nástrojů s cílem zefektivnit proces vývoje softwaru.

Obsah

1 Úvod

V současné době jsou softwarové projekty a jejich vývoj stále složitější a vývojářské týmy tak čelí rostoucím výzvám, které vyžadují neustálé zlepšování vývojového procesu. Jedná se např. o zajištění kvality a bezpečnosti softwaru a jeho rychlé dodání. Jedním z klíčových aspektů tohoto vývojového procesu je implementace kontinuální integrace (CI) a kontinuálního nasazení (CD), což umožňuje automatizovat testování a nasazení nových verzí produktu. Tato diplomová práce se zaměřuje na implementaci nástrojů CI/CD pro programovací jazyk JavaScript v prostředí GitLab a jejím cílem je provést analýzu užitečných nástrojů nejen pro sestavení a testování JavaScript aplikací v CI/CD, které pomohou zefektivnit proces vývoje softwaru, ale také navrhnout a implementovat ukázkovou aplikaci a při tom vybrané nástroje pro CI/CD použít a implementovat.

V úvodu práce (kapitole 2) je čtenář seznámen s problematikou životního cyklu vývoje softwaru a jeho modely, které definují určitý přístup k vývoji softwaru a specifikují činnosti v jeho jednotlivých fázích. Pozornost je věnována především modelu DevOps, mezi jehož základní principy patří právě využití kontinuální integrace a kontinuálního nasazení.

V kapitole 3 představena platforma GitLab, její historie, architektura, řešení a funkce které nabízí se zaměřením na nástroj GitLab CI/CD a možnosti konfigurace tzv. pipeline - posloupnosti automatických akcí vykonávaných v CI/CD.

V kapitole 4 je stručně popsán programovací jazyk JavaScript, jeho využití, nadstavby a knihovny. Je provedena analýza užitečných nástrojů a knihoven pro použití v automatizovaném procesu CI/CD pro sestavení, testování a další.

V kapitole 5 je navržena a implementována ukázková JavaScript aplikace pro evidenci pracovní doby zaměstnanců na projektech. S využitím nástrojů vybraných z předešlé analýzy jsou v kapitole 6 vzneseny požadavky na GitLab CI/CD pipeline pro automatické sestavení, testování a nasazení aplikace. Pipeline a její konfigurace je optimalizována, podrobně popsána a v závěru práce (v kapiole 7) je ověřeno a otestováno, zda splňuje požadavky a cíle práce.

Práce by měla poskytovat ucelený návod a přehled možností pro vývojáře a týmy, kteří chtějí zefektivnit vývoj svých JavaScriptových aplikací díky implementaci CI/CD procesů.

2 Životní cyklus vývoje softwaru

Životní cyklus vývoje softwaru (SDLC) popisuje proces vývoje softwaru a jeho dělení na jednotlivé části [12]. Tato kapitola, která je úvodem do problematiky vývoje softwaru a SDLC, popisuje jednotlivé fáze životního cyklu softwaru od tvorby požadavků až po nasazení do provozu a údržbu. Dále stručně popisuje vybrané modely SDLC přístupu k vývoji softwaru.

2.1 Fáze SDLC

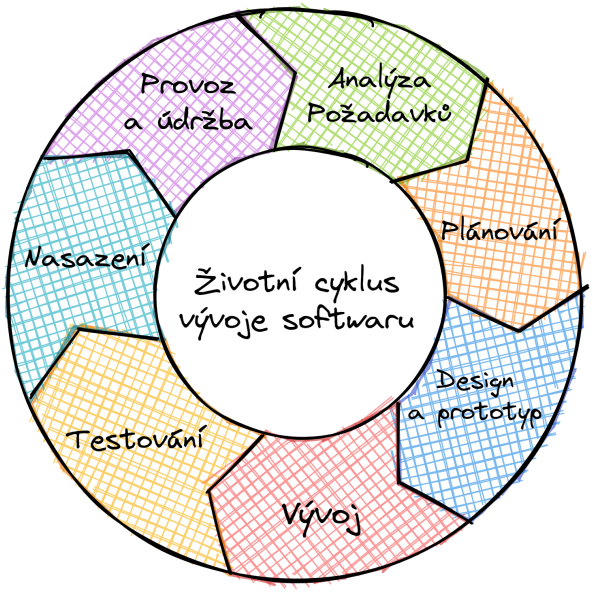
SDLC se skládá z několika fází, které na sebe navazují a případně (v závislosti na použité metodice) se opakují. Přesný počet a povaha jednotlivých fází záleží na použité metodice. Dle [17] však většina společností definuje SDLC s pěti až sedmi fázemi a uvádí těchto sedm typických fází (viz obrázek 2.1): analýza požadavků, plánování, design a prototypování, vývoj, testování, nasazení a provoz a údržba.

V první fázi, fázi analýzy požadavků, se získávají všechny potřebné informace od zákazníka, aby mohl být vytvořen produkt, který naplní jeho požadavky a očekávání. Je potřeba si ujasnit, co je cílem, jaký je účel produktu, kdo budou jeho koncoví uživatelé atd. Tyto informace, které jsou typicky zaneseny do dokumentu specifikací požadavků (DSP), je potřeba znát před započetím vývoje produktu [17, 33].

Ve fázi plánování je provedena studie proveditelnosti dle DSP, jejímž výstupem jsou odhadované náklady na zdroje (finanční, lidské, časové, HW,...) a harmonogram dílčích částí vývoje projektu [17].

Po plánování následuje fáze designu a prototypování, ve které vzniká dokument specifikace návrhu, který specifikuje zásadní aspekty projektu jako je architektura (SW, HW), seznam funkcí, návrh grafického rozhraní, zabezpečení a další. Zároveň je možné v této fázi vytvářet prototypy, které jsou sice časově náročné, ale mohou odhalit chyby v návrhu, jejichž náprava ve fázi vývoje by byla na zdroje mnohem nákladnější [17].

Samotné psaní kódu a tvorba produktu dle DSP a návrhu následuje ve fázi vývoje. I přesto, že produkt bude ověřován a testován až v následující fázi, již během vývoje by měl být kladen důraz na psaní kvalitního a dobře testovatelného zdrojového kódu. Tato fáze je časově nejnáročnější ze všech



Obrázek 2.1: Fáze SDLC

fází SDLC a jejím výstupem je testovatelný softwarový produkt, který splňuje požadavky specifikované v předchozích fázích [17].

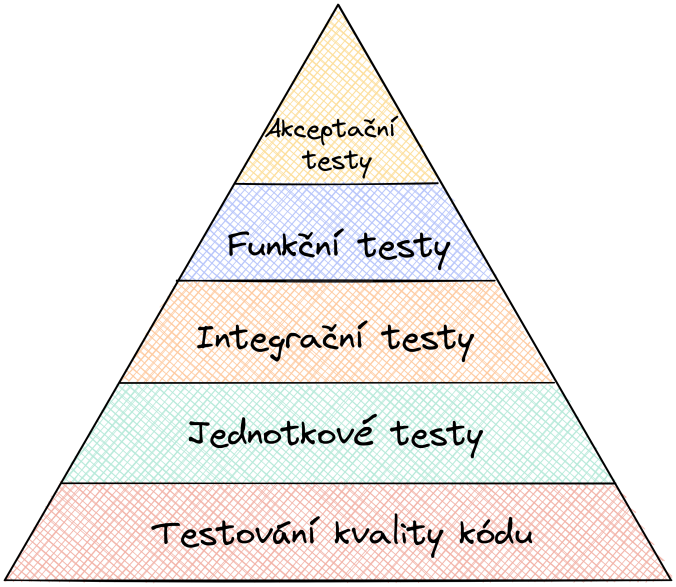
Aby se ověřilo, že software, který je výstupem z fáze vývoje, vyhovuje všem specifikovaným požadavkům a neobsahuje chyby, musí projít testováním. Existuje mnoho způsobů a metod testování viz dále. V případě, že testy odhalí chyby, nastává (dle konkrétní metodiky) krok zpět na fázi vývoje a vývojáři vytvoří novou verzi, která nalezené chyby opravuje a která je znovu podrobena fázi testování. Testovací fáze končí, když je výsledný produkt stabilní, bez chyb a splňuje všechny požadavky specifikované v předchozích fázích [17].

V případě, že je produkt důkladně otestován, je vše připraveno aby byl nasazen do produkčního prostředí a zpřístupněn jeho uživatelům. Nasazení, stejně jako testování, může probíhat automaticky a v případě, že software projde automatickým testováním bez chyb, může být automaticky nasazen. Tomuto přístupu se říká kontinuální nasazení (CD) a je více probráno v kapitole 2.3.5. V této fázi mohou být také vytvářeny uživatelské manuály, které slouží koncovým uživatelům produktu, aby věděli jak ho ovládat a jak s ním pracovat.

Po fázi nasazení je již produkt dostupný koncovým uživatelům, kteří ho používají. Zde ale práce na produktu nekončí. V této fázi je potřeba běh softwaru monitorovat a odhalovat nové chyby. Také by měla být zpracovávána zpětná vazba od uživatelů, kteří mohou upozorňovat na chyby, nebo dokonce dávat podněty pro vytvoření nových funkcionalit.

2.1.1 Testování

Pro ověřování, zda software splňuje specifikované požadavky a zda neobsahuje chyby, se používá mnoho testovacích způsobů a metod (viz obrázek 2.2):



Obrázek 2.2: Typy testování

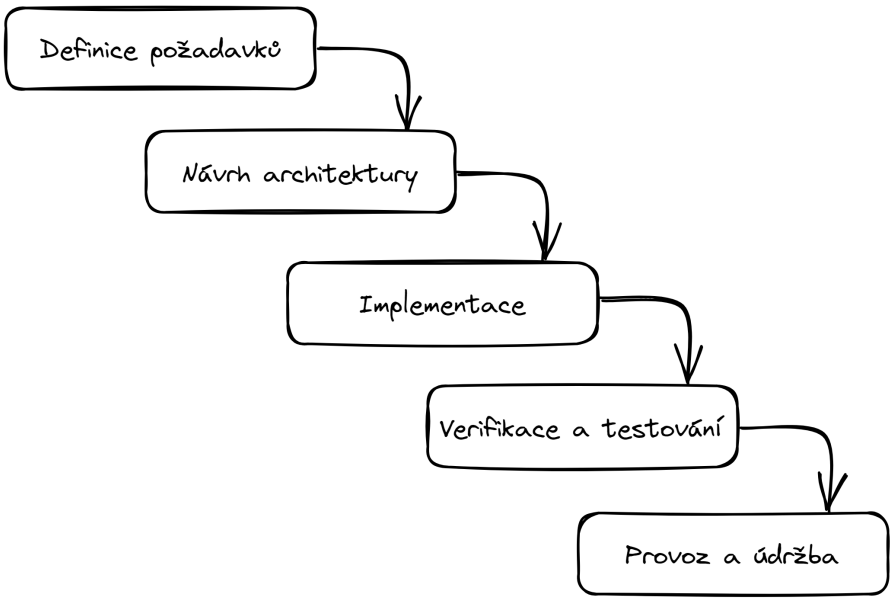
- **Testování kvality kódu** - ověřuje kvalitu kódu během statické analýzy pomocí programů zvaných *Linter*. Název pochází od programu *Lint*, který v roce 1978 vytvořil Stephen C. Johnson jako nástroj pro analýzu zdrojového kódu v programovacím jazyce C. Statická analýza dokáže odhalit chyby v syntaxi a potenciálně nebezpečné vzory v kódu, či udržovat konzistentní styl kódu napříč celým projektem. Některé problémy mohou být opraveny automaticky a kód může být programem naformátován.
- **Jednotkové testy** - ověřují funkcionalitu samostatných softwarových částí (jednotek), jako jsou funkce, třídy a metody a jsou zpravidla psány přímo vývojáři dané jednotky. Závislosti na ostatních částech softwaru se obvykle předávají v podobě tzv. mock objektů, které chování dané části napodobují bez nutnosti spuštění celého softwaru.
- **Integrační testy** - ověřují, zda jednotlivé komponenty systému (které již byly otestovány jednotkovými testy) fungují správně i při vzájemné kooperaci.

- **Funkční testy** - také označovány jako E2E testy (End to End), testují jednotlivé funkce systému z pohledu uživatele. Při těchto testech se similuje kompletní produkční prostředí, systém je spuštěn celý a nástroje pro automatické testování dokáží simulovat vstup od uživatele přes uživatelské rozhraní.
- **Akceptační testy** - tyto testy ověřují, zda kompletní výsledný produkt splňuje specifikaci a odpovídá všem požadavkům od zadavatele. Tyto testy obvykle (na rozdíl od všech předchozích) probíhají již na straně zadavatele.

2.2 Modely

Konkrétní fáze a činnosti v nich mohou být v každé společnosti různé i v závislosti na specifikách vyvíjeného produktu. Přesto je definováno mnoho obecných přístupů, které se během procesu vývoje softwaru používají. Tyto přístupy, či metodiky se označují také jako modely SDLC [4]. Dále budou probrány některé vybrané modely SDLC a to: vodopádový model, V-model, iterativní model, inkrementální model, spirálový model, agilní model, a model DevOps.

2.2.1 Vodopádový model



Obrázek 2.3: Vodopádový model

Vodopádový model, někdy také označován jako lineárně sekvenční model, je model SDLC, který má jasně definované, v sekvenci vykonávané tyto fáze [4] (viz obrázek 2.3): definice požadavků, návrh architektury, implementace, verifikace a testování a fáze provozu a údržby.

Ve vodopádovém modelu musí být každá fáze kompletně dokončena, než se uskuteční přechod na další fázi a nikdy nenastane krok zpět na fázi předchozí. Z toho důvodu je tento model málo flexibilní a nedokáže reagovat na změny požadavků. Vodopádový model je považován za nejstarší metodiku SDLC, avšak dle některých odborníků by nikdy neměl být použit pro skutečné projekty [6].

2.2.2 V-model

Tento model, někdy označován jako model verifikace a validace, je rozšířením vodopádového modelu. Jeho fáze jsou také vykonávány v sekvenci a každá fáze může začít, až pokud je fáze předchozí kompletně dokončena. U V-modelu je každé jednotlivé fázi, která je vykonávána shora dolů, přiřazena fáze testovací vykonávána zdola nahoru. Tím vzniká tvar připomínající písmeno V, odkud získal tento model svůj název [4].

Pokud první fází bude např. analýza požadavků a jejím výstupem dokument specifikace požadavků. Již v tomto bodě, než se první fáze dokončí a přejde se na fázi další (např. návrh architektury), jsou navrženy testy, které tuto fázi ověří, až se bude postupovat zpět nahoru. V tomto případě to budou akceptační testy a budou vykonány jako poslední fáze.

2.2.3 Iterativní model

U iterativního modelu je software vytvářen postupně během několika iterací, kdy každá iterace může probíhat stejně jako v případě vodopádového modelu. V každé iteraci je vytvořena malá část softwaru a takto se postupně vytváří komplexní produkt po malých částech, dokud není celý a kompletní [4].

Iterativní model nepožaduje mít na začátku kompletní veškeré specifikace, ale stačí mít specifikovanou pouze část produktu, která se v několika iteracích implementuje a poté se specifikují další části. Stejně tak je tento model mnohem flexibilnější než modely předchozí, protože dovoluje specifikaci měnit v průběhu vývoje mezi jednotlivými iteracemi. Produkt či jeho části mohou být také nasazeny již v průběhu vývoje a lze tak pružně reagovat na zpětnou vazbu od uživatelů.

2.2.4 Inkrementální model

Inkrementální model rozděluje výsledný produkt na několik částí (typicky dle funkcionalit). Každé z těchto částí, které jsou také nazývány jako inkrementy, jsou vyvíjeny postupně a samostatně a každý inkrement je na konci své fáze dostatečně otestován. Výsledný produkt se tedy postupně vytváří z jednotlivých inkrementů a po každém inkrementu může být nasazen, aby se zpřístupnil k otestování i uživatelům a tým mohl získat zpětnou vazbu a případně upravovat specifikaci během vývoje [4].

Inkrementální model je podobný předchozímu iterativnímu modelu. Rozdíl mezi těmito dvěma modely je v tom, že iterativní model tvoří celý produkt najednou, ale postupně ho pomocí jednotlivých iterací zdokonaluje, dokud není celý produkt kompletní. Naproti tomu inkrementální model netvoří celý produkt najednou, ale po částech (např. po jednotlivých funkcích), které jsou vždy kompletní a postupně se přidávají další a další funkce, dokud není produkt kompletní.

2.2.5 Spirálový model

Tento model je podobný inkrementálnímu modelu, ale klade větší důraz na analýzu rizik. Spirálový model má čtyři fáze: plánování, analýzu rizik, vývoj a zhodnocení.

Vývoj softwaru opakovaně prochází těmito fázemi v iteracích, které jsou v tomto modelu také nazývány jako spirály. Každá další spirála staví na předchozí a na konci každé spirály je dostupný prototyp. Na rozdíl od inkrementálního modelu nemusí být prototyp schopen provozu a nasazení.

2.2.6 Agilní model

Agilní metodika není přímo model SDLC, ale spíše přístup či skupina metod vývoje softwaru založených na iterativním a inkrementální vývoji, kde se požadavky a řešení vyvíjejí ve spolupráci mezi více samostatnými týmy [4].

Agilní metodika je flexibilní přístup, který podporuje adaptivní plánování, evoluční vývoj, časově ohraničený iterativní přístup a rychlou reakci na změny. Mezi základní prvky této metody patří také komunikace v týmu, flexibilita a schopnost přizpůsobit se, když se změní požadavky [4].

Jak již bylo zmíněno, agilní metodika vývoje je spíše koncepční rámec a existuje řada specifičtějších agilních modelů, mezi které patří např. Scrum, Kanban, Feature Driven Development či extrémní programování [4].

2.3 DevOps

Metodika DevOps je relativně novým přístupem k SDLC. Vznikla díky trendu aplikace agilních metod nejen na vývoj, ale i na provozní práci a obecnému posunu ke spolupráci mezi vývojovým a provozním oddělením ve všech fázích procesu SDLC [6]. Zkratka vznikla ze spojení výrazů Development (vývoj) a (IT) Operations (operace v provozu). V modelu DevOps vývojáři a provozní týmy úzce spolupracují, nebo fungují jako jeden tým, s cílem urychlit vývoj a nasazení kvalitnějších a spolehlivějších softwarových produktů a funkcí. Aktualizace produktů jsou malé, ale časté. Charakteristickými znaky modelu DevOps jsou neustálá zpětná vazba a zlepšování a automatizace manuálních vývojových procesů [6]. DevOps není nástroj nebo technologie, ale spíše soubor terminologií a může být považován za koncept, přístup, kulturu či filozofii, která zlepšuje celý životní cyklus vývoje a provozu softwaru [15]. V této podkapitole budou probrány základní principy, nástroje a koncepty které se ve spojení s touto metodou používají, včetně hlavního tématu této práce - kontinuální integrace.

2.3.1 Důvody k zavádění DevOps

Pokud je za použití tradičních metod SDLC vyvíjena nová funkce či oprava chyby, probíhají obvykle tyto kroky [15]:

1. Vývojový tým píše kód a implementuje novou funkci či opravu chyby. Nová verze softwaru je nasazena do vývojového prostředí a otestována vývojovým týmem.

2. Nová verze je předána QA[1] týmu a nasazena do QA prostředí, kde je otestována testery.

3. Otestovaná verze je předána provoznímu týmu, který ji nasadí do produkčního prostředí a je k dispozici koncovým uživatelům.

4. Provozní tým je zodpovědný za správu a údržbu produkční verze.

Tento přístup může mít ale různé možné problémy [15]:

- Nasazení nové verze od vývojového týmu až do produkčního prostředí může trvat týdny až měsíce.

[1] QA - Quality Assurance, zajištění jakosti

- Priority vývojového týmu, QA týmu a provozního týmu jsou odlišné a často chybí efektivní a účinná koordinace mezi týmy, která je důležitá pro hladký provoz. Vývojový tým se plně soustředí na nejnovější verzi ve vývoji, zatímco provozní tým se zajímá o stabilitu produkčního prostředí.
- Vývojový a provozní tým často nic neví o práci a pracovní kultuře druhého týmu.
- Každý tým pracuje v odlišném prostředí a spravuje různé konfigurace. Software, instalační a konfigurační soubory, databázové skripty, to vše může fungovat bezchybně v lokálním nebo vývojovém prostředí, ale v produkčním mohou nastat neočekávané problémy.
- Nastavení prostředí, konfigurace, a jednotlivé aktivity při nasazování nových verzí vyžadují ruční práci, která není snadno opakovatelná a je náchylná k chybám. Proces nasazování tak navíc musí být zdokumentován pro použití v budoucnosti a tvorba a udržování dokumentace jsou časově náročné.

Metoda DevOps dokáže těmto problémům předcházet díky základním principům, na kterých je založena a také díky nástrojům a postupům, které se k zavedení těchto principů používají (budou popsány v následujících kapitolách). To s sebou přináší vyšší výkon vývojových týmu, rychlejší vytváření lepších produktů a spokojenost zákazníků [20]. Hlavními výhodami DevOps jsou:

- Zkrácení vývojového cyklu
- Rychlé dodání nových verzí
- Lepší kolaborace v týmu
- Vyšší spolehlivost a menší chybovost kódu

2.3.2 Základní principy DevOps

Oproti přístupu v tradičních modelech SDLC popsanému výše, DevOps dříve oddělené týmy (vývojový a provozní) spojuje v jednu jednotku, která vytváří bezpečný kód a zároveň zlepšuje zrychluje životní cyklus vývoje softwaru. Mezi základy DevOps patří kultura spolupráce a komunikace, automatizované testování, vydávání a nasazování a časté iterace [23].

Jádrem metodiky DevOps jsou tyto klíčové principy, které mohou životní cyklus vývoje softwaru zlepšit [23]:

- **Iterativní přístup** - DevOps využívá stejné principy jako iterativní či agilní model, jejichž hlavní myšlenkou je vyvíjet software v krátkých iteracích, kde každá iterace zahrnuje jednotlivé fáze SDLC a po každé iteraci je k dispozici vylepšená verze produktu.

- **Kontinuita** - pojmy jako kontinuální vývoj, kontinuální testování či kontinuální nasazení popisují myšlenku, že každý krok v procesu vývoje je během něj prováděn nepřetržitě.

- **Automatizace životního cyklu vývoje softwaru** - před zavedením metodiky DevOps zahrnovaly všechny fáze vývoje softwaru lidský (manuální) zásah. Automatizace umožňuje společnostem vydávat nové verze klidně několikrát denně. Filozofií DevOps je automatizovat co nejvíce procesů SDLC, ať už jde o testování, slučování kódů, nasazování a další.

- **Spolupráce a komunikace** - nejen mezi vývojovými a provozními týmy, ale také mezi týmy starající se o testování, bezpečnost a další. Vzhledem k tomu, že každý tým má jiné priority a řeší problémy z jiné perspektivy, je pro spolupráci nezbytná kvalitní, častá a transparentní komunikace.

- **Neustálé zlepšování a minimalizace plýtvání zdroji** - kromě zmíněné automatizace opakujících se činností, která šetří čas a snižuje počet manuálních kroků nutných k vydání nové verze, by DevOps týmy měly neustále sledovat výkonnostní metriky, aby mohly odhalit oblasti, které potřebují další zlepšení.

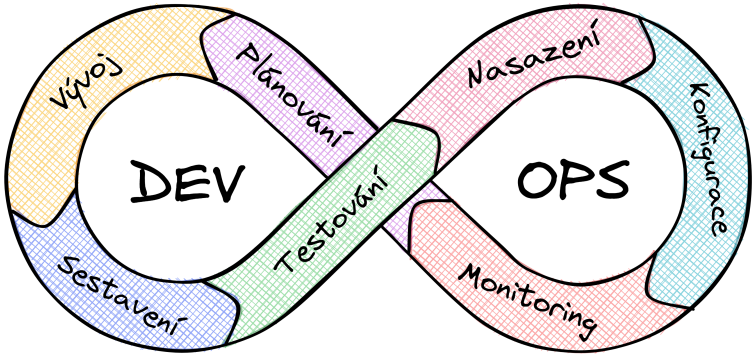
- **Zaměření na potřeby uživatele** - v každém procesu vývoje softwaru by se měly brát v úvahu potřeby skutečného uživatele produktu a týmy by měly hledat způsoby, jak tyto potřeby naplnit.

- **Krátké zpětnovazební smyčky** - s potřebami uživatele souvisí i zpětná vazba, která by měla být co nejrychleji zpracována (aby nedocházelo k plýtvání s časem) a smyčka k vyřešení zpracované zpětné vazby by měla být také co nejrychlejší.

- **Posun v rozsahu a zodpovědnostech** - díky tomu, že se týmy zapojují i do dalších fází životního cyklu, které nejsou pro jejich role klíčové, získávají za ně zodpovědnost. Vývojáři začnou být např. zodpovědní i za výkon a stabilitu, které jejich změny přinesou v provozní fázi.

2.3.3 Životní cyklus DevOps

Jak již bylo zmíněno, metoda DevOps využívá iterativního či agilního přístupu k vývoji softwaru a jednotlivé fáze SDLC se opakují v krátkých iteracích s cílem mít po každé iteraci k dispozici novou verzi produktu. Každá iterace zahrnuje jednotlivé fáze SDLC, ale díky kladení důrazu na kontinuitu nejsou hranice mezi jednotlivými fázemi u DevOps pevně dané a fáze jsou do jisté míry prováděny souběžně.



Obrázek 2.4: Fáze životního cyklu v DevOps

Mezi fáze životního cyklu DevOps patří (obrázek 2.4) [22]:

- **Plánování** - ve fázi plánování se organizuje práce, co je potřeba udělat, jednotlivé úkoly se prioritizují a monitoruje se jejich postup až k dokončení.
- **Vývoj** - v této fázi se tvoří návrhy a píše kód, který pochází od více vývojářů z týmu a musí se slučovat.
- **Sestavení** - z kódu a všech závislostí je sestaven spustitelný artefakt.
- **Testování** - ve fázi testování se ověřuje, zda kód funguje správně a zda splňuje nároky na kvalitu kódu a to ideálně pomocí automatického testování.
- **Nasazení** - otestovaný software je možné nasadit a zpřístupnit tak koncovým uživatelům.
- **Konfigurace** - v této fázi se spravuje a konfiguruje infrastruktura a prostředí, ve kterém produkt běží.
- **Monitoring** - běh softwaru v produkčním prostředí se neustále monitoruje, aby bylo možné co nejdříve odhalit potenciální problémy.

2.3.4 Nástroje a koncepty DevOps

Vybrané technologie, nástroje a koncepty zmíněné níže umožňují týmům naplnit principy DevOps ve všech jeho fázích a díky automatizaci se plně soustředit na kreativní práci:

- **Systémy pro správu verzí** - umožňují sdílení kódu mezi mnoho vývojáři. Kód je typicky uložen v centralizovaném úložišti a každý vývojář pracuje se svou lokální kopií. Systém poskytuje funkce pro synchronizaci kódu a identifikaci konfliktních úprav stejných částí kódu více vývojáři. Díky tomu mohou vývojáři pracovat na jednom projektu najednou, což vede ke zvýšení rychlosti vývoje. DevOps využívá repozitářů Git (viz kapitola 2.3.6), který nabízí více možností díky modernější architektuře oproti starším systémům [22].

- **Kontinuální integrace** - (CI - Continuous Integration) je takový přístup k integraci změn v kódu do hlavní (produkční) verze (označováno jako větev v Gitu), že integrace probíhá co nejčastěji po malých částech a po každé změně v kódu jsou automaticky spuštěny procesy, které kód otestují a sestaví. Kontinuální integrace přináší především efektivitu. Díky automatizaci ruční práce a testování změn v kódu je možné zkrátit iterace a rychleji a častěji nasazovat nové funkce s menším počtem chyb [22].

- **Kontinuální dodávka** - kontinuální dodávka (CD - Continuous Delivery) je krok, který následuje kontinuální integraci. Nejen že je kód po každé změně otestován a sestaven, ale sestavený software je také kontinuálně nasazován. Kontinuální dodávka tak připraví artefakty, které je možné nasadit do produkčního prostředí [22].

- **Kontinuální nasazení** - kontinuální nasazení (CD - Continuous Deployment), používá stejnou zkratku jako kontinuální dodávka a je snadné je považovat za stejný proces, ale rozdíl mezi nimi je. Kontinuální dodávka pouze připraví artefakt, nasaditelný do produkčního prostředí, ale stále je potřeba ruční zásah (typicky pouze kliknutí na jedno tlačítko), který nasazení provede. To umožňuje strategickou volbu, kdy skutečně k nasazení dojde. Kontinuální dodávka jde ještě o krok dále a nasazení proběhne automaticky po změně kódu v produkční větvi. Lze tak říci, že kontinuální nasazení zahrnuje kontinuální dodávku [22].

