FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Nasazovací strategie aplikací a jejich vliv na migrování relačních databází
Student:	Jan Kotrlík
Vedoucí:	Ing. Martin Komárek
Studijní program:	Informatika
Obor / specializace:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Popište běžné typy nasazovacích strategií aplikací, analyzujte jejich výhody a nevýhody a jejich užití v praxi.
Popište principy migrování relačních databází.
Identifikujte potenciální problémy a rizika spojená s migrací relačních databází v rámci běžných nasazovacích strategií.
Pro alespoň jednu vybranou nasazovací strategii navhrněte scénař pro bezpečnou migraci databází, včetně scénářů pro zotavení.
Demonstrujte a otestujte své závěry v jednoduché aplikaci v rámci minimalistického Kubernetes systému.

[1] LUKSA, Marko. Kubernetes in Action. Manning, 2018. ISBN 9781617293726.
[2] MORRIS, John. Practical Data Migration. British Computer Society, 2006. ISBN 9781902505718.

Elektronicky schválil/a prof. Ing. Pavel Tvrdík, CSc. dne 22. září 2023 v Praze.

Bakalářská práce

NASAZOVACÍ STRATEGIE APLIKACÍ A JEJICH VLIV NA MIGROVÁNÍ RELAČNÍCH DATABÁZÍ

Jan Kotrlík

Fakulta informačních technologií
Katedra počítačových systémů
Vedoucí: Ing. Martin Komárek
15. února 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Rád bych zde poděkoval Ing. Martinu Komárkovi za vedení mé bakalářské práce, za cenné rady a přípomínky, které mi poskytl. Rád bych zde také poděkoval své rodině, která mne při studiu podporovlala.

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 15. února 2024

Abstrakt

Cílem této bakalářská práce je analýza nasazovacích strategií aplikací a jejich vliv na proces migrování relačních databází. Klade si za cíl poskytnout detailní přehled běžných typů nasazovacích strategií, včetně analýzy jejich výhod a nevýhod a jejich praktického užití. Práce dále zkoumá principy migrování relačních databází a identifikuje potenciální problémy a rizika, která mohou vzniknout v rámci těchto procesů. Na základě teoretických poznatků navrhuje scénář pro bezpečnou migraci databází, včetně návrhu na zotavení po selhání. Teoretická část je doplněna praktickým příkladem na jednoduchém nasazení jedné nasazovací strategie, pro ověření navrhovaných řešení v praxi. Práce nabízí komplexní pohled na výzvy a řešení spojené s nasazováním aplikací při potřebě migrovat relačních databáze.

Klíčová slova nasazovací strategie, migrování relačních databází, kubernetes, A/B testing, Rolling updates, SQL, Node.js, zálohování a obnova databází

Abstract

The aim of this bachelor thesis is to analyze application deployment strategies and their impact on the process of migrating relational databases. It aims to provide a detailed overview of common types of deployment strategies, including an analysis of their advantages and disadvantages and their practical use. The thesis also examines the principles of relational database migration and identifies potential problems and risks that may arise in these processes. Based on the theoretical findings, it proposes a scenario for safe database migration, including a proposal for failure recovery. The theoretical part is complemented by a practical example on a simple deployment strategy to validate the proposed solutions in practice. The work offers a comprehensive view of the challenges and solutions associated with deploying applications with relational databases migrations.

Keywords deployment strategies, relational database migration, kubernetes, A/B testing, Rolling updates, SQL, Node.js, database backup and restoration

Seznam zkratek

API	Application Programming Interface
GUI	Graphical User Interface
HA	High Availability
HTTP	Hypertext Transfer Protocol
IaaC	Infrastructure as a Code
JSON	JavaScript Object Notation
ORM	Object-Relational Mapping
POC	Proof Of Concept
PR	Pull Request
RDBMS	Relational Database Management System
SQL	Structured Query Language
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Úvod

V současném dynamickém světě informačních technologií hrají aplikace a databáze klíčovou roli ve fungování a úspěchu organizací. Nasazování aplikací a správa datových úložišť jsou zásadní procesy, které vyžadují pečlivé plánování a strategii. Tato bakalářská práce se zaměřuje na nasazovací strategie aplikací a jejich vliv na migrování relačních databází, což je zásadní aspekt pro zajištění plynulého a efektivního přenosu dat mezi verzemi.

V první částí bude čtenář seznámen s běžně používanými nasazovacími strategiemi, jejich výhodami, nevýhodami a příklady jejich použití.

V druhé a třetí části pak bude diskutována problematika migrování relačních databází. Dále pak bude ke každé strategii navržena vhodná metodika. V poslední části bude demonstrována konkrétní strategie včetně scénáře pro zotavení.

Identifikace potenciálních problémů a rizik spojených s migrací relačních databází v rámci běžných nasazovacích strategií je dalším klíčovým bodem této práce. Mezi hlavní výzvy patří zachování konzistence dat a minimalizace výpadku služeb.

Pro vybrané nasazovací strategie práce navrhuje scénáře pro bezpečnou migraci databází, včetně scénářů pro zotavení v případě selhání. Tyto scénáře jsou zaměřeny na vytvoření robustního a odolného procesu migrace, který minimalizuje rizika a zajišťuje plynulý přechod pro všechny zúčastněné systémy.

V závěrečné části práce jsou teoretické poznatky demonstrovány a otestovány na jednoduché aplikaci v rámci minimalistického Kubernetes systému. Tento praktický přístup umožňuje ověřit efektivitu navrhovaných strategií a metod migrace v reálném prostředí a poskytuje užitečný vhled do praktických aspektů nasazování aplikací a migrace databází.

Cílem této bakalářské práce je provést rešerši běžně používaných nasazovacích strategií a principů, zhodnotit výhody a nevýhody jednotlivých přístupů a nastínit praktické příklady jejich užití v praxi. Dále pak analyzovat problematiku principů migrování relačních databazí a navrhnout postup pro bezpečnou migraci a případné zotavení. Následně je cílem vypracovat detailní plán nasazení a zotavení pro jednu vybranou nasazovací strategii a demonstrovat tyto přípravy na minimalistické aplikaci.

Výsledek této práce má představovat komplexní pohled na problematiku nasazovacích strategií aplikací a migrace relačních databází, přičemž klade důraz na praktické aspekty a výzvy spojené s těmito klíčovými procesy v oblasti informačních technologií.

Tuto práci jsem si vybral, protože se velmi úzce dotýká mého profesního zaměření a její problematiku jsem považoval za nedostatečně analyzovanou a řešenou.

Kapitola 1

Nasazovací strategie aplikací

1.1 Co jsou to nasazovací strategie

Každou aplikaci čeká po fázi vývoje a testování fáze nasazení. Kromě pilotních projektů a POC (Proof Of Concept) lze očekávat, že se aplikace bude dále vyvíjet a testovat a znovu a znovu nasazovat. **Nasazovací strategie je proces či metodika pro vydávání nových verzí softwaru**.[1] Může obsahovat i scénáře pro zotavení a obnovu předchozího stavu. Nemusí se týkat pouze produkčních prostředí - některé strategie jsou součástí celého vývojového cyklu aplikace.

Tyto metodiky se nezabývají pouze aplikacemi běžícími v cloudu, ale i desktopovými či mobilními aplikacemi, které také vyžadují nové verze, ať už kvůli nutnosti podporovat nový hardware, přidat funkcionalitu nebo opravit chybu. Tato práce se zaměří především na cloudové aplikace.

Všechny zde diskutované strategie nemají běžně užívané ekvivalenty v českém jazyce. Tento fakt práce respektuje a nepřekládá jejich názvy, neboť by byly poněkud krkolomné a mohly by mystifikovat čtenáře. Jejich anglické názevy jsou ve většině případů dostatečně výstižné a absense překladu tak nebude problematická.

1.2 Nedostupnost služby

Typickým důvodem adaptace nasazovací strategie je co největší snížení rizika nedostuponosti služby, které by negativně ovlivnila uživatelskou základnu, ať už je jakákoliv. V případě systémů týkajících se zdraví a života (záchranná služba, nemocnice, ...) pak může být nedostupnost fatální. I pro tyto případy lze ovšem dopustit omezení služby, pokud je řádně připraveno a včas komunikováno. Aplikace George od společnosit Česká Spořitelna kupříkladu prochází pravidelnými aktualizacemi a po jejich čas je aplikace nedostupná (obvykle v nočních hodinách z pátku na sobotu). Společnost Blizzard pak každý týden odstavuje na několik hodin všechny své služby, aby provedla aktualizaci a správu.[2, 3]

1.3 Strategie Recreate

Nasazení strategií Recreate probíhá tak, že se původní verze vypne, odstraní či deaktivuje a místo ní se nasadí nová[4]. Jedná se o nejjednodušší a nejvíce přímočarou strategii. Je i nejjednodušší na implementaci, pokud vývojář nebo správce systému chce nebo musí použít vlastní řešení. Je to základní stavební kámen nasazovacích strategií a většina pokročilejších strategií ji do jisté míry používá.

Při nasazení za použití strategie Recreate se nejprve současně běžící verze úplně zastaví, a až poté se spustí nová. Pokud je tento proces atomický a při nasazení nedojde k žádným problémům, lze strategii použít bez výpadku služby. Obvykle toho ale nelze dosáhnout - aplikace typicky vyžadují jak čas na ukončení, tak čas na opětovný start. Výpadku službyse tak nelze vyhnout. Výhodou tohoto principu je naprosté odstranění zdrojů a nová inicializace nové verze.

Její riziko tkví v zotavení. Součástí strategie jako takové není přechod na předchozí stav. Pokud tedy nasazení selže, stává se služba nedostupnou a je nutný zásah operátora, ať už přechodem zpět na funkční verzi, nebo opravou problému a vydáním nové verze.

Užití této metodiky se obvykle nachází již v samotném vývojovém procesu aplikací, a to především v prvních fázích projektu nebo při vytváření a testování různých architektur a technologií. V takových případech jsou změny často silně nekompatibilní a přechod z jedné verzne na druhou by pouze brzdil vývoj. Pak je nanejvýš vhodné všechny použité zdroje odstranit a znovu vytvořit pro nově nasazovanou verzi. To se může týkat nejen kódu samotného, ale i infrastruktury nebo služeb, které aplikace využívá (databáze, fronty, lambda funkce ...). I při použití nástrojů IaaS (Infrastructure as a Code), které dokáží aplikaci nahradit bez restartu[5] je v některých případech nutné současné nasazení úplně odstranit a znovu vybudovat.

Recreate je typickou strategií pro aktualizaci desktopových aplikací a operačních systémů jako takových. Využívá ji například nejpoužívanější[6] webový prohlížeč Google Chrome[7], nebo nejpopulárěnšjí[8] operační systémy Windows a macOS pro důležité aktualizace.

Recreate je také jediná strategie, kterou lze využít při nasazování nových verzí mobilních aplikací dvou nejpoužívanějších platforem na současném trhu[9] - App Store od společnosti Apple[10] a Google Play od společnosti Google[11]. U standardního nasazení skrz aplikační obchod se nejedná o atomickou operaci - aplikace se musí restartovat a aktualizace se tak neděje napozadí. Lze sice použít koncept OTA (Over The Air) aktualizace, který změnu před uživatelem fakticky skryje[5], ale i ten má svá omezení. Obecně tedy záleží na tom, jak invazivní a nepříjemný tento proces může být pro uživatele a dle toho je třeba k strategii přistupovat.

V kontextu cloudových deploymentů strategie nerozlišuje, zda běží jako jedna jediná instance na jedom serveru, nebo zda existuje více instancí, na jednom či na více serverech. Při nasazení se zastaví všechny instance na všech serverech a poté jsou opět puštěny v nových verzích.

1.4 Strategie Rolling updates, strategie Rolling deployment, strategie Ramped

Strategie známá také jako Ramped nebo Rolling deployment je velmi podobná strategii Recreate. Také dochází k nahrazení současné verze verzí novou. Zásadní rozdíl je ovšem v tom, že **v průběhu nasazení běží obě verze současně**. Nová verze pak musí splnit jedno nebo více kritérií (*"health-checks"*), které jsou monitorovány od okamžiku spuštění aplikačního procesu nové verze, aby byla systémem označena za připravenou. V tu chvíli dojde k přepnutí provozu z předchozí verze na verzi novou. Pokud nová verze kritéria nesplní, je její nasazování zastaveno a předchozí verze pokračuje v odbavování požadavků. Pokud je nasazení nové verze úspěšné, zahájí se proces odstranění předchozí, již nepoužívané verze. Jedná se tedy o **inkrementální** proces nasezení, proto se lze setkat i s pojmem *"incremental updates"*.

