ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta elektrotechnická

Katedra počítačů

Vytvoření aplikace pro účely demonstrace výhod a nevýhod použití microservices

Implementation of an application demonstrating microservices advantages and disadvantages

Bakalářská práce

Studijní program: Softwarové inženýrství a technologie

Vedoucí práce: Ing. Pavel Náplava

Vlastimil Krahulec

Praha 2018

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Krahulec** Jméno: **Vlastimil** Osobní číslo: **397776**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Vytvoření aplikace pro účely demonstrace výhod a nevýhod použití microservices

Název bakalářské práce anglicky:

Implementation of an application demonstrating microservices advantages and disadvantages

Pokyny pro vypracování:

Vytvořte aplikaci, na které bude možné demonstrovat výhody a nevýhody implementace aplikací na architektuře microservices. Smyslem aplikace je poukázat na aspekty, které s použitím microservices souvisí (nároky na vývoj, problémy synchronizace, odolnost vůči výpadkům atd.) a které jsou zásadní pro rozhodování o způsobu implementace. Postupujte následovně:

Seznam doporučené literatury:

-Newman, S; Building Microservices. Published by O?Reilly Media, Inc.
-Rejesh, RV.; Spring Microservices. Build scalable microservices with Spring, Docker, and Mesos, 2016, Packt Publishing, ISBN 978-1-78646-668-6
-Murray, N., Coury, F., Lerner, A. and Taborda, C., The Complete Guide to Angular, 2017 Fullstack.io"

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Pavel Náplava, katedra ekonomiky, manažerství a humanitních věd FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **19.02.2018** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **30.09.2019**

Ing. Pavel Náplava podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání Podpis studenta

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

Nemám námitky proti použití tohoto školního díla ve smyslu § 60 zákona č. 121/2000 Sb., o autorských právech a právech souvisejících, ve smyslu pozdějších znění tohoto zákona.

V Praze dne …… ……

(podpis autora)

Poděkování

Rád bych poděkoval panu Ing. Pavlu Náplavovi za vedení mé bakalářské práce, za jeho profesionální přístup a cenné rady. Dále bych rád poděkoval svému kolegovi a oponentovi Mgr. Tomáši Slavíkovi, za jeho cenné rady z oblasti vývoje microservices.

Abstrakt

Cílem bakalářské práce je vytvořit ukázkovou aplikaci pro demonstraci výhod a nevýhod architektury microservices. První část bakalářské práce je zaměřena na architekturu microservices, co to microservice je a jakým způsobem se aplikace na architektuře microservices implementuje. V druhé části jsou popsány vlastnosti dvou aplikací z korporátního prostředí, které jsou v současné době implementovány monoliticky. Třetí část bakalářské práce je zaměřena na popis vývoje backendu a frontendu ukázkové aplikace na architektuře microservices. Třetí část také obsahuje uživatelské scénáře, na kterých je demonstrována výhoda microservices. Závěrečná část se zabývá výhodami a nevýhodami architektury microservices včetně vyhodnocení náročnosti samotného vývoje.

Abstract

The aim of the bachelor thesis is to create a sample application to demonstrate the advantages and disadvantages of the microservices architecture. The first part of the bachelor thesis is focused on microservices architecture, what microservices is and how the application for the microservices architecture is implemented. The second part describes the properties of two corporate applications that are currently being implemented monolithically. The third part of the bachelor thesis is focused on description of backend and frontend development of sample application for microservices architecture. The third part includes user scenarios where the advantage of microservices is demonstrated. The last part deals with the advantages and disadvantages of the microservices architecture, including the evaluation of the complexity of the development itself.

Klíčová slova

Microservices architektura, pojem microservice, SOLID, vlastnosti microservices, automatizace microservices, vývoj microservices, monolitická architektura, implementace ukázkové aplikace, backend, Spring, maven, PostgreSql, Message broker, frontend, Angular 5, PrimeNg, uživatelské scénáře, výhody microservices, nevýhody microservices

Key words

Microservices architecture, concept microserices, SOLID, behavior microservices, automation microservices, development microservices, monolithic architecture, implementation sample application, backent, Spring, maven, PostgreSql, Message broker, frontend, Angular 5, PrimeNg, user scenarios, advantages microservices, disadvantages microservices

Obsah

Seznam zkratek

N-tier	…	vícevrstvá aplikace
SOLID	…	objektově orientovaný design
HTTP	…	Hypertext Transfer Protocol, internetový protokol
REST	…	Representational State Transfer
JEE	…	Java Enterprise Edition
WAR	…	Web Application Resource, formát souboru
EAR	…	Enterprise Application Archive, formát souboru
GUI	…	Graphical User Interface, grafické uživatelské rozhraní
UML	…	Unified Modeling Language, grafický jazyk pro vizualizaci

Seznam obrázků

Seznam tabulek

1 Úvod

Tématem bakalářské práce je „*Vytvoření aplikace pro účely demonstrace výhod a nevýhod použití microservices*. Toto téma jsem si zvolil z několika důvodů. Prvním z nich je snaha vyvinout si vlastní aplikací, protože jsem se s vývojem na architektuře microservices setkal již v praxi. Druhým důvodem je možnost rozšířit si znalosti o problematice microservices jako takové. Dále se jedná o můj zájem o danou problematiku všeobecně.

Celá bakalářská práce je rozdělena do čtyř kapitol. V první kapitole bakalářské práce je definován pojem microservices a také principy, které je vhodné při implementaci aplikace na architektuře microservices dodržovat. První kapitola je doplněna o vlastnosti microservices a jakým způsobem vyvíjet microservices v jazyce Java spolu s knihovnou Spring Boot.

Aplikace postavené na architektuře monolit jsou zmíněné ve druhé kapitole bakalářské práce. Jedná se o aplikace z korporátního prostředí, které jsou denně používány a spravovány. Jsou zde popsány jejich vlastnosti, technologie a zobrazeny diagramy nasazení.

Třetí kapitola je zaměřena na praktickou část bakalářské práce, kterou je samotná implementace ukázkové aplikace. Pro implementaci je vybrána aplikace ze třetí kapitoly, která je rozdělena na čtyři microservices. Implementace ukázkové aplikace je rozdělen na backend a frontend. Backend aplikace je vyvinut v jazyce Java spolu s knihovoun Spring Boot. Pro ukládání dat je použita databáze PostgreSql. Jednotlivé microservices jsou kompilovány nástrojem Maven. Komunikace mezi microservices je řešena nástrojem Message Broker. Frontend ukázkové aplikace je vyvinut za pomoci Angularu 2 s knihovnou UI (user interface) komponent PrimeNg. Třetí kapitola je doplněna o uživatelské scénáře, na kterých je demonstrována výhoda microservices.

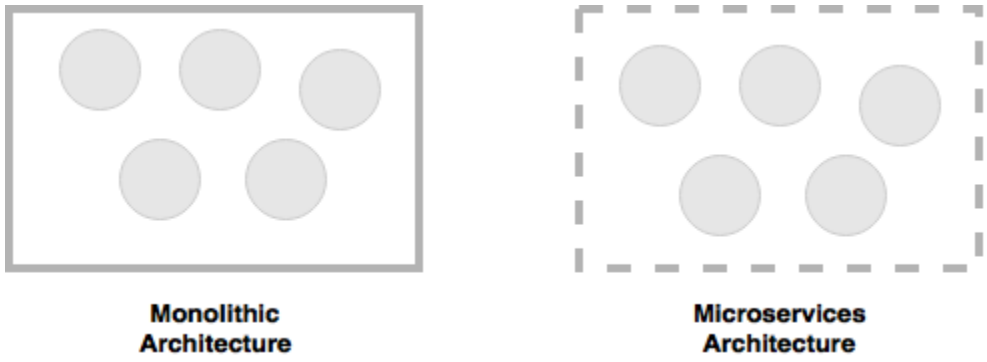
V poslední kapitole jsou popsány výhody a nevýhody microservices spolu s vyhodnocením náročnosti implementace na architektuře microservices.

2 Implementace na architektuře microservices

2.1 Pojem microservices

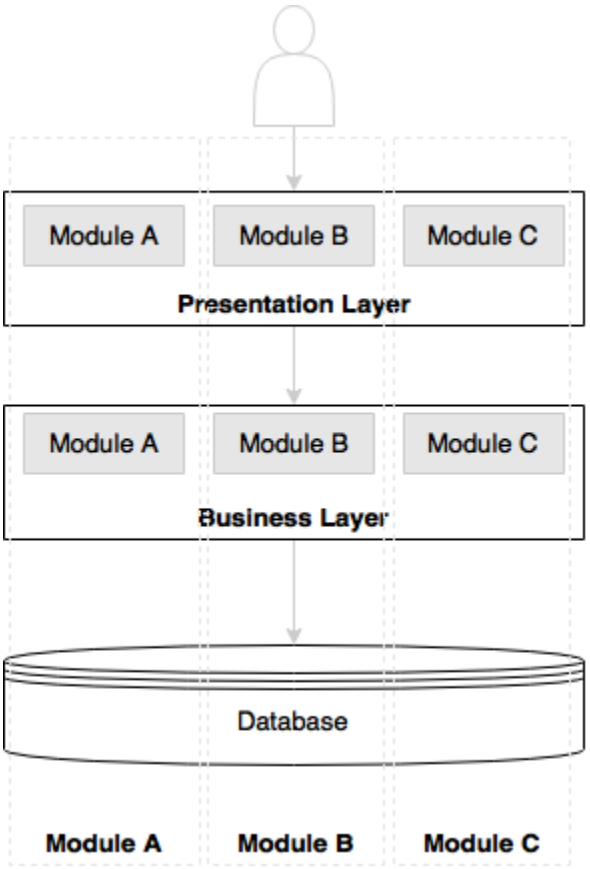
Co to je microservices? Microservices je architektonický styl pro vývoj softwaru. Jedná se o variantu servisně orientované architektury (SOA), která strukturuje aplikace do kolekce služeb volně spojených mezi sebou. Microservices poskytují způsob, jak vyvíjet více fyzicky oddělených aplikací. Architektura microservices umožňuje stálé dodávky komplexních aplikací.

Microservices pomáhají překonat hranice monolitických aplikací a stavbu logicky nezávislého menšího systému tvořeného menšími systémy, tak jak je znázorněno na následujícím **obrázku č. 2. 1**.



Obrázek 2. 1 Znázornění monolitické architektury a microservices architektury [1]

Architektura microservices není žádnou inovací. Mnoho organizací, jako jsou například Netflix nebo Amazon, již delší dobu pro své monolitické aplikace úspěšně používají techniku „devide and conquer“ (rozděl a panuj). Aplikace je rozdělena do menších jednotek, z nichž každá provádí vlastní funkci.

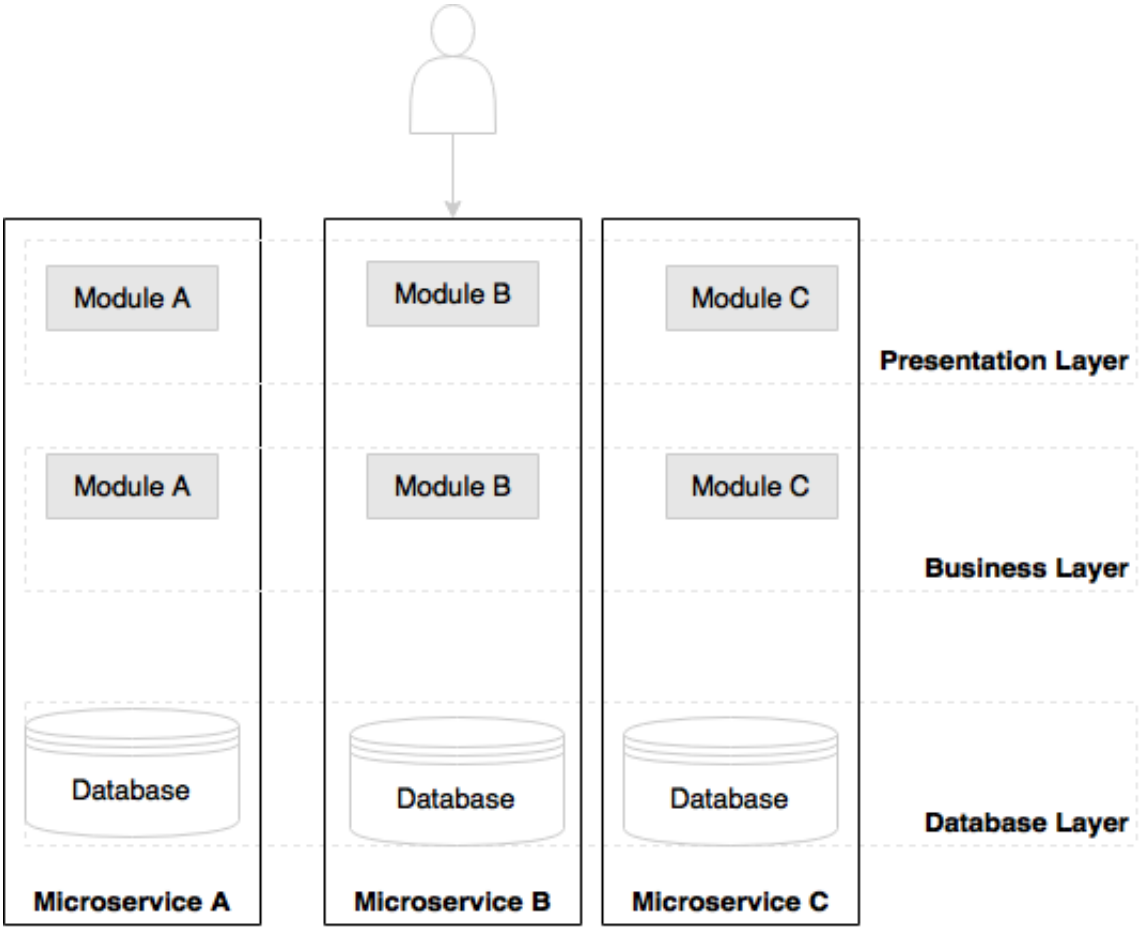


Obrázek 2. 2 Schéma monolitické aplikace [2]

