FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Synchronizace struktury relační databáze s GIT
Student:	Tomáš Krupička
Vedoucí:	Ing. Jan Matoušek
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Aktualizace a správa struktury databáze mohou být z mnoha důvodů problematické ve větších softwarových projektech. Cílem práce je vyvinout vhodný nástroj na propisování změn ve struktuře databáze z nástroje pro sledování verzí GIT do databáze s ohledem na specifické potřeby větších projektů s velkou provozní zátěží.

1) Analyzujte stávající stav a potřeby konkrétního většího projektu z pohledu synchronizace změn struktury databáze.
2) Proveďte rešerši existujících řešení.
3) Metodami softwarového inženýrství navrhněte vhodné řešení.
4) Řešení dle návrhu implementujte a řádně zdokumentujte.
5) Proveďte ověření Vašeho řešení vhodnými způsoby.
6) Shrňte získané zkušenosti a navrhněte možná vylepšení.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 7. února 2023 v Praze.

Bakalářská práce

SYNCHRONIZACE STRUKTURY RELAČNÍ DATABÁZE S GIT

Tomáš Krupička

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jan Matoušek
18. května 2023

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Tomáš Krupička. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Krupička Tomáš. *Synchronizace struktury relační databáze s GIT*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Poděkování vii

Prohlášení viii

Abstrakt ix

Seznam zkratek x

Úvod 1

Cíle práce 3

1 Představení prostředí 4
1.1 Databáze, Microsoft SQL Server 4
1.1.1 Databáze 4
1.1.2 Relační databáze 4
1.1.3 Microsoft SQL Server 5
1.1.4 T-SQL, procedury, triggery a uživatelské funkce 6
1.2 Verzovací systémy, Git, Azure DevOps 6
1.2.1 Verzovací systémy 6
1.2.2 Distribuované verzovací systémy 8
1.2.3 Git 9
1.2.4 Commity 9
1.2.5 Větve 11
1.2.6 Merge, Rebase 12
1.2.7 Vzdálené repozitáře 13
1.2.8 Azure DevOps 14
1.3 Společnost 15
1.3.1 Vývojářské týmy 15
1.3.2 Vyvíjený software 15
1.3.3 Statistika úprav v databázi 15
1.3.4 Role vývojářů v procesu vývoje 16

2 Současný proces k úpravě 17
2.1 Požadavky 17
2.2 Hotfixy 17
2.3 Standardní úpravy, review cyklus 17
2.4 Nasazení úprav s použitím VCS 18
2.5 Propsání změn do hlavní větve VCS 18
2.6 Nedostatky současného procesu 18
2.7 Diagram současného procesu 19

3 Existující software a konvence pro použití v novém procesu **20**
3.1 Nová synchronizační služba . . . 20
3.2 ServerEventLog . . . 20
3.3 Soubor RELEASE_ORDER . . . 21

4 Definice možných komplikací **22**
4.1 Neaktuálnost zdrojové větve – obecné komplikace . . . 22
4.2 Neaktuálnost zdrojové větve – úpravy tabulek v dočasných souborech . . . 23
4.3 Commitnuté jednorázové skripty/dočasné soubory . . . 23
4.4 Změny v produkční větvi VCS . . . 24
4.5 Více souběžných požadavků k nasazení . . . 24
4.6 Externí změny v databázových objektech . . . 24
4.7 Dlouho trvající nasazování . . . 25

5 Příklady existujících řešení **26**
5.1 Flyway . . . 26
5.2 Entity framework core . . . 27
5.3 Diplomová práce Petra Jindry . . . 27

6 Návrh řešení komplikací **28**
6.1 Více souběžných požadavků k nasazení . . . 28
6.2 Neaktuálnost zdrojové větve – obecné komplikace . . . 29
6.3 Commitnuté jednorázové skripty/dočasné soubory . . . 29
6.4 Neaktuálnost zdrojové větve – úpravy tabulek v dočasných souborech . . . 30
6.5 Změny v produkční větvi VCS . . . 31
6.6 Dlouho trvající nasazování . . . 31
6.7 Externí změny v databázových objektech . . . 32

7 Návrh upraveného vývojového procesu **34**
7.1 Nasazování hotfixů . . . 34
7.2 Nasazování standardních úprav . . . 34
7.3 Zpracování požadavku službou . . . 35
7.4 Zhodnocení výsledků vývojářem . . . 35
7.5 Diagram nového procesu . . . 35

8 Použité technologie pro vývoj **37**

9 Návrh zpracování požadavků **38**
9.1 Jednoduché zpracování požadavků ve frontě . . . 38
9.2 Zpracování více druhů požadavků . . . 39
9.3 Zasílání zpráv . . . 42

10 Popis a implementace komunikace s VCS a databází **43**
10.1 Obecné použití rozhraní . . . 43
10.2 Git . . . 43
10.3 Databáze MSSQL . . . 44

11 Popis procesu nasazování **45**

12 Implementace aplikace, cesta požadavku skrz třídy 48
12.1 Program . . . 48
12.2 IMessageable, IStateMessageable . . . 48
12.3 IReponseTarget a jeho implementace . . . 49
12.4 RestHelpers . . . 49
12.5 Request, MessageableReqeust . . . 49
12.6 EchoRequest, ShutdownRequest, ClearMergeQueueRequesst, MergeRequest . . . 50
12.7 RequestSource, ConsoleRequestSource . . . 50
12.8 IGitConnection, LocalGitConnection . . . 50
12.9 IDatabaseConnection, MssqlDatabaseConnection . . . 51
12.10 IDatabaseConnectionFactory a její implementace . . . 51
12.11 TimoutChecker . . . 51
12.12 Script . . . 51
12.13 DatabaseObject . . . 52
12.14 IDatabaseScriptRunner, ConnectionScriptRunner . . . 52
12.15 IDatabaseScriptRunnerFactory a její implementace . . . 52
12.16 Databázové výjimky . . . 53
12.17 PathDatabaseObjectExtractor . . . 53
12.18 MergeHelpers . . . 53
12.19 IDatabaseMergerParams, IGitMergerParams, MergerParamsRecord . . . 54
12.20 IMerger, GitMerger, DatabaseMerger . . . 54
12.21 IRequestConsumer a jeho implementace . . . 54
12.22 Cesta požadavku skrz třídy . . . 55

13 Testování 57
13.1 Unit testy . . . 57
13.2 Systémové testy . . . 57

14 Dotazník 59

15 Přidání rušení nasazení 62
15.1 Návrh procesu rušení požadavku . . . 62
15.2 Rušení s využitím map . . . 64
15.3 Alternativa s využitím setu . . . 64

Závěr 66

Obsah přiloženého média 72

Seznam obrázků

1.1 Příklad struktury tabulek v relační databázi 5
1.2 Schéma lokálního VCS [8] 7
1.3 Schéma centralizovaného VCS [8] 8
1.4 Schéma distribuovaného VCS [8] 9
1.5 Znázornění verzí ve VCS Git [10] 10
1.6 Oblasti zěmn ve VCS Git [10] 10
1.7 Divergující větve ve VCS Git [11] 11
1.8 Operace fast-forward merge VCS Git [12] 12
1.9 Operace three-way merge ve VCS Git [12] 13

2.1 Současný vývojový proces 19

4.1 Informování o merge konfliktu v Azure DevOps 23
4.2 Znázornění deadlocku při průběhu nasazování 25

6.1 Znázornění konfliktu mezi dvěma větvemi ve VCS Git 29

7.1 Upravený vývojový proces 36

9.1 Diagram využití jedné fronty pro zpracování požadavků 39
9.2 Zpracování požadavků při využití jedné fronty 40
9.3 Diagram využití dvou front pro zpracování požadavků 41
9.4 Zpracování požadavků při využití dvou front 41

11.1 Proces nasazení databázových změn 47

12.1 Průchod třídy MergeRequest aplikací a jejími třídami 56

14.1 Průzkum využití VSC při databázovém vývoji 60
14.2 Průzkum preference způsobu čištění historie větve 60
14.3 Průzkum názorů na potenciální funkce služby 61

15.1 Finální vývojový proces s rušením nasazení 63
15.2 Rušení požadavků s použitím setu 65

Seznam tabulek

1.1 Statistika nasazovaných změn [21] 15

Děkuji své rodině a přátelům za veškerou, zejména mentální, podporu, kterou mi dávali při psaní této práce. Děkuji Společnosti za umožnění pracovat na reálném projektu, který mi dal příležitost se více zaměřit na oblasti vývoje, o které se zajímám, a zjistit o nich něco více. A hlavně děkuji vedoucímu mé práce Ing. Janu Matouškovi za jeho neochvějné vedení bakalářské práce, pevné nervy, dobré připomínky a intenzivní výpomoc při řešení problémů, když se průběh práce zkomplikoval.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 18. května 2023 . . . .

Abstrakt

Cílem této práce je analýza a vypracování řešení způsobu využití verzovacího systému Git pro verzování a synchronizaci změn v databázovém modelu, s důrazem na použití ve velkém podnikovém prostředí. Práce staví na již rozpracovaném řešení pro synchronizaci změn z databáze do Gitu a dalších konvencích a softwaru používaných v cílovém prostředí. Dále navrhuje podobu kompletního systému zahrnující dříve zmíněné řešení a vlastní vypracované řešení pro propisování změn z Gitu do databáze. Výsledné řešení by mělo umožňovat efektivně využívat možností verzovacího systému Git a jeho nadstaveb (např. Azure Repos), a zároveň počítat s případnými komplikacemi, které můžou vyvstat z důvodu vyšší zátěže systému.

Klíčová slova služba, nasazování, verzování změn, struktura databáze, vývojářské týmy, vysoká zátěž, MSSQL, Git, Azure Repos, C#

Abstract

The goal of this thesis is to analyze and create a solution for ways of using the Git versioning system for versioning and synchronization of changes in a database model, focused mainly on usage in a large-scale company environment. This thesis is based on an existing solution for synchronizing changes from a database to Git and other conventions and software already used in target environment. The thesis also designs a system as a whole using said existing solution and own created solution for deploying changes made in Git to a database. The final solution should allow an effective usage of the Git versioning tool and its extensions (like Azure Repos) and also account for possible complications that can arise from a heavier load of the system.

Keywords service, deployment, change versioning, database structure, software development teams, heavy load, MSSQL, Git, Azure Repos, C#

Seznam zkratek

API	Application programming interface
BP	Bakalářská práce
CR	Code review
DB	Databáze
DBMS	Database management system
MR	Merge request
MSSQL	Microsoft SQL Server
ORM	Objektově-relační mapování
PoC	Proof of concept
PR	Pull request
REST	Representational State Transfer
SQL	Structured Query Language
T-SQL	Transact SQL
VCS	Version control system
XML	Extensible Markup Language

Úvod

Databáze jsou v dnešní době velice rozšířeným softwarem v prostředí softwarových projektů, zejména pak komerčních. Jedním z nejrozšířenějších typů databází jsou relační databáze.

Relační databáze se v základu sestávají z tabulek (relace), ve kterých jsou po řádcích uložené záznamy dat a ve sloupcích jednotlivé části (vlastnosti) daného záznamu, z nichž je možné vyčíst data, se kterými následně můžeme v připojeném softwaru pracovat. Dvě tabulky jsou poté spojeny pomocí vazeb mezi sloupci se shodnými hodnotami, čímž je možné pracovat s daty, která spolu souvisí a jsou rozdělena mezi více tabulek.

Kromě základu databáze v podobě tabulek se v podnikovém prostředí mnohdy pracuje i s jinými objekty uloženými v databázi, například procedurami (uložené programy, které je možné uživatelem databáze spustit pomocí jména) nebo triggery (automatické programy, které se spouští při změně dat v tabulce).

Podoba těchto objektů se může měnit díky průběžnému vývoji databáze. Toto se děje v situacích, kdy se daný softwarový projekt s časem vyvíjí, buďto v průběhu vývoje softwaru jakožto celku, nebo při iterativním přidávání změn do již funkčního projektu. Tato práce bude zaměřena právě na velké funkční projekty, jejichž vývoj stále probíhá za jejich provozu.

Nad rozsáhlým projektem komerčního rozsahu zřídkakdy pracuje pouze jeden vývojář, ale často celý tým/několik různých týmů. Ve správném týmu by mělo fungovat hodnocení kódu vývojáři navzájem, které může odhalit problémy/nepřesnosti při přidávání nových funkcionalit a takovýto kód musí být zrevidován před nasazením do produkčního prostředí. Nejen pro tento účel je vhodné mít funkční verzovací nástroj, který nám umožňuje zpětně nahlédnout do předchozích verzí kódu, a porovnávat změny oproti aktuálnímu kódu. Verzovací nástroj usnadňuje vývoj i díky faktu, že na projektu pracuje více vývojářů najednou, a mohou paralelně přidávat změny, které lze od více vývojářů jednoduše slučovat a aplikovat na sebe tak, aby se navzájem nepřepisovaly, ale inkrementálně přidávaly.

Verzovací systémy standardně fungují nad souborovým systémem a porovnávají obsah (většinou) textových souborů, se kterými vývojáři napřímo pracují. U databází je rozdíl, že pro použití ve verzovacím systému je databázový model třeba převádět do textové formy, která se může verzovat, a poté umožnit z textové formy změny propisovat přímo do databáze.

Ve velkém podnikovém prostředí toto obnáší další komplikace, které je třeba vyřešit. Z provozního hlediska je pro toto užití specifické, že databáze je často pod velkou zátěží, se kterou je třeba počítat. V takové situaci nejsou vždy vhodné podmínky pro nasazení vyvinutých změn a nástroj, který má nasazování zjednodušit, či plně zautomatizovat, toto musí respektovat.

Další problém vyvstává ze schopnosti již fungující společnosti a vývojářských týmů takovéto řešení začít používat. Existují mnohé komerční řešení, které řeší stejnou či podobnou problematiku, ale fungují jako celek, jež se při zavedení musí začít používat přesně jak je jeho chod

zamýšlena, a to omezuje značně možnost specifického použití v již daném prostředí. To může být pro některé společnosti komplikací, protože veškeré aktuálně používané postupy při vývoji softwaru musí být změněny a s tím jsou taktéž spojené značné investice a zdržení aktuálních projektů, na kterých je třeba pracovat.

Tato práce má za úkol prozkoumat možná řešení verzování databázových modelů a navrhnout vhodný systém pro použití ve velké, již fungující firmě, která ovšem prochází stálým IT vývojem. Dané řešení by mělo jít co nejjednodušeji zahrnout do již zavedených vývojových procesů a co nejméně omezovat flexibilitu v možnosti používání současných postupů, s umožněním postupného přechodu do stavu využívání zejména/pouze navrhovaného softwarového řešení. Součástí práce bude taktéž funkční zjednodušený software, demonstrující funkčnost navrhnutého řešení.

Práce staví na již rozpracovaném řešení evidování externích změn databázového modelu do verzovacího systému a doplňuje toto o způsoby, jak nové změny již přidané do verzovacího systému propsat do databáze. Celkový navrhnutý systém by měl vyhovovat zmíněným požadavkům, a aby dopady na vývojové týmy byly zároveň co nejmenší. Práce má posloužit právě těmto vývojářům, kterým značně usnadní vývojové procesy a umožní některé repetitivní kroky při vývoji automatizovat.

Účast na tomto projektu mi byla nabídnuta jistou nejmenovanou českou společností. Tuto nabídku jsem přijal, vzhledem k tomu, že sám pracuji v podobném prostředí IT vývoje. Díky tomu patřím do potenciální cílové skupiny, která může tento software využívat. Vývoj s podporou verzovacích systémů je pro mě blízkým tématem a rád umožním rozšířit jejich použití i do jiných oblastí než standardních zdrojových kódů.

Cíle práce

Hlavním cílem této bakalářské práce je navrhnout a vytvořit systém pro nasazování databázových změn, které jsou evidované ve verzovacím systému Git. Toto řešení musí být vhodné pro použití v cílovém prostředí, kterým je velká firma s jejími specifickými potřebami.

Cílem teoretické části je zanalyzovat současné možnosti, kterými jsou již existující nástroje/řešení (komerční i nekomerční) pro synchronizaci nasazování/synchronizaci změn v databázovém modelu. U takovýchto nástrojů je třeba zjistit, zda jsou v cílovém prostředí použitelné, či je třeba zvolit jiné řešení. Dále budou taktéž prozkoumány existující konvence a software fungující v cílovém prostředí, který může být do výsledného řešení zahrnut. K tomu je dalším cílem sepsat jednotlivé požadavky na cílový systém a zjistit názory cílové uživatelské skupiny, jaké funkcionality jsou pro použití důležité v různých týmech.

Protože použití v prostředí s vyšším využitím, než je v menších projektech běžné, s sebou nese dodatečné komplikace, cílem je taktéž tyto komplikace dobře zanalyzovat pro pozdější navrhnutí řešení, které tyto komplikace budou řešit v cílovém systému.

Hlavním cílem praktické části je návrh celkového systému, který bude řešit synchronizaci změn mezi repozitářem Git a databází, což zahrnuje použitý software a taktéž celkový workflow, kterého by se měl cílový uživatel držet pro ideální užívání synchronizační služby. V případě, že žádné existující řešení není vhodné, tak je dalším cílem navrhnout vlastní řešení. Toto řešení bude navazovat na vývoj již existující druhé části synchronizačního systému (řešící opačný směr synchronizace změn z databáze do Gitu).

Výsledný systém bude muset řešit dříve zmíněné komplikace, které byly v rámci teoretické části zanalyzovány, proto bude dalším cílem návrh, jak budou tyto komplikace v cílovém systému vyřešeny, aby nezasáhly do korektní funkčnosti systému (či jak při jejich přítomnosti dopad minimalizovat).

Pokud bylo zvoleno vlastní řešení, cílem praktické části práce je taktéž vyvinout zjednodušenou verzi softwaru, který bude demonstrovat funkčnost navrhnutého řešení dané problematiky (tzv. Proof of concept – PoC). Posledním dílčím cílem bude pro toto PoC navrhnout vhodný způsob, jak otestovat jeho korektní funkčnost, a toto testování provést pro finální validaci.

Systém má umožňovat poloautomaticky (po vyvolání uživatelskou akcí) nasadit změny z gitového repozitáře do databázového systému a informovat uživatele o průběhu a výsledku. Dle potřeb cílové firmy je databázovým systémem Microsoft SQL Server, a repozitář umístěný v rámci webu Azure DevOps (služba Azure Repos). Systém by měl počítat s možností více uživatelů pracujících se službou najednou a s možností nasazovat databázové změny i mimo tento systém. Při těchto situacích nesmí dojít k nevratnému přepisování změn (konkrétně ztrátě informací přepsáním provedených změn).

Kapitola 1
Představení prostředí

1.1 Databáze, Microsoft SQL Server

1.1.1 Databáze

Databáze (v textu používaná zkratka DB) je software, který slouží k ukládání velkého množství obvykle strukturovaných dat a jeho následného použití. Databází existuje více různých druhů jako například relační, objektové, grafové nebo NoSQL databáze.

Různé typy databází se liší zejména v tom, jaká data v nich mohou být a v jaké formě uložena. Různé druhy databází taktéž mohou používat různé způsoby, jak se lze nad uloženými daty dotazovat a jaké výsledky mohou dotazy vracet. Dotazy mohou sloužit k různým účelům. Jmenovitě se jedná například o získání konkrétního záznamu z databáze (jedné uložené položky), získání všech záznamů, které vyhovují zadaným podmínkám nebo agregaci dat a vracení souhrnných informací o nich. Podobnými dotazy je možné data taktéž vkládat, modifikovat a mazat.

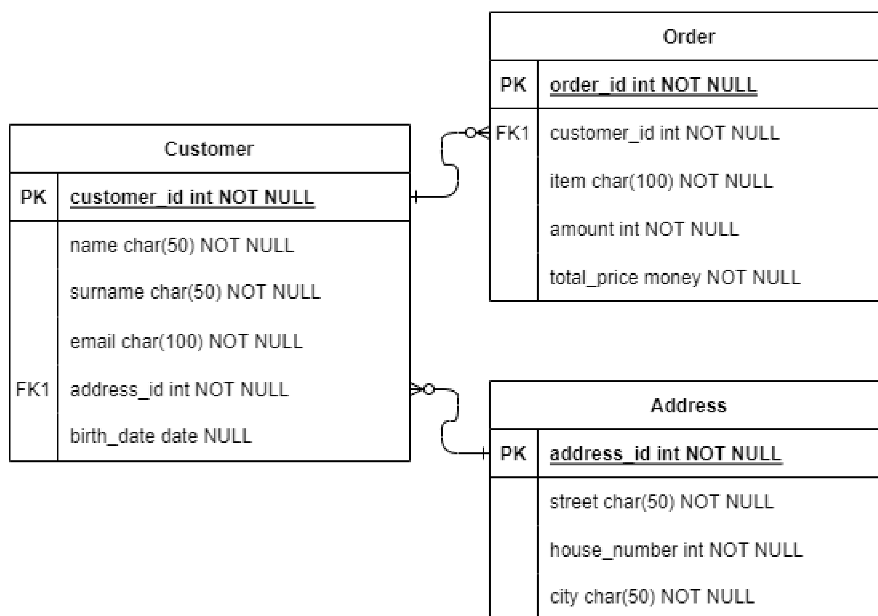
Databáze jsou často používané ve spojení s dalším softwarem, který databázi používá k ukládání dat, se kterými pracuje, a ke spouštění dotazů nad databází pro vykonávání funkcí daného softwaru.

V souvislosti s databázemi se používá taktéž zkratka DBMS (database management systém), která v některých případech znamená to samé, co databáze, někdy ale může být zastřešujícím pojmem pro rozsáhlejší systém.

1.1.2 Relační databáze

Relační databáze jsou typ databází, které ukládají strukturovaná data do tabulek (relací). Tabulky mají jednoznačnou strukturu, ve které jsou po sloupcích definované jednotlivé vlastnosti/atributy záznamů. Sloupce mají svůj datový typ, který určuje, jaký druh hodnoty může být v daném sloupci uložen. Mnohdy je možné nad sloupcem definovat dodatečná omezení, jaké hodnoty mohou být uloženy (například rozsah, či zda hodnota může být nevyplněna). Záznamy jsou poté v tabulce uloženy v řádcích (co řádek to jeden záznam).

Data nemusí být uložena pouze v jedné tabulce, ale mohou být rozdělena do více tabulek. Souvislost mezi záznamy ve více tabulkách identifikujeme pomocí klíčů. Každá tabulka má obvykle svůj klíč (zejména se jedná o primární klíč, tabulka může obsahovat i alternativní klíče), což je skupina sloupců (může být pouze jeden), pro které platí, že kombinace hodnot v těchto sloupcích je pro každý záznam v tabulce jedinečná. Tyto hodnoty mohou být poté použity v záznamu v jiné tabulce (ve sloupcích pro to určených) jako cizí klíč, který odkazuje na záznam v první tabulce a reprezentuje tím vazbu na tento záznam.



Obrázek 1.1 Příklad struktury tabulek v relační databázi

Pro dotazování nad relačními databázemi se používá dotazovací jazyk SQL (Structured query language). Jazyk je význačný svou výřečností – dotaz je podobný přirozenému jazyku a formulace odpovídají tomu, jak by dotaz mohl být v přirozeném jazyce položen. V rámci dotazu definujeme pouze co chceme dotazem získat. Konkrétní postup, který je použit k získání výsledku je poté na databázi. Příkladem základních dotazů jsou SELECT pro získání, INSERT pro vkládání, UPDATE pro úpravu a DELETE pro mazání záznamů. Jednotlivé databáze mohou implementovat použití různých dialektů SQL pro specifické užití s danou databází.

Dalším užitečným databázovým objektem, který lze v relačních databázích používat, je view (pohled). View funguje jako předem definovaný spustitelný SQL dotaz do tabulky či více tabulek. View lze použít k jednoduchému opakovatelnému dotazování do databáze. Mnohdy je taktéž možné omezit práva k přístupu do jednotlivých tabulek pro konkrétního uživatele a umožnit použít pouze view – tím je možné omezit, ke kterým informacím má uživatel přístup, například neumožnit zobrazit citlivé údaje uložené v některých tabulkách.

1.1.3 Microsoft SQL Server

Microsoft SQL Server (v textu používaná zkratka MSSQL) je DBMS vyvíjené Microsoftem pro širokou škálu použití. Jeden DBMS server může zahrnovat více databází, které jsou schopné mezi sebou komunikovat. MSSQL používá k dotazování v DBMS dialekt SQL zvaný Transact SQL (T-SQL).

Provoz serveru je možný na vlastních počítačích/serverech, hostovaný nebo v cloudové verzi zvané Azure SQL, což je varianta SQL Serveru provozovaná v obchodním modelu Platform as a Service, kdy Microsoft databázi hostuje, zajišťuje dostupnost a umožňuje zjednodušení správy a mnohé další výhody. [1]

Databázové objekty jsou uloženy v databázích a dále rozdělovány do skupin pomocí Schema, což je ekvivalent k jmennému prostoru, ke kterému lze například extra definovat přístupová práva jako celku.

1.1.4 T-SQL, procedury, triggery a uživatelské funkce

T-SQL (Transact SQL) je dialekt SQL vyvíjený Microsoftem pro použití v SQL Server DBMS. Dialekt obsahuje konstrukty pro fungování v rozsáhlejším prostředí serveru. Oproti standardnímu SQL, ve kterém se SQL script vykonává po jednotlivých dotazech, T-SQL přidává schopnost procedurálního zpracování kódu jako celku (do kterého jsou SQL dotazy zakomponovány). [2] [3]

Díky procedurálnímu zpracování kódu, T-SQL zavádí další užitečné typy objektů, které se v databázi mohou vyskytovat.

Uložené procedury jsou uloženými spustitelnými skripty, které je možné spouštět/volat podobně jako pohledy pro vykonání konkrétní složitější operace nad databází. Jde taktéž užít omezení přístupu podobně jako u pohledů. Procedury mohou být parametrizované. Zásadní výhoda uložených procedur (a dalších dále zmíněných uložených objektů) je, že při jejich vytvoření/úpravě (a následném použití) dochází k automatickému vygenerování a uložení exekučního plánu, který je používán DBMS pro určení konkrétního postupu vyhodnocení dotazu. Vygenerování exekučního plánu může být u složitých procedur v rozsáhlých databázích časově náročné a tento čas je při dalších voláních procedury ušetřen. [4]

Triggery jsou automaticky spouštěné skripty (podobné jako procedury), které se vážou na konkrétní operaci, při jejímž provedení se skript spustí. Kromě databázových procedur lze použít i kód nahraný na server napsaný s použitím .NET CLR (common language runtime). Existuje více druhů triggerů podle spouštěcí operace, zde budou pro účely bakalářské práce zmíněny DDL a DML triggery.

