UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

DIPLOMOVÁ PRÁCE

2019 Bc. Jakub Žufánek

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Podnikový informační systém postavený na architektuře microservice

Bc. Jakub Žufánek

Diplomová práce

2019

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2018/2019**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jakub Žufánek**
Osobní číslo:	**I17230**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Podnikový informační systém postavený na architektuře microservice**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

Cílem práce bude vytvořit podnikový informační systém postavený na moderní architektuře založené na mikroslužbách (microservice architecture) v podobě několika autonomních spolupracujících komponent. Aplikace bude implementována v aplikačním frameworku Spring s přihlédnutím na uživatelskou použitelnost. Text práce bude kromě samotné problematiky obsahovat i přehled použitých technologií, rešerši o případných SW alternativách, analýzu s realizací aplikace a uživatelskou příručku.

Rozsah grafických prací:

Rozsah pracovní zprávy: **50-60 stran**

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

NEWMAN, Sam. Building microservices: designing fine-grained systems. Sebastopol, CA: O'Reilly Media, 2015. ISBN 978-1491950357
CARNELL, John. Spring microservices in action. Shelter, Island, NY: Manning Publications Co., 2017. ISBN 978-1617293986
ALLAMARAJU, Subrahmanyam. RESTful Web services cookbook. Sebastopol,CA.: O'Reilly, c2010. ISBN 978-0596801687
MILLETT, Scott. Patterns, principles, and practices of domain-driven design. Indianapolis, IN: wrox, a Wiley Brand, [2015]. ISBN 978-1118714706
MARTIN, Robert C. Čistý kód: [návrhové vzory, refaktorování, testování a další techniky agilního programování]. Brno: Computer Press, 2009. ISBN 978-80-251-2285-3
KRUG, Steve. Don't make me think!: a common sense approach to Web usability. 2nd ed. Berkeley, Calif: New Riders Pub., c2006. ISBN 978-0321344755.

Vedoucí diplomové práce: **Ing. Jan Merta**
Katedra řízení procesů

Datum zadání diplomové práce: **22. října 2018**

Termín odevzdání diplomové práce: **18. května 2019**

Univerzita Pardubice Česká republika

L.S.

Ing. Zdeněk Němec, Ph.D.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 17. listopadu 2018

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 29.08.2019

Bc. Jakub Žufánek

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu mé diplomové práce panu Ing. Janovi Mertovi za vedení a konzultace po dobu zpracování diplomové práce. Rovněž bych chtěl jmenovitě poděkovat Gabriele Zichové. Také bych chtěl poděkovat Pavlovi Snítilovi za odborné konzultace ohledně Spring frameworku. V poslední v řadě bych chtěl poděkovat své rodině a svým přátelům za podporu.

ANOTACE

Cílem diplomové práce je vytvořit podnikový informační systém postavený na moderní architektuře založené na mikroslužbách (microservice architecture) v podobě několika autonomních spolupracujících komponent. Aplikace je implementována v aplikačním frameworku Spring s přihlédnutím na uživatelskou použitelnost. Text práce kromě samotné problematiky obsahuje i přehled použitých technologií, rešerši o případných SW alternativách, analýzu s realizací aplikace a uživatelskou příručku.

KLÍČOVÁ SLOVA

Domain-driven design, agilní vývoj, mikroservisní architektura, Java, Spring framework, Spring Boot, Spring Cloud, PostgreSQL

TITLE

Enterprise information system built on the microservice architecture

ANNOTATION

The aim of dimploma thesis is to create a business information system based on modern architecture, microservice architecture in the form of several autonomous cooperating components. The application is implemented in the Spring application framework with regard to user usability. In addition to the issue itself, the text of the thesis also contains an overview of the technologies used, a review of possible SW alternatives, an analysis with the implementation of the application and a user manual.

KEYWORDS

Domain-driven design, agile software development, microservice architecture, Java, Spring framework, Spring Boot, Spring Cloud, PostgreSQL

OBSAH

SEZNAM ILUSTRACÍ

SEZNAM ZKRATEK A ZNAČEK

API	Application Programming Interface
CI	Continuous Integration
CORS	Cross-origin Resource Sharing
DB	Databáze
DDD	Domain Driven Design
DI	Dependecy Injection
DTO	Data Transfer Object
EJBs	Enterprise Java Beans
GUI	Graphic User Interface
ID	Identification
IoC	Inversion of Control
IS	Informační Systém
IT	Informační Technologie
JPA	Java Persistence API
JSON	JavaScript Object Notation
JWT	JSON Web Token
MDD	Model Driven Design
MVC	Model View Controller
ORM	Object Relational Mapping
PC	Personal Computer
REST	Representational state transfer
RPC	Remote Procedure Call
SPA	Single Page Application

UML	Unified Modeling Language

ÚVOD

Účelem této práce je poskytnout čtenáři přehled a rozšířit mu vědomosti o postupech vývoje softwaru, aplikování metody Domain-driven design a následné promítnutí do mikroservisní architektury. Dále by měli být poskytnuty vědomosti o Spring frameworku a moderních technikách. Čtenář by měl být proveden od návrhu distribučního systému až po implementaci mikroservis.