- **Automatické testování** - klíčovým prvkem k plné adopci DevOps a kontinuální integrace je automatické testování. Díky kontinuální integraci je kód po každé změně automaticky sestaven a následně jsou

spouštěny testy, které ověří, že změny vyhovují nastaveným pravidlům pro kvalitu kódu a projdou všemi testy (jednotkové, integrační,..). Všechny chyby jsou tak odhaleny co nejdříve a díky tomu, že se změny integrují po malých částech, je jednodušší odhalit a vyřešit jejich příčinu [22].

- **Kontinuální monitoring** - kontinuální monitoring nepřetržitě zpracovává zpětnou vazbu a to jak z automatického monitoringu softwaru, kdy data přicházejí z prostředí, kde software běží, z logů apod., ale také zpětnou vazbu přicházející v různých formách přímo od uživatelů [13].

- **Infrastruktura jako kód** - správa IT infrastruktury často zahrnuje manuální procesy, které vyžadují konfiguraci fyzických serverů. Infrastruktura jako kód (IaC - Infrastructure as Code) umožňuje konfigurovat infrastrukturu prostředí pomocí definice v kódu. Konfigurační soubor s těmito definicemi se nachází ve stejném repozitáři jako vlastní kód softwaru a díky tomu je možno spravovat více verzí infrastruktury a konfigurace prochází všemi fázemi kontinuální integrace stejně jako jakýkoliv jiný kód [13].

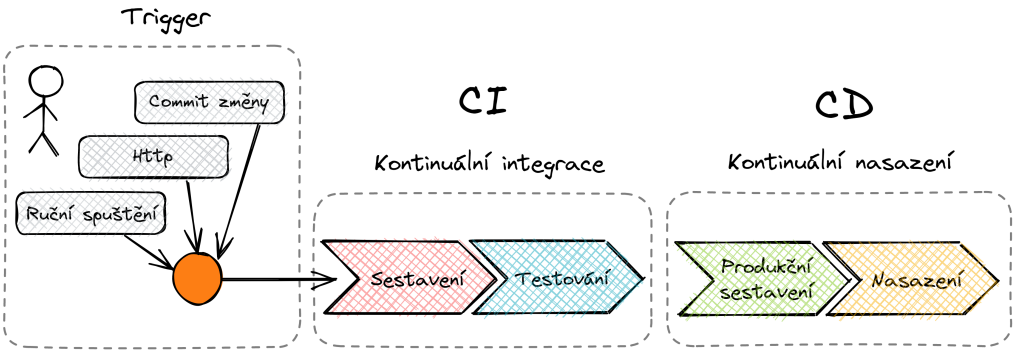
- **Kontejnery** - DevOps často využívá kontejnery, což jsou softwarové balíčky, které obsahují vše, co potřebuje software pro svůj běh - virtuální operační systém se všemi potřebnými knihovnami, konfigurací, a další. Díky tomu nemusí vývojáři řešit kompatibilitu mezi různými prostředími. Populárními technologiemi pro práci s kontejnery jsou Docker či Kubernetes [22].

2.3.5 CI/CD

Pojem CI/CD, který je spojením zkratek CI a CD je proces, který zahrnuje kontinuální integraci, kontinuální dodávku a/nebo kontinuální nasazení.

Základním prvkem CI/CD je tzv. pipeline (obrázek 2.5), která představuje sérii kroků či akcí (stage a joby), které jsou provedeny po spuštění nějakou událostí (triggerem), typicky nahráním změny v kódu (tzv. commit) do systému správy verzí. Systém pro správu verzí, který je úložištěm repozitáře poté notifikuje CI/CD platformu např. pomocí webhooku (HTTP request). Pipeline obsahuje jednotlivé akce z kontinuální integrace jako je sestavení a spuštění testů, které mohou následovat akce z kontinuální dodávky či kontinuálního nasazení [8].

Většina CI/CD platforem využívá principu *pipeline jako kód* pro konfiguraci pipeline, kdy je tato konfigurace uložena ve stejném repozitáři jako



Obrázek 2.5: CI/CD pipeline

vlastní zdrojový kód. Tento přístup má tu výhodu, že konfigurace může být upravována jakýmkoliv členem týmu, verzována a kontrolována jako každý jiný kód. Konfigurační soubor, který bývá typicky ve formátu YAML a musí dodržovat specifikace konkrétní platformy, definuje jednotlivé akce, prostředí pro sestavení softwaru a další parametry [8].

Binární soubory, dokumentace nebo archivy vygenerované v průběhu či jako konečné výsledky běhu pipeline jsou nazývané artefakty. Artefakty jsou persistentní po skončení běhu pipeline a lze s nimi dále pracovat. Příkladem mohou být reporty výsledků testů dostupné ke kontrole.

Platformy

CI/CD platformy poskytují nástroje pro implementaci kontinuální integrace a kontinuálního nasazení. Tyto platformy nabízí více či méně možností, jak propojit systém pro správu verzí s testovacími nebo jinými nástroji kontinuální integrace a kontinuálního nasazení. Populární CI/CD platformy jsou [11]:

- **Jenkins** - je open-source server pro automatizaci CI/CD procesů, který nabízí stovky pluginů pro lepší podporu sestavení a nasazení projektů. Jenkins je třeba nainstalovat na vlastní server (není k dispozici SaaS[2] možnost) a (typicky pomocí pluginu) napojit na systém správy verzí, který ho upozorňuje o nově nahraných změnách v kódu.
- **Travis CI** - je služba, která nabízí CI/CD nástroje a podporuje napojení na tyto systémy správy verzí: Github, Bitbucket, GitLab a Assembla.
- **CircleCI** - služba s certifikací FedRAMP[3], která nabízí CI/CD ná-

[2]SaaS - Software as a Service
[3]`https://www.fedramp.gov/`

stroje a zaměřuje se na výkon a rychlost vykonávání akcí pipeline.

- **TeamCity** - nástroj pro CI/CD od firmy JetBrains, který mimo jiné nabízí integraci do IDE (integrované vývojové prostředí), ve kterém vývojáři přímou píší kód.
- **Github Actions** - nástroj Github Actions je implementován přímo do služby Github, která poskytuje (pouze) cloudové úložiště a další nástroje pro správu verzí systému Git. Správa pipeline tak probíhá ve stejném prostředí jako správa kódu a jeho verzí. Jedná se o celosvětově nejpoužívanější CI/CD nástroj [11].
- **GitLab CI/CD** - GitLab je podobně jako Github nástroj pro správu repozitářů Git, který nabízí další funkce včetně CI/CD. Protože tato práci je zaměřena právě na GitLab, bude tato platforma detailněji probrána v další kapitole.
- **Bitbucket Pipeline** - Bitbucket je nástroj podobný předchozím a implementuje tak podporu CI/CD přímo do nástroje pro správu verzí systémů Git a Mercurial.
- **Azure DevOps** - je nástroj pro kompletní podporu vývoje s DevOps a podobně jako v předchozích případech lze v jednom systému obsluhovat jak kód tak CI/CD pipeline.

2.3.6 GitOps

Za zmínku stojí také model GitOps, který z principů DevOps vychází a zavedla ho v roce 2017 společnost Weaveworks[4]. Metoda využívá nástrojů jako je Git, IaC a CI/CD. Git je distribuovaný systém pro správu verzí, který je k dispozici na všech běžných vývojových platformách pod svobodnou licencí. Poskytuje každému vývojáři kompletní soukromou offline kopii softwarového úložiště a nástroje pro správu verzí a synchronizaci své kopii úložiště s kopií na serveru [16].

Základní myšlenkou GitOps je mít k dispozici úložiště Git, které obsahuje deklarativní popis infrastruktury, která je aktuálně požadována v cílovém prostředí a automatizovaný proces, který zajistí, aby prostředí odpovídalo popsanému stavu v úložišti. Pokud je potřeba nasadit novou verzi aplikace, stačí pouze aktualizovat Git úložiště a o vše ostatní se postará automatizovaný proces [1].

[4] `https://www.weave.works/technologies/gitops`

Dle [32] jsou základními prvky GitOps kromě IaC, CI a CD také požadavky na sloučení. Požadavky na sloučení (MR - Merge Request) slouží jako mechanismus pro formální schvalování změn, než se provede sloučení do hlavní verze (větve). MR slouží také jako místo pro diskuzi, kontrolu kódu a záznam změn pro audit.

3 GitLab

GitLab, který byl v předchozí kapitole zmíněn jako jedna z mnoha CI/CD platforem, je webová platforma, která kromě CI/CD nabízí další nástroje pro podporu všech fází životního cyklu DevOps a která umožňuje společnostem maximalizovat celkovou návratnost vývoje softwaru tím, že software bude dodáván rychleji, efektivněji a s ohledem na bezpečnost a dodržování stanovených předpisů [21].

Protože tématem této práce je kontinuální integrace právě na platformě GitLab, bude v této kapitole krátce představena jeho historie, řešení a nástroje, které nabízí, dále pak jeho verze a způsob instalace, architektura a nakonec samotná kontinuální integrace.

3.1 Historie

GitLab začal vznikat v roce 2011, kdy chtěl ukrajinský programátor webových aplikací Dmitriy Zaporozhets přejít na Git pro správu verzí a Github pro lepší komunikaci v týmu, ale to mu nadřízenými v jeho firmě nebylo povoleno. Dimitri potřeboval nástroj, který by mu nebránil ve vývoji kódu, byl snadno použitelný a usnadnil spolupráci s ostatními členy v týmu. Aby tyto problémy vyřešil, rozhodl se vytvořit GitLab jako svůj vedlejší projekt v Ruby on Rails[1], který později rozvíjel spolu se svým kolegou Valerijem Sizovem vedle jejich běžné práce [5].

Po této iniciativě začal projekt GitLabu postupně růst [14]:

- **2011: Start GitLabu** - v roce 2011 začíná Dmitriy Zaporozhets společně se svým kolegou Valerijem Sizovem pracovat na GitLabu.
- **2012: GitLab.com** - v roce 2012 objevil GitLab Sytse Sijbrandij a přichází s myšlenkou, že nástroj pro spolupráci programátorů by mohl mít otevřený kód, takže by do něj mohl kdokoliv přispívat. Sytse představil GitLab na fóru *Hacker News*, kde získal zájem stovky lidí o přístup do testovací verze. Ještě v tentýž rok Dmitriy vytvořil první verzi GitLab CI.
- **2013: Práce na plný úvazek** - velké organizace používající GitLab žádaly Sytse o přidání dalších funkcí, které by ony samy potřebo-

[1]Ruby on Rails je framework pro vývoj webových aplikací v programovacím jazyce Ruby.

valy. Ve stejný okamžik se Dmitriy rozhoduje, že by chtěl pracovat na GitLabu na plný úvazek. Syste a Dmitriy se tak spojili a představili *GitLab EE* s funkcemi žádanými velkými organizacemi.

- **2014: Společnost GitLab** - v roce 2014 byl GitLab oficiálně zapsán jako společnost s ručením omezeným. GitLab vydává novou verzi každý měsíc, v prosinci 2014 byla vydána verze 7.6. Na konci roku GitLab podává přihlášku do Y Combinatoru - technologického akcelerátoru v Silicon Valley, kterého se účastní na začátku roku 2015.

- **2016: Další růst** - v roce 2016 již více než 1000 lidí přispělo k vývoji GitLabu, více než 100000 firem a miliony uživatelů používají GitLab. Tým GitLabu se rozrůstá na více než 140 lidí a GitLab získává další investice na další rozvoj.

- **2020: Největší společnost pracující na dálku** - s více než 1200 členy ve více než 65 zemích je GitLab celosvětově největší společností, kde její zaměstnanci pracují vzdáleně. GitLab nevlastní žádné kanceláře nikde na světě.

- **2021: 10 let GitLabu** - v roce 2021 GitLab oslavil 10 let od prvního commitu a má více než 30 milionů uživatelů a 1400 zaměstnanců.

3.2 Verze a instalace

Jádro aplikace GitLab je označováno CE (Community Edition) a je distribuováno zdarma pod licencí MIT[2]. Žádná funkce, která se kdy dostala do CE, nebude nikdy odstraněna nebo přesunuta do uzavřené verze GitLab EE (Enterprise Edition) [5].

GitLab EE je nabízen ve třech balíčcích, které se liší funkcemi a cenou [28]:

- **Free** - GitLab EE verze zdarma, rozdíl oproti GitLab CE je ten, že již není licencován jako svobodný software s otevřeným kódem a nemusí obsahovat jen komponenty pod MIT licencí. Dokumentace GitLabu doporučuje instalovat GitLab EE v balíčku Free, protože přechod na placené balíčky je jednodušší, než z GitLab CE.

[2]MIT - tolerantní licence svobodného softwaru vytvořená na Massachusetts Institute of Technology.

- **Premium** - placená verze, obsahuje oproti Free verzi podporu a funkce jako agilní plánování, rozšířené nástroje pro CI/CD nebo nástroje pro rychlejší kontrolu kódu.
- **Ultimate** - placená verze, oproti Premium obsahuje navíc nástroje pro zabezpečení, řízení rizik nebo správu portfolia projektů v rámci celé organizace.

GitLab je možné instalovat na vlastní server (self-managed) nebo ho využít ve verzi SaaS - největší instanci GitLabu na světě, kterou spravuje přímo tým GitLabu a stará se o údržbu, opravy a aktualizace. Výhodou self-managed možnosti je naopak větší kontrola nad instancí a funkce jako rozšířené administrační nástroje, logy, Git hooky, a další [31].

Instalovat self-managed GitLab je možné na většinu Linuxových distribucí. Hardwarové a softwarové požadavky a návod na instalaci jsou popsány v dokumentaci[3].

V této práci bude dále využíván GitLab SaaS.

3.3 Řešení a funkce

GitLab se postupně z nástroje pro správu verzí (na systému Git) a pro podporu spolupráce mezi členy týmu stal kompletní platformou s řešením pro všechny fáze SDLC metodikou DevOps. Dále budou popsány základní řešení a funkce GitLabu.

Správa verzí

Správa verzí na systému Git je základním prvkem GitLabu. Díky Gitu mohou vývojáři pracovat na své lokální kopii a svou práci následně synchronizovat v centrálním úložišti GitLabu. GitLab nabízí přehledné prostředí pro práci s Git větvemi (větvení umožňuje odloučit se od hlavní linie vývoje a pokračovat v práci, aniž by bylo do hlavní linie zasahováno), graf větvení, zabezpečení větví a MR - požadavky na sloučení větví.

V případě, že vývojář dokončí práci ve své větvi, která je připravena na sloučení zpět do hlavní vývojové větve, vytvoří MR. V GitLabu jsou v detailu MR vidět informace jako z které do které větve se bude slučovat nebo jednotlivé změny mezi větvemi. Toto je místo, kde probíhá tzv. code review, kdy ostatní vývojáři ručně prochází jednotlivé změny a přímo k částem kódu mohou psát komentáře a připomínky. K MR lze přiřadit uživatel, který ho musí schválit před jeho sloučením.

[3] `https://docs.gitlab.com/ee/install/`

GitLab nedovolí sloučit MR, dokud ho neschválí přiřazený kontrolující uživatel, nejsou vyřešeny všechny jeho komentáře, existují konflikty mezi dvěma větvemi nebo některé akce z CI/CD skončily chybou.

Agilní plánování

Přímo v prostředí GitLabu lze najít také nástroje pro agilní plánování [27]. Pomocí nástroje Issues lze sledovat a spravovat jednotlivé požadavky a jejich stav a k těmto úkolům vést diskuzi, asociovat je s MR a tvz. Milestony. Milestony jsou vyšší jednotkou při plánování, obsahují více Issues a mají stanovený počátek a konec. Pro vizualizaci lze využít nástěnku ve stylu Kanban, kde je přehledně vidět stav jednotlivých úkolů. Zároveň je možné přímo uživateli k jednotlivým úkolům vykazovat odpracovanou dobu.

Analytické nástroje

Analytický nástroj Value Stream Analytics, který je určen pro manažery, jim umožňuje zjistit, jak dlouho týmu trvá dokončení každé fáze vývoje softwaru od plánování až po monitorování. Díky přehledu o tom, jak dlouho trvá přejít z jedné fáze do druhé, je možné určit oblasti v procesu vývoje, které potřebují zlepšit a integrovat tak kontinuální zlepšování [25].

Registr balíčků

Nástroj Registr balíčků v GitLabu funguje jako soukromý nebo veřejný registr pro běžné správce balíčků (npm, composer, maven, ...). Díky tomuto nástroji je možné balíčky vytvořené v rámci CI publikovat, sdílet a snadno použít jako závislost v jiných projektech. Dále je k dispozici nástroj Registr Kontejnerů, který funguje jako zabezpečený a soukromý registr pro obrazy kontejnerů (Docker nebo Open Container Initiative) [29].

Bezpečnostní nástroje

GitLab také poskytuje nástroje pro zajištění bezpečnosti a detekci potenciálních bezpečnostních rizik. Nástroje dokáží mimo jiné [30]:

- skenovat kód a provádět nad ním statickou analýzu
- skenovat použité závislosti a v nich hledat potenciální rizika
- skenovat definici infrastruktury pro známé potenciální rizika
- detekovat klíče a hesla uložená přímo v kódu

- dynamicky testovat aplikaci
- testovat rozhraní API

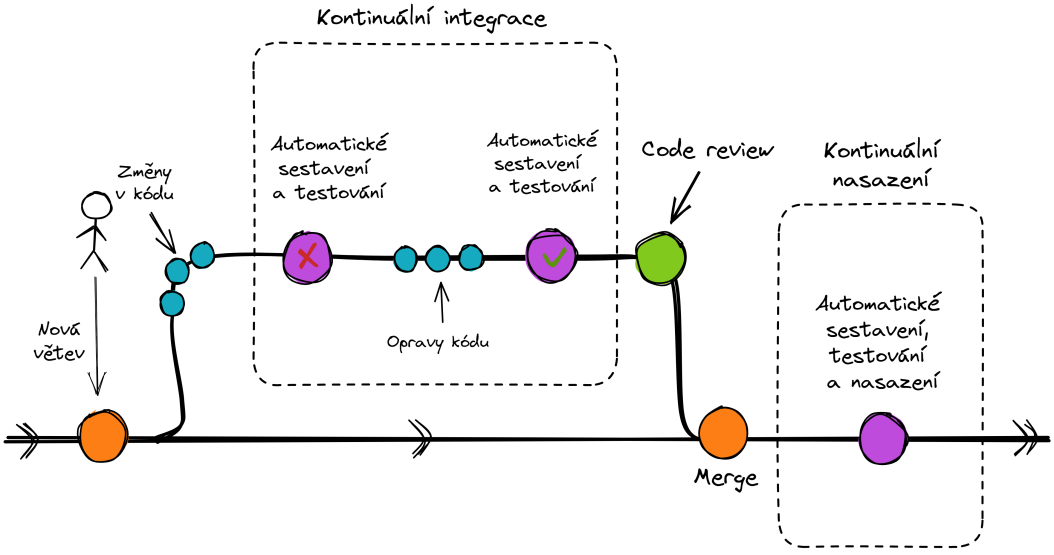
3.3.1 GitLab CI/CD

GitLab CI/CD je nástroj, který nabízí pokročilé funkce pro tvorbu, údržbu, nasazení a monitoring CI/CD pipeline pro automatizaci všech kroků potřebných k sestavení, testování a nasazení kódu do produkčního prostředí.

V prostředí GitLabu lze vidět jednotlivé pipeliny, jejich stav, logy a případné artefakty jako např. reporty výsledků automatických testů. Dále je možné nastavovat proměnné, ke kterým mají joby pipeline přístup a označovat je jako chráněné a poté k nim mají přístup jen uživatelé s dostatečnými právy a pipeline běžící nad chráněnými Git větvemi. Tyto chráněnné proměnné mohou být v logu z běhu jobu maskované. Toto bývá využíváno pro různé autentikační klíče při nasazování aplikace do produkčního prostředí.

K dispozici je také GitLab Auto DevOps - soubor předkonfigurovaných funkcí a integrací. Auto DevOps detekuje programovací jazyk a pomocí šablon CI/CD vytváří a spouští výchozí pipeline pro sestavení, testování a případně i nasazení aplikace.

Workflow



Obrázek 3.1: Git a CI/CD workflow

Když vývojář implementuje novou funkci, je postup, který je vidět na obrázku 3.1, následující:

1. Vytvoří si novou Git větev z produkční (či jiné) větve a v ní lokálně pracuje na úpravách kódu.
2. Commity odešle do repozitáře a tím (pokud není nastaveno jinak) spustí CI/CD pipeline.
3. GitLab CI/CD spustí automatické skripty pro sestavení a otestování aplikace.
4. Pokud některé skripty selžou, pipeline končí chybou a je potřeba chyby opravit a odeslat nové commity, které pipeline spustí znovu.
5. V případě úspěšného sestavení a testování je možné aplikaci nasadit do vývojového prostředí.
6. Následuje kontrola a schválení kódu (a případně i aplikace ve vývojovém prostředí) ostatními členy týmu.
7. Poté následuje merge (sloučení) změn zpět do produkční verze a tím je opět spuštěna CI/CD pipeline.
8. GitLab CI/CD spustí skripty pro sestavení, otestování a tentokrát i pro nasazení aplikace do produkčního prostředí.

GitLab Runner

GitLab Runner je aplikace, která spouští jednotlivé akce (joby) GitLab CI/CD pipeline. GitLab spravuje sdílené instance Runnerů pro GitLab SaaS a tyto instance jsou defaultně zapnuté pro všechny projekty. V nastavení projektu lze tyto instance vypnout a také registrovat instance provozované na vlastních serverech, v případě self-managed GitLabu je toto nutností. GitLab podporuje instalaci Runneru na operační systémy Linux, Windows a macOs. Při registraci Runneru je potřebovat zvolit tzv. executor, který určuje prostředí, ve kterém bude job spuštěn. GitLab podporuje tyto executory: SSH, Shell, VirtualBox, Parallels, Docker, Kubernetes nebo vlastní implementaci.

Sdílené Runnery v GitLab SaaS, které jsou automaticky spouštěny pro všechny joby, běží v Docker kontejneru na Linuxu. Job může volitelně definovat, na kterém runneru má být spuštěn.

Pipeline

Pipeline je komponenta GitLab CI/CD na nejvyšší úrovni. Každá pipeline obsahuje joby, které definují skripty co se mají vykonat a stage, které definují

kdy má který job být spuštěn (např. stage, která má joby pro testování se spustí až po stagy s joby pro sestavení aplikace). Joby jsou spouštěny pomocí GitLab Runneru a více jobů v jedné stage je vykonáváno paralelně.

Konfigurace CI/CD pipeline je definována v souboru *.gitlab-ci.yml*, který se nachází v kořenovém adresáři Git repozitáře. Tento soubor ve formátu YAML definuje jednotlivé joby a stage a další konfigurace pipeline. Soubor je možné upravovat lokálně spolu s dalšími běžnými soubory projektu a lze využít rozšíření pro kontrolu syntaxe pro některá integrovaná vývojové prostředí. Druhou možností je editace z webové aplikace GitLabu, kde je dostupný pipeline editor, který syntaxy také kontroluje.

Dále budou zmíněny pouze nejdůležitšjší klíčová slova, které lze v konfiguračním souboru použít, kompletní přehled lze najít v dokumentaci[4].

Stage

Jak již bylo zmíněno, stage říkají, kdy má být který job spuštěn a slučují tak jednotlivé joby do skupin či fází, které budou spouštěny sériově, ale joby v nich poběží paralelně. V souboru *.gitlab-ci.yml* se stage definují pomocí klíčového slova *stages* a jejich jmenným seznamem v pořadí tak jak se mají vykonat viz kód 3.1:

```
stages:
  - build
  - test
  - staging
  - production
```

Zdrojový kód 3.1: Definice stage

Pokud všechny joby v jedné stage skončí úspěšně, pipeline pokračuje ve vykonávání jobů z další stage. Pokud kterýkoliv job skončí s chybou, další stage není (obvykle) vykonána a pipeline končí předčasně.

Job

Joby, které se definují pomocí názvu daného jobu, říkají co se má vykonat pomocí klíčového slova *script* a v jaké fázi se to má vykonat pomocí klíčového slova *stage*.

[4] `https://docs.gitlab.com/ee/ci/yaml/index.html`

```yaml
job1:
  stage: build
  script: "execute-script-for-job1"

job2:
  stage: test
  script:
    - cd build
    - npm run test
```

Zdrojový kód 3.2: Definice jobu

Prostředí

Každý job může definovat, jakého se týká prostředí, pomocí klíčového slova *environment*, jména a případně i url adresy, kde bude aplikace nasazena. Toto se obvykle týká pouze jobů ve fázi nasazení a díky této specifikaci prostředí má job přístup k proměnným daného prostředí (viz dále) a v rozhraní GitLabu je potom přehled jednotlivých prostředí a jejich nasazení.

```yaml
deploy_staging:
  stage: deploy
  environment:
    name: staging
    url: https://staging.example.com
```

Zdrojový kód 3.3: Definice prostředí jobu

Proměnné

V souboru *.gitlab-ci.yml* lze pracovat se třemi typy proměnných:

- **Předdefinované proměnné** - které obsahují informace o jobu, pipeline a další jako např.:
  - **$CI_JOB_STAGE** - název stage
  - **$CI_COMMIT_TITLE** - první řádek z popisu commitu
  - **$CI_COMMIT_TITLE** - název větve

Další proměnné lze najít v dokumentaci[5].

- **Proměnné definované v rozhraní GitLabu** - chráněné proměnné jako jsou tokeny, hesla nebo klíče by měly být uloženy v nastavení GitLabu. Jednu proměnnou lze definovat pro každé prostředí zvlášť a díky tomu může být např. pro nasazení použit jiný klíč pro každé prostředí, ale definice jobu zůstává stejná. Tyto proměnné lze také označit jako chráněné a budou tak dostupné pouze pro pipeliny běžící pro chráněné větve (ty se také nastavují v prostředí GitLabu) a také jako maskované a ty pak nebudou vidět ve výpisu logu jobu.
- **Proměnné definované přímo v souboru** - a to buď globální nebo lokální uvnitř jobů:

```yaml
variables:
  GLOBAL_VAR: "A global variable"

job1:
  variables:
    JOB_VAR: "A job variable"
  script:
    - echo "Variables are '$GLOBAL_VAR' and '$JOB_VAR'"
```

Zdrojový kód 3.4: Definice a použití proměnných

Trigger

Jak již bylo zmíněno, trigger je nějaká akce, které vytvoří a spustí pipeline. Defaultní trigger, který se pokusí vytvořit pipeline je při každém push požadavku do repozitáře. Další možností je vytvoření pipeline pomocí HTTP požadavku na GitLab API nebo pomocí plánovaných pipeline v určitých intervalech. Pipeline je ale vytvořena (a spuštěna) pouze pokud obsahuje nějaký job. A každý job může definovat, kdy má být do pipeliny přidán a kdy ne pomocí klíčových slov:

- **rules** - umožňuje definovat komplexní pravidla kdy job přidat či vyřadit z pipeline, zda může job skončit chybou a vykonávání pipeline pokračuje a další. Příklad jobu v kódu 3.5 je přidán do pipeline pokud se jedná o push do MR nebo pokud jde o plánovanou pipeline.

[5] `https://docs.gitlab.com/ee/ci/variables/predefined_variables.html`

```yaml
job:
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
      allow_failure: true
    - if: $CI_PIPELINE_SOURCE == "schedule"
```

Zdrojový kód 3.5: Použití klíčového slova rules

- **only** - definuje, kdy má být job součástí pipeline, např. název větve, pouze MR a další.
- **except** - opak klíčového slova only, definuje tedy kdy job nemá být součástí pipeline.