DDL triggery jsou spouštěny DDL (Data definition language) operacemi. Tyto operace provádějí změnu databázových objektů, jedná se například o vytvoření nebo úpravu tabulky/procedury/etc. Sloužit mohou například k evidenci či kontrole změn, které jsou nad databázovými objekty provedeny. [5]

DML triggery jsou spouštěny DML (Data manipulation language) operacemi. Toto jsou operace, které manipulují s uloženými daty v tabulkách. Jedná se o operace mazání, vkládání a úpravy dat. Triggery mohou sloužit k složitějším kontrolám datové integrity, než je možné se základními omezeními nad sloupcem (například i napříč tabulkami). Zároveň je možné provádět dodatečné automatické změny v návaznosti na ty provedené původní operací. [6]

Uživatelské funkce jsou podobné vestavěným funkcím, které lze použít v SQL dotazech, ale jsou definované uživatelem. Funkce berou parametry a dle parametrů vrací výsledek. Výsledky mohou být různé podle typu funkce. Skalární funkce vrací jednu hodnotu (pro použití ve výrazech), zatímco tabulková funkce vrací tabulkový dataset (pro použití ve zdrojích dat dotazu podobně jako tabulka nebo pohled). [7]

1.2 Verzovací systémy, Git, Azure DevOps

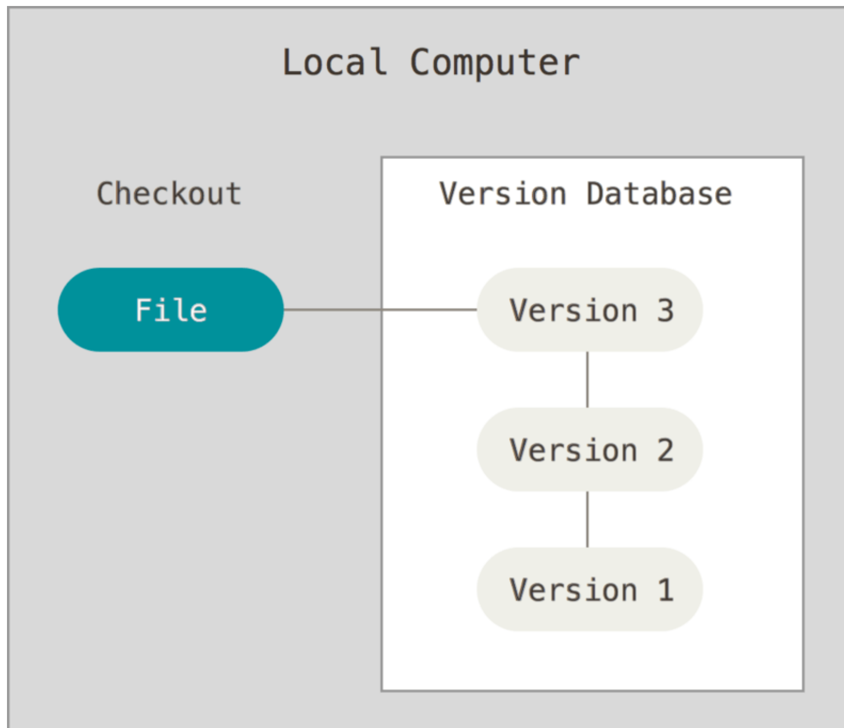
1.2.1 Verzovací systémy

Systém správy verzí (Version control system, v textu používaná zkratka VCS) je software, který umožňuje evidovat verze souborů v průběhu času. Umožňuje tím dělat zálohy starších verzí, což se může hodit v různých situacích. Můžeme se například chtít zpětně podívat na starší verzi souboru nebo si ji obnovit. Dále můžeme chtít zjistit, jaké změny byly v souboru provedeny (porovnat jednotlivé verze), či navíc i kdo dané změny provedl. Toto se může hodit například v případě, že software přestane fungovat a my chceme zjistit, jaká změna (v době, kdy začalo docházet k chybám) byla provedena. Obvyklým použitím VCS je verzování zdrojových kódu, ale jde použít na jakékoliv soubory.

VCS se historicky vyvíjely a jdou klasifikovat do 3 typů, které se liší zejména tím, kde jsou data uložena a co je na daném místě uloženo (případně v jaké formě). [8]

Lokální VSC jsou prvními verzemi VCS, které vznikaly jako alternativa k obyčejnému zálohování více verzí v samostatných složkách. Lokální VCS existuje pouze na počítači uživatele

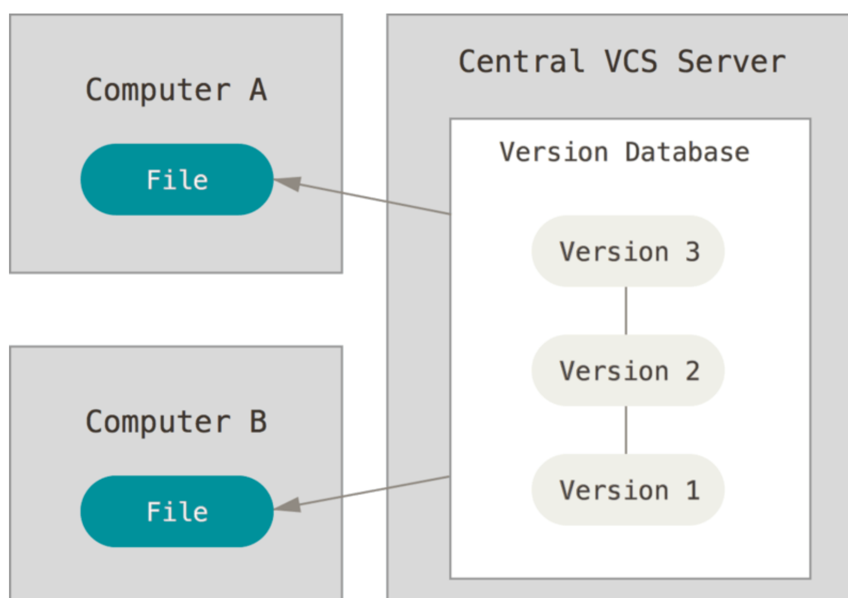
a eviduje verze lokálních souborů. Příkladem lokálního VCS je systém RCS, který je ještě dnes distribuován s mnohými počítači. Tento systém uchovává změny ve formě rozdílů mezi verzemi (patche), které je možné poskládat a získat libovolnou verzi souboru v čase. [8]



Obrázek 1.2 Schéma lokálního VCS [8]

Centralizované VCS řeší problém spolupráce více uživatelů (např. vývojářů) na jednom projektu, ve kterém se sdílí soubory. Jednotlivé verze souborů jsou uloženy na serveru a uživatelé si pouze stahují do svého počítače ze serveru konkrétní verzi souboru, na které pracují, a na server zasílají zpět provedené změny. Toto umožňuje základní kolaboraci, kdy změny, které provede jeden uživatel, vidí i ostatní. Zároveň to přináší i výhody v jednodušší správě verzovacího systému, kdy je třeba nastavovat pouze verzovací databázi na serveru oproti více instancím VCS na počítačích uživatelů. [8]

Tento postup s sebou nese ovšem zásadní nevýhodu, kterou je uložení verzí pouze na jednom místě. Problém nastane, pokud server, na kterém VSC ukládá verze vypadne, uživatelé nemohou na souborech spolupracovat, či mají k dispozici pouze aktuálně stáhnutou verzi souboru, ve které nemohou ukládat změny. Navíc, v případě, že dojde rozbití disku serveru, veškerá historie změn je nenávratně ztracena a jediné dostupné verze k obnově jsou ty, které mají na svých počítačích uživatelé aktuálně stažené. Tato nevýhoda je taktéž přítomna u lokálního VSC, kde ovšem dojde ke kompletní ztrátě dat. [8]



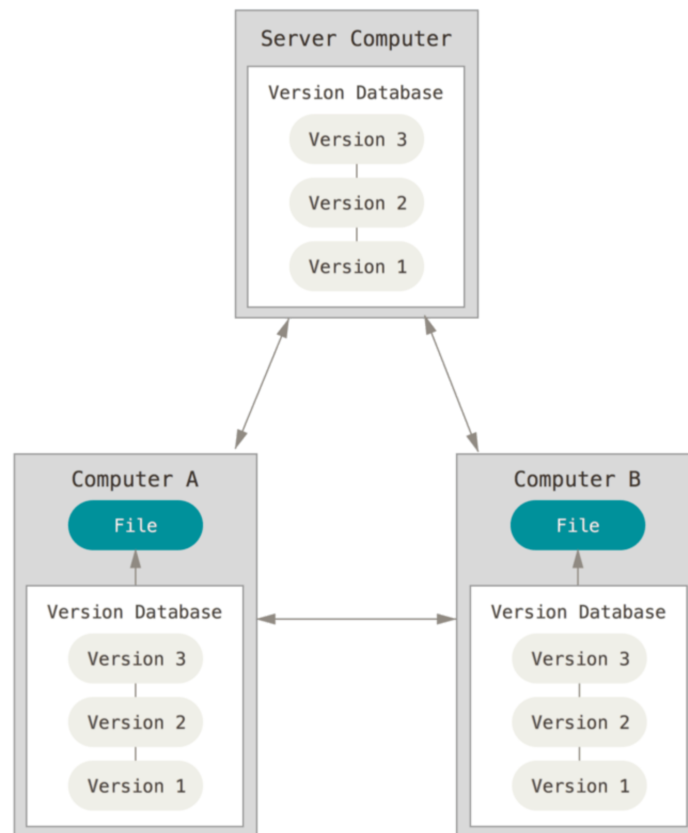
Obrázek 1.3 Schéma centralizovaného VCS [8]

Příkladem těchto systémů jsou systémy CVS, Subversion a Perforce. [8]

1.2.2 Distribuované verzovací systémy

Výše zmíněné problémy řeší distribuované VCS. Klient na straně uživatele nestahuje do lokálního počítače pouze jednu verzi souboru, na které může uživatel pracovat, ale zrcadlí celý vzdálený repozitář (či jeho části) a jeho kompletní historii změn. Toto platí pro každého uživatele. Server poté slouží jako reference pro toto zrcadlení, na kterou uživatelé zasílají svoje změny pro užití dalšími uživateli. [8]

Výhoda tohoto postupu je, že při výpadku serveru může každý uživatel plně pracovat se všemi verzemi všech souborů v repozitáři lokálně, komunikace se serverem totiž probíhá pouze při stahování či nahrávání nových verzi repozitáře. Zároveň, pokud dojde ke ztrátě dat na straně serveru, veškerou historii je možné obnovit od kteréhokoliv uživatele, díky faktu, že lokální kopie je kompletní zálohou všech dat původně na serveru uložených. [8]



Obrázek 1.4 Schéma distribuovaného VCS [8]

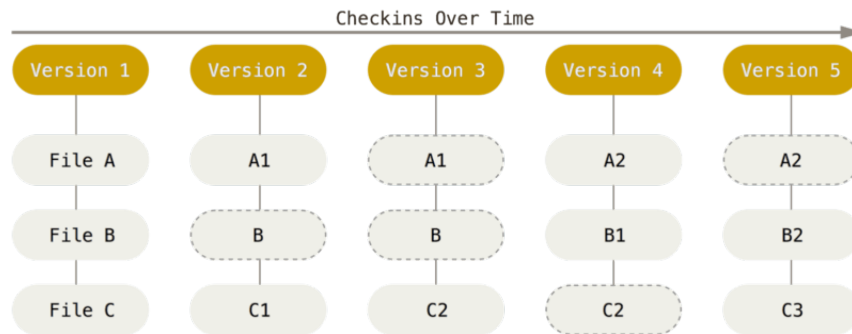
Git patří mezi distribuované VCS, dalšími příklady jsou například Mercurial, Bazaar nebo Darcs. [8]

1.2.3 Git

VCS Git vznikl v průběhu vývoje Linuxového jádra jako open source alternativa k nově zkomerčněnému VCS BitKeeper. Díky cílovému použití při vývoji Linuxového jádra byl Git navržen tak, aby splňoval požadavky jako je rychlost, jednoduchý návrh, podpora nelineárního vývoje ve velkém množství větví, distribuovatelnost a možnost použití pro velké projekty. [9]

1.2.4 Commity

Git se od jiných VCS odlišuje tím, jak eviduje historii změn a soubory. Většina ostatních VCS se primárně zaměřuje na soubory, a verze chápe jako seznamy změn v souborech, jejichž historie je uložená pomocí rozdílů mezi jednotlivými verzemi. Uvažování o repozitáři lze nazvat jako historii všech souborů. Oproti tomu se Git zaměřuje primárně na commity (verze/revize), kde každý commit má uložený kompletní stav repozitáře se všemi soubory v dané verzi. Pokud se soubor nezměnil, je pouze použit odkaz již na existující verzi, podobně při malých změnách dochází k výrazným optimalizacím pro ukládání verze souboru. Takto Git uvažuje o historii repozitáře jako o sérii snímků celého repozitáře (všech souborů). [10]

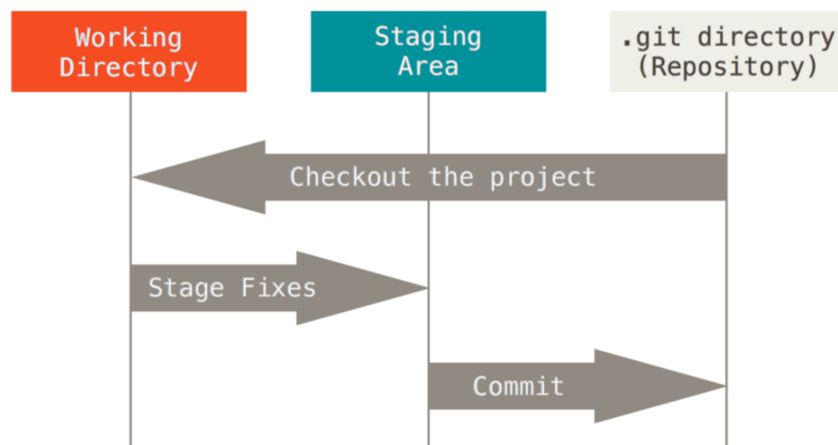


Obrázek 1.5 Znázornění verzí ve VCS Git [10]

„*Většina operací v systému Git vyžaduje ke své činnosti pouze lokální soubory a zdroje. Obecně platí, že informace z jiných počítačů v síti nejsou potřebné.*“ Toto je umožněno díky již dříve zmíněným vlastnostem distribuovaných VCS. Protože není potřeba čekat na odezvu serveru, většina operací, které se v rámci VCS Git provádí, jsou velice rychlé či téměř okamžitě vykonané. Zároveň je díky tomu možnost téměř plně pracovat s repozitářem bez připojení k síti, a tedy i vzdálenému serveru. [10]

V rámci VCS Git existují tři oblasti, kde jsou změny při práci uchovávány. Okamžitě poté, co je soubor vytvořen/upraven, Git nemá změny nikde zaevidované. Tyto změny jsou pouze v pracováním adresáři (working directory). Díky porovnání s ostatními oblastmi jdou tyto změny zobrazit. Další krok je přidání do oblasti připravených změn (staging area/index), například pomocí příkazu „git add“. Tím je VCS dán signál, že má změny začít evidovat a tím je připravit na trvalé zapsání do repozitáře. Index je pouze dočasná oblast pro uložení změn, tyto změny nejsou asociovány s žádným commitem. Následný příkaz k vytvoření commitu „git commit“ vezme všechny změny z indexu a z nich vytvoří nový commit, který je trvale v repozitáři VCS evidován. [10]

Mezi commity můžeme následně přecházet pomocí checkoutu příkazem „git checkout“, kdy se verze souborů z vybraného commitu nahrají do working directory a v případě checkoutu celého commitu VCS přepne aktuálně zobrazovaný commit na dříve vybraný. [10]



Obrázek 1.6 Oblasti zěmn ve VCS Git [10]

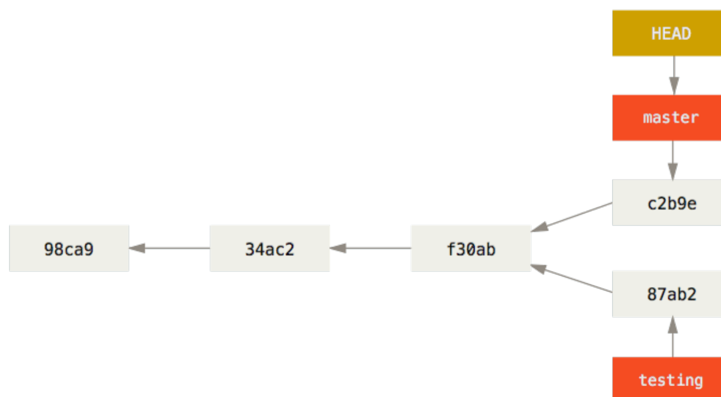
1.2.5 Větve

Větve jsou ve VCS způsobem, jakým lze evidovat více alternativních historií souboru/repozitáře. Větve mají mnohdy nějakou společnou část historie, přičemž se změny v určitém bodě divergují. Větve jsou dalším aspektem, ve kterém se VCS Git liší od mnohých jiných VCS. [11]

Ve VCS Git je větev reprezentována jako pojmenovaný (jménem větve) ukazatel na commit. Při vytvoření repozitáře se vytvoří zároveň i hlavní větev, která po vytvoření prvního commitu ukazuje na tento commit (daný commit je vrcholem větve). Dále ve VCS Git existuje dodatečný ukazatel HEAD, který ukazuje na aktuálně prohlíženou větev, či v některých situacích, pokud přejdeme mimo větev na samostatný commit, může ukazovat pouze na commit. Pro účely vysvětlování větví budeme předpokládat, že HEAD nám ukazuje na větev, nikoliv na commit. [11]

Ve chvíli, když vytvoříme nový commit v současné větvi, ukazatel větve, na kterou HEAD ukazuje se přesune na nový commit (HEAD stále ukazuje na tuto větev). Tento nový commit má za předka původní commit. Řetězec předků takto tvoří historii větve. [11]

Další operací, kterou můžeme provést, je vytvoření nové větve (příkazem „git branch <název>“). Tím se vytvoří nový ukazatel ukazující na stejný commit, na který ukazuje současná větev. Protože jediná operace, ke které došlo, je pouze vytvoření ukazatele, vytváření nových větví je ve VCS Git velice rychlé (téměř okamžité). Mezi větvemi se můžeme přesouvat pomocí dříve zmíněného checkoutu, který změní, kam nám HEAD ukazuje. Pomocí přepínání mezi větvemi můžeme následně přidat nové commity do obou větví. Oba nové commity mají společného předka, ale historie větví se divergovala, protože se jedná o různé commity. Těmito operacemi jsme vytvořili dříve zmiňovaný cílový stav alternativních historií souborů změněných v rámci commitů. [11]



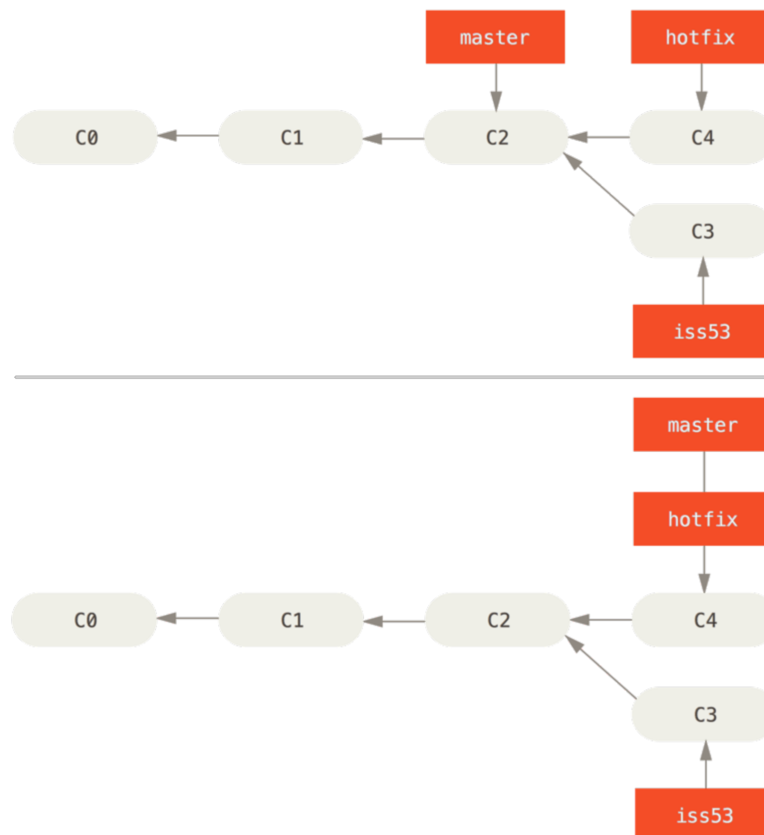
Obrázek 1.7 Divergující větve ve VCS Git [11]

Tento postup je mnohem rychlejší než u alternativních VCS, ve kterých může docházet ke kopírování celého repozitáře do nových složek, což v závislosti na velikosti repozitáře může zabrat dlouhou dobu. [11]

1.2.6 Merge, Rebase

Ve stavu, kdy v repozitáři máme dvě divergující větve (užívaný příklad, produkční a vývojovou), v nějakou chvíli nastane čas, kdy budeme chtít změny z vývojové větve propsat do větve produkční. K tomu lze použít operace merge (slučování větví). Pro slučování potřebujeme dvě větve, které mají rozdílnou historii. Zdrojová větev je ta, ze které budeme změny brát a cílová větev je ta, do které budeme změny propisovat. Protože v rámci BP bude řešeno slučování změn z vývojové do produkční větve, budou používané pojmy ve dvojicích produkční – cílová a vývojová – zdrojová zaměnitelné a konkrétní pojem bude použit dle kontextu. [12]

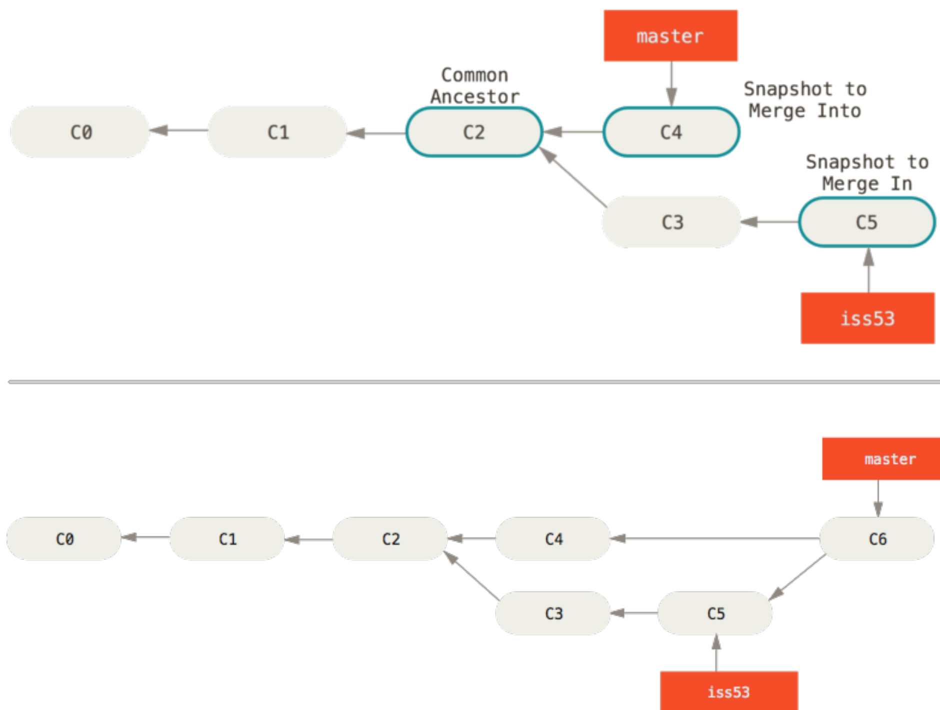
Pokud historie zdrojové větve obsahuje vrchol větve cílové, Git použije ke sloučení operaci fast-forward (rychlé posunutí dopředu). V této situaci je výsledkem operace pouhé přidání všech nových commitů z větve zdrojové, tudíž merge proběhne pouhým přesunutím ukazatele cílové větve na vrchol větve zdrojové. [12]



Obrázek 1.8 Operace fast-forward merge VCS Git [12]

Pokud toto neplatí, dojde k více používanému třícestnému slučování (three-way merge). Protože jsou historie větví rozdílné, je třeba nejprve zjistit, jaký commit mají větve společný. Tento commit se v kontextu slučování nazývá merge base. Jedná se o commit, který je obsažen v historii obou větví neboli společný předek. Nejlepší merge base pro použití je takový, pro který platí, že žádný pro žádný z ostatních potenciálních commitů není tranzitivně předkem. Tento commit je použit jako reference k určení, které změny byly provedeny na jednotlivých větvích v divergující historii a tedy musí být sloučeny. Při třícestném slučování jsou následně aplikovány změny, které byly provedeny ve zdrojové větvi, ale nebyly provedeny v cílové větvi. Výsledné

změny jsou reprezentovány novým commitem (a ten je výsledkem sloučení), na který se přesune ukazatel cílové větve. [12]



Obrázek 1.9 Operace three-way merge ve VCS Git [12]

Při třícestném slučování může dojít k merge konfliktům. To je situace, kdy část souboru byla změněna v obou větvích a VCS nedokáže automaticky určit, jak změny na sebe aplikovat. V takové situaci je do jinak automatického průběhu vtažen uživatel, který musí sloučení daných změn provést ručně, případně operaci zrušit. [12]

Alternativou k odstranění divergující historie je rebase (přeskládání) nad větví, který změní historii jedné větve tak, aby byla založena na vrcholu druhé větvě. To vyústí v čistší historii větve než při použití merge, ale za cenu změny historie – jsou vytvořeny nové commity na místě starých, což může způsobit problémy, pokud danou větvi používá více vývojářů a mělo by být ideálně použito tedy pouze na vývojových větvích používaných jedním vývojářem. [13]

Tento postup zde nebude do detailu rozebrán, ale jeho použití je zmíněno v textu BP.

1.2.7 Vzdálené repozitáře

Kolaborace ve VCS Git je umožněna pomocí vzdáleného repozitáře (remote). Vzdálený repozitář je kopie repozitáře umístěná v síti, ke které mají vývojáři přístup přes svoji lokální instalaci VCS, případně webové rozhraní vzdáleného repozitáře. V rámci lokálního repozitáře je možné

mít nakonfigurované libovolné množství repozitářů vzdálených, se kterými je možné pracovat. [14]

Nové změny ve vzdáleném repozitáři je možné si vyzvednou příkazem „git fetch". Ten nám stáhne nové větve/commity z větví ve vzdáleném repozitáři a vytvoří (případně přesune) ukazatel, který je použit pro větev ve vzdáleném repozitáři, čímž získáme vzdálenou větev. Oproti lokálním větvím nejdou vzdálené větve přímo upravovat, ale jdou použít jako reference k založení nové lokální větve sledující vzdálenou či sloučení změn do jedné z našich lokálních větví. Operací „git push" můžeme odeslat naši lokální větev do vzdáleného repozitáře, kde se vytvoří nová vzdálená větev a naše lokální bude nastavena na sledování dané vzdálené větve. [14]

Ve chvíli, kdy existuje lokální větev sledující vzdálenou, můžeme provádět operace aktualizující stav lokální či vzdálené větve. Pokud jsou na vzdálené větvi nové změny, tyto nové změny můžeme do lokální větve začlenit pomocí příkazu „git pull", který nové změny stáhne. Pull je spojením operací fetch a merge volaných po sobě. To znamená, že v případě, že se historie lokální větve rozchází se vzdálenou, dojde ke sloučení změn mezi lokální a vzdálenou větví (kde lokální je cílová a vzdálená je zdrojová). [15]

Opačným směrem pro nahrání nových změn provedených v lokální větvi do vzdálené větve využijeme příkazu „git push", který nové změny do vzdáleného repozitáře propíše. Pro korektní vykonání příkazu je třeba nemít rozdílnou historii oproti vzdálenému repozitáři. To můžeme opravit dříve zmíněným provedením stažení změn, které taktéž zahrnuje případné sloučení rozcházejících se historií. [15]

Dále je možné užít vynuceného přepsání pomocí „git push -f", které přepíše vzdálenou historii naší lokální. Možné použití je například pokud jsme na lokální větvi měnili historii kvůli zpětné opravě. Toto ovšem vytváří stejné komplikace jako rebase, mělo by to být tedy prováděno pouze na větvích, které používá pouze daný vývojář.