Dalším z principů strategie Rolling updates je dávkovaná distribuce aktualizací. Nové instance se nespouštějí všechny naráz, ale postupně. Velikost celku je obvykle jedním z konfiguračnch parametrů této strategie. Je tak vždy zachována určitá část infrastruktury v provozuschopném stavu, což zajišťuje kontinuitu služby pro uživatele.

Termíny Rolling updates, Rolling deployments a Ramped jsou v princupu synonyma. Ramped se vyskytuje méně, zřejmě proto, že zbylé dva pojmy výstižnější popisují nasazovací proces.

Nejkonzervativnější přístup velí nasazovat a odstraňovat jednu instanci po druhé. Celé nasazení je prohlášeno za chybové, pokud pokud není jedna dílčí aktualizace provedena úspěšně a celý systém se stejnou metodou vrací na předchozí verzi. Dává to operátorům čas a prostor pro

odhalení problémů a chyb, které nepokryla kritéria úspěšného nasazení jednotlivých instancí.

Nasazování serverů po jednom funguje velmi dobře pro menší nasazení. Je ovšem zdlouhavé a nepraktické pro aplikace běžící ve větších flotilách, a to ať už v jednom datacentru, nebo v různých geograficky oddělených provozech. Zároveň je obvykle nebezpečné nastavit parametr na dávku na 100 % - operátor se tím připravuje o prostor otestovat novou verzi a případně zastavit nasazení manuálně. V praxi se uplatňují dva přístupy k velikost dávky. Buď se celá flotila rozdělí na skupiny o stejném počtu serverů a ty se nasazují postupně jedna po druhé, nebo se dávka zvětšuje s úspěšným nasazením předchozích celků, například 10 %, 40 % a 50 %.

Kritérií pro úspěšné nasazení může být celá řada. Tím nejjednodušším je skutečnost, že proces aplikace po definovaný čas od startu nehavaroval. Velmi populární je kontrola odpovědi na HTTP dotaz, pokud aplikace vystavuje webové API (Application Programming Interface). Robustnější kontroly pak mohou například provést i *"smoke test"*, obsahující zevrubnější testování funkčnosti nové verze.

Tato medotika má nespornou výhodu v minimalizaci omezení služby. Při úspěšném i neúspěšném nasazaení uživatel nepocítí výpadek, což je často žádaným aspektem nasazovací strategie. Ani ona ovšem neochrání aplikaci před chybami, které se projeví až při jejím skutečném používání, jako jsou například nekompatibilita dat v databázi s novou verzí kódu, nebo chyba v optimalizaci algoritmu v nové verzi. Naopak zdánlivě úspěšný provoz nové verze může mít fatální následky pro celý systém. I s touto robustnější nasazovací variantou je tedy třeba mít připravené procesy pro zotavení.[12][13][14]

Rolling updates je první z analyzovaných strategií, při jejíž aplikaci běží dvě verze zároveň. To klade požadavek na kompatibilitu verzí se službami, s nimiž je aplikace integrována. Mohou to být API třetích stran, nebo třeba také databáze, využívání front a celá řada dalších. Je to tedy první princip velmi citlivý na nekompatibilní změny (*"breaking changes"*. Strategie Recreate je v tomto ohledu o něco přívětivější - jedna verze nahrazuje druhou, aniž by musely pracovat se zdroji zároveň. Klady a zápory obou přístupů tak musí zhodnotit provozovatelé a autoři aplikací. Práce se v dalších částech zaměřuje pouze na rizika spojená s kompatibilitou s relační databází.

Tento princip uplatňují cloudoví provozovatelé spravující orchestrační služby pro jednotlivé vývojáře i pro celé organizace, které si nezřizují vlastní řešení. Jsou to například AWS ElasticBeanstalks[15], AWS Elasctic Container Services[16], Google Cloud Apps[17] nebo Heroku[18].

1.5 Strategie Blue/Green

Myšlenka současně běžících verzí lze rozvinout za hranici potřeby rychlých, či dokonce atomických vydávání nových verzí. Pokud je již stanoven požadavek na zpětnou kompatibilitu minimálně současné a nové verze, může být výhodné nechat obě verze spuštěné vedle sebe a cíleně obě využívat zároveň.

Strategie Blue/Green tuto ideu implementuje a zavadí terminologii pro současnou verzi (*"Blue"*) a verzi novou (*"Green"*). Nová verze se nasadí vedle druhé a obvykle proti ní proběhne zevrubné testování. Pokud při něm nejsou nalezeny blokující problémy, správce nasazení manuálně **přepne provoz z prostředí Blue na prostředí Green**. To je faktický okamžik samotného nasazení nové verze. Výhoda této metodiky spočívá v tom, že prostředí Blue není ihned po přepnutí recyklováno, ale zůstává aktivní jako záloha. To umožňuje správcům prostředí velmi rychle přepnout na předchozí verzi, pokud by se v novém nasazení vyskytly potíže. Tento stav nejenže zvyšuje spolehlivost a dostupnost aplikací, ale také umožňuje týmům provádět bezpečnější aktualizace a testování v produkčním prostředí. Tímto způsobem se vývojáři mohou vyhnout mnoha běžným problémům spojeným s nasazováním nových verzí softwaru, jako jsou nepředvídané chyby a dlouhé doby výpadku.

Ačkoliv požadavek na kompatibilitu stanovuje již předchozí strategie, u Blue/Green a následujících strategií musí být dodržen striktně a plně. Zatímco v procesu Rolling updates lze postupovat tak, aby byl dopad nekompatibility omezen pouze na dobu samotného nasazení a lze jej do jisté míry tolerovat, v případě dvou dlouhodobě běžících prostředích vedle sebe je nekompatibilita zcela vyloučena. Její existence by negovala veškeré výhody, které Blue/Green představuje.

Nevýhodou Blue/Green může být náročnost na prostředky - zatímco u inkrementálních aktualizací může dojít k přetížení zdrojů pouze v době samotného nasazení (pomineme-li náročnost aplikace samotné), Blue/Green obvykle vyžaduje dvě samostatná a plně nasazená prostředí. Je také přirozeně náročnější na údržbu a implementaci.

Prostředí Blue se recykluje buď ve chvíli, kdy je Green prostředí prohlášeno za stabilní a plně funkční, nebo ve chvíli, kdy se vydává nová verze. Pokud není problematický fakt, že je vyžadován dvojnásobek zdrojů, užívá se v praxi spíše druhé varianty, neboť orchestrace dvou produkčních prostředí může být náročná a v důsledku kontraprodiktivní. Recyklace se ovšem řídí více faktory, než pouze náročností na prostředky, například dělkou vývojového cyklu. Pokud se nová verze aplikace vydává dvakrát měsíčně, lze argumentovat, že existence obou prostředí je praktičtější a bezpečnější po celou dobu vývoje aplikace.[19, 20, 21]

1.6 Strategie A/B Testing, Canary

Náročnost provozu dvou prostředí zároveň lze elegantně spojit s úmyslem testovat nové prostředí v produkci bez rizika problémů s celým nasazením konkrétní verze. Zároveň se ukazuje vhodné podrobit novou verzi zevrubnému testování větším publikem, než je interní tým testerů, ale zároveň menšímu, než jsou všichni uživatelé konkrétní aplikace, jejichž počet může být od jednotek po stovky milionů.

To je hlavní myšlenka A/B testingu. Nová verze se pustí vedle současné, ale v **omezeném měřítku nasazení a pro omezenou uživatelskou základnu**. Užitím tohoto principu se ovšem necílí pouze na kontrolu stability a funkčnosti. Její uplatnění tkví hlavně ve sledování **reakcí uživatelské základny na změny**, které přináší nová verze.

Například nové umístění tlačítka pro přidání do košíku může být matoucí a uživatelé tak přestanou v aplikaci nakupovat. Algoritmus pro vyhledání produktu může obsahovat chybu a vrátí pouze první tři nalezené produkty a frustrovaní zákaznící odcházejí ke konkurenci. Příkladů, kdy novinka v aplikaci vede k potížím, lze zřejmě ilustrovat mnoho. Pomocí vhodně užité strategie A/B testing lze tyto tendence mitigovat a předejít tak problémům, které ve chvíli, kdy nastanou, mohou být velmi těžce řešitelné a v krajím případě neřešitelné.

Stejně jako Blue/Green by správně implementovaná A/B Testing měla umožnit rychlý přesun z jedné verze na druhou, a to oběma směry. To vyžaduje ještě náročnější orchestraci, zvlášť pokud je odstraněn "luxus"dvou plně běžících prostředí. Oba principy lze nicméně úspěšně kombinovat. A/B Testing je naopak výrazně náročnější na implementaci a údržbu, pokud je nutné škálovat prostředí proti sobě. Tento typ aplikace A/B Testingu se nazývá **Canary strategie**.

Distribuce uživatelů mezi prostředími A a B by v ideálním případě měla být dynamicky pozměnitelná a je plně v moci správců nasazení. Obvykle se u nového prostředí začíná s jednotkami procent uživatelské základny a po vyhodnocení se buď sníží na nulu, nebo rozšíří na vyšší procento, v některých případech až na celou uživatelskou základnu. Rozdělení uživatelů mezu prostředí může být buď zcela náhodné, nebo lze pomocí funkcionalit jako jsou *cookies ve webovém prihlížeči* velmi specificky rozdělit, kterou verzi konkrétní uživatel uvidí.

Pro smysluplné využití tohoto nasazení je potřeba existence možnosti vyhodnotit a porovnat výsledky testování nové verze oproti staré. Obvykle se tento typ nasazení používá k ověření nových designů a nových funkcí nové verze, čili se týká spíše *frontednu* (část aplikace běžící v prohlížeči uživatele). Lze ji ovšem stejně tak použít i pro "neviditelné"aspekty aplikační funkcionality, jako je například řazení relevatních výsledků vyhledávání, nebo přechod na jinou platební bránu. K tomuto účelů existují dedikované nástroje, jako je například Google Tag Manager.[22, 23, 24]

1.7 Strategie Shadow deployment

Shadow deployment pak kombinuje Blue/Green a A/B Testing, ale je zaměřena spíše na **testování výpočetní optimalizace nové verze a ověření výpočetní stability**. V této metodice beží nová verze vedle současné, ale uživatelé s ní nekomunikují přímo. Dostává ovšem stejné požadavky, jako produkční aplikace. Na jejích výstupech lze jednak porovnat a ověřit správnost výsledků obou verzí a jednak testovat výkon nové verze. To se zvlášť hodí například při nasazování nových nebo upravených algoritmů, přechodu na jiný databázový engine a podobně. Konkrétní podoba nasazaení pak záleží na metrice, která má být měřena.[25]

1.8 Další nasazovací strategie

Strategie, principy a metodiky popsané výše je vhodné kombinovat pro dosažení co nejelpších výsledků nasazovacího procesu. Velmi často se tak i děje - například je celkem běžné aplikovat strategii Blue/Green formou Rolling updates, ale stejně tak lze použít metodu Recreate. Je ovšem možné implementovat i zcela specifiký proces pro konkrétní případ užití. Vše se odvíjí od požadavků na funčnost strategie a nezřídka i na její praktické uchopitelnosti a proveditelnosti.

Další nasazovací strategie jsou si principielně velmi podobné a liší se jen v implementačním detailu, často pouze ve jméně. Cílem této práce není ”vyčerpávající”seznam všech strategií, které lze vyhledat a které byly definovány, jako například **Feature flags**, **Toggles**, **Dark Launching** a další. Cílem této kapitoly je poskytnout čtenáři stručný, analytický přehled o běžných typech nasazovacích strategií a ukázat jejich rozdíly, výhody a nevýhody a vhodná užití v praxi.

Kapitola 2

Migrování relačních databází

2.1 Relační databáze

Relační databázové systémy (RDBMS) zaujímají dominantní postavení na trhu databázových systémů, což je zřejmé z jejich přetrvávající popularity a širokého používání v mnoha oblastech. Dle DB-Engines Ranking[26], který hodnotí databázové systémy podle jejich popularity, se mezi nejpopulárnější databázové systémy řadí relační databáze jako Oracle, MySQL, Microsoft SQL Server a PostgreSQL. Tyto systémy vedou v hodnocení popularity a užívání, a jsou preferovány pro mnoho aplikací díky jejich výkonu, spolehlivosti a podpoře transakcí. Naopak, objektově orientované databáze a další typy ne-relačních databází, jako jsou NoSQL databáze, zaujímají specifické segmenty trhu, kde jsou vyžadovány jejich unikátní vlastnosti, ale celkově mají menší tržní podíl.