Artefakty

Každý job může mít výstup v podobě archivu souborů a adresářů, který je označován jako artefakt. Tyto artefakty jsou poté dostupné z prostředí GitLabu a z GitLab API. Job definuje své artefakty pomocí klíčového slova *artifacts*, cestami k souborům a případně platností souborů (jinak je použito defaultní nastavení instance). Příklad v kódu 3.6 níže vygeneruje PDF soubor a uloží ho jako artefakt po dobu jednoho týdne.

```yaml
pdf:
  script: latex mycv.tex
  artifacts:
    paths:
      - mycv.pdf
    expire_in: 1 week
```

Zdrojový kód 3.6: Definice artefaktů

Cache

Cache jsou soubory, které mohou joby stahovat a ukládat pro další běh, místo toho aby je vždy znovu vytvářely a vykonávání je tak rychlejší a efektivnější. Joby definují cache soubory pomocí klíčového slova *cache* a cestou k souborům s unikátním klíčem, který cache jednoznačně identifikuje. Joby,

které používají stejný klíč, mají přístup ke stejné cache včetně jiných pipeline.

```
cache-job:
  cache:
    key: binaries-cache-$CI_COMMIT_REF_SLUG
    paths:
      - binaries/
```

Zdrojový kód 3.7: Využití cache

Paralelní vykonávání

```
linux:build:
  stage: build

mac:build:
  stage: build

lint:
  stage: test
  needs: []

linux:rspec:
  stage: test
  needs: ["linux:build"]

mac:rspec:
  stage: test
  needs: ["mac:build"]
```

Zdrojový kód 3.8: Ukázka paralelního vykonávání jobů

V běžném případě se joby v jedné stage vykonávají paralelně, ale joby z následující stage se začnou vykonávat až pokud doběhnou všechny joby ze stage předchozí. Pokud ovšem nějaký job z druhé stage potřebuje pro svůj běh pouze výstupy z některých jobů z první stage, může tyto závislosti definovat pomocí klíčového slova *needs*. Job je tak spuštěn ihned poté, co

doběhnou joby na kterých je závislý a nečeká tak na kompletní doběhnutí předchozí stage. Job dokonce může definovat *needs* jako prázdné pole a je spuštěn ihned po vytvoření pipeline, i když není součástí první stage. V kódu 3.8 je příklad definice závislostí mezi joby. V tomto případě je job *lint* spuštěn (spolu s joby *linux:build* a *mac:build*) ihned po vytvoření pipeline, protože definuje prázdné pole závislostí. Job *linux:rspec* je spuštěn ihned poté, co doběhne job *linux:build* protože je závislý pouze na něm. Analogicky je job *mac:rspec* závislý pouze na jobu *mac:build*.

Docker image

Při použití Docker executoru, definuje klíčové slovo *image* (viz kód 3.9), jaký Docker image má být pro běh jobu použit. Defaultní image má každý executor nastavený samostatně, proto je dobré toto vždy explicitně definovat a to buď defaultně pro všechny joby v pipeline, nebo u každého jobu zvlášť.

```
default:
  image: ruby:3.0

rspec:
  script: bundle exec rspec

rspec 2.7:
  image: registry.example.com/my-group/my-project/ruby:2.7
  script: bundle exec rspec
```

Zdrojový kód 3.9: Ukázka použití Docker image

Služby

Pomocí klíčového slova *services* lze definovat jakékoliv další Docker image, které jsou potřebné pro běh jobu. Tyto image jsou použity k vytvoření dalších kontejnerů a mohou s kontejnerem ve kterém běží job komunikovat. Službou může být jakákoliv aplikace, ale běžné použití bývá např. pro běh databáze viz příklad v kódu 3.10.

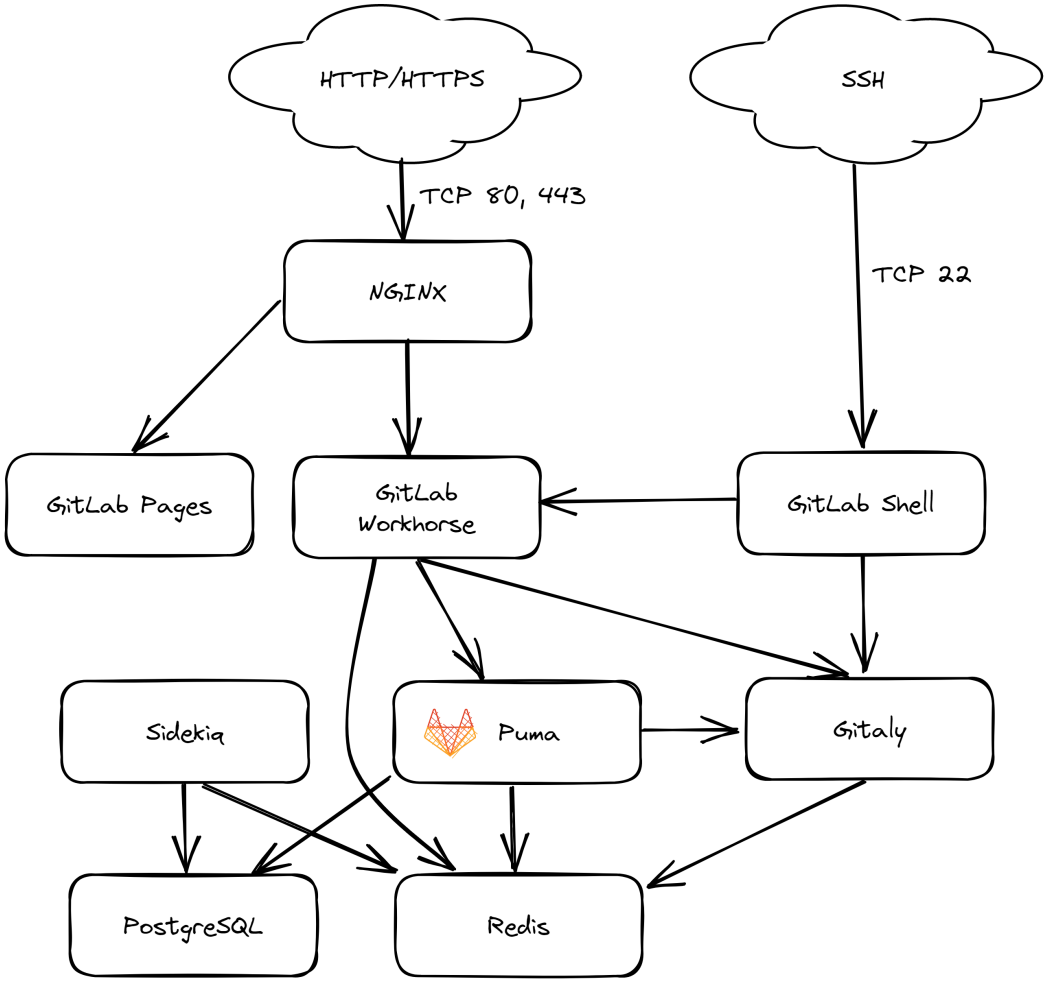
```
job1:
  services:
    - postgres:12.2-alpine
  variables:
  POSTGRES_DB: $POSTGRES_DB
  POSTGRES_USER: $POSTGRES_USER
  POSTGRES_PASSWORD: $POSTGRES_PASSWORD
  POSTGRES_HOST_AUTH_METHOD: trust
```

Zdrojový kód 3.10: Ukázka použití Docker služeb

3.4 Architektura

GitLab není monolitická aplikace, ale snaží se řídit unixovou filozofií, což znamená, že je rozdělen do jednotlivých modulů, které by měly dělat pouze jednu konkrétní věc a dělat ji dobře [5]. Zjednodušený diagram na obrázku 3.2, ukazuje závislosti hlavních komponent GitLabu, kterými jsou [26]:

- **NGINX** - je webový server, který směřuje všechny HTTP požadavky do příslušných subsystémů Gitlabu. Při přístupu k Git úložišti přes HTTP je použito rozhraní GitLab API k řešení autorizace, přístupu a obsluze objektů Git repozitáře.
- **GitLab Pages** - je modul, který umožňuje publikovat statické webové stránky přímo z úložiště v GitLabu. Toho lze využít jak pro osobní nebo firemní webové stránky, portfolia a prezentace tak pro generované dokumentace, reporty či výsledky testů.
- **GitLab Shell** - je program, který umožňuje přístup k Git repozitářům přes protokol SSH a využívá GitLab API pro ověření autorizovaných SSH klíčů. GitLab Shell přistupuje k vlastním Git repozitářům přes Gitaly a využívá úložiště Redis pro vkládání úloh pro Sidekiq.
- **GitLab Workhorse** - je program, který byl v GitLabu navržen tak, aby pomohl snížit zátěž z aplikačního serveru Puma. Jedná se o reverzní proxy server, který směruje HTTP požadavky, dokáže zpracovat stahování a nahrávání souborů a Git push/pull příkazy.
- **Puma** - je aplikační server v jazyce Ruby postavený na webovém frameworku Ruby on Rails, který poskytuje uživatelům vlastní webovou aplikaci GitLab a také GitLab (REST) API.



Obrázek 3.2: Zjednodušený pohled na architekturu GitLabu

- **Gitaly** - je služba navržená společností GitLab, která odstraňuje potřebu NFS pro ukládání souborů v Gitu. Gitaly zajišťuje veškerý přístup k systému Git v celém systému GitLabu a spouští tak Git operace z GitLab Shell, webové aplikace GitLab a GitLab API. Gitaly poskytuje API pro získání atributů z Gitu (větve, tagy a další metadata) a objektů (commity, soubory, rozdíly mezi verzemi a další).
- **Sidekiq** - je procesor úloh na pozadí, který stahuje úlohy z fronty úložiště Redis a zpracovává je. Přesunutí práce do úlohy na pozadí umožňuje GitLabu zajistit rychlejší cyklus mezi požadavkem a odpovědí.
- **Redis** - je neperzistentní úložiště typu klíč-hodnota v paměti, který GitLab využívá mimo jiné pro ukládání fronty úloh na pozadí (ke zpracování procesorem Sidekiq), dočasné mezipaměti, dat relace nebo tra-

sovacích bloků CI.

- **PostgreSQL** - je perzistentní SQL databáze, do které GitLab ukládá informace o projektech, uživatelích a oprávněních, metadata a další.
- **GitLab Runner** - je aplikace, která spouští jednotlivé akce (joby) GitLab CI/CD pipeliny. GitLab spravuje sdílené instance Runnerů pro GitLab SaaS (s omezenou měsíční minutáží dle vybrané verze - Free, Premium, Ultimate) a tyto instance jsou defaultně zapnuté pro všechny projekty. V nastavení projektu lze tyto instance vypnout. Kromě sdílených instancí lze u GitLabu SaaS registrovat instance provozované na vlastních serverech, v případě self-managed GitLabu je toto nutností.
- **Glab** - GLab či GitLab CLI je nástroj, který umožňuje přístup k GitLabu z příkazové řádky místo webového prohlížeče. Poskytuje např. příkazy pro správu úkolů, správu MR, práci s CI/CD pipeline a další.

Podrobný popis architektury GitLabu se všemi jeho komponentami lze nají v dokumentaci[6].

[6] `https://docs.gitlab.com/ee/development/architecture.html`

4 JavaScript

V současné době má celosvětová WWW síť více než miliardu webových stránek, ke kterým přistupují miliardy připojených zařízení [18], na kterých běží webový prohlížeč nebo podobný program, který je schopen webové stránky zpracovávat a zobrazovat. Většina webových stránek vkládá nebo načítá zdrojový kód napsaný v programovacím jazyce JavaScript [18]. JavaScript používalo podle průzkumu [34] v roce 2022 67.9 % profesionálních programátorů, což z něj činí nejrozšířenější programovací jazyk na světě.

V této kapitole budou popsány historické důvody pro vytvoření jazyka JavaScript, který, jak plyne ze zadání, bude dále používán v této práci, jeho využití, možná rozšíření a užitečné nástroje především pro automatické testování využitelné při kontinuální integraci.

JavaScript je vysokoúrovňový, objektově orientovaný programovací jazyk se slabou dynamickou typovou kontrolou, který byl představen v roce 1995 společností Netscape Communications Corporation, která v té době stála v čele oblasti vývoje webových prohlížečů se svým prohlížečem Netscape Navigator [18], jako jednoduchý jazyk, který byl zahrnut do definice webových stránek a interpretován prohlížečem, což umožnilo stránce dynamicky měnit svůj obsah a reagovat na interakci uživatele. Od té doby tento jazyk převzaly téměř všechny ostatní webové prohlížeče [7].

Po přijetí jazyka JavaScript mimo společnost Netscape byl v roce 1997 sepsán standardní dokument *ECMA-262* [24], který popisoval fungování jazyka, který měli dodržovat různé produkty podporující jazyk JavaScript. Tento standard se nazývá *ECMAScript*, podle mezinárodní organizace Ecma, která standardizaci prováděla a v praxi lze termíny ECMAScript a JavaScript zaměňovat jako dva názvy pro jeden a ten samý jazyk [7]. Existuje několik verzí standardu ECMAScript, od roku 2015 vychází nová verze pravidelně každý rok a aktuální nejnovější verze je ECMAScript 2022 [24].

Webové prohlížeče nejsou jedinou platformou, kde se JavaScript používá. Podporují ho některé databáze (např. MongoDB a CouchDB) jako skriptovací a dotazovací jazyk a existují platformy pro programování desktopových a serverových aplikací [7]. Za zmínku stojí především projekt *Node.js*, serverové prostředí JavaScriptu založené na prostředí *V8 engine* od společnosti Google a asynchronním modelu I/O událostí.

Navzdory svému úspěchu není JavaScript příliš vhodný jazyk pro vývoj a údržbu rozsáhlých aplikací, zejména kvůli slabé dynamické typové kontrole [2]. Tento nedostatek má odstranit jazyk *TypeScript* od firmy Microsoft,

který je rozšířením jazyka JavaScript o moduly, třídy, rozhraní a především statický typový systém. TypeScript je kompilován do JavaScriptu, ale typová kontrola, která probíhá v době kompilace, je pouze volitelná a stále tak lze používat dynamické typování jako v "čistém" JavaScriptu [2].

4.1 Frameworky

Aby se vývojáři mohli soustředit na vlastní vývoj funkcí aplikace a nemuseli znovu a znovu řešit ty samé věci, jako je např. připojení k databázi, mapování funkcí na URL adresy nebo autorizace uživatelů, existují tzv. frameworky. Frameworky jsou kolekce užitečných předpřipravených nástrojů, které programátorům usnadňují vývoj aplikací [3].

Mezi populární frontendové JavaScript frameworky pro programování webových aplikací patří např. Angular, Vue nebo React. Pro vývoj aplikací na serveru (backend) v Node.js se potom používá Express.js, Koa.js nebo Nest.js.

4.2 Nástroje pro CI

V této části bude analýza a výběr užitečných nástrojů nejen pro sestavení a testování JavaScript aplikací pro použití v GitLab CI/CD.

4.2.1 Sestavení

Při vývoji rozsáhlých aplikací jsou typicky části kódu rozděleny do mnoha JavaScript souborů (modulů) dle funcionalit a tyto moduly mezi sebou vytvářejí závislosti. Pro spuštění JavaScript kódu v prohlížeči, se do HTML vkládají tzv. script tagy, které odkazují na JavaScript soubor. V případě více souborů musí být tyto tagy ve správném pořadí kvůli jejich závislostem, globální proměnné a názvy funkcí definované v těchto souborech se překrývají a vznikají další možné problémy [10].

K řešení tohoto problému se používá tzv. bundler, nástroj pro sestavení, který zkompiluje několik JavaScript souborů do jednoho spustitelného. Bundler dokáže automaticky detekovat závislosti na použitých knihovnách nebo rozdělit finální kód do více souborů a automaticky je načítat pro zvýšení výkonnosti [19].

Mezi populární sestavovací nástroje patří např. Browserify, Rollup, Parcel, Gulp, Brunch a nebo Webpack, který je z nich nejpoužívanějším [35] a který, jak je uvedeno v zadání, bude dále v této práci používán.

4.2.2 Automatické testování

Důležitost testování softwaru jako fáze SDLC a součást kontinuální integrace a jednotlivé druhy testů již byly zmíněny v kapitole 2.1.1. Nástroje pro jednotkové, integrační a E2E testování lze rozdělit podle funkcí, které pro testování nabízí, přičemž některé z nich nabízí pouze jednu konkrétní funkci jiné kombinaci více funkcí. Je běžné používat kombinaci více nástrojů pro zajištění všech potřebných funkcí pro testování. Dále následuje seznam těchto funkcí společně s nástroji, které tyto funkce poskytují [36].

- **Spouštěče testů** - které spustí testy v prohlížeči nebo v Node.js dle konfigurace uživatele (Karma, Mocha, Jasmine, Jest, Cypress).
- **Testovací struktura** - umožňuje definici a organizaci testů pomocí klíčových slov (Mocha, Jasmine, Jest, Cypress).
- **Assertion funkce** - slouží ke kontrole, zda test vrací očekávanou hodnotu a v opačném případě vyhodí čitelnou výjimku (Chai, Jasmine, Jest, Cypress, Unexpected).
- **Generování výsledků testů** - po doběhnutí testu jsou vygenerovány reporty s výsledky (Mocha, Jasmine, Jest, Cypress, Karma).
- **Mock objekty** - jsou objekty, které napodobují chování reálných pro potřeby testování, aniž by musel běžet celý systém (jako v případě funkčních testů) (Sinon, Jasmine, enzyme, Jest).
- **Pokrytí kódu testy** - říká jaké procento kódu bylo otestováno (Istanbul, Jest, Deka).
- **Ovládání prohlížeče** - pro potřeby E2E testování a simulování akcí uživatele (Nightwatch, Phantom, Puppeteer, Cypress).

Framework Mocha, který je zmíněn v zadání a má být použit pro jednotkové a integrační testy je pouze testovací struktura a spouštěč testů, a musí být použit s dalšími nástroji pro assertion funkce nebo mock objekty. Mocha byl nejpoužívanější testovací JavaScript framework do roku 2019, ale od té doby jeho obliba klesá [35]. V současné době je nejpoužívanějším framework Jest, který obsahuje všechny potřebné funkce pro jednotkové a integrační testy v jednom nástroji a proto bude v této práci použit místo frameworku Mocha.

Nejpoužívanější frameworky pro E2E testování jsou Selenium, Nightwatch, Puppeteer, Playwright, TestCafe a Cypress [35, 36]. Selenium nebylo vytvořeno přímo k testování, ale jedná se o driver, přes který lze ovládat prohlížeč.

Instalace a ladění je u Selenia náročnější. Nightwatch je postaven na Seleniu a usnadňuje tak instalaci, přidává vlastní assertion a jiné funkce, nepodporuje ale Typescript. Puppeteer a Playwright fungují na principu ovládání prohlížeče přes rozhraní API pro vývojáře (DevTools protokol). Puppeteer neobsahuje funkce pro spouštění testů a testovací strukturu a je nutné ho použít spolu s dalšími nástroji. Playwright je relativně nový (2020). TestCafe a Cypress jsou podobné frameworky, které vkládají kód přímo do webu a tím ho ovládají. V této práci bude pro E2E testy použit framework Cypress protože je ze všech zmíněných nejpoužívanější a zavedený, podporuje Typescript, využívá nástroje Mocha pro testovací strukturu a oproti TestCafe nabízí více funkcí a zdarma.

Oba vybrané testovací nástroje Jest i Cypress mají funkce pro generování výstupů testů a pro analýzu pokrytí testy.

Statická analýza

Protože má JavaScript slabou dynamickou typovou kontrolu, jsou nástroje pro statickou analýzu ještě důležitější než u jazyků staticky typovaných [9]. Toto už z části není pravda u TypeScriptu, který dokáže při kompilaci detekovat možné chyby a stává se tak součástí statické analýzy. Mezi další nástroje patří ESLint, který je nejpoužívanějším nástrojem pro statickou analýzu [35] a je také zmíněn v zadání práce. ESLint dokáže detekovat možné chyby a potenciální problémy či nekonzistentní formátování. ESLint používá předdefinované pravidla, která lze konfigurovat a upravovat dle potřeby. Spolu s ESLintem bude použit také nástroj Prettier, který slouží pro zachování konzistence formátování a stylu kódu. Prettier navíc dokáže dle definovaných pravidel nevyhovující kód formátovat automaticky.

4.2.3 Další nástroje

Ze zdrojového kódu, který je dostatečně komentován tzv. dokumentačními komentáři, lze k tomu určenými nástroji vygenerovat dokumentaci (typicky ve formátu HTML pro prohlížení na webu). Dokumentační komentáře, specifické pro každý programovací jazyk, popisují jednotlivé komponenty kódu jako jsou třídy nebo metody a jejich chování, parametry nebo návratové hodnoty. CI/CD pipeline je ideální místo pro generování dokumentace, která je rovnou nasazena na webový server. Existuje několik knihoven pro generování dokumentace z JavaScriptu, z nich nejpopulárnější je JSDoc nebo TypeDoc, který má lepší podporu pro typescript a proto bude v této práci použit. Dále bude také použit nástroj Compodoc, který je speciálně vytvořen pro několik JavaScript frameworků a nabízí díky tomu funkce jako např. detekování

komponent specifických pro daný framework. Jeden z podporovaných frameworků je již zmíněný Node.js framework Nest.js, který bude také použit.

Dalším užitečným nástrojem pro použití v CI/CD GitLabu je Danger JS, který dokáže automatizovat některé kroky, které se provádějí při code review. Danger poskytuje přístup k MR a metadatům jako např. ke změněným souborům a nad nimi lze provádět analýzu a výsledky zveřejňovat jako zprávy u MR přímo v prostředí GitLabu.

5 Ukázková aplikace

Jak je uvedeno v zadání, součástí této práce je návrh a implementace aplikace pro evidenci pracovní doby zaměstnanců na projektech s využitím technologií a nástrojů, které poté budou použity v CI/CD pipeline v GitLabu. V této kapitole budou vzneseny požadavky na aplikaci a použité technologie, bude navržena její architektura a popsána její implementace.

Ukázková aplikace bude webová aplikace, ke které budou přes webový prohlížeč přistupovat jak zaměstnanci za účelem měření své pracovní doby, tak jejich zaměstnavatelé, kteří budou mít přístup k výkazům odpracované doby svých jednotlivých zaměstnanců. Jedna skupina zaměstnanců a jejich zaměstnavatele (např. firma) je jedním tzv. pracovním prostředím. Všechna pracovní prostředí (a jejich data) jsou od sebe oddělena, ale jeden uživatel aplikace (s unikátním přihlašovacím emailem) může být součástí více pracovních prostředí a v každém z nich mít jinou roli. Může tak být zaměstnancem více zaměstnavatelů a zároveň může být zaměstnavatelem několika skupin zaměstnanců. Uživatel používá pouze jedny přihlašovací údaje a mezi svými pracovními prostředími se může jednoduše přepínat.

V aplikaci existují celkem čtyři možné role uživatelů, které jsou členěny v hierarchii a každá další role má všechna práva jako role před ní:

- **Uživatel** - roli uživatel mají zaměstnanci, kteří pouze měří svůj odpracovaný čas na projektu a případně vidí svojí historii a výkazy.
- **Manažer** - uživatelé s rolí manažer vidí kromě svých také výkazy ostatních uživatelů v daném pracovním prostředí a mohou jim přiřazovat projekty.
- **Admin** - uživatel s rolí admin může spravovat uživatele daného pracovního prostředí, přidávat do něj uživatelé nové a nastavovat jim role.
- **Vlastník** - roli vlastníka získává ten, kdo dané pracovní prostředí vytvořil a může ho spravovat.

5.1 Funkční požadavky

Hlavní požadavky na funkce aplikace byly zaneseny do diagramu případů užití (UC), viz obrázek B.2 v příloze, spolu s nejnižší požadovanou rolí uživatele, se kterou je k těmto funkcím přístup. Níže následuje popis těchto požadavků.

- **UC1 Vytvořit profil (nepřihlášený)** - uživatel s dosud neregistrovaným emailem si může sám vytvořit účet a současně s ním je mu vytvořeno jeho defaultní pracovní prostředí. Toto defaultní prostředí nelze smazat, uživatel v něm má roli vlastníka a může do něj pozvat jiné uživatele (*UC17*).

- **UC2 Vytvořit pracovní prostředí** - kromě svého defaultního prostředí, které je každému vytvořeno automaticky při vytváření jeho uživatelského profilu, může kdokoliv vytvořit další pracovní prostředí (není označeno jako defaultní a lze smazat) a stává se jeho vlastníkem.

- **UC3 Změřit časový záznam** - hlavní funkce celé aplikace je měření odpracovaného času. Uživatel zvolí projekt na kterém chce pracovat, popíše činnost či úkol na kterém pracuje a zmáčkne tlačítko start. V průběhu měření musí uživatel vidět jak dlouho pracuje, nesmí mít možnost spustit více měření současně a to ani z jiného zařízení a pokud v průběhu měření z aplikace odejde měření musí pokračovat. Uživatel musí mít možnost zadat odpracovanou dobu i ručně - **UC3.1 Zadat záznam ručně**.

- **UC4 Zobrazit své reporty** - uživatel si může zobrazit historii svých záznamů odpracovaného času, filtrovat je dle časového období a projektu a může je shlukovat do dnů, týdnů nebo měsíců. Potom vidí za každý časový úsek pouze jeden záznam s hodnotami agregovanými z příslušných záznamů. Zároveň vidí celkový čas a odměnu za všechny záznamy vyhovující nastaveným filtrům.

- **UC5 Upravit časové záznamy** - uživatel může ručně upravovat záznamy odpracovaného času, kdy může měnit popisek, přiřazený projekt a začátek a konec měření, přičemž odpracovaný čas se z nich ihned přepočítá.

- **UC6 Zobrazit své statistiky** - kromě reportů, které nabízí podrobnou historii odpracovaného času, by měl mít uživatel možnost rychle vidět své statistiky jako sumu odpracovaného času za aktuální den, týden a měsíc.

- **UC7 Vytvořit požadavek na platbu** - uživatel může z vyfiltrovaných záznamů vytvořit požadavek na platbu. Každý záznam může být součástí pouze jednoho požadavku a cena je vypočtena dle hodinové sazby, kterou má uživatel přiřazenou ke každému projektu individuálně.

- **UC8 Zobrazit své platby** - uživatel si může zobrazit své požadavky na platby, filtrovat si je dle datumu a dle stavu (zaplaceno/nezaplaceno). Požadavky, které ještě nejsou označené jako zaplacené, lze odstranit.
- **UC9 Upravit svůj profil** - v nastavení svého profilu si může uživatel změnit jméno a heslo.
- **UC10 Změnit pracovní prostředí** - přihlášený uživatel vidí v jakém se aktuálně nachází pracovním prostředí a seznam pracovních prostředí do kterých má přístup, mezi nimiž se může jednoduše přepínat bez nutnosti opakovaného přihlašování. V jeden čas může být na více zařízení přihlášen ve více prostředích.
- **UC11 Zobrazit reporty uživatele (manažer)** - manažer si může zobrazit reporty kteréhokoliv uživatele z daného pracovního prostředí a filtrovat je úplně stejně jako své vlastní reporty (*UC4*).
- **UC12 Spravovat projekty (manažer)** - manažer může přidávat, upravovat a mazat projekty, kromě defaultního projektu, který je vytvořen každému pracovnímu prostředí automaticky při jeho vytváření.
- **UC13 Spravovat projekty uživatele (manažer)** - manažer přiřazuje jednotlivým uživatelům projekty a ti mají poté možnost tyto své přiřazené projekty vybrat při měření. Zároveň manažer přiřazuje uživateli a každému jeho projektu individuálně hodinovou sazbu.
- **UC14 Zobrazit platby uživatele (manažer)** - manažer si může zobrazit požadavky na platby kteréhokoliv uživatele z daného pracovního prostředí.
- **UC15 Schválit platby uživatele (manažer)** - manažer může označovat požadavky na platby jako zaplacené či nezaplacené.
- **UC16 Upravit profil uživatele (admin)** - admin může upravit profil kteréhokoliv uživatele z daného pracovního prostředí. Kromě emailu, jména a heslo nastavuje admin také uživatelské role či může uživatele z daného pracovního prostředí odebrat.
- **UC17 Přidat uživatele do pracovního prostředí (admin)** - admin může přidávat další uživatele do pracovního prostředí a to tak, že pokud již daný uživatel má na platformě profil (ale je součástí jiných

pracovních prostředí), zadá pouze jeho email (*UC17.1*). Pokud uživatel s daným emailem ještě profil nemá, vytvoří mu zcela nový profil (*UC17.2*)

- **UC18 Upravit pracovní prostředí (vlastník)** - pouze vlastník daného pracovního prostředí jej může upravovat, např. změnit název prostředí.