1.2.8 Azure DevOps

Azure DevOps je kolaborativní platforma vyvíjená Microsoftem, která má za účel umožnit jednoduchou kompletní spolupráci členů týmu při vývoji softwaru zastřešenou pod jednou platformou. Tato platforma nabízí nástroje, které jdou použít při různých částech procesu vývoje od tvorby zadání přes samotný vývoj až k deploymentu hotové aplikace. Platforma jde provozovat jako služba hostovaná v cloudu Microsoftem nebo na vlastních serverech zákazníka. [16]

Platforma v základu obsahuje různé služby, jmenovitě Azure Boards, Azure Repos, Azure Pipelines, Azure Test Plans a Azure Artifacts, které jde doplnit o dodatečné rozšíření k propojení s jinými populárními vývojovými produkty. V rámci BP budou využity služby Repos a Pipelines. [16]

Azure Repos je služba na platformě Azure DevOps zastřešující nástroje pro práci s VCS, jmenovitě systémy Git a TFVC. Služba funguje jako vzdálený repozitář VCS a přidává dodatečné funkce pro týmovou kolaboraci při vývoji jako jsou Pull requesty (použitá zkratka PR – požadavky na merge dvou větví, například vývojové do produkční; v jiných webových rozhraní VCS nazývané Merge requesty, s použitou zkratkou MR, tyto názvy a zkratky jsou zaměnitelné), v rámci kterých je možné jednoduše provádět revize změn a zhodnocení kódu jinými vývojáři s vyššími právy při Code review před tím, než jsou změny sloučeny. [17]

Azure pipelines je služba na platformě Azure DevOps, která umožňuje provádět automatické či poloautomatické operace nad vyvíjenou aplikací. Kvůli tomu úzce spolupracuje se službou Azure Repos, ze které získává zdrojový kód aplikace, se kterou provádí různé operace. Operace mohou zahrnovat například slučování změn, testování kódu, či sestavení a nasazení aplikace na produkční stroje. [18]

Pipeline může být spuštěna automaticky vyvolanou událostí, což může být například nahrání nové verze kódu do repozitáře nebo založení pull requestu. [19] Druhý způsob je ruční spuštění, při kterém je možné i například zadávat parametry pro danou operaci, jako například větev,

nad kterou se má operace provést, nebo image, který se má použít. [20] Všechny tyto možnosti a proces průběhu pipeline jsou plně modifikovatelné vývojářem, který danou pipeline navrhuje.

1.3 Společnost

Dotčenou společností je firma působící v Česku, pro jejíž užití bude vyvinut software v rámci této bakalářské práce. Z důvodu zachování důvěrnosti je v rámci práce v textu použito označení "Společnost".

1.3.1 Vývojářské týmy

Kromě fungování na trhu vyvíjí Společnost software pro vlastní účely, k čemuž má IT oddělení se zhruba 150 vývojáři. [21] Tito vývojáři jsou členěni do týmů, kdy každý tým spravuje nějakou oblast působení Společnosti, pro kterou vyvíjí/vylepšuje software.

Tým obvykle zahrnuje řadové vývojáře, dále seniorní vývojáře (a technologického leadera) s vyššími právy, kteří korigují správné směřování vývoje řadových vývojářů. Dále v týmu figurují testeři pro testování nových funkcionalit, analytici, kteří zadávají požadavky k vypracování dle potřeb Společnosti a projektový manažer, který spravuje organizaci požadavků, jejich prioritizaci a vypracování.

1.3.2 Vyvíjený software

Původní softwarová architektura ve Společnosti je monolitická, využívající programů komunikujících s databázovými servery, kdy velká část procesních operací je řešena pomocí uložených databázových procedur.

Nově Společnost zavádí architekturu kontejnerových mikroslužeb v rámci systému Kubernetes, neboli vývoj více samostatně funkčních bloků v podobě oddělenějších systémů, které spolu komunikují.

Jako datové úložiště využívá společnost několika databázových serverů systému MSSQL, které lze rozdělit do skupin produkčních serverů s replikací, a vývojových serverů pro užití k vývoji IT oddělením.

Společnost dále využívá různé vzdálené repozitáře git včetně Azure DevOps. V rámci této BP se bude pracovat výhradně s Azure DevOps repozitářem, na který je vývoj nového softwaru směřován.

1.3.3 Statistika úprav v databázi

Pro referenci k množství úprav, které na produkčních serverech probíhají, byl vybrán jeden z hlavních produkčních serverů. Na tomto serveru došlo za týden a za měsíc k tomuto množství úprav (vybrané typy objektů)

Tabulka 1.1 Statistika nasazovaných změn [21]

typ objektu	týden	měsíc
Pohledy	nižší desítky	nižší stovky
Triggery	jednotky	nižší desítky
Procedury	nižší stovky	vyšší stovky
Funkce	nižší desítky	nižší stovky
Tabulky + indexy	desítky	vyšší stovky

Dle těchto statistik je poznat, že nejčastěji upravovanými objekty ve Společnosti jsou procedury. U procedur došlo v rámci měsíční statistiky k nerovnoměrnému nárůstu úprav v tabulkách oproti týdenní statistice, což mohlo být způsobeno rozsáhlejšími úpravami v rámci některého požadavku (či absenci změn ve sledovaném týdnu), protože v jiných měsících se počet změn v tabulkách pohyboval spíš kolem nižších stovek, zatímco počty úprav ve zbytku objektů byl podobný, budeme to tedy považovat za statistickou chybu.

1.3.4 Role vývojářů v procesu vývoje

Vývojáři dostávají od analytiků požadavky k vypracování, kterým je určována priorita projektovým manažerem. Vývojář může pracovat na více požadavcích současně, zvláště, když u některých čeká na odezvu od jiného zaměstnance, se kterým komunikuje například kvůli upřesnění informací.

Vývoj změn je mnohdy řešen na dvou úrovních, úpravy v softwaru a úpravy v databázi. Tyto úpravy jsou evidovány v Gitovém repozitáři, kdy existuje zvlášť repozitář pro software (různé programy mohou mít různé repozitáře) a pro databázové objekty. Některé týmy (a jejich vývojáři) databázový repozitář ke své práci využívají, některé ne. Pro akutní opravy (hotfixy) v produkční prostředí není ani databázový repozitář užit, protože přidává procesní zdržení (bez toho může být úprava vytvořena a okamžitě nasazena).

Po vypracování úpravy tester testuje aplikaci, zda splňuje požadavky stanovené v zadání. Pokud nesplňuje, požadavek je vrácen zpět vývojáři, aby nedostatky opravil. Následně, pokud je otestování v pořádku, seniorní vývojář provede řadovému vývojáři po dokončení vývoje zhodnocení kvality a korektnosti kódu – code review (v textu používaná zkratka CR). Pokud CR projde, seniorní vývojář zajistí nasazení změn.

Kapitola 2
Současný proces k úpravě

Pro korektní analýzu toho, jak nově navržený proces zapadne do již fungující firmy je třeba si definovat, jak rámcově funguje současný vývojový proces a jaké má tento proces případné nedostatky, které můžeme naším novým procesem vylepšit. Tato analýza se bude zaměřovat na části vývoje, které souvisí s užitím databáze a VCS, což jsou hlavní stěžejními systémy, které budou novým systémem ovlivňovány.

2.1 Požadavky

Na počátku vývoje je vždy požadavek na novou funkci v softwaru nebo na opravu chyby, která se v produkčním prostředí projevila. Tento požadavek může vyžadovat úpravy v databázi, vyvíjeném softwaru nebo obojím. Případy, kdy k úpravám v databázi nedochází nebudeme řešit, protože nijak nefigurují v námi navrhovaném systému. Tímto omezením budeme předpokládat, že všechny požadavky vyžadují úpravu v databázi.

2.2 Hotfixy

V případě, že se jedná o opravu chyby, může jít o kritickou chybu, která vyžaduje okamžitou opravu (hotfix) bez jakéhokoliv zdržování. Proces se v tu chvíli řídí zrychleným CR. V tom případě vývojář vytvoří úpravu, seniorní vývojář mu zrychleně opravu zhodnotí v podobě zrychleného CR (často jsou tyto informace o vytvoření úpravy předány pouze přes interně používanou komunikační platformu, hovor nebo osobní kontakt) a v případě korektnosti úpravy ji i rovnou ručně nasadí.

Pokud oprava není korektní, vývojář je požádán o opravu úpravy. Připomínky nemusí být validní, v takovém případě o tom vývojář informuje a seniorní vývojář své připomínky zreviduje. Pokud jsou připomínky validní, cyklus se opakuje. V některých situacích může být seniorní vývojář i v pozici vývojáře, který tuto opravu vytváří – zde tuto situaci zahrneme do již definovaného cyklu tak, že proces stále platí, jen si CR provádí sám sobě kontrolou svého kódu.

2.3 Standardní úpravy, review cyklus

Pro ostatní úpravy si definujeme podobný cyklus, který bude ovšem využívat VCS k evidenci změn a následnému detailnímu CR. Vývojář si stáhne aktuální verzi skriptů, na kterých úpravu vypracuje, čímž je zmenšená šance, že připravené skripty budou neaktuální a změna přepíše

úpravu vytvořenou jiným vývojářem. Úpravy pro vlastní otestování nasadí na vývojový databázový server, následně je zaeviduje do VCS v podobě nových commitů a odešle je do vzdáleného repozitáře. [22]

V tento moment se do procesu vkládá tester, který úpravy na vývojové databázi otestuje. Protože se tester ovšem nijak nedostává do přímého kontaktu s úpravami (kódem úpravy) ani v DB ani ve VCS, v rámci procesu tento krok nebudeme modelovat a pro účely korektnosti namodelovaného procesu definujeme testování jako součást vývoje úprav.

Po dokončení vývoje (či konkrétně první iterace vývoje) vytvoří vývojář ve webovém rozhraní nový MR pro provedení CR seniorním vývojářem. V případě, že seniorní vývojář vytváří úpravy, CR zpravidla provádí jiný seniorní vývojář z důvodu kontroly kódu jinou osobou. CR je provedeno kontrolou kódu v rámci webového rozhraní, kde jde vidět i případnou neaktuálnost upraveného scriptu (v případě, že od vytvoření commitů došlo k úpravám jiným vývojářem). Pokud existují nedostatky, seniorní vývojář je popíše pomocí komentářů ke konkrétním částem kódu. Vývojář v tu chvíli dostane notifikaci o přidaných komentářích k jeho kódu. K těmto komentářům se může vyjádřit s případnými připomínkami (pokud nejsou komentáře srozumitelné či validní). Jakmile dojde ke shodě ve validitě nedostatků, stejně jako u hotfixů se vrací zodpovědnost na vývojáře, který musí svoji úpravu opravit. V tomto případě vypracuje novou verzi, kterou opět nahraje do VCS, čímž začíná cyklus CR a úprav znovu. Jakmile jsou úpravy vývojáře v pořádku, v procesu se přesouváme k nasazení úprav seniorním vývojářem. [22]

2.4 Nasazení úprav s použitím VCS

Nejprve dojde k uzavření MR, protože jeho užití v procesu již došlo ke konci. Důležitým faktem je to, že MR je pouze uzavřen, ale merge není reálně proveden, pro propsání změn do hlavní větve VCS dochází jiným postupem. K následnému nasazení úprav může využít seniorní vývojář připravený skript v repozitáři, který otevře změněné soubory v databázovém vývojovém prostředí. Seniorní vývojář poté ručně spouští všechny otevřené scripty pro jejich nasazení. Protože se jedná o ruční spouštění, může omylem některé scripty vynechat, proto je součástí procesu nasazení i kontrola, zda změny byly nasazeny, přes vývojáři dostupné nástroje. [22]

2.5 Propsání změn do hlavní větve VCS

MR je uzavřen bez sloučení změn do hlavní větve, k propsání změn musí dojít jinak. Toto má za úkol aktuálně funkční synchronizační služba, která periodicky sbírá snapshoty (snímky) struktury databáze a pokud došlo k nějakým změnám oproti poslednímu commitu v repozitáři, služba vytvoří nový commit, ve kterém jsou provedené změny od poslední kontroly obsaženy. Toto probíhá pro všechny změny provedené na databázi, ať byly vyvinuty/nasazeny s použitím VCS či ne.

2.6 Nedostatky současného procesu

Proces vývoje a zhodnocení kvality kódu je navržen rozumně, na tom není co měnit. Problematickou částí je nasazování a následná evidence změn v hlavní větvi repozitáře.

Ruční nasazování je část procesu, která zbytečně bere čas seniornímu vývojáři, protože musí ručně proklikávat skripty k jejich nasazení, a poté ještě provádět kontrolu. Kromě zdržování vývojáře od důležitějších činností je toto činnost, které podléhá nebezpečí výskytu lidské chyby. Vývojář může vynechat nasazení některého scriptu nebo omylem přepsat nasazování jiného vývojáře, kterého si v průběhu procesu nevšimne. Toto vyřeší automatizace procesu nasazování.

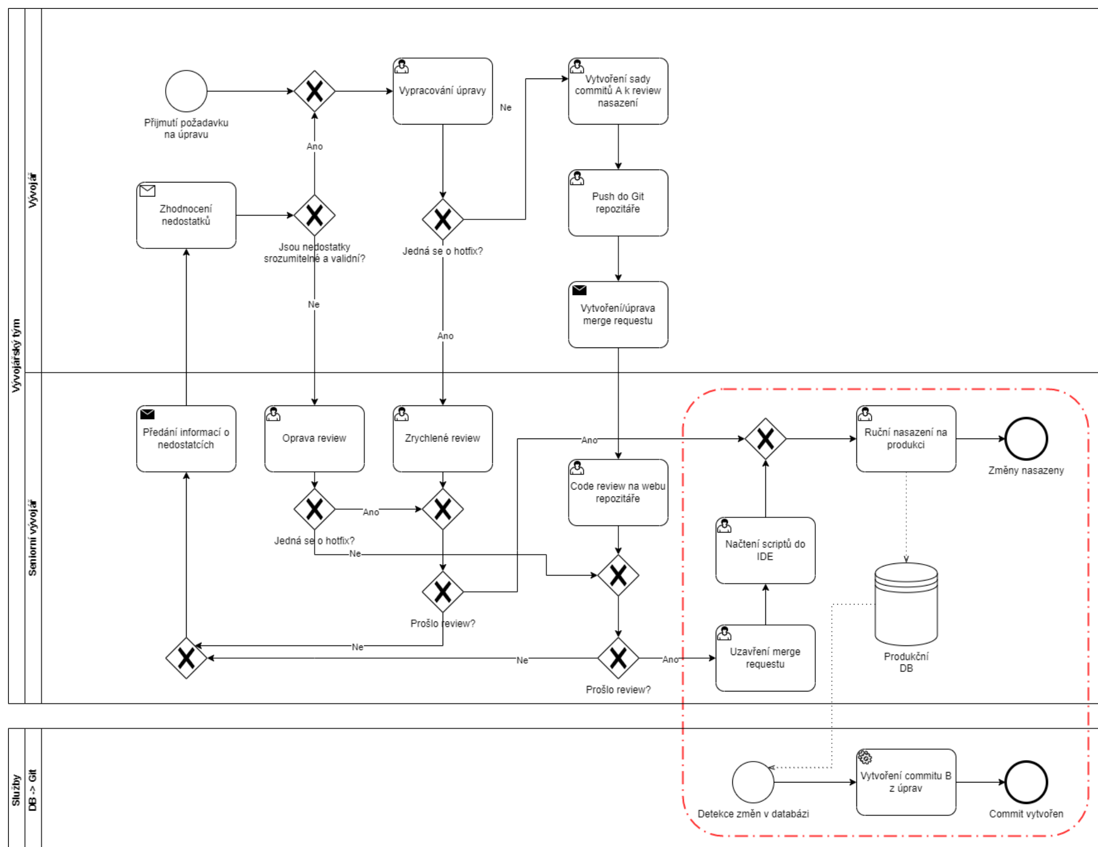
Dále současné automatické propsání změn z databáze do VCS má dvě nevýhody. První je způsob zpracování formou snapshotů, kdy v rámci jednoho snapshotu nelze rozpoznat změny

provedené jako součást více požadavků/více nasazení, pokud proběhly během jednoho kontrolního okna. Jediný zdroj, který je možné k dohledání použít jsou vývojové větve v repozitáři, které ovšem nemají žádnou vazbu na commity v hlavní větvi a jejich dohledání je proto komplikovanější.

Druhý problém, který s tímto faktem souvisí, je, že nově vytvořené commity synchronizační službou jsou jinými commity, než ty, které byly použity v rámci větve v MR. Tyto commity mohly obsahovat dodatečné informace ke změnám či jejich organizaci do více sousledných commitů. Toto je při použití současného postupu ztraceno. Oba problémy budou minimalizovány (částečným) slučováním změn z původní vývojové větve.

2.7 Diagram současného procesu

Níže je přiložen zjednodušený diagram, který modeluje proces popsaný v této kapitole. V tomto diagramu je část procesu s nedostatky vizualizována červeným orámováním.



Obrázek 2.1 Současný vývojový proces

Kapitola 3

Existující software a konvence pro použití v novém procesu

3.1 Nová synchronizační služba

Aktuálně je vyvíjena nová synchronizační služba [23], která v budoucnu nahradí dříve zmíněnou. Rozdíl v této službě (a ve VCS repozitáři, do kterého bude přispívat) oproti původní je, že nepracuje formou periodického tvoření snapshotů, ale okamžitě reaguje na každou provedenou DDL změnu v databázi. Tyto jednotlivé změny pak převádí na samostatné commity, které do vzdáleného repozitáře VCS odesílá. Toto řeší problém nemožnosti identifikace samostatných změn ve VCS, pokud v kontrolním okně dojde k více změnám ve stejném objektu.

V rámci návrhu a vývoje našeho softwaru budeme předpokládat, že používaná služba v cílovém prostředí je tato nová. Nový systém bude totiž takto fungovat v produkční prostředí jako software složený z dvou samostatných částí pracující v opačných směrech (DB -> VCS a VCS -> DB). Díky tomu můžeme využít faktů, že změny jsou propisovány z DB do VCS téměř okamžitě. Dále můžeme navrhnout případnou komunikaci mezi oběma novými službami, která bude v požadavku vývoje jejích funkcionalit zahrnuta, pokud tato komunikace bude potřeba.

Protože služba je výsledkem interního vývoje, její software ani zdrojové kódy nebudou přiloženy k BP, pouze její využití bude zmíněno při návrhu nového softwaru a v jeho zdrojovém kódu.

3.2 ServerEventLog

ServerEventLog (či EventLog) je tabulka v produkční databázi, do které se za účelem evidence změn zapisují okamžitě veškeré DDL události pomocí serverového triggeru. Historicky existovala pouze tato tabulka pro zkoumání historie databáze (a software, který její prohledávání zjednodušoval pomocí uživatelského rozhraní).

Později se zavedlo propisování změn synchronizační službou do VCS, ale tento postup má své dříve zmíněné nevýhody. Z toho důvodu evidence změn v tabulce stále funguje a je využívána jako dodatečný zdroj informací, které ve VCS nejsou. Toto následně způsobuje sémantický konflikt, protože dochází k existenci dvou zdrojů pravdy (tabulka vs VCS), kdy v každém existují jiná data pro různé použití.

Kompletní struktura tabulky a trigger, který tabulku plní, je stejně jako synchronizační služba výsledkem interního vývoje, proto její přesná struktura ani trigger nebudou k BP přiloženy. Níže jsou ovšem zmíněny některé obsažené sloupce, jejichž existence může být v rámci BP využita.

- Created – Datum a čas, kdy DDL operace proběhla
- LoginName – Login uživatele, který změnu provedl
- EventType – Typ události, může se jednat například o vytvoření tabulky, úpravu pohledu, smazání funkce etc.
- ObjectType – Typ ovlivněného objektu, může se jednat například o tabulku, pohled, funkci, trigger nebo proceduru
- DatabaseName – Jméno databáze, ve které je upravovaný objekt uložen
- SchemaName – Jméno schema, ve kterém je upravovaný objekt uložen
- ObjetName – Název objektu (pouze poslední část bez jména databáze a schema)

K BP bude přiložen skript pro vytvoření této zjednodušené verze tabulky.
Protože nebude k BP přiložen trigger, který plní tabulku, její postupné plnění daty o nasazovaných objektech je třeba při případném testování funkčnosti provádět ručně.

3.3 Soubor RELEASE_ORDER

RELEASE_ORDER je soubor uložený v kořeni databázového repozitáře, který je upravován vývojářem a následně užit v rámci původní synchronizační služby. Účel tohoto souboru je definovat pořadí skriptů, které jsou třeba v daném pořadí nasadit. Toto může být potřeba dodržet kvůli závislostem mezi upravovanými objekty, například když tvoříme více nových databázových objektů, které na sebe odkazují/využívají se.
Soubor má podobu obyčejného textového souboru, kdy na každém řádku je cesta (s kořenem ve složce repozitáře) ke skriptu, který se má v daném bodě nasadit. Pokud některé z upravených skriptů nejsou v souboru zmíněny, vývojář tím indikuje, že na pořadí jejich nasazení nesejde a mohou se po nasazení skriptů zmíněných v souboru nasadit v libovolném pořadí. Synchronizační služba nedefinuje přesně, jaké toto následné pořadí je. [22]
Soubor je již využíván při současném vývoji, jedná se tedy o používanou konvenci, na kterou jsou vývojáři zvyklí. Pro minimalizaci nutných změn v procesu k naučení vývojáři je možné použít stejnou podobu tohoto souboru, protože i při novém procesu bude třeba definovat pořadí nasazení skriptů, kdy ovšem v novém procesu nebude pořadí dodržováno seniorním vývojářem nasazujícím skripty, ale soubor s pořadím bude automaticky zpracován vyvinutým softwarem.

Kapitola 4
Definice možných komplikací

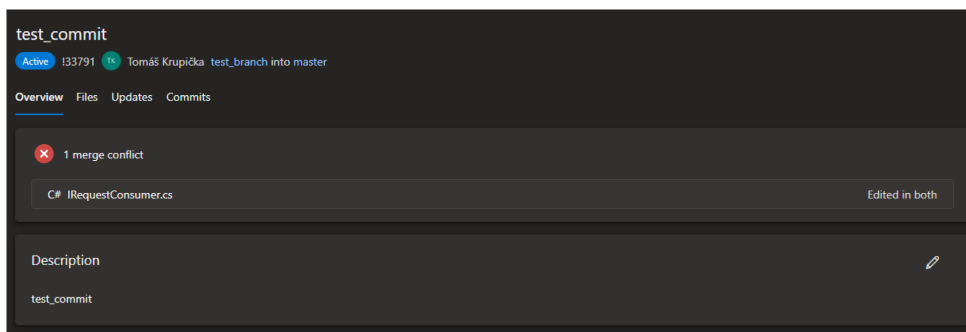
Kvůli určení využití služby ve Společnosti, kde pracuje hodně vývojářů, kteří mohou zasahovat do struktury databáze, se nevyhneme různým komplikacím, které mohou nastat při nasazování změn z MR. Tyto změny je třeba dobře sepsat a v návrhu softwaru navrhnout vhodné řešení, aby jejich přítomnost nezpůsobila žádné problémy/chyby při vykonávání nasazení či aby tyto problémy měly co nejmenší dopad.

4.1 Neaktuálnost zdrojové větve – obecné komplikace

První komplikace nastává přímo v repozitáři VCS. Pokud vývojář na svých změnách pracuje delší dobu, jeho větev může být oddělená od hlavní (produkční) větve ve starším commitu, než je aktuální vrchol produkční větve. První možnost, kdy toto může nastat je, že vývojář si svoji větev oddělil na začátku práce na požadavku a do své větve průběžně přidával změny. Druhá možnost vzniká z faktu, že vývojář větev sice oddělil až po dokončení práce, ale následně se měnily požadavky zadání (například po testování), vývoj prošel několika iteracemi a větev se za tu dobu stala neaktuální.

Pokud jsou již přidané změny v jiných souborech, než byly vývojářem změněny, tak k problému nedochází (rozhraní/obecná funkčnost by měla být dle správných vývojových konvencí zachována, jinak jsou změny hůře dohledatelné). V opačném případě jsou z pohledu VCS změny konfliktní nebo nejsou.

Pokud jsou změny konfliktní (došlo například k úpravě na stejných řádcích), bude seniorní vývojář nasazující změny přes webové rozhraní VCS upozorněn. Toto ovšem nesmí být zanedbáno při návrhu, protože může dojít k lidské chybě nebo změny budou do produkční větve přidány v okamžiku spuštění nasazení a vývojář se o konfliktu dozví pozdě.



Obrázek 4.1 Informování o merge konfliktu v Azure DevOps

Pokud změny nejsou konfliktní (například úpravy jsou v různých částech souboru), k žádnému upozornění nedojde. Problém ale může nastat, pokud například vývojář přidá do souboru funkcionalitu/proces, který byl jiným vývojářem již přidán na jiné místo v kódu a při případném nasazení by došlo k duplikování funkcionality/procesu v produkčním softwaru. Toto je už náročnější na podchycení seniorním vývojářem, který kontroluje kód a je vyšší šance, že bude spuštěno nasazení s tímto konfliktem, který VCS nenahlásí.

4.2 Neaktuálnost zdrojové větve – úpravy tabulek v dočasných souborech

Druhá komplikace již obsažena ve změnách v repozitáři VCS je specifická pro použití s databázovými skripty. Velká část upravovaných objektů jsou pohledy a procedury (viz Statistika databázových změn v 1.3.3). Vývojář ovšem může taktéž v rámci svého vývoje upravovat strukturu tabulek. Oproti kupříkladu pohledům a procedurám, které jsou v repozitáři uloženy jako spustitelné skripty pro jejich úpravu (ALTER / CREATE OR ALTER skripty), jsou tabulky uloženy jako skripty pro případné vytvoření tabulky. Pokud vývojář vypracuje úpravu struktury tabulky, nelze využít stejné skripty (které budou pouze upraveny), pokud by nemělo dojít ke znovuvytvoření tabulky, a tedy i ztrátě uložených dat.

Tyto změny jsou ve VCS reprezentovány (a následně jsou nasazeny) pomocí migračních skriptů (často jedním skriptem nad jednou tabulkou), které jsou uloženy v samostatných souborech. Struktura tabulky se mění v produkční větvi VCS díky synchronizační službě, která provedené změny ve struktuře tabulek (a jiných objektů) zapisuje do VCS. Soubory skriptů pak existují ve vývojové větvi souběžně spolu se soubory s původní strukturou tabulky. Díky tomu VCS nedokáže poznat souvislost mezi změnami v těchto souborech (skriptem a tabulkou) a určit, že se jedná o konfliktní změny.