Co to ovšem relační databázové systémy jsou? Autor současně nejpoužívěnjšího databázové enginu, Oracle, říká:

> *"Relační databáze je typ databáze, která uchovává a zpřístupňuje data, které spolu souvisejí. Relační databáze jsou založeny na relačním modelu, intuitivním a přímočarém způsobu reprezentace dat v tabulkách."[27]*

Relační databáze tvoří tabulky provázané vazbami - relacemi. V relační databázi je každý řádek v tabulce záznamem s možností unikátní identifikace srkz takzvaným *klíč*. Relačním modelem je pak myšleno oddělení logické datové struktury od fyzických datových úložišť - datových tabulek, pohledů a indexů.[27, 28]

Migrování relačních databází v kontextu nasazovacích strategií obykle znamená **změnu databázového schéma**[29]. Lze se setkat i s významem změny databázoveho enginu, například migrace "z Oracle na MySQL". Tato práce se zaměří na změnu schéma, ačkoliv v případě migrace "mezi stroji", lze uplatnit podobné principy.

V optimálním případě nemusí být databáze nikdy migrována - zadání pro vývoj softwarové aplikace je naprosto jasné, na začátku tvorby architekrury je dobře navržené a po celou dobu existence aplikace jej není nutné měnit. Jistě lze najít i takové příklady.

V praxi se tak ovšem děje zřídkakdy. Na databázi je kladeno mnoho požadavků (ve smyslu vlastností) a ty se s časem mění a vyvíjejí. Častými důvody pro migraci jsou:

- Vylepšení výkonu - optimalizace schématu
- Přidání nebo upravení vlastností aplikace
- Integrace s novými systémy - požadavky na kompatibilitu

- Konsolidace a odstranění zastaralých dat
- Zvýšení bezpečnostních aspektů aplikace - například hašování hesel, šifrování na úrovni sloupce, ...
- Migrace na nové technologie nebo verze - i databázové systémy se vyvíjejí a některé vlastnosti mohou být vyřazeny či nahrazeny novými
- Normalizace - slouží k eliminaci redundance a zvýšení integrity dat
- Zjednodušení dotazů - refaktorizací schematu lze dosáhnout zjednoduššení dotazů, což obvykle vede k vyšší čitelnosti kódu, jeho snadnější údržbě a optimalizaci výkonu

Výčet důvodů jistě není kompletní, ale obsahuje pádné důvody, díky nimž má skutečně smysl důsledně se zabývat migrováním databází. Z praxe se ukazuje, že se jedná o činnost, která se dotýká takřka každého vydání nové verze aplikace.

2.2 Základní principy migrování databázi

Při práci s migracemi je zásadní podmínkou **dodržení kompatibility databázového schéma s kódem aplikace**. Z tohoto důvodu migrace úzce souvisí s nazsazovacími strategiemi. Při jejich běhu obvykle dochází k výměně kódu a proto je nutné databázové schéma (a v některých případech i data samotná) připravit na novou verzi aplikace.

Úpravy schéma relační databáze lze rozřadit do třech hlavních skupin v závislosti na tom, jaká změna schéma v dané skupině probíhá. Těmi jsou:

- Přidávání sloupců, vazeb, indexů, pohledů - aditivní, nedestruktivní
- Změny v současném schématu - přejmenování názvů, změna typů sloupců, slučování sloupců - modifikační, potenciálně destruktivní
- Odstraňování dat - destruktivní

Migrace se nicméně nemusí nutně týkat jen změny struktury schématu - migrace může měnit i data samotná, ať už pomocí nějakého druhu automatické konverze (například typ *řetězec* na typ *datum*) nebo modifikací dat samotných, například přidání předvolby k telefonním číslům. Tyto akce lze nicméně provádět i v rámci aplikačního běhu, tj. mimo migrace, a často se tomu tak i děje, pokud není změna poměrně triviální, až "statická"a lze ji provést v rámci migračních příkazů. I na tyto migrace se ovšem uplatňují obecné migrační principy.

Na migrování databází lze tedy pohlížet spíše jako na proces, než jako na izolované akce. Zároveň není nezvyklé, že nová verze kódu obsahuje více "migračních příkazů"najednou. Taková situace může nastat velice jednodušše - dva vývojáři pracují na dvou různých funkcionalitách, obě vyžadují manipulace s databázovým schéma a obě jsou naplánované k vydání do příští verze. Nijak spolu ovšem nesouvisí, takže vývojáři pracují naprosto samostatně a jejich kód se tak potká až ve výsledném "balíčku". Bylo by velmi neefektivní a obvykle i nebezpečné snažit se pak obě migrace spojit do jedné. To nutí i k verzování migrací, kdy je obvykle nutné i správné řazení migrací, respektive zajistit jejich aplikaci je ve správném pořadí.

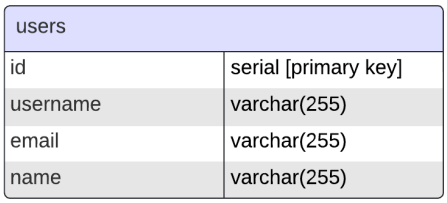
Migrace jsou tedy ze své podstaty **inkrementální**. Jedna migrace by v optimálním případě měla obsahovat pouze jeden typ změny, pro snadnější aplikaci, testování a případné řešení problémů.

Při každé další migraci kódu hovoříme o **"migrování nahoru"**, či migraci nahoru. Je to zažitý pojem, se kterým se často setkáme i v rámci nástrojů pro migraci, čímž se zabývá následující kapitola.

Z logiky věci tedy lze do migrací implementovat nejen funkcionalitu pro změnu schéma po skončení migrace, ale také **opačný proces této** změny, čili změnu z migrovaného stavu na stav před migrací, tzv. **"migraci dolů"**. Toho lze vcelku úspěšně využít pro ochranu dat, potažmo nasazovacího procesu a je to jeden z validních aspektů zotavení, ale pouhý fakt, že existuje i možnost migraci "zrušit"není samospásný a může vést k více problémům, než by měl řešit.

Migrace se provádějí v jazyku k tomu určému, SQL (Structured Query Language). Všechny nástroje, které představí další podkapitola, převádí abstraktní reprezentace schema a jeho migrací do SQL příkazů. Je tak vhodné tento jazyk použít i pro demostraci základních migračních princpů, doplněných o vizuální reprezentaci struktury a jejích změn.

Pro ilustraci typů migrací postačí velmi jednoduchá databázová tabulka bez relačních vztahů:



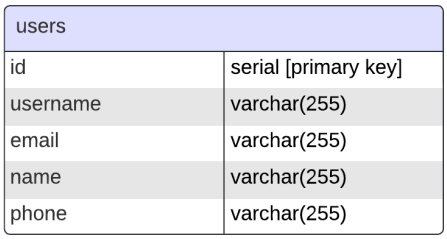
Obrázek 2.1 Základní schéma tabulky

Tabulka je velmi prostá, ale pro ilustraci principů modifikace schématu bohatě postačí. SQL kód, který takovou tabulku připraví vypadá takto.

Výpis kódu 2.1 Inicializace tabulky

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255),
    email VARCHAR(255),
    address VARCHAR(255)
);
```

První migrace demostroje aditivní princip. Ten představuje typický případ užití změny schéma - například dle nového byznys požadavku je třeba ke každému uživateli uchovávat i telefonní kontakt.



Obrázek 2.2 Tabulka po přidání telefonního čísla

Výpis kódu 2.2 Přidání telefonního čísla

```
    ALTER TABLE users ADD COLUMN phone VARCHAR(255);
```

Pomineme-li nutnost udělat změnu kvůli byznys požadavku, je tento princip nejvhodnější aplikací migrací. Přidaní sloupnce totiž nenarušuje kompatibilitu s běžnými SQL příkazy jako *SELECT*, *UPDATE* nebo *SET*, pokud není nový sloupec nastaven jako *mandatory* (”požadovaný”) a nemá defaultní hodnotu.

Po aplikaci této migrace lze stále úspěšně vyhodnotit následující dotazy:

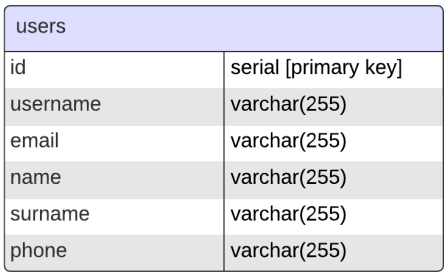
Výpis kódu 2.3 Přidání telefonního čísla

```
    SELECT * FROM users;
    SELECT id, name, email, address FROM users;
);
```

To přináší nespornou kompatibilní výhodu - pokud je v kódu chyba, která nepočítá s novou podobou schéma, neprojeví se a není pro aplikaci kritická.

Žádný z dalších principů tuto přívětivou vlastnost nemá. Pokud už nelze zabránti nekompatibilitě, je nutným minimem ochrana data a jejich integrity.

Dalším principem změny je změna modifikační. Ta upravuje schéma do jiné podoby, jak demonstruje nová podoba tabulky v 2.3.



Obrázek 2.3 Tabulka po destruktivní změně jména

Rozdělení jména v ne příliš prakticky navrženém způsobu jeho uložení je jistě validní požadevek. Toto rozdělení je nicméně nekompatibilní s předchozí verzí - ať už verze pracovala přímo se jménem sloupce *name*, nebo vracela celou tabulku a tu poté zpracovávala na úrovni kódu.

Dalším aspektem tohoto problému je samotný formát dat. Bylo by zřejmě příliš optimistické usuzovat, že každý uživatel bude mít jedno křestní jméno a jedno příjmení a je tak možné pouze rozdělit sloupec *name* na dvě části podle mezery. V tu chvíli vzniká problém jak s formátem, tak integritou dat. Toto je navíc velmi přímočarý a nekomplikovaný případ.

Tabulka 2.4 navrhuje poněkud citlivější přístup k problému a demonstruje nedestruktivní modifikaci sloupce pomocí aditivní metody:

users	
id	serial [primary key]
username	varchar(255)
email	varchar(255)
name	varchar(255)
name_name	varchar(255)
name_surname	varchar(255)
phone	varchar(255)

Obrázek 2.4 Tabulka po nedestruktivní změně jména

Výpis kódu 2.4 Rozdělení jména nedestruktivním způsobem

```
    ALTER TABLE users ADD COLUMN name_name VARCHAR(255);
    ALTER TABLE users ADD COLUMN name_surname VARCHAR(255);
```

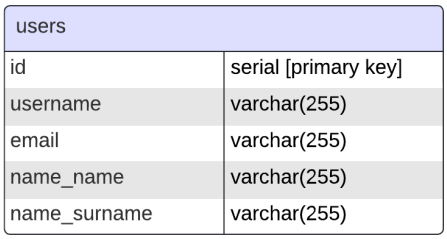
Tato změna zanechává původní data na místě a připravuje prostor pro data nová, v požadovaném formátu. V tomto případě by bylo možné i ”předvyplnit”nové sloupce pomocí heuristiky k tomu určené a přirozeně poskytnout uživateli možnost samostatně data upravit. To je jistě možné udělat i v předcházejícím případě. Nespornou výhodou tohoto principu je možnost zvrácení změn - původní data zůstavají na místě a lze se tedy vrátit k čase před migrací jak na úrovni kódu, tak na úrovni dat. Tento princip tedy jistým způsobem přistupuje k modifikaci jako k aditivu a teží z jeho výhod.

Nevýhodou pak je duplikace dat a možná mírná ztráta výkonu - tyto aspekty musí být brány vpotaz při rozhodování o typu a použití migrace, stejně jako příprava plánu pro odstranění těchto duplicit a konsolidaci dat.

Nejkritičtěším a nejnebezpečnějším principem je odstraňování dat z databáze. Chyba v tomto procesu může mít vážné následky a ne vždy je možné (či doknoce realistické) obnovit data ze zálohy. Tabulka 2.5 poměrně jasně ilustruje, co se stane se schéma (a daty), pokud provedeme následující migraci:

Výpis kódu 2.5 Odebrání sloupce a dat s telefonními čísly