Na předchozím **obrázku č. 2. 2** je zachycené schéma zobrazující tradiční N-tier (více vrstvá) aplikaci rozdělenou na prezentační vrstvu, byznys vrstvu a databázovou vrstvu. Jednotlivé moduly A, B, C reprezentují tři rozdílné byznys domény. Každá vrstva obsahuje všechny tři byznys domény. Prezentační vrstva má webové komponenty ze všech tří modulů, byznys vrstva obsahuje logiku všech tří modulů a databázová vrstva obsahuje všechny databázové tabulky modulů, které jsou propojeny mezi sebou. Ve většině případů jsou vrstvy fyzicky rozprostřeny, zatímco jednotlivé moduly uvnitř vrstev jsou pevně propojeny.

Na druhou stranu, jak je zobrazeno na schématu **obrázku č. 2. 3**, má každá microservice vlastní prezentační vrstvu, byznys vrstvu a databázovou vrstvu. Zároveň se jednotlivé microservices neovlivňují a nejsou mezi sebou nijak propojeny.

Pro komunikaci mezi microservices neexistuje žádný standartní mechanismus, ale obecně platí, že microservices pro komunikaci mezi sebou využívají široce používané protokoly, jako například HTTP, REST nebo protokoly pro zasílání zpráv (JMS, AMPQ).



Obrázek 2. 3 Schéma microservices aplikace [3]

2.2 Principy microservices

Při návrhů a následné implementaci microservices je nutno dodržovat určité zásady (principy).

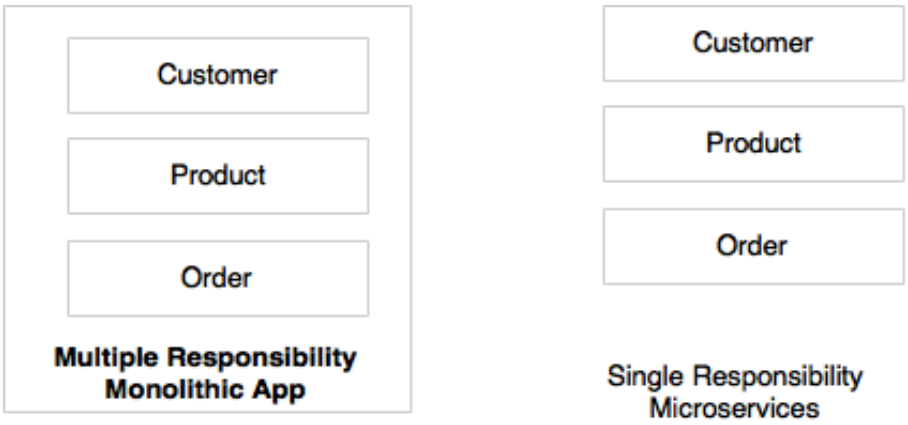
2.2.1 Odpovědnost

Jedním z důležitých principů pro implementaci microservices je, že každá microservice by měla mít pouze jednu odpovědnost za právě jednu věc. Jedná se o tzv. „Single responsibility“ princip, který je součástí návrhového vzoru SOLID.

SOLID je zkratkou pro pět návrhových principů, jejichž cílem je učinit návrhy softwaru srozumitelnějšími, flexibilnějšími a udržitelnějšími.

Princip jedné odpovědnosti (Single Responsibility Principe – SRP)

„Třídy by měly mít jedinou odpovědnost / jediný důvod ke změně. Každá třída by měla mít odpovědnost za právě jednu věc, která by měla být jasně vystižena jejím názvem. “ [9]



Obrázek 2. 4 Responsibility [4]

Princip odpovědnosti je zobrazen na **obrázku č. 2. 4**, kde je diagram s třemi různými byznys moduly. Z obrázku je patrné, že než budovat všechny služby do jedné aplikace, je lepší mít tři různé služby, kde každá z nich je odpovědná za vlastní funkcionalitu a změna odpovědnosti jedné neovlivní chování ostatních.

2.2.2 Samostatnost

Dalším principem microservices je samostatnost a nezávislost. Jinými slovy, microservices mají v sobě všechny knihovní závislosti, webové servery, kontejnery nebo virtuální stroje. Jsou to vlastně nezávisle nasazené služby.

Při vývoji aplikace na architektuře SOA je výsledek zabalen do archivu WAR nebo EAR, který je poté nasazen do JEE aplikačního serveru (JBoss, WebLogic). Microservices mají jiný přístup. Výsledná aplikace je zabalena do archivu JAR, který obsahuje všechny závislosti (knihovny, aplikační server) a poté je aplikace spuštěna jako samostatný proces Java.

Microservices mohou být vloženy do vlastního kontejneru. Kontejnery jsou přenosné (lze je nasadit na jiné prostředí), nezávisle ovladatelné a mají jednoduché prostředí pro nasazení. Kontejnerová technologie jako je například Docker je ideálním řešením pro deployment microservices.

2.3 Vlastnosti microservices

V této části popíši vlastnosti microservices, které jsou podstatné, ale nikoliv nutné pro jejich implementaci. Jedná se o zásady, které by měl vývojář chápat, než aby je znal pouze teoreticky.

2.3.1 Charakteristika služeb v microservices

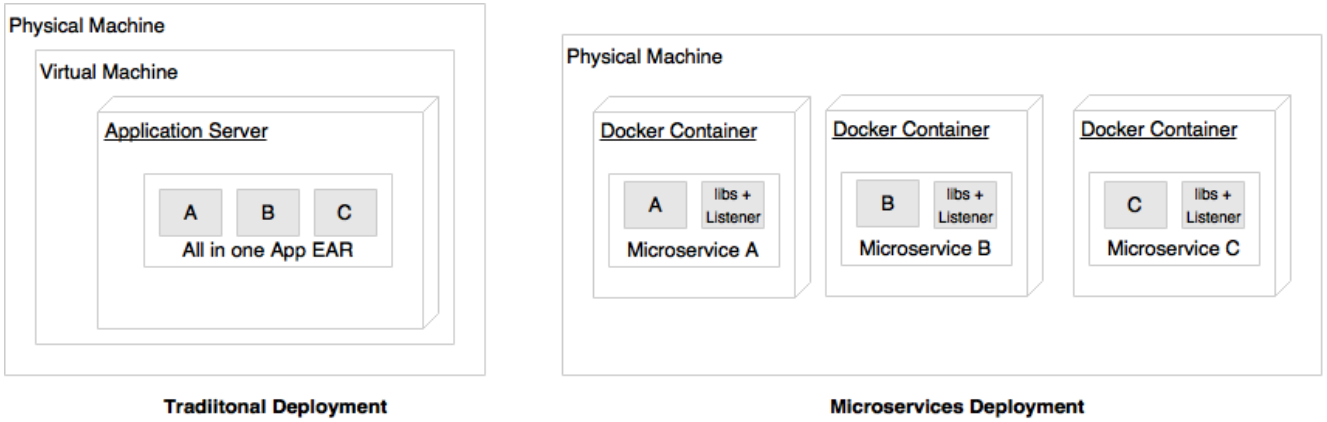
Architektura microservices je podobná architektuře SOA, proto mnoho charakteristik služeb (services) definovaných v architektuře SOA jsou použitelné i pro služby v architektuře microservices.

- **Poskytování služeb (Service contract):** V microservices se pro poskytování a popis služeb používá schéma JSON a rozhraní REST. Pro dokumentaci služeb se například používá framework Swagger.
- **Nízká provázanost (Loose coupling)** Služby microservices jsou nezávislé a volně provázané. Jediné propojení mezi microservices je za pomocí zpráv (tzv. messaging) a rozhraní REST přes protokol http.
- **Abstrakce (Service abstraction):** Microservices by měli poskytovat abstrakci služeb, knihoven a prostředí (environment).
- **Opětovné použití (Service reuse):** Služby microservices jsou opakovaně použitelné. Lze k nim přistupovat z mobilních zařízení nebo jiných systémů
- **Bezstavovost (Statelessness):** Microservices nesdílí nic s žádným sdíleným stavem. Pokud je potřeba udržet stav, je možné ho uchovat v databázi nebo paměti.
- **Spolupráce (Service interoperability):** Služby v microservices si mezi sebou předávají zprávy pomocí rozhraní REST. V komplikovanějších případech je pro přenos zpráv použito protokolů využívající paměti. Například Rabbit MQ, Avro, Zero MQ.
- **Slučitelnost (Service composeability):** Microservices jsou kompozitní. Kompozice lze dosáhnout pomocí služeb orchestrace.

2.3.2 Lightweight microservices

Každá implementovaná microservice má být spjata pouze s jednou byznys doménou tak, aby prováděla pouze jednu funkci.

Aplikační servery, na které se microservices nasazují, by měly být také lightweight. Například Jetty nebo Tomcat jsou lepší volbou než WebLogic.



Obrázek 2. 5 Lightweight microservices [5]

Diagram na **obrázku č. 2. 5** zobrazuje microservices nasazené v kontejnerech Dockeru. Každý kontejner představuje jednu byznys doménu a zároveň obsahuje logiku s veškerými knihovnami. Díky tomu je jednoduché přenést celý kontejner do jiného prostředí. Protože neexistuje závislost na fyzické infrastruktuře, jsou microservices v kontejnerech Dockeru snadno přenositelné.

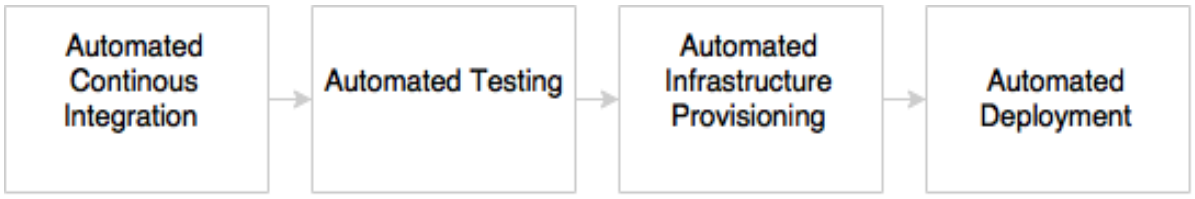
2.3.3 Microservices s vícejazyčnou architekturou

Jelikož jsou microservices autonomní (samostatné kapitola 2.2.2) a abstraktní, tak vše, co je za API, může mít různé architektury pro různé služby.

- Různé služby používají různé verze stejných technologií. Jedna microservice může být napsána na platformě jazyka Java 1.7 a další v Java 1.8.
- Různé programovací jazyky se používají k vývoji rozdílných microservices. Například jedna microservice je vyvinuta v jazyku Java a další v jazyku Scala.
- Používají se rozdílné databáze pro ukládání dat. Jedna microservice může ukládat data do MySql databáze druhá do PostgreSql a třetí do NoSql databáze (MongoDb).

2.3.4 Automatizace microservices

Velký počet microservices je obtížné spravovat, proto je vhodné když je zavedena automatizace životního cyklu microservices. Jedná se o automatizovaný build, testy, nasazení a škálovatelnost.



Obrázek 2. 6 Automatizace microservices [6]

Na **obrázku č. 2. 6** je znázorněné schéma automatizace microservices, které se obvykle používá během fáze vývoje, testování, releasu a nasazení.