4.3 Commitnuté jednorázové skripty/dočasné soubory

Další komplikace vychází opět z dříve zmíněných migračních skriptů tabulek a taktéž například jednorázových skriptů (vložení dat do tabulky) či jiných dočasných souborů (soubor RELEASE_ORDER pro určení pořadí nasazení skriptů).

Tyto soubory jsou součástí MR a jsou obsaženy v commitech, které vývojář předává ke CR a následnému nasazení. Problém je, že po sloučení s produkční větví by tyto skripty (či změny v daných souborech) neměly zůstat jako součást repozitáře (respektive, neměly by být viditelné v commitu, ze kterého si další vývojář bude odpojovat svoji vývojovou větev). Pokud by došlo ke kompletnímu standardnímu sloučení změn, do cílové větve by se promítly změny všechny, včetně těchto dočasných souborů / jednorázových skriptů.

Tyto skripty je třeba jasně odlišit a zajistit, aby v hlavní produkční větvi nezůstávaly po vývojářích, aby jiný vývojář mohl pracovat na čisté větvi pouze s původními databázovými objekty, které může upravovat anebo sám přidávat vlastní dočasné soubory/skripty.

4.4 Změny v produkční větvi VCS

V současném stavu přispívají do produkční větve databázového repozitáře 2 skupiny subjektů – synchronizační databázová služba a seniorní vývojáři.

Přímé úpravy od vývojářů jsou prováděny zřídkakdy, pouze, pokud je třeba ručně upravit produkční větev z nouzových důvodů (například se ve větvi objevily soubory, které byly omylem přidány/sloučeny z jiné větve). Hlavní zdroj úprav produkční větve je synchronizační služba, která přidává změny, které jsou nasazeny ručně na databázi vývojářem. Protože (dle požadavků) tyto procesy mohou stále probíhat i po zavedení nového procesu, tyto změny se mohou ve VCS repozitáři stále objevovat.

Při návrhu je tedy třeba zohlednit, že synchronizační služba (či seniorní vývojář) může kdykoliv upravit produkční větev, když se na ní zrovna bude pracovat v rámci nového procesu. To způsobí rozdílnou historii v lokální větvi oproti větvi ve vzdáleném repozitáři a kvůli tomu nepůjde odeslat změny do vzdáleného repozitáře bez dodatečných kroků.

4.5 Více souběžných požadavků k nasazení

Ve Společnosti funguje několik samostatných týmů, kdy každý tým vyvíjí funkcionality, které se vztahují k doméně, kterou tento tým spravuje. Mnohdy ale tyto týmy pracují nad stejnými databázemi a sdílí tedy jedno produkční prostředí. V souvislosti s touto prací to má následek, že sdílí i jednu produkční větev ve VCS. Každý tento tým má minimálně jednoho seniorního vývojáře (často více) s právy na nasazování změn do databáze a merge změn do produkční větve VCS (pokud je to třeba).

Díky tomu se může stát, že ve stejnou chvíli bude chtít nasadit změny více seniorních vývojářů ve stejném časovém okně (probíhá nasazování změn vývojáře A a vývojář B chce nasadit svoje změny). Je třeba počítat s tím, že pokud tato situace nastane, nesmí se stát, že souběžné příkazy k nasazení změn se vzájemně negativně ovlivní, a například tím způsobí nevalidní stav VCS/databáze.

4.6 Externí změny v databázových objektech

Tato komplikace je hodně podobná a souvisí se „Změnami v produkční větvi VCS“ 4.4. Na rozdíl od dříve popsané komplikace, která se zaměřuje na změny ve větvi VCS a následnou nemožnost jednoduchého odeslání změn do vzdáleného repozitáře, zde se zaměříme na změny v databázi.

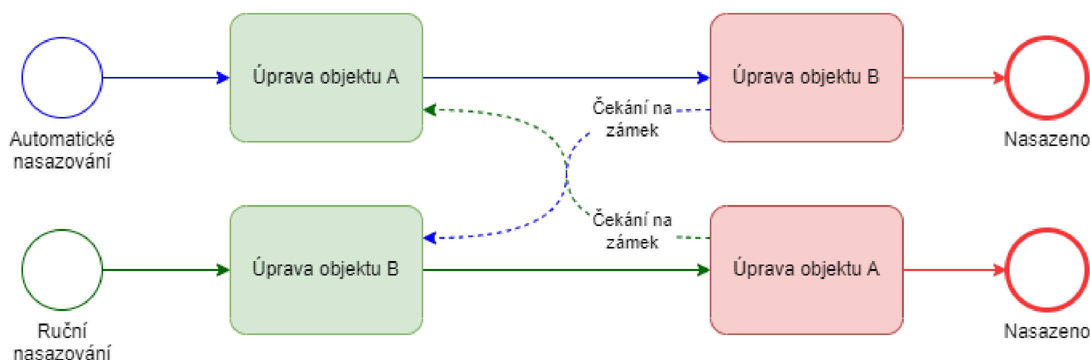
Opět, díky nutnosti stálého umožnění změn prováděných mimo tento systém, může dojít ke změnám v databázi v průběhu nasazovacího procesu. To může způsobit dva problémy.

První problém nastane, pokud nasazení externích změn proběhne dříve než našich. Tyto změny následně svojí úpravou přepíšeme a ručně nasazené změny budou přepsány. V nejhorším případě se na tuto situaci okamžitě nepřijde, ale až ve chvíli, kdy v produkčním prostředí něco nebude fungovat/bude fungovat nekorektně kvůli chybějícímu kódu z ruční úpravy. Stejný problém může nastat z opačného směru (externí změny přepíšou naše), ale to je problém, který je na zodpovědnosti vývojáře, který provádí ruční změnu (jedná se o nestandardní postup a musí tedy zajistit, aby jeho postup něco nepokazil), nebudeme tedy tuto situaci v rámci návrhu řešit.

Druhý problém souvisí s transakčním zpracováním – pokud se více změn nasazuje v rámci transakce, zámky, které brání v úpravě objektů, jsou drženy až do konce transakce. Pokud skript,

který automaticky nasazujeme běží v transakci, ovlivňuje více objektů, a to samé platí pro ručně nasazovaný skript, může dojít k deadlocku.

Deadlock je situace, kdy dvě transakce (obecně pracovní vlákna) čekají na zámek nad objektem, který drží ta druhá. Protože ani jedna z transakcí nemůže být dokončena dříve, než se provedou všechny její operace, zámky nebudou nikdy uvolněny a vykonávání obou transakcí se nikdy nedokončí, pokud není ručně zrušeno. Toto se taktéž nazývá cyklickou závislostí. V případě SQL serveru toto nemusí nastat, protože monitoruje transakce pro výskyt deadlocků a v případě nálezu jeden z procesů (transakcí) ukončí. [24] Toto nemusí ovšem platit u jiných databázových serverů, pro které může být navrhovaný proces použit.



Obrázek 4.2 Znázornění deadlocku při průběhu nasazování

V případě, že zmíněné ruční nasazování je vykonáváno vývojářem, této situace si může všimnout a svoji transakci zrušit. Ovšem v případě, kdy se nejedná o člověka, nýbrž jinou automatizovanou službu, obě transakce může být zaseklé, do té doby, než si toho někdo z vývojového týmu všimne / databázový server transakci automaticky ukončí. I přes možné vyřešení třetí stranou je třeba s touto situací počítat a snažit se ji zabránit.

4.7 Dlouho trvající nasazování

Databázové skripty mohou běžet různě dlouhou dobu. Některé, například změny procedury, jsou vykonány téměř okamžitě, jiné, například operace nad tabulkami, mohou trvat dlouho. Protože nasazování probíhá nad produkční databází, probíhající nasazování může blokovat jiné kriticky důležité procesy, které nesmí být zpomalovány.

Je třeba umožnit definovat, jak dlouho by standardně operace nasazení měla trvat, a pokud tomu tak není, operaci přerušit. Dále, pokud by došlo k blokování kritických procesů, je třeba navrhnout postup, který umožní procesy odblokovat. [25]

Kapitola 5

Příklady existujících řešení

V rámci této kapitoly si uvedeme příklady možných existujících řešení, které každé nabízí jiný pohled na problematiku verzování a nasazování databázových změn. Možnost užití těchto řešení bude zhodnocena na základě jejich výhod a nevýhod. Pokud žádné z řešení nevyhovuje použití dle definovaných požadavků na vylepšení procesu, bude třeba vytvořit nové vlastní řešení.

5.1 Flyway

Flyway je open source řešení správy verzí databáze vyvíjené firmou Redgate. Software je nabízen v základní variantě zdarma, či v placené komerční verzi s dodatečným softwarem, který může být použit velkými týmy ve firmách. Flyway podporuje velké množství DBMS včetně MSSQL a v komerční verzi nabízí integrace pro CD/CI software včetně Azure DevOps. [26]

Flyway funguje primárně na principu migračních skriptů, které mohou být napsány v různých jazycích včetně SQL. Tyto migrační skripty jsou nasazovány pomocí desktopové aplikace, příkazové řádky nebo poskytovaného API. [27] Migrační skripty jdou taktéž importovat ze souborového systému (při vhodné konvenci pojmenování). Pro migrační skripty dokáže Flyway taktéž vytvářet skripty, které reverzují provedené změny pro vrácení na dřívější verzi. Flyway eviduje provedené změny v tabulce změn v databázi, kterou používá jako referenci při nasazování migrací. [28]

Pro použití s MSSQL nabízí dále Flyway možnost evidování změn v souborovém systému s podporou VCS ve formě souborů pro každý databázový objekt. Díky tomu je možné sledovat historii podoby jednotlivých objektů mezi verzemi. [29]

V komerční verzi Flyway nabízí možnost získání reportu změn, které byly na databázi provedeny mimo nástroj Flyway (drift report) a užití tohoto reportu pro vygenerování migračních skriptů. Při použití dalšího softwaru od Redgate existuje i postup, jak tyto změny automaticky detekovat a spouštět na základě detekce automatické procesy (např. vytvoření migrace z drift reportu). [30] Protože migrace je vytvořena na základě drift reportu, a ne jednotlivých změn, více změn provedených po sobě může být takto evidováno jako jedna změna.

Společnost toto řešení používá pro některé projekty a ze zkušeností je třeba zmínit, že kvůli evidenci migrací v tabulce může dojít k problémům, pokud tabulka není aktuální. Do toho je potřeba poté ručně zasáhnout a pro použití v rámci automatického systému to může být překážkou. [31]

Flyway je ideálním kandidátem na použití pro řešení problematiky této BP. Flyway nabízí spoustu výhod v rámci komerční verze, které jsou pro řešení použitelné. Bohužel, nevýhodami je nespolehlivost závislosti na tabulce historie migrací, která může vyvolat problémy v automatickém režimu provozu, a vytváření jedné migrace z více změn při detekci driftu databáze. Protože

jedním z požadavků je evidování reálných provedených změn, nikoliv souhrnu změn provedených za období, toto řešení není vhodné k použití v cílovém prostředí.

5.2 Entity framework core

Entity framework core je open source knihovna vyvíjená Microsoftem, která je součástí platformy .NET. Tato knihovna se zaměřuje primárně na vývoj aplikací, ve kterých je při použití frameworku možné jednoduše přistupovat k datům v tabulkách pomocí objektově-relačního mapování (ORM). [32]

Knihovna umožňuje definovat podobu tabulek v kódu (pomocí ORM) a taktéž při nutnosti její změny definovat migrace mezi jednotlivými verzemi. Změny ve struktuře tabulek mohou být poté automaticky provedeny při připojení aplikace k databázi. Provedené migrace jsou následně evidovány v tabulce v databázi. [33]

Při použití tohoto postupu lze využít VCS tím, že kód aplikace (a ORM včetně migrací) je evidován ve VCS. Tímto můžeme zpětně zjistit například, kdo dané změny v aplikaci vytvořil.

Tento postup je použitelný v situacích, kdy je s databází používaná pouze jedna aplikace, či každá aplikace spravuje svůj vlastní set dat, která využívá. Protože ve Společnosti toto neplatí (existuje více aplikací, které se připojují ke stejné databázi a využívají stejné tabulky), toto nejde použít.

Dále, EF core umožňuje spravovat pouze tabulky. Protože chceme evidovat i jiné objekty (například procedury nebo triggery), i z tohoto důvodu se jedná o nevhodné řešení.

5.3 Diplomová práce Petra Jindry

V rámci hledání alternativních řešení byla nalezená diplomová práce od Petra Jindry. [34]

Jeho přístup se zaměřuje na převod gramatiky migračních skriptů, vytvořených programátory a používaných k úpravě objektů, na grafy, ze kterých je možné zpětně vygenerovat migrační skript pro nasazení a taktéž databázové objekty evidovat v kompaktní formě XML souborů s jejich strukturou.

Protože k tvoření grafů dochází v paměti programu, je možné grafy porovnávat a případně i kombinovat, což je možným způsobem řešení detekce komplikací už na úrovni aplikace bez nutnosti pokusu o nasazení.

Diplomová práce ovšem neřeší možnost nasazování skriptů ručně a následnou evidenci těchto ručně nasazených skriptů ve VCS, což je pro potřeby Společnosti potřeba zahrnout.

Dále, pro účely vývoje s použitím VCS, chceme, aby ve VCS byly přímo evidovány skripty, které mohou být jednoduše uživateli upravovány, což zjednodušuje následné CR. Protože jsou v tomto řešení databázové objekty evidovány ve formě XML souborů, které je reprezentují, toto řešení není pro použití ve Společnosti vhodné.

Kapitola 6
Návrh řešení komplikací

Před konkrétním návrhem postupu, který bude řešit nasazování skriptů je třeba navrhnout řešení všech dříve zmíněných komplikací, aby výsledný program/proces byl co nejvíce robustní vůči všem situacím, které mohou nastat v rozsáhlém produkčním prostředí Společnosti, ve které se na vývoji podílí velké množství vývojářů. Navrhnutá řešení budou použita při implementaci jako vlastnosti, které musí být pro korektní chod splněny, či postupy, které musí být dodrženy.

6.1 Více souběžných požadavků k nasazení

Aby nedocházelo ke kolizím při více požadavcích k nasazení najednou, nebudeme více požadavků najednou zpracovávat – požadavky budeme řadit do fronty a vždy vykonáváme pouze jeden požadavek najednou (první ve frontě). K tomu, abychom mohli takto koordinovat nasazování při využití Azure pipeline, nabízí se dvě řešení.

První řešení je použít Azure pipeline agent pool s pouze jedním agentem. Azure pipeline agent je prostředí pro virtuální stroj, na které když je zaslána pipeline ke zpracování, v rámci agenta se nainstaluje veškerý agent software (resp. se kompletně připraví prostředí), který je třeba k vykonání pipeliny. Tento software/prostředí lze nakonfigurovat pro každou pipeline dle potřeby. Nevýhoda tohoto postupu je, že prostředí funguje pouze v rámci běhu pipeliny a je zvenku nepřístupné pro dodatečné příkazy/konfiguraci, která není součástí prvotní konfigurace / spouštěcích parametrů. Naopak výhodou je, že prostředí lze použít v rámci pipeline s více kroky (joby), kde mezi jednotlivými joby lze předávat objekty, které joby vytvoří a dále s nimi pracovat. [35] Díky použití poolu s pouze jedním agentem je zajištěno, že se vždy bude vykonávat pouze jedno nasazení najednou.

Druhé řešení zahrnuje hostovanou službu, která funguje jako webový server, a na ní jsou z Azure pipeline přes REST api zasílány požadavky k nasazení [36], které jsou v rámci aplikace řazeny do fronty. Služba v průběhu vykonávání nasazení komunikuje s Azure DevOps opět přes REST api, předává průběžné informace o průběhu, a nakonec i výsledek nasazení. Výhoda tohoto postupu je, že server běží i mimo vykonávání nasazování a je vždy přístupný zvenčí. Díky tomu je možné na něj zasílat i jiné požadavky, případně v průběhu nasazování modifikovat jeho průběh.

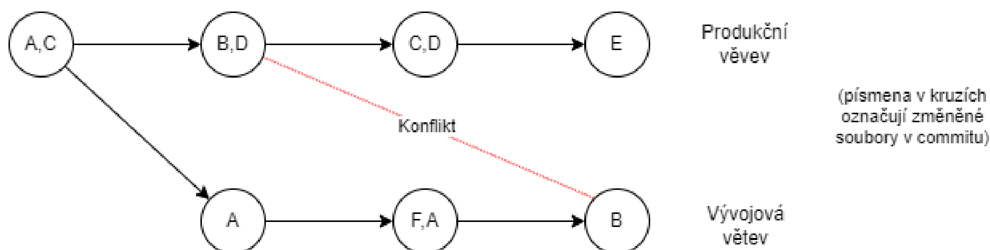
Pro využití v tomto případě jsem se rozhodl k druhé možnosti, protože umožňuje do budoucna větší možnosti modifikace / využití služby. V rámci vytvořeného PoC bude služba fungovat pouze lokálně a přijímat příkazy na standardním vstupu (a dávat zpětnou vazbu na standardním výstupu), ale bude navržena tak, aby byla jednoduše upravitelná pro možnost provozu v podobě webového serveru.

6.2 Neaktuálnost zdrojové větve – obecné komplikace

Problém s konfliktními i nekonfliktními problémovými změnami můžeme vyřešit najednou. Jak bylo zmíněno v definici komplikace 4.1, nekonfliktní problémové změny vždy spočívají v jiné úpravě stejného souboru. V podobném duchu jsou konflikty jsou vyvolány dvěma změnami ve stejném souboru.

„*Pokud jste tutéž část stejného souboru změnili odlišně ve dvou větvích, které chcete sloučit, Git je nebude umět sloučit čistě. Pokud se oprava problému #53 týkala stejné části souboru jako větev hotfix, dojde ke konfliktu při slučování (merge conflict)*“ [12]

Díky tomu můžeme komplikaci zjednodušit na problém, kdy v cílové větvi proběhly libovolné změny ve stejném souboru jako ve větvi zdrojové.



Obrázek 6.1 Znázornění konfliktu mezi dvěma větvemi ve VCS Git

K nalezení změněných souborů na obou větvích využijeme nejprve nalezení nejlepšího merge base commitu pro vrcholy obou větví. Tímto commitem je nejnovější commit, jež je předkem obou vrcholů větví. Tento commit je používán pro standardní three-way merge, což je postup, který budeme při merge používat. Commit lze ve VCS získat pomocí příkazu git merge-base. [37] Tento commit bude většinou commitem, který byl použit jako počáteční vrchol při vytvoření vývojové větve. Oproti tomuto commitu můžeme poté provést diff (nalezení všech provedených změn) z cílové větve a zdrojové větve a získat oba seznamy změněných souborů.

Pokud v těchto dvou seznamech budou společné soubory, znamená to, že vývojář provedl změny ve stejném souboru, ve kterém byly provedeny změny i v produkční větvi (pravděpodobně někým jiným). V kontextu databázových změn se jedná o změnu stejných databázových objektů. Při tomto nálezu merge zrušíme a vrátíme nasazujícímu vývojáři informaci o tom, že byly nalezeny konflikty a v kterých souborech jsou.

Následný postup záleží na povaze konfliktů, ke kterým došlo. Pokud tyto konflikty jsou bezpečné (neovlivní funkčnost), je možné buďto nasadit skripty ručně, anebo požádat vývojáře o rebase větve, aby byla oproti produkční aktuální a konflikty tím byly vyřešeny pro automatické nasazení. Pokud konflikty jsou oprávněné, vývojář je musí opravit, ideálně provést rebase nad produkční větví a nasazovací proces se může zopakovat s již nekonfliktní historií větve.

6.3 Commitnuté jednorázové skripty/dočasné soubory

Aby byl repozitář pro každého vývojáře čistý, bude muset řešení splňovat vlastnost, že změny, které jsou mezi vrcholem produkční větve před a po merge mohou být pouze v souborech, které jsou databázovými objekty.

Díky jasně definované struktuře repozitáře lze databázové objekty poznat podle jejich umístění v repozitáři. Pokud je soubor mimo složku databáze, nejedná se o databázový objekt (do této skupiny souborů spadá i například soubor RELEASE_ORDER). Dále, dle konvencí, které se při vytváření změn dodržují, spustitelné skripty by měly být taktéž ve složce databáze, ke které patří. Zde se jedná zejména o migrační skripty pro tabulky, které nelze dát do původních skriptů samotné tabulky. Zároveň existují podsložky v databázové složce, které oddělují jednotlivé typy

databázových objektů evidovaných ve VCS. Proto skripty, které nejsou databázovými objekty musí být umístěny přímo v databázové složce, a nikoliv v podložkách. Tímto poznáme zbytek jednorázových skriptů, které nejsou mimo databázové složky.

K řešení této komplikace jsou dva návrhy, jak jde vyřešit.

První řešení spočívá ve vytvoření čisticího commitu, který je připojený za vrchol větve, kterou slučujeme. Tento commit bude obsahovat opak všech změn, které byly nad dočasnými soubory provedeny od merge base. Tyto změny se vyruší se změnami, co byly provedeny, a výsledkem bude původní stav dočasných souborů (či, pokud byly nově vytvořeny, soubory budou odstraněny). Změny získáme tak, že si načteme podobu dočasných souborů z merge base. Ve VCS Git toho lze docílit pomocí operace Checkout paths s použitím merge base jako referenčního commitu. [38] Výhodou tohoto postupu je, že commity, které jsou přidány do produkční větve, jsou stejné, jaké byly ve vývojové větvi, tzn. i ve stejné podobě. Zároveň lze v produkční větvi zpětně dohledat, jaké jednorázové skripty byly použity k provedení změn (například jaké migrační skripty byly použity ke změně struktury tabulky) zobrazením commitu mezi původním a novým vrcholem větve.

Druhé řešení zahrnuje úpravu všech commitů tak, aby v historii produkční větve změny vůbec nebyly obsaženy. K tomu je třeba projít každý commit ve zdrojové větvi a pokud obsahuje změny v dočasných souborech, vytvořit jeho kopii, která nebude tyto změny obsahovat. Ve VCS Git toho lze docílit například pomocí použití interaktivního rebase, který umožňuje projít historii větve a jednotlivé commity v průběhu upravit. Výsledkem je změněná historie větve dle provedených úprav. Protože je historie změněná, ve změněných commitech se jedná o nové commity. [39] Výhodou tohoto řešení je čistá historie produkční větve.

V rámci implementace PoC jsem zvolil první řešení, ale obě řešení jsou korektní a vyhovují požadavkům na systém. Proto správnost tohoto rozhodnutí bude vyhodnocena v rámci dotazníku zaslaného cílovým uživatelům.

6.4 Neaktuálnost zdrojové větve – úpravy tabulek v dočasných souborech

Problém vyvstává z toho, že změny, které jsou ve zdrojové větvi v dočasných souborech (skriptech) nejsou provázatelné s ovlivněnými objekty.

K řešení této komplikace využijeme automaticky plněného EventLogu, který eviduje reálné změny, které byly nad databázovými objekty provedeny. Protože trigger, který zachycuje všechny DDL události provedené nad databázovými objekty, nerozpoznává rozdíl mezi ručním nasazením a nasazením naší službou, můžeme tohoto faktu využít. Naše nasazování bude probíhat pomocí vlastního databázového uživatele s vysokými právy (aby šlo nasadit vše co bude posláno do služby), který nebude jinak používán.

Protože součástí EventLogu je čas nasazení a identifikátor ovlivněného objektu (o třech částech – databáze, schema a jméno), pokud si poznamenáme čas začátku procesu nasazování, můžeme najít všechny změny, které naše služba v rámci běhu skriptů provedla, a díky tomu i seznam všech reálně dotčených objektů našimi změnami.

Soubory databázových objektů v repozitáři mají jasně definovanou strukturu cesty/názvů, ze které můžeme extrahovat identifikátor daného objektu. Pokud toto provedeme pro všechny změněné soubory od merge base našeho MR (získaného stejným způsobem jako u minulé komplikace 6.3), dostaneme seznam změněných databázových objektů na produkční větvi.

Tyto dva seznamy můžeme následně porovnat, a pokud obsahují společný databázový objekt, našli jsme takový objekt, který byl změněn jak na zdrojové, tak na cílové větvi, a získáváme konflikt. Postup, jak tento konflikt může být vyřešen je pak shodný jako u předcházející komplikace s neaktuálností větve 6.2.

Díky těmto poznatkům ovšem vzniká nová komplikace, a to, že DDL události v EventLogu budou přítomny jak pro naše změny, tak pro změny externí, a obojí bude zachyceno synchroni-

zační službou. V rámci naší služby ovšem budeme taktéž provádět merge commitů vytvořených vývojářem do produkční větve. Tím vzniká problém, protože by se mohlo stát, že synchronizační služba se pokusí o to automaticky vytvořit nový commit z našich změn, které jsou již do repozitáře přidány.

Toto ovšem může být vyřešeno na straně synchronizační služby tím, že bude odfiltrovávat všechny netabulkové změny provedeny naším databázovým uživatelem, protože tyto změny budou prováděny v commitech přímo na již exitujících souborech s DDL skripty, které budou přidány do produkční větve. Výjimku tvoří tabulkové úpravy, pro které je nutno vytvořit extra skripty, které přidány nebudou, a synchronizační služba jejich záznamy v EventLogu zpracuje jako normálně. [40]

6.5 Změny v produkční větvi VCS

Řešení této komplikace si rozdělíme na dvě části podle subjektu, který do produkční větve zasahuje.

Hlavním zasahujícím subjektem je automatická synchronizační služba. Zde je potřeba vynutit její součinnost při nasazování, aby změny neprováděla. Toho můžeme docílit tím, že na začátku zpracovávání požadavku zašleme na synchronizační službu požadavek k pozastavení její činnosti. Synchronizační služba dokončí synchronizaci aktuálně zpracovávaných změn, a následně nebude kontrolovat EventLog pro další změny, a tedy změny ani nebudou propisovány do VCS. Až po zpětné vazbě o dokončení rozdělané synchronizace si stáhneme nejnovější produkční větev z VCS, čímž získáme nejnovější změny, které byly do požadavku k pozastavení provedeny.

Po dokončení naší práce na nasazování skriptu do DB a odeslání změn do vzdáleného repozitáře VCS zašleme opět požadavek na znovuobnovení její práce, a synchronizační služba si obdobně stáhne ze vzdáleného VCS změny, které provedla naše služba. Tímto zajistíme, že obě služby budou mít vždy nejnovější verzi změn v rámci VCS při své práci, a najednou bude změny provádět pouze jedna z nich.

Pro tuto komunikaci bude muset služba vystavit nový komunikační endpoint, ale protože funguje v podobě mikroslužby na serveru, je to implementovatelné bez větších zásahů do návrhu synchronizační služby. V rámci PoC bude tato komunikace pouze znázorněna komentáři v kódu v místě budoucího provedení, protože daný endpoint zatím na straně synchronizační služby nebyl implementován. [40]

Pokud zasahujícím subjektem je vývojář, jedná se o nestandardní postup, jehož korektní výstup je na zodpovědnosti daného vývojáře, jehož chování není možné ovlivnit implementací. Pro případ, že by do vzdálené větve zasáhl v průběhu zpracování, výsledkem bude rozdílná historie v rámci lokální větve služby a větve ve vzdáleném repozitáři. Pro korektní dokončení procesu nasazení v tu chvíli dojde k vynucenému přepsání vzdálené větve lokální historií.