```
    ALTER TABLE users DROP COLUMN phone;
);
```



Obrázek 2.5 Tabluka po desktrutivním odstranění telefonního čísla

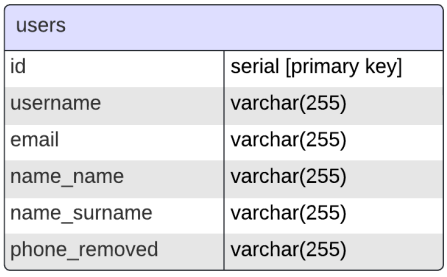
Není neobvyklá potřeba konsolidace dat v databázi, jak práce diskutuje na začátku podkapitoly. Odstranění nepotřebných dat je jedním z cílů této aktivity. Ne vždy lze ovšem bezpečně a bez problémů v zotavit ostraněná data. Proto je nanenejvýš nezbytné se na tyto procesy patřičně a s předstihem připravit a zajistit tak co nejmenší možné riziko nenávrátné ztráty dat.

V tomto modelovém případě bylo byznys požadavkem určeno neuchovávat telefonní čísla uživatelů a vývojář, poněkud optimisticky, připravil, odladil a aplikoval migraci pro žádanou změnu.

Po migraci bylo zjištěno, že telefonní čísla uživatelů jsou klíčovým prvkem v procesu fungování aplikace a bylo rozhodnuto obnovit je ze zálohy. Ta je ovšem tak velká, že její obnova by vyžadovala odstavení služby na několik hodin, což není možné

Taková situace není ojedinělá a je nanejvýš vhodné jí a jím podobným předcházet. V rámci konsolidace dat je jistě v jisté chvíli nutné data nezvratně odstranit. Takovou akci je ovšem obvykle možné odložit alespoň do chvíle, kdy je naprosto zřejmé, že data určená ke smazání již nepoužívá žádná část aplikace a nemají žádnou byznysovou hodnotu, takže jejich odstraněním nedojde ke škodě ani problémům s kompatibilitou.

Jednou z možností takovéto ”předčasné ochrany”je data, která v další verzi nebudou potřeba, skrýt na úrovni databáze. Toho lze dosáhnout prostým přejmenováním sloupce na hodnotu, která se v žádném dotazu navyskytuje (což lze zaručit jak statickou, tak dynamickou analýzou kódu) a nechat několik verzí používat takto modifikované schéma.



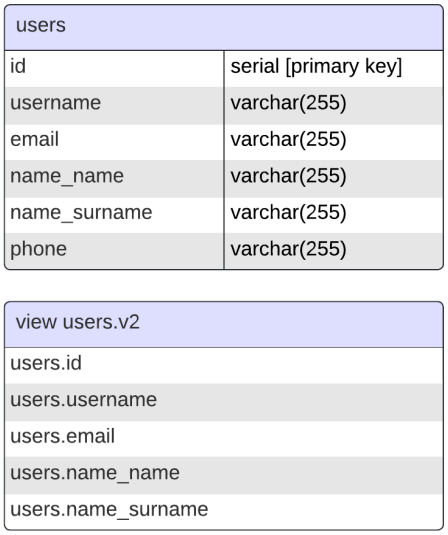
Obrázek 2.6 Tabulka po přejmenování telefonního čísla

Výpis kódu 2.6 Odebrání telefonního čísla nedestruktivním způsobem

```
    ALTER TABLE users DROP COLUMN phone;
);
```

Až ve chvíli, kdy je naprosto zřejmé, že aplikace operuje i bez těchto dat a že data již skutečně nejsou potřeba je možné naplánovat konsolidaci.

Dalším principem ochrany dat je užití pohledů. Vhodně užité pohledy data také fakticky skryjí před aplikačním kódem. V takovém případě se schéma de facto nemění a není tedy kritické, pokud by v aplikaci pořád existovala reference na předchozí definici schéma. Navíc nelze pohledy využívat pro modifikaci dat - důvod jejich existence má spíše charakter optimalizace vyhledávácích dotazů. Výhodou jejich užití je schopnost navrácení se k předchozí verzi schéma, neboť to se nezměnilo.



Obrázek 2.7 Tabulka s pohledem pro odstranění telefonního čísla

Principy lze do jisté míry kombinovat a každou migraci tak lze optimalitovat tak, aby bylo riziko ztráty dat a nekompatibility co nejmenší. Vždy je ovšem třeba zvážit všechny aspekty takové migrace a posoudit, jaký by mělo snížení výkonu či nárůst velikost databáze dopad na aplikaci samotnou a na její provoz.

2.3 Nástroje a technologie používané k migraci

Existují dva hlavní přístupy k migraci databází - **stavové** a **změnové** migrace. Stavové migrace prakticky znamenají manuální zásah do schéma - správce databáze je provádí pomocí příkazového řádku nebo GUI (Graphical User Interface). Pro menší a málo se měnící projekty můžou postačovat, ale obvykle jsou považovány za poněkud ”neohrabanou”, nebezpečnou metodu, pokud dochází k častým zásadním změnám ve schéma databáze. Změnové migrace se na druhou stranu vyznačují tím, že definují operace potřebné k převedení známého stavu databáze do požadovaného. Tento systém vyžaduje nepřerušený řetězec migračních souborů od počátečního bodu, od vytvoření prvních tabulek a relací, až po současný zamýšlený stav schéma. Volitelně pak inicializaci databáze (první migraci) může doprovázet i **”seed”**, neboli naplněni databáze iniciačními či testovacími daty.

Nedestruktivní SQL příkazy aplikované v modelové ukázce v předchozí kapitole představují stavové migrace, neboť nejsou nijak orchestrovány při vyšší složitosti by nemuslo být zřejmé ani jejich pořadí. Je tedy poměrně nepraktické používat je přímo a nevyužít migračního nástroje, neboť je velmi nebezpečné aplikovat všechny příkazy pokaždé, když je potřeba nějakou migraci

provést. Ne všechny databázové enginy totiž na všech modifikačních příkazech podporují statement *"IF NOT EXISTS"*, což by zapříčinilo havarování některých migrací. Správce migrace by tedy musel vést v patrnosti, které migrace již proběhly a které má provést při novém nasazování a to poskytuje příliš mnoho prostoru pro chybu.

I pokud je *"statement"* podporován, pořád existuje risk ztráty dat. Pokud byl například v průběhu aplikace napřed odebrán a pak opět přidán sloupec s telefonním číslem uživatelem, opakovaná migrace odebrání by odstranila všechna data, která se v databázi vyskytla po jeho opětovném přidání v některé z dalších verzí aplikace.

Migrační nástroje tento problém řeší tak, že samy implementují nějaký způsob sledování historie provedených migrací a jednodušše přeskakují již provedené migrace. To se typicky děje tak, že si migrační nástroj udržje svou vlastní tabulku v databázi, nad níž migrace probíhají, ve které drží informace o již proběhlých migracích.

Migrační nástroje existují v mnoha programovacích jazycích, všechny ovšem implementované migrace interně převádějí do SQL kódu (pokud jej nevyužívají přímo v implementaci migrací), který při aplikaci pouštějí nad databází.

Mezi velmi oblíbené migrační nástroje patří **Liquibase**[30, 31], otevřený nástroj postavený na jazyce Java, který podporuje definici změn databázového schématu v souborech *changelog* ve formátech XML (Extensible Markup Language, YAML (YAML Ain't Markup Language), JSON (JavaScript Object Notation) nebo SQL. Tyto soubory umožňují verzování a sledování historie změn, což zjednodušuje proces nasazení a zpětného odstranění změn v databázích (čili migrace "nahoru"i "dolů"). XML definice pak může vypadat například takto:

Výpis kódu 2.7 Iniciační migrace pro nástroj Liquibase

```
<?xml version="1.0" encoding="UTF-8"?>
<databaseChangeLog
    xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    <changeSet id="1" author="kotrlja1">
        <createTable tableName="users">
            <column name="ID" type="int" autoIncrement="true">
                <constraints primaryKey="true" nullable="false"/>
            </column>
            <column name="name" type="varchar(100)">
                <constraints nullable="false"/>
            </column>
        </createTable>
        <rollback>
            <dropTable tableName="users"/>
        </rollback>
    </changeSet>
</databaseChangeLog>
```

Při migraci databázového schéma s Liquibase je prvním krokem vytvoření nebo generování changelog souboru, který obsahuje definice změn (*"change sets"*). Tyto změny jsou poté aplikovány na cílovou databázi prostřednictvím příkazu *update*, který Liquibase vykoná po porovnání stavu databáze s definovanými změnami v changelogu.

Liquibase lze snadno integrovat s aplikacemi vytvořenými pomocí Spring Boot[32], což umožňuje automatické spouštění migrací databázového schématu při startu aplikace. Spring Boot konfigurace umožňuje specifikovat umístění *"master changelog"* souboru a další parametry, jako jsou kontexty migrace, což umožňuje jemnější kontrolu nad tím, které změny se mají v různých prostředích aplikovat.

Liquibase převádí migrace na jazyk SQL (pokud nejsou rovnou implementovány v tomto ja-

yzce) a historii změn drží v tabulkách **DATABASECHANGELOG** a **DATABASECHANGELOGLOCK**. Liquibase podporuje širokou škálu relačních databázových systémů, včetně MySQL, PostgreSQL, Oracle, Microsoft SQL Server a další. Díky tomu je možné používat Liquibase napříč různými projekty a týmy bez nutnosti měnit nástroj při změně databázové platformy.

Její nevýhodou může být nutnost přítomnosti běhového prostředí Java, což pro aplikace psané v jiných jazycích může znamenat zbytečné přidání závislosti, která s sebou nese všechna rizika s tím spojená. Autoři Liquibase nicméně poskytují docker obraz[33], který mohou vývojáři a operátoři využít, místo aby přidávali Java do svých vlastních prostředí.

Dalším, na Javě postaveným a oblíbeným nástrojem je **Flyway**[34], v mnohém podobný Liquibase. Jeho základem jsou migrační skripty, psané v SQL nebo "nativně"v jazyce Java. Svou historii uchovává v tabulce **flyway_schema_history**. Java kód pro inicializaci tabulky users:

Výpis kódu 2.8 Iniciační migrace pro nástroj Flyway

```
public class DatabaseMigration {
    public static void main(String[] args) {
        Flyway flyway = Flyway.configure()
                .dataSource(
                    "jdbc:yourDatabaseUrl",
                    "yourUsername",
                    "yourPassword"
                )
                .locations("classpath:db/migration")
                .load();

        flyway.migrate();
    }
}
```

Flyway podporuje jak verzované, tak opakovatelné migrace, přičemž verzované migrace mají jedinečnou verzi a aplikují se právě jednou, zatímco opakovatelné migrace se aplikují vždy, když se změní jejich kontrolní součet. Veškeré migrace se provádějí v rámci jedné databázové transakce, což zajišťuje konzistenci databáze.[35]

Populárním migračním nástrojem je i **Sequelize**[36], Node.js ORM (Object-Relational Mapping) knihovna pro relační databáze jako jsou PostgreSQL, MySQL, MariaDB, SQLite a Microsoft SQL Server. Každá migrace obsahuje dva základní aspekty: *up* a *down* (tedy migrace "nahoru"a "dolů"). Sequelize umožňuje vytvoření inicializačního modelu a automatické generování migrace, které vyžaduje specifikaci jména modelu a atributů. Pokud je aplikace, pro kterou jsou migrace připravovány, psána také v Node.js (respektive v JavaScriptu, či případně v modernějším TypeScriptu), je výhoda užití toho nástroje zřejmá. I v Sequelize lze implementovat přímo SQL pžíkazy, jde to nicméně proti konceptu relačního mapování abstraktních entit na objekty. Historii migrací pak Sequeliye uchovává v tabulce **SequelizeMeta**. Migrace vytvoření tabulky pak může vypadat například takto:

Výpis kódu 2.9 Iniciační migrace pro nástroj Sequelize

```
'use strict';

module.exports = {
  up: async (queryInterface, Sequelize) => {
    await queryInterface.createTable('users', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      name: {
        type: Sequelize.STRING(100),
        allowNull: false
      }
    });
  },