- Fáze vývoje je automatizovaná pomocí nástrojů na správu verzí, jako je například Git nebo pomocí nástrojů pro integraci jako jsou Jenkins, Travis CI. Tyto nástroje v sobě zahrnují kontrolu kvality kódu a spouštění automatických testů.
- Automatizace infrastruktury se provádí za pomocí již zmíněné kontejnerové technologie Docker. Pro automatizované nasazení se používají nástroje typu Spring Cloud, Kubernetes nebo Mesos.

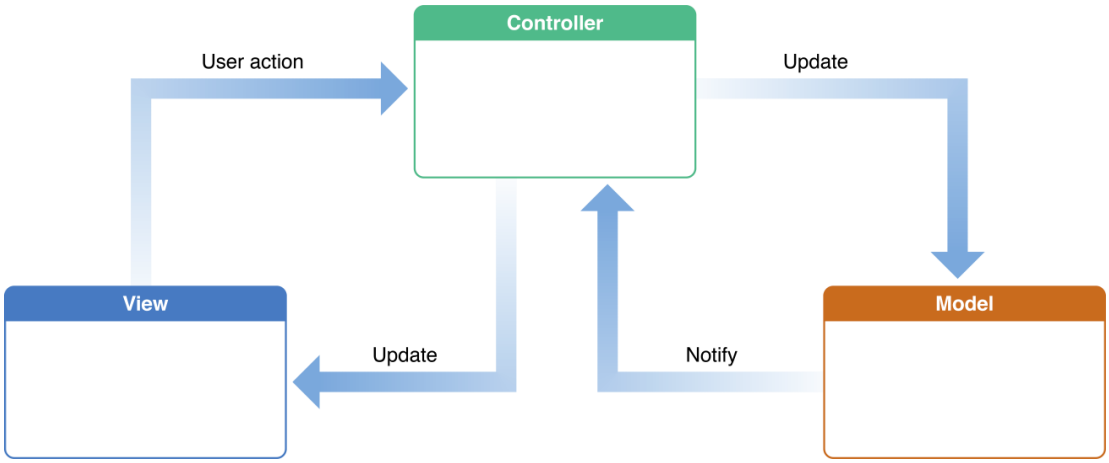
2.4 Vývoj microservices

Jak jsem již popsal v podkapitole 2. 3. 3, microservices lze vyvíjet v jakémkoliv objektově orientovaném jazyce. K samotnému vývoji nestačí pouze daný jazyk (Java, C#), ale je potřeba mít i flexibilní framework. V jazyce Java je to Spring Boot Framework. Tento framework velice dobře pomáhá programátorovi při vývoji microservices a zároveň je v souladu se zásadami a charakteristikami uvedenými v předchozích kapitolách.

Pro vývoj microservices v jazyce Java je nutné mít následující komponenty:

- Knihovnu jazyka Java JDK minimálně ve verzi 1.8
- Nástroj pro buildování Maven nebo Gradle
- Spring Boot framework

Vývoj microservices pomocí frameworku Spring Boot je podobný vývoji jakékoliv enterprise aplikace, která využívá návrhový vzor MVC (Model – View - Controller) **obrázek č. 2. 7**. Model obsahuje persistenci (datový model) a byznys logiku (výpočty, pravidla). View je reprezentace dat z modelu, v případě microservices, se jedná o JSON schéma, které je vystavováno přes REST Api. Controller aktualizuje model a vystavuje REST Api.



Obrázek 2. 7 MVC vzor[1] [10]

Spring Boot Framework používá jeden přístup ke konfiguracím, nejčastěji se jedná o soubory typu yml nebo properties. Tento přístup ulehčí vývoj a sníží se úsilí psaní velkého množství kódu. Při psaní aplikace za pomoci Spring Bootu není nutné používat XML konfigurace, ale stačí využít springové anotace. Spring Boot sám rozezná potřebnou konfiguraci ze souboru POM od Mavenu.

