VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Online nástroj pro analýzu softwarových projektů

Online Tool for Software Projects Analysis

2019

Martin Hruška

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Hruška**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Online nástroj pro analýzu softwarových projektů
Online Tool for Software Projects Analysis

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je navrhnout a vytvořit nástroj, který umožní definovat a provádět analýzy nad projekty, které jsou hostovány v systému GitHub nebo v nějaké instanci aplikace GitLab. K analýzám bude využita aplikace SonarQube a proces bude maximálně automatizovaný. Nástroj bude navržen tak, že bude bude dostupný „on-line" prostřednictvím webového prohlížeče a bude umožňovat přidávání možných analýz formou zásuvných modulů.

1. Proveďte analýzu informací poskytovaných z aplikace GitHub a GitLab, které hostují projekty. Prozkoumejte API GitHub-u a aplikace GitLab za účelem získání těchto informací.
2. Vytvořte architektonický návrh vytvářeného nástroje. Snažte se o použití technologií založených na HTML5.
3. Proveďte implementaci nástroje spolu s vybranými zásuvnými moduly a proveďte testování.
4. Soustřeďte se na znovupoužití funkcí použitých v jednotlivých analýzách. Funkce budou nabízet zásuvné moduly. Analýzy budou definovány s využitím skriptovacího či vizuálního jazyka.
5. Všechny zdrojové kódy či skripty budou uloženy v repozitáři aplikace, která je dostupná na adrese http://git.cs.vsb.cz. Vytvořené aplikace bude možno sestavit podobným způsobem jako nabízí systém maven.

Práce bude zejména obsahovat:
1. Přehled informací, které poskytují aplikace GitHub a GitLab, a API, které je dostupné pro jejich získání.
2. Model návrhu architektury.
3. Zdrojové kódy nástroje spolu s vybranými zásuvnými moduly.
4. Skripty pro automatické testování či popis testovacích scénářů.
5. Vyhodnocení dosažených výsledků a naznačení dalšího rozvoje.

Seznam doporučené odborné literatury:

[1] CAMPBELL, G. Ann a Patroklos P. PAPAPETROU, 2013. SonarQube in Action. 1 edition. Shelter Island, New York: Manning Publications. ISBN 978-1-61729-095-4.
[2] DAVIS, Christopher W. H., 2015. Agile Metrics in Action: Measuring and Enhancing the Performance of Agile Teams. 1 edition. Shelter Island, NY: Manning Publications. ISBN 978-1-61729-248-4.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Kožusznik, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2019

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 23. dubna 2019

. . . .

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 23. dubna 2019

. . . .

Rád bych poděkoval své rodině za trpělivost při psaní této diplomové práce a také svému vedoucímu práce za odborné konzultace, které byly velmi velkým přínosem.

Abstrakt

Cílem této diplomové práce je navržení a vytvoření webové aplikace, která umožňuje definování a provádění analýz nad projekty, které jsou hostovány v GitHubu nebo GitLabu a pro analýzu je používaná aplikace SonarQube, kde celý tento proces je maximálně automatizovaný. Je provedena analýza stávajících řešení v oblasti verzovacích systému a také analýza API GitHubu a GitLabu pro vytvoření unifikovaného rozhraní, pomocí kterého aplikace komunikuje s těmito servery. Pro vytváření dotazů, které jsou posílány na verzovací systémy vzniklo jednoduché uživatelské rozhraní, kde tyto dotazy jsou tvořeny pomocí přetahovatelných bloků, které lze dle potřeby upravovat a vytvářet nové. Součástí této práce je také zhodnocení výsledků provedených analýz a naznačení dalšího možného rozvoje.

Klíčová slova: SonarQube, analýza, GitHub, GitLab, API, webová aplikace, automatizace, diplomová práce

Abstract

Purpose of this diploma thesis is to design and develop web application that allows to define and perform analysis on projects which are hosted in GitHub or GitLab and SonarQube application is used for analysis where the whole process is maximally automated. Analysis is performed on existing solutions in the field of version control systems as well as GitHub and GitLab API analysis to create unified interface through which application communicate with these servers. For building queries that are sent to version control systems, a simple user interface has been created, where these queries are created by draggable blocks that can be edited or created as needed. Part of this work is also an evaluation of the results of performed analyzes and an indiccation of further possible development.

Key Words: SonarQube, analysis, GitHub, GitLab, API, web application, automatization, master thesis

Obsah

Seznam použitých zkratek a symbolů 15

Seznam obrázků 17

1 Úvod 19

2 Analýza současných řešení v oblasti automatizace testů 21

3 Průzkum existujících řešení v oblasti verzovacích systémů 23
- 3.1 Distribuované verzovací systémy . . . 23
- 3.2 Git . . . 23
- 3.3 AWS CodeCommit . . . 24
- 3.4 Centralizované verzovací systémy . . . 24

4 Analýza GitHub a Gitlab 27
- 4.1 Možnosti využití verzovacích nástrojů . . . 27
- 4.2 Pojmy v Gitu . . . 27
- 4.3 Statistiky . . . 28
- 4.4 Analýza API GitHubu a GitLabu . . . 29

5 Použité technologie 33
- 5.1 Serverová část . . . 33
- 5.2 Klienstká část . . . 33

6 SonarQube 37
- 6.1 Co je SonarQube . . . 37
- 6.2 Skenery . . . 37

7 Serverová část 41
- 7.1 Autofac - Dependency Injection . . . 41
- 7.2 REST (Representational State Transfer) . . . 42
- 7.3 Datový model . . . 42
- 7.4 Architektura . . . 43
- 7.5 Komunikace s GitHub a GitLab . . . 46
- 7.6 Vrstvy aplikace . . . 46
- 7.7 Aktivitní diagram průběhu analýzy . . . 48

8 Klientská část 51
8.1 Výhoda využití TypeScriptu 51
8.2 Komunikace s aplikací SonarQube 52
8.3 Jádro klientské aplikace 53
8.4 Bázové třídy 54
8.5 Views 54
8.6 Routování 55
8.7 Skládání dotazů 56
8.8 Dostupné moduly 56

9 Úspěšnost analýzy 61
9.1 JavaScript 61
9.2 Python 61
9.3 Ruby 62
9.4 TypeScript 62
9.5 Java 63
9.6 Shrnutí 63

10 Závěr 65

Literatura 67

11 Prerekvizity 69
11.1 .NET Core 69
11.2 .NET Framework 69
11.3 Java 69

Seznam použitých zkratek a symbolů

JSON	–	JavaScript Object Notation
API	–	Application Programming Interface
REST	–	Representational State Transfer
HTML	–	Hyper Text Markup Language
UoW	–	Unit of Work
UI	–	User Interface
MVC	–	Model View Controller
End-point	–	Koncový bod komunikace

Seznam obrázků

1 Zastoupení programovacích jazyků - [13] 28
2 Uživatelské rozhraní aplikace SonarQube 37
3 Dostupné pluginy pro SonarQube - [10] 39
4 Databázový model 43
5 Třídní diagram serverové části 44
6 Unit of Work pattern 45
7 Diagram komunikace se servery GitHubu a GitLabu 46
8 Aktivitní diagram průběhu analýzy 49
9 4 + 1 Kruchten diagram 51
10 Ukázka UI 52
11 Komunikace se serverem SonarQube 53
12 Zobrazení dostupných projektů 58
13 Třídní diagram klientské vrstvy 59
14 Webpack analýza 61
15 AngularJS analýza 61
16 Django analýza 62
17 Django rest framework analýza 62
18 Analýza projektu napsaném v Ruby 62
19 Angular analýza 63

1 Úvod

Analýza a samotné testování je v dnešní době nedílnou součástí vývojového cyklu aplikací, která přímo ovlivňuje celkovou kvalitu výstupu.

V případě špatné analýzy, absence testování, popřípadě špatné revize kvality kódu je výsledná kvalita softwaru velmi často na špatné úrovni a životnost vzniklého produktu může skončit již při předání klientovi.

U takto špatně navržených projektů je následný vývoj značně komplikovaný a samotný kód se stává nečitelným a špatně udržitelným a nějaké větší zásahy do struktury aplikace mohou nabourat funkcionalitu ostatních komponent.

Hlavní myšlenkou této diplomové práce je vytvoření online nástroje pro analýzu projektů, které jsou uloženy na službách GitHub a GitLab. Součástí je i vytvoření průzkumu již existujících služeb podobných GitHubu a GitLabu a jejich následné srovnání.

Součástí třetí kapitoly je přehled existujících verzovacích systémů a jejich srovnání, kde v následující kapitole je provedena analýza API GitHubu a GitLabu, jsou definovány základní pojmy při práci s repozitáři a také se blíže podíváme na současný stav četnosti projektů napsaných v konkrétním programovacím jazyce, na základě kterých se pak odvíjí závěrečné testování, kdy jsou vybrány první čtyři programovací jazyky dle četnosti výskytu repozitářu na GitHubu a pro přímé srovnání s nejčetnějším jazykem JavaScript je jako poslední jazyk vybrán TypeScript, který je nádstavbou JavaScriptu. Výsledky testování jsou detailněji probrány v kapitole 9.

V 5. kapitole jsou specifikovány použité technologie jak pro klientskou, tak i pro serverovou část, kde následující 2 kapitoly se zabývájí již konkrétní implementaci jak serverové, tak i klientské části a jsou také podrobně popsány architektury obou části.

V kapitole zabývající se klientskou vrstvou se detailněji podíváme také na získávání dat z těchto služeb, které je implementováno formou grafického rozhraní, kde uživatel vytváří dotazy na tyto služby skrze grafické bloky, které lze přesouvat do vyznačené části v uživatelském rozhraní a díky tomu lze velmi snadno vytvořit výsledný požadavek na data. Aplikace umožňuje vytváření dotazů pomoci relativně nového jazyka GraphQL.

Celý koncept aplikace je postaven na myšlence co nejsnadnějšího rozšiřování funkcionality, kdy jednotlivé části aplikace musí fungovat jako samostatné moduly, které využívají vystavené API, skrze které proudí veškerá komunikace.

Pro tuto potřebu bylo vytvořeno například unifikované rozhraní pro dotazy na služby GitHubu a GitLabu, kde veškeré požadavky na tyto služby budou putovat skrze jeden end-point a server se již postará o zpracování tohoto požadavku a následné přeformátování získaných dat z těchto repozitářů.

2 Analýza současných řešení v oblasti automatizace testů

Součástí této kapitoly je průzkum existujících řešení, které se zabývají problematikou automatizovaného testování.

Při analyzování této problematiky nebylo nalezeno obdobné řešení, které by odpovídalo celkovému konceptu této prace, kde by uživatel mohl provádět analýzy libovolných projektů nacházejících se na službě GitHub, popřípadě na jiné službě zabývající se správou zdrojových kódů. Všechna dostupná řešení v této oblasti jsou spjata s integrací aplikace do vyvíjeného softwaru, kde je potřeba provést konfiguraci. Jako příklad lze uvést integraci SonarQube s TFS, kdy je nutné předem připravit konfigurace pro různé scénáře užití. Taková konfigurace se vztahuje pouze k jednomu projektu, kterou ne vždy lze přenést na jiný vyvíjený software.

V rámci analýzy této problematiky jsem však narazil na existující řešení v oblasti získávání dat ze služby GitHub, které bylo sepsáno v jazyce JavaScript a knihovně React. Přestože se toto řešení zabývá alespoň z části touto problematikou, není tato knihovna využita ve výsledné práci, jelikož daná knihovna nevyhovovala některým požadavkům, mezi které může patřit například požadavek pro tvorbu dotazů na danou službu pomocí grafického rozhraní a dále by použití této knihovny vedlo k obtížnější integraci a unifikaci funkcí a komponent.

Tuto práci lze tedy z širšího pohledu na základě těchto skutečností považovat za tzv.**Proof of Concept**, kde je kladen důraz na co největší automatizaci testování projektů, kde uživatel je schopen provést analýzu libovolného softwaru hostovaných na službách GitHubu a GitLabu pomocí jednoduchého uživatelského rozhraní.

3 Průzkum existujících řešení v oblasti verzovacích systémů

Součástí této kapitoly je provedena analýza současných řešení v oblasti verzovacích systémů, které jsou rozděleny do dvou základních typů.