Diplomová práce je rozdělena na dvě části, teoretickou a praktickou část. Teoretická část seznamuje čtenáře se základními informacemi o vývoje softwaru, kde z velké části bude rozvedena moderní metodika Agilního vývoje. Dále budou čtenáři poskytnuty informace o Domain-driven designu, který slouží jako návrh pro mikroservisní architekturu. Mikroservisní architektura bude využívat framework Spring.

V praktická části bude prvně představeno zadání fiktivní společnosti XDeliverer, která se zabývá doručováním zásilek. Na základě tohoto zadání bude navrhnut systém podle Domain-driven designu a následná transformace do mikroservisní architektury. Implementace bude provedena ve frameworku Spring, přesněji je využit Spring Boot a Spring Cloud.

Všechny aplikace budou napsány v Java 8 s využitím databáze PostgreSQL. Autor bude chtít aplikovat zabezpečení OAuth 2, které je v dnešní době hojně využíváno. V diplomové práci v části implementace systému budou aplikovány zásady čistého kódu, důraz bude kladen na dodržení objektově orientovaného programování, využití návrhových vzorů a využití moderních technik.

Autor si toto téma vybral možnosti aplikování mikroservis v praxi a rozšíření znalostí ve frameworku Spring. Samotnému vypracování diplomové práce předcházelo studium odborných článků a specializovaných publikací s odpovídající tématikou. Dále byly využity zkušenosti a znalosti autora, které získal v průběhu studia na vysoké škole a v praxi. Hlavním zdrojem informací se staly zejména tyto knihy: Domain-driven design: tackling complexity in the heart of software od Erica Evanse, Spring Microservices od Rajesh RV.

Cílem diplomové práce je vytvořit podnikový informační systém postavený na moderní architektuře založené na mikroslužbách (microservice architecture) v podobě několika autonomních spolupracujících komponent.

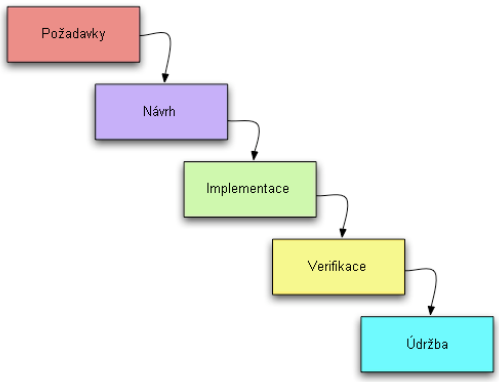
1 VÝVOJ SOFTWARU

Vývoj softwaru je proces, při němž je užíváno různých nástrojů a který je řízen specifickými pravidly. Cílem je vytvoření aplikačního softwaru. Od zrodu informačních technologií vzniklo velké množství různých metodik vývoje softwaru. Mezi nejvýznamnější patří přístup vodopádový, spirálový, agilní atd. Tyto různé způsoby vývoje se uplatňují na různé projekty. Pro lepší porovnání a představu jsou výše jmenované způsoby vývoje softwaru popsány v následující podkapitolách. [1]

1.1 Vodopád

Vodopádový vývoj je sekvenční vývojový proces, který uspořádáním procesů připomíná padající tok, tedy vodopád. Na následujícím obrázku (Obr. 1) lze vidět jednotlivé etapy. Jednotlivé etapy jdou postupně za sebou, nicméně může také docházet k jejich překrývání v rámci vývoje. Při postupu je třeba se zaměřit především na realizaci celého systému najednou, na plánování, časové rozvržení, termíny, rozpočet a jiné důležité prvky procesu. Po celou dobu projektu je dodržována přísná kontrola, a to prostřednictvím revizí schvalování uživatelem na konci jednotlivých fází apod. [1]

Etapy vodopádového vývoje



Obrázek 1 - Grafické znázornění vodopádového vývoje [2]

- **Požadavky –** Specifikace zadání od zákazníka a následné stanovení požadavků.
- **Návrh –** Analýza a návrh softwaru.
- **Implementace –** Implementace softwaru.
- **Verifikace –** Testování a integrace softwaru.
- **Údržba –** Provoz a údržba.

Výhody vodopádového přístupu

- Nejlepší struktura při stálých požadavcích.
- Jednoduchý z hlediska řízení.

Nevýhody vodopádového přístupu

- Některé fáze neumožňuji paralelní běh s jinými fázemi.
- Zákazník není schopen přesně stanovit veškeré požadavky předem, tudíž se zde špatně reaguje na změny požadavků.
- Zákazník vidí produkt až v posledních fázích projektu.