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <scope>runtime</scope>
</dependency>
```

Například z předchozího příkladu maven konfigurace Spring Boot pozná, že projekt je nastaven na databázi Spring Data JPA a HSQL. Další vlastností Spring Bootu je, že umožňuje při vývoji microservices, zabalit celou službu se všemi knihovnami a aplikačním serverem do spustitelného Java Jar souboru.

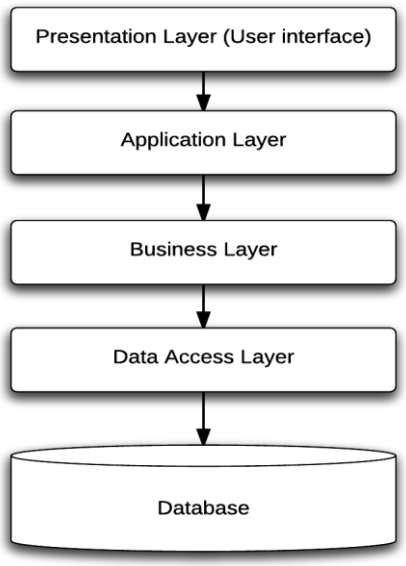
[1] Twilio blog: https://www.twilio.com/blog/2015/11/getting-started-with-spring-mvc-and-twilio.html

3 Aplikace implementovány monoliticky

V této části bakalářské práce popíšu aplikace, se kterými jsem se setkal v oblasti bankovního korporátního sektoru. Aplikace, které popíši, jsou v současné době implementovány a postaveny na architektuře monolit. Architektura monolit je postavena jako jeden celek a využívá servisně orientovanou architekturu (SOA), která je rozdělena na tři vrstvy. Prezentační vrstvu, byznys vrstvu a databázovou vrstvu **obrázek č. 3. 1**.

Každá vrstva monolitické aplikace obsahuje všechny byznys moduly. Prezentační vrstva má webové komponenty (HTML, JavaScript, Css, JSF, JSP) ze všech modulů, byznys vrstva obsahuje logiku všech modulů a je psaná pouze jednou technologií. Takže pokud je logika aplikace psaná pomocí jazyka Java a frameworku Spring, tak i celá byznys vrstva musí být napsaná stejnou technologií. Databázová vrstva obsahuje všechny tabulky modulů, které jsou vzájemně propojeny.

Změna v kterémkoliv modulu monolitické aplikace ovlivňuje vlastnosti ostatních modulů. Monolitická aplikace je sestavena (zabalena) jako jediný logicky spustitelný soubor Ear nebo War, který je následně nasazen na aplikační server (např. Tomcat). Pokud je v aplikace potřeba provést změny nebo opravit chyby, je nutné aktualizovat celou aplikace a znovu nasadit na server.



Obrázek 3. 1 Vrstvy monolitické architektury[2] [11]

[2] DZone: https://dzone.com/articles/splitting-a-monolith-application-into-services

3.1 Aplikace evidence objednávek (EVO)

První aplikace, kterou popíši, je aplikace pro evidenci objednávek (dále jen Evo). Tato aplikace je v současné době implementovaná jako monolit. Aplikaci Evo využívají interní zaměstnanci a oddělení lidských zdrojů. Tato aplikace usnadňuje práci s evidencí a nabízí přehled jednotlivých objednávek zaměstnancům lidských zdrojů (HR).

3.1.1 Popis aplikace EVO

Jak jsem již napsal v prvním odstavci, aplikace Evo se používá pro evidenci interních objednávek. Přes tuto aplikace lze objednat externí vzdělávací akce – kurzy, školení semináře, konference a workshopy. Dále lze přes aplikaci objednat licence, odbornou literaturu apod.

Zaměstnanec se rozhodne, že se zúčastní školení, které potřebuje ke svému rozvoji. V aplikaci Evo, založí novou objednávku, do které vloží důležité údaje o školení jméno, datum, čas, místo, webovou stránku a vybere dodavatele školení. Dále vloží účastníky, kteří se chtějí školení zúčastnit. Nakonec vloží cenu školení. Objednávku uloží a ta je dále zpracovaná. O každém stavu objednávky je účastník informován emailem.

Důležité části, z kterých se aplikace skládá:

Objednávka

Objednávka může být například objednání kvalifikačního kurzu (jazykový kurz, technický kurz, soft skills), školení, semináře, konference, workshopu nebo licence. Každá objednávka obsahuje dodavatele, uživatele a cenu. Objednávka může být s účastníky nebo bez účastníků. Pokud je objednávka bez účastníků, objednává se licence nebo odborná literatura.

Dodavatel

Dodavatel je fyzická nebo právnická osoba, která dodává odběrateli zboží (licence, odborná literatura) nebo služby (kurzy, školení).

Účastník

Zaměstnanec korporace, který se účastní kvalifikačního kurzu, školení, semináře, nebo workshopu, pokud se jedná o objednávku bez účastníků, není vybrán žádný zaměstnanec.

Rozpočet

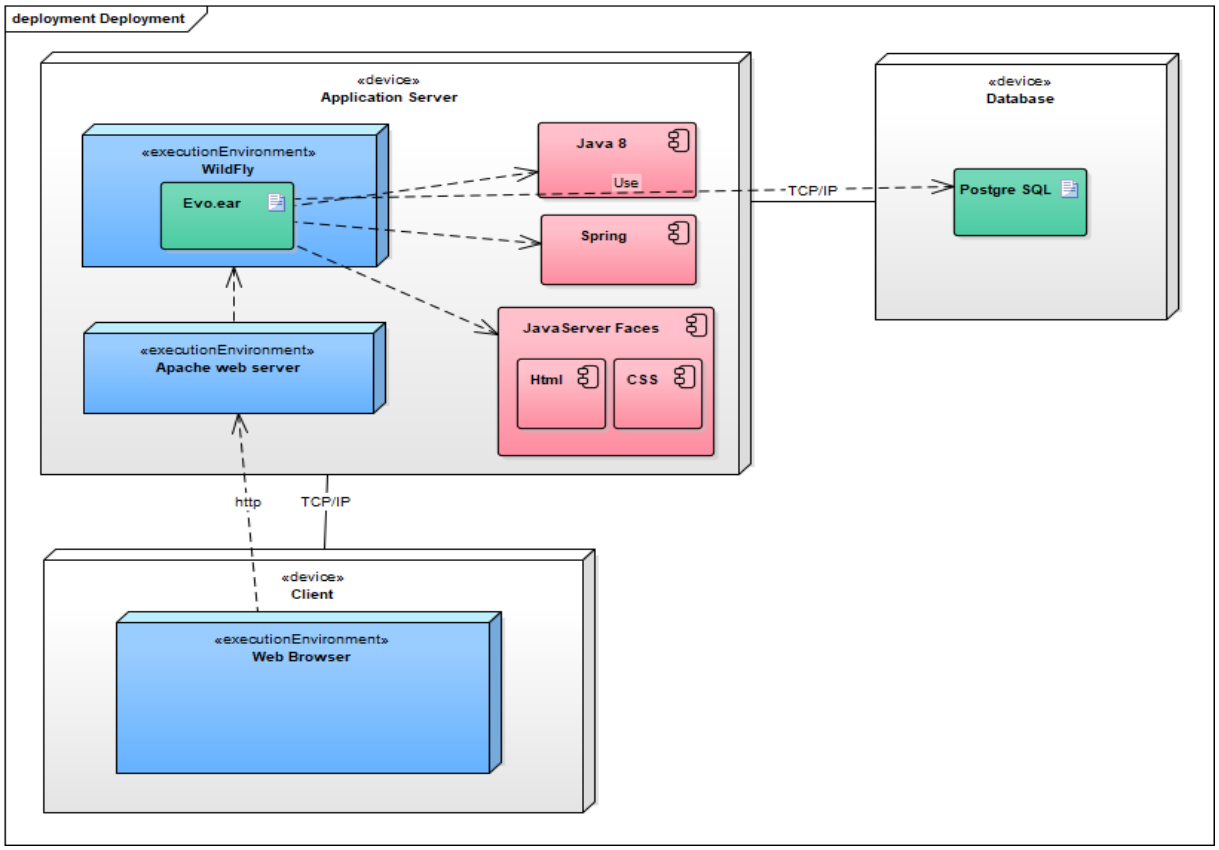
Finanční tok, z kterého se objednávka zaplatí.

Email

Notifikační emaily, které jsou zasílány účastníkům a pracovníkům HR oddělení. Emaily informují o změně stavu objednávky.

3.1.2 Architektura a technologie EVO

Aplikace Evo, je v současné době postavena na architektuře monolit. Má jednu PostgreSql databázi obsahující schéma pro aplikaci Evo a schéma uživatelů. Backend aplikace je napsán v jazyce Java 8 a frameworku Spring. Frontend aplikace (GUI) je implementován v JSF (JavaServer Faces). Pomocí nástroje Maven, je celá aplikace zabalena do archivu EAR a nasazena na aplikačním serveru. Pokud je potřeba do aplikace přidat novou vlastnost nebo opravit chybu, ať už na straně frontendu nebo backandu musí se celá aplikace znovu zabalit do archivu EAR a nasadit na aplikační server. Diagram nasazení aplikace EVO je na **obrázku č. 3. 2**.



Obrázek 3. 2 Diagram nasazení aplikace Evo

3.2 Aplikace Kaizen

Druhá aplikace, která je v současné době implementována jako monolit, je aplikace evidující náměty na zlepšení pracovních standardů (zkráceně Kaizen) od interních zaměstnanců. Aplikace organizuje práci členů Kaizen skupin a publikuje výkazy o námětech a výsledcích práce členů Kaizen skupin. Vybraní interní zaměstnanci mají role, ve kterých zajišťují celý proces řešení námětu.

3.2.1 Popis aplikace Kaizen

Interní zaměstnanec má nápad na zlepšení pracovního standardu. Námět zadá do aplikace Kaizen. V aplikaci popíše problém a návrh na zlepšení. K námětu je přidělena skupina lidí, která námět na zlepšení řeší. Skupinu vede moderátor, který zadává úkoly. Jednotlivé skupiny se scházejí na meetignu, kde řeší zadané úkoly. K setkáním lze přizvat externího uživatele, který rozumí dané problematice popsané v námětu. Zadání a řešení námětu je zobrazeno na schématu **obrázku č 3. 3**.

Důležité části aplikace:

Námět:

Námět je jakýkoliv návrh, který zlepší pracovní standard. Může to být například pracovní postup, změna pracovního prostředí nebo vylepšení aplikace, s kterou zaměstnanec pracuje. Námět obsahuje popis problému a návrh na zlepšení.

Úkol:

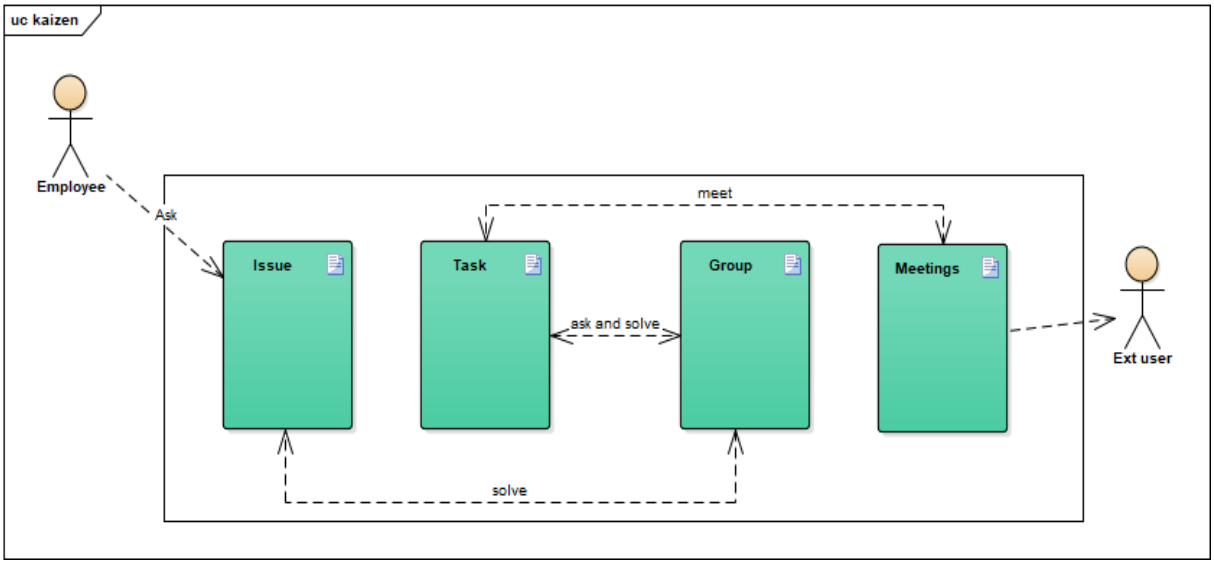
Námět je rozdělen na jednotlivé úkoly, které řeší přidělená skupina lidí. Úkoly rozděluje moderátor skupiny.

Skupina:

Skupina lidí, která řeší přidělené náměty na zlepšení a jednotlivé úlohy. Skupinu vede moderátor.

Meeting:

Schůzky, kde se scházejí jednotlivé skupiny lidí a řeší jednotlivé úkoly. Schůzi vede moderátor skupiny. K meetingu lze přizvat externí osobu, která rozumí dané problematice.

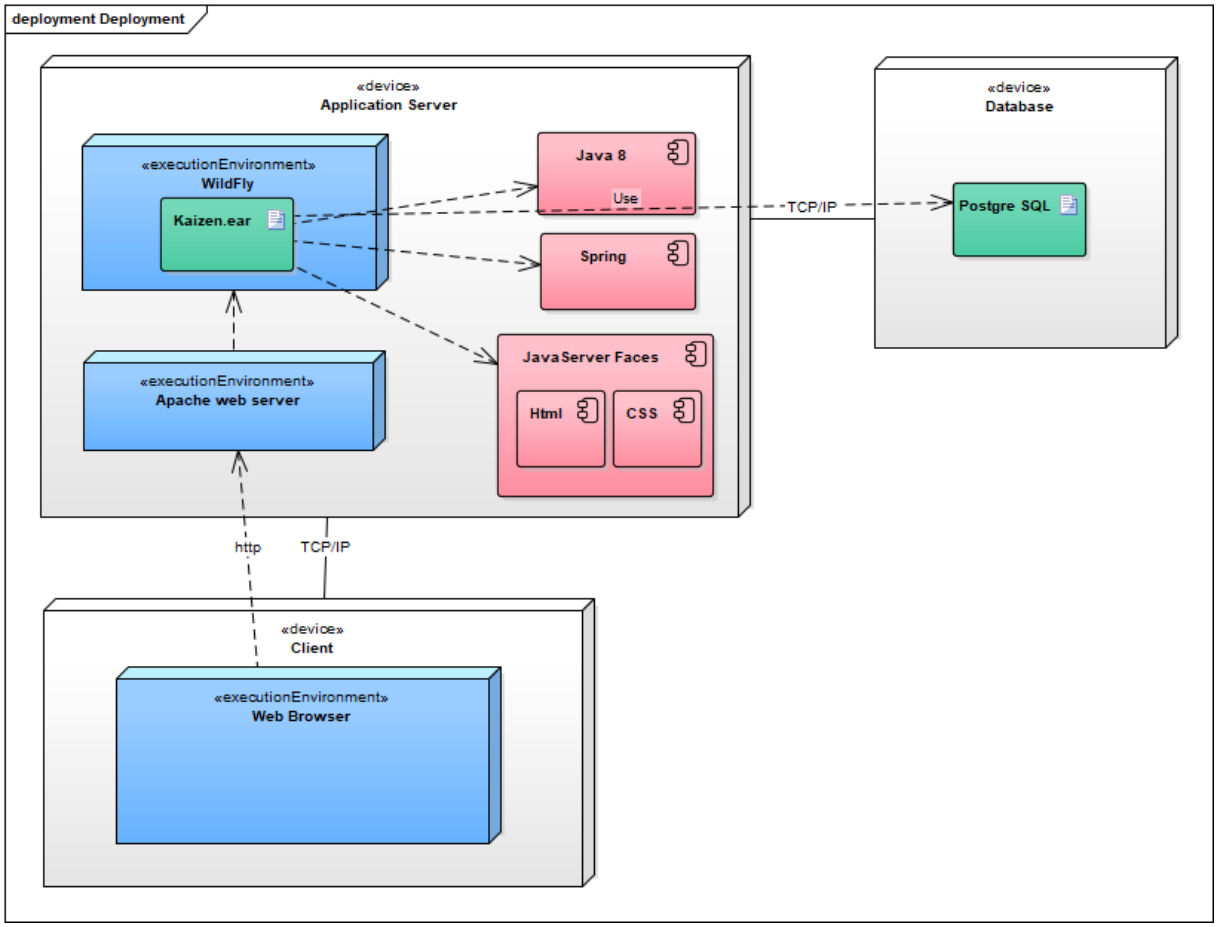


Obrázek 3. 3 Zadání námětu

3.2.2 Architektura a technologie aplikace Kaizen

Aplikace Kaize, je v současné době stejně jako aplikace Evo postavena na architektuře monolit. Má jednu PostgreSql databázi obsahující schéma pro aplikaci Kaizen a schéma uživatelů. Backend aplikace je napsán v jazyce Java 8 a frameworku Spring. Frontend aplikace (GUI) je implementován v JSF technologii (JavaServer Faces). Pomocí nástroje Maven, je celá aplikace zabalena do archivu EAR a nasazena na aplikačním serveru (WildFly).

Pomocí protokolu http komunikuje webový prohlížeč s webovým serverem (Apache) a ten dále komunikuje s aplikací. Pokud je potřeba do aplikace přidat novou vlastnost nebo opravit chybu, ať už na straně frontendu nebo backandu, musí se celá aplikace znovu zabalit do archivu EAR a nasadit na aplikační server. Diagram nasazení aplikace Kaizen je na **obrázku č. 3. 4**.



Obrázek 3. 4 Diagram nasazení aplikace Kaizen

4 Implementace ukázkové aplikace microservices

Pro demonstraci architektury microservices, jsem vybral aplikaci evidence objednávek (EVO), která je postavena na architektuře monolit. Aplikaci jsem rozdělil na 4 microservices (4 byznys domény) Order-service (objednávka), Vendor-service (dodavatel), Users-service (účastníci) a Email-Service.

Důvodem rozdělení na 4 microservices je ten, aby jedna služba (microservice) měla na starosti pouze svoji vlastní byznys doménu. Jinými slovy know how (logika) by měla zůstat pouze v jedné mikro službě. Pokud má microservice na starosti více byznys domén, pak se již nejedná o microservice, ale opět o monolit.

Microsrvice order řeší byznys logiku objednávek. Vendor-service má nastarosti byznys logiku dodavatelů. User-service obstarává byznys logiku uživatelů aplikace a účastníků, kteří jsou přiděleni k objednávce. Email-service odesílá emaily nezávisle na ostatních microservices. Email je odeslán po založení objednávky.

Pro vývoj ukázkové aplikace microservices jsem vybral programovací jazyk Java 8, Spring Boot, databázi PostgreSql nástroj Maven a Angular 5. Angular 5 spolu s knihovou PrimeNg jsem použil pro vývoj forntendové části (GUI).

4.1 Backend

Backend microservices je rozdělen na tři vrstvy. Prezentační vrstvu, byznys vrstvu a databázovou vrstvu. Jedná se o EEA (Enterprise application architecture) architekturu, jejíž schéma je na **obrázku č. 4. 4.** Obrázek zachycuje diagram komponent jedné microservice (Order-Service). Zbylé microservices mají stejný diagram komponent. Výjimku tvoří microservice email, která nemá databázovou vrstvu.

Databázová vrstva obsahuje entity, které jsou za pomoci JPA (Java Persistence API) mapované do relační databáze. Mezi servisní vrstvou a entitami jsou Spring data repository, která mají za úkol přistupovat k jednotlivým objektům v databázi.

```
@Transactional
public interface OrderRepository extends PagingAndSortingRepository<Order,Long> {
}
```

Obrázek 4. 1 Spring data repository

4.1.1 Spring data repository

Spring data repository redukují velké množství kódu, který je potřebný pro implementaci DAO (data access object) vrstvy. Základní rozhraní obsahuje veškeré CRUD operace.

Servisní vrstva předává data controlleru, která vystavují rest api rozhraní pro frontend. Rest API rozhraní posílá data ve formátu JSON přes protokol Http. Než jsou objekty poslány ven ze služby přes rest api rozhraní, je potřeba objekty přemapovat do objektů DTO (Data transfer object). DTO jsou objekty, které nesou data mezi procesy - jedná se o přepravku dat. Pro převod objektů z databáze na DTO objekty je zapotřebí Java knihovna ModelMapper.