3.1 Distribuované verzovací systémy

Jedná se o systém, kdy se vytvoří kopie celého repozitáře na lokální disk a ne pouze binární snímek repozitáře a díky této vlastnosti jsme schopní pracovat bez nutnosti přístupu k internetu a všechny propagované změny kódu jsou prvně uložený do lokálního repozitáře a až při provedení operace push jsou veškeré změny odeslány na server.

3.2 Git

Git je distribuovaný systém, který je primárně určený pro správu a sledování změn ve zdrojovém kódu během celého vývoje softwaru. Git byl vytvořen v roce 2005 jako podpůrný nástroj při vývoji Linux kernelu, díky kterému si vývojáři mohli sdílet svůj kód.

Výhodou Gitu je, že vytváří kompletní kopie jednotlivých repozitářů a volba, který z nich je hlavní je vždy na uživateli. Participující uživatelé na projektu mají k repozitářům přístup a současně s tím mají práva na zápis do nich.

V praxi to vypadá tak, že vývojář si stáhne určitou verzi vyvíjeného softwaru, kde tato verze se naklonuje na lokální disk a vznikne tak lokální repozitář, ve kterém se provádějí veškeré změny nad zdrojovým kódem a současně s tím je možné, aby uživatel měl takových repozitářů více.

V průběhu vývoje softwaru se v repozitáři vytvářejí větve, kde se provádí implementace nových funkcí nebo například také úpravy již existujícího kódu a díky tomu je repozitář reprezentovaný stromovou strukturou [2].

3.2.1 GitHub

GitHub je aplikace poskytující webové služby, která byla spuštěna v roce 2008, jejichž aktuálním vlastníkem je Microsoft. Jedná se o verzovací nástroj podporující vývoj softwaru, který je v dnešní době rozsáhle používaný pro hostování jak open source projektů, tak i pro soukromé účely, kdy pro vytvoření soukromého repozitáře pro hostování projektu již není potřeba mít zaplacen měsíční poplatek od ledna 2019.

3.2.2 GitLab

Stejně jako GitHub, tak i GitLab je webová služba pro správu a verzování kódu aplikací, která byla spuštěna v roce 2011. GitLab oproti GitHubu umožňuje využití jejich softwaru na serverech třetích stran.

3.3 AWS CodeCommit

Jedná se o službu hostovanou společností Amazon, který lze využít pro vytvoření privátních repozitářů, kde lze ukládat dokumenty, zdrojové kody, nebo také binární soubory v cloudu. Jedná se o velmi bezpečnou službu, ve které lze hostovat také privátní Git repozitáře.

Tato služba eliminuje potřebu správy verzovacího systému a spolu s tím odpadá také nutnost řešení škálovatelnosti infrastruktury v případě potřeby.

AWS je postavená na bázi Git repozitářů a tedy poskytuje veškeré příkazy, které nám Git poskytuje. Tyto příkazy budou podrobněji vysvětleny v následující kapitole.

3.4 Centralizované verzovací systémy

Typickým příkladem centralizovaných verzovacích systémů je TFVC od Microsoftu. Při požadavku na získání aktuálního kódu ze vzdáleného repozitáře obdržíme pouze binární snímek repozitáře, neboli "snapshot", který je uložen na centrálním repozitáři a historie je takto sledována na serveru. Při každé změně kódu která je odeslaná na server, neboli check-in, takto propagujeme změny pro všechny uživatele.

Problém nastává, kdy dva uživatelé chtějí měnit stejnou část kódu. Při změně je odeslána na server informace o prováděné změně a ostatním uživatelům je tak blokován check-in do doby, než vývojář, který provedl změnu jako první neprovede svůj check-in. Tato vlastnost pak vede k vytvoření větších check-inů a snižuje se možnost vrácení změn zpátky v případě, že tato možnost bude nutností, jelikož hrozí, že dojde ke ztrátě některých změn, o které bychom nechtěli přijít.

U centralizovaných systému všichni uživatelé pracují na stejné větvy a nové větve se používají pouze při vytváření nové funkcionality a oproti distribuovaným systémům nelze pracovat bez přístupu k internetu.

3.4.1 Team Foundation Service

Jedná se o "all-in-one"řešení od Microsoftu, které obsahuje funkcionality pro Agilní projektové řízení, verzovací systém, průběžné integrace a release management. Součástí TFS je Team Foundation Version Control, který je jedním ze dvou základních možností pro správu zdrojových kódů. TFVC patří do skupiny centralizovaných verzovacích systémů. Druhou možností je již zmíněný Git, kde jeho podpora byla přidána v roce 2013.

3.4.2 Subversion

Subversion, také znám pod zkratkou SVN je volně dostupný verzovací systém založený společností CollabNet v roce 2000. Jedná se o velmi populární verzovací systém na trhu.

Větve jsou vytvářeny jako adresáře uvnitř repozitáře a díky této vlastnosti vznikají stromové konflikty, které jsou způsobeny změnami uvnitř stromové struktury. K těmto konfliktům dochází

velmi často a při vzniku těchto konfliktů SVN neumožní provést commit, dokud všechny konflikty nejsou vyřešeny.

3.4.3 CVS

Dalším ze zástupců centralizovaných verzovacích systému, je CVS, neboli Concurrent Versions System. Vznikl v roce 1986, avšak v dnešní době se již jedná o ne příliš používaný systém v porovnání s konkurenčními systémy.

4 Analýza GitHub a Gitlab

Pro úspěšné získávání informací ohledně projektů uložených v aplikaci GitHub a GitLab s následnou analýzou daných projektů bylo potřeba v první řadě provést analýzu jejich stávající API, pomocí kterého lze vytvářet dotazy na požadovaná data. Pomocí těchto dotazů můžeme provádět filtraci projektů na základě specifických kritérií, jakými jsou například možnosti hledání projektů, dle určitého jazyka, ve kterým byl projekt napsaný, dále provádět různé druhy třídění, například podle hodnocení jednotlivých repozitářů, oblíbenosti, autora apod.

4.1 Možnosti využití verzovacích nástrojů

Verzovací nástroje se používají převážně pro účely sdílení kódu aplikací mezi jednotlivými týmy pro zvýšení efektivity vývoje. Hlavní výhodou použití těchto nástrojů je minimalizování rizika ztráty dosažené práce a možnost dohledání veškerých změn, které byly provedeny v rámci vývoje aplikace v tzv. verzích, které se vytvářejí pomocí pull requestů. Tyto pull requesty se následně synchronizují se specifickou větví, ve které je obsažen zdrojový kód aplikace, který slouží jako výchozí bod pro všechny následující vývojové úkoly.

V rámci této větve jsou většinou zablokovány veškeré přímé změny kodů, avšak změny kódu lze do této větve dostat tak, že se vytvoří nová větev, která je odvozená z této bázové větve a následně po dokončení daného vývojového úkolu se vytvoří pull request, který po schválení provede synchronizaci obou větví a vytvoří se nová verze, neboli commit na cílové větvi [14].

4.2 Pojmy v Gitu

- **Větev** - jedná se o sadu commitů, kde vývojář vytváří nové funkcionality, popřípadě různé úpravy nad kódem nebo opravy chyb. V Repozitáři vždy bývá jedna hlavní větev, která se nazývá **Master**, kde se obvykle nachází poslední změny v kódu. Přímé úpravy v této větvi jsou zpravidla zakázány a nové změny se do této větve dostávají pouze skrze **Pull requesty**.

- **Commit** - tento pojem je reprezentován hned třemi funkcemi, který Git podporuje. Pomocí commitu lze jednoznačně identifikovat konkrétní změny v průběhu vývoje, dále tento pojem reprezentuje činnost, kdy chceme uložit nové změny do repozitáře a posledním významem je prezentace seznamu všech změn v kódu od předchozího commitu.

- **Pull request** - díky tohoto požadavku lze provést synchronizaci dvou větví, zpravidla to bývá synchronizace hlavní vývojové větvě s větví, která z ní přímo vychází. Tyto pull requesty jsou revidovány zkušenějším vývojářem, který ho buď schválí nebo zamítne.

  V případě schválení daného pull requestu se obsah větve, na které byl vytvořený pull request spojí s cílovou větví a zdrojová větev se zpravidla maže, aby nedocházelo k zbyteč-

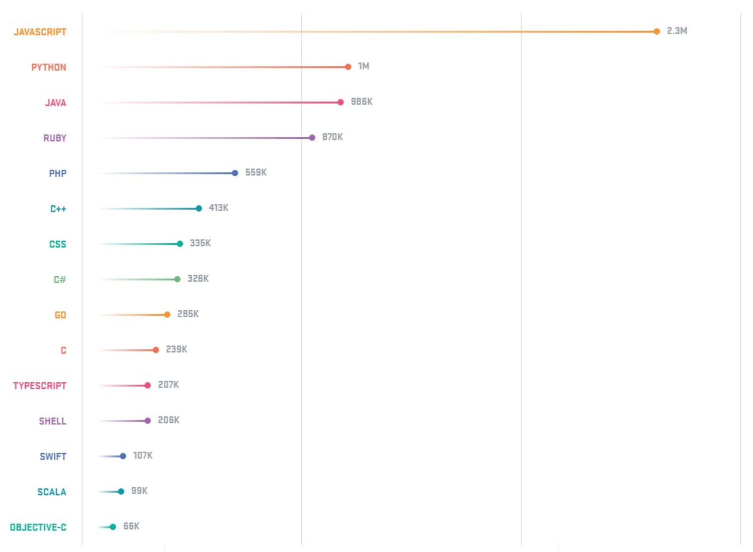
nému hromadění již nepoužívaných větví. V opačném případě vzniká zpětná vazba formou komentářů, které jsou přidávány na řádky kódu, aby bylo zřejmé, kde se má provést úprava.

- **Clone** - jedná se o vytvoření lokální kopie ze vzdáleného repozitář, kde pak můžeme libovolně upravovat zdrojové kódy.
- **Conflict** - konflikty vznikají v případě, kdy 2 vývojáři zasáhnou do stejné části kódu nebo také při přesunutí souboru na jiné místo, kdy někdo jiný v tomto souboru prováděl změny. Tyto konflikty je potřeba řešit manuálně, kdy se do větve, ve které vznikl konflikt provede příkaz **merge** z cílové větve a poté je nutné spojit vzniklé změny.
- **Merge** - díky tomuto příkazu je možné kopírovat obsah jedné větve do druhé. Při tomto příkazu dojde k synchronizaci obou větví společně se všemi provedenými **commity**.
- **Cherry-pick** - tento příkaz umožňuje také kopírovat obsah jedné větve do druhé, rozdíl oproti příkazu **merge** je ten, že nepřesouvá všechny **commity**, ale pouze jeden vybraný a tímto dokážeme ovlivnit, co přesně chceme vykopírovat.

4.3 Statistiky

Na GitHubu je od založení vytvořeno více než 67 milionů repozitářu a přes 24 milionů uživatelů, kteří spravují tyto repozitáře. Z toho jich je více jak 25 milionu aktivních. Za aktivní se počítají pouze ty, ve kterých se během jednoho roku provedla nějaká změna v kódu, popřípadě se například přidal komentář..

Dle posledních statistik z roku 2017 je patrné, že největší zastoupení mají aplikace, které jsou napsány v jazyce JavaScript. Tyto data jsou odvozena vůči vzniklým pull requestům během roku 2017.



Obrázek 1: Zastoupení programovacích jazyků - [13]

Na základě těchto statistik jsou vybrány programovací jazyky, které je aplikace schopná analyzovat, avšak za předpokladu, že aplikace SonarQube má dostupný analýzer nebo plugin pro daný jazyk a současně je ve veřejně dostupné verzi [13].

4.4 Analýza API GitHubu a GitLabu

4.4.1 GitHub API v3

Pro vyhledávání projektů hostovaných v GitHubu a které se zároveň nenachází v privátním repozitáři, je používán endpoint, */search/repositories* na který se dotazujeme pomocí **GET** requestu a pro filtrování a řazení jsou dostupné následující parametry [8].

- **q** - jedná se o povinný parametr, který je určen pro filtrování dle hledaného výrazu. Ve výchozím stavu se zadaný výraz vyhledává vždy v názvu a popisu projektu. Toto chování lze však upravit přidáním speciálního modifikátoru *in*.