1.2 Spirálový vývoj

Spirálový vývoj vychází z Vodopádového vývoje. Tento vývoj se liší zejména iterativním přístupem a podrobnou analýzou rizik. Spirálový vývoj je rozdělen do celkem čtyř etap, a to konkrétně analýza, vyhodnocení, vývoj, plánování, viz obrázek číslo 2. Někdy se uvádí i rozčlenění na více stádií, kterými jsou specifikace, analýza, návrh, implementace, zavádění, analýza rizik. [1]

Etapy spirálového vývoje

Obrázek 2 - Grafické znázornění spirálového vývoje [2]

- **Analýza** – Definují se cíle, alternativy a rozsah iterace.
- **Vyhodnocení** – Vyhodnocení alternativ, řešení rizik.
- **Vývoj** – Implementace produktu a kontrola.
- **Plánování** – V této etapě probíhá plánování další iterace.

1.3 Agilní vývoj

Agilní vývoj je v této práci rozepsán více podrobněji, a to z důvodu důležitosti pro Domain-driven design (DDD). Tato metodika je iterativní a inkrementální. V posledních letech se agilní vývoj rozmohl a našel i svá uplatnění v jiných oborech s trochou modifikace způsobu vývoje. Manažer týmu by se měl soustředit na požadavky klienta, ale také na kvalitu a produktivitu členů týmu. U agilního vývoje pracuje tým, který má malý počet členů. Tento tým se skládá z analytiků, vývojářů a testerů atd. Malý tým má tendenci se sebezdokonalovat, což je proces, který má po technické stránce základ v jednotlivých členech, ale také v lepší komunikaci a lepší spolupráci celého týmu. Agilní vývoj má tu výhodu, že lépe reaguje na změny požadavků klienta a zároveň zapojuje klienta do vývoje. K problematice agilního vývoje je potřeba vysvětlit dva termíny: agilní metodika a agilní technika. [1,4]

Agilní metodika

Agilní metodika je jednou z metodik pro vývoj softwaru. Je zaměřena na způsob rozvržení a verifikace práce. Výsledkem agilní metodiky je lépe organizovaná práce. [1,4]

Agilní technika

Agilní technika je zaměřena na členy týmu. Pomocí této techniky dosáhneme větší produktivity, kvalitnějšího softwaru, odstranění chyb nebo špatných návyků a v neposlední řadě také přesnějšího dodržení specifikace. Výsledkem agilní techniky je kvalitnější produkt. [1,4]

Rozdíly mezi agilní metodikou a technikou jsou následující:

- Agilní metodika je zaměřena na možnost změny. V agilní technice se této problematice nevěnuje pozornost.
- Agilní metodika zapojuje vývojáře a vedení.
- Agilní technika je konkrétní činnost.
- Agilní metodika se soustředí na organizaci práce, a proto lze aplikovat agilní metodiku i v jiné oblasti nežli v IT.

Etapy projektu

1. Nultá etapa
2. Analýza změny
3. Implementace požadované vlastnosti/í
4. Ukázka klientovi
5. Jestliže není aplikace hotová, vracíme se k bodu 2

6. Jestliže aplikace je hotová, přesouváme se k etapě údržba a rozvoj

Nutno podoktnout, že etapy 2–4 jsou brány jako **jedna iterace**. Tato jedna iterace probíhá tak dlouho, dokud není projekt úspěšně dokončen, nezměnily se priority klienta nebo nedošel rozpočet na produkt. Jednotlivé etapy jsou brány sekvenčně a pouze za určitých okolností se vrací k určitým etapám. Níže jsou popsány jednotlivé etapy. [1,4]

Nultá etapa

V nulté etapě se celý projekt přivádí do chodu. Začíná se vždy krátkou analýzou, po které následuje naprogramování základní činnosti. Volí se takové množství aplikace, aby se dalo předvést klientovi. [4]

První etapa – Analýza změny

V první etapě analytik dostane priority od klienta, které jsou zásadní pro další iteraci. V této fázi je potřeba provést analýzu a naplánovat další iterace. Je žádoucí brát v potaz, že některé práce nebyly nebo nemusí být dokončeny, a proto je nutné je začlenit do plánu další iterace. Tyto nedokončené práce jsou zahrnuty jen za předpokladu, že mají jistou prioritu a klient je stále vyžaduje. [4]

Analytici a vývojáři společně upravují model, aby vyhovoval daným změnám. Dále se také připravují akceptační testy, změna datového modelu, změna business logiky atd.