5.2 Nefunkční (mimofunkční) požadavky

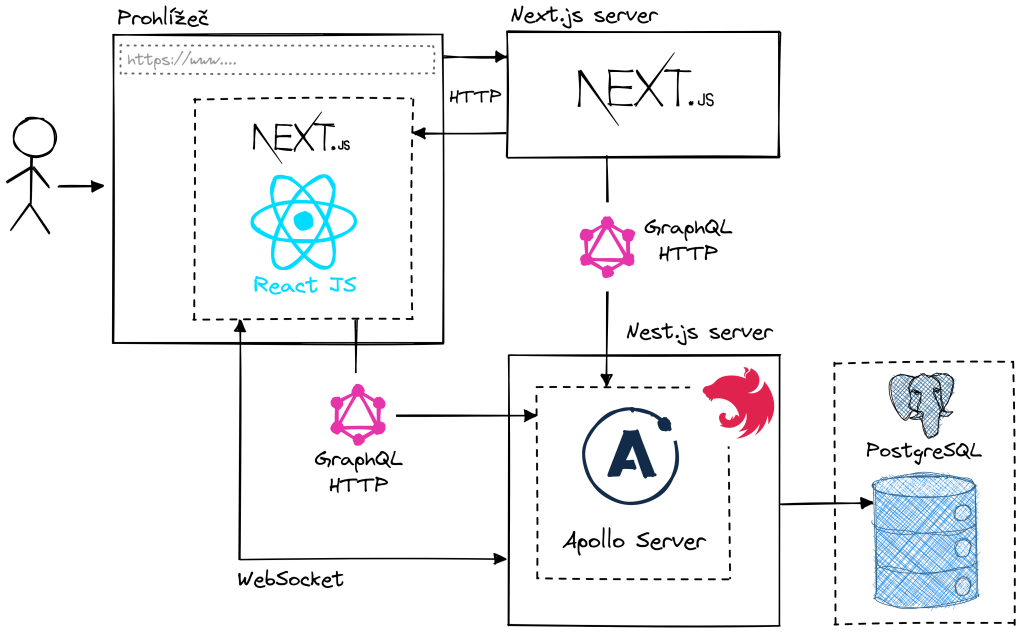
Před vlastním návrhem aplikace, její architektury a výběrem technologií byly kromě funkčních požadavků výše vzneseny i tyto další požadavky na aplikaci:

- **Dostupnost** - aplikace je dostupná odkudkoliv přes webový prohlížeč.
- **Použitelnost** - aplikace by měla být vytvořena tak, aby její uživatelské rozhraní bylo uživatelsky přívětivé a byla v něm snadná orientace.
- **Automatická synchronizace** - aplikace by měla být schopna automatické synchronizace a uživatel připojený ke svému účtu z více zařízení vidí provedené změny v reálném čase na ostatních zařízeních.
- **Bezpečnost** - aplikace by měla bezpečně zacházet s daty a neumožnit přístup k nim neautorizovaným uživatelům.
- **Testovatelnost** - kód aplikace by měl být psán tak, aby byl testovatelný vybranými nástroji i pro potřeby kontinuální integrace.
- **Srozumitelnost** - kód aplikace by měl být dostatečně komentován tak, aby byl snadno pochopitelný pro použití jinými vývojáři a aby z něj šla vygenerovat dokumentace vybranými nástroji.

5.3 Architektura a technologie

Výběr některých nástrojů spolu s odůvodněním byl již zmíněn v kapitole 4. Zde bude tento výběr upřesněn v rámci jednotlivých komponent architektury, jejíž schéma je vidět na obrázku 5.1.

V architektuře typu klient-server přistupuje uživatel ke klientské aplikaci přes webový prohlížeč. Na klientské části (frontend) je použit framework Next.js, který je postavený na knihovně React. React je knihovna pro tvorbu



Obrázek 5.1: Architektura a technologie aplikace

uživatelského rozhraní z částí zvaných komponent, které následným skládáním tvoří celé obrazovky, stránky a aplikace. Next.js poskytuje k Reactu další užitečné nástroje např. pro směrování, různé optimalizace či vykreslování na serveru. V případě aplikace pouze v Reactu vrací server prázdnou HTML stránku, jejíž obsah je vytvořen dynamicky v JavaScriptu až v prohlížeči. Jak je vidět ze schématu architektury (obrázek 5.1) komponenta Next.js server, znázorňuje serverovou část aplikace ve frameworku Next.js, která dokáže některé stránky sestavit přímo na serveru (SSR).

Klientská aplikace (a to jak v prohlížeči tak Next.js server), komunikuje s aplikačním serverem (backend) který je postaven na Node.js frameworku Nest.js. Komunikace mezi klientem a serverem probíhá pomocí API rozhraní a dotazovacího jazyka GraphQL. Na serveru je použita knihovna Apollo Server, která je součástí frameworku Nest.js a pomocí které je vystavěno rozhraní API. Na klientovi je použita knihovna Apollo Client, která toto API využívá.

Kromě komunikace přes GraphQL API, které funguje na protokolu HTTP a po odpovědi serveru na požadavek klienta je spojení ukončeno, je potřeba také zajistit požadavek automatické synchronizace (viz nefunkční požadavky v kapitole 5.2). Toho je docíleno pomocí protokolu WebSocket, který vytvoří dlouhodobé plně duplexní spojení mezi klientem a serverem a díky tomu může server posílat požadavky na klienta a notifikovat ho např. o změnách v jeho účtu, které byly provedeny z jiného zařízení.

Z dalších použitých nástrojů na frontendu stojí za zmínku např. CSS framework Tailwind CSS pro stylování komponent, knihovna MUI, která nabízí řadu předdefinovaných komponent pro React nebo knihovna Socket.IO pro komunikaci přes WebSocket (je použita i na backendu). Pro statickou analýzu je na frontendu použit zmíněný ESLint a Prettier a pro jednotkové testy, testování komponent a E2E testy je použit framework Cypress. Na backendu je ESLint a Prettier použit také a pro jednotkové, integrační testy a E2E testy API je použit framework Jest. Jak frontend tak backend je naprogramován v TypeScriptu s použitím nástroje Webpack pro sestavení.

5.4 Adresářová struktura

Frontend a backend jsou samostatné oddělené aplikace a každá má svůj vlastní soubor *package.json*, který definuje své závislosti, skripty a metadata pro využití balíčkovacími nástroji *npm* či *yarn*, který byl při tvorbě použit, protože oproti npm dokáže balíčky instalovat paralelně a tedy rychleji. Obě aplikace se ale nacházejí ve stejném Git repozitáři a díky tomu stačí jedna CI/CD pipeline, která poběží při změně v kterékoliv z nich. Dále bude popsána adresářová struktura projektu, kde budou vypíchnuty pouze nejdůležitější složky a soubory, které jsou důležité pro pochopení architektury aplikace nebo CI/CD pipeline.
Kořenový adresář projektu obsahuje tyto složky a soubory:

- **backend** - projekt aplikace backendu
- **frontend** - projekt aplikace frontendu
- **public** - statické soubory pro nasazení na GitLab Pages
- **.gitlab-ci.yml** - definice CI/CD pipeline
- **docker-compose.yml** - definice pro sestavení a spuštění multi-kontejnerové aplikace pomocí Dockeru

Adresář aplikace backendu obsahuje tyto složky a soubory:

- **src** - obsahuje vlastní zdrojové kódy aplikace ve frameworku Nest.js
  - **auth** - zdrojové kódy týkající se autentizace a autorizace uživatelů
  - **core** - zdrojové kódy týkající se funkcí jádra aplikace, které jsou použity napříč celou aplikací

    - **database** - zdrojové kódy týkající se přístupu k databázi
    - **features** - zdrojové kódy jednotlivých funkcí aplikace, jsou dále rozděleny do logických modulů
- **test** - obsahuje jednotkové testy, integrační testy a E2E testy API
- **.env** - obsahuje proměnné prostředí
- **.eslintrc.js** - konfigurace nástroje ESLint pro statickou analýzu
- **.prettierrc** - konfigurace nástroje Prettier pro statickou analýzu
- **app.yaml** - konfigurace pro nasazení aplikace na Google Cloud Platform
- **dangerfile.ts** - skript pro nástroj Danger.js
- **Dockerfile** - definice pro sestavení Docker image aplikace backendu
- **package.json** - metadata a závislosti projektu backendu
- **webpack.config.js** - konfigurace nástroje Webpack pro sestavení

Adresář aplikace frontendu obsahuje tyto složky a soubory:

- **components** - obsahuje React komponenty použité napříč celou aplikací
- **cypress** - obsahuje definici jednotkových testů, testů komponent a E2E testů celé aplikace (backend + frontend dohromady)
- **lib** - zdrojové kódy funkcí aplikace
- **pages** - obsahuje React komponenty pro každou stránku aplikace, které jsou strukturované ve složkách dle URL adresy
- **public** - statické soubory, např. obrázky
- **queries** - definice GraphQL dotazů
- **styles** - CSS styly
- **types** - vygenerované typy a pomocné funkce z GraphQL API a definice dotazů
- **.env** - proměnné prostředí

- **.eslintrc.js** - konfigurace nástroje ESLint pro statickou analýzu
- **.prettierrc** - konfigurace nástroje Prettier pro statickou analýzu
- **app.yaml** - konfigurace pro nasazení aplikace na Google Cloud Platform
- **codegen.yml** - konfigurace nástroje pro generování kódu z GraphQL API a definice dotazů
- **Dockerfile** - definice pro sestavení Docker obrazu aplikace frontendu
- **package.json** - metadata a závislosti projektu frontendu

5.5 Databáze

Ze schématu architektury, viz obrázek 5.1, je vidět, že pro perzistentní ukládání dat využívá backend objektově-relační databázi PostgreSQL. Na obrázku B.1 v příloze je vidět ER diagram, který znázorňuje datový model a jednotlivé tabulky v databázi:

- **users** - v tabulce jsou uloženy obecné údaje o uživateli jako email, jméno, příjmení a heslo, které je uloženo v šifrované formě pomocí funkce pro odvození klíče scrypt[1].
- **workspaces** - v tabulce jsou uloženy jednotlivé pracovní prostředí a u každého je označení, zda se jedná o defaultní prostředí svého vlastníka viz tabulka *user_workspaces*.
- **projects** - v tabulce jsou uloženy projekty, které jsou k pracovnímu prostředí přiřazeny pomocí cizího klíče *workspaceId*, také zde je sloupec *default*, který udává, zda se jedná o defaultní projekt daného pracovního prostředí.
- **user_workspaces** - tabulka propojuje uživatele a pracovní prostředí pomocí dvojice primárních klíčů *userId* a *workspaceId*. Každá dvojice uživatel - pracovní prostředí může mít v tabulce pouze jeden záznam a pokud tomu tak je, znamená to, že uživatel je součástí daného pracovního prostřed a sloupec roles říká, jaké role uživatel v tomto pracovním prostředí má.

[1] `https://www.tarsnap.com/scrypt.html`

- **user_projects** - podobně jako v předchozím případě, tabulka propojuje uživatele a projekt pomocí dvojice primárních klíčů *userId* a *projectId*. Sloupec *price* udává hodinovou sazbu uživatele na daném projektu.
- **entries** - tabulka obsahuje jednotlivé změřené záznamy odpracovaného času. Pomocí cizích klíčů je napojená na uživatele, pracovní prostředí a projekt. (Pracovní prostředí je v tomto případě redundantní záměrně i přesto, že by se dalo zjistit z relace na projekt. Tato informace se ale nikdy nemění a zjednodušují se tak některé dotazy.) Záznam má dále datum a čas začátku a konce (pokud je konec `null`, měření stále probíhá), popis a případně odkaz na platbu jejíž je součástí.
- **payments** - tabulka obsahuje požadavky na platby, kde kromě odkazů na uživatele a pracovní prostředí je celková cena, čas, datum a čas prvního a posledního záznamu které jsou součástí platby a informace, kdy byl požadavek vytvořen a zda byl zaplacen.

5.6 API

Jak bylo zmíněno, komunikace mezi frontendem a backendem probíhá přes GraphQL API, které backend vystavuje a frontend má jeho URL adresu definovanou jako proměnnou prostředí a volá na něj HTTP požadavky pomocí zmíněné knihovny Apollo Client. GraphQL je dotazovací jazyk, který pomocí srozumitelné struktury a silné typové kontroly umožňuje klientům specifikovat, která data potřebují a pouze ty jsou přenášena v odpovědi. Na rozdíl od REST API, které poskytuje mnoho endpointů a pro získání požadovaných dat je potřeba provést mnoho požadavků, u GraphQL je možné tyto požadavky sloučit do jednoho. Existují tři typy požadavků v GraphQL a to *query* pro získání dat, *mutation* pro změnu dat (vytvoření, editace nebo smazání) a *subscription* pro poslouchání změn v reálném čase. Subscription použit nebude a dlouhodobé spojení bude nahrazeno WebSocketem a knihovnou Socket.io z důvodu možné obousměrné komunikace.

Nástroj *GraphQL Code Generator*, který je použit na frontendu dokáže vygenerovat typy (třídy a rozhraní) a pomocné funkce v TypeScriptu ze specifikace API a z dotazů, které jsou na frontendu uloženy ve složce *queries* v souborech s koncovkou `.graphql`. V kódu 5.1 je vidět ukázka query pro získání projektů, ke kterým má přihlášený uživatel přístup.

Takto si klient definuje vlastní query *myProjects* a uvnitř ní specifikuje data, která od serveru požaduje. V tomto případě požaduje projekty ze ser-

```
query myProjects {
  projects: myProjects {
    id
    name
    default
    workspaceId
  }
}
```

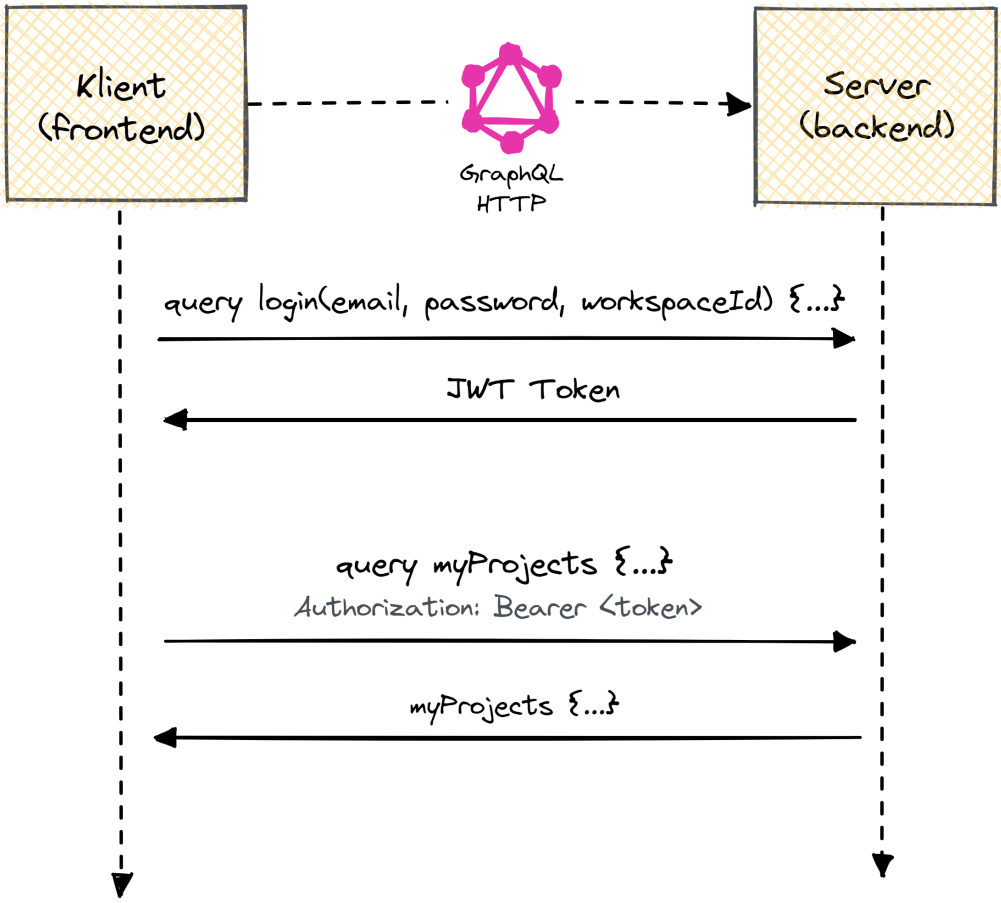
Zdrojový kód 5.1: GraphQL query pro získání projektů

verem definované query *myProjects* a ke každému projektu jeho ID, jméno, informaci zda se jedná o defaultní projekt a ID pracovního prostředí. Pro použití v React komponentě je vygenerovaná funkce *useMyProjectsQuery*, která provede volání API a vrací data očekávaného typu dle specifikace query.

V případě uvedené query potřebuje backend ověřit, zda se dotazuje přihlášený uživatel, potřebuje znát jeho ID a v kterém pracovním prostředí je momentálně přihlášen, aby dokázal vrátit seznam projektů z daného prostředí, ke kterým má uživatel přístup. K tomu je využit standard JWT pro šifrování datové JSON struktury tzv. tokeny, které svým privátním klíčem zašifruje server a klient ho pak používá pro autentizaci.

Princip přihlašování uživatelů je vidět na obrázku 5.2. Uživatel zadá své přihlašovací údaje (email a heslo) a případně zvolí, k jakému pracovnímu prostředí se chce přihlásit. Volbu prostředí dělá automaticky aplikace frontendu, která si pamatuje ID posledního prostředí, kde byl uživatel přihlášen a přihlašuje ho automaticky opět do něho. Pokud na zařízení ještě uživatel přihlášen nebyl, server automaticky zvolí defaultní pracovní prostředí uživatele. Server ověří přihlašovací údaje a pokud je vše v pořádku, vytvoří JWT token a odesílá ho klientovi v odpovědi. Klient následně vkládá tento token do HTTP hlavičky *Authorization* a server pomocí něj veškeré další přístupy uživatele ověřuje. JWT token má pouze omezenou platnost a proto je na odpovědnosti klienta ho v pravidelných intervalech obnovovat pomocí query *refreshToken*. Veřejné query pro přihlašování nebo registraci samozřejmě HTTP hlavičku *Authorization* nemají. Stejný JWT token se používá i při otevírání WebSocket spojení.

Aby server z JWT tokenu, který získá z HTTP hlavičky, dokázal zjistit o kterého se jedná uživatele a z kterého pracovního prostředí aktuálně přistupuje, je potřeba tyto informace vložit do JWT tokenu. Struktura JWT

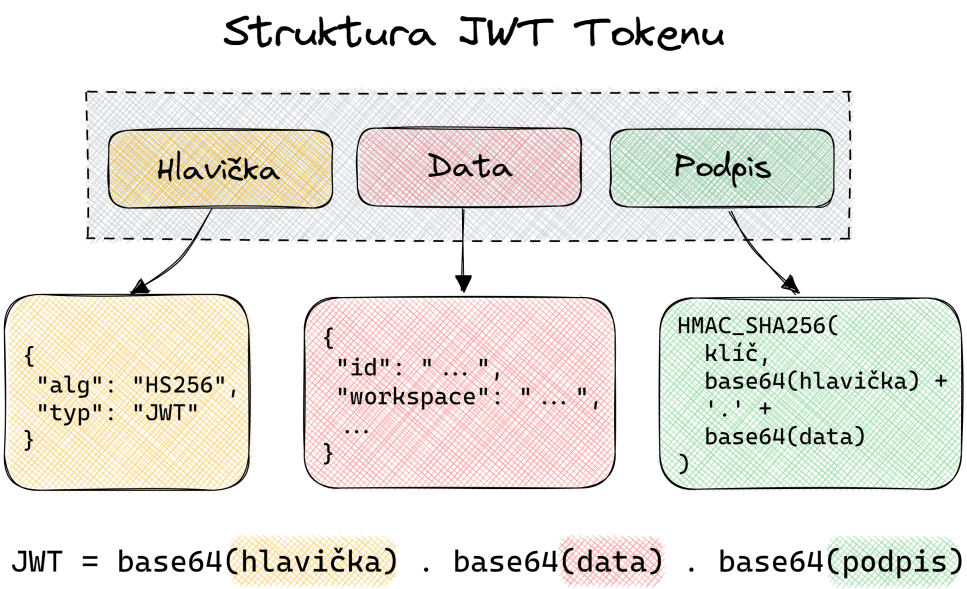


Obrázek 5.2: Použití JWT tokenu při volání API

tokenu je znázorněna na obrázku 5.3 níže a obsahuje tyto části:

- **Hlavička** - která říká, jaký algoritmus je použit ke generování podpisu (viz dále). V tomto případě na obrázku je označení `HS256`, které označuje použitý kryptografický algoritmus `HMAC-SHA256`.
- **Data** - označováno také jako *payload*, obsahuje vlastní data přenášená v tokenu, v případě této aplikace se přenáší mimo jiné hlavně ID uživatele a ID pracovního prostředí.
- **Podpis** - který je sestaven tak, že se pomocí algoritmu specifikovaného v hlavičce a privátního klíče (serveru) zašifruje řetězec, který se složí z hlavičky a dat v *base64* kódování oddělené tečkou.

Finální JWT token je řetězec, který je vytvořen tak, že se v *base64* kódování zakódují všechny tři části struktury tokenu a oddělí se tečkami.



Obrázek 5.3: Struktura JWT tokenu

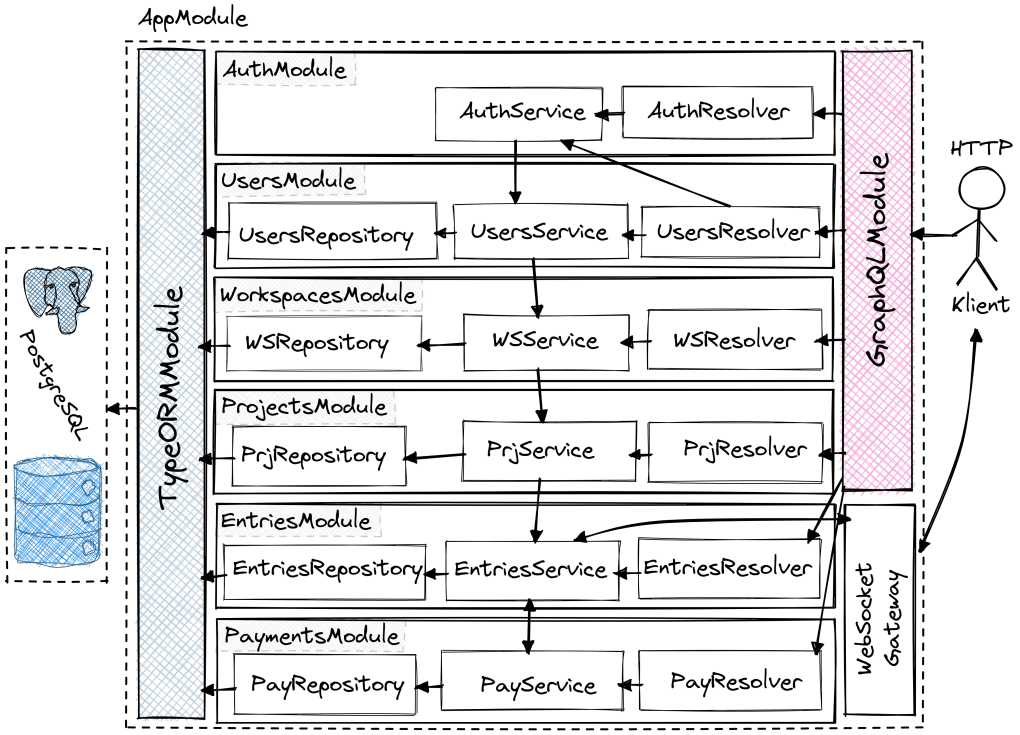
Takto sestavený token posílá klient v každém svém požadavku na API a server, poté co ověří podpis a platnost tokenu, ověří, jaké má uživatel práva a zda může provádět požadovanou operaci. A díky takto navržené struktuře, kdy ID pracovního prostředí je přímo součástí JWT tokenu, je při přepnutí pracovního prostředí uživatelem pouze vygenerován nový JWT token a v jeden okamžik může být uživatel přihlášen na dvou různých zařízení v různých pracovních prostředích.

5.7 Backend

Úkolem backendu, který obsahuje vlastní aplikační logiku, je vystavit API pro potřeby klientů (frontendu) a pro každý zpracovaný požadavek na API ověřit, zda má klient dostatečná práva, získat nebo zapsat data z/do databáze a vrátit výsledek v požadovaném formátu.

Architektura backendu vychází z architektury definované frameworkem Nest.js, kdy je kód rozdělen dle funkcionalit do tzv. modulů. Každý modul definuje své závislosti na jiných modulech a seznam komponent, které naopak poskytuje dalším modulům, které jsou závislé na něm. Framework se pak postará o předání konkrétních instancí těchto závislostí pomocí tzv. vkládání závislostí (DI).

Na obrázku 5.4 je vidět dělení aplikace backendu na jednotlivé moduly, hlavní komponenty v jednotlivých modulech a komunikace mezi nimi.



Obrázek 5.4: Moduly backendu

- **GraphQLModule** - tento nainstalovaný modul je součástí knihovny *nestjs/graphql* a vystavuje GraphQL API. Každý požadavek zpracuje a směruje na další komponentu tzv. resolver. Resolver je třída, která pomocí anotací označí své metody jako vstupní body pro zpracování GraphQL query nebo mutace. Resolver je odstíněn od zdroje dat a pro zpracování požadavku typicky využívá služeb tzv. service tříd, které mají přes tzv. repository přístup ke zdroji dat (databázi). Návratová hodnota z metody resolveru je vrácena jako odpověď na požadavek klientovi. V ukázce kódu 5.2 je vidět zpracování mutace pro spuštění nového měření, která je označena anotací `@Mutation` s návratovou hodnotou typu `Entry` a argumenty typu `StartEntryArgs`. Zpracování se předá do komponenty `EntriesService`.

- **TypeORMModule** - tento modul je nainstalovaný z knihovny *nestjs/typeorm* a poskytuje přístup k databázi pomocí objektově relačního mapování, kdy se entity (třídy v TypeScriptu), které pro jednoduchost nebyly zaneseny do schématu výše, mapují na tabulky v relační databázi. Z entit jsou také vygenerovány pomocné třídy tzv. repository, které poskytují funkce pro manipulaci s daty v dané tabulce (viz např. `UsersRepository`).

```
@Mutation(() => Entry)
async startNewEntry(
  @Args() args: StartEntryArgs,
  @CurrentUser() user: User,
): Promise<Entry> {
  return this.entriesService.startNewEntry(args, user);
}
```

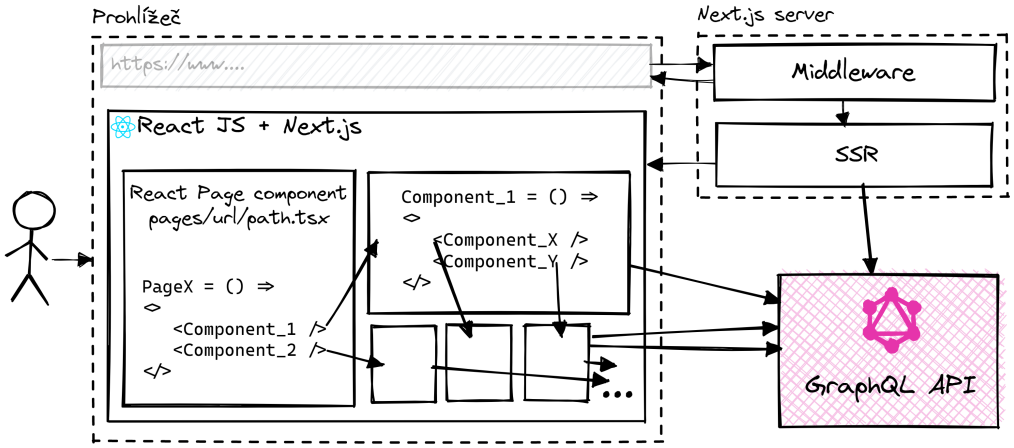
Zdrojový kód 5.2: Resolver mutace pro spuštění nového měření

- **AuthModule** - modul který má na starosti autentizaci a autorizaci uživatelů, `AuthResolver` zpracovává query pro přihlášení nebo obnovení JWT tokenu.
- **UsersModule** - má na starosti funkce pro správu uživatelů, získání informací o uživatelích, editaci profilu, registraci nového účtu a další.
- **WorkspacesModule** - tento modul poskytuje funkce pro manipulaci s pracovními prostředími.
- **ProjectsModule** - má na starosti funkce pro správu projektů, editaci projektů a přístup uživatelů k projektům.
- **EntriesModule** - má na starosti funkce pro měření odpracovaného času a manipulace s těmito záznamy. Tento modul komunikuje také s komponentou `WebSocket Gateway`, pomocí které dokáže číst a zapisovat data z/do WebSocketu. Např. pokud uživatel spustí či zastaví měření, je odeslána tato informace na všechna zařízení, kde je přihlášen ten samý uživatel a klient tuto změnu ihned v reálném čase promítne v uživatelském rozhraní.
- **PaymentsModule** - má na starosti funkce pro správu plateb, vytváření plateb, označování stavu plateb a další.