  Ukázka dotazu pro vyhledávání slova **analysis** pouze v popisu projektu:

  https://api.github.com/search/repositories?q=analysis+in:description

- **sort** - volitelný parametr, pomocí kterého je možné třídit výstupní data dle námi zvolených kritérií, například podle počtu hvězdiček udaných u projektů.

  https://api.github.com/search/repositories?q=analysis+&sort=stars

- **order** - volitelný parametr, pomocí kterého můžeme řadit data od nejmenšího počtu výskytu po největší a naopak.

  https://api.github.com/search/repositories?q=analysis+&sort=stars&order=desc

Výsledný dotaz na toto API může vypadát nějak takhle.
https://api.github.com/search/repositories?q=analysis+language:csharp&sort=stars&order=desc
Výsledkem tohoto dotazu je JSON, který má následující obsah. Z důvodu množství dat přenesených pro jeden projekt je daná ukázka JSON objektu značně zredukována.

```
"items": [
{
"id": 858127,
"name": "pandas",
"full_name": "pandas-dev/pandas",
"html_url": "https://github.com/pandas-dev/pandas",
"description": "Flexible and powerful data analysis / manipulation library
    for Python, providing labeled data structures similar to R data.frame
    objects, statistical functions, and much more",
```

```
"url": "https://api.github.com/repos/pandas-dev/pandas",
"contributors_url": "https://api.github.com/repos/pandas-dev/pandas/
    contributors",
"commits_url": "https://api.github.com/repos/pandas-dev/pandas/commits{/sha}
    ",
"git_commits_url": "https://api.github.com/repos/pandas-dev/pandas/git/
    commits{/sha}",
"comments_url": "https://api.github.com/repos/pandas-dev/pandas/comments{/
    number}",
"issues_url": "https://api.github.com/repos/pandas-dev/pandas/issues{/number
    }",
"pulls_url": "https://api.github.com/repos/pandas-dev/pandas/pulls{/number}"
    ,
"pushed_at": "2019-02-17T06:36:47Z",
"homepage": "https://pandas.pydata.org",
"size": 148111,
"watchers_count": 18172,
"language": "Python",
"has_issues": true,
"open_issues": 2928,
"watchers": 18172,
"default_branch": "master",
}....]
```


Výpis 1: Formát odpovědi ze serveru GitHub

4.4.2 GitHub API v4

Tato verze API podporuje dotazování pomocí jazyka GraphQL a nabízí pouze jeden end-point, na který se posílají veškeré požadavky a tímto je zajištěna flexibilita integrace a možnost definovat požadavek tak, abychom obdrželi pouze ta data, která reálně využijeme. GitHub API ve verzi v3 toto neumožňuje.

Nespornou výhodou této varianty je fakt, že na získání jakýchkoliv dat nám stačí poslat pouze jeden požadavek, který je napsaný v dotazovacím jazyce GraphQL, kde bychom oproti předchozí verzi museli odeslat více rozdílných dotazů, abychom dosáhli stejného výsledku [8].

4.4.3 GitLab API

Obdobně, jako GitHub, tak i GitLab nabízí dvě varianty API, kdy jedno je založené na REST API a druhou variantou je API založené na dotazovacím jazyce GraphQL, avšak oproti GitHubu

je tato varianta ve velmi brzké fázi vývoje, a proto v rámci této diplomové práce na ní nebude brán zřetel. Pro využití vystavených end-pointů je nutné mít vytvořený GitLab účet a je potřeba mít vygenerovaný token, který musí být součástí každého dotazu na server, v opačném případě takový dotaz nebude zpracován a skončí autentifikační chybou [9].

5 Použité technologie

Pro vývoj této aplikace jsou použity nejmodernější technologie, které se standartně využívají při vývoji dnešních webových aplikací.

5.1 Serverová část

Tématem této sekce je souhrn technologií použitých pří vývoji webové aplikace, které jsou určené pro serverovou část.

5.1.1 .Net Core 2.2

Jedná se o open source platformu pro vývoj aplikací pro operační systémy Windows, Mac OS a Linux. Tento framework je vybrán oproti staršímu .NET Frameworku z důvodu multiplatformnímu využití a také z důvodu dlouhodobé kontinuální podpory ze strany Microsoftu.

5.1.2 Autofac

Jedná se o IoC kontejner, který se stará o vytváření jednotlivých instancí tříd dle jejich závislostí a díky této vlastnosti lze velice lehce škálovat jakoukoliv aplikaci nezávisle na její velikosti. V praxi to znamená, že pokud nám nějaká třída závisí na jiné, nemusíme vytvářet její instanci společně se všemi potřebnými parametry, ale IoC kontejner se postará o vytvoření dané instance a všech jejich závislostí za nás dle konfigurace nastavení.

5.1.3 Entity Framework Core

Entity framework je open source nástroj pro objektově relační mapování, který je určen pro .NET Core Framework. Tento nástroj řeší proces transformace relační databázové struktury na objektový model, který lze použít v objektovém programovacím jazyce. Entity framework umožňuje dva základní přístupy k mapování. Prvním přístupem je tzv. "code-first", který je také využit v této práci, kdy se manuálně vytvoří objektový model, který je poté namapován na databázi. Druhým přístupem je tzv. "database-first", který se používá na již existující databázi.

5.1.4 MSSQL

Součástí diplomové práce je také použita MSSQL databáze, která nám slouží pro uchovávání dat, pro dotazování a rovněž i pro výsledky analýz.

5.2 Klienstká část

V této sekci se podíváme na technologie, které jsou použity pří vývoji klientské části webové aplikace.

5.2.1 Node Package Manager

Samotný název napovídá, že se jedná o nástroj, který má na starosti správu JavaScriptových knihoven v projektu. Jednotlivé knihovny se definují v souboru package.json. Soubor se nachází v rootu složky projektu, který je určen pro klientskou část.

5.2.2 TypeScript

TypeScript je open source programovací jazyk, který byl vytvořen firmou Microsoft. Jedná se o nádstavbu jazyka JavaScript, který je rozšířením standardu ECMAscript 5. TypeScript nám umožňuje psát objektově a díky této vlastnosti máme dobrou typovou kontrolu, kterou známe z objektově orientovaných programovacích jazyků, jakými jsou např. Java, C#. TypeScript nám tedy umožňuje psát třídy, rozhraní, enumy, datové typy, generiku apod.

Výsledný TypeScriptový kód je kompilován do JavaScriptu a je možné tedy v TypeScriptu psát rovněž JavaScriptové kódy [11].

5.2.3 GraphQL

GraphQL je open source dotazovací jazyk, který je určený na posílání požadavků na API pro získávání dat. Do budoucna se s tímto jazykem počítá jako s nástupcem REST API. Tento jazyk byl vytvořený společností Facebook a v roce 2015 byla uvedena první verze pro veřejnost. Pomocí tohoto jazyka dokážeme velmi efektivně získávat potřebná data a můžeme si definovat, které data budeme chtít obdržet ze serveru, což má za následek snížení rozsahu přenesených dat ze serveru na klienta.

5.2.4 Webpack

Webpack je JavaScriptový nástroj, který se skládá z jednotlivých modulů, které nám zajišťují například kompilaci TypeScriptového kódu do JavaScriptového, konverzi less nebo sass stylů do CSS, minifikací JavaScriptového kódu a jeho bundlování, což znamená, že ve výsledku máme celou klientskou aplikaci v jednom JavaScriptovém souboru nebo v neposlední řadě také načítání různých obrázkových formátů [12].

5.2.5 React

React je JavaScriptová knihovna, která slouží pro vytváření klientských aplikací. Tato knihovna je vyvinuta společnosti Facebook a v posledních letech její popularita stále roste.

Obrovskou výhodou této knihovny je fakt, že když už se programátor naučí psát webové aplikace v této knihovně, tak by měl být již schopný psát i webové aplikace v knihovně React Native, jelikož syntaxe a samotné principy jsou identické.

React využívá tzv. Virtuální dokumentový objektový model, který se vytváří v paměti a při změnách se tento model velmi efektivně aktualizuje. V praxi to znamená, že pokud se změní

například nějaký záznam v tabulce na celé stránce, React se postará o to, že se překreslí pouze ta část, ve které došlo ke změně, nikoliv celá stránka a tím je zajištěna vysoká rychlost aplikace [6].

5.2.6 Material UI

Material UI je komponenta postavená na knihovně React, která implementuje Material Design od společnosti Google. Tato komponenta obsahuje širokou škálu již vytvořených komponent, jakými jsou například DropDown listy, dialogová okna, tabulky, různé formy menu apod.

5.2.7 Bootstrap 4

Bootstrap je front-endový framework pro snadný a rychlý vývoj responzivních webů, kde se obsah dynamicky přizpůsobuje s ohledem na používané rozlišení. Je jedním z nejrozšířenějších frameworků, které se používají při vývoji moderních webových aplikací [7].

6 SonarQube

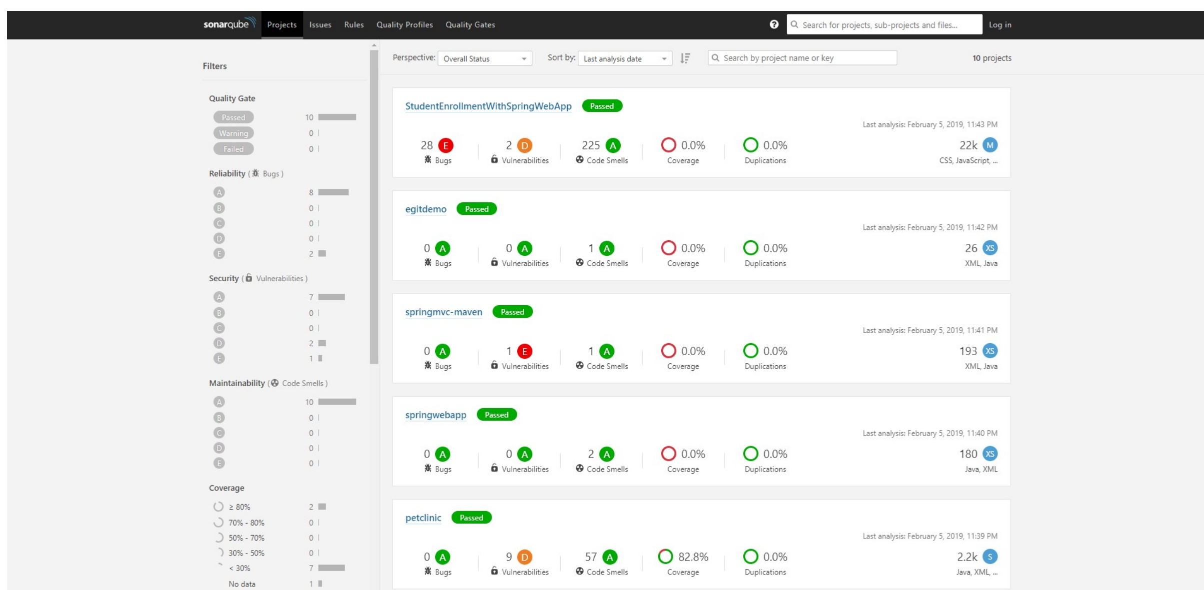
6.1 Co je SonarQube

SonarQube je open source platforma vyvíjená společností SonarSource za účelem kontinuálního analyzování kvality kódu a vytváření automatických revizí díky statické analýze zdrojového kódu.

Výsledkem těchto revizí jsou detekce chyb v kódu, do kterých se řadí například špatné využití funkcí, možných null referencí, která by mohla vést k pádu systému apod. Dále tzv. "code smell", mezi které se řadí například duplikace kódu, velké množství parametrů v metodě, zbytečná složitost metody, která by se mohla rozpadnout do více dílčích metod.

Výstupem této analýzy je také množství duplikovaného kódu, procentuální pokrytí zdrojového kódu jednotlivými testy, možná bezpečností rizika použitých pluginů v daném projektu [1].

SonarQube má podporu pro analýzu více jak dvaceti programovacích jazyků, kde podpora analýzy pro některé jazyky je dostupná pouze v placené verzi.



Obrázek 2: Uživatelské rozhraní aplikace SonarQube

6.2 Skenery

Jelikož jednotlivé projekty se mohou lišit jazykem, ve kterém je software sepsaný, popřípadě i buildovacím nástrojem, který je použitý pro překlad, SonarQube má dostupných hned několik druhů Skenerů, které jsou speciálně určeny pro různé typy buildovacích nástrojů.