  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('users');
  }
};
```

Posledním nástrojem, který práce zmíní, je otevřená knihovna **node-pg-migrate**[37]. Jak název napovídá, knihovna je implementovaná pro JavaScript (a také pro modernější TypeScript) a podporuje pouze databázový engine PostgreSQL. Díky tomu je ovšem pro tuto implementaci databázového stroje optimalizovaná.

Kapitola 3

Problémy a rizika migrace databází

3.1 Identifikace potenciálních problémů

Z analýzy migračních principů a nasazovacích strategijí lze shrnout potenciální problémy a rizika do několika bodů:

- Selhání migrace
- Ztráta dat způsobená migrací
- Nekompatibilita aplikačního kódu a databázové schéma po migraci

Selhání migrace může mít několik různých příčin. Asi nejtypičtějším důvodem je chyba v kódu migrace. Další příčinou pak může být přerušení způsobené přerušením konektivity mezi migračním prostředtím a databází samotnou, nebo pád procesu, a to jak aplikačního, tak serverového.

Před přerušením spojení a pády procesů by měl provozovatele aplikace ochránit migrační nástroj. Všechny výše zmiňované nástroje migrují databáze v takzvaném *transakčním* módu. Ten, zjednodušeně řečeno, zajišťuje, ze se SQL příkazy provedou buď všechny, nebo žádný. Také ”zamyká”obsah databáze po dobu transakce - nemůže se tak stát, že během migrace dat nebo schématu jsou v databázi měněna data. Tyto příkazy jsou pozdrženy a aplikovány až po skončení transakce, což klade další požadavky na kompatibilitu.

Ztráta dat způsobená migrací obvykle představuje ještě vážnější hrozbu. Ani periodické zálohování dat nemusí stačit jako ochrana před tímto problémem. Databáze velkých projektů mohou nabývat velikosti více než stovky gigabajtů, pracovat pak s takoými daty ve smyslu obnovy je v nejlepším případě obtížné a zdlouhavé. Jedinou relevantní ochranu před ztrátou dat představuje používání nedestruktivních principů při implementaci migrací a skutečně důsledná verifikace dat, která byla dostatečně dlouho dobu v jakési ”karanténě”- aplikace je již nepotřebovala a nepoužívala a ani z byznysového pohledu skutečně nejsou potřeba.

Nekompatibilita aplikačního kódu a databázové schéma po migraci v sobě obsahuje několik scénářů, ke kterým v tomto problému může dojít. Nekompatibilita kódu se současným stavem schéma může způsobit pády aplikace, a to jak hned po jejím startu, tak během chodu a zpracovávání uživatelské akce. Obě varianty mají velmi nepříjemný dopad na uživatelský zážitek.

Dále, pokud je kompatibilita takového rázu, že se data přepisují jiným způsobem, než byl autorův záměr (například jsou někde prohozená jména sloupců), dochází ke kontinuálnímu znehodnocování dat, které se může projevit či odhalit až dlouho poté, co existovala relevantní záloha

dat. Ta navíc neobsahuje změny provedené po problematickém migrování a úplná záchrana dat je pak velmi komplikovaná, nezřídka zhola nemožná.

Obecně lze pak stanovit, že byť nelze zaručeně zanránit každé problematické situaci, která může v tomto konktextu nastat, lze jim předejít důsledným dodržováním základních ”best practices”vývoje softwarových aplikací. Těmi jsou:

- *code reviews* - revize (kontrola) kódu jiným vývojářem, než je jeho autor, před nasazením do produkce
- *CI/CD* (Continuous Integration/Continuous Deployment) - kontinuální integrace a kontinuální nasazování
- Automatizované i manuální testování migrací

3.2 Strategie pro minimalizaci rizik

Code reviews, neboli *revize kódu*, jsou klíčovým procesem ve vývoji softwarových aplikací, a zvlášť v problému migrování relačních databází. Především umožňují identifikovat a opravit chyby v rané fázi vývojového cyklu, což vede k ušetření času a nákladů spojených s pozdějšími opravami. Zvyšují také kvalitu kódu tím, že podporují dodržování osvědčených postupů a standardů v psaní aplikačního kódu, neboť i migrace jsou zdrojovými kódy. Code reviews také podporují sdílení znalostí mezi členy týmu, což vede k lepšímu porozumění projektu jako celku a k rozvoji dovedností vývojářů. Navíc mohou pravidelné revize kódu zlepšit bezpečnost, a to nejen aplikace samotné, ale i její databáze, právě srkz migrace. Zároveň otevírají prostor pro optimalizaci dotazů a databázové struktury jako celku. Revize kódu zvyšují spolupráci v týmu a vedou k dramatickému snížení migračního rizika.

Pokud lze o některém z aspektů aplikačních testů říct, že je skutečně kritický, je to jistě **testování migrací**. To by mělo být zevrubné, opakované a ”úplné”. Je velice vhodné testovat jak celý migrační řetěz (od inicializace až po nejposlednější migraci), tak i migrace dílčí. Některé nástroje (napřkílad Sequelize) umožnují pouštět i konkrétní migrace. Je tedy na nejvýš vhodné již ve fázi testování kódu pracovat nad stavem, jaký je na produkci, a při každém běhu testů provádět všechny nové migrace. Zároveň je vhodné otestovat i migrování ”dolů”, čili z nového stavu na stav součané produkce. Tím se odhalí jak chyby v migracích nových, ve smyslu nahoru, tak případné potíže se zotavením pomocí migrací dolů. Pro testování migrací je vhodné naplnit databázi daty, která následné integrační testy dokáží otestovat. Nejlepším případem je pak možnost testovat nad reálnými produkčními daty, které se zkopírují z produkční databázové instance. To je přirozeně velmi často problematické - data podléhají utajení, rapidně se zvyšuje riziko úniku dat, a produkční data mohou mít řádově vyšší stovky gigabajtů. Lze ale silně doporučit periodicky připravovat *anonymizovaný*, *reprezentativní* vzorek produkčních dat. Při skutečně důkladném testování se pak velmi snižuje pravděpodobnost, že dojde k prolému při ”ostrém”nasazení. Testování obvykle dokáže odhalit všechny výše zmíněné druhy problémů s migrováním databází - odhalit chyby v migračních skriptech, zjistit případnou ztrátu integrity dat a zajistit dostatečnou persistenci.

CI/CD systémy hrají naprosto klíčovou roli v předcházení migračních rizik. Počet opakování migrací nad daty zvyšuje možnost, že se projeví skrytá chyba a dává tak jak vývojářům, tak testerům možnost odhalit ji včas. V optimálním případě je CI/CD systém kompletně automatizovaný a vývojář se tak nemusí o nic starat a zároveň se odbourává ”lidský faktor”.

Ruku v ruce s CI/CD systémy jde i samotný proces vývoje softwaru. Pokud provozovatel aplikace implementuje kód přímo na produkčním stroji a proti produkčním datům, není pro CI/CD užití. Je nanejvýš vhodné mít alespoň dvě prostředí - jedno pro vývoj a jedno produkční. Větší, zaběhlejší a zkušenější týmy pak obvykle jdou ještě dál a mají prostředí per vývojáře a v nejlepším možném případě mají několik stabilních prostředí (například *dev*, *staing* a *prod*) a dále pak dynamicky vytvářená prostředí per změnu kódu. Ta může být reprezentována například

pull requesty nebo *merge requesty* (za předpokladu, že vývoj probíhá nad verzovaným prostředím systému git).

Síla CI/CD se pak projevuje zvlášť v kombinaci s automatickým testováním nad všemi vzniknuvšími prostředími. Optimálním scénářem je testovat každou dílčí změnu kódu, granularitně začínající na úrovni PR (Pull Requestu). Testovat jednotlivé *commity* je obyvkle příliš časově a finančně náročné. Pokud každý PR navíc manuálně otestuje tester (jak na úrovni aplikaci, tak na úrovni integrity dat), riziko migračního problému se velice eliminuje.

Kapitola 4

Migrovaní relačních databází v nasazovacích strategiích

4.1 Strategie migrování

Důležitým aspektem strategie migrování pro jakoukoliv nasazovací strategii je její **načasování**. Při otevření běžné aplikace implementované v Node.js lze v souboru *package.json* nalézt tento kód pro spuštění aplikace v produkčním prostředí:

Výpis kódu 4.1 Migrace jako součást startovacího příkazu

```
{
    ...,
    "start:prod": "npm run migrate && npm start",
    ...
}
```

Autor kódu tím jasně dává najevo, že migrace jsou součástí startovacího procesu a jsou tak neoddělitelně spjaty s nasazením kódu a inicializací aplikace.

Tento přístup je velmi obvyklý a je typický pro menší projekty, POC projekty, projekty v ranné fázi vývoje a pro neprodukční prostředí.

Obdobně je pro konteinerizované aplikace obvyklé užití *initContaineru* ve stejném smyslu a principu:

Výpis kódu 4.2 Migrace jako initContainer