5.8 Frontend

Úkolem frontendu je prezentace dat a jejich modifikace prostřednictvím GraphQL API poskytované backendem. Frontend zná URL adresu API jako svou proměnnou prostředí.

Architektura frontendu vychází z principů frameworků Next.js a React a je znázorněna na obrázku 5.5. V serverové části frontendu prochází každý



Obrázek 5.5: Architektura frontendu

HTTP požadavek přes tzv. *middleware*, který dokáže požadavek odchytit a změnit jeho standardní způsob zpracování, např. přesměrování pokud uživatel není přihlášen a nemá pro zobrazení stránky dostatečná práva. Pokud požadavek přesměrován není, je dále zpracován Next.js frameworkem, který pomocí SSR vykreslí stránku na serveru a odesílá ji jako odpověď na požadavek do prohlížeče. Sestavení stránky na serveru se sestává ze stejných React komponent jako v prohlížeči (viz dále), pro jednoduchost toto není do diagramu zaneseno. React komponenty se skládají z dalších komponent a tvoří tak tzv. strom komponent, ze kterých jsou pomocí frameworku React vygenerovány HTML a JavaScript soubory webové stránky. Jednotlivé komponenty mohou využívat funkcí, které komunikují s GraphQL API pomocí knihovny Apollo Client a které byly vygenerovány již zmíněným nástrojem *GraphQL Code Generator*. React komponenty jsou stylovány pomocí CSS stylů a knihoven TailwindCSS a MUI.

5.9 Spuštění aplikace

Pro spuštění frontendu a backendu pro ladění při vývoji jsou v souboru *package.json* připraveny skripty `dev` respektive `start:dev`, které spustí frontend na portu 3000 a backend na portu specifikovaném v souboru *backend/.env*. Před spuštěním je třeba nastavit další proměnné prostředí a to parametry pro připojení k databázi na backendu a URL adresu GraphQL API na frontendu.

Pro jednodušší práci je připraven soubor *docker-compose.yml*, který pomocí nástroje *Docker Compose* sestaví a spustí multi-kontejnerovou aplikaci, skládající se z kontejnerů backendu, frontendu a Postgres databáze. Všechny

proměnné prostředí jsou nastaveny (soubor *.env*), stačí tedy pouze příkaz: `docker-compose up` a celá aplikace je spuštěna.

Pro samostatné sestavení a spuštění jak frontendu tak backendu postačí příkazy `yarn build` a `yarn start`. Podrobnější informace k instalaci a ovládání aplikace viz uživatelská příručka v příloze A.

6 Tvorba CI/CD pipeline

V této kapitole bude popsána samotná tvorba CI/CD pipeline v prostředí GitLab a implementace potřebných nástrojů do ukázkové aplikace z předchozí kapitoly. Nejprve bude popsána metodika využívání Git repozitáře, která byla při vývoji ukázkové aplikace aplikována, následně budou vzneseny cíle a požadavky na CI/CD pipeline a použité nástroje a nakonec budou představeny implementační a konfigurační detaily CI/CD pipeline, konfigurace a implementace nástrojů využívaných v pipeline do ukázkové aplikace, způsob nasazení aplikace a prezentace výstupů.

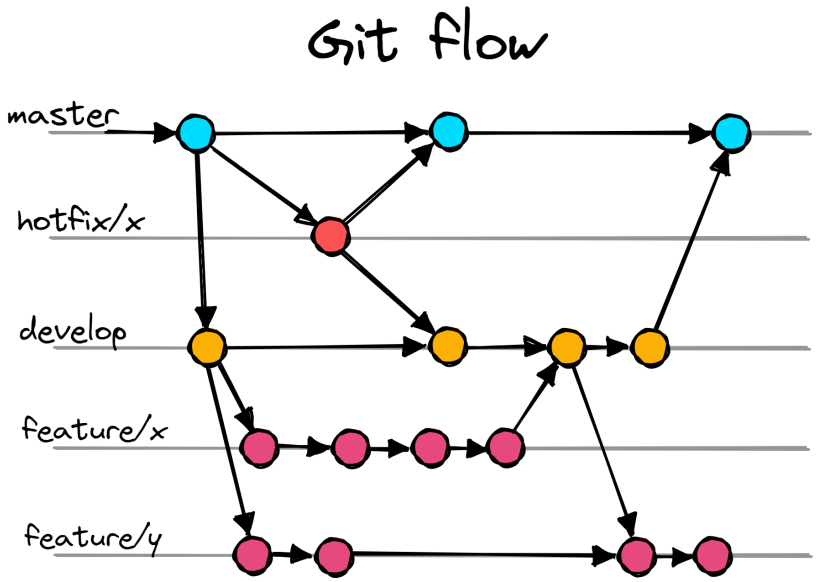
6.1 GitLab repozitář

Pro potřeby této práce byl použit GitLab SaaS ve verzi *Free*, ve kterém byl vytvořen projekt pro ukázkovou aplikaci. Pro spouštění CI/CD jobů byl použit sdílený GitLab Runner (viz kapitola 3.3.1). Repozitář je veřejně[1] přístupný.

Při vývoji ukázkové aplikace byla stanovena metodika používání Git větví (tzv. Git flow), kterou je nutné mít specifikovanou i pro potřebu tvorby pipeline, protože, jak bude dále určeno, pipeline a její joby budou vytvářeny v závislosti na dané větvi.

Na obrázku 6.1 jsou znázorněny používané větve, jmenné konvence a slučování změn mezi větvemi. Hlavní větví je větev `master`, ze které se vždy dělá nová verze a nasazení do produkčního prostředí. `Master` tak obsahuje vždy kód verze, která je aktuálně k dispozici koncovým uživatelům v produkčním prostředí. Větev `master` je v GitLabu nastavena jako chráněná a nelze do ní odesílat commity přímo, ale pouze přes MR. V typickém případě se vytváří MR z větve `develop`, která slouží k vývoji a do které vývojáři přidávají nové funkce nebo opravy chyb. Nedělají to ale napřímo, protože v takovém případě by několik vývojářů pracovalo současně na jedné větvi a teoreticky i na stejných souborech a navíc by změny pro jednotlivé funkce nebyly přehledně rozdělené. Místo toho si vývojáři z větve `develop` vytvářejí své tzv. *feature* větve, které pojmenovávají `feature/popis_funkce` a do nich implementují změny týkající se dané funkcionality či opravy chyby. Poté, co je nové funkce hotová, se feature větev slučuje zpět do `develop` větve přes MR, který je ideálním místem pro spuštění automatických testů a kontrolu kódu

[1] `https://gitlab.com/Fori5/dp`



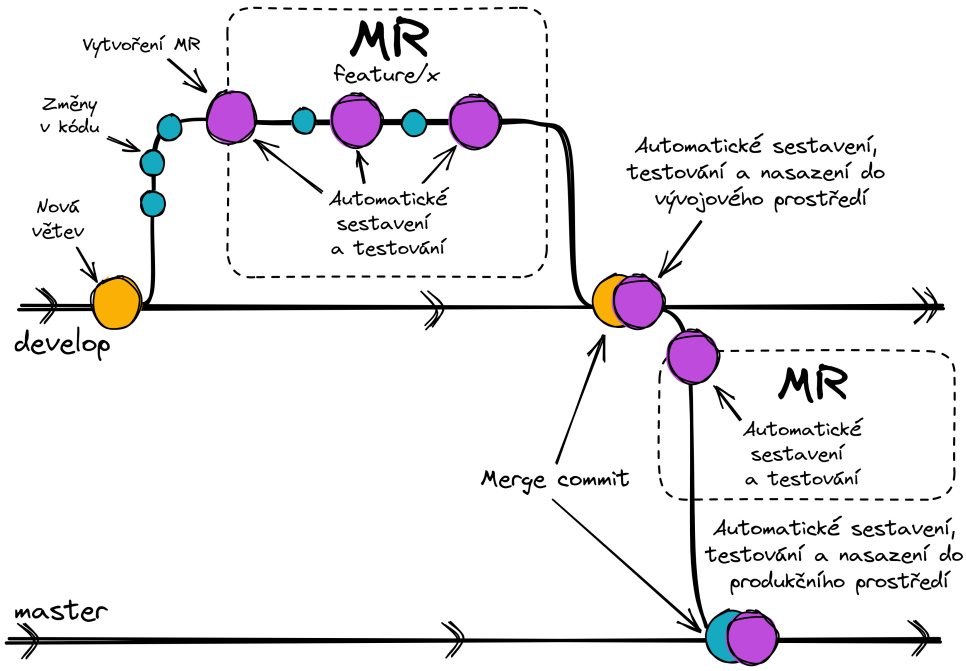
Obrázek 6.1: Git flow

dalšími vývojáři. Z `develop` větve může být nasazena aplikace do testovacího prostředí. Větev `develop` je také chráněná stejně jako `master`, z důvodu přístupu k chráněným proměnným (viz kapitola 3.3.1), commity přímo do ní ale odesílat lze. V případě nutné rychlé opravy do produkční verze, kdy `develop` větev nemusí být ve stavu, kdy by bylo možné ji sloučit do větve `master`, je vytvořena tzv. *hotfix* větev, která se sloučí přímo do `master` větve (a také do `develop` větve).

6.2 Cíle a požadavky

V zadání práce je uvedeno, že cílem pipeline je pro každý MR ověřit, že proběhlo úspěšné sestavení, nejsou detekovány žádné chyby ani varování ze statické analýzy kódu, proběhly úspěšně všechny jednotkové a integrační testy a další užitečné nástroje. Tyto požadavky byly dále rozšířeny o automatické nasazení do vývojového a produkčního prostředí při každém commitu do větve `develop` respektive `master`. Toto je znázorněno na obrázku 6.2, kde při vytvoření nové *feature* větve z větve `develop` a commitů do ní se žádná pipeline nevytváří. Až v momentu vytvoření MR je vytvořena pipeline pro sestavení a testování, která se spouští jak v moment vytvoření MR tak při každém dalším commitu do MR. Toto je identické pro jakýkoliv MR tedy i z větve `develop` do větve `master`. V případě sloučení MR je vytvořen tzv. merge commit do cílové větve a v ten moment je vytvořena pipeline, která kromě sestavení a testování obsahuje také automatické nasazení do

prostředí v závislosti na cílové větvi. V případě commitu do `master` větve je navíc provedeno nasazení vygenerovaných HTML dokumentací na GitLab Pages[2].



Obrázek 6.2: Automatické nástroje v GitLab flow

Kromě výše zmíněných požadavků na pipeline, které říkají, kdy se které akce mají spouštět, byly definovány i další požadavky na pipeline a to:

- **Rychlost** - běh a vykonávání pipeline by měl být co nejrychlejší, měla by se použít cache, tam kde to je výhodné a joby, u kterých je to možné by měly běžet paralelně.
- **Konfigurovatelnost** - proměnné pipeline by neměly být součástí souboru *.gitlab-ci.yml*, ale měly by být definovány v prostředí GitLabu, obvzláště jedná-li se o chráněné proměnné jako jsou klíče nebo o proměnné jejichž hodnota je závislá na typu prostředí.
- **Výstupy** - výstupy z jednotlivých jobů jako je přehled proběhlých testů, pokrytí testy, atd., by měly být ve formátu, který dokáže GitLab prezentovat přímo u výsledků konkrétní pipeline a jednotlivých jobů.
- **Srozumitelnost** - kód definice pipeline v souboru *.gitlab-ci.yml* by měl být srozumitelný a komentovaný, aby mu rozuměli ostatní vývojáři v týmu.

[2] `https://fori5.gitlab.io/dp/`

6.3 Implementace nástrojů do aplikace

Ještě před vlastní tvorbou CI/CD pipeline a její definice v souboru *.gitlab-ci.yml* byly do ukázkové aplikace implementovány potřebné nástroje, které byly vybrány v předchozích kapitolách. Všechny potřebné knihovny byly nainstalovány pomocí nástroje *yarn* a jsou tak součástí souborů *package.json*. Pro jednodušší spouštění uvnitř CI/CD pipeline byly v těchto souborech vytvořeny skripty, které lze spustit příkazem `yarn "název skriptu"`. Dále následuje popis těchto skriptů a implementačních detailů jak pro backend tak pro frontend.

6.3.1 Backend

Pro backend byly v kapitolách 4.2 a 5.3 vybrány nástroje ESLint, Prettier, Jest, Danger.js a Compodoc, které je možné spustit následujícími skripty.

build

Skript *build* má formát

```
nest build --webpack --webpackPath ./webpack.config.js
```

a sestaví aplikaci pomocí nástroje Webpack, jehož konfigurace se nachází v souboru *webpack.config.js*, kde lze nastavit např. vstupní bod aplikace, výstupní soubor nebo optimalizace a použité pluginy. Výstupem skriptu je v Node.js spustitelný soubor *dist/main.js*.

format

`prettier --write "src/**/*.ts" "test/**/*.ts"` je formát skriptu *format*, který spustí automatické formátování zdrojových souborů v TypeScriptu ve složkách *src* a *test* pomocí nástroje Prettier. Tento skript v CI spouštěn nebude, ale je vhodné ho provést vždy před nahráním změn do vzdáleného repozitáře (existují také nástroje a pluginy, které toto zajistí sami). Konfigurace nástroje Prettier je v souboru *.prettierrc*, jehož obsah je vidět v kódu 6.1.

Pravidla definují konzistentní styl zdrojových textů, jako používání pouze jednoduchých uvozovek, povinné čárky za parametry na konci řádky, šířku odsazení nebo maximální délku řádky.

```
{
  "singleQuote": true,
  "trailingComma": "all",
  "endOfLine": "auto",
  "tabWidth": 4,
  "printWidth": 120
}
```

Zdrojový kód 6.1: Obsah souboru .prettierrc

lint

Skript *lint* má formát `eslint "src/**/*.ts" --max-warnings=0` a spustí statickou analýzu nad TypeScript soubory ze složky *src* pomocí nástroje ESLint. Konfigurace nástroje ESLint je v souboru *.eslintrc.js*, kde je definováno, že se mají použít pravidla z již předpřipravených konfigurací (pluginů) *typescript-eslint/recommended* a *prettier/recommended*. Některá nevyhovující pravidla z těchto pluginů jsou ovšem v konfiguraci vypnuta. Tím, že je použit plugin pro nástroj Prettier, jsou při analýze kódu ověřována i pravidla formátování z konfigurační souboru *.prettierrc* (kód 6.1) zmíněného výše. Hodnota 0 parametru `max-warnings` při spouštění skriptu zajistí, že běh končí chybou i v případě, že ESLint či Prettier detekují pouze varování.

test

Skript *test*, který má formát

```
jest --outputFile test-results.json --json --ci
↪  --reporters=default --reporters=jest-junit --coverage
```

spustí jednotkové a integrační testy pomocí nástroje Jest. Výsledky testů budou zapsány do souborů *test-results.json* a *rspec.xml*, ve kterém budou ve formátu JUnit. Přepínač `coverage` říká, že se má provést také pokrytí testy a výsledky ve formátech HTML a Cobertura budou uloženy ve složce *coverage*. Konfigurace nástroje Jest je součástí souboru *package.json*, kde jsou specifikovány formáty výstupů a reportů a že se mají spouštět testy ze souborů s názvem končícím na *.spec.ts*. Ve složce *test* je implementováno několik ukázkových testů.

test:e2e

Skript *test:e2e*, který má formát

```
jest --config ./test/jest-e2e.json --outputFile
↪  test-results.json --json --ci --reporters=default
↪  --reporters=jest-junit
```

a spustí E2E testy API opět pomocí nástroje Jest. Konfigurace pro E2E testy je v souboru *./test/jest-e2e.json* a specifikuje, že se budou spouštět testy s názvem končícím *.e2e-spec.ts*. Formát výsledků je stejný jako v předchozím případě. Pro potřeby funkčních testů je nutný běh celé aplikace včetně databáze, parametry pro připojení k databázi se berou z proměnných prostředí (soubor *.env*) a testy si databázi sami naplní potřebnými testovacími daty. Nyní následují ukázky testů ze souboru *auth.e2e-spec.ts*. Všechny testy jsou uvnitř skupiny testů specifikované pomocí definice `jest.describe`. V kódu 6.2 je před spuštěním testů z dané skupiny, proveden kód uvnitř definice `beforeAll`, kde je vytvořen testovací Nest.js modul a z něho je spuštěna aplikace. V databázi je vytvořen testovací uživatel pro kterého je získán JWT token.

```
beforeAll(async () => {
  const moduleFixture: TestingModule =
      await Test.createTestingModule({
        imports: [AppModule]
      }).compile();
    app = moduleFixture.createNestApplication();
    await app.init();
    const auth = app.get<AuthService>(AuthService);
    const user = await createTestUser(app.get(DataSource));
    token = await auth.createJwtToken(user);
});
```

Zdrojový kód 6.2: Vytvoření testovacího modulu (auth.e2e-spec.ts)

Nejprve se testuje přístup bez JWT tokenu (viz kód 6.3), kdy se provede HTTP požadavek na root aplikace a očekává se v odpovědi HTTP stavový kód *401* (Unauthorized).

Další test v kódu 6.4 je již komplexnejší a testuje požadavek na přihlášení, kdy posílá požadavek na GraphQL API pomocí přihlašovací query `loginQuery`, která zde pro úsporu místa není uvedena. Query jsou předány jako parametry přihlašovací údaje testovacího uživatele. Očekává se návratový stavový kód *200* a JWT token, který je hned použit k dalšímu requestu a testuje se, zda je ověření JWT tokenu úspěšné.

```
test('UnAuthorized', () => {
    return request(app.getHttpServer())
      .get('/').expect(401);
});
```

Zdrojový kód 6.3: Test neautorizovaného přístupu (auth.e2e-spec.ts)

```
test('Login', () => {
  return request(app.getHttpServer())
    .post('/graphql')
    .send({ query: loginQuery, variables: {...}})
    .expect(200).expect((res) => {
      expect(res.body.data.login.user.email).toBe(...);
      loginToken = res.body.data.login.token;
    })
    .then(() => {
      return request(app.getHttpServer().get('/')
        .auth(loginToken, { type: 'bearer' })
        .expect(200);
    });
  });
});
```

Zdrojový kód 6.4: E2E test přihlášení (auth.e2e-spec.ts)

doc

`compodoc src -p tsconfig.json -d ./doc` je formát skriptu *doc*, který pomocí nástroje Compodoc vygeneruje HTML dokumentaci ze zdrojových TypeScript souborů a uloží její soubory do složky *doc*. Aby byla dokumentace kvalitní a měla nějakou informační hodnotu, je potřeba dostatečně komentovat jednotlivé třídy, atributy, metody a parametry. Nástroj Compodoc dokáže detekovat komponenty architektury aplikace ve frameworku Nest.js jako např. výše zmíněné moduly a vykreslí závislosti mezi nimi.

doc graphql

Skript *doc graphql* má formát

```
nest start --entryFile generate-schema && spectaql
↪  spectaql.config.yml -t ./doc/graphql
```

a nejprve ze zdrojových kódů vygeneruje soubor *schema.graphql* s popisem GraphQL API, který následně předá nástroji SpectaQL, pomocí něhož vygeneruje dokumentaci ke GraphQL API ve formátu HTML. Konfigurace, ve které lze nastavit další parametry jak bude výsledná vygenerovaná dokumentace vypadat, se nachází v souboru *spectaql.config.yml* a soubory vygenerované dokumentace jsou uloženy do složky *doc/graphql*.

danger

Skript *danger* má formát `danger ci` a spustí vykonávání skriptů pomocí nástroje Danger.js. Tento nástroj přistupuje k metadatům z Gitu a GitLabu a proto bude spuštěn pouze z GitLab CI/CD a to pouze z MR. Danger.js spustí skripty v souboru *dangerfile.ts*, který si z knihovny danger importuje potřebné funkce `message`, `danger`, `warn` a `fail`, které přidají do MR v GitLabu komentář a graficky znázorní závažnost sdělení viz kapitola 4.2.3. V ukázce ze skriptu v kódu 6.5 jsou příklady využití. V prvním případě se pouze vypíše zpráva se seznamem změněných souborů v daném commitu. Ve druhém případě se zjišťuje, zda byla provedena změna v souboru *backend/package.json* a zároveň nebyla provedena změna v souboru *backend/yarn.lock*. Soubor *yarn.lock* je vygenerován nástrojem yarn v momentu instalace závislostí pomocí příkazu `yarn install` a obsahuje graf všech závislostí projektu a jejich verzí a zajišťuje, že při instalaci z toho souboru budou vždy instalace závislostí totožné. Instalace v rámci CI/CD vždy vychází pouze ze *souboru yarn.lock*. No a pokud byla provedena změna v souboru *package.json*, ale v souboru *yarn.lock* ne, je možné, že vývojář přidal či změnil nějaké závislosti, ale nespustil příkaz `yarn install`.

```
const modifiedMD = danger.git.modified_files.join('- ');
message('Changed Files in this PR: \n - ' + modifiedMD);
const p = danger.git.modified_files
  .includes('package.json');
const l = danger.git.modified_files.includes('yarn.lock');
if (p && !l) {
    warn('Changes made to backend/package.json, but...');
}
```

Zdrojový kód 6.5: Ukázka ze souboru dangerfile.ts

I přes to, že Danger.js je nainstalován v projektu aplikace backendu, obsahuje kontrolní funkce i pro frontend, protože repozitář je pro obě části aplikace společný a skripty budou spuštěny pro změny v kterékoliv z nich.

6.3.2 Frontend

Pro frontend byly v kapitolách 4.2 a 5.3 vybrány nástroje ESLint, Prettier, Cypress, Danger.js a TypeDoc, které je možné spouštět těmito skripty.

build

Skript *build* má formát `next build` a sestaví aplikaci pomocí nástroje Webpack, který je již součástí frameworku Next.js a jeho vlastní konfigurace je možná v souboru *next.config.js*. Výstupem skriptu je složka *.next/standalone* s Node.js spustitelným souborem *server.js*.

lint

Skript *lint* má formát `eslint next lint --max-warnings=0` a spouští statickou analýzu nad zdrojovými soubory pomocí nástroje ESLint. Konfigurace nástroje ESLint je v souboru *.eslintrc.js*, kde je definováno, že se mají použít pravidla z již předpřipravených konfigurací (pluginů) *next/core-web-vitals*, *typescript-eslint/recommended* a *prettier/recommended*. Některá nevyhovující pravidla z těchto pluginů jsou ovšem v konfiguraci vypnuta. Tím, že je použit plugin pro nástroj Prettier, jsou při analýze kódu ověřována i pravidla formátování z konfigurační souboru *.prettierrc*.

cypress:component

Skript *cypress:component* má formát `cypress run --component` a pomocí nástroje Cypress spustí jednotkové testy a testy React komponent ze souborů končících na *.cy.ts* a *.cy.tsx*. Konfigurace nástroje Cypress se nachází v souboru *cypress.config.ts*. Testy komponent využívají příkazu `cy.mount`, kterému se předává inicializovaná instance React komponenty. Cypress komponentu vykreslí a dalšími příkazy lze testovat např. obsah vytvořeného HTML. V kódu 6.6 je příklad testu komponenty `TimeView`, které se předává čas ve dnech, hodinách, minutách a sekundách.

Příkazem `cy.mount` se komponenta vykreslí, příkaz `cy.get('span')` vyhledá v HTML element `span` a pomocí `contains` se testuje, že obsahem nalezeného tagu je správně naformátovaný předaný čas. Výstupem testů je složka *junit* se soubory reportů ve formátu JUnit.

```
describe('<TimeView />', () => {
  it('renders', () => {
    cy.mount(<TimeView days={0} hours={0}
                    minutes={0} seconds={0} />)
    cy.get('span').contains('0:00:00')
  })
})
```

Zdrojový kód 6.6: Test komponenty TimeView

cypress:e2e

Skript *cypress:e2e* má formát `cypress run --e2e` a spustí E2E testy pomocí nástroje Cypress. Tentokrát jsou spuštěny testy ze souborů končících na *.e2e.cy.ts*. E2E testy požadují běh celé aplikace, tzn. frontendu i backendu s databází, která je naplněna testovacími daty, a očekávají, že frontend je dostupný na `localhostu` na portu `3000`. Jsou využívány příkazy Cypress pro manipulaci s prohlížečem jako `cy.visit`, pro zadání URL adresy webu, `cy.get` pro získání HTML elementu, nad kterým lze volat další příkazy např. pro validaci obsahu, nastavení obsahu, stisknutí tlačítka apod. Výstupem testů jsou stejně jako v případě testů komponent JUnit reporty ve složce *junit* a také report pokrytí E2E testy ve složce *coverage*.

Test na ukázce v kódu 6.7 testuje flow přihlášení uživatele. Test očekává, že v databázi existuje uživatel s emailem `test@test.com` a heslem `test`. Pomocí příkazu `cy.visit` se načte přihlašovací obrazovka, příkazem `cy.get` se vyhledají vstupní pole přihlašovacího formuláře pro email a heslo a příkazem `type` se tyto pole vyplní přihlašovacími údaji. Následně se simuluje stisk tlačítka příkazem `click`. Nyní je třeba počkat na zpracování požadavku a přesměrování, což je provedeno otestováním hodnoty v titulku stránky, který je rozdílný na domovské stránce po přesměrování a na přihlašovací stránce. Poté se testuje hodnota URL adresy. Následně je web přesměrován na URL `/settings/profile`, kde se nachází stránka nastavení profilu uživatele, a hledá se vstupní emailové pole a testuje se, zda jeho hodnota odpovídá emailové adrese přihlášeného uživatele.

merge-junit

Protože příkazy *cypress:component* a *cypress:e2e* generují mnoho souborů s reporty výsledků testů ve formátu JUnit, je potřeba tyto výsledky sloučit do jednoho souboru pro potřeby získání hromadných výsledků nástroji

```
describe('Login', () => {
  it('should login', () => {
    cy.visit('http://localhost:3000/login')
    cy.get('#loginEmail').type('test@test.com')
    cy.get('#loginPassword').type('test')
    cy.get('button').click()

    cy.title().should('eq', 'DP')
    cy.url().then(url => {
      expect(url).to.be.oneOf(['http://localhost...','...'])
    })

    cy.visit('http://localhost:3000/settings/profile')
    cy.get('input[type=email]').should('have.value','...')
  })
})
```

Zdrojový kód 6.7: E2E test přihlášení uživatele

v GitLab CI/CD. Toho je dosaženo nástrojem junit-report-merger a příkazem `jrm ./rspec.xml ./junit/*.xml`. Příkaz vezme všechny XML soubory ze složky *junit* a vytvoří z nich jeden soubor *rspec.xml*.

cypress:dev

Při spouštění testů při vývoji je možné využít skript *cypress:dev*, který má formát `cypress open` a spustí Cypress v prohlížeči spolu s nástroji pro vývojáře, pomocí kterých lze např. spouštět testy ve více prohlížečích, vidět vykreslené komponenty u testů komponent nebo v reálném čase vidět jednotlivé akce E2E testů.

doc

Skript *doc* má formát `typedoc --options typedoc.json` a pomocí nástroje TypeDoc vygeneruje HTML dokumentaci ze zdrojových TypeScript souborů ze složek *pages*, *components*, *lib* a *types*. Dokumentace je uložena do složky *doc*. Konfigurace nástroje se nachází v souboru *typedoc.json*.

6.4 Implementace pipeline

V této části bude konečně popsána samotná implementace CI/CD pipeline a obsah souboru *.gitlab-ci.yml* včetně jeho fragmentů kódu. Nejprve bude pipeline popsána z širšího pohledu a rozdělena na stage a joby, poté budou jednotlivé joby popsány samostatně.

```
image: node:16

stages:
  - build
  - test
  - deploy
```

Zdrojový kód 6.8: Konfigurace pipeline: image a stage

V kódu 6.8 je vidět, že defaultní Docker image pro všechny joby je `node:16`. Joby které vyžadují jiný image mohou toto defaultní nastavení přepsat. Dále jsou v kódu specifikované tři stage:

- **build** - stage obsahuje tyto joby:
  - **build-be** - sestaví backend
  - **build-fe** - sestaví frontend
- **test** - stage obsahuje tyto joby:
  - **danger** - spustí definované skripty pomocí nástroje Danger.js
  - **lint-test-be** - spustí statickou analýzu kódu backendu
  - **unit-test-be** - spustí jednotkové a integrační testy na backendu
  - **e2e-test-be** - spustí E2E testy na backendu
  - **lint-test-fe** - spustí statickou analýzu kódu frontendu
  - **unit-test-fe** - spustí jednotkové, integrační a komponentové testy na frontendu
  - **e2e-test-fe** - spustí E2E testy na frontendu
- **deploy** - stage obsahuje tyto joby:
  - **deploy-be-dev** - spustí nasazení backendu na vývojové prostředí

- **deploy-be-prod** - spustí nasazení backendu na produkční prostředí
- **deploy-fe-dev** - spustí nasazení frontendu na vývojové prostředí
- **deploy-fe-prod** - spustí nasazení frontendu na produkční prostředí
- **pages** - spustí nasazení statických souborů na GitLab Pages

Uvedené stage, respektive jejich joby, jsou spouštěny sériově (pokud není specifikováno jinak) v takovém pořadí, v jakém jsou stage definovány.