Do budoucna bude možné například vytvořit webové rozhraní, ve kterém vývojář, který chce ruční změny provést, bude moci aktuální stav služby ověřit a případně její chod pozastavit (zašle na službu pozastavovací požadavek), provést změny, a následně službu znovu spustit. K pozastavení by došlo až po dokončení současné práce na nasazení. Protože si služba vždy stahuje nejnovější verzi větve na začátku zpracování nasazovacího požadavku, vždy tím korektně získá nejnovější verzi větve z VCS s případnými ručně aplikovanými změnami.

6.6 Dlouho trvající nasazování

Tato komplikace jde jednoduše vyřešit nastavením časového limitu na vykonání databázového skriptu, který bude předán službě spolu s commitem k nasazení. Pokud časový limit vyprší z důvodu delšího provádění, než bylo předpokládáno, dojde k přerušení aktuálně prováděného skriptu, což vyústí v jeho neúspěšné nasazení a zrušení i zbytku nasazování v rámci transakce.

Pokud je časový limit nastaven korektně pro délku transakce, ale dojde k neočekávanému blokování, je třeba vynuceně provádění skriptů přerušit. Protože nasazení jednoho skriptu je plně v režii databázového serveru (se kterým služba jen komunikuje), v takovou situaci bude třeba provádění ručně přerušit externím příkazem. Ten může být vykonán ze standardního vývojového prostředí po připojení k dané databázi. V rámci služby toto může být implementováno jako administrativní požadavek, který by byl zaslaný například přes webové rozhraní, ale v rámci PoC toto řešeno nebude díky existenci jednodušší alternativy.

6.7 Externí změny v databázových objektech

Komplikaci se změnami, které naše úprava může přepsat bude řešena na úrovni VCS a na úrovni databáze.

Protože jsme si dříve definovali, že naše služba bude pozastavovat synchronizační službu, která nejprve dokončí synchronizaci změn, které byly nasazeny 6.5, změny provedené do momentu pozastavení služby budou již propsané do VCS. Tyto změny budeme moci detekovat způsobem popsaným u komplikace „Neaktuálnost zdrojové větve – obecné komplikace“ 6.2.

Změny, které byly provedeny po pozastavení služby určíme pomocí času, který jsme získali pro kontrolu reálně ovlivněných objektů 6.4. Podobně, jako u kontroly ovlivněných objektů můžeme ze seznamu změn v EventLogu získat seznam změn, které byly provedeny od daného momentu, ale nebyly provedeny naší službou (použitý login není loginem používaným službou). Opět využitím stejného postupu porovnáme seznam objektů ovlivněných námi a seznam objektů ovlivněných jiným uživatelem. Pokud seznamy obsahují společný objekt, došlo ke konfliktní změně, a změny je třeba vrátit zpět.

Pokud záznam času, který označuje počátek nasazování, provedeme před pozastavením služby, mohou existovat objekty, které budou v seznamu změn v Gitu i v seznamu změn z EventLogu, což není problém, protože v obou případech chceme při konfliktu nasazení zrušit. Důležité je, že protože se tyto množiny mohou časově (dle času nasazení) překrývat, nemůže existovat nasazený objekt, který bychom tímto postupem nezachytili.

Další věc, kterou je třeba zvážit, je, jak se chovají transakce při nasazování změn databázových objektů. V předchozích případech jsme zvažovali změny, které byly nasazeny, tedy, pokud byly spuštěny v transakci, transakce již byla potvrzena. Při provedení DDL změn se po celou dobu trvání transakce drží nad ovlivněnými objekty Schema modification lock. Tento lock zajistí, že pouze jedna transakce může najednou upravovat daný databázový objekt, a pokud se jiná transakce o úpravu pokusí, musí čekat, než bude zámek uvolněn (transakce skončí). [41] Pokud je transakce s úpravou v průběhu, může dojít ke dvěma případům.

První případ nastane, pokud běží transakce služby, a před první úpravou daného objektu se spustí jiná transakce, která tento objekt upraví. Naše transakce bude v tu chvíli čekat na doběhnutí druhé transakce, než databázový objekt bude moci upravit, a v práci pokračovat. V tuto chvíli již bude tato změna obsažena v EventLogu a proběhlou změnu služba detekuje jako konflikt.

V druhém případě transakce naší služby objekt upraví a druhá transakce čeká na úpravu. Naše transakce tím první vyřeší všechny potřebné operace (například zapsání změn do VCS), a až poté zámek uvolní ukončením transakce. K následné úpravě dojde až po nasazení všech změn, a proto půjde o korektní souslednost nasazení, která není konfliktem (cizí transakce přepsala naše změny) a bude následně díky synchronizační službě korektně zaznamenána ve VCS.

Protože změny, které z EventLogu čteme (a mohou být konfliktní) jsou vždy již z potvrzených transakcí, můžeme zvolit izolační úroveň pro naše transakce READ COMMITED, kdy v rámci transakce vidíme pouze změny provedené jinými transakcemi, tkeré již byly potvrzené (například záznamy v EventLogu z triggeru). [42]

Nakonec, případné deadlocky, které mohou vzniknout kvůli dvěma probíhajícím transakcím, budou vyřešeny již zmíněným časovým limitem 6.6, který při delším provádění operace, než bylo

očekáváno, nasazení přeruší, a tím deadlock vyřeší (zámky nad aktuálně upravovanými objekty budou uvolněny).

Kapitola 7

Návrh upraveného vývojového procesu

Jak jsme si definovali u původního procesu, CR cykly jsou navrženy ideálně, proto nebudou v novém procesu upravovány. Úprava procesu se zaměří na to, co se děje po úspěšném CR, kdy má dojít k nasazení.

7.1 Nasazování hotfixů

V případě, že je úpravou hotfix, nové úpravy nejsou evidovány ve VCS. Protože naše navrhovaná služba se pro získání informací o MR opírá o VCS, v této situaci naše služba nejde použít a musí se použít původní postup ručního nasazení. Toto znamená, že do databáze budou stále nasazovány některé úpravy ručně, a tedy že i z tohoto důvodu bude muset být v provozu i synchronizační služba, která některé úpravy bude odesílat do VCS.

7.2 Nasazování standardních úprav

Ruční proces nasazování ostatních úprav bude ve většině případů nahrazen nasazením pomocí vyvíjené služby. Po proběhnutí CR a zhodnocení změn, které jsou součástí MR, může vývojář dospět k názoru, že změny automaticky nasadit nejdou, nebo je třeba opatrného ručního nasazení. V tom případě použije starý postup k ručnímu nasazování s použitím VCS. Pokud změny jdou automaticky nasadit, seniorní vývojář zvolí vhodný čas, kdy má k nasazení dojít. V tento čas zašle požadavek k nasazení na službu, a vyčká na výsledek nasazení.

Zaslání požadavku bude probíhat přes webové rozhraní Azure DevOps, kde ve službě Azure Pipelines bude připravena nasazovací pipeline k ručnímu spuštění. Při ručním spuštění bude vývojář požádán o zadání parametrů, kterými bude zdrojová větev, která se má nasazovat, a časový limit pro proces nasazení v sekundách. Tento limit se aplikuje pouze na samotný proces nasazování – pokud v nasazovací frontě již budou další požadavky, ty budou vykonány první. Doba, za kterou vývojář získá informace o výsledku, díky tomu může být delší než nastavený časový limit.

Definice pipeline bude zahrnovat přijetí zmíněných parametrů od vývojáře, a následný „Invoke REST Api“ task, kterým přes REST api zašle požadavek na službu, běžící na přes síť přístupném serveru. [36] Pro zabezpečení komunikace (aby nešlo zasílat požadavky kýmkoliv ručně) bude využito vytvořené připojení ke službě a autorizace přístupovým tokenem, obojí uložené ve službě Azure Pipelines. [43]

7.3 Zpracování požadavku službou

Po přijetí požadavku a odpovědi o korektním přijetí je MR s potřebnými údaji zařazen do fronty požadavků k vykonání. V případě, že nedojde k odstranění požadavku z fronty, než na něj přijde řada (požadavek je stále ve frontě), je zahájen nasazovací podproces, který bude dále vysvětlen v samostatné kapitole 11.

Na začátku podprocesu dojde k signalizování začátku výkonu nasazování přes REST api poskytované službou Azure Pipelines. Dále v průběhu podprocesu bude probíhat průběžné zasílání informací o průběhu nasazování (logových zpráv) přes stejné REST api. Po vykonání nasazení (úspěšném či neúspěšném) dojde k následnému finálnímu zavolání api k signalizování úspěšnosti či neúspěšnosti nasazení, a tedy i celé pipeline. Pokud dojde k odstranění požadavku z fronty (z různých důvodů), je taktéž zaslána informace o nevykonání požadavku přes api. Tímto postupem jde předávat webové službě a vývojáři detailní informace například o tom, v jakém stavu aktuálně nasazení je. [44]

7.4 Zhodnocení výsledků vývojářem

Po dokončení pipeline je vývojář automaticky o dokončení informován, případně může živě sledovat zasílané logové zprávy. Pokud nasazení proběhlo korektně, seniorní vývojář uzavře MR (bez merge), a tím práce na požadavku končí, protože změny jsou nasazeny do databáze a sloučeny v hlavní větvi VCS (změny, které sloučeny nebyly budou následně zpracovány synchronizační službou).

Pokud nasazení neproběhlo korektně, seniorní vývojář prohlédne log nasazování a zjistí, z jakého důvodu se nasazení korektně nedokončilo. Dle těchto zjištění lze zvolit několik postupů.

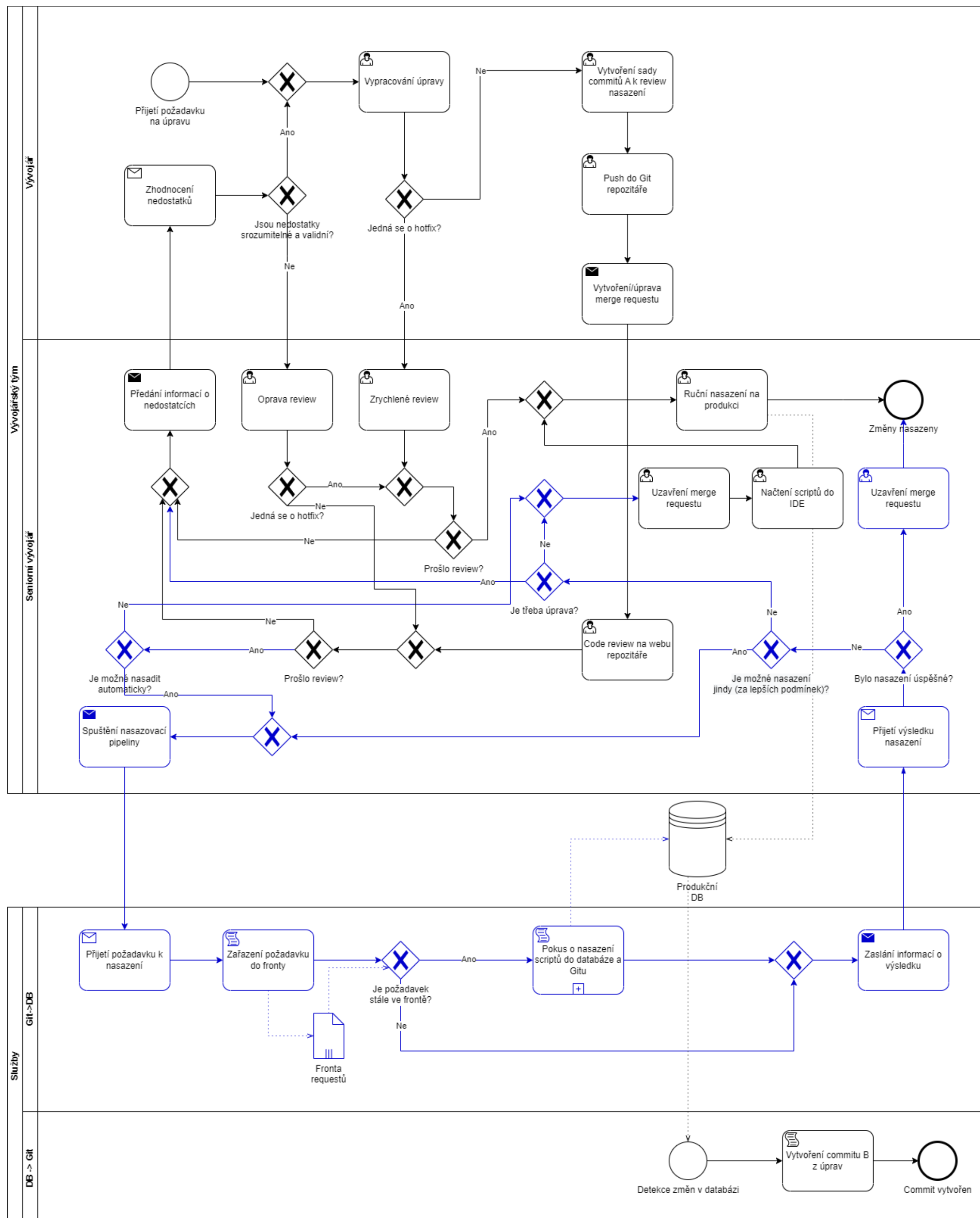
Pokud se nedokončilo z důvodu odstranění z fronty nebo protože vypršel timeout, může se pokusit o automatické nasazení jindy, za lepších podmínek (například, když bude server méně zatížen).

Seniorní vývojář může určit, že skripty jsou korektní (nejsou v nich chyby, které by byly třeba upravit), ale automatické nasazení na něčem selhalo, kvůli čemuž není dobrým nápadem se o automatické nasazení pokoušet znovu. V tomto případě provede seniorní vývojář ruční nasazení dle stejného workflow, jako v původním procesu.

Pokud jsou s nasazením nějaké problémy, které vyžadují pozornost vývojáře (například chyba ve skriptech, či nejčastěji kolizní změny), je třeba tyto skripty upravit. O problémech napíše seniorní vývojář komentáře do MR a vrátí požadavek k dořešení vývojáři. Tím se dostáváme do původního cyklu CR a po průchodu se může nasazení opakovat.

7.5 Diagram nového procesu

Na níže uvedeném digramu je znázorněno kompletní nový proces rozšiřující výše zmíněný zjednodušený proces vývoje požadavku (kapitola 2). Pro odlišení upravených částí jsou tyto části v diagramu vyznačené modrou barvou.



Obrázek 7.1 Upravený vývojový proces

Kapitola 8
Použité technologie pro vývoj

Pro vývoj služby byl zvolen jazyk C#, s použitím platformy .NET 7.

.NET (7) je open source multiplatformní verze .NET Frameworku od Microsoftu. Tato platforma podporuje různé jazyky pro vývoj, jako například C#, F# nebo Visual Basic. Kód napsaný s použitím .NET je kompilován do mezikódu CIL (common intermediate language), který je spoušten ve virtuálním stroji CLR (common language runtime). Platforma podporuje typovou a paměťovou bezpečnost s automatickou správou alokované paměti pomocí garbage collectoru. Platforma taktéž nabízí širokou nabídku knihoven pro užití při vývoji. [45]

C# je programovací jazyk vyvíjen Microsoftem. Jedná se o objektově a komponentně orientovaný jazyk, s C-like syntaxí. Jazyk podporuje mnohé moderní konstrukty, jako nullovatelné typy s ochranou přístupu (při vývoji lze v kódu jednoduše kontrolovat nekorektní práci s případnými nullovými referencemi), užití lambda funkcí a asynchronní volání funkcí. Dále nabízí například syntaxi LINQ pro jednoduché práce s kolekcemi. [46]

Tento jazyk a platforma byly zvoleny z důvodu cílového užití s jinými službami od Microsoftu, pro které v rámci .NET existují nativní knihovny a široká podpora ze strany dokumentace a ukázkových kódů. Zároveň ve Společnosti je C# spolu s .NET používaným jazykem pro vývoj mikroslužeb, tudíž tato volba ctí štábní kulturu. .NET byl dále zvolen oproti .NET Frameworku kvůli cílovému použití i na jiných než windowsových počítačích/serverech.

Kapitola 9

Návrh zpracování požadavků

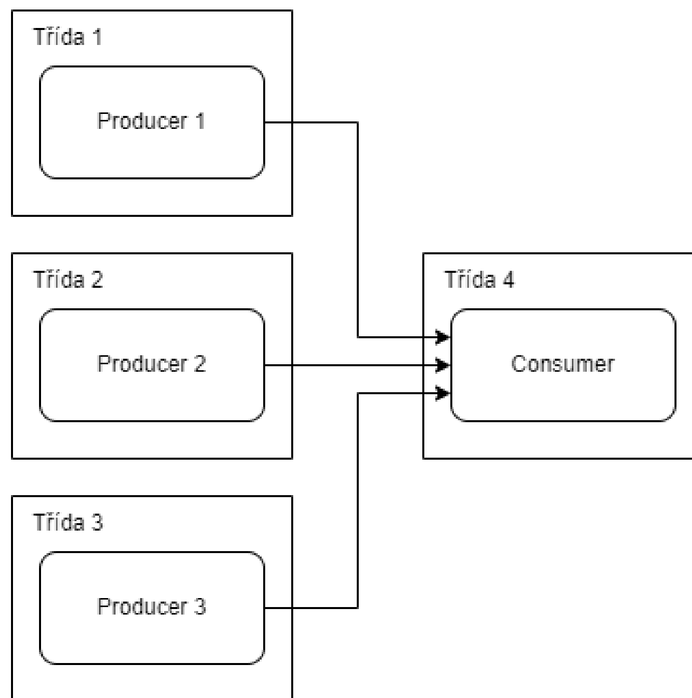
9.1 Jednoduché zpracování požadavků ve frontě

Nejprve je dobré si určit, jak bude naše služba fungovat co se týče vláken. Vyvíjená služba bude zpracovávat požadavky, které jí budou zasílány z různých zdrojů. Ve verzi, která je součástí této bakalářské práce, to budou příkazy ze standardního vstupu. U této verze stačí jednoduchý cyklus, který přijme požadavek, vykoná ho a čeká na přijetí dalšího požadavku.

Toto ovšem nelze aplikovat u cílové verze, která bude mít podobu webového serveru, na který budou zasílány požadavky klienty (uživateli webového rozhraní zasílající požadavky). V případě, že bychom využili stejný princip, tak by každé vykonávání požadavku blokovalo přijímání dalších požadavků od ostatních klientů, než se současně zpracovávaný požadavek dokončí. To by se projevilo vysokou odezvou serveru, což je nežádoucí.

Toto můžeme jednoduše vyřešit implementací pracovního vlákna, které požadavky zpracovává, mezitím co hlavní vlákno přijímá požadavky a předává je do pracovního vlákna ke zpracování, až na ně přijde ve frontě řada. Zároveň to otevírá možnost mít i více zdrojů požadavků, který každý může běžet ve vlastním vlákně.

Toto je klasický příklad producer-consumer problému v synchronizaci více vláken, kdy jedno či více vláken přidává objekty do fronty ke zpracování, a jiné (či více jiných vláken) objekty z fronty odebírá. Problém vyvstává z faktu, že je třeba omezit vlákna, aby najednou ke společnému prostředku (fronta) nepřistupovaly najednou, aby byla zajištěna konzistence čtení/zápisu dat.

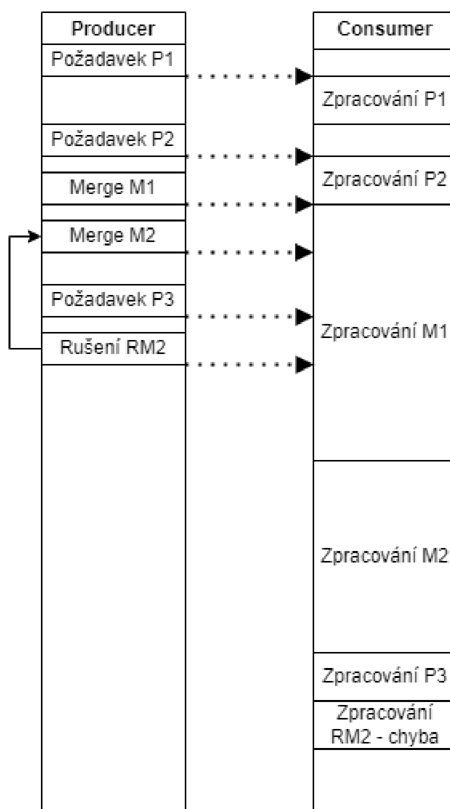


Obrázek 9.1 Diagram využití jedné fronty pro zpracování požadavků

V jazyce C# lze toto jednoduše implementačně vyřešit s pomocí třídy BufferBlock z knihovny Dataflow. Tato třída nám poslouží jako fronta, přes kterou můžeme předávat objekty (požadavky) ke zpracování voláním metody Post (z rozhraní ISourceBlock) a odebírat voláním metody Consume (z rozhraní ITargetBlock), se zajištěním korektní synchronizace při přístupu z více vláken současně. [47]

9.2 Zpracování více druhů požadavků

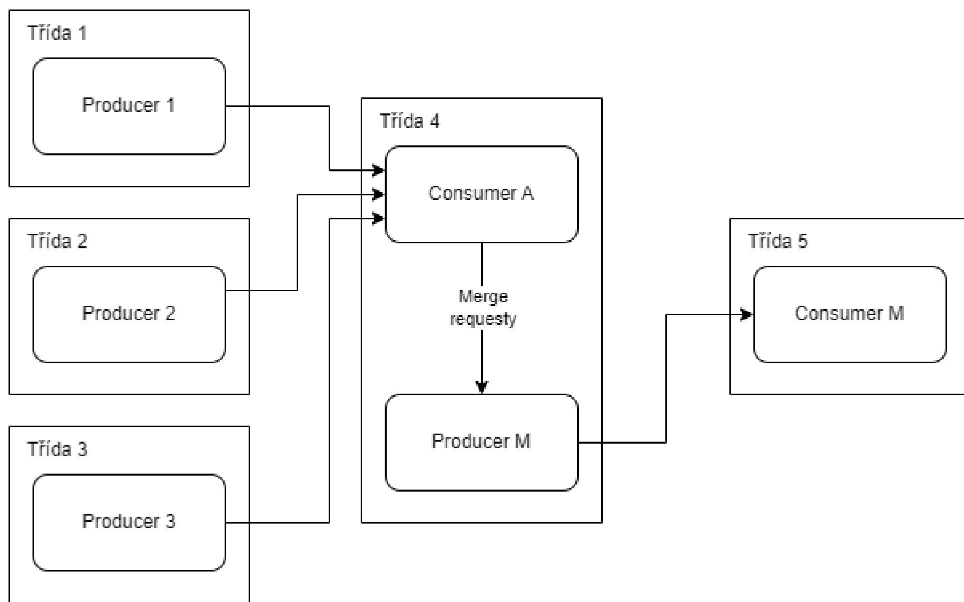
Pokud bychom do služby zasílali pouze požadavky na nasazení databázových skriptů, stačila by nám tato jedna fronta. Problém nastane, pokud budeme chtít zpracovávat i jiné požadavky, které budou mít administrativní povahu jako například rušení požadavků (konkrétních / celé fronty), vypnutí služby nebo zaslání logů. Pokud by bylo rušení požadavku zpracováno v rámci stejné fronty jako nasazování databázových skriptů, požadavek by byl zpracován až po zpracování všech předchozích požadavků, tedy ke zrušení by nikdy nedošlo, protože zpracování zrušení by nastalo až po zpracování nasazení.



Obrázek 9.2 Zpracování požadavků při využití jedné fronty

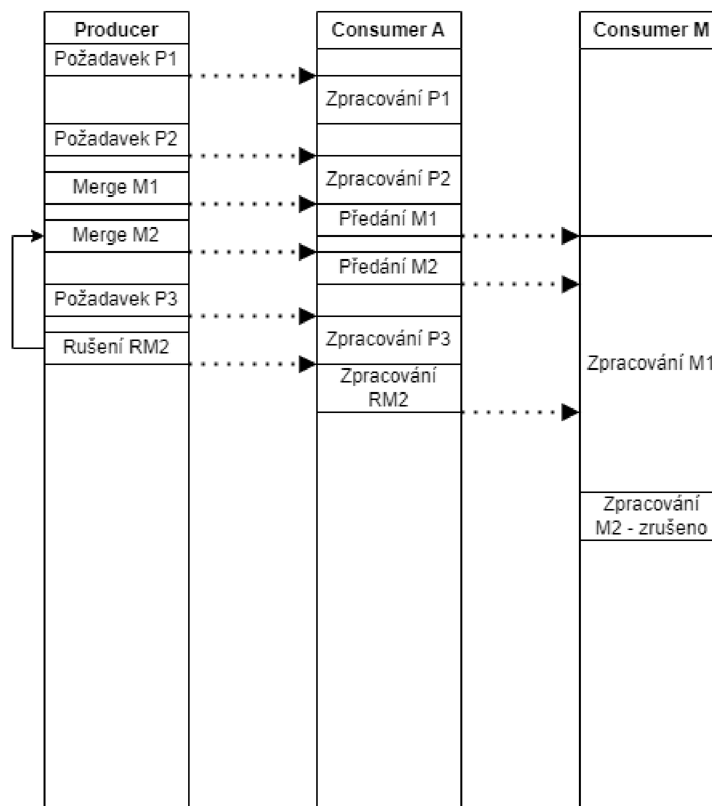
Díky databázové komunikaci a operacím budou nasazovací požadavky nejvíce časově náročné. Během času, co se jeden požadavek zpracovává na databázovém serveru, můžeme v jiném vlákně zpracovávat ostatní požadavky, které nejsou závislé na komunikaci s databází. Jednoduchým řešením je zpracovávat tyto požadavky v hlavním vlákně, ale pokud jakýkoliv požadavek bude časově náročný, může opět dojít k zablokování hlavního vlákna, a tedy i serveru.

Řešením bude vytvoření další fronty ve vlastním vlákně (nová úroveň producer-consumer), která v naší službě bude zařazena mezi zdroje požadavků a pracovní frontu nasazování skriptů. Zdroje budou (až na výjimky) pouze zařazovat skripty do této fronty. Nová fronta bude při zpracování požadavky filtrovat – pokud se bude jednat o požadavek k nasazení, předá ho do nasazovací fronty. Pokud ne, tak ho sama zpracuje.



Obrázek 9.3 Diagram využití dvou front pro zpracování požadavků

Tímto dojde k rychlému zpracování (předání) nasazovacích požadavků a možnosti následně zpracovat pozdější administrativní požadavky ve frontě většinou dříve, než dojde k reálnému zpracování (nasazení) nasazovacích požadavků.



Obrázek 9.4 Zpracování požadavků při využití dvou front

9.3 Zasílání zpráv

Začátek a dokončení zpracování požadavku frontou je provázeno se zasláním stavových zpráv. Podobně během procesu zpracování potřebujeme mít možnost odesílat logové zprávy, většinou zdroji požadavku (terminál, Azure Pipeline), nebo do souboru s logy.

Toto chování můžeme implementovat pomocí tříd, které budou implementovat společné rozhraní cíle pro zasílání stavových/logových zpráv. Instance třídy daného typu cíle bude atributem instance requestu, díky čemuž jakákoliv třída/funkce která bude mít přístup k instanci requestu bude moci taktéž přes request posílat zprávy na tyto cíle.