```
initContainers:
  - name: kotrlja1-demo-app
    image: kotrlja1:v1
    imagePullPolicy: Always
    command: ["/bin/sh", "-c"]
    args:
      - |
        npm run migrate
```

Pokud se tým nerozhodne migrovat produkční databázi manuálně, oba výše popsané způsoby představují validní užití migrační metodiky. *initContainers* má nicméně značnou nevýhodu - při

nasazování dvou a více instancí (*"replicas"*) se pokusí migrovat v každé z nich, což sice ve výsledku funguje, ale vygeneruje velké množství chybových hlášek, protože první konteiner, který začne migrovat zablokuje spojení pro všechny ostatní, poté druhý a tak dále. *initContainers* tedy není ideálním řešením. Variantou je použít jiný druh kubernetes zdroje, ve kterém lze indentifikovat konkrétní repliku a migrovat pouze v ní - to má ovšem řadu dalších úskalí.

Lepší variantu představuje *batch job*. Ten běží před samotným nasazením a poskytuje tak ideální příležitost pro migraci. Jeho nastavení navíc podporuje možnost pustit jej pouze jednou, takže nedochází k opakovaným pokusům a nasazení správně skončí po prvním pokusu, ať už je úspěšný či neúspěšný.

Problém všech těchto nástrojů tkví v tom, že neexistuje zpětná vazba mezi startem aplikace a migrací. Pokud se migrace zdaří (ve smyslu migračního příkazu), aplikace se spustí a pokud selže, je na migrační strategii, respektive obvykle na operátoru nasazování, aby vyřešil náhle vyniklý problém.

Do jisté míry lze automatizovat i migrování dolů, ale v takovém případě je nutné řešit kolik migrací dolů má zotavovací proces provést. Migrace navíc nesmí být destruktivní, protože pak se databáze dostává do stavu, kdy byla provedena nezvratná migrace, ale s databází komunikuje neaktuální verze kódu. Kompatibilitě kódu se dále věnuje následující kapitola.

Praxe ukazuje, že pro velké projekty je výhodné **oddělit vydávání verzí kódu a migrování databázového schématu**. Princip tohoto rozdělení je následující:

1. Vzniká požadavek na úpravu schéma
2. Připraví se vydání **pouze kódu**, který je kompatibilní se současnou i novou podobou schéma
3. Nová verze se vydá do produkce
4. Připraví se vydání **pouze nových, nedestruktivních migrací**
5. Nová verze migrací se vydá do produkce

Tento proces přidává další ochranu dat a kompatibility mezi kódem aplikace, databázovým schéma a daty. Oproti předchozím příkladům, ve kterých je migrace těsně spjatá s vydáním nové verze kódu, poskytuje výhodu jednoduchého přechodu na předchozí verzi bez nutnosti manipulace s databází. Dále pak při případném problému s novou verzí migrací (nebo i aplikací samotnou) lze "pouze"odmigrovat dolů a není třeba měnit verzi aplikace, což může být samo o sobě problematické ve smyslu výpadku služby, jak disktuje první kapitola.

Naopak pro proces obnovy dat (pokud není potřeba provést obnovu dat co nejdříve) platí obrácený postup. Nejprve vydat verzi s migracemi, které navrátí schéma i s daty do původní podoby, a poté buď vrátit aplikaci na původní verzi, která data i skutečně využívá, nebo vydat novou verzi kódu. Půdvodní verze nicméně nemá žádné informace a nových migracích, které schéma upravily, je tedy nevhodné tento proces automatizovat bez dalších obslužných nástrojů a je výhodnější vydat novou verzi, která má původní kód, ale nové migrace. Nejvýhodnější je pak nespoléhat se na "migrace dolů"vůbec, a v případě nutnosti změny jednodušše vydat novou verzi migrací, které vracejí schéma do požadovaného tvaru. Tímto způsobem implementace úplně opadá potřeba implementovat a řešit i "migrace dolů", které mohou být samy o sobě poměrně nebezpečné. Vede to i na další procesní doporučení, a to **oddělovat verzování aplikace a migrací**, potažmo oddělovat i jejich kód. Na úrovni konteinerů to lze provést poměrně přímočaře - vznikají obrazy jak pro aplikaci, tak pro migrace.

Koncept má ovšem celou řadu úskalí. Vyžaduje velkou míru disciplíny při plánování vývoje a vydávání nových verzí. Je náročnější na manuální testování, neboť tester musí brát v potaz dvě verze aplikace a dvě verze databázového schéma. Velmi komplikuje možnosti *"hot-fixů"*, pokud je jich nutno a přicházejí v době mezi vydáním kódu a vydáním migrací.

I při použití této robustnější metody je ovšem stále nutné mít na paměti migrační principy a

vyhýbat se destruktivním operacím. Před těmi obecně neposkytuje ochranu žádná strategie a v takovém případě nezbývá než se spoléhat na databázové zálohy.

Validní strategií migrování databáze je i varianta krátkodobé odstavení služby od provozu. Lze říci, že se jedná o běžnou praxi, jak vyplývá z analýzy v první kapitole, kdy ani společnosti s mnoha mililiardovými zisky neztácejí na reputaci a nezažívají odchod uživatelské základny při aplikaci tohoto principu.

Ten má celou řadu výhod - při plánování výpadku lze zahrnout i prostor pro případnou nápravu chyb. Lze více testovat jak migrace tak aplikace, a dělat tak s větší segmentací dat. Týmy zodpovědné za všechny aspekty nasazování nové verze jsou obvykle dostupné a nehrozí tak, že v případě problému není aktivní člen týmu, který by byl schopen problém řešit. I v případě destruktivních migrací pak lze při dobrém plánování přistoupit i k tak razantnímu kroku, jako je obnova mnoha set gigabajtových záloh.

Výčet výhod tím jistě nekončí, nicméně proti němu stojí fakt, že služba nějakou dobu (i vyšší jednotky hodin) není v provozu. Při včasné a zřetelné komunikaci je tento problém do jisté míry řešitelný (respektive je to pro uživatele bez pochyby lepší varianta, než neplánovaná několikadenní odstávka).

Obvykle je zodpovědností vyššího managementu, aby posoudil dopad odstávky na uživatelskou základnu, kvalitu služby a v neposlední řadě potenciální finanční impakt.

4.2 Kompatibilita aplikace a schéma

Princip odděleného vydávání kódu a migrací požaduje kompatibilitu mezi kódem a alespoň dvěma verzemi databázového schématu. Tato vlastnost je nicméně implicitně vynucena všemi nasazovacími strategiemi popsanými v první kapitole, s vyjímkou strategie Recreate.

Tento fakt si lze snadno uvědomit při představě aplikace, která je zároveň spuštěna ve více instancích (a potažmo na více strojích). Strategie Recreate ve svém principu napřed všechny instance zastaví a posléze spustí v nové verzi. Jinými slovy se nikdy ”nepotkají”dvě různé verze kódu, respektive dvě různé verze kódu vyžadující různé schéma.

Ostatní diskutované strategie ze své podstaty vyžadují kompatibilitu, byť pouze na omezenou dobu. Strategie Rolling updates by v těsně spjatém principu migrovala s první instancí (nebo první dávkou) a v případě nekompatibility by pak zbytek nasazení přestal fungovat, což by jistě mohlo způsobit celou řadu problémů. Strategie jako A/B Testing nebo Blue/Green by pak principiálně přestanou úplně dávat smysl, protože při existenci migrací v nové verzi by nikdy nemohly využít svých výhod.

4.3 Zotavení

Při dodržování migračních principů by dle jejich teorie nikdy nemělo dojít k nenávratné ztrátě dat. I přesto je nanejvýš vhodné udržovat **zálohy databázových dat**, a to v několika časových snímcích, pokud samozřejmě velikost databáze nečiní tento požadavek nereálným.

Samotné zálohy ovšem nestačí, je naprosto nutné mít k nim vypracovaný, otestovaný a aktualizovaný **scénář obnovy dat**. Ten obvykle říká odkud a jak data obnovit a za jakých podmínek (např. za podmínky odstavení služby od provozu).

Potřeba obnovit data je micnéně vždy komplikovaná a obvykle při ní nejde udržet aplikaci ve stavu HA (High Availability). Obecné doporužení je implementovat a navrhovat migrace a kód tak, aby k nutnosti obnovení nemuselo nikdy dojít.

S tím se pojí i občasná nutnost přejít na předchozí verzi aplikace. To se zdaleka nemusí týkat jen problémů s migracemi a s databází obecně. V každém kódu se může vyskytnout chyba, která nejde rychle opravit, nebo je to příliš riskatní se o to pokoušet. Nutnost může přijít i vynucením od služby třetí strany - i v jejich API se může objevit chyba. Ale lze si představit i byznysový požadavek.

Stejně jako pro migrace i pro aplikaci platí, že musí existovat proces pro takový přechod a musí být otestovaný a aktualizovaný. Musí také stále existovat verze, na kterou je požadavek se navrátit. Musí k ní existovat správná konfigurace. A verze přirozeně musí být kompatibilní, nejen s databázovým schéma, ale i se službami, které integujre.

Takto lze shrnout základní principy pro zotavování. Jejich konkrétní užití a implikace diskutuje další podkapitola.

4.4 Migrační a zotavovací scénáře nasazovacích strategií

Ze strategií diskutovaných v této práci stojí trochu stranou strategie **Recreate**. Ta jako jediná striktně nevyžaduje kompatibilitu kódu a databázového schéma mezi dvěma po sobě jdoucími verzemi. To je vlastnost, která má uplatnění ve dvou případech:

1. Pokud je aplikace a její architektura, infrastruktura a datábázový model v tak ranné fázi vývoje, že je praktičtější každé prostředí plně recyklovat
2. Pokud je potřeba provést *"breaking change"* změnu

V prvním případě ovbykle dojdeme do stavu, kdy se z počátku prudce měnící vlastnosti aplikace stabilizují a tento typ nasazování přestane být praktický a začne být nebezpečný (zvlášť od druhého vydání do produkce dále).

V případě druhém je to skutečně velice krajní, až neortodoxní řešení a nelze jej doporučit.

I pro strategii Recreate je možné psát migrace nedestruktivním způsobem. Je to nakonec vlastnost migrací samotných a nikoliv vlastnost nasazení. V případě Recreate se s procesem nasazení a migrace obtížně manipuluje. Nově nasazovaná verze totiž nemá žádný kontext od verze předchozí - je obtížné zjistit, o kolik migrací migrovat dolů a hrozí tak riziko nárůstu nekompatibility. Nepraktické je to i pro testování procesu migrace a zotavení. Celý koncept je poměrně náročný na automatizaci - pro vypořádání se s překážkami je obvykle nutné implementovat velké množství obslužných nástrojů. Navíc se strategie nedokáže vyhnout výpadku služeb, který se s neproběhnuvší migrací či aplikačními problémy znatelně prodlužuje.

U této strategie nezbývá než spolehnout se na zálohu dat a procesy obnovy, které je samo o sobě náročné udržovat a implementovat, aniž by to navíc komplikoval princip strategie samotné.

Všechny ostatní strategie diskutované v této práci lze označit jako **mandatorně-kompatibilní**. Platí pro ně tedy stejná pravidla v kontextu nasazování migrací a obnovy, což je velice výhodné, neboť pak lze, za jistých podmínek, měnit strategie podle potřeby. To samo o sobě s sebou přirozeně nese své vlastní výzvy, ty se nicméně netýkají migrování relačních databází.

Mandatorně-kompatibilní nasazovací strategie tedy vždy nejprve nasadí novou verzi pouze na část produkčních instancí (na jak velkou a na jak dlouho odpovídá první kapitola této práce).

V případě opomenutí požadované kompatibility (a neodděleném vydávání verzí kódu a migrací) se pravděpodobně stane, že první nasazená "dávka"odmigruje databázi a pokud se nic nepokazí, nasazovací proces pokračuje dalšími dávkami. To nicméně znamená, že zbývající produkční instance prakticky přestanou fungovat, respektive přestane fungovat jejich aplikační část, která neumí pracovat s novým schéma databáze.

To je situace, se kterou se dokže vypořádat Rolling updates, a to tak, že postupem času donasadí zbytek instancí. Do procesu ovšem může vstoupit například monitoring, který zjistí, že velká část produkční aplikace vykazuje chybovost a podle toho buď jenom alertovat, nebo v případě velmi aktivního monitorovacího procesu i jednat a nasazování úplně zastavit.

V horším případě nasazení selže již na první dávce, a to buď rovnou na migracích, nebo na spuštění aplikačního procesu nové verze. V obou případech je nutno řešit, do jakého stavu se databáze, respektive její schéma dostalo, a jak se s tím vypořádat.

Při použití nedestruktivních migračních principů nevzniká potřeba migrovat zpět, protože i změněné schéma je stále kompatibilní s předchozí verzí, respektive je jednoduché se k němu vrátit. V opačném případě je nutné obnovit zálohu databáze.

Pro strategie Blue/Green, A/B testing a Shadow deployment nekompatibilita naprosto znemožňuje použitelnost těchto nasazení. Část instancí je dedikována, aby běžela v jiné verzi libovolně dlouho, pro smysluplné použití nejméně v řádu dnů, ale spíše týdnů a měsíců. Nekompatibilita efektivně "odstřihne"celou jednu část nasazení, což je více či méně kritické pro každou z trojice jmenovaných.

V rámci A/B testingu to znamená, že buď menší, nebo naopak větší skupina uživatelů buď nebude moci použít aplikaci vůbec, nebo, v lepším případě, nástroje obsluhující instance samotné označí část B jako neoperující a celé nasazení se pak degraduje na "A Testing".

U nasazení Blue/Green se opět setkáme s degradací, a to na "Green nasazení", protože přepnutí zpět na Blue není díky nekompatibilitě možné.

Nejvíce kritická situace nastává v případě Shadow deployment - nová verze nemůže běžet vůbec, a nebo naopak běží pouze nová verze, přičemž předchozí verze nedostává žádný provoz, takže pro uživatele je služba efektivně nedostupná.

Pokud je tedy kompatibilita nastavena jako požadavek, problém migrování databázi se značně redukuje. Redukuje se dokonce do takové míry, že je implicitně vynuceno užití nedestruktivních migrací. Aby se i mezi verzemi kódu, které se liší v potřebě například konkrétního sloupce, docílilo kompatibility, nesmí nová verze (s novými migracemi) sloupec smazat a dokonce ani nedestruktivně přejmenovat. Elegantním způsobem je tak dosaženo optimálních principů jak z pohledu nasazovacích strategií, tak z pohledu migrování relačních databází.

Může se tedy zdát, že princip oddělení vydávání verzí kódu a migrací pozbývá smysl. Není již absolutně nutný, nicméně je stále výhodný. Zabraňuje totiž nutnosti migrovat zpět při problému s novou verzí aplikace - ta se totiž díky skutečně silné kompatibilitě může kdykoliv vrátit o verzi zpět a drasticky se tak redukuje prostor pro jakoukoliv nedostupnost služby. Proces nasazení je díky oddělení také jednodušší - lze se zaměřit buď pouze na nasazení kódu, nebo nasazení migrací, změnšuje se komplexnost celého procesu, což je vždy pozitivní aspekt.

Není záhodno opomenout potřebu zálohovat data a udržovat aktuální a funkční procesy pro obnovu databáze. I sebepřísnější proces, principy a metodika nedokážou pokrýt všechny možné problémy, které mohou nastat a ztráta dat je obvykle nevratná a zřídkakdy neústí v závažné problémy. I při neúnosně velkém objemu dat je třeba vyvinout maximální snahu držet alespoň jednu zálohu pro případ nouze.

Kapitola 5

Plán nasazení, migrace, zotavení a testování strategie A/B Testing

5.1 Architektura nasazení, aplikace a migrování

Demonstrace principů proběhla na strategii nasazení A/B Testing, protože její orchestrace není triviální (není implementovaná jako typ *deploymentu* v kubernetes konfiguraci) a lze na ní názorně ilustrovat principy, jež diskutuje a analyzuje tato práce.

Testovací aplikace představuje API server, implementovaný v jazyce *Node.js*. Incializaci a migrování obstaral migrační nástroj *node-pg-migrate* a databázovým enginem byl tedy zvolen *PostgreSQL*.

API server není typickým kandidátem pro A/B Testing, neboť v praxi se touto metodikou testuje spíše uživatelská odezva na změny designu a obecně funkcionality viditelné uživatelem služby. Změna funkcionality nicméně může znamenat i změnu v API části aplikace (například změnu v algoritmu pro vyhledávání výrobků), což může následně vyžadovat změnu v databázovém schéma, konkrétně představenou optimalizací tabulek. Případ užití tedy není uměle vytvořený a lze tak uvažovat o praktickém použití. Pro demonstrační účely bude navíc hodnotnější prezentovat výstupy aplikace jako krátké odpovědi serveru, než jako snímky obrazovky.

Řízení žádostí mezi A a B instancemi bylo pro jednoduchost kontrolováno skrz HTTP (Hypertext Transfer Protocol) hlavičku, aby bylo deterministicky možné dostat odpověď od prostředí A nebo prostředí B.

Pro nasazení byly implementovány obslužné nástroje pro zálohování a obnovu databáze (která byla z důvodů praktičnosti uložena v rámci kubernetes zdrojů na stejném clusteru). Tyto jsou plně automatizovány na úrovni nasazovacího procesu, což v případě zálohování automatizuje tento proces (jak periodicky, tak při novém nasazení) a pro obnovu dat minimalizuje možnost chyby.

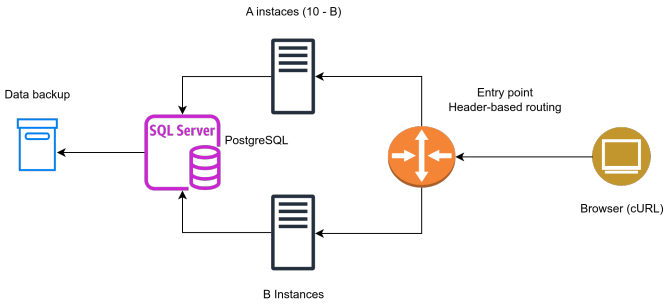
Jako kubernetes posloužilo **minikube**[38] - minimalistický kubernetes klastr, vytvořený pro snadné lokální použití, bez nutnosti inicializace celého kubernetes systému. Minikube je navíc integrováno s dockerem, což demonstraci dále usnadňuje.

Aplikace byla pro použití v kubernetes konteinerizována. Reálný projekt by byl verzován pomocí systému k tomu určenému (git), pro účely demonstrace byla však aplikace verzována v rámci složek, aby bylo možno sledovat vývoj a proces testované aplikace (a testovaného nasazení).

Pro konfiguraci kubernetes byl použit šablonovací jazyk **Helm**[39]. Díky němu je aplikace schopná například dynamicky měnit poměr A a B instancí, což je u A/B Testing strategie užitečná vlastnost. Dále lze pak také díky proměnným prostředí řídit nasazovací proces, čehož bylo využito pro samostané nasazení databáze, určení verze kódu a migrace a v neposlední řadě i k obnově

databázových dat ze zálohy.

Vizuálně architekturu reprezentuje tento diagram:



Obrázek 5.1 Diagram demontrované architektury

Strategie A/B testing pracuje s dvěma nasazeními - ”A”pro stabilní verzi a ”B”pro testovací verzi. V demonstraci se v každém nasazovacím procesu nasazasují obě prostředí, byť v některých částech obě na stejnou verzi. To je výhodné, protože pak lze migrování (a případné seedování) automatizovat a bylo by nebezpečné, kdyby se seedovat a migrovat pokoušela obě prostředí. Takto byly tyto procesy delegovány na prostředí ”B”, které má typicky novější funkcionality, než ”A”.

Dle analýzy strategie pro bezpežné migrováná jsou verze (a docker obrazy) aplikace a migrování striktně odděleny. Díky tomu lze demonstrovat i bezpečné odebírání dat z aplikace.

Nasazení dále využil kubernetes zdroj *apps* typu *Deployment*, který automaticky nasazuje nové konteinery strategií Rolling updates. To je velmi výhodné, neboť se tím snižuje riziko nedostupnosti služby - nové verze dostávají požadavky až ve chvíli, kdy je klastr prohlásí za přípravené

5.2 Implementace nasazení a obslužných nástrojů

Nasazení muselo být upraveno pro jednoduché lokální použití. Například použilo databázi přímo v kubernetes zdrojích a bylo tedy nutné přizpůsobit pořadí nasazování tak, aby databáze již existovala v době, kdy se pouští první migrace. V klasickém produkčním nasazení by databáze byla s největší pravděpodobností umístěna mimo klastr (protože v něm není doopravdy persistentní) a tento problémem by se neprojevil.

Dalším ústupkem se stalo seedování databáze. Aby byl lokání testovací proces co nejjednodušší, opět byl zvolen postup ”řetězového nasazení”, kdy se databáze seeduje pouze pokud se nastaví dedikovaný přepínač (v tomto případě jím byla Helm proměnná).

Pro procesy zálohu a obnovy databáze posloužily kubernetes zdroje typu *batch*, a to konkrétně *CronJob* (pro periodické zálohování) a **Job** pro migrování, seedování a obnovu databáze.

Všechny tři zdroje byly skrz Helm řízené proměnnými, přičemž byly všechny tři volitelnou součástí každého nasazení nové verze. Seedování na obykle nemá své zdroje vůbec - v tomto případě bylo použito pro jednoduchost demonstrace - nebylo tak nutné implementovat do aplikace funkcionalitu pro naplnění databáze daty.

Výpis kódu 5.1 Konfigurace pro periodické zálohování databáze