6.4.1 Proměnné

V prostředí GitLabu (Settings -> CI/CD -> Variables), byly definovány proměnné, ke kterým lze přistupovat v souboru *.gitlab-ci.yml* pomocí notace `$NAZEV_PROMENNE`.

Proměnná `DANGER_GITLAB_API_TOKEN` obsahuje API token GitLabu pro použití nástrojem Danger.js a je dostupná ve všech prostředích a větvích. Ostatní proměnné jsou označené jako chráněné (Protected) a je k nim přístup pouze z chráněných větví (`master` a `develop`, viz kapitola 6.1). Proměnné `DOTENV_BACKEND` a `DOTENV_FRONTEND` obsahují proměnné prostředí pro backend a frontend a obě jsou uvedeny dvakrát - pro vývojové a pro produkční prostředí. Obsah těchto proměnných bude použit pro obsah soboru *.env* při nasazení aplikace. Proměnné `PROJECT_ID` a `SERVICE_ACCOUNT` obsahují údaje potřebné pro nasazení aplikace (více informací bude uvedeno dále) a opět jsou obě ve variantách jak pro vývojové tak pro produkční prostředí.

6.4.2 Cache

Pro urychlení sestavení je definovaná jak na backendu tak na frontendu cache pro složku *node_modules*, která obsahuje soubory nainstalovaných knihoven a je výsledkem příkazu `yarn install`. Klíčem cache je obsah souboru `yarn.lock`, jehož vznik a účel byl popsán v kapitole 6.3.1 u příkazu *danger*.

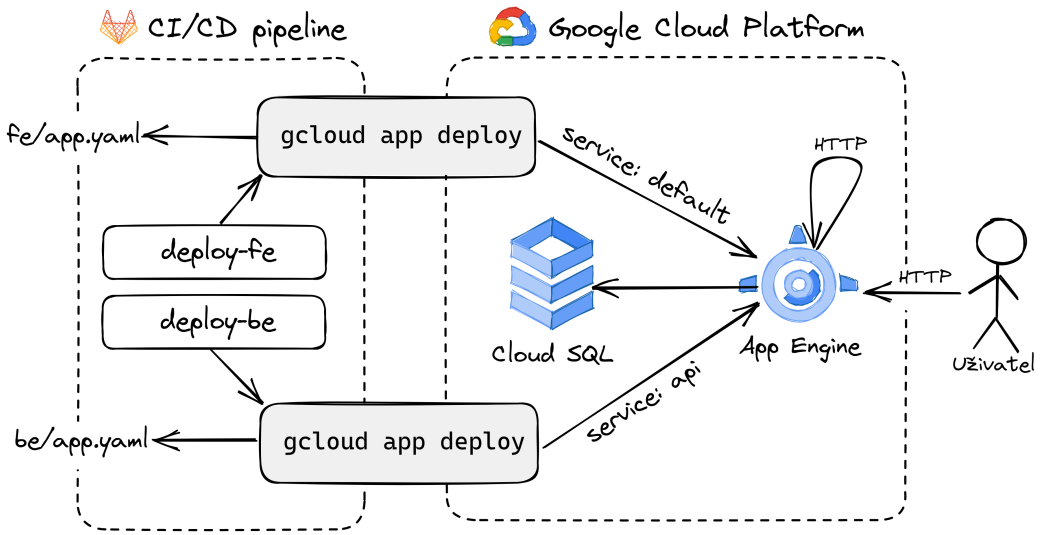
Pokud nebyly změněny závislosti projektu a nebyl tedy změněn ani soubor *yarn.lock*, lze použít složku *node_modules* z cache místo nové instalace závislostí. Definice cache pro backend je vidět v kódu 6.9. Cache je definována jako job s tečkou na začátku, který nebude v pipeline vytvořen, a bude použit pouze pro rozšíření definice jiných jobů. A protože soubor *gitlab-ci.yml* je ve formátu YAML, lze použít tzv. kotvy (anchor), kdy pomocí znaku `&` je definována část označená jako kotva a následně se lze pomocí znaku `*` na tuto část odkázat. Další joby tak mohou tuto cache využít pomocí

`*be-cache`, konkrétní příklady budou uvedeny dále. Hodnota `pull-push` atributu `policy` říká, že se daný job na začátku svého běhu pokusí z cache číst a na konci do ní zapisuje.

```
.be-cache: &be-cache
  key:
    files:
      - backend/yarn.lock
  paths:
    - backend/node_modules
  policy: pull-push
```

Zdrojový kód 6.9: Konfigurace pipeline: cache

6.4.3 Nasazení



Obrázek 6.3: Diagram nasazení na Google Cloud

Joby ve stage *deploy* mají na starosti nasazení aplikace do vývojového či produkčního prostředí. V této práci byly z ekonomických důvodů obě prostředí totožná a vždy byla nasazena pouze vývojová nebo produkční verze. Aplikace se nasazuje na službu App Engine platformy Google Cloud, která umožňuje hostování aplikací při minimální konfiguraci, která je z velké části automatická včetně zabezpečení, škálování, vyvážení zátěže apod. Na jedné

infrastruktuře App Enginu může odděleně běžet více tzv. služeb, které budou využity pro frontend a backend.

Na obrázku 6.3 je znázorněno nasazení z CI/CD pipeline. Job `deploy-fe` nasazuje frontend (služba *default*) pomocí příkazu `gcloud app deploy`, který čte ze souboru *fe/app.yaml*, který specifikuje běhové prostředí (nodejs), proměnné prostředí, automatické přesměrování nebo parametry automatického škálování. Identicky probíhá nasazení backendu a využití souboru *be/app.yaml*. Backend má k dispozici Postgres databázi díky Google Cloud službě Cloud SQL, jejíž parametry pro připojení jsou součástí proměnné `DOTENV_BACKEND`.

6.4.4 Joby

V této části budou uvedeny jednotlivé joby, u každého z nich budou popsány podmínky, za kterých je job součástí pipeline a je spuštěn, které příkazy vykonává a jestli jsou jeho výstupem nějaké artefakty. Protože je často velmi podobný, bude kód definice jobu ukázán jen u některých a případně bude o nepodstatné (i povinné) atributy zkrácen. Kompletní kód je vidět v souboru *gitlab-ci.yml*.

```
build-be:
  stage: build
  script:
    - cd backend
    - yarn install --frozen-lockfile
    - yarn build
  cache:
    - <<: *be-cache
  artifacts:
    paths:
      - backend/dist
      - backend/node_modules
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH == "master" ||
                          $CI_COMMIT_BRANCH == "develop"
```

Zdrojový kód 6.10: Job build-be

build-be

Job *build-be*, který je součástí stage *build* a jehož definice je vidět v kódu 6.10, sestaví aplikaci backendu pomocí zmíněného skriptu *build*. Job využívá cache *be-cache*, jeho artefakty jsou složky *dist*, která obsahuje výsledný sestavený kód a *node_modules* s nainstalovanými knihovnami. Tyto dvě složky poté budou dostupné jobům z další stage. Job *build-be* je spuštěn v případě commitu do MR nebo do větví `master` a `develop`, jak je definováno pomocí klíčového slova `rules`.

build-fe

Job *build-fe*, součást stage *build*, je podobný jobu *build-be* výše, jen sestaví aplikaci frontendu. Job využívá cache *fe-cache* pro složku *node_modules*, jeho artefakty jsou složky *.next/static*, *.next/standalone* a *node_modules*. Job je opět spuštěn v případě commitu do MR, nebo `master` a `develop` větve.

danger

Job *danger*, který je součástí stage *test* a jehož definice je vidět v kódu 6.11, spustí nástroj Danger.js pomocí skriptu *danger*. Skripty a soubor *dangerfile.ts* byly zmíněny v kapitole 6.3.1. Job je spuštěn pouze pro commity do MR a pomocí klíčového slova `need` definuje, že pro svůj běh potřebuje pouze dokončení jobu *build-be*, kvůli instalaci knihovny Danger.js, a nemusí tak čekat na všechny ostatní joby z předchozí stage (*build-fe*).

```
danger:
  stage: test
  script:
    - yarn danger
  needs:
    - build-be
  only:
    - merge_requests
```

Zdrojový kód 6.11: Job danger

lint-test-be

Job *lint-test-be*, který je součástí stage *test*, spustí statickou analýzu kódu backendu pomocí skriptu `lint`. Job definuje, že pro svůj běh potřebuje pouze dokončení jobu *build-be* a je spuštěn pro každý commit do MR a větví `master` a `develop`.

unit-test-be

Job *unit-test-be*, jehož zkrácená část je vidět v kódu 6.12, je součásti stage *test* a spustí jednotkové a integrační testy na backendu pomocí skriptu `test`. Job pro svůj běh potřebuje pouze dokončení jobu *build-be* a je spuštěn pro každý commit do MR a větví `master` a `develop`. Artefakty jobu jsou výsledky testů a pokrytí testy. Pomocí klíčového slova `reports` jsou označeny reporty testů ve formátu JUnit a Cobertura a pomocí klíčového slova `coverage`, je definován regulární výraz, který je hledán ve výpisu z vykonávání jobu a který obsahuje informaci o pokrytí testy. Všechny tyto reporty dokáže GitLab zpracovat a prezentovat přímo u výsledku běhu dané pipeline (viz kapitola 7.2).

```
unit-test-be:
  script:
    - yarn test
  artifacts:
    paths:
      - backend/test-results.json
      - backend/rspec.xml
      - backend/coverage
    reports:
      junit:
        - backend/rspec.xml
      coverage_report:
        coverage_format: cobertura
        path: backend/coverage/cobertura-coverage.xml
  coverage: '/Lines\s*:\s*(\d+.?\d*)%/'
```

Zdrojový kód 6.12: Job unit-test-be

e2e-test-be

Job *e2e-test-be*, viz kód 6.13, je součástí stage *test* a spustí pomocí skriptu `test:e2e` E2E testy pro backend pro každý commit do MR nebo větví `master` a `develop`. Job pro svůj běh potřebuje pouze dokončení jobu *build-be* a jeho artefakty jsou výsledky testů stejně jako u předchozího jobu *unit-test-be*. Protože E2E testy potřebují kompletní aplikaci backendu včetně databáze, definuje job Docker službu s obrazem `postgres:12.2-alpine`, která spustí Postgres databázi. Její parametry jsou brány z proměnných, proto je job definuje pomocí klíčového slova `variables`. A protože backend čte své proměnné prostředí ze souboru *.env*, jsou tyto parametry změněny i v něm pomocí příkazu echo.

```
e2e-test-be:
  services:
    - postgres:12.2-alpine
  variables:
    POSTGRES_DB: e2e_test
    POSTGRES_USER: postgres
    POSTGRES_PASSWORD: postgres
    POSTGRES_HOST_AUTH_METHOD: trust
  script:
    - echo DB_HOST=postgres >> ".env"
    - echo DB_NAME=e2e_test >> ".env"
    - yarn test:e2e
```

Zdrojový kód 6.13: Job e2e-test-be

lint-test-fe

Job *lint-test-fe*, který je součástí stage *test*, spustí statickou analýzu kódu frontendu pomocí skriptu `lint`. Job definuje, že pro svůj běh potřebuje pouze dokončení jobu *build-fe* a je spuštěn pro každý commit do MR a větví `master` a `develop`.

unit-test-fe

Job *unit-test-fe*, jehož zkrácená část je vidět v kódu 6.14, je součásti stage *test* a spustí jednotkové, integrační a komponentové testy na frontendu pomocí

skriptu `cypress:component`. Protože knihovna Cypress potřebuje mít dostupné binární soubory stejnojmenného programu, používá job Docker image `cypress/base:16.18.1`, který má tyto soubory již nainstalované. Výsledky testů jsou sloučeny do jednoho souboru pomocí skriptu `merge-junit`. Job pro svůj běh potřebuje pouze dokončení jobu *build-fe* a je spuštěn pro každý commit do MR a větví `master` a `develop`. Artefakty jobu jsou výsledky testů a reporty podobně jako u jobu *unit-test-be*.

```
unit-test-fe:
  image: cypress/base:16.18.1
  script:
    - yarn cypress:component
    - yarn merge-junit
```

Zdrojový kód 6.14: Job unit-test-fe

e2e-test-fe

Job *e2e-test-fe*, jehož část je vidět v kódu 6.15, je součástí stage *test* a spustí E2E testy pro frontend pomocí skriptu `cypress:e2e` pro každý commit do MR nebo větví `master` a `develop`. Protože E2E testy frontendu vyžadují běh kompletní aplikace včetně backendu, job pro svůj běh potřebuje dokončení jobů *build-fe* i *build-be*. A protože backend pro svůj běh potřebuje databázi, definuje job potřebu Docker služby s obrazem `postgres:12.2-alpine`, která spustí Postgres databázi stejně jako u jobu *e2e-test-be*. Opět je definován Docker image `cypress/base:16.18.1`. Při vykonávání skriptu job nejprve spustí na pozadí aplikaci backendu, které do souboru *.env* předal parametry pro připojení k databázi. Poté předá URL adresu backendu do souboru *.env* frontendu a musí znovu sestavit frontend, protože framework Next.js čte proměnné ze souboru *.env* v době kompilace. Poté je spuštěn také frontend a pomocí nainstalovaného nástroje *postgresql-client* je naplněna databáze testovacími daty pomocí SQL skriptu ze souboru *frontend/cypress/data/init.sql*. Následně jsou konečně spuštěny E2E testy, sloučeny jejich výsledky a ukončeny obě aplikace. Artefakty jobu jsou opět výsledky testů a reporty.

deploy-be

Job *deploy-be* (viz kód 6.16) není v pipeline vytvářen, ale jedná se pouze o šablonu, kterou využívají joby *deploy-be-dev* a *deploy-be-prod*. Job definuje docker image `google/cloud:sdk-alpine` pro potřeby programu *gcloud*.

```
e2e-test-fe:
  script:
    - yarn global add pm2
    - cd backend
    - pm2 start --name backend yarn -- start
    - cd ../frontend
    - yarn cypress install --force
    - echo BACKEND_URL=http://localhost:5000 >> ".env"
    - yarn build
    - pm2 start --name frontend yarn -- start
    - psql -U postgres -d e2e_test -h postgres -f init.sql
    - yarn cypress:e2e
    - yarn merge-junit
  after_script:
    - pm2 delete backend && pm2 delete frontend
```

Zdrojový kód 6.15: Job e2e-test-fe

Nejprve je překopírován obsah proměnné `DOTENV_BACKEND` do souboru *.env*. Poté se provede autorizace do nástroje *gcloud*, kterému je předán soubor s obsahem proměnné `SERVICE_ACCOUNT` a příkazem `gcloud app deploy` je následně nasazena aplikace backendu na App Engine. Příkazu je předáno ID projektu z proměnné `PROJECT_ID` a konfigurační soubor *app.yaml*.

```
.deploy_be: &deploy_be_definition
  image: google/cloud-sdk:alpine
  stage: deploy
  script:
    - echo "$DOTENV_BACKEND" > .env
    - echo $SERVICE_ACCOUNT > /tmp/$CI_PIPELINE_ID.json
    - gcloud auth activate-service-account --key-file
                                /tmp/$CI_PIPELINE_ID.json
    - gcloud app deploy --project $PROJECT_ID app.yaml
```

Zdrojový kód 6.16: Šablona jobu deploy_be

Šablona *deploy_be* je vidět v kódu 6.16, skutečný job *deploy-be-dev*, který tuto šablonu využívá je vidět v kódu 6.17. Job definuje prostředí a to, že má být spuštěn pouze pro commit do větve `develop`. Job *deploy-be-prod*

je totožný, jen definuje prostředí `prod` a je spuštěn pro commit do větve `master`.

```
deploy-be-dev:
  <<: *deploy_be_definition
  environment:
    name: dev
    url: https://api-dot-dp-378511.ey.r.appspot.com
  only:
    - develop
```

Zdrojový kód 6.17: Job deploy-be-dev

deploy-fe

Jako u backendu i zde je šablona, kterou poté využijí joby pro nasazení na vývojové a produkční prostředí. Oproti backendu je ale u frontendu nutné aplikaci znovu sestavit po změně proměnných prostředí. Jsou tak vytvořeny dvě šablony *pre_deploy_fe_definition* a *deploy_fe_definition*. První z nich, která má Docker image `node:16`, zapíše do *.env* souboru proměnné dle prostředí (definované v jobu *deploy-fe-dev 1/2* pro větev `develop` nebo *deploy-fe-prod 1/2* pro větev `master`) a sestaví aplikaci. Druhá šablona má Docker image `google/cloud-sdk:alpine` a provede nasazení na App Engine. Tuto šablonu využívají joby *deploy-fe-dev 2/2* (`develop`) a *deploy-fe-prod 2/2* (`master`). Pojmenování jobů s označením `1/2` a `2/2` na konci způsobí, že v prostředí GitLabu budou joby vizuálně sloučeny do jednoho.

pages

Job *pages*, který je součástí stage *deploy*, slouží k nasazení statických souborů na GitLab Pages. Jedná se o již zmíněné HTML dokumentace, které jsou vygenerovány na backendu i frontendu skriptem `doc`. Dále je na backendu skriptem `doc:graphql` vygenerovaná dokumentace pro GraphQL API. V rámci jobu je vytvořena složka *public*, do které jsou přesunuty tyto vygenerované soubory a také HTML výstupy z analýzy pokrytí testy z předchozích jobů. Celá složka *public* je artefaktem jobu a díky tomu, že job se jmenuje *pages*, GitLab automaticky artefakty z jobu nasadí na GitLab Pages. Job je spuštěn pouze pro větev `master`, aby publikovaná dokumentace byla vždy pro aktuální verzi v produkčním prostředí.

7 Ověření pipeline

V této kapitole bude ověřeno a otestováno, že navržená a implementovaná pipeline splňuje všechny požadavky ze zadání, které byly upřesněny v kapitole 6.2.

7.1 Způsob ověření

Při ověřování pipeline bude napodobován běžný proces implementace nové funkce do aplikace viz kapitola 3.3.1, kdy si vývojář vytvoří svojí vlastní větev, do které implementuje novou funkci a následně vytvoří MR do větve `develop` a poté do větve `master`. Při tom se bude sledovat, zda se pipeline vytváří dle pravidel definovaných v kapitole 6.2 a zda všechny nástroje implementované v pipeline fungují správně a dávají požadované výstupy. Bude se tedy testovat, že:

- **Konfigurace pipeline je validní** - ověření, zda definice pipeline v souboru *.gitlab-ci.yml* je validní YAML soubor a použití klíčových slov a hodnot odpovídá specifikaci GitLab CI/CD.
- **Vytvoření pipeline** - pipeline je vytvořena a obsahuje správné joby tak jak má, v závislosti na commitu do MR, větve `develop` nebo `master`.
- **Sestavení** - joby pro sestavení správně sestaví aplikaci a pokud se vyskytne chyba, končí chybou jak daný job, tak celá pipeline.
- **Statická analýza** - joby pro statickou analýzu správně detekují možné chyby a varování a pipeline pak končí chybou.
- **Testy** - joby pro jednotkové, integrační a E2E testy správně detekují chyby a pipeline končí chybou. Pokud testy proběhnou v pořádku, jsou vidět výsledky testů a pokrytí testy přímo v prostředí GitLabu.
- **Danger.js** - v případě, že je detekována změna souboru *package.json*, ale není změněn soubor *yarn.lock* (viz kapitola 6.3.1) a jedná se o MR, Danger k němu přidá komentář.
- **Nasazení** - aplikace je v případě commitu do větve `develop` nebo `master` nasazena na vývojové respektive produkční prostředí.

- **Dokumentace** - v případě commitu do větve `master` jsou vygenerované všechny tři dokumentace (backend, frontend, GraphQL API) a nasazené na GitLab Pages.
- **Cache** - funguje cache a urychluje vykonávání pipeline

7.2 Proces ověřování

Dále následuje popis jednotlivých provedených akcí, spolu s ukázkou vytvořené pipeline, jejím výsledkem a výstupy a popisem, které body z předchozího seznamu tím byly ověřeny. Veškeré kroky jsou prováděny přímo do veřejného GitLab repozitáře[1], který byl použit k implementaci ukázkové aplikace i pipeline a lze tak jejich skutečnost a výsledky ověřit.

1. Nová větev

V prvním kroku je vytvořena nová větev `feature/test_pipeline` z větve `develop`. Do větve je uděláno několik commitů a je udělán push do vzdáleného repozitáře. Je ověřeno, že pipeline nebyla vytvořena, protože žádné joby nemají být spuštěny pro commit do jiné větve než `develop` nebo `master` a nebo do MR. Tím, že se pipeline nevytvoří, je zároveň i ověřeno, že soubor *.gitlab-ci.yml* je validní dle specifikace, protože v opačném případě by byla vytvořena prázdná pipeline se zprávou "Unable to create pipeline" a "yaml invalid".

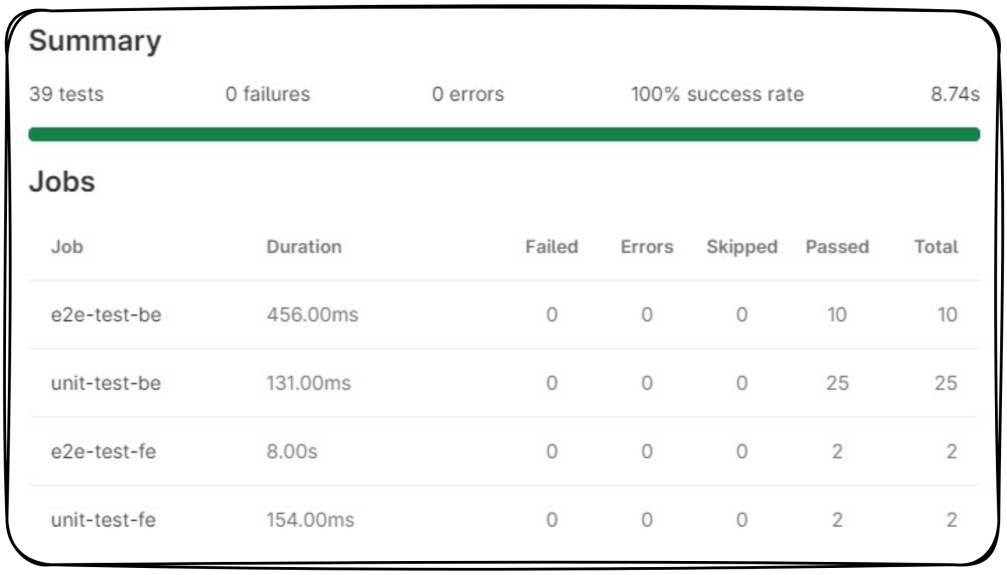
Obrázek 7.1: MR feautre/test_pipeline -> develop

[1] `https://gitlab.com/Fori5/dp`

Obrázek 7.2: Informace o stavu pipeline po úspěšném dokončení

2. MR do větve develop

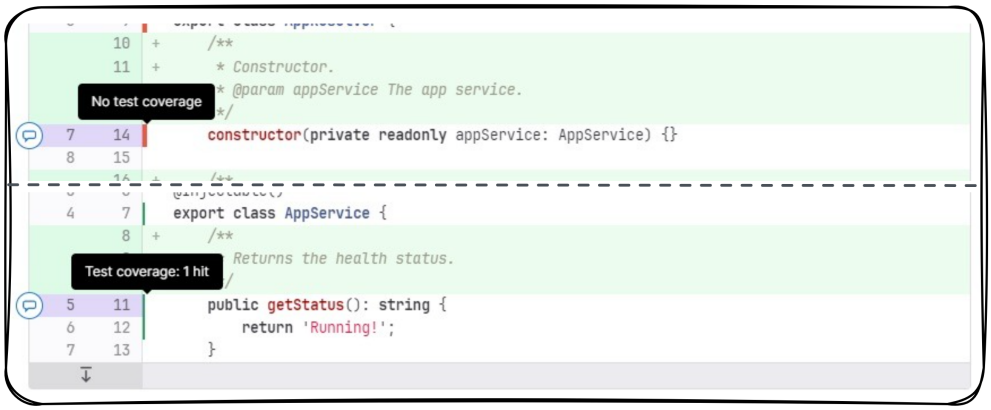
Ve druhém kroku je z větve `feature/test_pipeline` vytvořen MR[2] (viz obrázek 7.1) do větve `develop`. Po vytvoření MR je ihned vytvořena a spuštěna pipeline, protože (interně) se jedná o commit do MR. Ověřuje se, že pipeline doběhla v pořádku (viz obrázek 7.2) a obsahuje pouze joby ze stage *build* a *test*. Tím je také částečně ověřeno, že fungují joby pro sestavení aplikace a testování, a sice, že nehlásí žádné negativně pozitivní chyby a testy potřebují pro svůj běh nainstalované závislosti a sestavenou aplikaci z předchozích jobů.



Obrázek 7.3: Výsledky testů v jobech pipeline

Dále je ověřena přítomnost výsledků testů přímo v detailu pipeline v prostředí GitLabu v záložce *Tests*, které jsou rozděleny dle jobů (viz obrázek 7.3) a lze se jimi proklikat až na výsledky jednotlivých testovacích sad. V seznamu jobů pipeline, u těch které to podporují, by také měla být informace o sumě

[2] `https://gitlab.com/Fori5/dp/-/merge_requests/26`



Obrázek 7.4: Informace o pokrytí testy v MR

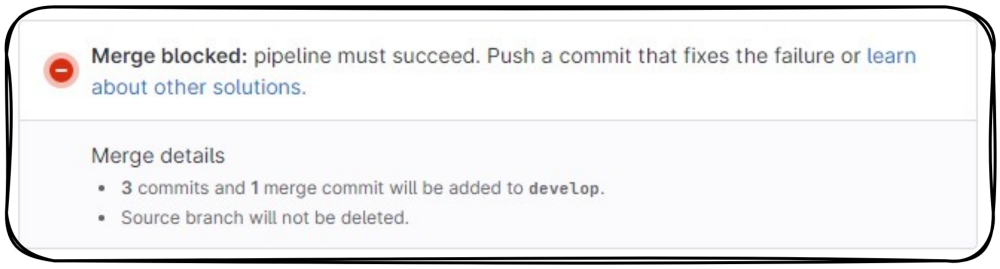
výsledku analýzy pokrytí testy v procentech a přímo v MR v záložce *Changes*, kde jsou vidět změny v daném MR, by měly být zvýrazněny pokryté a nepokryté bloky kódu (viz obrázek 7.4). Tím je ověřeno správné generování reportů výsledků testů a výsledků analýzy pokrytí testy.

3. Chyba v sestavení

V tomto kroku se bude ověřovat, zda joby ve stage *build*, správně detekují chybu překladu a celá pipeline končí chybou. Do MR je přidán commit, ve kterém jsou syntaktické chyby, které zajistí chybu překladu frontendu i backendu. Je ověřeno, že oba joby ze stage *build* končí chybou stejně jako celá pipeline (viz obrázek 7.5) a žádné další joby nejsou vůbec spuštěny.

Obrázek 7.5: Informace o stavu pipeline a jobů po chybě

Díky tomu, že pipeline selhala a že je to tak nastaveno v repozitáři, nelze nyní MR sloučit (viz obrázek 7.6), dokud nebudou chyby opraveny a nová pipeline nedoběhne v pořádku a bez chyb.



Obrázek 7.6: Blokované sloučení MR

4. Chyba ve statické analýze

V tomto kroku se bude ověřovat detekce chyb ve statické analýze v jobech *lint-test-be* a *lint-test-fe*. Do MR je tedy přidán commit, který opravuje chyby překladu zanesené do kódu v předchozím kroku, a který zajistí detekování nějaké chyby nebo varování ve statické analýze. Např. na backendu přidáme do nějaké části kódu definici proměnné `x` pomocí klíčového slova `var` (viz kód 7.1). Nástroj ESLint v tomto případě hlásí chybu na použití klíčového slova `var` místo `let` nebo `const` a varování, že proměnná nebyla nikde použita a že chybí středník na konci řádky. Na frontendu je zaneseno porušení pravidel formátování definovaných nástrojem Prettier.