Tyto skenery provádějí analýzu příslušných projektů, avšak pro úspěšnou analýzu je potřeba daný projekt ihned přeložit po vytvoření instance SonarQube skeneru. Tento proces bude více rozepsán v pozdějších sekcích.

6.2.1 Jazyk C#

Pro projekty postavené na jazyce C# má SonarQube k dispozici dva skenery, jeden slouží pro projekty postavené na .NET Frameworku a druhý je určen pro projekty postavené na .NET Core.

Bohužel ne všechny projekty lze analyzovat, jelikož existují jistá omezení, kdy analýzu nelze provést. V první řadě je nutné, aby analyzovaný projekt byl sestavován MSBuild verzí 14 a výše. Starší verze nejsou podporovány. Toto omezení se týká projektů postavených nad .NET Frameworkem.

Pro analyzování .NET Core projektů je nutné, aby daný projekt byl postaven na .NET Core ve verzi 2.0 a vyšší. Nižší verze nejsou rovněž podporovány.

6.2.2 Jazyk Java

Pro projekty postavené na jazyce Java je k dispozici hned několik druhů skenerů pro různé buildovací nástroje. Aktuálně existuje podpora pro buildovací nástroje Maven, Gradle, ANT, avšak v rámci diplomové práce budou implementovány moduly podporující Maven a Gradle. ANT nebude podporován z důvodu mizivého procenta projektů postavených na tomto nástroji v poslední době.

I pro projekty napsané v jazyce Java a buildované pomocí nástroje Maven existuje omezení, kdy lze analyzovat pouze ty projekty, které používají Maven ve verzi 3 a výše.

6.2.3 Ostatní jazyky

Mimo již zmíněné dva jazyky, pro které jsou speciálně vytvořené vlastní skenery, SonarQube disponuje ještě jedním univerzálním analyzátorem, který dokáže provádět analýzu mnoha dalších jazyků. Všechny podporované jazyky, firmou stojící za vznikem aplikace SonarQube, jsou zobrazeny v následujícím seznamu, kde jazyky, které mají za názvem hvězdičku jsou dostupné pouze v placené verzi SonarQube.

Tato podpora jazyků je realizována skrze pluginy, které stačí stáhnout a přesunout do složky **plugins** aplikace SonarQube. Nutno podotknout, že existuje ještě celá řada pluginů, která ještě více rozšiřuje podporu dalších jazyků, kde můžeme najít třeba pluginy pro jazyky **Perl**, **Haskell** nebo **F#**, avšak i zde jsou některé z těchto pluginů zpoplatněny [10].

SonarABAP *
SonarApex *
SonarC#
SonarCFamily C/C++ *
SonarCFamily ObjC *
SonarCOBOL *
SonarCSS
SonarFlex
SonarGo
SonarHTML (prev. SonarWeb
SonarJava
SonarJS
SonarKotlin
SonarPHP
SonarPLI *
SonarPLSQL *
SonarPython
SonarRPG *
SonarRuby
SonarScala
SonarSwift *
SonarTS
SonarTSQL *
SonarVB6 *
SonarVB
SonarXML

Obrázek 3: Dostupné pluginy pro SonarQube - [10]

7 Serverová část

V rámci této kapitoly se podrobněji podíváme na to, jakým způsobem je navržena serverová část aplikace, skrze kterou budou chodit veškeré požadavky z klientské vrstvy na vystavená API.

Pro velmi snadnou integraci modulů, které by komunikovaly jak se servery vzdálených repozitářů, tak i s mnou postaveným serverem, bylo potřeba vymyslet unifikované rozhraní, na kterém jsou tyto moduly postavené bez nutnosti zásahu do zdrojového kódu.

7.1 Autofac - Dependency Injection

Pro odstranění závislostí mezi jednotlivými třídami je využit vzor Inversion of Control, Jedná se o mechanismus, kdy třída, která potřebuje mít vytvořenou instanci jiné třídy pro využití jejích metod, si vytvoří danou instanci na základě jejich závislostí.

Pro správné vytváření těchto instanci je nutné, si v první řadě jednotlivé třídy zaregistrovat do kontejneru. Pro registraci tříd je v aplikaci implementován modul *ServerModuleBase*

Všechny třídy, jejichž název končí textem *Repository*, *Controller* a *Service* by již měly být přidány do IoC kontejneru díky níže uvedenému kódu. Tato implementace je uskutečněna z důvodu, aby nebylo potřeba každý nový kontroler, repozitář a servisní vrstvu dodatečně přidávat zde do kontejneru.

```
builder.RegisterAssemblyTypes(GetType().Assembly)
    .Where(t => t.Name.EndsWith("Repository")
    || t.Name.EndsWith("Controller")
    || t.Name.EndsWith("Service"))
    .AsImplementedInterfaces()
    .InstancePerLifetimeScope();
```

Výpis 2: Ukázka registrace do kontejneru dle názvu

V případě, že třída závisí současně na několika dalších třídách, tak nám již výše uvedené registrování nestačí a je nutné explicitně zaregistrovat požadované závislosti.

```
builder.RegisterType<ProcessService>().AsImplementedInterfaces().
    InstancePerLifetimeScope();
builder.RegisterType<AnalyzeService>().AsImplementedInterfaces().
    InstancePerLifetimeScope();
builder.RegisterType<ProcessRepository>().AsImplementedInterfaces().
    InstancePerLifetimeScope();
```

Výpis 3: Ukázka explicitní registrace tříd

7.2 REST (Representational State Transfer)

Pro komunikaci mezi klientem a serverem je použit architektonický styl REST, který nám definuje endpointy, na které se budou posílat požadavky ze strany klienta.

Server nabízí 4 základní implementace pro zpracování dat.

- **GET** - v rámci aplikace metody označené tímto typem budou sloužit převážně pro získávání jednotlivých informací dle jejich unikátního identifikátoru, který je na serveru reprezentován datovým typem **GUID**. Jedná se o globání unikátní identifikátor, díky použití tohoto datového typu je zajištěna vlastnost, že každý záznam v databázi bude mít svůj unikátní klíč skrze všechny záznamy v naší databázi.
- **POST** - metody s tímto označením nám v aplikaci mohou sloužit jak pro ukládání záznamů do databáze, tak i pro získavání seznamů dat, na které lze uplatnit filtrování dle zadaných kritérii.
- **UPDATE** - skrze metody označené tímto typem dochází pouze k aktualizaci již existujících záznamů v databázi.
- **DELETE** - mazání záznamu v databázi je reprezentováno metodou s tímto označením, kde však nebude docházet k fyzickému smazání záznamu z databáze, ale pouze nastavení údaje, který nám drží informaci o čase smazání na aktuální čas, tímto budeme vědět, že nad daným záznamem byla provedena operace smazání a dále již není zobrazován v klientské vrstvě a současně s tím nepřijdeme o žádná data, která by se nám mohla hodit někdy v budoucnu.

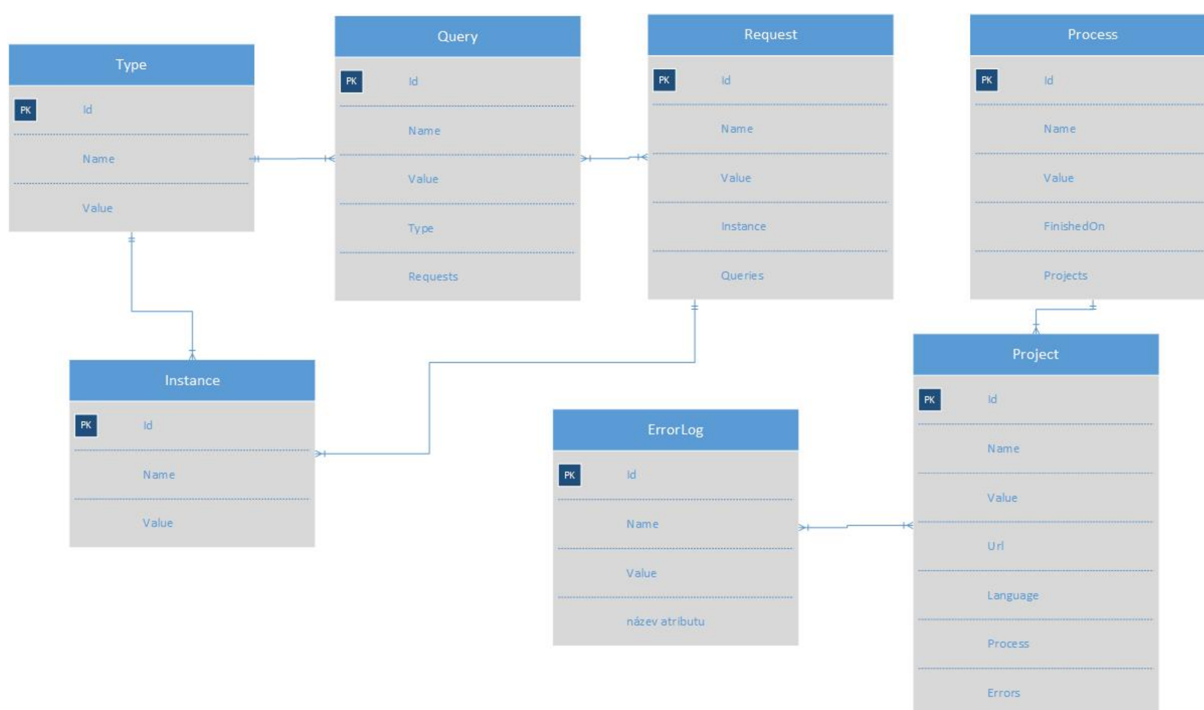
7.3 Datový model

Součástí aplikace je vytvořený datový model, který popisuje strukturu dat a vazby mezi jednotlivými daty, díky kterým lze v konečné fázi s nimi manipulovat, popřípadě nějakým způsobem je upravovat.

Níže jsou uvedené pouze základní modely aplikace, které hrají důležitý faktor pro ziskávání dat z API, popřípadě hrají velkou roli při analýze samotných projektů.

- **BaseEntity** - jakýkoliv model, který je určen pro uložení do databáze musí dědit z této bázové třídy.
- **ProcessEntity** - vzniká při vytvoření požadavku na analýzu projektu nebo několika projektů současně.
- **ProjectEntity** - obsahuje informace ohledně projektu, který byl podroben analýze a je svázán s jednotlivým procesem.

- **RequestEntity** - aplikace umožňuje ukládání již vytvořených požadavků, které byly směrovány na vzdálené servery repozitářů. Pro tyto účely nám slouží tato entita, která si drží veškeré informace o tom, jaké parametry byly použity a kam byl požadavek odeslán, abychom ho mohli kdykoliv zrekonstruovat, bez nutnosti opětovného definování v klientské vrstvě.
- **TypeEntity** - tato entita udržuje definici parametrů, které jsou přijímány službami GitHub a GitLab. Jako parametr si můžeme například představit filtrování dat dle programovacího jazyka.
- **QueryEntity** - entita udržující již konkrétní hodnoty, dle kterých se mají filtrovat záznamy. Například jsme si zvolili, že chceme provést filtrování na základě jazyka, neboli typu a hodnotou této entity je již konkrétní jazyk, například programovací jazyk Java.