Implementace požadované vlastnosti/í

V této etapě by měl být tým naprosto soběstačný a měl by se sám řídit. Manažer pravidelně kontroluje plnění úkolů, odstraňuje problémy, které brání vývojářům v práci a v neposlední řadě konzultuje změny do další iterace. Celý tým se snaží dodat vše v plánované iteraci. V dobře fungujícím týmu je role manažera skoro minimálně vytížená. Tato etapa trvá nejdéle. [4]

Ukázka klientovi

Zákazníkovi se ukazují pouze plně dokončené práce. Další nedokončené práce se musí skrýt/odstranit v Graphics User Interface (GUI). V této fázi je také zákazník informován o nedokončených pracích. V neposlední řadě zákazník zhodnocuje změny za danou iteraci a upřesňuje, které změny se budou nadále implementovat. [4]

1.3.1 Scrum

Scrum je jednou z metodik agilního vývoje, která byla zavedena už na počátku devadesátých let. Tato metodika je neznámější a nejvíce rozšířena. Ideální počet přijímaných osob do týmu

je od čtyř do patnácti. Tento tým by měl pracovat ideálně v jedné místnosti, a to z důvodu lepší spolupráce a hlavně komunikace. [5]

U s*crumu* jsou pracovníci rozděleny do dvou skupin, a to na tzv. „prasata“ a „kuřata“. Toto označení skupin vychází z povídky O praseti a kuřeti. Prase dává maso a kuře vejce. Prase je v tomto případě zapojeno a kuře se pouze s problémem potýká. Jinými slovy prasata jsou lidé, kteří jsou přímo zapojeni do vývoje aplikace, a kuřata uživatelé produktu, manažeři apod., tedy lidé, kteří přímo nezodpovídají za vývoj. Mezi „prasata“ patří: **product owner** a **scrum master**. Mezi „kuřata“ patří: **stakeholders** a **manažeři**. [5]

- **Product Owner** je osoba, která je zodpovědná za úkoly, co budou implementovány v dalším *sprintu*. Správný *product owner* se doslova stará o klientův produkt a hledá různá vylepšení.
- **Scrum Master** je osoba zodpovědná za plynulý chod *sprintu*. Stará se o vývojáře a řeší s nimi funkčnost počítačů, dostupnost potřebných softwarů, spory atd. Tato osoba má na starost všechny problémy, který ohrožují *sprint* z pohledu rychlosti. *Scrum master* může být převlečený manažer nebo analytik, ale nikdy by neměl být programátorem. Problém programátora jako *scrum mastera* je, že nedokáže odfiltrovat programátory od rušivých vlivů.
- **Stakeholders** jsou lidé, kteří částečně zastupují zákazníka. Do této skupiny patří například testeři, kteří vrací různé chyby a mají připomínky k odvedené práci.
- **Manažeři** jsou osoby, které pomáhají nastavit prostředí.

Při *scrumu* je velmi prospěšná přítomnost zákazníka. Zákazník se adaptuje do diskuzí, zodpovídá nejasné otázky a svými připomínkami pomáhá, aby vznikl užitečný produkt, respektive produkt, který si sám přeje. [5]

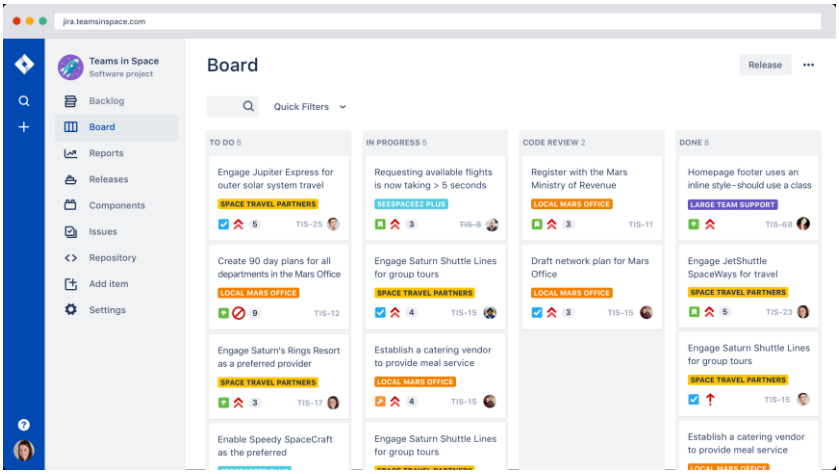
Před uvedením postupu s*crum* je potřeba ujasnit základní termíny, a to: *Sprint*, *Board*, *Product Backlog*, *Release Planing*, *Daily stand-up*, *retrospektiva*.

Sprint

Sprint je doba trvání jedné iterace. *Sprint* se skládá z plánování, *daily stand-upů*, vývoje, *retrospektivy*.