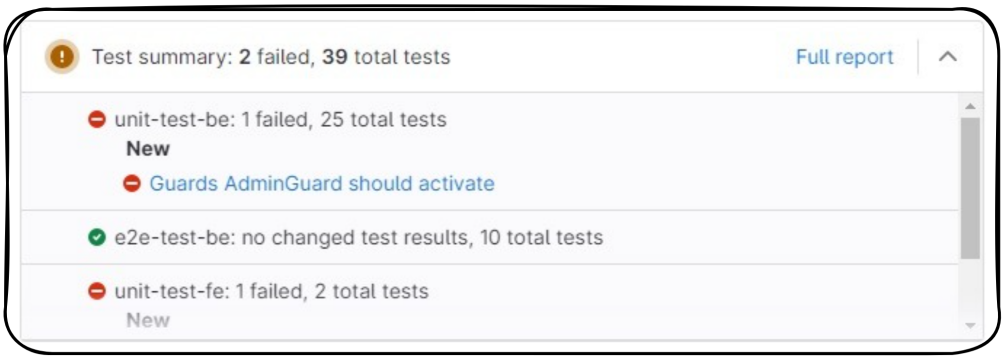
```
health(): string {
  var x = 10
  return this.appService.getStatus();
}
```

Zdrojový kód 7.1: Chyba ve statické analýze na backendu

Je ověřeno, že oba joby, jak *lint-test-be* tak *lint-test-fe* končí chybou stejně jako celá pipeline. Při tomto testu bylo zjištěno, že framework Next.js (frontend), pouští statickou analýzu automaticky při překladu a tak chybou skončil už job *build-fe*. Defaultní konfigurace frameworku Next.js je ale taková, že překlad končí chybou, pouze pokud ESLint detekuje chyby, ale varování nevadí. Proto bylo ověřeno, že varování odchytí job *lint-test-fe*.

5. Chyba v jednotkových testech

V tomto kroku se bude ověřovat detekce chyb v jednotkových a integračních testech v jobech *unit-test-be* a *unit-test-fe*. Do MR je přidán commit, který opět opravuje předchozí chyby a který zajistí chybu některého testu na



Obrázek 7.7: Výsledky testů v MR

backendu i frontendu. Na backendu je udělána změna ve třídě `AdminGuard`, která má na starosti ověření, že uživatel má minimálně práva administrátora, a která je součástí jednotkových testů v souboru v *test/auth/guards.spec.ts*. Změna zajistí, že uživatel s rolí admin je zamítnut. Na frontendu je udělána změna v komponentě `TimeView`, ve které je změněno formátování času, které je testováno v testu v souboru *cypress/components/TimeView.cy.tsx*. Z výsledku pipeline je ověřeno, že zmíněné dva joby skončily chybou stejně jako celá pipeline. V detailu pipeline v záložce *Tests* lze vidět, které testy skončily chybou včetně výpisu z jejich vykonávání. Přímo v MR je také souhrn výsledků testů, viz obrázek 7.7.

6. Chyba v E2E testech

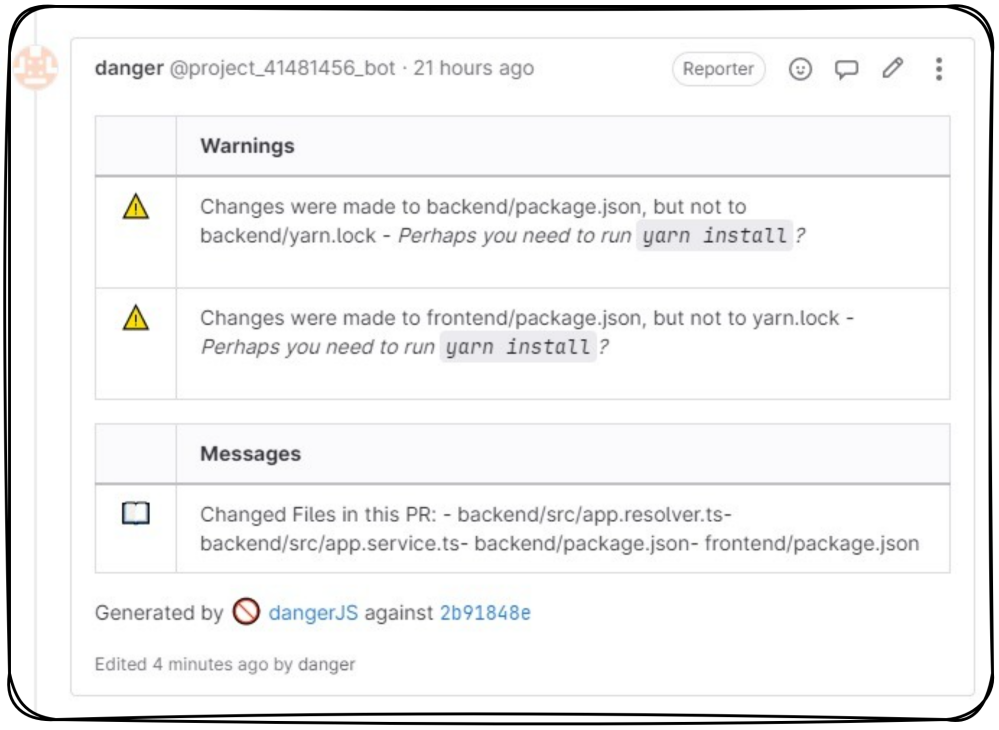
V dalším kroku bude provedeno to samé jako v kroku předchozím, jen pro E2E testy. Na backendu je udělaná chyba ve zpracování GraphQL query *login* resolveru `AuthResolver`, která způsobí, že i když jsou přihlašovací údaje správné API vrací chybu `UNAUTHENTICATED`. Query je testována v souboru *test/auth.e2e-spec.ts*. Tato chyba by měla být odhalena i testy na frontendu v souboru *cypress/e2e/auth.e2e.cy.ts*, kde je již popsaný E2E test (viz kapitola 6.3.2), který provádí přihlášení uživatele. Je ověřeno, že pipeline končí chybou kvůli chybě jobů *e2e-test-be* a *e2e-test-fe* a že jsou opět vidět výsledky v detailu pipeline a MR jako v předchozím kroku.

7. Danger JS varování

V dalším kroku je ověřena funkčnost jobu *danger* a skriptů v souboru *be/dangerfile.ts*, které byly popsány v kapitole 6.3.1 a ověřují mimo jiné spuštění příkazu `yarn install` po změně souborů *package.json*. Stačí tedy udělat commit, ve kterém budou změněny závislosti v souborech *package.json* na

Obrázek 7.8: MR připravený ke sloučení

backendu i frontendu, např. změna verze některé knihovny, ale nebude provedena instalace pomocí nástroje yarn a soubory *yarn.lock* tak zůstanou beze změny. Protože byly všechny testy opraveny a všechny joby i celá pipeline doběhnou bez chyby, je MR označen jako připraven ke sloučení (viz obrázek 7.8). Je ověřeno, že job *danger* a nástroj Danger.js přidá do MR komentář varující, že po změně souborů *package.json* nebyl proveden příkaz `yarn install` (viz obrázek 7.9).



Obrázek 7.9: Komentář od nástroje Danger.js

8. Zneplatnění cache

V tomto kroku je proveden příkaz `yarn install` z adresářů projektů backend i frontend a díky tomu, že byly v předchozím kroku změněny závislosti v souborech *package.json*, budou změněny soubory *yarn.lock* a tyto změny budou součástí dalšího commitu do MR. Je ověřeno, že z komentáře v MR zmizelo varování o chybějícím příkazu `yarn install`.

Díky tomu, že klíčem cache souborů ve složkách *node_modules* jsou hodnoty souborů *yarn.lock* (viz kapitola 6.4.2), lze v tuto chvíli ověřit, zda cache funguje a přináší urychlení. U výpisu jobů pro sestavení je vidět, že byly instalovány závislosti a že cache není dostupná ("Checking Cache - WARNING: file does not exist"), ale u předchozích pipeline (kde byla použita cache) závislosti instalovány nejsou ("Already up-to-date.") a cache existuje ("Downloading cache.zip, Successfully extracted cache"). Vykonání jobů *build-be* a *build-fe* v předchozí pipeline, která měla složky *node_modules* dostupné z cache z předchozích pipeline, trvalo **3:58 min.** respektive **4:45 min.** U předchozích pipeline byly časy podobné +- 10 sekund. Časy jobů z tohoto kroku pak trvaly **6:13 min.** pro job *build-be* a **6:38 min.** pro job *build-fe*. Z výpisů lze ověřit, že cache funguje a z těchto výsledků vyplývá, že u jobů pro sestavení urychlí vykonávání o více jak **2 min** (cca 32 %).

Obrázek 7.10: Informace o nasazení nové verze - Google Cloud

9. Merge do větve develop

V tomto kroku je provedeno sloučení MR (větve `feature/test_pipeline` do větve `develop`) a je ověřeno, že je správně vytvořená pipeline pro commit do větve `develop`. Je ověřeno, že pipeline obsahuje joby dle požadavků - kromě jobů pro sestavení a testování (bez jobu *danger*) navíc joby pro nasazení na vývojové prostředí (*deploy-be-dev*, *deploy-fe-dev 1/2*, *deploy-fe-dev 2/2*). Je ověřeno, že pipeline doběhne v pořádku a aplikace je nasazena na

vývojové prostředí (obrázek 7.10). Informace o posledním nasazení je vidět také v prostředí GitLabu (Deployments -> Environments, obrázek 7.11).

Obrázek 7.11: Prostředí dev v GitLabu

10. MR větve develop do větve master

V dalším kroku, potom co je ověřeno, že aplikace funguje na vývojovém prostředí, je vytvořen MR[3] z větve `develop` do větve `master`. Ověřuje se, že je vytvořena správná pipeline s joby pro MR a že doběhne v pořádku.

Obrázek 7.12: GitLab repozitář

11. Merge do větve master

V posledním kroku je provedeno sloučení MR z předchozího kroku a ověřuje se, že je vytvořena pipeline s joby pro commit do větve `master`. Pipeline obsahuje joby pro sestavení, testování (kromě jobu *danger*) a pro nasazení na produkční prostředí (*deploy-be-prod*, *deploy-fe-prod 1/2*, *deploy-fe-prod 2/2*) a GitLab Pages (*pages*). Je ověřeno, že se provede nasazení na produkční prostředí stejným způsobem jako v kroku 9 u vývojového prostředí. Dále jsou ověřeny výstupy z jobu *pages*. Na GitLab Pages[4] je nasazen soubor,

[3] `https://gitlab.com/Fori5/dp/-/merge_requests/27`
[4] `https://fori5.gitlab.io/dp`

který obsahuje odkazy na HTML dokumentace (backend, GraphQL API, frontend) a HTML výsledky analýzy pokrytí testy. Je ověřeno, že všechny stránky byly správně vygenerovány. Na hlavní stránce repozitáře jsou také vidět dvě tzv. badge (viz obrázek 7.12), jedna informuje že poslední pipeline z větve `master` skončila úspěšně a druhá o jejím pokrytí testy.

7.3 Zhodnocení ověření

Předchozími kroky byly úspěšně ověřeny všechny požadavky na pipeline, použité nástroje a výstupy. Jednotlivé kroky lze ověřit v MR[5] z veřejného repozitáře[6].

Pipeline splňuje všechny požadavky ze zadání na automatické sestavení a jednotkové a integrační testování po commitu do MR. Z dalších užitečných nástrojů byl implementován nástroj Danger.js, který upozorňuje na různé problémy v kódu a přidává komentáře do MR, nástroje pro generování dokumentace a analýzu pokrytí testy. Navíc byly implementovány také E2E testy. Kromě commitu do MR, byly přidány joby pro nasazení na vývojové a produkční prostředí po commitu do větve `develop` a `master`.

Mezi výhody pipeline patří její optimalizace pomocí cache a paralelizace jobů a to, že jsou spouštěny skripty definované v souborech *package.json*, což umožňuje jejich snadnou modifikaci bez nutnosti zásahu do samotné pipeline. Přesto jsou některé sripty v pipeline rozsáhlé, např. u E2E testů, kde je nutné spustit celou aplikaci včetně databáze a tyto skripty by možná bylo vhodné přesunout do samostatných souborů. Dále by některé akce v pipeline bylo možné sloučit do jednoho jobu, např. statická analýza společně s jednotkovými testy, tím by došlo k dalšímu urychlení vykonávání, protože u některých akcí je připravení prostředí pro běh akce časově náročnější než její samotné vykonávání. Pipeline by mohla být dále rozšířena např. o další analýzy kódu pro nástroj Danger.js, nebo o další typy testů, např. o testy zátěžové.

[5] `https://gitlab.com/Fori5/dp/-/merge_requests/26`
[6] `https://gitlab.com/Fori5/dp`

8 Závěr

V rámci této diplomové práce byla prozkoumána problematika kontinuální integrace (CI) a kontinuálního nasazení (CD) pro JavaScript projekty v prostředí GitLab a nástroji GitLab CI/CD. Po analýze a výběru vhodných nástrojů pro použití v CI/CD pipeline byla implementována ukázková JavaScript (TypeScript) aplikace, do které byly tyto nástroje implementovány a použity v procesu jejího vývoje.

Ukázková aplikace je webová aplikace pro měření a evidenci odpracované doby zaměstnanců na projektech. Zaměstnanci se přihlašují do webového rozhraní, ve kterém si měří svůj čas strávený prací na konkrétním projektu a vidí přehledy a reporty své odpracované doby za určité období dle určitého projektu. Zaměstnavatelé potom mají přehled o odpracovaném čase svých zaměstnanců a tyto výstupy si mohou různě filtrovat. Aplikace je architektonicky rozdělena na dvě samostatné části - backend a frontend a díky tomu bylo možné vyzkoušet více nástrojů a technologií pro použití v CI/CD pipeline.

Navržená a implementovaná CI/CD pipeline, která se skládá celkem ze sedmnácti různých automatizovaných akcí, ověří kvalitu kódu pomocí statické analýzy nástroji ESLint a Prettier, provede sestavení aplikace nástrojem Webpack nebo spustí jednotkové, integrační, komponentové a E2E testy nástroji Jest a Cypress. Jako další užitečný nástroj byl použit Danger.js, který dokáže analyzovat změny kódu v GitLab MR a výsledky analýzy do něj automaticky vkládat jako komentáře. Ze zdrojových kódů aplikace a vytvořeného API rozhraní byly automaticky vygenerovány HTML dokumentace nástroji TypeDoc, Compodoc a SpectaQL. Tyto dokumentace a další užitečné výstupy byly automaticky nasazeny na službu GitLab Pages. Pipeline automaticky nasazuje aplikaci na vývojové a produkční prostředí na službu App Engine platformy Google Cloud. Pro použití pipeline byly navrženy pravidla pro Git workflow, která by měla být dodržována při vývoji aplikace.

Výstupy akcí pipeline, jako např. výsledky automatických testů, jsou ve formátu, který GitLab CI/CD dokáže zpracovat a prezentovat přímo v grafickém prostředí GitLabu ve srozumitelné formě i pro členy týmu neznalých daných technologií.

V poslední kapitole této práce byla na příkladu implementace nové funkce do aplikace ověřena funkčnost a splnění požadavků implementované CI/CD pipeline a použitých nástrojů. Výsledkem práce je plně funkční CI/CD pi-

peline pro GitLab, která zefektivní proces vývoje aplikace a ověření použitelnosti vybraných nástrojů demonstrované na reálném projektu JavaScript aplikace. Práce tak splňuje zadání v celém rozsahu a kromě požadavku na automatické sestavení a jednotkové a integrační testování byly navíc implementovány další užitečné kroky v CI/CD pipeline od analýzy změn v kódu, přes E2E testy a generování dokumentace až po automatické nasazení aplikace. Běh pipeline a akcí v ní je optimalizovaná pomocí cache a paralelizace.

Zkratky

API Application Programming Interface 25, 26, 29, 30, 35, 41–46, 48–51, 58, 60, 65, 73, 79, 83, 84

CD Continuous Delivery, Continuous Deployment 1, 12–16, 20–23, 31, 33, 35–37, 42, 53, 56, 60, 63–65, 67, 74, 84, 85

CI Continuous Integration 1, 12–16, 20–23, 31, 33, 35–37, 42, 53, 56, 60, 63–65, 67, 74, 84, 85

CLI Command Line Interface 31

CSS Cascading Style Sheets 42, 43, 51

DI Dependency Injection 48

DSP Dokument Specifikace Požadavků 2

E2E End to End, funkční testy 5, 34, 35, 42, 43, 58, 62–64, 70, 71, 74, 79, 83–85

ER Entity Relationship 44

HTML Hypertext Markup Language 33, 35, 41, 51, 55, 57, 59–63, 73, 83, 84

HTTP Hypertext Transfer Protocol 13, 25, 29, 41, 45, 46, 51, 58

I/O Input/Output 32

IaC Infrastructure as Code 13, 15, 16

IDE Integrated Development Environment 15

JSON JavaScript Object Notation 46

JWT JSON Web Tokens 46–48, 50, 58

MIT Massachusetts Institute of Technology, svobodná licence 18

MR Merge Request 16, 19, 20, 25, 26, 31, 36, 53, 54, 60, 68–71, 74–84

PDF Portable Document Format 26

QA Quality Assurance 8, 9

REST Representational State Transfer 29, 45

SaaS Software as a Service 14, 19, 22, 31, 53

SDLC Software Development Life Cycle 2, 3, 5–11, 19, 34

SQL Structured Query Language 31, 71

SSH Secure Shell 29

SSR Server Side Rendering 41, 51

UC Use Case 37

URL Uniform Resource Locator 33, 43, 45, 50, 51, 62, 71, 91

WWW World Wide Web 32

XML Extensible Markup Language 63

YAML YAML Ain't Markup Language 14, 23, 65, 74

Literatura

[1] Beetz, F. – Kammer, A. – Harrer, D. S. *GitOps: Cloud-native Continuous Deployment*. InnoQ Deutschland GmbH, 2021. ISBN 978-3982112688.

[2] Bierman, G. – Abadi, M. – Torgersen, M. Understanding TypeScript. In *ECOOP 2014 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. s. 257–281. doi: 10.1007/978-3-662-44202-9_11. Dostupné z: http://link.springer.com/10.1007/978-3-662-44202-9_11. ISBN 978-3-662-44201-2.

[3] Sufryan Uzayr. *Getting the Most out of Node.js Frameworks: The Essential Tools and Libraries*. CRC Press, 2022 edition. ISBN 1032067535.

[4] Dora, S. K. – Dubey, P. Software development life cycle (SDLC) analytical comparison and survey on traditional and agile methodology. *ABHINAV National monthly refereed journal of research in science and technology*. 2013, s. 22–30.

[5] Evertse, J. *Mastering GitLab 12*. Packt Publishing, 2019. Dostupné z: https://www.packtpub.com/free-ebook/mastering-gitlab-12/9781789531282. ISBN 9781789531282.

[6] Half, R. *6 Basic SDLC Methodologies: Which One Is Best?* [online]. [cit. 2023-03-01]. Dostupné z: https://www.roberthalf.com/blog/salaries-and-skills/6-basic-sdlc-methodologies-which-one-is-best.

[7] Haverbeke, M. *Eloquent JavaScript*. No Starch Press, 3rd edition edition, 2018. ISBN 9781593279509.

[8] Henschel, J. A comparison study of managed CI/CD solutions. 2020. Dostupné z: https://git.cubieserver.de/jh/cs-e4000-seminar/raw/commit/a9b2cba6ec2c187f7a9b34c1f4a9ab8dc5097f71/cs-seminar_2020-04-11.pdf.

[9] Hughes, K. *Static analysis in JavaScript* [online]. [cit. 2023-04-04]. Dostupné z: https://blog.logrocket.com/static-analysis-in-javascript-11-tools-to-help-you-catch-errors-before-users-do/.

[10] Laurila, S. Comparison of JavaScript Bundlers, 2020. Dostupné z: http://www.theseus.fi/handle/10024/345959.

[11] Ortega, R. *Best CI/CD Tools for DevOps* [online]. [cit. 2023-03-08]. Dostupné z: https://bluelight.co/blog/best-ci-cd-tools.

[12] S, S. A Study of Software Development Life Cycle Process Models. *SSRN Electronic Journal*. 2017. ISSN 1556-5068. doi: 10.2139/ssrn.2988291. Dostupné z: http://www.ssrn.com/abstract=2988291.

[13] Sharma, S. *DevOps For Dummies*. John Wiley & Sons, Inc., ibm limited edition edition, 2014. ISBN 978-1-118-73378-3.

[14] Sijbrandij, S. *History of GitLab* [online]. [cit. 2023-03-11]. Dostupné z: https://about.gitlab.com/company/history/.

[15] Soni, M. *DevOps for Web Development*. Packt Publishing, 2016. ISBN 978-1786465702.

[16] Spinellis, D. Git. *IEEE Software*. 2012, 29, 3, s. 100–101. ISSN 0740-7459. doi: 10.1109/MS.2012.61. Dostupné z: http://ieeexplore.ieee.org/document/6188603/.

[17] Velimirovic, A. *What is SDLC? Understand the Software Development Life Cycle* [online]. [cit. 2023-04-19]. Dostupné z: https://phoenixnap.com/blog/software-development-life-cycle.

[18] Wirfs-Brock, A. – Eich, B. JavaScript. *Proceedings of the ACM on Programming Languages*. 2020-06-14, 4, HOPL, s. 1–189. ISSN 2475-1421. doi: 10.1145/3386327. Dostupné z: https://dl.acm.org/doi/10.1145/3386327.

[19] *11 Node.JS Bundler and Build Tools to Know as JS Developer* [online]. [cit. 2023-03-31]. Dostupné z: https://geekflare.com/node-js-bundler-and-build-tools/.

[20] *What is DevOps?* [online]. [cit. 2023-03-08]. Dostupné z: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-devops/.

[21] *Unify the entire DevOps lifecycle with GitLab* [online]. [cit. 2023-03-11]. Dostupné z: https://about.gitlab.com/stages-devops-lifecycle/.

[22] *A guide to getting started in DevOps* [online]. GitLab, 2022. [cit. 2023-03-02]. Dostupné z: https://page.gitlab.com/resources-ebook-beginners-guide-devops.

[23] *4 Must-know DevOps principles* [online]. [cit. 2023-03-02]. Dostupné z: https://about.gitlab.com/blog/2022/02/11/4-must-know-devops-principles/.

[24] *ECMA-262* [online]. [cit. 2023-03-30]. Dostupné z: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[25] *GitLab Value Stream Analytics* [online]. [cit. 2023-03-12]. Dostupné z: https://about.gitlab.com/stages-devops-lifecycle/value-stream-analytics/.

[26] *GitLab architecture overview* [online]. [cit. 2023-03-23]. Dostupné z: https://docs.gitlab.com/ee/development/architecture.html.

[27] *Agile Planning* [online]. [cit. 2023-03-12]. Dostupné z: https://about.gitlab.com/solutions/agile-delivery/.

[28] *Gitlab Pricing* [online]. [cit. 2023-03-12]. Dostupné z: https://about.gitlab.com/pricing/.

[29] *Package Registry* [online]. [cit. 2023-03-12]. Dostupné z: https://docs.gitlab.com/ee/user/packages/package_registry/index.html.

[30] *GitLab Security and Governance* [online]. [cit. 2023-03-12]. Dostupné z: https://about.gitlab.com/solutions/dev-sec-ops/.

[31] *Comparison of GitLab self-managed with GitLab SaaS* [online]. [cit. 2023-03-22]. Dostupné z: https://docs.gitlab.com/ee/install/migrate/compare_sm_to_saas.html.

[32] *A beginner's guide to GitOps and how it works* [online]. Gitlab, 2022. [cit. 2023-03-02]. Dostupné z: https://page.gitlab.com/resources-ebook-beginner-guide-gitops.

[33] *SDLC (Software Development Life Cycle) Phases, Process, Models* [online]. [cit. 2023-02-27]. Dostupné z: https://www.softwaretestinghelp.com/software-development-life-cycle-sdlc/.

[34] *2022 Developer Survey* [online]. [cit. 2023-03-30]. Dostupné z: https://survey.stackoverflow.co/2022.

[35] *State of JS 2022* [online]. [cit. 2023-03-31]. Dostupné z: https://2022.stateofjs.com/.

[36] Zaidman, V. *An Overview of JavaScript Testing in 2022* [online]. [cit. 2023-03-31]. Dostupné z: https://medium.com/welldone-software/an-overview-of-javascript-testing-7ce7298b9870.

A Uživatelská dokumentace

A.1 Instalace a spuštění aplikace

Nejjednodušším způsob jak nainstalovat a spustit aplikaci je pomocí Dockeru a nástroje Docker Compose. Je připraven soubor *docker-compose.yml*, který obsahuje konfiguraci pro spuštění multi-kontejnerové aplikace a ve složkách *backend* a *frontend* jsou připraveny soubory *Dockerfile* pro sestavení kontejnerů aplikací. Spolu s backendem a frontendem je spuštěna i databáze PostgreSQL. Všechny parametry konfigurace jsou uloženy v souboru *.env*. Aplikace je spuštěna příkazem `docker-compose up`. Frontend je poté dostupný na portu 3000 a backend na portu 4000.

Druhou možností je spustit obě aplikace samostatně. Je potřeba mít nainstalované:

- **Node.js** - testováno na verzi 16
- **PostgreSQL** - parametry databáze je potřeba nastavit v souboru *backend/.env*
- **Yarn**

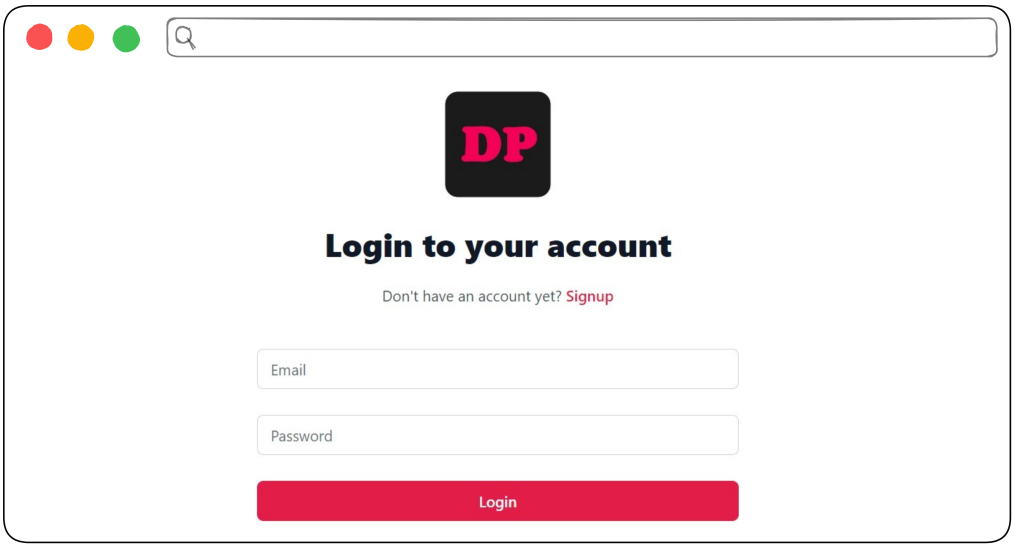
Z adresáře backendu jsou nainstalované závislosti příkazem `yarn install`, backend je sestaven příkazem `yarn build` a spuštěn příkazem `yarn start`.

Pro spuštění frontendu je potřeba nastavit parametry v souboru *frontend/.env* a to hlavně URL adresu backendu, jehož port se nastavuje v souboru *backend/.env*. Závislosti frontendu jsou nainstalovány příkazem `yarn install`, je sestaven příkazem `yarn build` a spuštěn příkazem `yarn start`.