Obrázek 4: Databázový model

7.4 Architektura

7.4.1 MVC

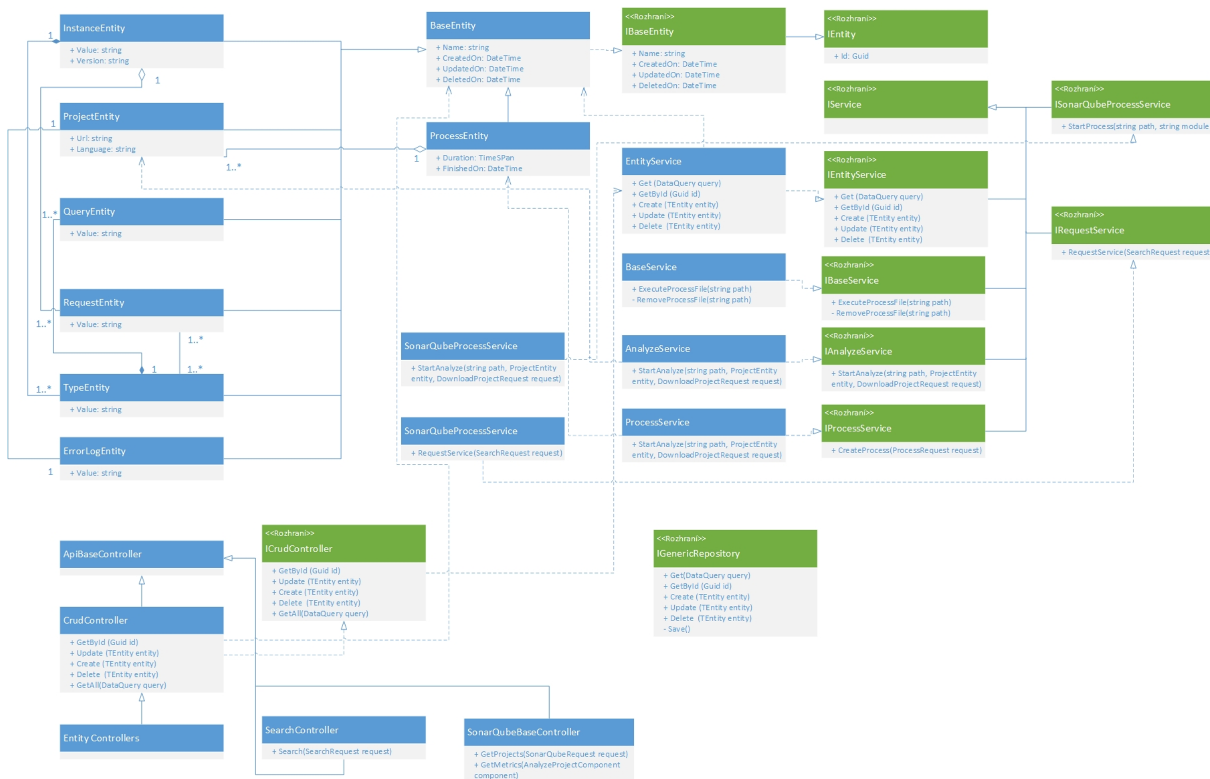
Serverová část je postavena na architektuře MVC, která je využívána ve většině moderních webových aplikací a je složena ze 3 základních částí.

1. Model: Jedná se o reprezentaci dat, které jsou uložený v databázi, se kterými se následně pracuje v aplikaci.

2. View: Tato vrstva je zodpovědná za převod dat do nějaké podoby, která je následně zobrazena uživateli. V rámci aplikace máme vystavené pouze jedno View, na které je navázaná klientská vrsta, která je postavená na knihovně React.

3. Controller: Kontrolery nám zpracovávají veškeré klientské dotazy, avšak další funkčnost nemají. Jsou zodpovědné pouze za přijetí požadavku a následně jeho odeslání do dalších vrstev, kde dochází k dalšímu zpracování a výsledky těchto operací jsou následně odeslány zpátky na klientskou část skrze kontroler.

7.4.2 Diagram tříd



Obrázek 5: Třídní diagram serverové části

7.4.3 Repository a Unit of Work pattern

Jedná se o návrhové vzory, které jsou určeny pro vytvoření abstraktní vrstvy mezi logickou business vrstvou a perzistenční vrstvou aplikace. Díky implementace těchto návrhových vzorů lze manipulovat s více entitními třídami v jednom zápisu do databáze [5].

7.4.4 Generická Repository

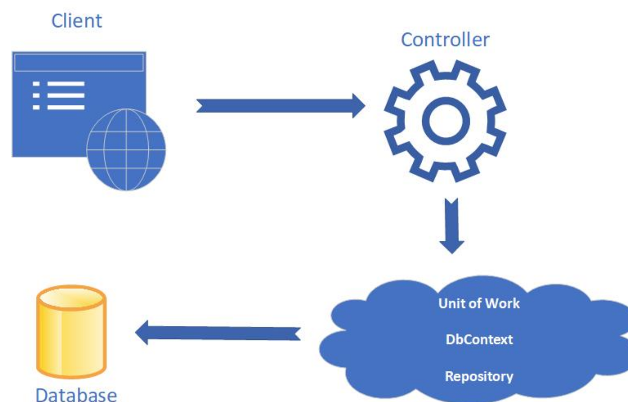
Vytváření separovaných repository tříd pro každou entitu by vedlo k redundantnímu kódu a je zbytečné implementovat stejnou funkcionalitu několikrát, například funkci pro uložení dat do databáze. Jednak je to velmi nepraktické, ale současně s tím se výrazně snižuje dlouhodobá udržitelnost kódu a při jakékoliv změně bychom museli provést úpravy na všech ostatních místech.

```
public interface IGenericRepository<T>
{
    IEnumerable<T> Get(DataQuery query);
    IEnumerable<T> FindBy(Expression<Func<T, bool>> expression);
    T GetById(Guid id);
    T Add(T entity);
    T Delete(T entity);
    T Update(T entity);
    void Save();
}
```

Výpis 4: Rozhraní bázové repository

7.4.5 Unit of Work

Jedná se o koncept, díky kterému lze efektivně implementovat Repository pattern. Unit of Work zajišťuje použití jedné instance databázového kontextu při využití několika repository. V této třídě je implementována metoda *Save()* pro potvrzení uložení dat do databáze. Všechny tyto repozitáře následně využívají stejnou instanci databázového kontextu a díky tomu můžeme ukládat více záznamů v jednom zápisu do databáze [5].



Obrázek 6: Unit of Work pattern

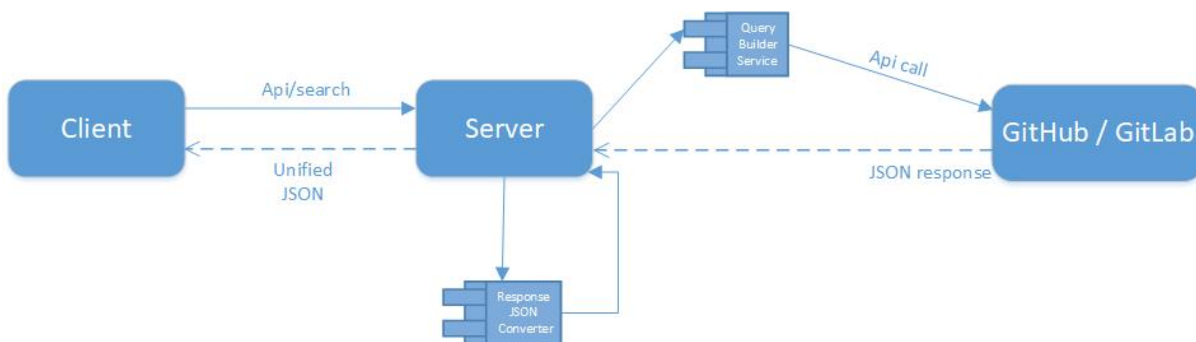
7.5 Komunikace s GitHub a GitLab

Pro komunikaci se servery repozitářů vznikl jeden unifikovaný end-point, který zpracovává veškeré klientské požadavky na hledání projektů se zadanými parametry.

Na server se pošle požadavek, který musí obsahovat informaci o tom, na jaký server se máme dotazovat a také filtrovací parametry, ze kterých se poskládá výsledný dotaz v servisní vrstvě, která je pro to určená a následně je tento požadavek odeslán na vzdálený server.

Výsledkem dotazu je JSON, který je ještě před odesláním na klienta zpracován v námi sestaveném konvertoru.

Takto se můžeme dotazovat na různá API vystavená těmito službami skrze jedno místo a díky tomu lze snadno vytvářet jednotlivé rozšiřující moduly.



Obrázek 7: Diagram komunikace se servery GitHubu a GitLabu

7.6 Vrstvy aplikace

Kvůli odstínění logiky od jednotlivých kontrolerů je zapotřebí implementace servisní vrstvy, která se stará o samotné zpracovávání požadavků. Servisní vstva je rozdělena do několika typů, kde každá z nich má na starosti část celého procesu analýzy dat, popřípadě zpracování požadavků.

7.6.1 Procesní vrstva

Procesní vrstva se stará o vytvoření procesu, který nám drží veškeré informace o projektu, či projektech, které byly analyzovány při jednom uživatelském požadavku. Tento záznam vzniká vždy při obdržení požadavku na provedení analýzy nad projekty hostovaných na službách GitHub nebo GitLab. Po provedené všech analýz nad projekty je následně tento záznam uložen do databáze

7.6.2 Analytická vrstva

Analytická vrstva řeší veškerou analýzu jednotlivých projektů a skládá se z jednotlivých fází.

1. **Získání zdrojového kódu** - Abychom mohli podrobit projekt analýze, je potřeba mít dostupné všechny zdrojové kódy aplikace. V případě, že se ve složce již stejný projekt nachází, je tato fáze přeskočena.

2. **Detekce buildovacího nástroje** - Aby mohla proběhnout samotná analýza projektu pomocí aplikace SonarQube, je zapotřebí prvně provést detekci prostředí, ve kterém je projekt vyvíjen. V první řadě se zjíšťuje, jaký programovací jazyk je zvolen při vývoji a následně pomocí jakého buildovacího nástroje by měl být projekt sestaven.

   - **.NET Core** - pro zjištění, zda se jedná o .NET Core projekt, je nutné náhlédnout do souboru s příponou *.csproj* a následně najít tag *<TargetFramework>* , který nám určuje, jaká verze frameworku je použita. V případě absence tohoto tagu víme, že se nejedná o .NET Core projekt.
   - **.NET Framework** - detekce probíhá obdobně, jako je popsáno výše, avšak zde je potřeba hledat tag *<TargetFrameworkVersion>*, který nám říká, že tento projekt je sestaven pomocí .NET Framework.
   - **Maven** - v případě, že se nám vyskytne soubor *pom.xml* v kterékoliv složce, víme, že daný projekt je sestaven pomocí Mavenu.
   - **Gradle** - v případě, že se nám vyskytne soubor *build.gradle* v kterékoliv složce, víme, že daný projekt je sestaven pomocí Gradle.

   Nutno podotknout, že ne vždy se detekce provede úspěšně, jelikož existují případy, kdy například u **.NET** projektů nelze detekovat používaný framework, a to z důvodu, že v daném tagu, který má obsahovat tuto informaci je použitá proměnná, jejichž hodnota se nachází v jiném souboru a není možnost, jak tento problém efektivně řešit, jelikož není znám název tohoto souboru ani jeho umístění, ve kterém je možné dohledat tuto klíčovou informaci.

3. **Spuštění analýzy** - Průběh analýzy se skládá z několika fází, které si popíšeme níže.

   - **Vytvoření instance SonarQube** - Abychom mohli analyzovat projekt, je nezbytné znát výsledek předchozí operace, pomocí které lze správně určit, který ze SonarQube skeneru se má použít.
     - **.NET Core** - pro spuštění sepsaných testovacích scénářů je nutné v tomto případě spustit následující *dotnet.exe* process obsahující následující výraz.
       test cesta-k-projektu –collect \"code coverage\"

       Vzhledem k tomu, že SonarQube podporuje import reportů testů pouze s příponou *.coveragexml* je nutné tyto reporty převést do požadovaného souboru pomocí

CodeCoverage.exe toolu, který je dostupný skrze IDE *Visual studio.*

SonarQube analýza je následně definována těmito příkazy:

dotnet <cesta k SonarScanner.MSBuild.dll> begin /k:"project-key"
dotnet build <cesta k solution.sln>
dotnet <cesta k SonarScanner.MSBuild.dll> end

– **.NET Framework**
  Analýza probíhá obdobně, jako je tomu u .NET Core projektu s tím rozdílem, že pro definování příkazů pro analýzu pomocí aplikace SonarQube, není možné použít process *dotnet.exe.*

  SonarQube analýza je následně definována těmito základními příkazy, kde je však možné rozšiřovat nebo specifikovat konkrétní chování díky nepovinným parametrům:

  SonarScanner.MSBuild.exe begin /k:"project-key"
  MSBuild.exe <path to solution.sln> /t:Rebuild
  SonarScanner.MSBuild.exe end

7.6.3 Error log

V případě neúspěšné analýzy projektu je chyba zaznamenána a uložena do error logu, který je reprezentovaný záznamem v databázi a je propojený se záznamem projektu.