```
@RequestMapping(method = RequestMethod.GET)
public ResponseEntity<List<OrderDTO>> getAll(){
    List<Order> orders = new ArrayList<>();
    orderRepository.findAll().forEach(orders::add);
    List<OrderDTO> orderDTOS = modelMapper.map(orders,new TypeToken<List<OrderDTO>>(){
    }.getType());
    orderDTOS.forEach(orderDTO -> orderDTO.setUsersId(orderService.getUsersId(orderDTO.getId())));
    return new ResponseEntity<>(orderDTOS, HttpStatus.OK);
}
```

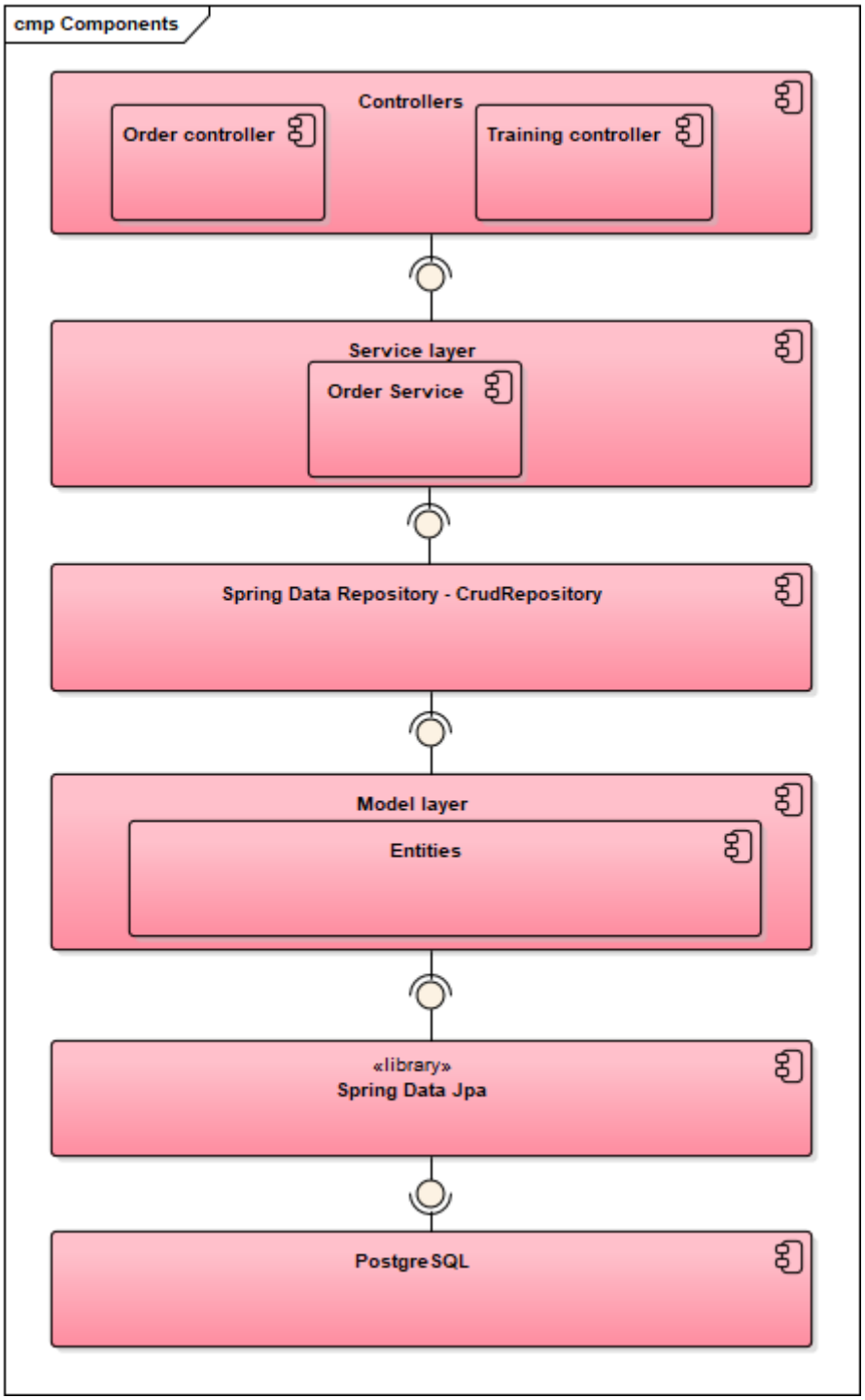
Obrázek 4. 2 Metoda getAll rest contrelleru Order

4.1.2 Model Mapper

Cílem model mapper je usnadnit mapování objektů. Programátorovi stačí určit objekt, který se má mapovat na jiný objekt. Oba objekty musí dodržovat stejnou konvenci. Na **obrázku č. 4. 3** je objekt Order mapován na objekt OrderDTO.

```
OrderDTO orderDTO = modelMapper.map(orderRepository.findOne(id),OrderDTO.class);
```

Obrázek 4. 3 Model mapper



Obrázek 4. 4 Diagram komponent microservice Order service

4.1.3 Databáze

Pro ukládání dat jsem u microservices použil databázi PostgreSQL. Tuto databázi jsem vybral, z důvodu jednoduché instalace, nastavení a podpory Spring Boot s Hibernate.

„PostgreSQL (nebo zkráceně Postgres) je plnohodnotným relačním databázovým systémem s otevřeným zdrojovým kódem. Běží nativně na všech rozšířených operačních systémech včetně Linuxu, UNIXů a Windows. Splňuje podmínky ACID (Atomicita, Konzistence, Izolovanost, Trvalost), plně podporuje cizí klíče, operace JOIN, pohledy, spouště a uložené procedury. Obsahuje většinu SQL92 a SQL99 datových typů, např. INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL a TIMESTAMP. Nechybí ani podpora moderních datových typů jako je JSON nebo XML.“[3] [12]

4.1.4 Maven

„.Maven je rozšířený systém pro správu a sestavování aplikací postavených nad platformou Java. Jeho využitím odpadá závislost na konkrétním IDE, protože všechny informace potřebné ke kompilaci a sestavení programu jsou přímo obsaženy ve speciálních souborech pom.xml.

Principem systému Maven je vytvoření objektového modelu nad zdrojovým kódem, se kterým lze provádět různé operace. Nejčastěji se jedná o kompilaci, kontrolu, vytvoření balíků, atd. Model projektu je definován v souborech pom.xml, které se nachází v kořenovém adresáři každého projektu. V těchto adresářích lze spouštět příkazy mvn s parametry, které načtou model z odpovídajícího souboru pom.xml a provedou zadanou akci. Klíčovou funkcí systému Maven je řešení závislostí. To znamená, že není nutné ručně kopírovat knihovny (JAR, WAR, EAR) a umisťovat je na classpath.“[4] [14]

Při vývoji jednotlivých microservices jsem použil dva modely projektu (dva soubory pom.xml). Jeden kořenový (parent) modul **obrázek č. 4. 5** a druhý modul, který dědí z kořenového. Modul, který je potomkem předka se zapisuje příkazem <modules>.

```
<modules>
    <module>order-service-war</module>
</modules>
```

Všechny microservices mají příkaz <packaging> nastaven na formát archivu Jar. Po spuštění příkazu `mvn clean install` maven zabalí microservices do archivu Jar se všemi závislostmi a aplikačním serverem Tomcat.

[3] PostgreSQL: https://postgres.cz/wiki/PostgreSQL
[4] Vojta Hordějčuk: http://voho.eu/wiki/maven/

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>cz.fel.cvut.order.service</groupId>
    <artifactId>order-service</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.7.RELEASE</version>
    </parent>

    <modules>
        <module>order-service-war</module>
    </modules>

    <build...>

    <repositories...>
</project>
```

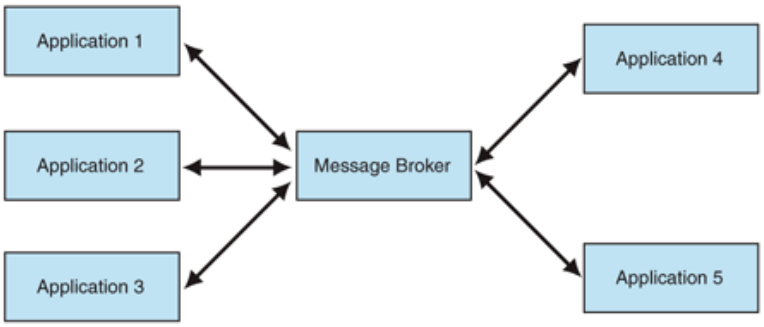
Obrázek 4. 5 Pom.xml microservice Order-service

4.1.5 Message broker

Každá microservice se chová jako samostatná služba. Má vlastní databázi a vlastní logiku. Problém ale nastane, pokud jedna microservice potřebuje data nebo logiku z jiné microservice. V ukázkové aplikaci (Evo) je po vytvoření objednávky odeslán email s informací o založení objednávky. Email odesílá microservice Email-service, ta ale nemá informace o tom, o kterou objednávku se jedná a komu má být zaslána. Nezná email příjemce. Jinými slovy, Email-service potřebuje data z Order-service a User-Service. Řešením je platforma Message broker, který řeší komunikací mezi microservices.

Message broker (zprostředkovatel) je platforma, která zprostředkovává a zpracovává komunikaci mezi dvěma aplikacemi (nebo microservices). Místo aby aplikace komunikovaly mezi sebou, komunikují pomocí message broker. Jedna aplikace odešle zprávu message broker a poskytne mu názvy ostatních příjemců. Message broker zprávu přijme a vyhledá registrované aplikace podle jejich názvu, poté jim předá zprávu.

Komunikace mezi aplikacemi obsahuje pouze odesílatele, message broker a určené příjemce. Message broker neodesílá zprávu žádným jiným aplikacím, jen těm, kterým je zpráva určena. Na **obrázku č. 4. 6** je zachycena komunikace message brokera.



Obrázek 4. 6 Komunikace Message Broker[5] [13]

Message broker může vystavovat různá rozhraní aplikacím, s kterými komunikuje a přenášet mezi aplikacemi zprávy. Jinými slovy message broker nepotřebuje, aby aplikace měly společné rozhraní.

Při vývoji ukázkové aplikace jsem použil Message Broker RabbitMQ, který podporuje protokoly (AMQP, STOMP, HTTP) pro přenos zpráv mezi službami. Jedná se o open source software.

4.2 Frontend

Frontendová část ukázkové aplikace je implementována pomocí Angularu 5 s knihovnou PrimeNg. Frontend přijímá data z backendu prostřednictvím formátu JSON. Na **obrázku č. 4. 7** je zobrazena služba napsaná v Type Scriptu, která volá backend pomocí Http.

4.2.1 Angular 5

Angular je open source JavaScript framework pro tvorbu single-page aplikací v HTML a JavaScript. Type Script je jazyk, v kterém se píší angular komponenty. Type Script je vytvořen a spravován firmou Microsoft. Jedná se o nadstavbu jazyka JavaScript. Angular je navržen tak, aby pomohl programátorovi oddělit zobrazovací logiku od aplikační logiky. Pro tento účel využívá Angular návrhového vzoru MVC (Model-View-Controller). Angular umožňuje pomocí speciálního zápisu vkládat dynamický obsah do statického HTML. Tento zápis se nazývá Data Binding. Zápis Data Binding je zobrazen na **obrázku č. 4. 8** řádek s kódem {{this.countOrder}}. Pro vývoj v angularu je nutné mít nainstalovaný software Node.js.