Kapitola 10

Popis a implementace komunikace s VCS a databází

V rámci této kapitoly si popíšeme, jak naše aplikace komunikuje s používaným externím softwarem. Protože komunikace silně závisí na použitých programovacích postupech, které byly zvoleny při tvorbě programu, v této kapitole taktéž dojde k nástinu implementace této komunikace.

10.1 Obecné použití rozhraní

V rámci PoC je řešena komunikace s dvěma externími systémy. Jedná se o VCS Git a databázi Microsoft SQL Server. Oba tyto systémy fungují nezávisle na naší službě, k přístupu je třeba využít poskytované API, které je zprostředkované knihovnami.

V rámci kódu jsou vytvořené pro přístup ke Gitu a databázi rozhraní, jež jsou následně implementovány konkrétními třídami. Toto umožňuje jednodušší změnu implementace (například při použití jiné knihovny), například při použití jiného než lokálního přístupu k VCS Git, nebo jiné databáze než MSSQL. Ostatní třídy pracují pouze s těmito rozhraními, čímž jsou odstíněny od detailů použité implementace. Příkladem může být převádění výjimek specifických pro databázovou knihovnu na vlastní definované typy výjimek, které mohou být použity i při práci s potenciálními jinými knihovnami a ostatní třídy je mohou bezpečně odchytávat pomocí použití definovaného typu výjimky.

10.2 Git

Pro komunikaci s VCS Git byla vybrána knihovna LibGit2Sharp, která je C# verzí nativní C knihovny LibGit2 pro práci s lokálním repozitářem VCS Git. Takto knihovna pracuje s lokálním repozitářem na cestě knihovně předané a nabízí funkce pro provádění různých operací, jak by byly psány do příkazové řádky terminálu. [48]

Bohužel, jeden ze zásadních nedostatků knihovny je absence podpory pro vzdálené repozitáře připojené přes protokol SSH. Protože se jedná o důležitý protokol, který bude pravděpodobně užit v produkčním prostředí k napojení lokálního repozitáře uloženého na serveru služby na vzdálený repozitář Azure Repos, operace se vzdáleným repozitářem jsou implementovány pomocí volání příkazů “git xxx” v terminálu operačního systému.

Vytvořené rozhraní IGitConnection definuje funkce pro operace, které služba bude využívat. Nejedná se tedy o kompletní rozhraní pro užití knihovny. V případě nutnosti užití více operací

podporovanými VCS může být rozhraní rozšířeno. Implementované jsou operace Commit, Merge, Checkout (i checkout souborů), Reset, zjišťování validity a SHA větve/commitu, získávání seznamu změn mezi commity a hledání společné merge base. Z operací se vzdáleným repozitářem jsou implementovány operace Push, Pull a Fetch, kde ovšem push a pull používají vynucené přepsání lokální/vzdálené větve.

Všechny funkce rozhraní pracují pouze s názvem commitu/větve, a ne s objekty poskytovanými knihovnou kvůli univerzálnosti rozhraní bez vazeb na použitou knihovnu.

Díky tomu, že připojení k repozitáři pracuje pouze s lokálními soubory, v kódu služby je vytvořeno pouze jedno připojení pro užití během celého chodu aplikace.

10.3 Databáze MSSQL

Pro napojení na DBMS SQL Server jsou použity dvě knihovny, System.Data.SqlClient a Microsoft.SqlServer.SqlManagementObjects.

System.Data.SqlClient je knihovna, která je součástí .NET frameworku (je ovšem distribuována samostatně). Tato knihovna slouží ke komunikaci s databází MSSQL. V základu pro práci s databází normálně stačí pouze tato knihovna. Má ovšem nevýhodu v podobě nutnosti zasílání skriptů jako jednotlivých dávek.

Protože pracujeme s databázovými skripty, které jsou primárně vyvíjeny v dedikovaném vývojovém prostředí, obsahují skripty mnohdy i oddělovače dávek (příkazy GO). Tyto oddělovače MSSQL nerozpoznává, a pokud pošleme na server celý skript i s oddělovači (jako jednu dávku), dojde k chybě, protože se jedná o neplatný skript. Toto můžeme vyřešit buďto ručním rozdělováním skriptů do jednotlivých dávek nebo využitím knihovny, co to bude řešit za nás.

Microsoft.Sql.SqlManagementObject je knihovna vyvíjená Microsoftem, která je postavena na knihovně System.Data.SqlClient a rozšiřuje její užití o SQL Server management objekty (SMO). Ty jsou navrženy k složitějším operacím určeným ke správě MSQQL, podobné, jaká je umožněna přes dedikované vývojové prostředí. Jednou z výhod, kterou tato knihovna přináší je právě možnost spouštět skripty, které obsahují oddělovače dávek. [49] Proto v rámci implementace rozhraní jsou použity (převážně) funkce, které poskytuje tato knihovna.

Vytvořené rozhraní pro napojení na databázi IDatabaseConnection podobně jako rozhraní pro Git definuje pouze funkce pro operace, které jsou používány službou. Těch je jenom malé množství, protože oproti jiným aplikacím nepotřebujeme provádět základní SQL operace, ale pouze spouštíme skripty, u kterých nás pouze zajímá, zda jejich vykonání proběhlo v pořádku. Výjimkou je volání procedury k získání dat z EventLogu. Definované rozhraní zahrnuje operace pro získání a úpravu časového limitu pro vykonání skriptů, spuštění skriptu, spuštění dotazu s vrácením jeho výsledného datasetu (a zavolání procedury s výsledkem), zahájení a ukončení transakce, zjištění informace, zda je transakce aktivní a změnu aktuálně používané databáze.

Protože komunikace probíhá po síti, je dobré nedržet po celou dobu chodu aplikace otevřené připojení k databázi. Užití implementace připojení je tedy vytvářeno pro každý proces nasazení zvlášť, po jehož ukončení jsou používané prostředky připojení uvolněny.

Kapitola 11

Popis procesu nasazování

Proces nasazování jsou kroky, které jsou ve službě provedeny, aby byl vykonán požadavek nasazení. Část procesu, o které zde bude řeč, tedy funguje nezávisle na formě a pořadí získávání a zpracování požadavků (například formou fronty), tento postup byl popsán v dřívější kapitole 9. Popisovaný proces tedy probíhá zvenku pouze jako zavolání funkce nasazení, která bere jako argument merge request se všemi potřebnými informacemi, a vrací pravdivostní hodnotu o tom, zda bylo nasazení úspěšné. Postupy použité v tomto procesu vychází zejména z řešení dříve zanalyzovaných komplikací (kapitola 6), které mohou nastat v průběhu procesu a proces na ně musí umět náležitě reagovat.

V průběhu celého výkonu nasazování dochází k průběžnému zasílání zpráv o průběhu dokončení jednotlivých podoperací přes rozhraní cíle zpráv, které je funkci předáno jako atribut instance merge requestu.

Jako první krok si získáme a uložíme počáteční čas operace, tento čas budeme dále využívat při kontrole EventLogu. Následuje odeslání požadavku na synchronizační službu k pozastavení její činnosti. Proces čeká na zpětnou vazbu od služby, že zpracovala všechny změny přijaté do okamžiku přijetí požadavku k pozastavení. Teprve po získání odpovědi proces pokračuje dál. Tímto jsme vešli do části procesu, během kterého víme, že nebude docházet ve vzdáleném repozitáři ke změnám. Protože synchronizační služba zatím nemá vystavené API pro tento požadavek, pozastavení (a následné spuštění, které bude později zmíněné) je v kódu pouze naznačené komentářem a není v rámci PoC implementované.

Díky absenci dodatečných změn si nyní stáhneme novou podobu vzdáleného repozitáře do lokálního, konkrétně cílovou větev a zdrojovou větev (či zdrojový commit, dále je zmiňována pouze větev). Pro účely případného vrácení změn na cílové větvi si uložíme SHA původního vrcholu cílové větve pro pozdější referenci.

Následně provedeme první kontrolu konfliktů změn – zjistíme seznam změněných souborů v cílové větvi a zdrojové větvi za použití jejich společné merge base. Pokud existuje soubor, ve kterém jsou změny v obou historiích, došlo ke konfliktu a merge zrušíme.

Pokud ke konfliktům nedošlo, stále mohou být konflikty přítomny (například v upravovaných tabulkách jednorázovými scripty), ty zjistíme až po pokusu o nasazení. Proto nyní přejdeme k nasazení scriptů na databázi.

S pomocí získaného seznamu změněných souborů na zdrojové větvi si načteme všechny změněné scripty. Pokud je vyplněný soubor RELEASE_ORDER, seřadíme vypsané skripty dle pořadí v souboru a zbytek připojíme za ně. V seznamu taktéž odstraníme soubor RELEASE_ORDER, protože se jedná o konfigurační soubor, nikoliv soubor s databázovým skriptem. Následuje vytvoření připojení k databázi a začátek transakce. Poté nasazujeme jednotlivé scripty v dříve určeném pořadí. Pokud se cokoliv při nasazování pokazí, merge zrušíme (bez nasazování zbytku). To může zahrnovat ztrátu připojení k databázi, neplatný skript, nebo vypršení časového limitu. Časový

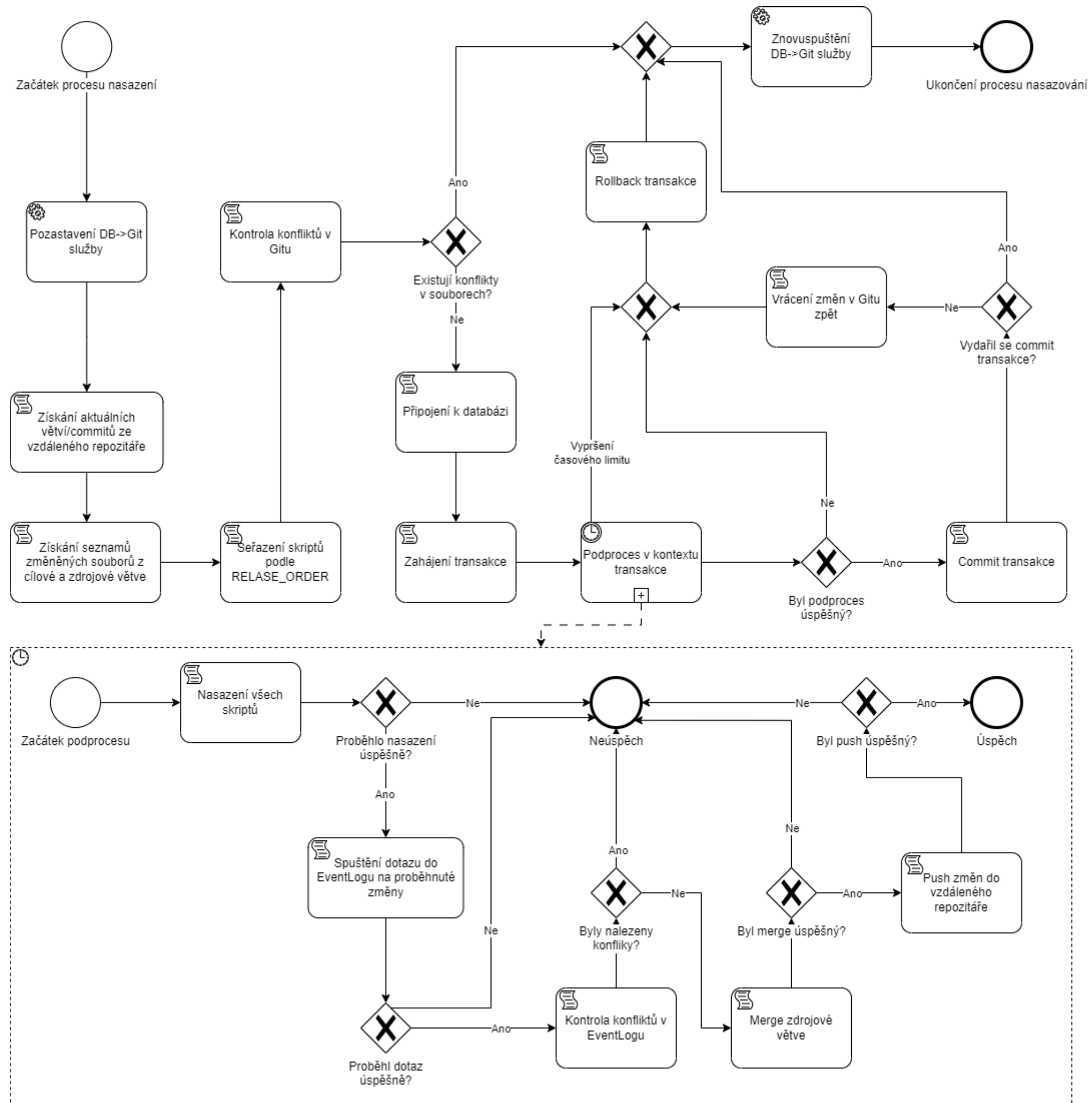
limit je zde řešen tak, že se pro každý spouštěný skript nastaví časový limit rovný zbývajícímu celkovému času pro nasazení.

Transakci ještě nepotvrzujeme (neprovádíme commit transakce), protože proběhne druhá kontrola konfliktů. Pro tuto kontrolu spustíme proceduru, která vrátí dataset se všemi změněnými databázovými objekty od zaznamenaného času na začátku procesu. Tyto změny rozdělíme do skupin změn provedených nastaveným uživatelem služby (změny provedené službou v rámci tohoto nasazení) a změny provedené jinými uživateli. Poté porovnáme dvojice "reálné změny provedené námi" – "změny provedené ostatními" a "reálné změny provedené námi" – "změny provedené na cílové větvi". Pokud v libovolné z dvojic existuje společný databázový objekt, provedené změny jsou konfliktní a merge proto zrušíme.

V posledním kroku změny ve zdrojové větvi vyčistíme od změn v dočasných souborech, v rámci PoC implementace je řešeno čistění vytvořením nového commitu, co změny vrátí zpět. Tyto změny následně sloučíme do hlavní větve a změny odešleme do vzdáleného repozitáře.

Tímto proběhl celý proces nasazení, s různými možnými výsledky. Pokud se nasazení zdařilo, pak databázovou transakci potvrdíme. Pokud se toto potvrzení transakce nepovedlo, změny v transakci vrátíme zpět (transakci zrušíme) a vrátíme zpět změny, které jsme provedli v repozitáři (a odešleme je do vzdáleného repozitáře). Pokud se nasazení nepovedlo (bylo zrušeno v průběhu), transakci taktéž zrušíme.

Posledním krokem (nezávisle na výsledku) je odeslání požadavku na znovuspuštění synchronizační služby, kterou jsme na začátku procesu nasazení pozastavili. Tím proces nasazování končí a z funkce vracíme výsledek procesu, který může být následně odeslán přes API, jak bylo zmíněno v procesu zpracování požadavku.



Obrázek 11.1 Proces nasazení databázových změn

Kapitola 12

Implementace aplikace, cesta požadavku skrz třídy

V rámci této kapitoly si detailně vysvětlíme, jak je služba implementovaná a popíšeme jednotlivé třídy, které jsou použity v průběhu zpracování požadavku.

12.1 Program

Program je hlavní třída služby, která obsahuje hlavní funkci Main. V rámci hlavní funkce si vytvoříme instance všech objektů, které jsou potřeba pro chod aplikace, jako poskytovatele parametrů, factory, mergery nebo consumery requestů. Instance mohou být použity pro zavolání jejich funkcí nebo jsou většinou předány dalším třídám jako parametry jejich konstruktorům.

V rámci PoC jsou tyto instance tříd vytvořeny přímo ve funkci Main, v budoucí produkční verzi aplikace bude tato logika nahrazena užitím frameworku pro Dependency injection. Všechny třídy jsou proto navrženy tak, aby pro svůj chod neužívaly konkrétní implementace (konkrétní třídu), jejichž instance si například vytváří za chodu (stále užito, kde to smysl dává kvůli pevnému navázání na implementaci v dané třídě), ale instance k použití jsou jim předány při vytvoření.

Nejdůležitější akce, které probíhají ve funkci Main jsou spuštění zpracovávání front jednotlivých consumerů požadavků, a spuštění produkce požadavků, což v aktuální implementaci znamená spuštění cyklu pro čtení řádků s požadavky z terminálu v rámci ConsoleRequestSource. V budoucí implementaci toto bude nahrazeno spuštěním webového serveru, ze kterého budou při zaslání požadavků do front předávány požadavky k vykonání.

12.2 IMessageable, IStateMessageable

Tyto rozhraní jsou užity jako rozhraní, které třídu umožňují použít v kontextu, kdy je třeba zaslat zprávy o průběhu požadavků.

IMessageable je rozhraním pro zasílání obyčejných zpráv (textu), například logových zpráv.

IStateMessageable je rozhraním pro zasílání stavových zpráv o požadavku, tedy zahájení zpracování požadavku a ukončení zpracování požadavku spolu s jeho výsledkem (úspěch/neúspěch).

12.3 IReponseTarget a jeho implementace

Toto rozhraní a jeho implementace (IdentifiedResponseTarget, ConsoleResponseTarget, AzureDevOpsTarget, CompositeResponseTarget) slouží jako cíle zasílání informačních zpráv o průběhu požadavků. Třídy reprezentují konkrétní cíle, na které jsou zprávy pro požadavek zasílány. Cíl je zpravidla vytvořen pro každý požadavek zvlášť (i když jsou parametry cíle shodné) pro korektní zasílání stavových zpráv.

IReponseTarget je rozhraní, které je sloučením rozhraní IMessageable a IStateMessageable. Neslouží pouze jako sloučení těchto rozhraní, ale toto rozhraní implementují pouze finální cíle pro korektní užití v kontextu. Tzn. v jiných třídách (které taktéž implementují obě rozhraní), které nejsou finálními cíli, toto rozhraní užito není.

IdentifiedResponseTarget je abstraktní třída implementující IResponseTarget, která umožňuje pro cíl definovat identifikátor (string), který je ve zprávách použit. Toto může být například názvem a identifikátorem požadavku, se kterým je tento cíl používán.

ConsoleResponseTarget je třída dědící z IdentifiedResponseTarget, která reprezentuje cíl zpráv jako výpis na standardní výstup. Před každou zprávu připojuje použitý identifikátor, aby bylo možné na výstupu rozpoznat, ke kterému požadavku zprávy patří. Pro zasílání stavových zpráv je využita stejná logika jako pro zasílání textových zpráv

AzureDevOpsTarget je třídou implementující IResponseTarget (nevyužívá identifikátor). Třída není v rámci PoC užita, ale demonstruje, jak bude služba v budoucí verzi implementace komunikovat s webovou službou AzureDevOps (konkrétně Azure Pipelines). Zasílání zpráv je implementováno jako volání REST api dané webové služby. U tohoto cíle se liší způsob zasílání stavových a textových zpráv, protože api používá pro informaci o stavu a zaslání logových zpráv různé URL adresy a různou podobu těla požadavku.

CompositeResponseTarget demonstruje možnost užití více cílů dohromady (například standardní výstup a Azure Pipeline). Toto je implementováno předáním seznamu všech cílů v konstruktoru. Při zasílání zpráv je následně voláno dané zaslání zprávy pro každý cíl zvlášť.

12.4 RestHelpers

Statická třída RestHelpers poskytuje funkce, které jsou používané pro zjednodušení volání REST api v rámci služby. Aktuálně je součástí pouze funkce PostData, která slouží k zaslání požadavku POST, což je využito třídou AzureDevOpsTarget.

12.5 Request, MessageableReqeust

Tyto dvě abstraktní třídy jsou předky pro všechny typy požadavků, které budou předávány v rámci služby mezi jednotlivými třídami, které je budou zpracovávat.

Request je pouze základní třídou bez jakékoliv logiky, která je určena pro použití ve frontách, kam mohou být požadavky různých typů zařazeny.

MessageableRequest je potomkem Requestu, který obsahuje instanci třídy implementující rozhraní IResponseTarget (předaným v konstruktoru), která je použita pro zasílání zpráv na daný cíl. Třída taktéž implementuje rozhraní IMessageable a IStateMessageable pro použití v různých kontextech, kdy je třeba zasílat zprávy a pro zjednodušení volání metod zasílání zpráv (volání je předáno cíli). Třída neimplementuje rozraní IResponseTarget, aby nešly požadavky vkládat do sebe jako cíle zpráv (ze sémantických důvodů).

12.6 EchoRequest, ShutdownRequest, ClearMergeQueueRequesst, MergeRequest

Tyto třídy reprezentují jednotlivé požadavky, které je možné v rámci služby zpracovávat. Všechny třídy dědí z třídy MessageableRequest pro zasílání zpráv informujících o průběhu zpracování. Součástí požadavku (ve formě atributů) jsou veškeré informace specifické pro daný požadavek, které jsou třeba pro jeho zpracování (atributy požadavku). Některé požadavky nebyly v práci zmíněny a jsou implementovány jako demonstrace dalších možných požadavků, které služba může zpracovávat.

EchoRequest je požadavek, jehož zpracování spočívá v zaslání textu (který je atributem) na cíl.

ShutdownRequest je požadavek, který vypne službu. Atributem je boolovská hodnota Immediate označující, zda má k vypnutí dojít okamžitě, tedy bez zpracování zbytku požadavků ve frontě. V takovém případě, pokud jsou nějaké požadavky ve frontě zařazené, všechny jsou zrušeny. Služba po zpracování tohoto požadavku nepřijímá žádné další požadavky.

ClearMergeQueueRequest (CMQR) je požadavek, sloužící k vyčištění fronty merge requstů. Neovlivňuje to požadavky, které ještě ve frontě nejsou zařazeny (byly odeslány po CMQR), pouze již zařazené a zatím nezpracované. Všechny takovéto požadavky ve frontě jsou zrušeny.

MergeRequest je požadavkem na nasazení databázových skriptů z větve/commitu. Její identifikátor je atributem požadavku. Dalším atributem je časový limit v milisekundách na vykonání požadavku (od začátku zpracování). Zpracování tohoto požadavku je hlavním zaměřením této práce, vysvětleno je v ostatních částech.

12.7 RequestSource, ConsoleRequestSource

Tyto třídy slouží jako zdroj požadavků. Třídy obsahují logiku, která vytváří instance dříve zmíněných požadavků s jejich parametry. Tyto požadavky jsou následně přidány do fronty ke zpracování.

RequestSource je základní abstraktní třída pro zdroj požadavků. Její konstruktor bere instanci IRequestConsumeru, který zpracovává požadavky, a ten je uložen ve třídě jako její atribut. Třída používá typový parametr TRequest, který určuje, jaké typy požadavků jsou generovány. Použitý Consumer musí požadavky tohoto typu podporovat. Třída dále definuje abstraktní metody pro spuštění a ukončení generování požadavků.

ConsoleRequestSource je třída dědící z RequestSource. Tato třída je zdrojem požadavků ze standardního vstupu, na který jsou psány příkazy, jejichž syntaxe je logikou třídy definovaná. Třída generuje různé typy požadavků, proto je jako typový parametr použitý obecný Request. Požadavky generované touto třídou mají jako cíl zpráv vytvořený ConsoleResponseTarget (standardní výstup).

12.8 IGitConnection, LocalGitConnection

IGitConnection je rozhraní, které umožňuje službě provádět operace nad repozitářem Git. Toto rozhraní (a část implementace lokálního připojení) bylo popsáno v kapitole „Implementace komunikace s VCS a Databází“ (podkapitola 10.2). Rozhraní dědí z rozhraní IDisposable pro bezpečné uvolňování používaných prostředků.

LocalGitConnection je třída implementující rozhraní IGitConnection, která pracuje s lokálním repozitářem Git. Pro komunikaci s repozitářem využívá knihovnu LibGit2Sharp a příkazy přes příkazovou řádku OS. Třída jako parametr konstruktoru bere cestu k lokálnímu Git repozitáři, se kterým následně umožňuje pracovat.

12.9 IDatabaseConnection, MssqlDatabaseConnection

IDatabaseConnection je rozhraní, které umožňuje službě provádět operace nad databází. Toto rozhraní (a část implementace připojení k databázi MSSQL) bylo popsáno v kapitole "Implementace komunikace s VCS a Databází" (podkapitola 10.3). Rozhraní pracuje pouze s textem skriptů, které vykonává. Rozhraní dědí z rozhraní IDisposable pro bezpečné uvolňování používaných prostředků.

MssqlDatabaseConnection je třída implementující rozhraní IDatabaseConnection, která pracuje s databází MSSQL. Pro komunikaci s databází využívá knihovny SqlClient a SqlManagementObjects. Třída bere jako parametry informace nutné k navázání připojení k databázi neboli URI serveru, DB, ke které se při připojení k serveru připojit a kombinaci uživatelského jména a hesla pro přihlášení. Užití třídy je navrženo tak, že připojení k databázi je navázáno ve chvíli, kdy je spuštěn první příkaz nad databází. Díky tomu není třeba spojení explicitně navazovat, a připojení není drženo od chvíle, kde je instance třídy vytvořena (čímž se sníží doba, po kterou je připojení drženo otevřené).

12.10 IDatabaseConnectionFactory a její implementace

Toto rozhraní a její implementace MssqlDatabaseConnectionFactory slouží k užití jinými třídami, které chtějí v rámci svého chodu vytvářet nové připojení k databázi. Protože to znemožňuje instanci třídy předat v konstruktoru (například pro každé volání funkce třídy je tvořena nová instance), existují tyto třídy, které umožňují dané třídě vytvářet instance, ale s možností změny implementace.

To je vyřešeno tak, že v konstruktoru dané třídy je předaná třída implementující obecné rozhraní IDatabaseConnectionFactory, které poskytuje funkci, která vrací nově vytvořenou instanci obecného připojení. V závislosti na použité implementaci se může jednat například o připojení k různým typům databází, či připojení s různým způsobem autentizace.

Rozraní IDatabaseConnectionFactory je obecným rozhraním pro třídy, které generují připojení k databázi. Obsahuje pouze jednu funkci CreateConnection, která vrací obecnou instanci IDatabaseConnection (jejíž konkrétní implementace závisí na implementaci použité factory).

Třída MssqlDatabaseConnectionFactory implementuje rozhraní IDatabaseConnectionFactory a je určena pro vytváření připojení k databázi MSSQL. V konstruktoru bere jako parametry stejné informace k připojení jako MssqlDatabaseConnection, které ukládá jako atributy, a používá je při vytváření nových instancí připojení.

12.11 TimoutChecker

TimoutChecker je třída, která reprezentuje časovač pro získání zbývajícího času od jeho spuštění. Při vytvoření instance je jí předaný čas v milisekundách, který reprezentuje čas, který má být časovačem odpočítáván. Spuštěním se vytvoří nová instance stopek (třída Stopwatch) a spustí se. Časovač je možné vynulovat zastavením (instance stopek se odstraní). Zbývající čas je možný získat getterem (Timeout), který zjišťuje rozdíl mezi nastaveným časem a časem zaznamenaným stopkami (omezeným hodnotami mezi 0 a počátečním časem).

12.12 Script

Script je record, který obsahuje rozšířené informace o databázovém skriptu, který má být vykonán. Kromě samotného textu skriptu obsahuje taktéž název (například pro užití v logových zprávách) a databázi, na které má být skript vykonán.

12.13 DatabaseObject

DatabaseObject je record, který reprezentuje identifikátor databázového objektu o třech částech – databáze, schema a název.