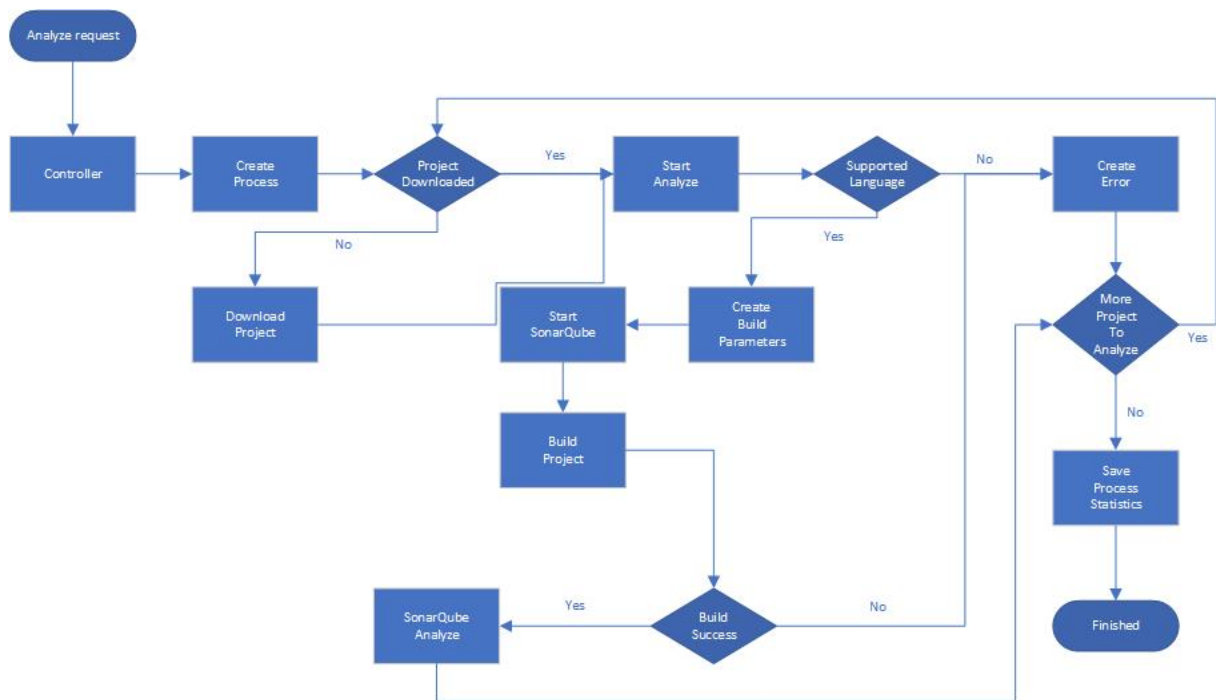
Tento log nám slouží pro získání informací o tom, z jakého důvodu není možné provést analýzu, jelikož ne všechny projekty lze takto automatizovaně podrobit analýze a i přesto chceme mít dostupná data o tomto selhání v klientské vrstvě, abychom mohli na závěr zhodnotit výsledky dosažené touto automatizací. Tyto chybové hlášky jsou extrahovány z výpisu procesu, který je generovaný během analýzy.

7.7 Aktivitní diagram průběhu analýzy

V této sekci je zahrnutá ukázka průběhu analýzy projektů pomocí aktivitního diagramu od uživatelského kliknutí pro započtení analýzy až po její dokončení.

7.7.1 Datová vrstva

Datová vrstva slouží jak pro ukládání samotných výsledků provedených analýz a procesů, tak i pro vytváření nových záznámu, které nám mohou sloužit pro vytváření dotazů na API verzovacích systémů.



Obrázek 8: Aktivitní diagram průběhu analýzy

V případě potřeby použití této vrstvy je nutné, aby nově vytvořený model dědil z třídy *BaseEntity*

```
public interface IEntityService<T> : IService where T : BaseEntity
{
    IEnumerable<T> Get(DataQuery query);
    T GetById(Guid id);
    T Create(T entity);
    T Update(T entity);
    T Delete(T entity);
}
```

Výpis 5: Ukázka rozhraní pro práci s daty

Je také pro nově vytvořený model nutné vytvořit vlastní repozitář, kde v případě potřeby lze přetížit jednotlivé metody. Níže je ukázka repozitáře pro **Process** entitu, kde stačí pouze vytvořit repozitář, který je určený pro konkrétní entitu a o veškerou funkcionalitu se již postarají funkce, které jsou implementovány ve třídě **GenericRepository**. V případě potřeby je možné tyto funkce přetížit a rozšířit o další funkcionalitu.

```
public class ProcessRepository : GenericRepository<ProcessEntity>
```

```
{
    public ProcessRepository(AppContext context) : base(context)
    {
    }
}
```

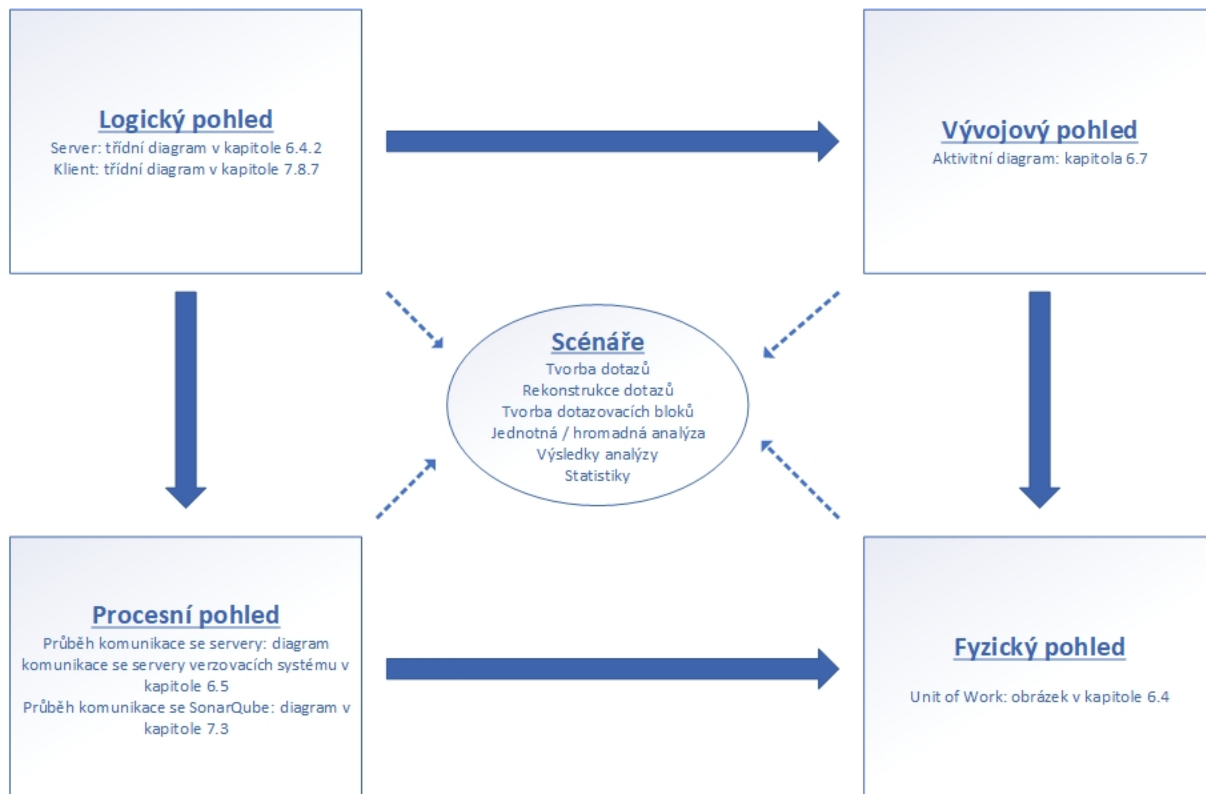
Výpis 6: Ukázka implementace konkrétního repozitáře

Metody datové vrstvy:

- **GetAll** - slouží pro získání všech záznamů z databáze
- **GetById(Guid id)** - slouží pro získání jednoho záznamu na základě unikátního klíče
- **Add(T entity)** - slouží k vytvoření nového záznamu
- **Update(T entity)** - slouží pro aktualizaci již vytvořeného záznamu
- **Delete(T entity)** - slouží ke smazání již existujícího záznamu

8 Klientská část

Architektura klientské části se skládá z několika na sobě nezávislých částí pro zachování jejich přenositelnosti a umožnění velice snadnému rozšiřování jednotlivých komponent.



Obrázek 9: 4 + 1 Kruchten diagram

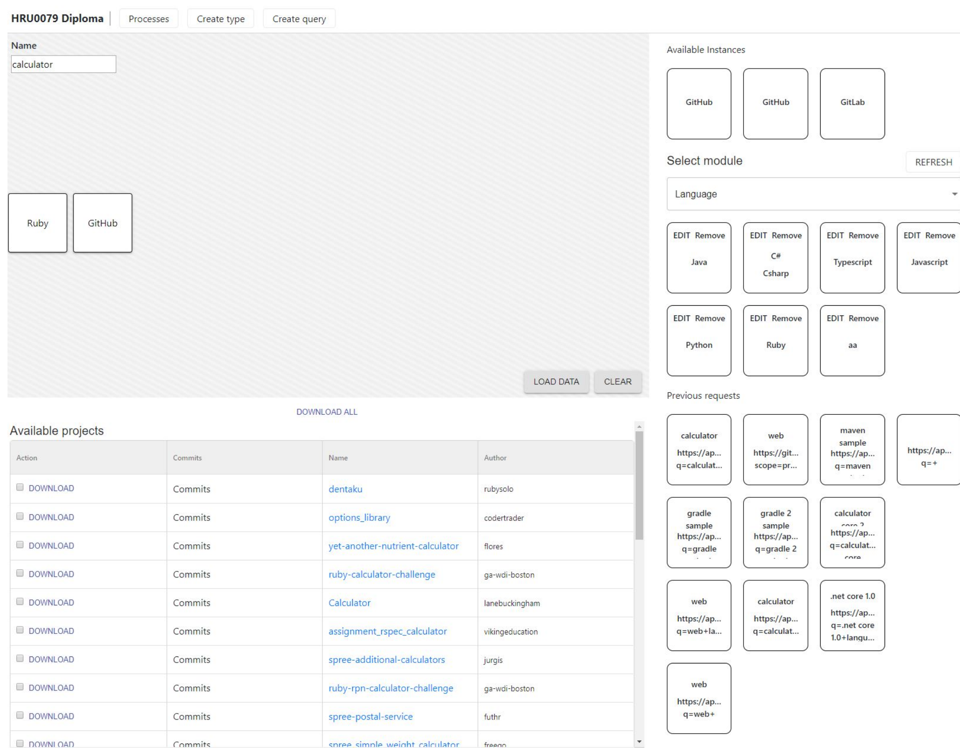
8.1 Výhoda využití TypeScriptu

Klientská část je postavena na knihovně React a jako programovací jazyk je zvolen TypeScript z důvodu lepší udržitelnosti kódu a silné typové kontrole. Nabízela se také možnost využití JavaScriptu s podporou flow pro typovou kontrolu, avšak konfigurace tohoto rozšíření je daleko náročnější a současně s tím při rozsáhlejším projektu je typová kontrola velmi pomalá a stává se, že ověření typu objektu v rozsáhlejších projektech trvá i déle, než 5 sekund a díky těmto neduhům je zvolen jako programovací jazyk TypeScript [11].

```
{
  "compilerOptions": {
  "baseUrl": "ClientApp",
  "module": "es6",
  "moduleResolution": "node",
```

```
    "target": "es5",
    "jsx": "react",
    "sourceMap": true,
    "lib": [ "es6", "dom" ],
    "types": [ "webpack-env" ],
  },
    "exclude": [
      "bin",
      "node_modules"
    ]
  }
```

Výpis 7: konfigurace TypeScriptu v tsconfig.json

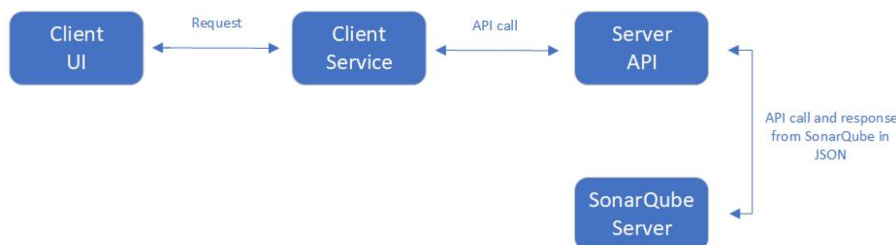


Obrázek 10: Ukázka UI

8.2 Komunikace s aplikací SonarQube

Aby bylo možné získávat reporty jednotlivých analýz ze serveru spuštěné aplikace SonarQube, bylo potřeba vytvořit end-point, který umožňuje zpracovávat požadavky z klientské vrstvy, které se následně odeslaly na SonarQube server.