A.2 Ovládání aplikace

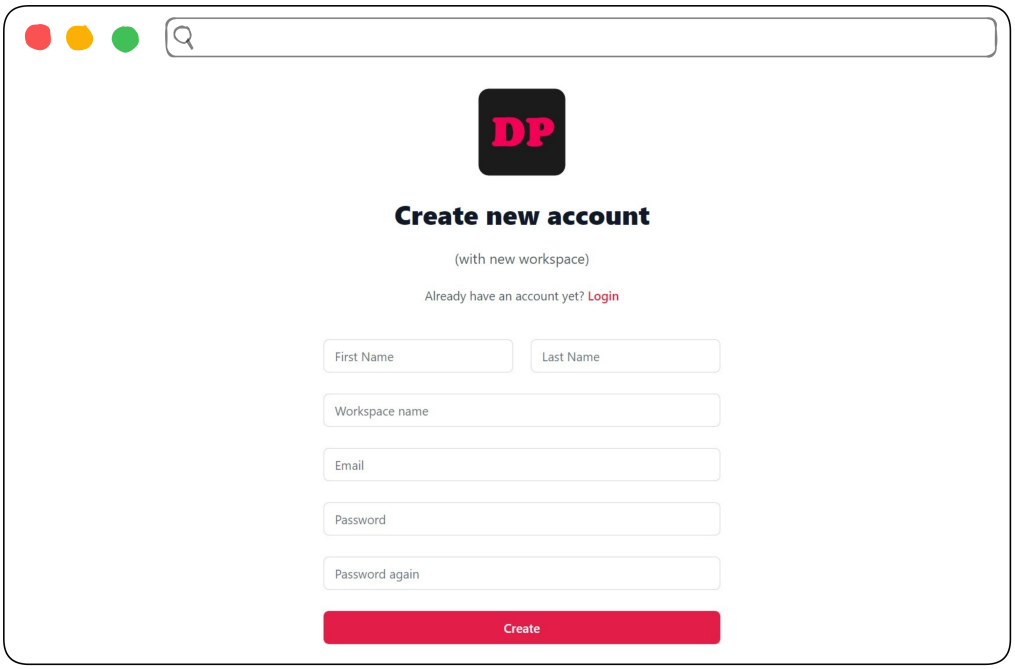
Přihlášení a registrace

Při příchodu nepřihlášeného uživatele na web je automaticky přesměrován na přihlašovací formulář viz obrázek A.1. Uživatel zadá své přihlašovací údaje a pokud jsou správné, je přesměrován na nástěnku viz obrázek A.3.



Obrázek A.1: Přihlašovací formulář

Po kliknutí na odkaz *Sign up* je uživatel přesměrován na registrační formulář viz obrázek A.2. V registračním formuláři (obrázek A.2) uživatel zadá své údaje a musí zadat i název svého defaultního pracovního prostředí, ve kterém je vlastníkem a které bude vytvořeno při registraci.



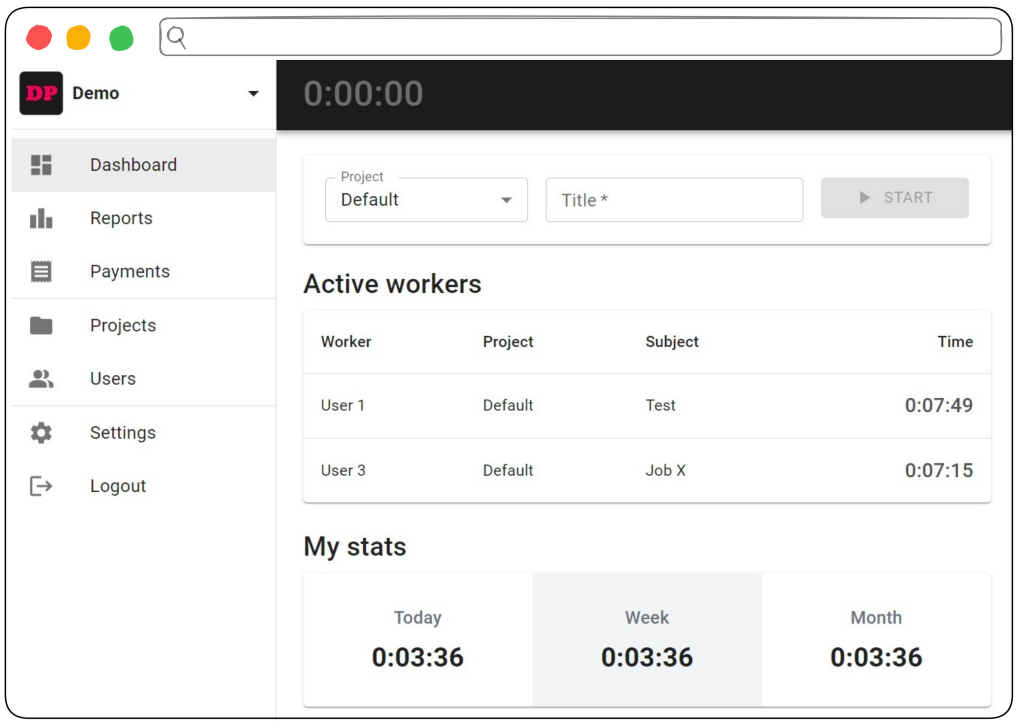
Obrázek A.2: Registrační formulář

Nástěnka

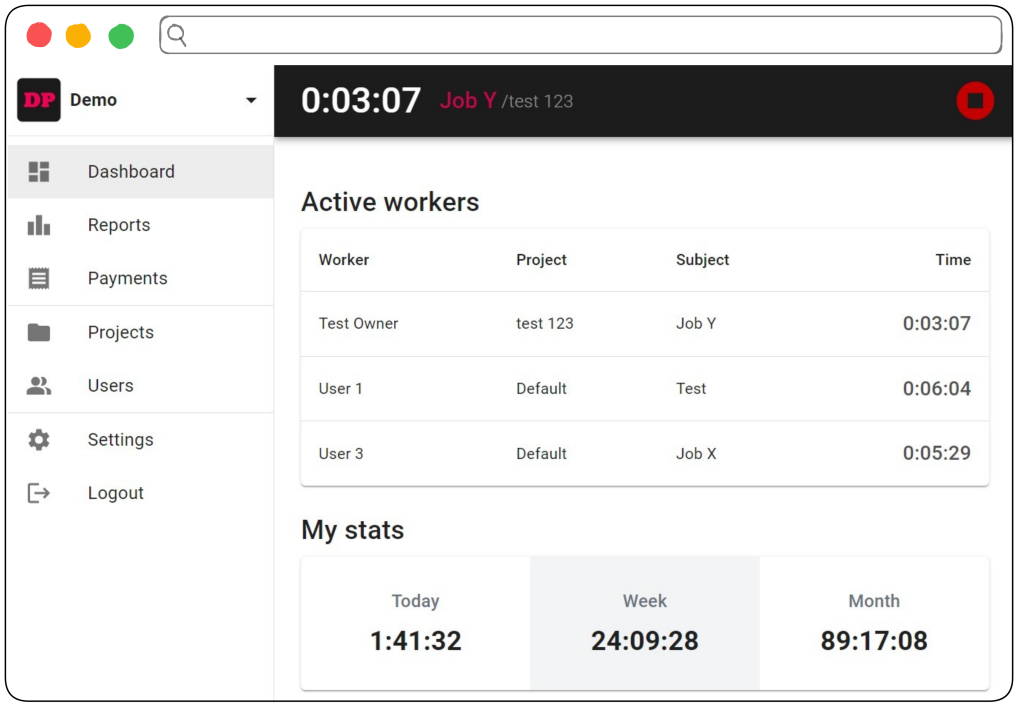
Po přihlášení se uživatel dostane do aplikace. Na levé straně je vždy menu, které obsahuje odkazy na jednotlivé části aplikace. Na obrázcích v této dokumentaci je menu pro uživatele s rolí *vlastník*. Pro ostatní role budou nepřístupné položky v menu skryté. V levém horním rohu je možné přepnout pracovní prostředí. Na obrázcích je vidět, že uživatel je v pracovním prostředí s názvem *Demo*.

Na nástěnce (Dashboard, viz obrázek A.3) je zobrazen seznam běžících měření ostatních uživatelů z pracovního prostředí (toto vidí jen role *manažer* a výše) a statistiky přihlášeného uživatele. V horní části je pomožné vybrat projekt, zadat popisek měření a spustit měření kliknutí na tlačítko *Start*.

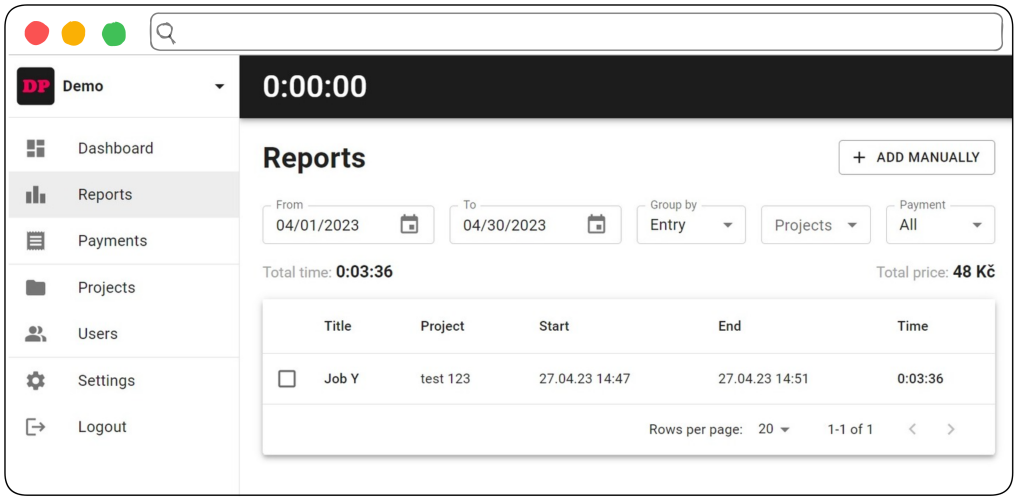
Pokud měření probíhá, je vidět aktuální stav měření v horní části aplikace (a to ze všech jejích stránek) a červené tlačítko (v pravém horním rohu) pro zastavení měření viz obrázek A.4.



Obrázek A.3: Nástěnka



Obrázek A.4: Nástěnka se spuštěným měřením

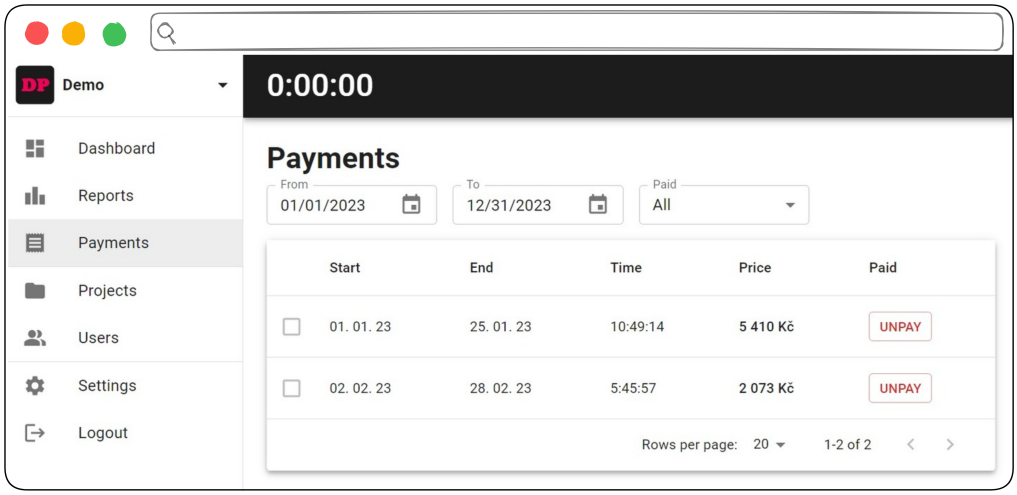


Obrázek A.5: Reporty

Reporty

V sekci *Reporty* (viz obrázek A.5) jsou zobrazeny reporty odpracovaného času přihlášeného uživatele. Záznamy lze filtrovat dle data, projektu a stavu platby. Záznamy lze slučovat (filtr *Group by*) dle dne, týdne a měsíce. Přímo z tabulky lze jednotlivé záznamy mazat a upravovat dvojitým kliknutím na

dané políčko. Kliknutím na tlačítko *ADD MANUALLY* je možné přidat záznam ručně.



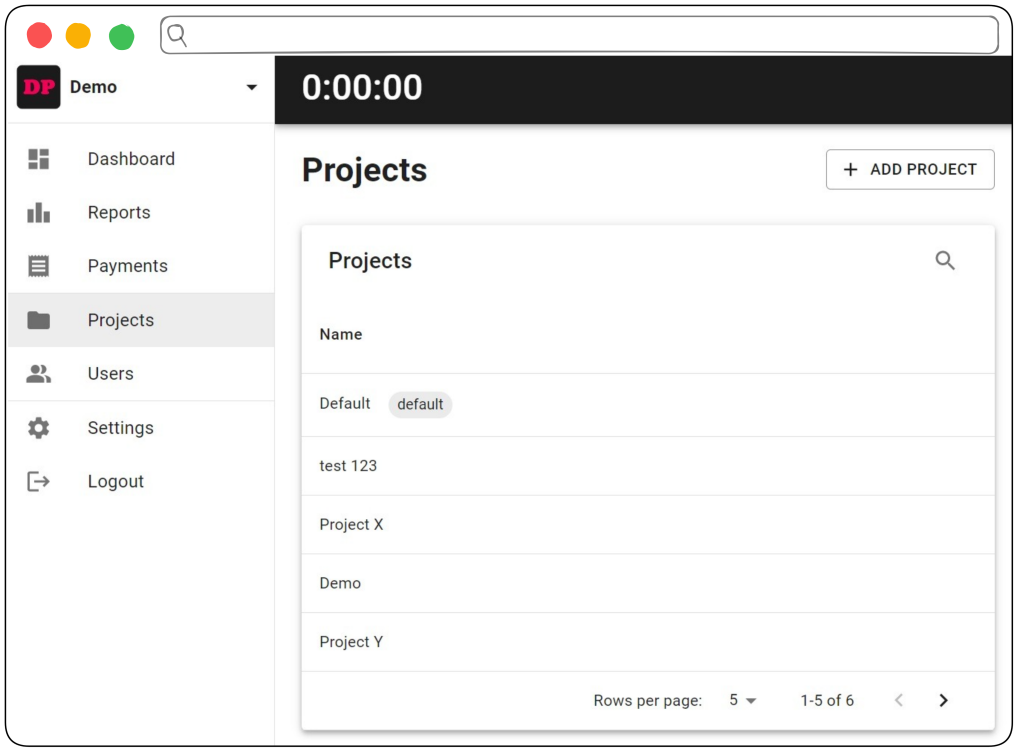
Obrázek A.6: Platby

Platby

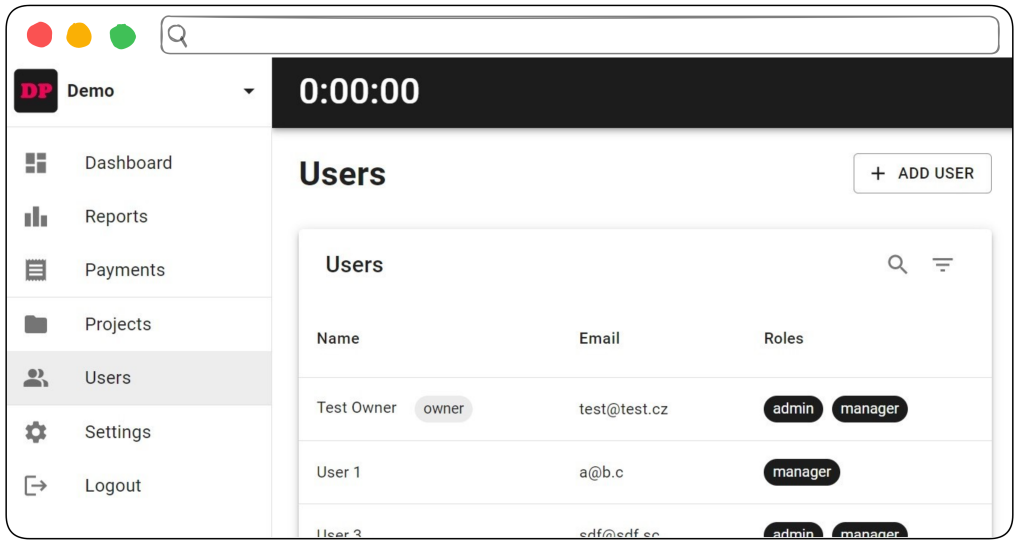
V sekci *Platby* (viz obrázek A.6) jsou zobrazeny požadavky na platby přihlášeného uživatele. Nový požadavek na platbu lze vytvořit v sekci reporty, při výběru pouze nezaplacených položek se zobrazí tlačítko pro vytvoření požadavku. Pouze uživatelé s rolí *manažer* a výše mohou požadavky označovat jako zaplacené nebo nezaplacené. Uživatel může požadavek smazat pouze pokud ještě nebyl zaplacený.

Projekty

V sekci *Projekty* (viz obrázek A.7) je možné spravovat projekty pracovního prostředí. Do této části mají přístup pouze uživatelé s rolí *manažer* a výše. Je možné měnit název projektů, mazat je a vytvářet nové.



Obrázek A.7: Projekty



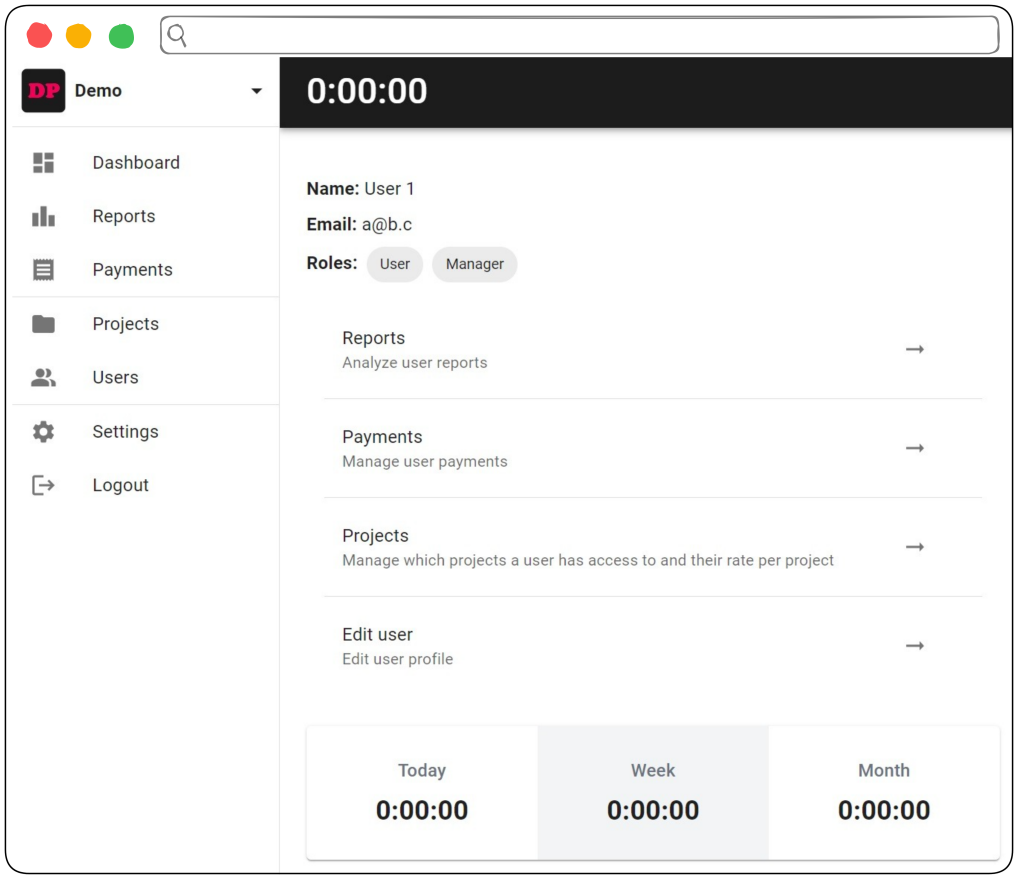
Obrázek A.8: Seznam uživatelů

Správa uživatelů

V sekci *Uživatelé* (viz obrázek A.8) je možné spravovat uživatele pracovního prostředí. Do této části mají přístup pouze uživatelé s rolí *manažer* a výše

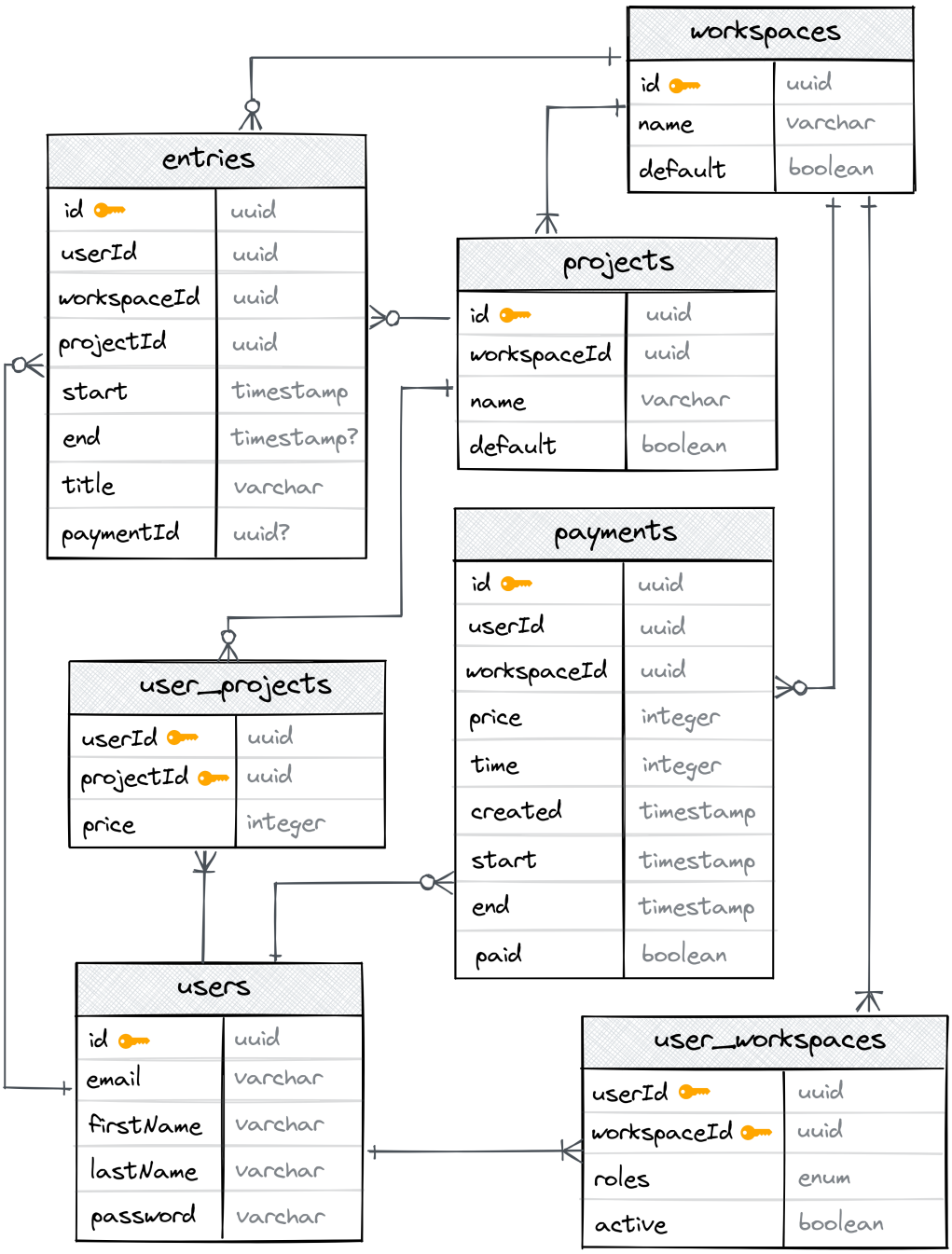
avšak přidávat, upravovat a mazat uživatele může pouze uživatel s rolí *admin* nebo *vlastník*.

Po kliknutí na uživatele se zobrazí detail uživatele (viz obrázek A.9). Odtud je možné přejít na reporty a platby daného uživatele, spravovat ke kterým projektům má uživatel přístup a jakou hodinovou sazbu na daném projektu má.

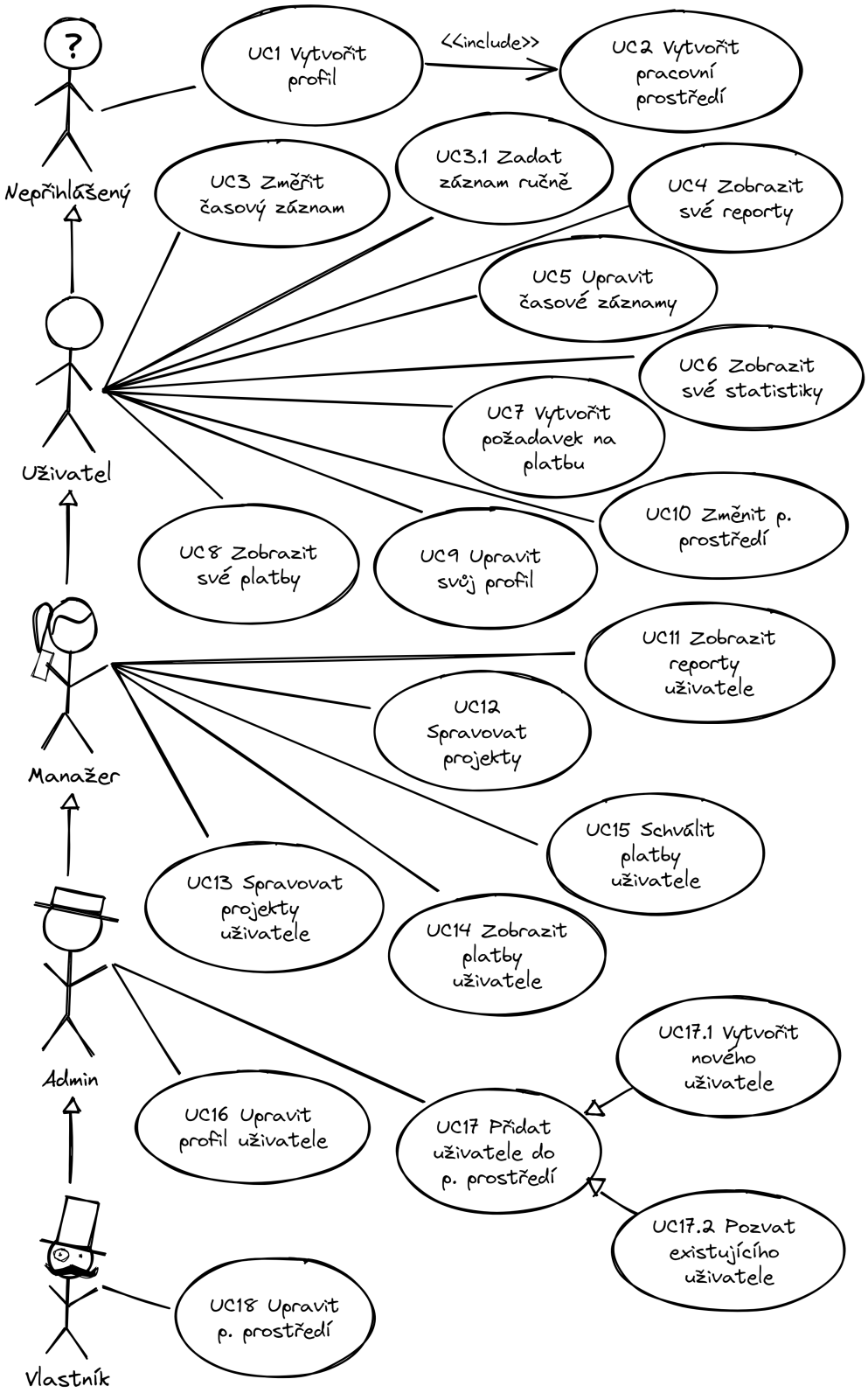


Obrázek A.9: Detail uživatele

B Přílohy



Obrázek B.1: ER model databáze



Obrázek B.2: Diagram případů užití

C Elektronické přílohy

Adresářová struktura dodaných elektronických příloh je následující:

- **Text_prace** - obsahuje všechny "zdrojové" soubory, tj. .tex, .png apod. a výsledný PDF soubor této práce (Forejt_Martin_DP_2023.pdf)
- **Poster** - soubory posteru ve formátech .pub a .pdf
- **Aplikace_a_knihovny**
  - **backend** - zdrojové kódy aplikace backendu
  - **frontend** - zdrojové kódy aplikace frontendu
  - **.gitlab-ci.yml** - konfigurační soubor CI/CD pipeline
  - **docker-compose.yml** - konfigurace pro spuštění multi-kontejnerové aplikace pomocí Dockeru
  - **README.md** - popis adresáře a návod na spuštění aplikace
- **Readme.txt** - obsah této přílohy