12.14 IDatabaseScriptRunner, ConnectionScriptRunner

Toto rozhraní a jeho implementace slouží ke spouštění databázových skriptů v kontrolovaném prostředí. Toto zahrnuje zejména možnost spuštění pouze jedné transakce a možnost nastavení časového limitu pro celou transakci, který se propisuje do časových limitů pro jednotlivé prováděné operace.

IDatabaseScriptRunner je rozhraní, které vystavuje funkce pro provádění komplexnějších operací nad databází. Kromě základních operací (spuštění/ukončení transakce, spuštění jednoho skriptu) umožňuje pro transakci nastavit časový limit a spouštět více skriptů předaných seznamem v rámci jednoho volání.

Rozhraní taktéž zjednodušuje práci s připojením k databázi pomocí dvou postupů. Zaprvé, funkce signalizují úspěšnost provedení operace návratovou hodnotou (pravdivostní hodnota u obyčejných skriptů, či hodnota null u datasetu s výsledkem dotazu). Tímto je volající třída odstíněna od případných výjimek, které při volání mohou nastat. Zadruhé, funkce berou jako parametry nikoliv pouze texty skriptů, ale instance recordu Script, který může obsahovat dodatečné informace, jako databázi, nad kterou je třeba skript spustit, nebo jeho jméno pro případnou identifikaci. Těchto informací může implementace rozhraní využívat různým způsobem.

ConnectionScriptRunner implementuje rozhraní IDatabaseScriptRunner. Jako parametry konstruktoru bere připojení k databázi (IDatabaseConnection), které za chodu užívá, cíl obyčejných (logových) zpráv pro zasílání průběžných informací o vykonávaných skriptech (například chyby, které nastaly), a informaci o tom, zda má při mazání instance zařídit uvolnění prostředků používaných připojením.

K řešení časového limitu pro celou transakci používá TimoutChecker, který vytvoří při začátku transakce, pro kterou je nastaven časový limit, a poté před každým spuštěním skriptu nad databázovým připojením nastaví časový limit pro vykonání skriptu na zbývající čas získaný z TimoutCheckeru. Implementace dále využívá cíle zpráv předaného v konstruktoru pro zasílání logových informací o průběhu na tento cíl. Informace z instancí skriptů, které jsou předávány při volání funkcí, využívá k identifikaci daného skriptu v logových zprávách a taktéž k nastavení správné databáze před zavoláním provedení samotného skriptu.

12.15 IDatabaseScriptRunnerFactory a její implementace

Toto rozhraní a její implementace ConnectionScriptRunnerFactory odpovídá dvojici IDatabaseConnectionFactory a MssqlDatabaseConnectionFactory ve funkčnosti a účelu. Taktéž se jedná o rozhraní a jeho implementaci, sloužící k využití třídami, které v rámci jejich funkčnosti vytváří nové instance tříd (zde implementací IDatabaseScriptRunner), a toto nám umožňuje určit konkrétní implementaci vytvářené instance předáním konkrétní implementace IDatabaseScriptRunnerFactory konstruktoru třídy.

Rozraní IDatabaseScriptRunnerFactory je obecným rozhraním pro třídy, které generují instance třídy pro spouštění databázových skriptů. Obsahuje pouze jednu funkci CreateScriptRunner, která vrací obecnou instanci IDatabaseScriptRunner (jejíž konkrétní implementace závisí na implementaci použité factory). Tato funkce bere taktéž jeden nepovinný parametr, kterým je cíl zasílání logových zpráv runneru (IMessageable).

Třída ConnectionScriptRunnerFactory implementuje rozhraní IDatabaseScriptRunnerFactory a je určena pro vytváření runnerů používajících IDatabaseConnection. V konstruktoru bere jako

parametr instance IDatabaseConnectionFactory, protože pro vytváření nových runnerů potřebuje vždy vytvořit novou IDatabaseConnection, což umožňuje dříve zmíněná factory. Protože tyto připojení nejsou vytvářeny v kontextu, který by automaticky bezpečně uvolňoval prostředky připojení, vytvořený IConnectionScriptRunner bude parametrem v konstruktoru instruován, aby při jeho zničení prostředky uvolnil.

12.16 Databázové výjimky

Tři třídy databázových výjimek (DatabaseConnetionException, DatabaseScriptException a DatabaseTimeoutException) dědí z obecných výjimek (Exception) a jsou použity jako výjimky nezávislé na konkrétní implementaci databázového připojení. V logice tříd databázového připojení (např. MssqlDatabaseConnection) mohou být za chodu vyvolány výjimky specifické pro použitou knihovnu, které jsou následně převedeny na tyto, aby šly dále obecně používat dalšími třídami v rámci služby (například třídou ConnectionScriptRunner).

Výjimka DatabaseConnectionException je vyvolána situací, kdy se třídě databázového připojení nepodaří připojit k databázi, nebo když spuštění skriptu selže z důvodu, že připojení k databázi selhalo (došlo k odpojení od databáze například ztrátou internetového připojení).

Výjimka DatabaseTimeoutException je vyvolána situací, kdy dojde k vypršení časovému limitu. Tento časový limit může být různého charakteru v závislosti na konkrétní implementaci, v rámci aktuální implementace se jedná o nastavený časový limit na provedení databázového skriptu.

Výjimka DatabaseScriptException je obecnou výjimkou, která je vyvolána při spuštění databázového skriptu, kdy dojde k výjimce, ale nejedná se o situace zmíněné výše. Může se jednat například o nedostatečná práva k provedení operace, nebo o spuštění nevalidního skriptu.

12.17 PathDatabaseObjectExtractor

PathDatabaseObjectExtractor je třída, která usnadňuje práci s extrahováním instance DatabaseObject vyvozené z cesty v rámci repozitáře Git.

Třída funguje tak, že v konstruktoru jsou předány formátovací stringy, které označují umístění jednotlivých částí identifikátoru v rámci cesty pomocí oddělovačů jednotlivých segmentů cesty (například složky oddělené symbolem /) a pozici v seznamu rozdělených částí. Formátovací string je modifikovatelný na různé oddělovací symboly a umožňuje do hloubky prohledávat jednotlivé podsegmenty pomocí dalších oddělovačů.

Třída poskytuje funkce, které pro zadanou cestu vrátí (dle nakonfigurovaného formátovacího stringu) danou část identifikátoru (databáze, schema, jméno), které jdou poté složit do celého DatabaseObjectu.

12.18 MergeHelpers

MergeHelpers je statická třída, která poskytuje statické funkce, které mohou posloužit k různým operacím, které jsou třeba provést při procesu nasazování/merge změn ve VCS.

Implementované funkce aktuálně slouží k provádění složitějších stringových operací, jako například získávání DatabaseObjectu z cesty, seznamu databázových složek a instance PathDatabaseObjectExtractor nebo cesty ke složce databáze z cesty a seznamu databázových složek.

12.19 IDatabaseMergerParams, IGitMergerParams, MergerParamsRecord

Rozhraní IDatabaseMergerParams a IGitMergerParams jsou rozhraní, které umožňují třídám Mergerů číst z poskytovatele parametrů parametry, které následně užívají za svého chodu. To je zajištěno definicí Getterů v rámci těchto rozhraní pro jednotlivé atributy.

MergerParamsRecord je jednoduchou implementací obou rozhraní za použití recordu, kde jsou všechny parametry definovány staticky v konstruktoru.

12.20 IMerger, GitMerger, DatabaseMerger

IMerger je společné rozhraní pro třídy, které definuje jednu veřejnou metodu – Merge. Tato metoda bere jako parametr Merge request. Implementace metody (a třídy) slouží k nasazení/sloučení změn, které jsou evidované ve VCS Git pod danou větví/commitem, jejíž název je součástí předaného MR.

GitMerger je třída implementující rozhraní IMerger. Třída v konstruktoru bere jako parametry parametry pro merge (rozhraní IGitMergerParams) a instanci připojení k VCS Git (IGitConneciton).

Třída se specializuje pouze na sloučení změn v rámci VCS Git bez jakékoliv interakce s databází, ovšem stále s dodržením pravidel konfliktů, které mohou ve VCS nastat, které byly definovány ve dřívější kapitole 6 (v rámci VCS se jedná o jakékoliv změny ve stejném souboru) a s čištěním výsledných slučovaných změn. Použité postupy pro sloučení změn již byly vysvětleny v dřívější kapitole 11. Třída taktéž má funkci pro zjištění, zda jde MR sloučit do cílové větve.

DatabaseMerger je třída implementující rozhraní IMerger. Tato třída (a její funkce Merge) slouží ke kompletnímu sloučení změn a nasazení databázových skriptů z předaného MR v parametru funkce. K tomu třída bere jako parametry konstruktoru parametry pro merge (IDatabaseMergerParams), připojení k VCS Git (IGitConnection), instanci třídy GitMerger, která zpracovává merge ve VCS, a factory pro získání runnera databázových skriptů (IDatabaseScriptRunner).

I přes užití třídy GitMerger tato třída využívá VCS k provádění některých operací, proto je připojení potřeba. V průběhu zpracování požadavku třída vytváří nový runner (a spolu s ním i připojení k databázi), k čemuž je využitá předaná factory.

V rámci implementace je celkový proces nasazení členěn do 3 funkcí (2 z nich jsou privátní volané jinými funkcemi), která každá pracuje v určitém kontextu – první provádí celý merge, druhá operace v rámci pozastavené DB -> Git služby a třetí v rámci funkční transakce nad databází. Kompletní proces nasazení byl detailně vysvětlen v dřívější kapitole 11.

12.21 IRequestConsumer a jeho implementace

IRequestConsumer je rozhraním, které je použito pro frontu požadavků, do které je možné přidávat požadavky, a fronta dané požadavky zpracovává. V závislosti na konkrétní implementaci toto může probíhat synchronně (v rámci stejného vlákna) či asynchronně (v jiném vlákně). Rozhraní užívá typový parametr, který určuje, které typy požadavků je možné frontou zpracovávat. Rozhraní poskytuje metody pro přidání požadavku do fronty, vyčištění fronty, a označení fronty jako kompletní (nebude již přijímat další požadavky ke zpracování – toto slouží k ukončení činnosti vlákna).

DataFlowRequestConsumer je abstraktní třídou implementující rozhraní IRequestConsumer. Třída si stále ponechává typový parametr, který byl použitý v rozhraní (třídu lze dále specializovat). Tato implementace už řeší konkrétní formu zpracování fronty, a to jako asynchronní frontu, řešenou pomocí třídy BufferBlock z knihovny DataFlow (použití vysvětleno v dřívější kapitole 9).

Tato implementace implementuje všechny metody definované rozhraním a přidává asynchronní metodu ConsumeAsync, která slouží k asynchronnímu zahájení zpracování fronty. Dále definuje jedinou (neimplementovanou) abstraktní metodu ConsumeItem, která slouží ke zpracování jednoho požadavku v rámci fronty, což může být implementováno konkrétním potomkem za použití společné logiky zpracování fronty z této třídy.

MergeRequestConsumer je třída, která je potomkem třídy DataFlowReqeustConsumer a specializuje se na zpracování MergeRequestů. Instance třídy je ve službě použita jako druhá fronta, které jsou předávány všechny MR ke zpracování. Konstruktor této třídy bere jako parametr IMerger, který je použit k nasazení/merge změn v rámci MR (toto umožňuje například v rámci testování použít pouze GitMerger). V rámci implementace služby je danou třídou DatabaseMerger. Samotné zpracování jednoho požadavku je implementováno jako volání funkce Merge na předaném IMergeru.

Protože tato fronta provádí většinu operací, ve kterých může dojít k chybám (které nebyly předpokládané a nebyly v rámci implementace odchyceny), volání funkce Merge je obaleno try-catch blokem, který zachycuje všechny druhy výjimek a chybové hlášky předává cíli zpráv z MR k informování uživatele (který následně informuje správce služby). Toto zahrnuje plnou informovanost o případných neošetřených výjimkách, které je v rámci služby třeba opravit, a zároveň to zajišťuje, že při výskytu takovéto výjimky nedojde k přerušení chodu pracovního vlákna, které zpracovává frontu (což by vyústilo v nefunkčnost služby, než bude restartovaná).

RequestProcessor je třída, která je potomkem třídy DataFlowRequestConsumer. Fronta se nespecializuje na žádný typ požadavků neboli umožňuje zpracovávat všechny druhy (jako typový parametr je použit obecný Request). Tato fronta je ve službě použita jako prostředník mezi zdrojem požadavků (RequestSource) a frontou MergeRequestů (MergeRequestConsumer). Tato fronta má zpracování požadavku implementované tak, napřímo zpracovává všechny požadavky s výjimkou MR, které předává ke zpracování instanci třídy MergeRequestConstumer. Tato instance je třídě předána jako parametr konstruktoru. Některé požadavky taktéž ovlivňují chod fronty MR tím, že na ni volají její metody, například signalizaci konce zpracování SetComplete. Detaily způsobu využití těchto dvou front jsou vysvětleny v dřívější kapitole 9.

12.22 Cesta požadavku skrz třídy

Cesta požadavku začíná v třídě RequestSource (například ConsoleRequestSource), kde je požadavek (Request) konkrétního typu vytvořen na základě uživatelského vstupu. Pro plný průchod procesem bude tímto požadavkem MergeRequest. Pro požadavek se taktéž v této třídě vytvoří IResponseTarget, na který jsou v průběhu zpracování požadavku zasílány informace o jeho průběhu. Požadavek je poté předán zdrojem do fronty obecných požadavků třídy RequestProcessor.

Požadavek čeká ve frontě, než na něj dojde řada ke zpracování. Pokud by se nejednalo o MR, tato třída by požadavek zpracovala a tím by došlo ke konci průchodu službou. Místo toho je požadavek zařazen do fronty merge requestů třídy MergeRequestConsumer.

Požadavek opět čeká v této další frontě, než na něj dojde řada ke zpracování. Jakmile na něj přijde řada, je spuštěno zpracování požadavku, kterým je zavolání funkce Merge třídy DatabaseMerger, tato funkce představuje samotný proces nasazení.

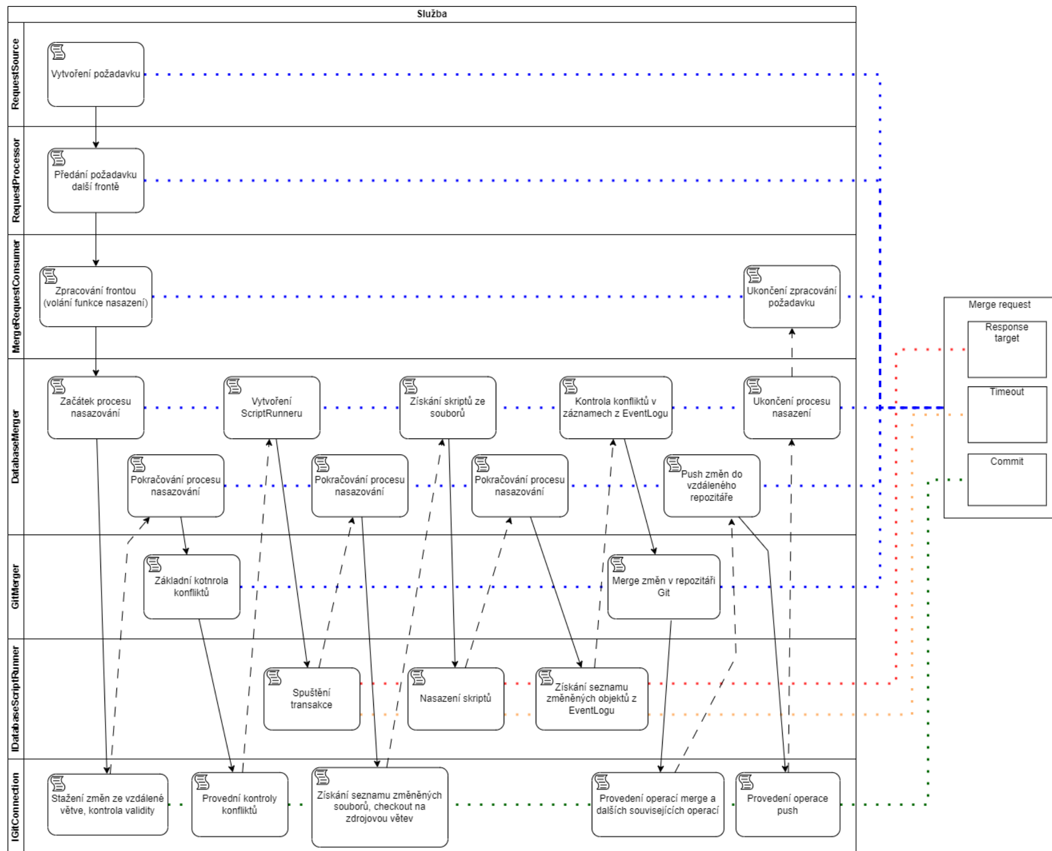
Průběh nasazování řídí DatabaseMerger, který volá funkce z jiných tříd. Těmto funkcím (či vytvořeným instancím) předává informace z MR, což je buďto celý MR, název commitu (využívaným operacemi nad VCS) nebo kombinace IMessageable objektu (buďto samotný MR nebo ResponseTarget, který je součástí MR) a časového limitu (využívaných operacemi nad databází).

Takto jsou z funkce Merge napřímo volané funkce IGitConnection (s předaným commitem), funkce GitMergeru (s předaným MR), který následně volá funkce IGitConnection (s předaným commitem) a je vytvořena nová instance IDatabaseScriptRunneru, které je v konstruktoru předána instance implementující IMessageable a následně při vytvoření transakce je této instanci

runneru předaný časový limit, který je využíván v průběhu provádění skriptů. Implementace ConnectionScriptRunner navíc interně volá funkce připojení (IDatabaseConnection), ale do těch se nedostanou žádné informace z původní instance MR.

Konkrétní průběh nasazování byl detailně popsán v dřívější kapitole 11.

Na diagramu níže je znázorněn celkový průběh zpracování MR (za předpokladu, že je úspěšný) a jaké třídy využívají MR či jeho části v průběhu zpracování požadavku.



Obrázek 12.1 Průchod třídy MergeRequest aplikací a jejími třídami

Kapitola 13
Testování

Testování vytvořeného PoC služby je prováděno pomocí automatických unit testů a ručních systémových testů.

13.1 Unit testy

Unit testy jsou automatické testy, které testují funkčnost jednotlivých samostatných částí programu, jako jsou funkce nebo třídy. Vhodně napsané unit testy pomáhají zaručit, že daná část (například funkce) funguje, jak má, a tím zjednodušit hledání příčiny případných budoucích chyb v programu. Pro automatizaci unit testů existují různé frameworky, které tvorbu a následné spouštění testů usnadňují, jako například NUnit, XUnit, JUnit a další. [50]

Protože velká část služby nějakým způsobem interaguje s externím softwarem (například databáze či Git), je složité pro mnohé třídy Unit testy vytvořit. Pro otestování pomocí unit testů byly zvoleny třídy Script, MergeHelpers a PathObjectExtractor. Tyto třídy obsahují funkce, které pouze provádí deterministický převod vstupních parametrů na výstupní, proto jsou na unit testy vodné. Pro automatické testování služby unit testy jsou testy v testovém projektu tvořeny za pomocí frameworku NUnit.

U jiných tříd, které provádí různé zpracování (jako například implementace IRequestConsumer nebo IMerger), jde jejich vnitřní logiku otestovat pomocí mockovaných implementací rozhraní, které tyto třídy užívají, čímž lze nasimulovat pro účely testu deterministické odpovědi volaných funkcí. Návrh rozhraní (zejména konstruktorů) tříd dopředu počítal s tím, aby služba byla dobře otestovatelná i těmito způsoby, což je další důvod, proč mnohé třídy nepoužívají konkrétní implementace v jejich kódu, ale užité implementace pro danou třídu jsou předávány v konstruktoru.

V rámci implementace PoC nebyly tyto složitější unit testy vytvořeny, ale díky vhodnému návrhu je možné tyto testy přidat bez velkého zásahu do implementace tříd v budoucích iteracích vývoje služby.

13.2 Systémové testy

Systémové testy jsou druhem testů, které zhodnocují software jako celek, jestli funguje, jak má. End to end testy jsou podruhem systémových testů, které testují software v situacích (a způsobem), který napodobuje reálné použití daného softwaru v cílovém prostředí. Toto může zahrnovat komunikaci s databází, využití sítě, či interakci s jiným softwarem. (zdroj: typy testů)

V rámci naší služby jdou tyto testy použít pro otestování realistických situací, co mohou při provozu služby nastat. Vedle vhodných podmínek se jedná například o definované kompli-

kace, které mohou nastat (které byly zmíněny v dřívější kapitole 4). Takto můžeme vytvořit sadu testovacích scénářů, kdy, pokud všechny scénáře dávají při otestování očekávaný výsledek, služba funguje korektně. Tyto scénáře jsou definovány jako počáteční podmínky, akce potřebné k provedení testu, a výsledek, který je při provedení testu očekáván.

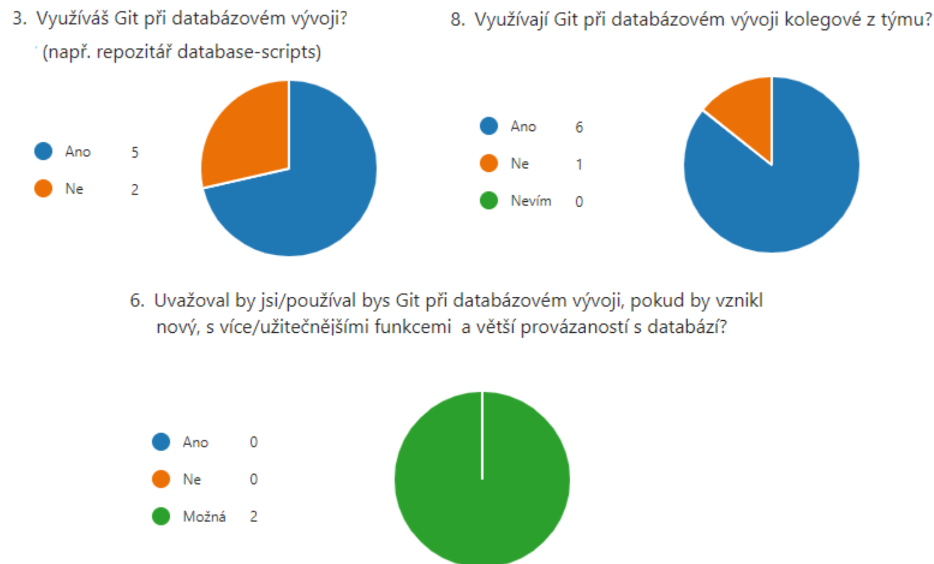
Testovací scénáře jsou vypsány v souboru Testing.md u zdrojového kódu služby. Aplikace byla těmito scénáři otestována, což potvrzuje její korektní funkčnost v těchto situacích.

Kapitola 14
Dotazník

V souvislosti s řízením se iterativním vývojem byl v rámci zhodnocení funkcí současných i budoucích proveden průzkum mezi zaměstnanci Společnosti. Průzkum byl cílen na IT oddělení (konkrétně vývojáře), jehož zaměstnanci pracují s databázemi. Jedná se o vývojáře (normální i seniorní), či technické vedoucí týmů (pozice techlead). Tito zaměstnanci jsou potenciálními budoucími uživateli vyvíjené služby. Společnost má v tomto oddělení cca 150 zaměstnanců. [21]

Dotazník, vytvořen přes platformu Microsoft Forms, byl rozeslán 24. 4. 2023. V době psaní tohoto textu bylo zasláno 7 odpovědí, se zastoupením lidí z 4 různých týmů a z pozic jak řadových, tak seniorních vývojářů. Toto nám dává i při malém počtu respondentů dobré zastoupení jak z různých rolí, které vývojář může mít, tak z různých týmů, které službu mohou různým způsobem využívat.

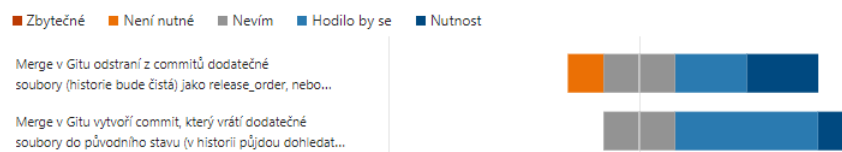
Dle první části dotazníku (viz Obrázek 14.1), 5 z těchto zaměstnanců VCS Git při vývoji využívá a ostatní by jeho využití zvážili při zlepšení funkčností/výhod, které využití VCS (spolu s vyvíjenou službou) nabízí. U 6 z dotázaných dále víme, že jejich kolegové z týmu VCS při vývoji využívají. Díky těmto odpovědím víme, že cílová skupina vývojářů je vhodná pro průzkum názorů, jelikož systém bude v budoucnu využívat po jeho nasazení do provozu.



Obrázek 14.1 Průzkum využití VSC při databázovém vývoji

Další část dotazníku zjišťovala názory dotazovaných zaměstnanců na potenciální vylepšení / vlastnosti služby, jejichž možnost implementace vyvstala v průběhu vývoje. Tyto názory následně mohou být využity v dalších iteracích vývoje služby pro prioritizaci jednotlivých součástí.

Sada otázek obsahovala vyjádření se ke dvěma různým způsobům řešení odstranění dočasných skriptů z výsledného commitu v hlavní větvi VCS. První možnost reprezentovala odstranění změn v souborech (a odstranění souborů) ze všech slučovaných commitů, aby byla historie čistá. Druhá možnost naopak reprezentovala implementované řešení neboli přidání čisticího commitu, jehož výsledkem je korektní stav větve po merge, ale se stálou možností dohledání dočasných souborů v historii. Dle výsledků (viz Obrázek 14.2) má první možnost silnější názory („nutnost“ oproti „není nutné“), zatímco druhá možnost shledává vlažnější, ale dominantně pozitivní odezvu. Volba druhé možnosti se ukázala jako správná, s možností doprogramování volitelné první možnosti pro účely některých projektů/týmů, které by tento postup využily radši.

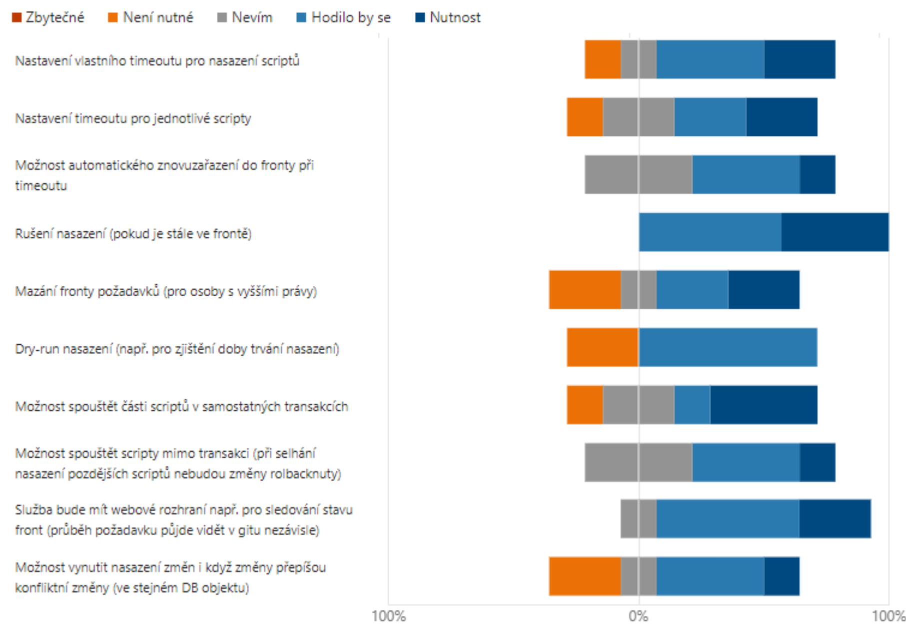


Obrázek 14.2 Průzkum preference způsobu čištění historie větve

Zbytek sady otázek byl na obecné vlastnosti, které by nový systém měl/neměl mít. Názory vzešly různorodé, ale dle výsledků (viz Obrázek 14.3) jsou hodně chtěné vlastnosti (v tomto pořadí):

- Rušení nasazení
- Webové rozhraní služby

- Možnost automatického znovuzařazení do fronty při vypršení časového limitu
- Spouštění mimo transakci



Obrázek 14.3 Průzkum názorů na potenciální funkce služby

Tyto vlastnosti budou na základě zpětné vazby prioritizovány při další iteracích vývoje služby.
Vzhledem k jednoduchosti implementace rušení nasazení bylo rozhodnuto, že tento požadavek bude zahrnut ještě jako součást této bakalářské práce.