Přímé dotazování z klientské vrstvy na SonarQube server není možné kvůli CORS a kvůli zachování bezpečnosti aplikace a proto veškeré požadavky putují skrze vystavený kontroler na serveru.



Obrázek 11: Komunikace se serverem SonarQube

8.3 Jádro klientské aplikace

Třída Application představuje jádro klientské části, která využívá návrhového vzoru jedináček, jelikož v rámci klientské části chceme mít vytvořenou pouze jednu instanci této třídy, která je navíc přístupná z jakékoliv komponenty.

Tato třída nám drží informace o konfiguraci projektu, popřípadě lze zde zapisovat klíčová data, která jsou potřebná pro správné fungování ostatních komponent a také nám zpřístupňuje další funkcionalitu, která je určena například pro vytváření dotazu na serverové API.

Níže je ukázka jádra klientské části.

```
export class Application {
  private readonly _configuration: IConfiguration;
  private readonly _githubClient: GithubClient.IGithubClient;

  private constructor() {
    this._configuration = {
    githubApiUrl: "https://api.github.com",
  }
    this._githubClient = new GithubClient.GithubClient(this._configuration.
        githubApiUrl);
  }

  private static get getInstance(): Application {
    if (!this._instance) {
      this._instance = new Application();
    }
    return Application._instance;
```

```
    }

    public static get apiClient(): GithubClient.IGithubClient {
      return Application.getInstance._githubClient;
    }
  }
```

Výpis 8: Klientské jádro

8.4 Bázové třídy

V aplikaci jsou vytvořené 4 bázové třídy, které jsou určené pro budoucí rozšiřování, ze které by veškeré komponenty měly dědit.

- **ComponentBase** - tato bázová komponenta je určená pro veškeré komponenty, které jsou využívány v níže uvedených komponentách. Obsahuje funkce, které usnadňují práci se stavy jednotlivých komponent
- **ListBase** - tato komponenta je určena pro zobrazení tabulek, která obsahuje základní implementace metod a díky této vlastnosti již není potřeba tyto metody implementovat opakovaně.
- **DetailBase** - komponenta sloužící pro zobrazení dat, které jsou získány na základě unikátního klíče a stejně jako v předchozím případě obsahuje základní implementace pro práci s daty.
- **ViewBase** - pro odstínění logiky od zbytku aplikačních částí vznikla tato komponenta, která by měla využívat veškeré komponenty, které jsou odvozené z výše uvedených bázových tříd. Slouží k zobrazení finální stránky pod určitou URL a obsahuje implementace metod, které jsou určení pro navigaci mezi stránkami v rámci aplikace.

8.5 Views

Pro maximální škálovatelnost aplikace bylo potřeba vytvořit komponentu, která se stará o samotné zobrazení obsahu stránek, kde její obsah je vyskládán z jednotlivých komponent, které lze následně využít kdekoliv jinde.

Všechny komponenty, které budou sloužit pro zobrazování obsahu musí dědit z třídy **ViewBase**, která má implementované potřebné metody pro korektní vykreslování a obsahuje funkce pro správné přesměrovávání mezi stránkami aplikace.

8.6 Routování

Nedílnou součástí aplikace je také routování, díky kterému aplikace zná další **Views**, které jsou v aplikaci dostupné a pod jakou URL adresou a díky tomu lze provést přesměrování.

Každé taková zobrazovací komponenta obsahuje interface obsahující veškeré potřebné parametry určené pro správné přesměrování.

Routování je složené ze tří částí, kdy v konstantě **paths** jsou definovány prvni segmenty adresy, následně je vždy vytvořena pro každé **view** její routovací funkce s případnými parametry, která volá generickou funkci obstarávající správné sestavení výsledné url na základě vstupních parametrů.

```
export const paths = {
  commitList: "/commits",
}
export function getFullUrl<TParameters>(baseUrl: string, parameters?:
   TParameters) {
  let result = baseUrl;
  if (parameters) {
    Object.getOwnPropertyNames(parameters).map((name, index) => {
    result += (index === 0 ? "?" : "&") + name + "=" + encodeURIComponent(
        parameters[name]);
  });
}
  return result;
}
export function getCommitListUrl(owner: string, repositoryName: string,
   projectType) {
  return getFullUrl<ICommitListRouteParameters>(paths.commitList, {
    owner: owner,
    repositoryName: repositoryName,
    projectType: projectType
  });
}
export const routes = <Layout>
  <Route exact path={paths.commitList} component={CommitListView} />
</Layout>;
```

Výpis 9: Ukázka routování

8.7 Skládání dotazů

Pro vytváření dotazů odesílaných na služby GitHub a GitLab vzniklo jednoduché uživatelské rozhraní, které je rozděleno na 4 části. Tyto části jsou poskládáne z jednotlivých modulů, kterým jsou věnovány následující části kapitoly.

Úvodní stránka aplikace je určena pro skládaní dotazů, která je rozdělená na již zmíněné 4 části, kde levá horní část je určena pro skládání dotazů. Tato plocha podporuje drag&drop funkci a také obsahuje textové pole, které je určeno pro zadání textu, který se vyhledává na GitHubu a GitLabu v popisu projektu, popřípadě jeho názvu. Pro jednoznačné určení této plochy je použito bílo šedé šrafování.

Skládání dotazu se děje pomocí vydefinovaných bloků, které jsou součástí jednotlivých modulů, dále už jen selektory, které jsou zobrazeny v pravé části obrazovky.

V případě, že není vybraná hodnota instance repozitáře, je vždy zobrazen pouze selektor pro vybrání repozitáře, na který se odesílá dotaz. Vybrání bloku se provádí pouhým přetažením do vyšrafované plochy. Po zvolení instance máme následně dostupné všechny ostatní selektory pro větší specifikaci dotazu.

8.8 Dostupné moduly

Aplikace obsahuje několik zásuvných modulů, o kterých si povíme v této sekci. Tyto moduly jsou vytvořené s ohledem na možné další rozšiřování, popřípadě vytváření nových modulů a veškerá funkcionalita je vytvořena v bázové komponentě pro určitý typ modulu a má vystavené vlastní rozhraní, skrze které lze ovlivňovat jejich chování.

8.8.1 Selektor instance

Tento modul slouží pro vybrání repozitáře, na který se budou sestavené požadavky odesílat. Vždy může být zvolený pouze jeden repozitář, kde jeho volba probíhá přetažením bloku do vyhrazeného pole.

V případě, že bychom chtěli vybrat i jinou instanci, tak námi předchozí vytvořený dotaz zanikne z důvodu zajištění kompatibility při dotazování, jelikož API GitHubu a GitLabu jsou značně odlišné.

8.8.2 Typové Selektory

Jedním ze základních modulů aplikace jsou tzv. selektory, které nám slouží pro vytváření dotazů, který je odeslaný na vybrané API repozitáře.

Tyto selektory jsou reprezentovány formou jednotlivých bloků, které lze přesouvat do pole, které je proto určené a z nich se pak následně sestavuje výsledný dotaz. Selektory řadíme do určitých skupin, kde každá skupina je prezentována svým unikátním typem.

Pod pojmem typ si můžeme představit situaci, kdy uživatel aplikace chce filtrovat data dle určitých podmínek, například podle jazyka, ve kterém je projekt napsaný, popřípadě zda-li velikost projektu nepřesahuje určitou velikost, filtrace projektů, kde počet tzv. "forků"přesahuje nějakou hodnotu, filtrování dle specifického uživatele a mnoho dalšího.

Tyto všechny kategorie jsou tedy reprezentovány daným typem, který je nutné vybrat, abychom mohli tvořit dotazy na konkrétní hodnoty.

8.8.3 GraphQL selektor

GitHub obsahuje podporu dotazování se pomocí GraphQL a z tohoto důvodu je vytvořen modul podporující tento dotazovací jazyk. Pro dotazování pomoci GraphQL je nutné mít aktivní účet na GitHubu a mít vytvořený token, který slouží pro autentizaci uživatele. Bez vygenerovaného tokenu nebude možné tento modul používat.

Výhodou dotazovaní se pomocí tohoto modulu je to, že si můžeme dotahovat pouze ta data, která opravdu chceme, kde oproti standartním REST API získáváme vždy veškerá data, kde větší část z nich jsou pro využití v rámci této aplikace zbytečná.

Níže je uvedená krátká ukázka, jak takový dotaz pomocí GraphQL může vypadat. Ve výsledku dostaneme prvních 100 záznamů, které jsou napsané v jazyce JavaScript a v popisu projektu mají text "web"a mají více jak 5 odběrů a jednotlivé záznamy budou obsahovat pouze název projektu a primární jazyk, ve kterém je projekt sepsaný.

```
{
search(query: "web language:JavaScript followers:>5 ", first: 100 type:
    REPOSITORY ) {
   repositoryCount
   nodes {... on Repository {
      name
      primaryLanguage{name}
      }}
   }
}
```

Výpis 10: Ukázka dotazu pomocí GraphQL

Tvorba takového dotazu probíhá v textovém poli, kde je nutné daný dotaz sepsat a následně přesunout jako všechny ostatní definované selektory do příslušného pole.

Takto sepsané dotazy pomocí tohoto jazyka nelze kombinovat s jinými typy selektorů.

8.8.4 Query rekonstruktor

Aplikace obsahuje modul, který je určený pro rekonstrukci již vytvořených dotazů, které byly odeslány na server. Uživatel má možnost si každý dotaz takto uložit do databáze, kdy samotné uložení se provádí při odeslání požadavku. V případě, že v databázi již existuje dotaz, který byl jednou vykonán, uložení se neprovede i přesto, že uživatel si explicitně určil, že chce dotaz uložit.

Prezentace těchto dat na klientské vrstvě je identická se selektory. Rekonstrukce dotazu probíhá tak, že při přetažení zvoleného bloku se na základě hledaného výrazu a všech parametrů vygenerují veškeré bloky v poli, které je pro to určené.

V případě, že uživatel takto využije danou rekonstrukci a provede modifikaci nějakého parametru nebo hledaného výrazu a stejný záznam není uložen v databázi, provede se jeho uložení, pokud to uživatel požaduje.

8.8.5 Modul pro zobrazení ukončených procesů

Tento modul je určený pro zobrazení všech analýz, které byly provedeny nad projekty bez ohledu na to, zdali je analyzován jeden nebo více projektů.

Takto vytvořené procesy nesou informace o jednotlivých projektech, které jsou podrobeny analýze, jejich počet, procentuální úspěšnost, výsledky testu a různá další data, které jsou součástí výsledku nabízených skrze aplikaci SonarQube.

8.8.6 Prohlížeč projektů

Tento modul nám slouží pro zobrazení dat, které dostáváme jako výsledek zpracovaného dotazu ze služeb GitHub a GitLab. Skrze tento modul můžeme zvolit, jaký projekt budeme chtít analyzovat, popřípadě lze zvolit množinu projektů k analýze a také nám umožňuje vybrat konkrétní commit projektu, který je podroben analýze.

Available projects

Action	Commits	Name	Author
DOWNLOAD	Commits	dentaku	rubysolo
DOWNLOAD	Commits	options_library	codertrader
DOWNLOAD	Commits	yet-another-nutrient-calculator	flores
DOWNLOAD	Commits	ruby-calculator-challenge	ga-wdi-boston

Obrázek 12: Zobrazení dostupných projektů

8.8.7 Třídní diagram

Pro nastínění architektury klientské vrstvy je vytvořen zjednodušený třídní diagram, který obsahuje všechny podstatné třídy, které jsou odpovědné za funkční celek této aplikace.