Board

Board neboli nástěnka, reprezentuje aktuální děj sprintu/projektu. Nástěnka slouží jako přehled pro klienta. Dobrou podporu *scrum* nástěnky mají webové aplikace Trello a Jira od Attlassian. Tato nástěnka je rozdělena do následujících sloupců, viz obrázek číslo 3:



Obrázek 3 - Ilustrativní Jira nástěnky pro agilní vývoj [25]

- **Product backlog** – Zde se objevují úkoly, který jsou navrženy pro implementaci. Některé firmy praktikují finanční ocenění těchto úkolů, a tak dávají lepší přehled zákazníkovi o finanční stránce projektu. Z **product backlogu** se přesouvají do sloupce **to do**. Akci plánuje pouze *product owner* s možností účasti zákazníka a managmentu. Sloupec se plní na základě zákazníka pod dohledem *product ownera*.
- **To do** – Tento sloupec obsahuje aktuálně naplánované úkoly na daný sprint. Dané úkoly si rozebírají jednotliví vývojáři na *daily stand-upech* nebo jsou úkoly přiřazovány *scrum masterem* ke specifickému vývojáři. V neposlední řadě se dělají časové odhady, a to pomocí Fibonacciho posloupnosti. Na konci *sprintu* by měl být v ideálním případě tento sloupec prázdný. Vývojáři přesouvají úkoly z **to do** sloupce do sloupce **in progress**.
- **In progress** – Sloupec obsahuje aktuální úkoly, které jsou ve vývoji. Odtud vývojáři přesouvají otázky do sloupce **code review**.
- **Code review** – Jedná se o sloupec s úkoly, které čekají na kontrolu od ostatních vývojářů. Vývojář zhlédne daný kód a ověří jeho funkčnost. Po akceptaci tímto vývojářem se přesouvá úkol do sloupce **done**.
- **Done** – Poslední sloupec slouží ke shromažďování všech hotových úkolů za daný *sprint*. Na konci *sprintu* se předvedou tyto úkoly zákazníkovi, který je následně přesune do sloupce **production** nebo vrátí zpět do sloupce **product backlog**.

V každé firmě je nástěnka vedena jinak. Ať už se týká názvosloví, nebo lehké modifikace chování ve smyslu, že se přidá ještě sloupec **review by product owner**. Tento sloupec například následuje po **code review**. V uvedeném sloupci provádí kontrolu *product owner,* aby byla ověřena funkcionalita úkolu. Ověření probíhá z důvodu, že *product owner* je nejvíce seznámen s problematikou a dokáže odchytit mnoho chyb před ukázáním funkcionality klientovi.

Release Planing

Plánování probíhá na začátku každého *sprintu* a účastní se ho celý tým, především *product owner*. Je zde vhodná i přítomnost klienta.

Daily stand-up

Daily stand-up je schůzka, která probíhá na začátku pracovního dne. Na schůzce se setká celý tým jakožto *scrum master*, vývojáři a ostatní příslušníci týmu. Tato schůzka probíhá ve stoje, a to proto, aby každý člen týmu šel rovnou k věci a nestál moc dlouho. Doba trvání *daily stand-upu* by v ideální případě měl trvat 10-15 minut. Během setkání každý člen týmu zodpoví následující otázky:

- Co jsem dělal včera?
- Co budu dělat dnes?
- Mám s něčím problém?

Na základě odpovědí na dané otázky řeší *scrum master* vzniklé problémy. Komunikace je zde velice užitečná, protože vývojář řeší svůj problém a v dosti případech se stává, že odpověď ví jiný člen týmu. Pokud nikdo z kolegů nedokáže poradit, pak se problém řeší se *scrum masterem*.

Retrospektiva

Tato schůzka je na konci každého *sprintu*. Členové týmu jednotlivě vyjadřují svoje názory ohledně spolupráce s ostatními členy. Jsou zde vyzdviženy klady spolupráce, které je třeba uchovat v nadcházejících *sprintech*. Mezi pozitiva lze zařadit například ochotu řešení problémů v týmu, shodu stejných vývojových prostředí atd. Jsou zde však řešeny i záporné věci, mezi které spadá například nedodržení termínu úkolů, špatně odvedený code review atd. Zápory je třeba do příštího sprintu odstranit a klady ba naopak zachovat.

Ukázka průběhu jednoho Sprintu

Jako názorný přiklad je stanovena délka *sprintu* na jeden týden. První řadě se naplní *product backlog* na nástěnce, kde jsou podle priority seřazeny úkoly.

1) **Release Planning** je proveden první den sprintu. *Product owner* přesune prioritní úkoly do sloupce **to do**. Zadání úkolů se nadále již nesmí změnit, pouze je zde možnost vyměnit úkol za stejně časově náročnou jinou úlohu. Během přesouvání úkolů se provede časový odhad. V prvním sprintu se rozplánuje časový odhad na 100 %, každý člen z týmu je vytížený, z čehož lze odhadnout celkovou efektivnost týmu. V dalších *sprintech* by se měla odhadnout práce na 70 % pracovní doby a 30 % času by se mělo zanechat jako rezerva. Rezerva je důležitá z hlediska výskytu skrytých problémů. V prvních sprintech je problematické odhadnout časovou náročnost na daný tým a dělá se časová optimalizace po každém sprintu.
2) **Daily stand-up** je proveden na začátku každého dne. Za přítomností všech členů týmů. Každý člen zodpoví následující otázky: Co jsem dělal včera? Co budu dělat dnes? Mám s něčím problém? Celý daily stand-up vede scrum master.
3) **Retrospektiva** je provedena na konci sprintu. Každý člen vyjádří svůj názor a zhodnotí, co se povedlo a co ne. Na tato hodnocení zpětně reaguje celý tým a snaží se zavést nějaké zaopatření.
4) **Ukázka klientovi.** Po každém *sprintu* by se mělo ukázat zákazníkovi, co se stihlo a co naopak ještě schází. Na základě toho klient rozhoduje, co se bude implementovat dál.