[5] Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/ff648849.aspx

```typescript
@Injectable()
export class OrderService {

  constructor(private http: HttpClient) { }

  public getAllOrder(): Observable<OrderDTO[]>{
    return this.http.get<OrderDTO[]>( url: `${environment.orderService}/order`);
  }

  public getOrder(id:number): Observable<OrderDTO>{
    return this.http.get<OrderDTO>( url: `${environment.orderService}/order/${id}`);
  }

  public createOrder(order: OrderDTO): Observable<StatusDTO>{
    return this.http.post<StatusDTO>( url: `${environment.orderService}/order/add`,order);
  }

  public updateOrder(order: OrderDTO,id:number): Observable<StatusDTO>{
    return this.http.put<StatusDTO>( url: `${environment.orderService}/order/${id}`,order);
  }

  public deleteOrder(id:number): Observable<StatusDTO>{
    return this.http.delete<StatusDTO>( url: `${environment.orderService}/order/${id}`);
  }

  public getAllTraining(): Observable<TrainingDTO[]>{
    return this.http.get<TrainingDTO[]>( url: `${environment.orderService}/training`);
  }
```

Obrázek 4. 7 Order service pro volání backendu v Type Script

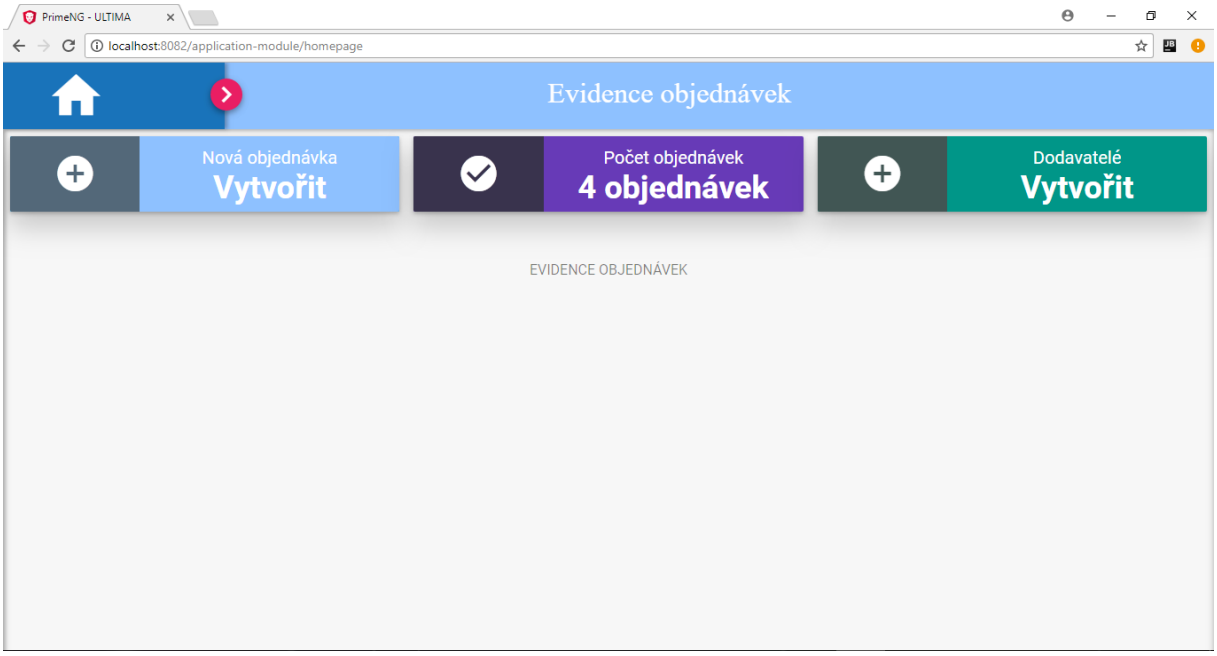
```html
<div class="ui-g dashboard offer">

  <div class="ui-g-4 ui-sm-12">
    <div class="ui-g card colorbox colorbox-2" [routerLink]="['/application-module/new-orderer']">
      <div class="ui-g-4 colorbox-left">
        <i class="material-icons">add_circle</i>
      </div>
      <div class="ui-g-8">
        <span class="colorbox-name">Nová objednávka</span>
        <span class="colorbox-count">Vytvořit</span>
      </div>
    </div>
  </div>

  <div class="ui-g-4 ui-sm-12">
    <div class="ui-g card colorbox colorbox-3">
      <div class="ui-g-4 colorbox-left">
        <i class="material-icons">check_circle</i>
      </div>
      <div class="ui-g-8">
        <span class="colorbox-name">Počet objednávek</span>
        <span class="colorbox-count">{{this.countOrder}} objednávek</span>
      </div>
    </div>
  </div>
```

Obrázek 4. 8 Order service pro volání backendu v Type Script

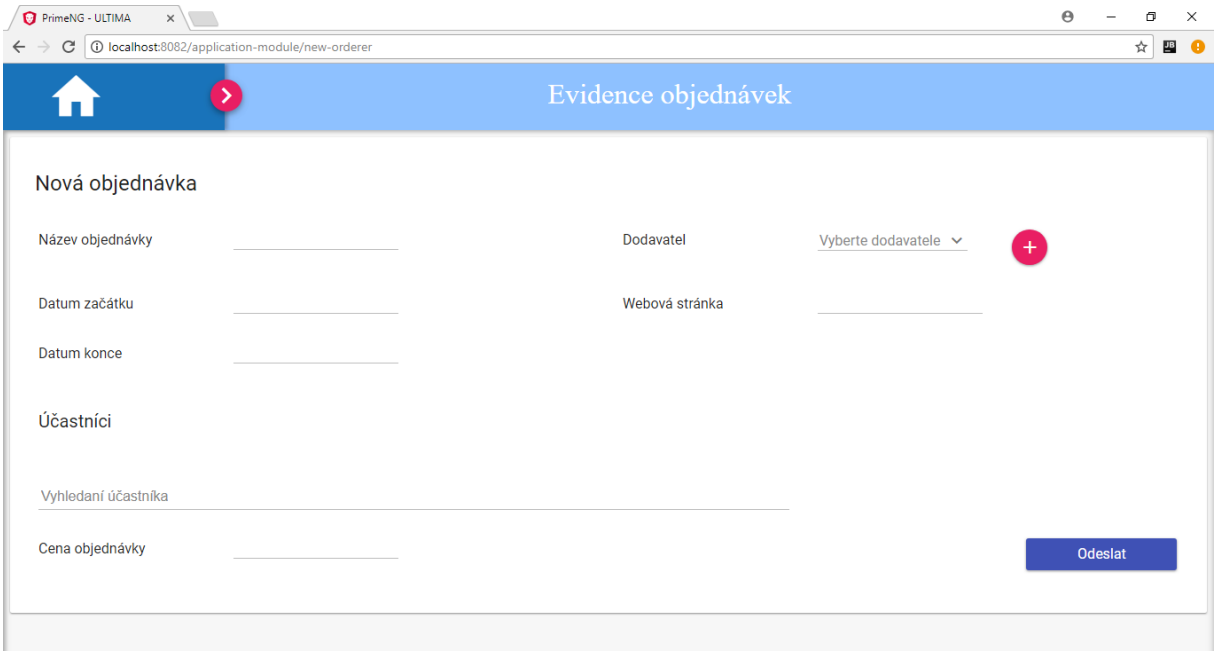
4.2.2 PrimeNg

PrimeNg je knihovna obsahující sbírku UI (User Interface) angular komponent. Knihovna obsahuje velké množství komponent pro tvorbu angular aplikace. Jedná se o tlačítka, tabulky, menu, diagramy, mapy, texty atd.