Obrázek 13: Třídní diagram klientské vrstvy

9 Úspěšnost analýzy

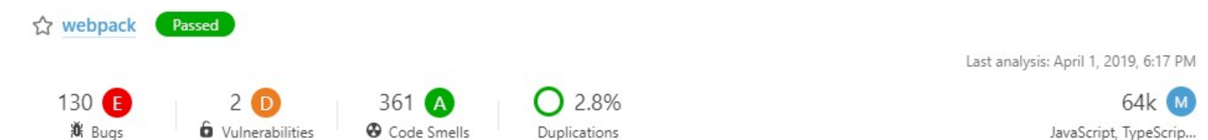
Pro vyhodnocení úspěšnosti automatického analyzování projektů skrze aplikaci je na závěr proveden test, kde je náhodně vybráno 15 projektů, které jsou napsány v jazyce Java, Ruby, Python, JavaScript a TypeScript ze služby GitHub, kde je použité klíčové slovo web.

Tyto jazyky jsou vybrány dle počtu zastoupení ve službě GitHub, jejíž statistika je uvedena v kapitole 3. Pro zajímavé srovnáni je přidán i jazyk TypeScript, který je nádstavbou JavaScriptu.

9.1 JavaScript

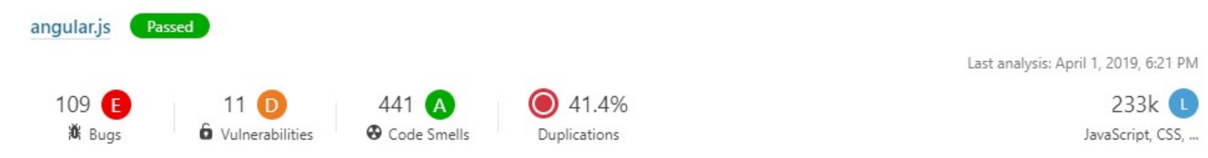
V případě JavaScriptových projektů je úspěšně provedena analýza ve všech případech, kdy rozpětí chyb v jednotlivých projektech je velmi výrazné. V námi analyzovaných projektech činilo rozmezí od 1 do 259 chyb.

Překvapivé výsledky jsou zjištěny u velmi využívaného JavaScriptového nástroje **Webpack**, který je rovněž využíván v této aplikaci, kde výskyt chyb v dané verzi je vyčíslen na hodnotu 130.



Obrázek 14: Webpack analýza

Zajimavým zjištěním jsou také výsledky u frameworku **AngularJS**, kde duplicita kódu přesahovala hranici 40%. Výskyt chyb je rovněž vysoký, ale v porovnáním s počtem řádků se tato hodnota zdá být zanedbatelná.



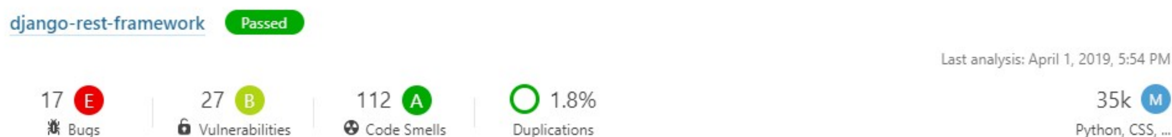
Obrázek 15: AngularJS analýza

9.2 Python

Při analyzování projektů napsaných v jazyce Python je dosaženo velmi dobrých výsledků. Je vybráno 15 projektů a až na jeden projekt se úspěšně povedlo všechny analyzovat. Jeden projekt nebylo možné stáhnout, přestože byla obdržena všechna data k danému projektu. Počet chyb u analyzovaných projektů se pohyboval v rozmezí 0 až 84, což v průměru činí 19 chyb na jeden projekt.

Mezi analyzovanými projekty byl i velmi populární framework **Django** spolu s **Django rest framework**

Obrázek 16: Django analýza



Obrázek 17: Django rest framework analýza

9.3 Ruby

Při vyhodnocení výsledků analýzy projektů napsaných v jazyce Ruby se množství chyb v jednotlivých projektech pohybovalo v rozmezí 0 až 9 a jedná se tak o vůbec nejlepší naměřené hodnoty ze všech projektů, které podlehly statické analýze pomocí aplikace SonarQube.

Obrázek 18: Analýza projektu napsaném v Ruby

9.4 TypeScript

Analýzá projektů napsaných v TypeScriptu byla nejdéle trvající analýzou ze všech. V zahrnutých projektech byl i velmi populární framework **Angular**, kde analýza této knihovny zabrala přibližně 20 minut.

Právě při analyzování této knihovny dostáváme zajímavé porovnání s jejím předchůdcem **AngularJS**, kdy její novější TypeScriptová verze značně nabrala na počtu řádků a drasticky se snížila duplikace kódu na necelých 6%.

Z výsledků analýz lze také usuzovat, že projekty napsané v jazyce TypeScript mají daleko menší chybovost a menší duplikaci kódu oproti projektům v JavaScriptu. Průměrný počet chyb je přibližně 4x menší oproti analyzovaným projektům v JavaScriptu.

Obrázek 19: Angular analýza

9.5 Java

U projektu napsaných v programovacím jazyce Java byla úspěšnost analýzy výrazně nižší. Z 15 projektů bylo úspěšně analyzováno 9 a to díky výrazně odlišné struktuře projektu, kde projekt se může skládat ze samostatných dílčích spustitelných projektů, převážně se jedná o demo ukázky. Dále nižší úspěšnost je ovlivněna kompatibilitou použitých skenerů a verzí buildovacího nástroje.

Nižší úspěšnost analýzy je tedy hlavně ovlivněna nutností sestavit projekt předtím, než lze provést samotnou analýzu a v případě chybného sestavení je celý tento proces následně neúspěšný.

9.6 Shrnutí

Přestože automatizace analýz pomocí aplikace SonarQube dosahuje velmi dobré procentuální úspěšnosti, je nutno podotknout, že výsledky analýz mohou být v jistých případech velmi zkreslené a výstupy nemusí odpovídat realitě, jelikož projekty mohou mít například specifické knihovny, které se nepodaří stáhnout, popřípadě nastavení analýzy pro určitý projekt vyžaduje specifické nastavení pro skener, který je použitý aplikací SonarQube.

10 Závěr

Cílem této diplomové práce je vytvoření webové aplikace, která umožňuje definování a provádění analýz nad projekty, které se nacházejí ve verzovacích systémech GitHub nebo v nějaké instanci aplikace GitLab. Pro provádění jednotlivých analýz a vytváření reportů je použita open source aplikace SonarQube. Je potřeba, aby celý proces byl maximálně automatizovaný, aby ho jakýkoliv uživatel mohl používat bez nutností jakýchkoliv dodatečných znalostí.

Ještě před samotným vývojem aplikace bylo potřeba provést analýzu současných možností jednotlivých API verzovacích systému GitHub a GitLab, kde v současném stavu lze konstatovat, že API GitHubu nabízí daleko lepší možností při sbírání dat ohledně projektů na základě konkrétních požadavků. GitLab API například neumožňuje filtrování dle programovacího jazyka a možnosti filtrování jsou značně omezené, a proto také není kladen příliš velký význam na tuto platformu při vývoji aplikace.

Samotná aplikace je rozdělená do dvou částí, kdy backendová část je postavená na platformě .NET Core 2.2 a klientská vrstva je postavená na JavaScriptové knihovně React s podporou TypeScriptu.

Samotné definování jednotlivých dotazů pro získání potřebných dat k analýze projektů z již zmíněných verzovacích systémů je prováděno vizuální podobou, kde máme předem definované skupiny filtrů, kde každá skupina obsahuje několik položek, které stačí pouze přetáhnout do příslušné oblasti, ze které se následně sestaví výsledný dotaz pro získání dat.

Aplikace umožňuje provádět jednotlivé analýzy projektů, popřípadě analýzu projektu k dané verzi, avšak je také implementována podpora pro hromadnou analýzu několika projektů současně, kde lze následně při hromadném analyzování porovnávat jednotlivé výsledky na základě námi definovaných kritérii ve velmi přívětivém uživatelském prostředí.

Výsledná architektura aplikace je navržena a sestavena tak, že lze velmi jednoduše provádět další rozšíření o jednotlivé moduly pro užší integraci s aplikací SonarQube, jak na klientské, tak i na backendové vrstvě.

Tato práce mi přinesla velmi užitečné poznatky v oblasti testování a analýzy kódu a vidím velký potenciál pro budoucí využití této práce v praxi v oblasti automatizovaného testování.

Literatura

[1] CAMPBELL, G. Ann a Patroklos P. PAPAPETROU. SonarQube in action. Shelter Island, New York: Manning, [2014]. ISBN 9781617290954.

[2] CHACON, Scott. Pro Git: [everything you need to know about the Git distibuted source control tool]. 2nd ed. New York, NY: Apress, 2014. ISBN 9781484200773.

[3] GAMMA, Erich. Design patterns: elements of reusable object-oriented software. Reading, Mass.: Addison-Wesley, c1995. ISBN 9780201633610.

[4] LOELIGER, Jon a Matthew MCCULLOUGH. Version control with Git. Second edition. Beijing: O'Reilly, [2012]. ISBN 978-1449316389.

[5] ASP.NET MVC | Microsoft Docs. [online]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/mvc/index.

[6] React – A JavaScript library for building user interfaces. React – A JavaScript library for building user interfaces [online]. Copyright © 2019 Facebook Inc. [cit. 14.04.2019]. Dostupné z: https://reactjs.org/.

[7] Bootstrap · The most popular HTML, CSS, and JS library in the world.. Bootstrap · The most popular HTML, CSS, and JS library in the world. [online]. Dostupné z: https://getbootstrap.com/.

[8] GitHub API v3 | GitHub Developer Guide. GitHub Developer | GitHub Developer Guide [online]. Copyright © 2019 GitHub Inc. All rights reserved. [cit. 14.04.2019]. Dostupné z: https://developer.github.com/v3/.

[9] GitLab API | GitLab. GitLab Documentation [online]. Dostupné z: https://docs.gitlab.com/ee/api/.

[10] Continuous Inspection | SonarQube. Continuous Inspection | SonarQube [online]. Copyright © 2008 [cit. 14.04.2019]. Dostupné z: https://www.sonarqube.org/.

[11] Documentation · TypeScript . TypeScript - JavaScript that scales. [online]. Copyright ©2012 [cit. 14.04.2019]. Dostupné z: https://www.typescriptlang.org/docs/home.html.

[12] webpack. webpack [online]. Dostupné z: https://webpack.js.org/.

[13] The State of the Octoverse reflects on 2018 so far, teamwork across time zones, and 1.1 billion contributions. [online]. Dostupné z: https://octoverse.github.com/2017.

[14] Hackernoon. Dostupné z: https://hackernoon.com/top-10-version-control-systems-4d314cf7adea

11 Prerekvizity

Pro analýzování projektů je potřeba splnit hned několik požadavků.

11.1 .NET Core

- Stažení a instalace posledního .NET Core SDK 2.2, avšak doporučuji i stáhnutí .NET core SDK 2.1.
- Stažení SonarQube scanneru pro podporu analýzy projektů postavených nad platformou .NET Core.

11.2 .NET Framework

- Stažení a instalace posledního .NET Framework 4.7.2.
- Stažení SonarQube scanneru pro podporu analýzy projektů postavených nad platformou .NET Framework.

11.3 Java

11.3.1 Maven

- Instalace Java SDK ve verzi 8u201 nebo pozdější verzi.
- Pro podporu analýzy projektů postavených na buildovacím nástroji Maven je potřeba provést její stažení ze stránky https://maven.apache.org/download.cgi. Při vývoji je použita poslední verze 3.6.0. Následně je nutné vytvořit systémovou cestu do složky aplikace. Například D:\apache-maven-3.6.0\bin
- Následně je potřeba provést úpravu souboru settings.xml, který se nachází ve složce \conf.

```
<settings>
<pluginGroups>
<pluginGroup>org.sonarsource.scanner.maven</pluginGroup>
</pluginGroups>
<profiles>
<profile>
<id>sonar</id>
<activation>
<activeByDefault>true</activeByDefault>
</activation>
<properties>
```

```
<!-- Volitelná URL serveru. Defaultně je nastaveno http://localhost
    :9000 -->
<sonar.host.url>
server url
</sonar.host.url>
</properties>
</profile>
</profiles>
</settings>
```

Výpis 11: Maven settings.xml