Po provedení fáze je třeba se vrátit zpět k bodu 1 a to k *release planning*. Je důležité zmínit, že práce přesčas je zakázána. Zaměstnanec, který odpočíval, pracuje lépe a víc se soustředí na práci, což vede k tomu, že dělá méně chyb.

Výhody agilního vývoje

- Malý počet členů v týmu, který vede k lepší spolupráci a organizaci.
- Pravidelné dodávání softwaru.
- Agilní vývoj reaguje dobře na změny zadání.
- Zapojení klienta/investora do spolupráce.
- Ustálený tým vede k lepším výsledkům. Je zde snaha se stále zdokonalovat.
- Postupem času lze zlepšit odhad časové náročnosti práce.

Nevýhody agilního vývoje

- Náchylné na změny členů v týmu. Tato nevýhoda se týká týmu pracujícím v delším časovém horizontu.
- Problémy prvních časových odhadů práce.

2 DOMÉNOVĚ ŘÍZENÝ NÁVRH

Doménově řízený návrh lze nejčastěji nalézt pod zkratkou DDD a pod anglickým názvem Domain-Driven Design. V první řadě by autor chtěl zmínit, že tímto tématem se zabývá kniha *Domain-Driven Design: Tackling Complexity in the Heart of Software* od Erica Evanse, která byla vydána v roce 2003. Dlouho po vydání této knihy nebylo o DDD skoro vůbec slyšet, a to proto, že rok 2003 nebyl zrovna ideální pro vývoj mikroservis, které jsou úzce spjaty s DDD. V roce 2003 byla Java 5, Enterprise JavaBeans neboli EJBs a pomalu rozvíjející Spring framework. Toto období útlumu trvalo přibližně až do roku 2016, kdy byl zaznamenán velký rozmach mikroservis. Začala doba experimentování a aplikace začaly vyžadovat skrze svoji náročnost implementaci mikroservis. Domain-driven design je jakýsi přístup návrhu softwaru, který vychází ze zkušenosti mnoholetých vývojářů. Velmi důležité je podotknout, že tito vývojáři jsou zaměřeni na objektově orientované programování. Domain-driven design se uplatňuje ve vývojové fázi analýzy a návrhu. Domain-driven design podporuje agilní vývoj a návrh pro mikroservisní architekturu. [5, 6, 7]

Při modelování se hledí shora dolů. Není nutné se zde primárně se soustředit na to, které technologie se použijí, ale zaměřuje se zejména na tzv. byznys pravidla. Při implementaci jsou uplatňovány dva nástroje, a to:

- nástroj pro taktický návrh,
- nástroj pro strategický návrh.

Nástroje vzájemně spolupracují prostřednictvím *všudypřítomného jazyka*, jak lze vidět na obrázku číslo 4. O těchto nástrojích je pojednáno dále v práci.

Obrázek 4 - Domain-driven design diagram [6]

Domain-driven design nespecifikuje určité metodiky, ale určitý způsob myšlení nad daným modelováním. Zejména je zde snaha modelovat tak, aby se doménoví a softwaroví experti co nejvíce přiblížili k reálnému modelu. V ideálním případě by měl konečný vymodelovaný model odpovídat reálnému modelu v poměru 1:1. Pro lepší představivost a porozumění jsou zde uvedeny příklady z reálného světa. [5, 6, 7]

Podle Erica Evanse je DDD přístup k vývoji komplexního systému, ve kterém jsou uplatňována tato pravidla:

1. Naleznout komplexní jádro v kritické části domény.
2. Snaha o společné objevování ve spolupráci doménovými a softwarovými experty.