```
---
apiVersion: batch/v1
kind: CronJob
metadata:
  name: ab-testing-db-backup
spec:
  schedule: "3 2 * * *"
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: Never
          containers:
            - name: ab-testing-a-backup
              image: postgres:15.5-alpine3.19
              imagePullPolicy: Never
              command: ["/bin/bash", "-c"]
              args:
                - |
                  pg_dump -O -x $DATABASE_URL \
                    > /backup/database_backup_$( date +%s ).sql
              env:
                - name: DATABASE_URL
                  value: "postgresql://ab:ab@ab-testing-postgres/ab"
```

Výpis kódu 5.2 Konfigurace obnovu schéma a dat databáze

```
{{- if .Values.RESTORE_DATABASE }}
---
apiVersion: batch/v1
kind: Job
metadata:
  name: ab-testing-restore
  annotations:
    "helm.sh/hook": pre-upgrade
    "helm.sh/hook-weight": "-10000"
    "helm.sh/hook-delete-policy": before-hook-creation
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: ab-testing-a-restore
          image: postgres:15.5-alpine3.19
          imagePullPolicy: Always
          command: ["/bin/bash", "-c"]
          args:
            - |
              to_restore=$( \
                find /backup -type f -exec ls -t1 {} + | head -1 \
              )
              if [[ -n $to_restore ]] ; then
                psql ${DATABASE_URL}/postgres -c "drop database ab;"
                psql ${DATABASE_URL}/postgres -c "create database ab;"
                psql ${DATABASE_URL}/ab < $to_restore
              else
                echo "NO DB BACKUPS FOUND!"
                exit 1
              fi
{{- end }}
```

Zálohu a obnovu dat demonstruje ukázka v následující podkapitole.

Migrace a seedování se staly součástí nasazení nové aplikační verze pro prostředí ”B”, stejně tak seedování. Migrace a aplikace se sestavily do oddělených obrazů - lze tak pak pokračovat v migračním řetězci pro jakoukoliv manipulaci s databází, aniž by se musela měnit verze aplikace.

5.3 Demonstrace konceptu

Aplikace byla navržena velmi prostě - připojí se do dabáze a vrátí do pole serializované obsahy tabulek *cars* a *motorbikes* - konkrétní výstup závisel na nasazené verzi a stavu databáze.

Proces demonstrace byl automatizován do jediného shell skriptu, aby bylo možné proces pouštět opakovaně a verifikovat výsledky a výstupy.

Scénář testovacího procesu měl za cíl demonstrovat destruktivní i nedestruktivní použití migrací, práci s verzemi a možnost obnovy dat, jak ze zálohy, tak z nedestruktivně přejmenované tabulky:

1. Nasazení stejné verze na obě prostředí, migrace
2. Nasazení nové verze na B, přidání tabulky

3. Destruktivní migrace - odebrání tabulky, aniž by obě prostředí byla kompatibilní
4. Obnova schéma, dat a přechod na předchozí verze
5. Příprava na změnu schéma - nová verze kódu
6. Pouze migrace
7. Migrace obnovující data v rámci databáze
8. Přechod na původní verzi

Nejprve bylo nutné incializovat prostředí samotné:

Výpis kódu 5.3 Inicializace minikube, docker build

```
$ minikube start --driver=docker
$ minikube addons enable ingress
$ eval $(minikube docker-env)
$ ./build_docker.sh
```

V prním kroku byla nasazena pouze datábáze a *batch job*, který měl na starosti zálohování databáze. Ten periodicky jednou denně zálohoval databázi, pro účely ukázky pouze na disk vytvořený v rámci klastru.

Následně byly nasazeny první verze obou prostředí a prostředí "B"automaticky zálohovalo databázi (byť prázdnou) a provedlo první migraci - přidání tabulky *cars*. Následujím log úředstavuje je výstup procesu *node-pg-migrate* a výpis tabulky *pgmigrations*.

Výpis kódu 5.4 Inicializace databáze

```
$ npm run migrate up

> migrate
> node-pg-migrate up

> Migrating files:
> - 1707952914993_create-cars-table
### MIGRATION 1707952914993_create-cars-table (UP) ###
CREATE TABLE "products" (
  "id" serial PRIMARY KEY,
  "cars" text NOT NULL
);
INSERT INTO "public"."pgmigrations" (name, run_on)
VALUES ('1707952914993_create-cars-table', NOW());
```

Výpis kódu 5.5 Výpis migrační tabulky

```
# select * from pgmigrations;
id |                 name                 |           run_on
----+--------------------------------------+----------------------------
1 | 1707952914993_create-cars-table    | 2024-02-15 02:32:34.063331
(1 row)
```

Již s první nasazenou verzí bylo možné otestovat obě prostředí:

Výpis kódu 5.6 cURL prostředí A

```
$ curl ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro"]
$ curl -H "B-header: route" ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro","Honda CBR500R","BMW G310R"]
```

Další verze přidala tabulku *motorbikes* včetně seed dat. Po migraci do tabulky *pgmigrations* přibyl řádek s novou migrací.

Výpis kódu 5.7 Výpis migrační tabulky po druhé migraci

```
# select * from pgmigrations;
id |                 name                 |           run_on
----+--------------------------------------+---------------------------
1 | 1707952914993_create-cars-table      | 2024-02-15 02:32:34.063331
2 | 1707952915000_add-motorbikes-table | 2024-02-15 02:36:43.470327
(2 rows)
```

Dále demontrační skript nasadil verzi s destruktivní migrací. Ta smazala tabulku *cars*, aniž by na to byla kterákoliv verze aplikace připravena. To vyústilo v nedostupnost služby.

Výpis kódu 5.8 Demonstrace nedostupnosti služby

```
Test API / A: curl ‘minikube ip‘
Server error
Test API / B: curl -H "B-header: route" ‘minikube ip‘
Server error
```

Protože byla migrace záměrně destruktivní, nebyla jiná možnost, než obnovit ztracená data.

Výpis kódu 5.9 Nasazovací příkaz obnovující databázi

```
$ helm -n ab-testing upgrade --install --wait --timeout 1m1s \
    --set RESTORE_DATABASE=true \
    --set INSTANCES_A=8 --set VERSION_A=v2 \
    --set INSTANCES_B=2 --set VERSION_B=v2 \
    --set VERSION_MIGRATION=v2 \
    ab-testing-deployment ./helm/
```

Pro přípravu na změnu databázového schéma byla postupně do obou prostředí vydána verze kódu, který již nepovyžaduje tabulku cars. V dalším nasazení pak proběhla samotná migrace, která nedestruktivně přejmenovala tabulku **cars** a umožnila tím případnou obnovu dat, aniž by bylo nutné obnovovat zálohu (pro standardní produkční aplikaci by obnova dat obvykle znamenala ztrátu dat akumulovaných v době od poslední zálohy).

Výpis kódu 5.11 Kód nedestruktivní migrace

```
exports.up = (pgm) => {
  pgm.renameTable('cars_deleted', 'cars');
};
```

Výpis kódu 5.10 Kód nedestruktivní migrace