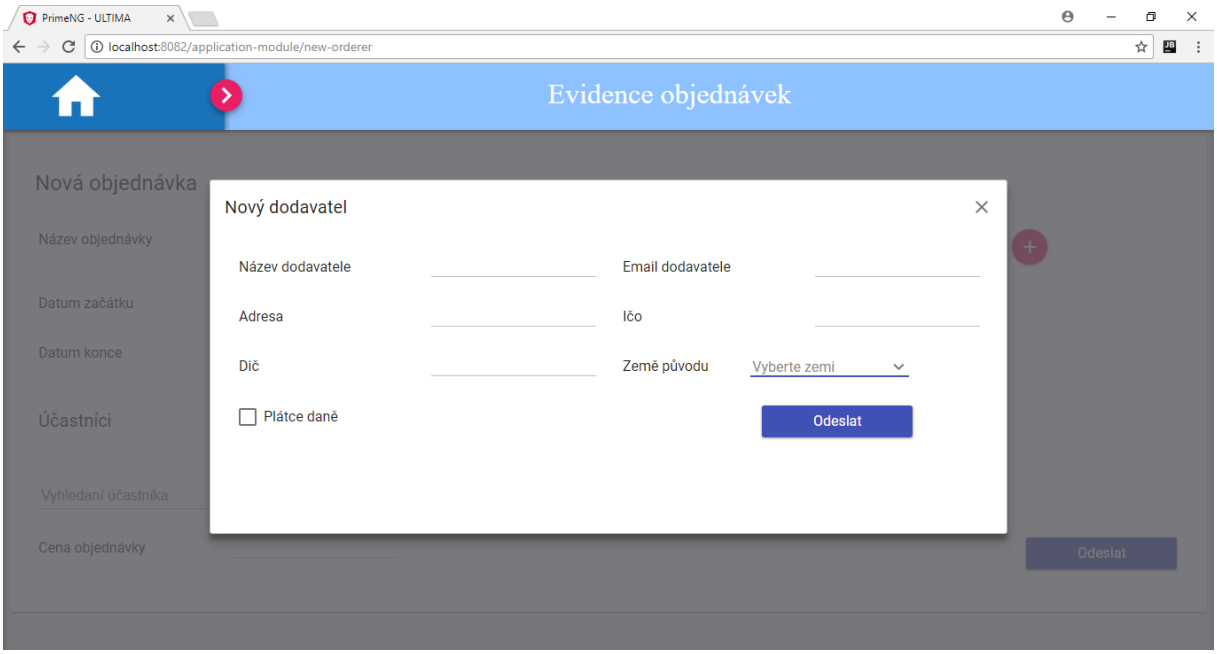
4.2.3 Gui Aplikace



Obrázek 4. 9 Domovská stránka



Obrázek 4. 10 Založení nové objednávky

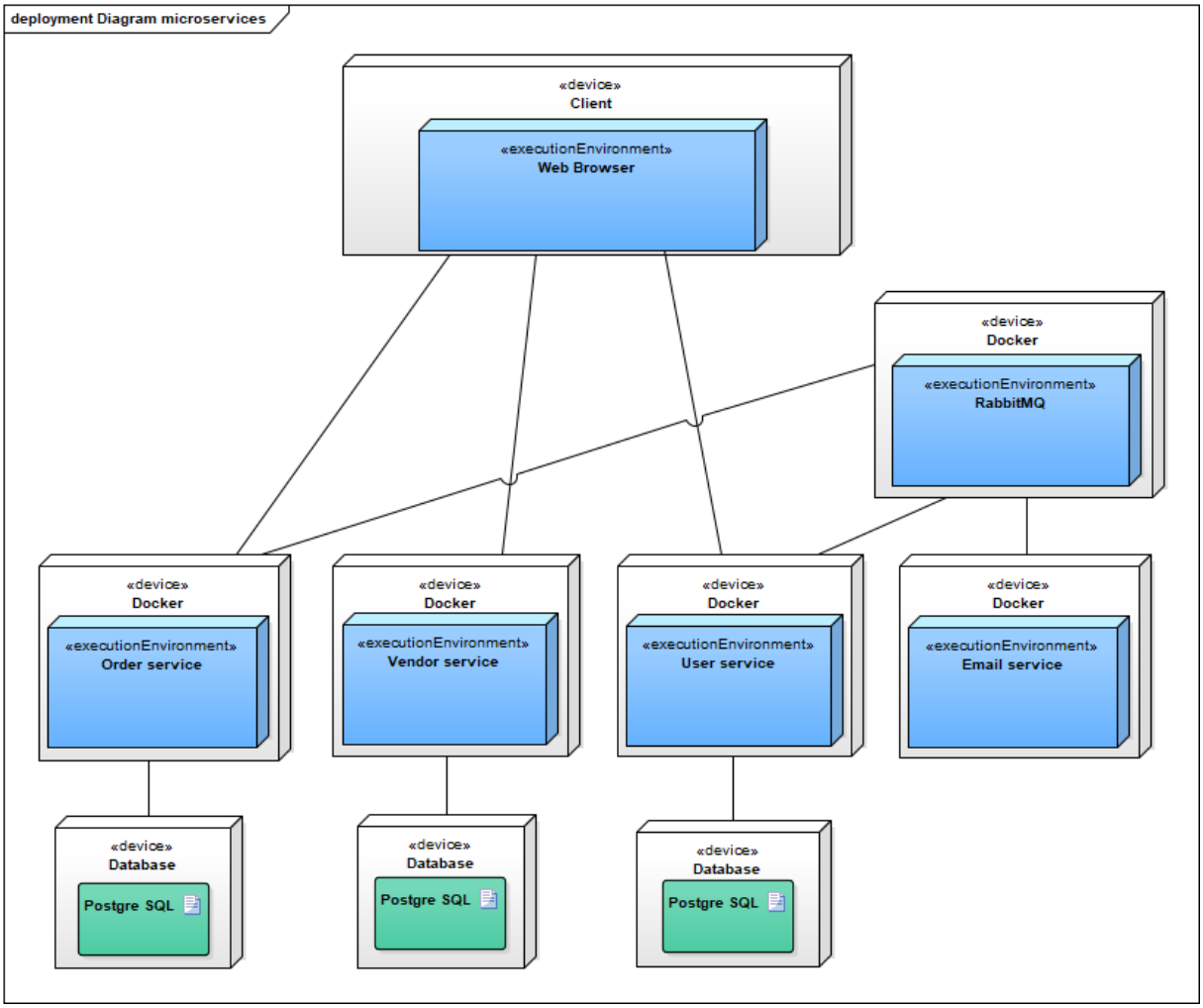


Obrázek 4. 11 Přidání nového dodavatele do objednávky

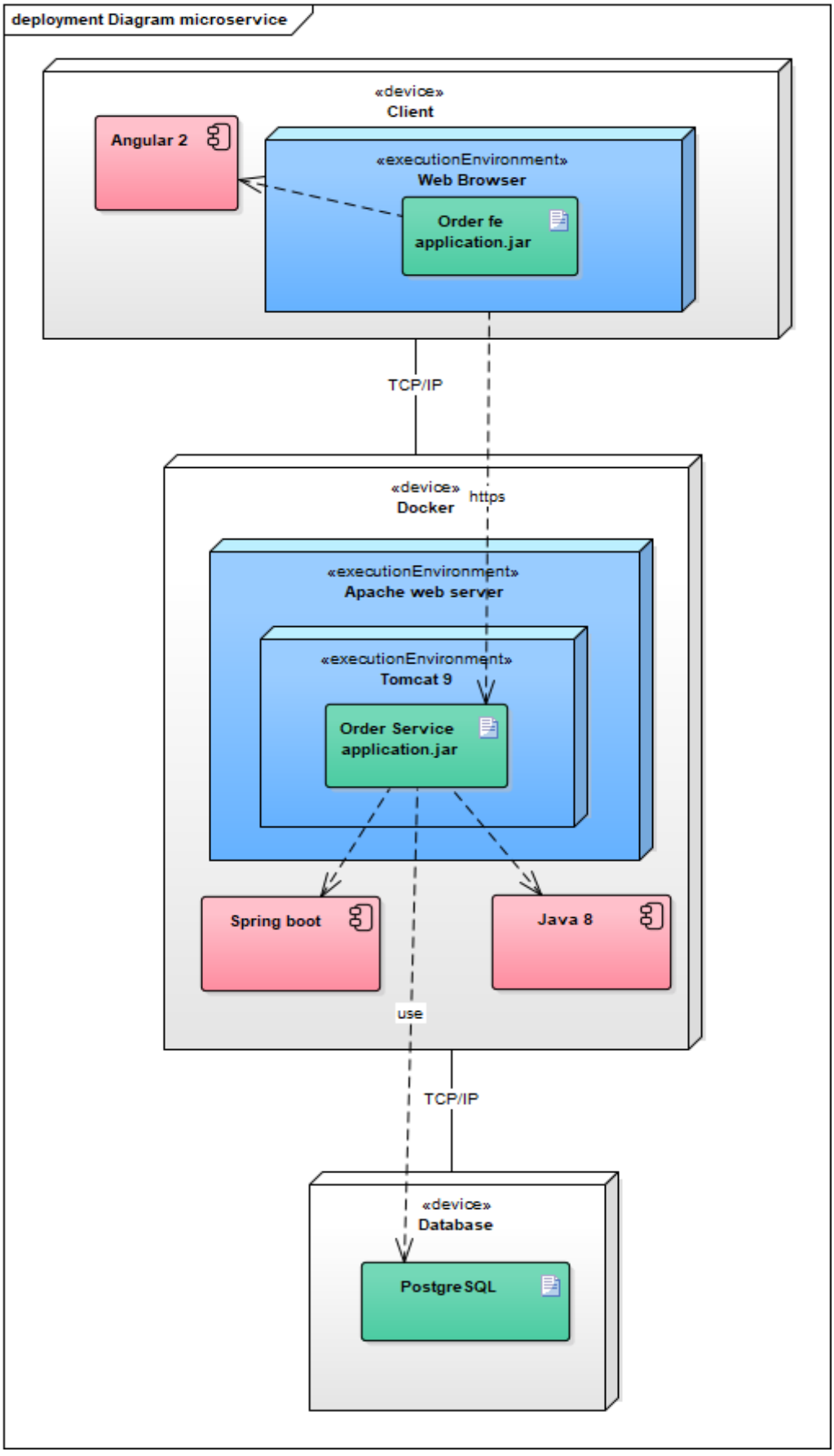
4.3 Diagram nasazení

Diagram nasazení obsahuje popis umístění jednotlivých částí aplikace na jednotlivé fyzické uzly a způsob jejich komunikace. Na **obrázku č. 4. 12** je povrchní diagram nasazení všech čtyř microservices. Order-service, Vendor-service, User-service a Email-Service. Microservices komunikuji s webovým prohlížečem přes protokol Http. Detailní diagram microservice Order je na **obrázku č 4. 13**. Ostatní microservice májí totožný diagram, proto je zde v UML zachycena pouze jedna. Vyjímku tvoří Email-Service, která na rozdíl od ostatních tří microservices nemá databázi.

Microservice Order, User a Email jsou propojeny s message brokerem, který obstarává komunikaci mezi těmito microservices pomocí protokolu AMQP (Advanced Message Queuing Protocol).



Obrázek 4. 12 Diagram nasazení aplikace microservices



Obrázek 4. 13 Detailní diagram nasazení microservice Order service

4.4 Uživatelské scénáře

V této části bakalářské práce demonstruji výhodu microservices na ukázkové aplikaci. Pro testování uživatelského rozhraní a demonstrace výhod microservices jsem použil program SeleniumIDE, který je doplňkem prohlížeče Chrome. Testování jsem realizoval na pěti scénářích.

4.4.1 Popis scénářů

Před každým scénářem je popsán stav jednotlivých microservices a celé aplikace. Dále následuje samotný popis scénáře a výstup.

Vysvětlivky symbolů v tabulkách:

- ● dostupná služba
- ● dostupná služba, pouze pro čtení (Get)
- ● nedostupná služba
- Dostupnost znamená, že služba reaguje na dotazovací metody get, post, put, delete

Scénář č. 1

Vstupní nastavení aplikace: všechny microservices jsou dostupné

Microservice	Order-service	Vendor-service	User-service	Email-service	Celá aplikace
Stav	●	●	●	●	●

Tabulka č. 1 stav služeb scénář č. 1

1) Založení objednávky
   - Vstup na domovskou stránku
   - Kliknout na tlačítko vytvořit objednávku
   - Vyplnit údaje o objednávce
   - Vybrat dodavatele
   - Vložit účastníky
   - Kliknout na tlačítko odeslat

Výstup: objednávka vytvořena, odeslání emailu účastníkům

Scénář č. 2

Vstupní nastavení aplikace: služba Vendor-service je nedostupná, zbylé služby jsou dostupné

Microservice	Order-service	Vendor-service	User-service	Email-service	Celá aplikace
Stav	●	●	●	●	●

Tabulka č. 2 stav služeb scénář č. 2a

2a) Založení objednávky

- Vstup na domovskou stránku
- Kliknout na tlačítko vytvořit objednávku
- Vyplnit údaje o objednávce
- Dodavatele nelze vybrat, zobrazeno dialogové okno s chybovou hláškou
- Vložit účastníky
- Kliknout na tlačítko odeslat

Výstup: objednávka vytvořena bez dodavatele, odeslání emailu účastníkům.

Změna nastavení aplikace: zprovoznění služby Vendor-service

Microservice	Order-service	Vendor-service	User-service	Email-service	Celá aplikace
Stav	●	●	●	●	●

Tabulka č. 3 stav služeb scénář č. 2b

2b) Editace objednávky

- Vstup na domovskou stránku
- V levém menu vybrat položku editace objednávek
- Vyhledat objednávku a kliknout na tlačítko editovat (ikona tužky)
- Vyberte dodavatele
- Kliknout na tlačítko aktualizovat

Výstup: objednávka aktualizována a byl přidán dodavatele.

Scénář č. 3

Vstupní nastavení aplikace: služba Email-service je nedostupná, zbylé služby jsou dostupné.

Microservice	Order-service	Vendor-service	User-service	Email-service	Celá aplikace
Stav	🟢	🟢	🟢	🔴	🟢

Tabulka č. 4 stav služeb scénář č. 3a

3a) Založení objednávky

- Vstup na domovskou stránku
- Kliknout na tlačítko vytvořit objednávku
- Vyplnit údaje o objednávce
- Vybrat dodavatele
- Vložit účastníky
- Kliknout na tlačítko odeslat

Výstup: objednávka vytvořena, email nebyl odeslán účastníkům

Změna nastavení aplikace: zprovoznění služby Email-service

Microservice	Order-service	Vendor-service	User-service	Email-service	Celá aplikace
Stav	🟢	🟢	🟢	🟢	🟢

Tabulka č. 5 stav služeb scénář č. 3b

3a) Odeslání emailu

- Email automaticky odeslán po zprovoznění služby Email-service

Výstup: email odeslán účastníkům

Scénář č. 4

Vstupní nastavení aplikace: služba Email-service je nedostupná, služba Vendor-service dostupná pouze pro čtení, zbylé služby jsou dostupné.

Microservice	Order-service	Vendor-service	User-service	Email-service	Celá aplikace
Stav	🟢	🟡	🟢	🔴	🟢

Tabulka č. 6 stav služeb scénář č. 4a

4a) Založení objednávky a vložení nového dodavatele

- Vstup na domovskou stránku
- Kliknout na tlačítko vytvořit objednávku
- Vyplnit údaje o objednávce
- Kliknout na tlačítko přidat nového dodavatele
- Vyplnit údaje o novém dodavateli
- Dodavatel neuložen, zobrazeno dialogové okno s chybovou hláškou
- Vložit účastníky
- Kliknout na tlačítko odeslat

Výstup: objednávka vytvořena, email neodeslán účastníkům, nevytvořen dodavatel

Změna nastavení aplikace: zprovoznění služby Email-service a Vendor-service

Microservice	Order-service	Vendor-service	User-service	Email-service	Celá aplikace
Stav	🟢	🟢	🟢	🟢	🟢

Tabulka č. 7 stav služeb scénář č. 4b

4b) Editace objednávky a vložení nového dodavatele

- Email automaticky odeslán po zprovoznění služby Email-service
- Vstup na domovskou stránku
- Kliknout na tlačítko vytvořit nového dodavatele
- Vyplnit údaje o novém dodavateli
- Kliknout na tlačítko odeslat

- V levém menu vybrat položku editace objednávek
- Vyhledat objednávku a kliknout na tlačítko editovat (ikona tužky)
- Vyberte dodavatele
- Kliknout na tlačítko aktualizovat

Výstup: objednávka aktualizována, přidán dodavatel a odeslán email účastníkům.

Scénář č. 5

Vstupní nastavení aplikace: služba User-service dostupná pouze pro čtení, zbylé služby jsou dostupné.

Microservice	Order-service	Vendor-service	User-service	Email-service	Celá aplikace
Stav	●	●	●	●	●

Tabulka č. 8 stav služeb scénář č. 5a

5a) Editace objednávky přidání účastníků

- Vstup na domovskou stránku
- V levém menu vybrat položku editace objednávek
- Vyhledat objednávku a kliknout na tlačítko editovat (ikona tužky)
- Přidat účastníky
- Změnit dodavatele
- Kliknout na tlačítko aktualizovat

Výstup: na objednávce se neaktualizoval seznam účastníků

Změna nastavení aplikace: zprovoznění služby a User-service

Microservice	Order-service	Vendor-service	User-service	Email-service	Celá aplikace
Stav	●	●	●	●	●

Tabulka č. 9 stav služeb scénář č. 5b

4b) Editace objednávky

- Vstup na domovskou stránku
- V levém menu vybrat položku editace objednávek
- Vyhledat objednávku a kliknout na tlačítko editovat (ikona tužky)
- Přidat účastníky
- Kliknout na tlačítko aktualizovat

Výstup: na objednávce se aktualizoval seznam účastníků

Na scénářích je demonstrována odolnost celé aplikace proti výpadku jedné nebo více microservices. Celá aplikace je dostupná i přesto, že jedna z microservices je zcela nedostupná. Například ve scénáři č. 2 je nedostupná Vendor-service. Při založení nové objednávky, nelze přiřadit dodavatele, ale objednávku založit lze, jelikož objednávky má na starosti jiná microservice (Order-service), která má vlastní databázi a oddělenou logiku od microservice Vendor-service. Služba Vendor-service může být nedostupná delší dobu a přesto to neovlivní dostupnost celé aplikace. Jinými slovy, uživatel, který v aplikaci pracuje, může nadále zakládat, upravovat a mazat objednávky, přestože jiná služba (microservice) není dostupná.

Pokud by byl stejný scénář aplikován na totožnou aplikaci, která by byla postavena na architektuře monolit, při nedostupnosti některé z části by zároveň byla nedostupná celá aplikace. Uživatel by s aplikací nemohl pracovat.

5 Vyhodnocení, výhody a nevýhody microservices

V závěrečné části bakalářské práce popíši výhody a nevýhody microservices vycházející z poznatků získaných při návrhu, implementaci a testování ukázkové aplikace. Dále vyhodnotím náročnost implementace ukázkové aplikace jako takové.