3. Psát software, který explicitně reflektuje model. Používat *všudypřítomný jazyk*.

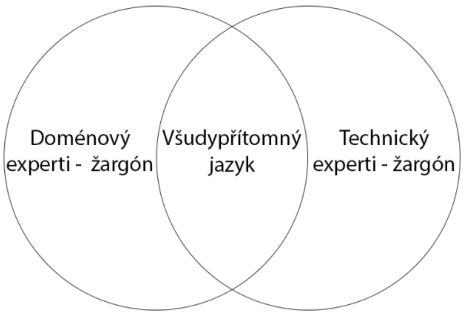
Bod číslo jedna shrnuje následující fakta. Je zde snaha hledat takové jádro, které reflektuje daný *model*. Jako příklad je uvedena situace v bance. V bance jsou zaměstnáni bankéři, hypoteční poradci, operátoři zákaznických linek, banka zároveň může poskytovat úvěry, lze si v ní směnit peníze do různých světových měn atd. U všech těchto pojmů (bankéři, operátoři, úvěry, směna) je snaha zařadit do jedné „krabičky" a následně najít jádro problému. Ku příkladu první model má název „Hypotéka". Do tohoto modelu spadá několik uvažovaných prvků – hypoteční poradce, úvěr a dále veškeré předměty a procesy s tím související. Jako druhý příklad lze uvést *model* nesoucí název „Zákaznická linka". V tomto *modelu* lze uvažovat opět několik prvků – jednotlivé operátory zákaznických linek. Podobný postup se používá na další a další procesy a pokračuje se dále v rozvíjení myšlenky. Celý objevování/modelování za účelem nalézt *model* probíhá v diskuzích doménových a technických expertů, který jsou zmíněni v bodě číslo 2. [5, 6, 7]

Bod číslo dva shrnuje následující poznatky. Pro objevování kritických částí je nutná spolupráce tzv. *doménových* a *technických expertů*. V první řadě je potřeba zmínit, kdo představuje doménového experta a kdo technického experta. [7]

Doménový expert je člověk, který rozumí dané specifické problematice. Pro názornou ukázku lze uvést aplikaci pro poskytování hypotéky. V tomto případě je doménovým expertem hypoteční poradce nebo člověk, který rozumí a orientuje se v hypotékách. Tento člověk stanoví chování za určitých pravidel, jinak řečeno byznys pravidla, která by měla daná aplikace splňovat. Příklad: Na hypoteční stránce je možné nalézt formulář pro žádost o hypoteční půjčku. Formulář je z části validovaný byznys pravidly. Například věk zákazníka musí být větší nebo rovno 18 let a menší než 65 let, aby mu mohla být poskytnuta hypoteční půjčka. [7]

Technický expert je člověk, který se pohybuje v oboru vývoje softwaru. Patří jsem frontend, backend vývojáři, ale i testeři atd. Tito experti mají zkušenosti s vývojem softwaru a na základě požadavků od klienta a doménových expertů volí technologie, který budou pro vývoj použity. Vedou mezi sebou diskuze týkající se podrobností vývoje, tedy například zda daná SQL databáze vydrží předpokládanou zátěž, dále na této úrovni probíhá rozhodování mezi různými platformami Java, C#, Ruby, Python atd. V neposlední řadě se diskutují i dané frameworky Spring boot, .NET Core, Django atd. Diskuze o technologiích v DDD jsou až sekundární. [7]

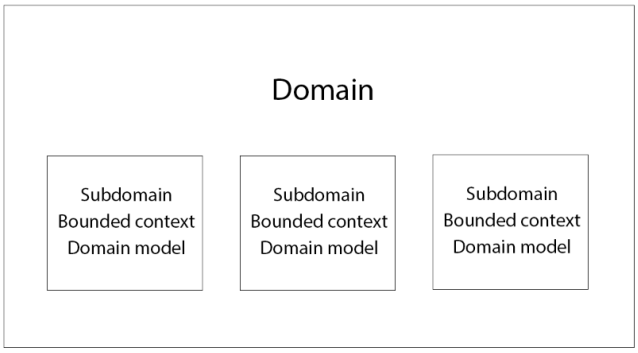
Bod číslo tři vymezuje následující problematiku. Software by v závěru měl reflektovat celý *model*. Názvy *domén* a jednotlivých vlastností by měl reflektovat kód. Je zde zejména snaha najít nějaký kompromis v názvu jednotlivých *domén* tak, aby jednoznačně reflektoval kód. Na první pohled se to může zdát jednoduché, ale v reálném životě je to složitější a funguje to trochu jinak. Více lidí většinou znamená i více názorů na danou věc, ať už se jedná debaty skrze terminologii nebo funkčnost. Mnoho lidí se na to dívá z jiné perspektivy, prosazuje jiný přístup, principy a terminologii. Pro představu můžeme porovnat *doménové* a *technické experty*, kteří zaručeně budou mít na jednu věc jiný pohled. Porovnání je znázorněno na obrázku číslo 5, kde je patrné, že je potřeba užít *všudypřítomný jazyk*, aby si *doménoví* a *techničtí experti* vzájemně porozuměli. O *všudypřítomným jazyku* je zmíněno později v práci. [5, 6, 7]



Obrázek 5 - Komunikace mezi experty. Zdroj: autor

2.1 Model domény (Domain model)

Doména je určitý prostor, který daná aplikace řeší. *Doména* muže být například bankovnictví, filmová databáze, internetový obchod, řízení letového provozu atd. Doménovou abstrakci tvoří *model domény*. *Doména* může obsahovat další *subdomény*. Pro lepší představu *domény* poslouží obrázek číslo 6. [6, 7, 8]