```
/* eslint-disable camelcase */

exports.shorthands = undefined;

exports.up = (pgm) => {
  pgm.renameTable('cars', 'cars_deleted');
};

exports.down = (pgm) => {
  console.log('No down migrations');
};
```

Díky aplikaci konceptu oddělení verzí a obrazů aplikace a dat nebylo potřeba implementovat potenciálně nebezpečnou migraci dolů - ta byla řešena další migrací (nahoru).

V posledním nasazení se pak aplikace vrátila na původní verzi, která pracovala s oběma tabulkami.

Výpis kódu 5.12 Redukovaná ukázka běhu demonstračního procesu

```
Init database
Release "ab-testing-deployment" does not exist. Installing it now.
Deploy A/v1 B/v1, migrations v1
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro"]
Deploy A/v1 B/v2
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro"]
Test API / B: curl -H "B-header: route" ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro","Honda CBR500R","BMW G310R"]
Deploy breaking destructive A/v2 B/v3, migrations v3
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
Server error
Test API / B: curl -H "B-header: route" ‘minikube ip‘
Server error
Restore database, revert to A/v2 B/v2, migrations v2
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro","Honda CBR500R","BMW G310R"]
Test API / B: curl -H "B-header: route" ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro","Honda CBR500R","BMW G310R"]
Deploy preparation for cars table removal, A/v2 B/v4, migrations v4
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro","Honda CBR500R","BMW G310R"]
Test API / B: curl -H "B-header: route" ‘minikube ip‘
["Honda CBR500R","BMW G310R"]
Unify A/v4 B/v4
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
["Honda CBR500R","BMW G310R"]
Test API / B: curl -H "B-header: route" ‘minikube ip‘
["Honda CBR500R","BMW G310R"]
Apply non-destructive removal migration A/v4 B/v5, migrations v5
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
["Honda CBR500R","BMW G310R"]
Test API / B: curl -H "B-header: route" ‘minikube ip‘
["Honda CBR500R","BMW G310R"]
Unify A/v5 B/v5, migraitions v5
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
["Honda CBR500R","BMW G310R"]
Test API / B: curl -H "B-header: route" ‘minikube ip‘
["Honda CBR500R","BMW G310R"]
Revert database scheme and data A/v5 B/v5 migrations v6
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
["Honda CBR500R","BMW G310R"]
Test API / B: curl -H "B-header: route" ‘minikube ip‘
["Honda CBR500R","BMW G310R"]
Revert app version A/v2 B/v2 migrations v6
Release "ab-testing-deployment" has been upgraded. Happy Helming!
Test API / A: curl ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro","Honda CBR500R","BMW G310R"]
Test API / B: curl -H "B-header: route" ‘minikube ip‘
["Ford Mustang","Chevrolet Camaro","Honda CBR500R","BMW G310R"]
```

5.4 Zhodnocení výsledků implementace a testování

Pro demonstraci nasazení strategie A/B Testing byla navržena jednoduchá aplikace a pro demonstraci problémů a řešení migrování relační databáze byla implementována v šesti verzích.

Strategie samotná byla implementována pro kubernetes klastr. Pro obsluhu datábáze byly implementovány nástroje na zálohu a obnovení dat.

Migrace relační databáze byly implementovány v destruktivním i nedestruktivním módu.

Ukázkový běh nasazení jedenácti nasazovacích cyklů úspěšně demonstroval typické problémy a jejich řešení při nasazování aplikací a migračních skriptů. V rámci testování odstranil a obnovil data pro oba způsoby migrací.

Implementace a test ověřily a potvrdily koncepty navržené v analytické části práce, konkrétně potenciální problémy s destruktivním typem migrací a výhody v separaci verzí kódu aplikace a migrací.

Kapitola 6
Závěr

Cílem této bakalářské práce bylo analyzovat běžné strategie nasazování aplikací a jejich vliv na procesy migrování relačních databází. Na základě těchto požadavků bylo zapotřebí provést rešerši běžně užívaných principů, zhodnotit jejich výhody a nevýhody, a nastínit praktické příklady jejich užití v praxi. K tomuto účelu byla vyvinuta jednoduchá aplikace nasazená v minimalistickém Kubernetes systému, což umožnilo detailní zkoumání a testování vybrané nasazovací strategie.

Během této práce byl proveden průzkum principů migrování relačních databází a výzev s ním spojených. Pro každou nasazovací strategii byl navržen postup pro bezpečnou migraci a případné zotavení. Pro A/B Testing strategii pak byly vypracovány scénaře jak pro nasazení, tak pro zotavení v případě neúspěchu.

Tato strategie byla následně implementována a testována na zmíněné aplikaci. Díky tomuto praktickému testování bylo možné nejen ověřit funkčnost a efektivitu zvolené nasazovací strategie v rámci migrování relační databáze, ale také identifikovat potenciální rizika a problémy a navrhnout řešení.

Výsledkem této práce je analýza nasazovacích strategií aplikací s ohledem na migrování relačních databází, včetně návrhu a testování konkrétního scénáře pro bezpečnou migraci. Tato práce poskytuje přínos k teoretickému i praktickému pochopení problematiky, a nabízí základ jak pro další výzkum, tak pro praktické využití.

Práce na ni navazující by se například mohly věnovat migrování distribuovaných databází, či zkoumat problematiku migrace jiných typů databázových systémů, jako jsou objektově nebo dokumentově orientované.

Bibliografie

1. *Deployment Best Practices* [online]. U.S. Geological Survey, [b.r.] [cit. 2024-02-10]. Dostupné z: https://www.usgs.gov/software-management/deployment-best-practices.
2. *Zlepšujeme naše systémy* [online]. Česká Spořitelna, [b.r.] [cit. 2024-02-10]. Dostupné z: https://www.csas.cz/cs/zpravy-z-banky/2024/02/08/odstavka-09-10-unor-2024.
3. *StarCraft II Maintenance* [online]. Blizzard Entertainment, [b.r.] [cit. 2024-02-10]. Dostupné z: https://us.battle.net/support/en/article/179605.
4. *Deployments* [online]. The Kubernetes Authors, 2023 [cit. 2024-02-10]. Dostupné z: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#recreate-deployment.
5. *How can you update your React Native mobile app without requiring a new download?* [online]. LinkedIn, [b.r.] [cit. 2024-02-10]. Dostupné z: https://www.linkedin.com/advice/0/how-can-you-update-your-react-native-mobile.
6. *Top Browsers Market Share* [online]. similarweb, 2024 [cit. 2024-02-10]. Dostupné z: https://www.similarweb.com/browsers/.
7. *Díky Chromu pracujete s aktuálními nástroji* [online]. Google, [b.r.] [cit. 2024-02-10]. Dostupné z: https://www.google.com/chrome/update/.
8. *Operating System Market Share Worldwide* [online]. StatCounter, 2024 [cit. 2024-02-10]. Dostupné z: https://gs.statcounter.com/os-market-share.
9. *Mobile Operating System Market Share Worldwide* [online]. StatCounter, 2024 [cit. 2024-02-10]. Dostupné z: https://gs.statcounter.com/os-market-share/mobile/worldwide.
10. *Create a new version* [online]. Apple, [b.r.] [cit. 2024-02-10]. Dostupné z: https://developer.apple.com/help/app-store-connect/update-your-app/create-a-new-version/.
11. *Prompt users to update to your latest app version* [online]. Google, [b.r.] [cit. 2024-02-10]. Dostupné z: https://support.google.com/googleplay/android-developer/answer/13812041.
12. *Rolling update* [online]. Amazon Web Services, [b.r.] [cit. 2024-02-10]. Dostupné z: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/deployment-type-ecs.html.
13. *Performing a Rolling Update* [online]. Google Cloud Platform, [b.r.] [cit. 2024-02-10]. Dostupné z: https://cloud.google.com/kubernetes-engine/docs/how-to/updating-apps.
14. *Performing a Rolling Update* [online]. The Kubernetes Authors, [b.r.] [cit. 2024-02-10]. Dostupné z: https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/.

15. *AWS Elastic Beanstalk* [online]. Amazon Web Services, [b.r.] [cit. 2024-02-11]. Dostupné z: https://aws.amazon.com/elasticbeanstalk/.
16. *Amazon Elastic Container Service* [online]. Amazon Web Services, [b.r.] [cit. 2024-02-11]. Dostupné z: https://aws.amazon.com/ecs/.
17. *App Engine* [online]. Google Cloud Platform, [b.r.] [cit. 2024-02-11]. Dostupné z: https://cloud.google.com/appengine.
18. *How Heroku Works* [online]. Heroku, [b.r.] [cit. 2024-02-11]. Dostupné z: https://devcenter.heroku.com/articles/how-heroku-works.
19. *Blue/Green Deployments* [online]. AWS, 2024 [cit. 2024-02-15]. Dostupné z: https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/bluegreen-deployments.html.
20. *What is blue green deployment?* [online]. RedHat, 2019 [cit. 2024-02-14]. Dostupné z: https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment.
21. *Blue-Green Deployments Are Built for the Cloud* [online]. Tammy Xu, 2022 [cit. 2024-02-14]. Dostupné z: https://builtin.com/software-engineering-perspectives/blue-green-deployment.
22. *What is A/B Testing? The Complete Guide: From Beginner to Pro* [online]. Alex Birkett, 2023 [cit. 2024-02-15]. Dostupné z: https://cxl.com/blog/ab-testing-guide/.
23. *WHAT IS A/B TESTING & WHAT IS IT USED FOR?* [online]. Anna Vallee, 2016 [cit. 2024-02-15]. Dostupné z: https://online.hbs.edu/blog/post/what-is-ab-testing.
24. *The Ultimate Guide to A/B Testing* [online]. Kitakabee, 2023 [cit. 2024-02-15]. Dostupné z: https://www.browserstack.com/guide/a-b-testing.
25. *What is a Shadow Deployment?* [online]. GILAD DAVID MAAYAN, 2023 [cit. 2024-02-15]. Dostupné z: https://devops.com/what-is-a-shadow-deployment/.
26. *DB-Engines Ranking - Trend Popularity* [online]. Solid IT, 2024 [cit. 2024-02-12]. Dostupné z: https://db-engines.com/en/ranking.
27. *'What is a Relational Database (RDBMS)?'* [online]. Oracle, [b.r.] [cit. 2024-02-12]. Dostupné z: https://www.oracle.com/database/what-is-a-relational-database/.
28. *Relational Database* [online]. Wikipedia contributors, 2024 [cit. 2024-02-12]. Dostupné z: https://en.wikipedia.org/wiki/Relational_database.
29. *How to migrate a database schema at scale* [online]. Aleberto Gimeno, 2020 [cit. 2024-02-12]. Dostupné z: https://blog.logrocket.com/how-to-migrate-a-database-schema-at-scale/.
30. *Welcome to the Liquibase Community* [online]. Liquibase, 2024 [cit. 2024-02-14]. Dostupné z: https://www.liquibase.org/.
31. *GitHub - liquibase/liquibase - Main Liquibase Source* [online]. Liquibase, 2024 [cit. 2024-02-14]. Dostupné z: https://github.com/liquibase/liquibase.
32. *Spring Boot* [online]. VMware Tanzu, 2024 [cit. 2024-02-14]. Dostupné z: https://spring.io/projects/spring-boot.
33. *Official Liquibase Docker images* [online]. Liquibase, 2024 [cit. 2024-02-14]. Dostupné z: https://hub.docker.com/r/liquibase/liquibase.
34. *Flyway* [online]. Red Gate Software, 2024 [cit. 2024-02-14]. Dostupné z: https://flywaydb.org/.
35. *Database Migrations with Flyway* [online]. baeldung, 2024 [cit. 2024-02-14]. Dostupné z: https://www.baeldung.com/database-migrations-with-flyway.

36. *Sequelize* [online]. Sequelize Contributors, 2024 [cit. 2024-02-14]. Dostupné z: `https://sequelize.org/`.

37. *node-pg-migrate* [online]. Salsita, 2022 [cit. 2024-02-14]. Dostupné z: `https://github.com/salsita/node-pg-migrate`.

38. *minicube* [online]. The Kubernetes Authors, 2024 [cit. 2024-02-14]. Dostupné z: `https://minikube.sigs.k8s.io/docs/`.

39. *The package manager for Kubernetes* [online]. Helm Authors, 2024 [cit. 2024-02-14]. Dostupné z: `https://heml.sh`.

Obsah příloh