Kapitola 15

Přidání rušení nasazení

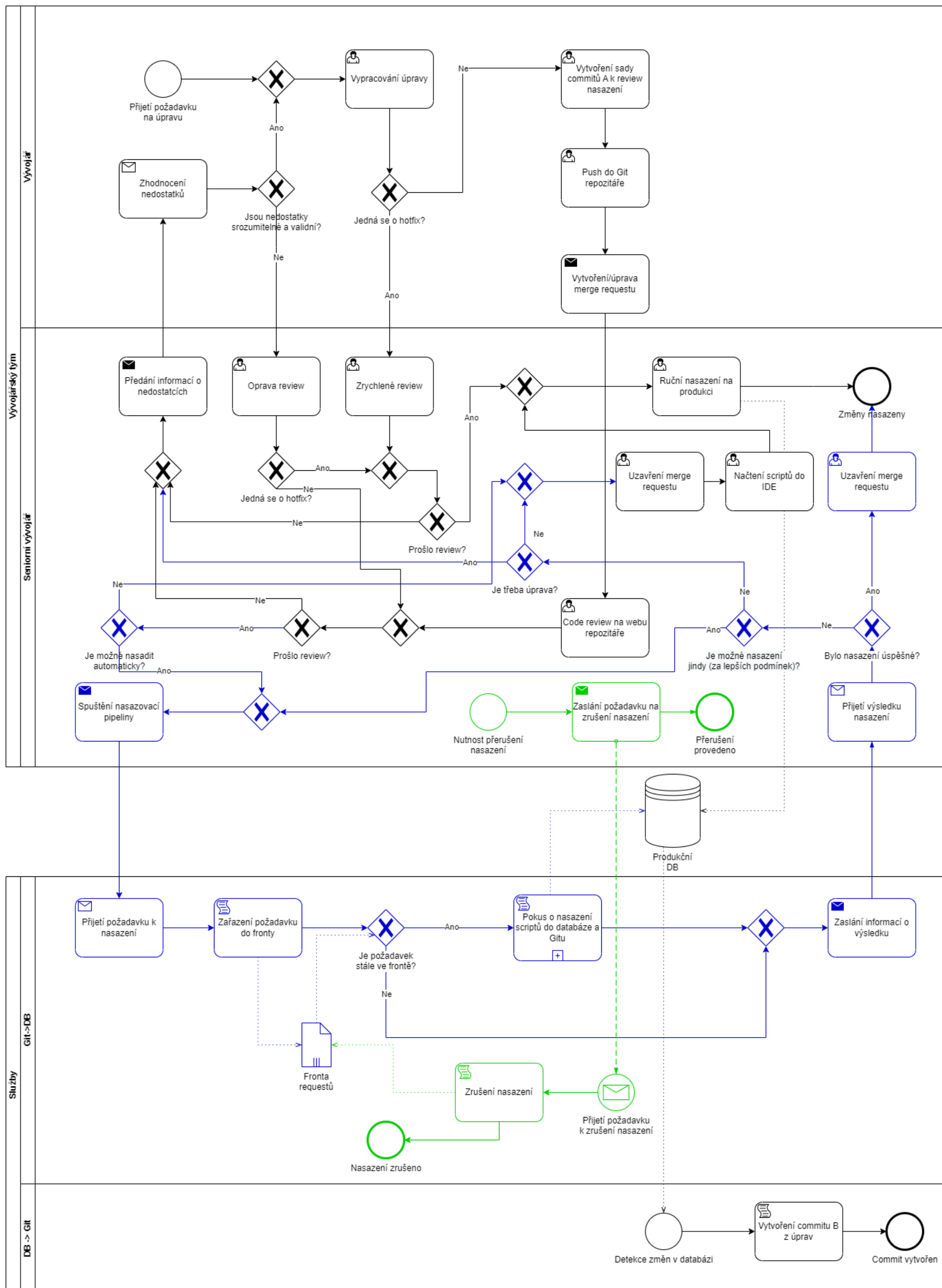
15.1 Návrh procesu rušení požadavku

Možnost přidání rušení MR není tolik složitá na implementaci díky tomu, jak jsme si navrhli naší aplikaci. Rušení MR lze rozdělit do dvou různých situací, podle stavu zpracování MR.

Pokud chce uživatel zrušit již nasazovaný MR, je to možné provést ručním zrušením transakce pomocí příkazu z vývojového prostředí. To je nejspolehlivější způsob, protože pokud by toto bylo řešeno přes zaslání požadavku na službu a rušený MR by byl zpracován dříve, než bude zpracováno zrušení, k zrušení kvůli prodlevě nedojde. Zde se budeme zaměřovat proto pouze na rušení požadavků, jejichž zpracování ještě nezačalo.

Proces bude upraven tak, že jako další akci, kterou může seniorní vývojář provést, je zaslat požadavek na zrušení MR. Toto může být z pohledu uživatele řešeno jako ruční spuštění další nastavené pipeline (s nastavitelným parametrem, jež bude větví, kterou měl rušený požadavek nasazovat), která zašle přes Rest api požadavek ke zrušení na službu, které zrušení ve frontě vykoná.

Na níže přiloženém diagramu je znázorněn proces s přidanou možností rušit MR. Přidané části pro tuto funkcionalitu jsou znázorněny zeleně.



Obrázek 15.1 Finální vývojový proces s rušením nasazení

Tento požadavek bude reprezentován novou třídou dědící z MessageableRequest – Merge-RequestCancellationRequest. Atributem tohoto požadavku bude název commitu/větve (dále zmi-

ňována pouze větev), jejíž nasazení se má zrušit.

Průběh zpracování tohoto požadavku bude probíhat jako u jiných administrativních požadavků – zpracován bude pomocí fronty obecných požadavků (třídy RequestProcessor) v prvním pracovním vlákně, v rámci čehož se zavolá metoda rušení na třídě zpracovávající frontu MR (MergeRequestConsumer).

15.2 Rušení s využitím map

Protože u fronty požadavků nelze užít náhodného přístupu k uloženým datům (u fronty lze pouze vkládat a odebírat na začátku), je třeba přidat další strukturu k zajištění korektní interní funkčnosti třídy. K tomu využijeme dvě mapy (v C# implementovány třídou Dictionary). Jednu, která eviduje, kolik je ve frontě požadavků na nasazení z dané větve, a druhou, která eviduje, kolik daných MR (opět dle větve) bylo zrušeno. Protože pracujeme s více vlákny, která budou přistupovat k daným kolekcím, přístup bude ošetřen mutexem, který zajistí, že pouze jedno vlákno může najednou číst/upravovat záznamy v daných kolekcích.

Při přidání nového požadavku do fronty MR přidáme 1 k počtu v mapě požadavků ve frontě u dané větve (s výchozí hodnotou 0). Obdobně, při zpracování nového požadavku k zrušení MR přidáme 1 v mapě zrušených požadavků u dané větve (opět s výchozí hodnotou 0).

U rušení ovšem platí, že pokud záznam v mapě zrušených požadavků měl větší (nemělo by nastat) nebo rovnou hodnotu počtu požadavků v mapě požadavků ve frontě, znamená to, že bylo zasláno zrušení MR, který není ve frontě (a pokud MR pro danou větev ve frontě je, byl již zrušen). V tomto případě požadavek neuspěl. V opačném případě byl požadavek úspěšně vykonán. Tak jako tak je uživateli zaslána informace o výsledku, jako u jiných požadavků.

Proces zpracování MR je následně upraven tak, že před spuštěním nasazovacího procesu je zkontrolováno, zda kromě nenulového počtu v mapě daných požadavků ve frontě (stav s nulovým počtem by neměl nastat) se zkontroluje, zda je nulový počet daných požadavků ke zrušení v mapě zrušených MR. Pokud je, znamená to, že nasazení nikdo nezrušil, od záznamu v mapě požadavků ve frontě je odečteno 1 (při 0 je záznam smazán) a požadavek je standardně vykonán. Pokud ovšem je počet nenulový, znamená to, že dané nasazení bylo někým zrušeno, od záznamů v obou mapách je odečteno 1 (opět, při 0 je záznam smazán), MR není nasazen a je zahlášen neúspěch.

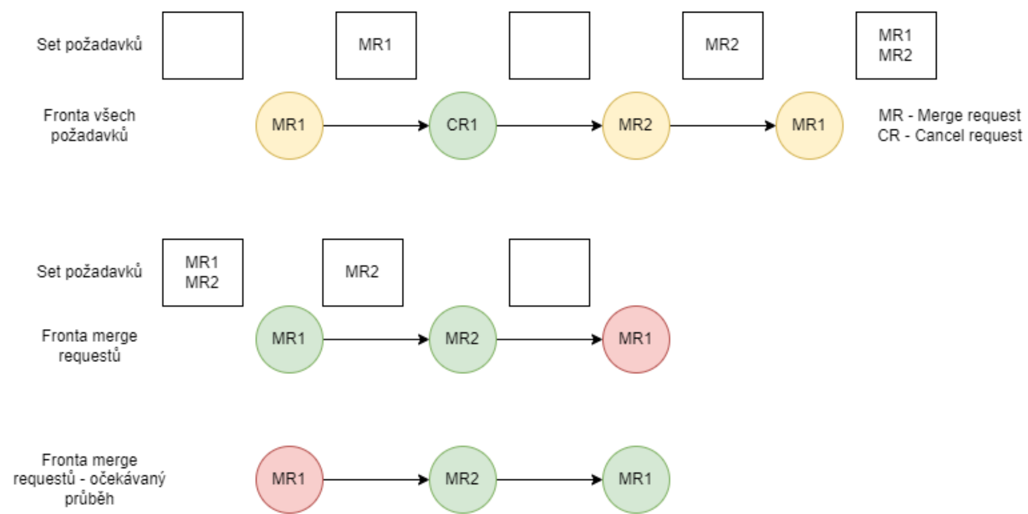
15.3 Alternativa s využitím setu

Alternativní přístup je použít pouze set (množinu) a umožnit pouze jeden aktivní (nezrušený) MR ve frontě. Přidání požadavku do fronty přidá do množiny záznam, pokud již existuje, nasazení je již zařazeno a požadavek nebude proveden. Následné zrušení tento záznam opět odstraní a tím umožní nové přidání do fronty. Tento přístup ovšem může vyústit v neočekávané chování. Níže je uveden příklad, pro který uvážíme, že než jsou všechny požadavky odeslány, žádný z požadavků na nasazení nebyl ještě zpracován RequestProcessorem.

Uživatel vydá požadavek na nasazení větve A. Poté si ovšem uvědomí, že dřív, než by měly být nasazeny změny z větve A, je třeba nasadit změny z větve B. Proto zašle zrušení požadavku na nasazení větve A (které se vykoná okamžitě), a následně po sobě požadavky na nasazení větvě B a větve A. Očekávané chování je takové, že se nejprve nasadí větev B a poté větev A.

Bohužel, protože fronta požadavků k nasazení není ovlivněná operacemi nad setem, ve frontě jsou stále požadavky v pořadí A, B, A. Při nasazování prvního MR A existuje v setu záznam A, proto je MR nasazen. Totéž je pravda pro B. U druhého A k nasazení nedojde, protože již neexistuje záznam v setu. Toto vyústí v opačné nasazení, než jsme si definovali jako očekávané chování.

Z tohoto důvodu není použití setu vhodné a je třeba užít dříve zmíněné dvě mapy.



Obrázek 15.2 Rušení požadavků s použitím setu

Závěr

Jedním z nejdůležitějších cílů v rámci teoretické a praktické části pro úspěch práce byl rozbor komplikací, které mohou v cílovém prostředí Společnosti nastat, a následný návrh jejich řešení. Obě části byly vypracovány díky osobním zkušenostem, prozkoumání existujících schopností databáze MSSQL, ale taktéž díky konzultacím s cílovými uživateli systému pracující ve Společnosti.

Tyto konzultace umožnily lépe pochopit potřeby cílového systému a byly následně využity při vývoji služby. Cíl zjistit názory uživatelské základny byl později více naplněn rozesláním a vyhodnocením dotazníku, který hlavně cílil na zjištění důležitosti různých variant směřování dalšího vývoje tohoto softwaru. Jedna z nabízených možností (rušení nasazení požadavků) byla vyvinuta jako dodatečná součást BP, což demonstrovalo flexibilitu softwaru pro budoucí vylepšení procesu.

Dále byl splněn cíl rozboru možných existující řešení, v rámci kterého byly vybrány zástupci různých systémů, které řeší podobnou problematiku jako tato BP. Byly zmíněny výhody a nevýhody těchto systémů a za pomoci dříve získaných informací bylo zhodnoceno, zda jsou dané systémy vhodné k použití v cílovém prostředí. Protože žádné z řešení požadavkům nevyhovovalo, zvolenou variantou byla tvorba vlastního řešení na míru definovaným požadavkům. Hlavními faktory, které ve zvolení vlastního řešení vyústily, byly možnost použití s používaným softwarem ve Společnosti (například konkrétní typ a vlastnosti databáze nebo množství softwaru, který databázi využívá) a nutnost spolehlivosti funkčnosti řešení při zachování možnosti zvolený software pro nasazování změn vždy nepoužívat.

V rámci praktické části došlo k již dříve zmíněnému splnění cíle navrhnout řešení všech definovaných komplikací. Soubor navrhnutých řešení jednotlivých komplikací následně umožnil splnění cíle navrhnutí celého systému, kterým je vlastní řešení, které do jednoho velkého procesu zakomponovalo navrhnutá řešení komplikací. Tento proces byl detailně popsán a znázorněn s pomocí diagramů, který proces znázornily z několika různých úhlů pohledu.

Došlo taktéž k vyvinutí navrhnuté aplikace ve zjednodušené formě PoC, která funguje jako interaktivní aplikace v terminálu, ale je plně připravena na převod do formy služby v podobě webového serveru. Pro vývoj aplikace byl zvolen jazyk C# kvůli jednoduché dostupnosti vhodných knihoven pro práci s užívaným softwarem (například databáze) a konvencím, které jsou v cílovém prostředí používány.

Posledním dílčím cílem bylo tuto aplikaci vhodně otestovat, což bylo provedeno vytvořením sady unit testů a definicí řady systémových end to end testovacích scénářů. Všechny tyto testy byly na aplikaci provedeny a tím byla ověřena její funkčnost.

Vzhledem ke splnění všech dílčích cílů byl splněn hlavní cíl této bakalářské práce návrhu a vytvoření celkového systému pro synchronizaci databázových změn pro použití vývojářskými týmy ve velké firmě. Nebyl vytvořen plně funkční systém, ale pouhé PoC, což ovšem bylo definováno jako rozsah vývoje pro použití v rámci této bakalářské práce.

Následný vývoj bude pokračovat zejména úpravou softwaru do podoby, která půjde použit v produkčním prostředí Společnosti, což bude úprava do podoby webového serveru a napojení na interní software, který Společnost využívá. V dalších iteracích může následně dojít k implementaci různých dodatečných funkcionalit, jejichž priorita byla zjištěna v rámci rozeslaného dotazníku.

Přínos této práce je zejména ve snaze vymyslet jiné, alternativního řešení, jak databázové změny evidovat a nasazovat, a jaké možné benefity toto řešení může přinést.

Bibliografie

1. *Azure SQL Documentation*. What is Azure SQL Database? [online]. Microsoft, 2023-03-03. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/azure-sql/database/sql-database-paas-overview?view=azuresql.
2. CLARK, Daniel. *Dataquest*. SQL vs T-SQL: Understanding the Differences [online]. 2021-03-04. [cit. 2023-05-18]. Dostupné z: https://www.dataquest.io/blog/sql-vs-t-sql/.
3. HUGHES, Adam. *Techtarget*. T-SQL (Transact-SQL) [online]. 2019-09. [cit. 2023-05-18]. Dostupné z: https://www.techtarget.com/searchdatamanagement/definition/T-SQL.
4. *SQL Server*. Stored Procedures (Database Engine) [online]. Microsoft, 2023-04-03. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view=sql-server-ver16.
5. *SQL Server*. DDL Triggers [online]. Microsoft, 2023-03-01. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/sql/relational-databases/triggers/ddl-triggers?view=sql-server-ver16.
6. *SQL Server*. DML Triggers [online]. Microsoft, 2023-03-01. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/sql/relational-databases/triggers/dml-triggers?view=sql-server-ver16.
7. *SQL Server*. User-defined functions [online]. Microsoft, 2022-11-19. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/sql/relational-databases/user-defined-functions/user-defined-functions?view=sql-server-ver16.
8. SCOTT CHACON, Ben Straub. *Pro Git*. Úvod - Správa verzí [online]. 2. vyd. 2017-01-22. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/book/cs/v2/%5C%C3%5C%9Avod-Spr%5C%C3%5C%A1va-verz%5C%C3%5C%AD.
9. SCOTT CHACON, Ben Straub. *Pro Git*. Úvod - Stručná historie systému Git [online]. 2. vyd. 2017-01-22. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/book/cs/v2/%5C%C3%5C%9Avod-Stru%5C%C4%5C%8Dn%5C%C3%5C%A1-historie-syst%5C%C3%5C%A9mu-Git.
10. SCOTT CHACON, Ben Straub. *Pro Git*. Úvod - Základy systému Git [online]. 2. vyd. 2017-01-22. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/book/cs/v2/%5C%C3%5C%9Avod-Z%5C%C3%5C%A1klady-syst%5C%C3%5C%A9mu-Git.
11. SCOTT CHACON, Ben Straub. *Pro Git*. Větve v systému Git - Větve v kostce [online]. 2. vyd. 2017-01-22. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/book/cs/v2/V%5C%C4%5C%9Btve-v-syst%5C%C3%5C%A9mu-Git-V%5C%C4%5C%9Btve-v-kostce.

12. SCOTT CHACON, Ben Straub. *Pro Git*. Větve v systému Git - Základy větvení a slučování [online]. 2. vyd. 2017-01-22. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/book/cs/v2/V%5C%C4%5C%9Btve-v-syst%5C%C3%5C%A9mu-Git-Z%5C%C3%5C%A1klady-v%5C%C4%5C%9Btven%5C%C3%5C%AD-a-slu%5C%C4%5C%8Dov%5C%C3%5C%A1n%5C%C3%5C%AD.
13. SCOTT CHACON, Ben Straub. *Pro Git*. Větve v systému Git - Přeskládání [online]. 2. vyd. 2017-01-22. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/book/cs/v2/V%5C%C4%5C%9Btve-v-syst%5C%C3%5C%A9mu-Git-P%5C%C5%5C%99eskl%5C%C3%5C%A1d%5C%C3%5C%A1n%5C%C3%5C%AD.
14. SCOTT CHACON, Ben Straub. *Pro Git*. Základy práce se systémem Git - Práce se vzdálenými repozitáři [online]. 2. vyd. 2017-01-22. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/book/cs/v2/Z%5C%C3%5C%A1klady-pr%5C%C3%5C%A1ce-se-syst%5C%C3%5C%A9mem-Git-Pr%5C%C3%5C%A1ce-se-vzd%5C%C3%5C%A1len%5C%C3%5C%BDmi-repozit%5C%C3%5C%A1%5C%C5%5C%99i.
15. SCOTT CHACON, Ben Straub. *Pro Git*. Větve v systému Git - Vzdálené větve [online]. 2. vyd. 2017-01-22. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/book/cs/v2/V%5C%C4%5C%9Btve-v-syst%5C%C3%5C%A9mu-Git-Vzd%5C%C3%5C%A1len%5C%C3%5C%A9-v%5C%C4%5C%9Btve.
16. *Azure DevOps*. What is Azure DevOps? [online]. Microsoft, 2022-10-10. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/devops/user-guide/what-is-azure-devops?view=azure-devops.
17. *Azure Repos Documentation*. What is Azure DevOps? [online]. Microsoft, 2022-11-15. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/devops/repos/get-started/what-is-repos?view=azure-devops.
18. *Azure Pipelines*. What is Azure Pipelines? [online]. Microsoft, 2022-04-12. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops.
19. *Azure Pipelines*. Specify events that trigger pipelines [online]. Microsoft, 2023-02-10. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/devops/pipelines/build/triggers?view=azure-devops.
20. *Azure Pipelines*. Runtime parameters [online]. Microsoft, 2022-02-24. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/devops/pipelines/process/runtime-parameters?view=azure-devops&tabs=script.
21. *Interní systém Společnosti* [interní systém]. Společnost, [b.r.]. [cit. 2023-05-07].
22. N., Zdeněk. *Interní příručka Společnosti pro vývoj a nasazování databázových změn* [interní systém]. Společnost, 2022-11-29. [cit. 2023-05-07].
23. Č., David. *Nová synchronizační služba* [interní software]. Společnost, [b.r.].
24. *SQL Server*. Deadlocks guide [online]. Microsoft, 2023-03-21. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-deadlocks-guide?view=sql-server-ver16.
25. D., Jiří. *Konzultace na téma funkcí vyvíjené služby* [elektronická korespondence]. 2023-03-29.
26. *Flyway Enterprise* [soft.]. Redgate, [b.r.]. [cit. 2023-05-18]. Dostupné z: https://www.red-gate.com/products/flyway/enterprise/.
27. *Flyway Documentation*. Welcome To Flyway [online]. Redgate, 2023-05-15. [cit. 2023-05-18]. Dostupné z: https://documentation.red-gate.com/fd/welcome-to-flyway-184127914.html.
28. *Flyway Documentation*. Migrations [online]. Redgate, 2023-05-15. [cit. 2023-05-18]. Dostupné z: https://documentation.red-gate.com/fd/migrations-184127470.html.

29. *Flyway Documentation.* Schema model [online]. Redgate, 2022-11-17. [cit. 2023-05-18]. Dostupné z: https://documentation.red-gate.com/fd/schema-model-138347045.html.
30. *Flyway Documentation.* Drift Detection [online]. Redgate, 2022-11-14. [cit. 2023-05-18]. Dostupné z: https://documentation.red-gate.com/fd/drift-detection-149127470.html.
31. V., Dalibor. *Konzultace na téma užívání užívání Flyway ve Společnosti* [elektronická korespondence]. 2023-05-05.
32. *Entity Framework.* Entity Framework Core [online]. Microsoft, 2021-05-25. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/.
33. *Entity Framework.* Migrations Overview [online]. Microsoft, 2023-01-12. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli.
34. JINDRA, Petr. *Verzování databází* [online]. Brno, 2013 [cit. 2023-05-18]. Dostupné z: http://hdl.handle.net/11012/27242. Diplomová práce. Vysoké učení technické v Brně. Fakulta strojního inženýrství. Ústav automatizace a informatiky.
35. *Azure Pipelines.* Azure Pipelines agents [online]. Microsoft, 2023-05-10. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/devops/pipelines/agents/agents?view=azure-devops&tabs=browser.
36. *Azure Pipelines.* InvokeRESTAPI@1 - Invoke REST API v1 task [online]. Microsoft, 2023-05-02. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/devops/pipelines/tasks/reference/invoke-rest-api-v1?view=azure-pipelines&tabs=yaml#inputs.
37. *Git documentation.* git merge-base [online]. 2022-12-12. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/docs/git-merge-base.
38. *Git documentation.* git checkout [online]. 2023-04-17. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/docs/git-checkout/.
39. SCOTT CHACON, Ben Straub. *Pro Git.* Git Tools - Rewriting History [online]. 2. vyd. 2023-04-10. [cit. 2023-05-18]. Dostupné z: https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History.
40. Č., David. *Konzultace na téma využití synchronizační služby* [elektronická korespondence]. 2023-04-14.
41. DIMITRIJEVIC, Nikola. *SQL Shack.* All about locking in SQL Server [online]. 2017-06-16. [cit. 2023-05-18]. Dostupné z: https://www.sqlshack.com/locking-sql-server/.
42. *SQL Server.* SET TRANSACTION ISOLATION LEVEL (Transact-SQL) [online]. Microsoft, 2023-03-21. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver16.
43. *Azure Pipelines.* Manage service connections [online]. Microsoft, 2022-11-28. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/azure/devops/pipelines/library/service-endpoints?view=azure-devops&tabs=yaml#generic-service-connection.
44. *azure-pipelines-extensions.* HttpRequestSampleWithoutHandler [soft.]. Microsoft, 2022. [cit. 2023-05-18]. Dostupné z: https://github.com/Microsoft/azure-pipelines-extensions/tree/master/ServerTaskHelper/HttpRequestSampleWithoutHandler.
45. *.NET fundamentals documentation.* What is .NET? Introduction and overview [online]. Microsoft, 2023-03-15. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/core/introduction.

46. *C# documentation*. A tour of the C# language [online]. Microsoft, 2023-05-04. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp.
47. *Advanced .NET programming*. How to: Implement a producer-consumer dataflow pattern [online]. Microsoft, 2021-09-15. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-implement-a-producer-consumer-dataflow-pattern.
48. *LibGit2Sharp Hitchhiker's Guide to Git* [online]. 2016-05-24. [cit. 2023-05-18]. Dostupné z: https://github.com/libgit2/libgit2sharp/wiki/LibGit2Sharp-Hitchhiker's-Guide-to-Git.
49. *SQL Server*. Overview (SMO) [online]. Microsoft, 2023-04-03. [cit. 2023-05-18]. Dostupné z: https://learn.microsoft.com/en-us/sql/relational-databases/server-management-objects-smo/overview-smo?view=sql-server-ver16.
50. *Software Testing Help*. Types Of Software Testing: Different Testing Types With Details [online]. 2023-04-25. [cit. 2023-05-18]. Dostupné z: https://www.softwaretestinghelp.com/types-of-software-testing/.

Obsah přiloženého média

```
readme.txt ........................ popis obsahu média
├─ src ............................ solution se zdrojovými kódy implementace a testů
├─ diagrams ....................... draw.io diagramy, které byly použity k vytvoření obrázků diagramů
├─ databasescripts ................ složka s databázovými scripty, které je možné použít pro testování
└─ thesis
   ├─ src ......................... zdrojová forma práce ve formátu XeLaTeX včetně všech použitých souborů
   │  └─ graphic .................. obrázky použité v práci
   └─ text ........................ text práce
      └─ thesis.pdf ............... text práce ve formátu PDF
```