5.1 Výhody

5.1.1 Samostatnost

Microservices jsou samostatné a nejsou vzájemně závislé jedna na druhé. Každá microservice může být implementována v jiném jazyce, s odlišnou konfigurací, v různých verzích programovacího jazyka nebo jinou databází. Vývojář si může vybrat vhodný způsob implementace microservice. Tato výhoda přináší flexibilitu, jak navrhnout nejlépe vyhovující řešení nákladově efektivnějším způsobem.

V ukázkové aplikaci jsem pro každou microservice zvolil odlišnou konfiguraci souboru pom.xml. Microservice “order“, “vendor“ a “user“ mají každá svou databázi. Čtvrtá microservica “email“ databázi ne využivá.

5.1.2 Inovace

Na microservices lze testovat nové technologie, procesy nebo algoritmy. Lze napsat jednu microservice s novou funkčností na nové technologii. V případě, že služba nepracuje podle očekávání, lze ji snadno nahradit nebo přepsat. Náklady na změnu jsou oproti nákladům na změnu monolitu nižší, protože jakákoliv změna v monolitické aplikaci není tak jednoduchá jako u microservices.

Například pokud je nutné navýšit verzi knihovny Spring Boot, v microservice bude méně změn ve zdrojovém kódu. Naproti tomu u monolitu bude tato změna významně složitější.

5.1.3 Škálovatelnost

Aplikace implementovaná jako monolit, je zabalena do souboru WAR, nebo EAR a může být škálovatelná pouze jako celek. Naproti tomu u aplikace implementované na microservices, může být každá služba škálovatelná zvlášť. Aplikaci postavenou na microservices je možné škálovat horizontálně (přidáním dalších instancí aplikace), vertikálně (rozpad aplikace na více funkcí), nebo rozdělením aplikace na stejné jednotky.

5.1.4 Odolnost vůči výpadkům

Aplikace postavená na architektuře microservices je odolná vůči výpadkům. Tato výhoda byla demonstrována v kapitole 4. 4. 1. na uživatelských scénářích. Pokud není dostupná některá z microservices, aplikace jako celek je z pohledu uživatele stále v určitých omezeních dostupná. Například ve scénáři č. 2 je nedostupná služba Vendor-service a přesto lze objednávky zadávat do aplikace. V případě monolitu je při nedostupnosti některé z komponent aplikace nepoužitelná.

5.1.5 Schopnost náhrady

Protože jsou microservices samostatné a nezávislé moduly, umožňují nahrazení jedné microservice za druhou. Například služba User-service z ukázkové aplikace lze nahradit službou s podobnými vlastnostmi. Službu Email-service je možné nahradit například službou třetích stran. Nahrazení jedné služby za druhou bude mít minimální dopad na ostatní služby, protože jsou mezi sebou volně propojeny.

5.1.6 Jednoduchost nasazení

Microservices dovolují nasadit jednu službu nezávisle na zbytku celé aplikace. Tato výhoda umožňuje rychlejší nasazení nových změn zdrojového kódu. V případě, že je v aplikaci nebo zdrojovém kódu chyba, může být služba, ve které problém nastal, rychle odstavena, opravena a znovu nasazena.

Naproti tomu monolitická aplikace vyžaduje při změně zdrojového kódu, nebo opravy chyby, opětovné nasazení celé aplikace. Časté nasazování aplikace je rizikové, z důvodu možného zanesení chyb do celého systému.

5.1.7 Koexistence různých verzí

Microservices se při buildu společně zabalí s aplikačním serverem do jednoho spustitelného souboru JAR. Díky této možnosti mohou existovat různé verze služeb ve stejném čase vedle sebe. Poté stačí nakonfigurovat aplikační bránu, která rozhodne, na kterou verzi služby brána pošle dotaz.

5.1.8 Podpora DevOps

„DevOps je složenina anglických výrazů Development a (IT) Operations. Jedná se o přístup k vývoji software, který zdůrazňuje komunikaci, spolupráci a integraci mezi vývojářem a odborníky na informační technologie z provozu. DevOps je reakcí na vzájemnou závislost vývoje softwaru a IT provozu. Jeho cílem je pomoci organizaci rychle produkovat softwarové aplikace a služby. Jednoduché procesy jsou v DevOps jasně popsány. Cílem je maximalizovat předvídatelnost, účinnost, bezpečnost a udržovatelnost provozních procesů. Tento cíl je velmi často podporován automatizací. DevOps integrace se zaměřuje na dodání produktu, testování kvality, rozvoj produktu a vydávání oprav a nových verzí “[6] [15]

Microservices jsou podporované DevOps, protože umožňují automatické testování, nasazení a rychlé změny ve vývoji. Microservices se snadněji vyvíjejí menším týmům.

5.2 Nevýhody

5.2.1 Rozpad na microservices

Při vývoji složitějších systémů, může být problém na kolik microservices aplikaci rozdělit. Pokud je aplikace rozdělena na velký počet microservices, nebo naopak na menší počet microservices, může nastat problém, že daná microservice řeší více byznys domén, než jaké řešit má, nebo naopak méně byznys domén, než by bylo záhodno. Tato situace pak musí být řešena komunikací mezi ostatními službami. Každá microservice by měla řešit je vlastní doménu (know how).

5.2.2 Testování

Testování aplikace implementované na architektuře microservices, může být obtížnější než testování aplikace postavené na monolitu. Protože u microservices se musí vzít v úvahu potřeba vytvoření nových testovacích prostředí. Propojení mezi databázemi a ostatními službami. U monolitické aplikace postačuje nasadit WAR na aplikační server a zajistit propojení s databází.

[6] Wikipedia: https://cs.wikipedia.org/wiki/DevOps

5.2.3 Transakce mezi databázemi

Každá microservice má vlastní databázi, do které nepřistupuje žádná další služba. Problém nastane ve chvíli, kdy ukládáme data do dvou různých microservices. Transakce se může nacházet buďto v konzistentním, nebo nekonzistentním stavu.

„Datová transakce v relační databázi splňuje ACID.

- Atomicity – změny prováděné v transakci jsou buď realizovány jako celek nebo vůbec.
- Consistency – databáze je konzistentní před a po transakci
- Isolation – během transakce jsou data se kterými pracuje izolovány od dalších transakcí
- Durability – úpravy provedené transakcí jsou natrvalo uloženy v databázi“

5.2.4 Komunikace mezi microservices

Dvě microservices mohou mezi sebou sdílet data a odpovědnosti. V ukázkové aplikace mezi sebou sdílejí data Order-service a Vendor-service. Aby tyto dvě služby mezi sebou komunikovali, je zapotřebí třetí subjekt, který tuto komunikaci zprostředkuje. V případě, že zprostředkovatel komunikace není dostupný, pak není dostupná část funkcionality microservices. V ukázkové aplikaci byl použit zprostředkovatel RabbitMQ.

5.3 Vyhodnocení implementace ukázkové aplikace

Vývoj aplikace postavené na architektuře microservices se svou časovou náročností v pozdější fázi vývoje nijak zásadně neliší od vývoje aplikace postavené jako monolit. Přesto je v počáteční fázi vývoje nutné vynaložit více času, protože je zapotřebí naučit se vyvíjet a pracovat s novou technologií a novou architekturou, ať už se jedná o technologii zprostředkovatele (Message Broker) a komunikace mezi službami, nebo o jiný způsob vývoje celé architektury. Velkým plusem pro vývoj aplikace na architektuře microservices je, že každá microservice může být napsána v jiném programovacím jazyce viz. kapitola 2. 3. 3., Tato schopnost usnadní práci začínajícímu vývojáři, protože se nutně nemusí učit nový programovací jazyk, ale může microservise naprogramovat v programovacím jazyce, který ovládá.

Přechod z monolitické architektury na microservices architekturu není podmíněn investicí do nové infrastruktury. Důvod je ten, že microservices je samostatná jednotka s aplikačním serverem, která tudíž může bez problému fungovat v cloudu.

6 Závěr

Cílem bakalářské práce bylo naimplementovat aplikaci na architektuře microservices tak, aby bylo možné demonstrovat výhody a nevýhody použití architektury microservices. Ukázková aplikace měla poukázat na vlastnosti, které s použitím microservices souvisí a jsou zásadní pro rozhodování o způsobu implementace.

V první části jsem se zaměřil na popis vlastností architektury microservices a vhodné principy, které by měl každý vývojář při implementaci aplikace na architektuře microservices dodržovat. V první části jsem se dále zmínil o komponentách a technologiích, které jsou potřebné pro vývoj aplikace na architektuře microservices.

Před samotným vývojem ukázkové aplikace jsem ve třetí části bakalářské práce popsal funkčnosti dvou aplikací z bankovního sektoru, které jsou v současné době implementovány monoliticky a které by bylo vhodné implementovat na architektuře microservices. Jedná se o aplikace, které používá HR oddělení. První aplikace sloužila k evidenci objednávek a druhá pro evidenci námětů. Obě aplikace používají totožnou architekturu a technologie (Java, Spring, JSF, Maven).

Hlavní částí bakalářské práce je kapitola 4, "Implementace ukázkové aplikace". Z předešlé kapitoly jsem pro implementaci ukázkové aplikace zvolil aplikaci pro evidenci objednávek. Aplikaci jsem rozdělil na 4 microservices (4 byznys domény) tak, aby každá microservice zodpovídala pouze za vlastní byznys doménu. Pro vývoj ukázkové aplikace microservices jsem vybral programovací jazyk Java 8, Spring Boot, databázi PostgreSql nástroj Maven a Angular 5. Angular 5 spolu s knihovou PrimeNg jsem použil pro vývoj forntendové části (GUI). Backend microservices je rozdělen na tři vrstvy. Prezentační vrstvu, byznys vrstvu a databázovou vrstvu. Součástí 4 kapitoly jsou uživatelské scénáře, které demonstrovaly výhodu aplikace implementované na architektuře microservices. Jedná se o výhodu odolnost vůči výpadkům, kdy v případě nedostupnosti některé microservices zůstává zbytek aplikace stále funkční.

Výstupem závěrečné části jsou výhody a nevýhody architektury microservices spolu s vyhodnocením náročnosti implementace na této architektuře. Z celkového srovnání je patrné, že vývoj aplikací za pomoci microservices je jednoznačným přínosem a mohu jej, mimo jiné z vlastní zkušenosti, doporučit.

Literatura

[1] RAJESH, Rv. 2016. *Spring Microservices*. Birmingham: Published by Packt Publishing Ltd., str. 5, ISBN 978-1-78646-668-6

[2] RAJESH, Rv. 2016. *Spring Microservices*. Birmingham: Published by Packt Publishing Ltd., str. 6, ISBN 978-1-78646-668-6

[3] RAJESH, Rv. 2016. *Spring Microservices*. Birmingham: Published by Packt Publishing Ltd., str. 7, ISBN 978-1-78646-668-6

[4] RAJESH, Rv. 2016. *Spring Microservices*. Birmingham: Published by Packt Publishing Ltd., str. 9, ISBN 978-1-78646-668-6

[5] RAJESH, Rv. 2016. *Spring Microservices*. Birmingham: Published by Packt Publishing Ltd., str. 13, ISBN 978-1-78646-668-6

[6] RAJESH, Rv. 2016. *Spring Microservices*. Birmingham: Published by Packt Publishing Ltd., str. 14, ISBN 978-1-78646-668-6

[7] SAM NEWMAN, 2015. *Building Microservices.* United States of America: Published by O'Reilly Media, Inc., 978-1-491-95035-7

[8] NATE MURRAY, FELIPE COURY, ARI LERNER, AND CARLOS TABORDA. 2017. *ng-book 2, The Complete Guide to Angular*. Published in San Francisco, California by Fullstack.io.

Elektronické zdroje

[9] *zdroják.cz: Návrhové principy: SOLID* [online]. Dostupné z: https://www.zdrojak.cz/clanky/navrhove-principy-solid/

[10] *Getting started with Spring MVC and Twilio* [online]. Twillio Blog. Dostupné z: https://www.twilio.com/blog/2015/11/getting-started-with-spring-mvc-and-twilio.html

[11] *Splitting a Monolithic Application Into Services* [online]. DZone / Integration zone. Dostupné z: https://dzone.com/articles/splitting-a-monolith-application-into-services

[12] *PostgreSQL Wiki* [online]. postgres.cz. Dostupné z: https://postgres.cz/wiki/PostgreSQL

[13] *Message Broker*, *Microsoft* [online]. MSDN Microsoft. Dostupné z: https://msdn.microsoft.com/en-us/library/ff648849.aspx

[14] *Maven, Digitální wiki* [online]. Vojta Hordějčuk. Dostupné z: http://voho.eu/wiki/maven/

[15] *DevOps* [online]. Wikipedia, Otevřená Ecyklopedie. Dostupné z: https://cs.wikipedia.org/wiki/DevOps