Obrázek 6 - Ukázka struktury domény. Zdroj: autor

Model je systém abstrakcí, který popisuje vybrané aspekty *domény* a lze jej použít k řešení problémů s touto doménou. *Model* je vymezení určité oblasti podle *ohraničeného kontextu*,

který reflektuje sám sebe a zároveň používá *všudypřítomný jazyk*. O *ohraničeném kontextu* a *všudypřítomným jazyku* je pojednáno později. *Model* potřebuje být jasný, ne velký. Správně vymodelovaný *model*, který je následně promítnut do kódu, dokáží číst i *doménoví experti*. *Doménoví experti* jsou schopni odvodit chování a částečně rozumí tomu, co daný kód dělá. Po vymodelování modelu je možné efektivně implementovat kód. [6, 7, 8]

Znalost o doméně je promítnuta do mnoha formátů, a to například do funkční specifikace, testů, UML diagramů a kódu. Z DDD vyplývá pět postupů, jak efektivně modelovat doménu:

1. **Provázání modelu a implementace**. *Model* bez možnosti implementace je v oblasti komerčního softwaru zbytečný a implementace, která není založena na *modelu*, snadno sklouzne k řešení problémů, které jsou mimo obor domény. Následně zde vzniká riziko opomenutí těch skutečně důležitých věcí.
2. **Model reflektující jazyk doménových expertů.** Aby *model* během celého vývoje odrážel skutečnosti dané *domény*, je třeba používat ve všech jeho reprezentacích shodný jazyk, který vychází z jazyka *doménových expertů*.
3. **Model reflektující znalosti.** Objekty *modelu* nejsou pouze datovým schématem, v *modelu* musíme zachytit také chování objektů a pravidla *domény*.
4. **Destilace modelu.** *Model* se během vývoje stále mění, integrují se do něj nově objevené koncepty domény a odstraňují se nebo modifikují ty, které se ukážou jako irelevantní.
5. **Komunikace a experimentování.** *Model* je předmětem komunikace mezi *doménovými experty* a dalšími vývojáři. Tím, jak se jej snažíme popsat větami, se často objeví dosud skryté souvislosti. Prostřednictvím diskuzí a brainstormingů je možné rychle procházet a posuzovat jeho různé variace.

2.2 Nástroj pro strategický návrh (Strategic Design tools)

V objektově orientovaném designu je k jednotlivým věcem přistupováno tak, že co jedna věc to samostatný objekt. Každý objekt má své specifické vlastnosti, metody atd. V nástroji pro strategický návrh se jednotlivé objekty zasazují do určitého kontextu, proto se zde přemýšlí pouze kontextově. [6, 7, 8]

Co je kontext?

Kontext znamená souvislost jevů nebo událostí, prostředí, podmínek atd. Kontext si lze z toho nejjednoduššího pohledu představit jako obal kolem nějaké věci. Příkladem může být bankéř, v jehož kompetencí je pouze sjednávání smluv v bance. Když se bankéř přesune

do pouště nebo jiného prostředí, kde nenachází klienty ke sjednání smluv, jeho pozice ztratí smysl. Přesun způsobí, že se mu markantně změnilo prostředí, ve kterém není schopen fungovat, anebo v daném prostředí není jeho práce natolik významná, aby byla využívána klienty. Druhým příkladem je pohled z pozice zákazníka, který má jiný význam v internetovém obchodě (kde může nakupovat, přidávat věci do košíku apod.) a odlišný význam na zákaznické lince (kde se věnuje pozornost zejména péči o zákazníky, reklamacím, radám apod.). Uvažovanému zákazníkovi se změnilo prostředí, dostává se do jiné pozice, která má zcela odlišný význam od pozice první, a proto se mu změnily i možnosti prováděných akcí. [6, 7, 8]

Nástroj pro strategický návrh má několik částí. Jednotlivé části nástroje pro strategický návrh jsou znázorněny na obrázku číslo 7 a jsou popsány níže v jednotlivých podkapitolách.

Obrázek 7 - Diagram nástroje pro strategický návrh [6]

Pro dovykreslení hlavní myšlenky, principu, jak celý *model* funguje, autor uvede několik příkladů. Prvním z nich je postavení stavebního inženýra (neboli projektanta) a klienta. Projektant představuje *doménového experta*. První otázka, kterou projektant položí klientovi, by měla být ve smyslu: „Jaký typ domu byste si představoval?“. Tato otázka je kladena z prostého důvoda, a to že projektant se potřebuje bavit v jednom určitým kontextu, jelikož existuje mnoho typů domů např. řadový domek, statek, chata, chalupa atd. Klient zvolí statek. Po volbě typu domu nastává snaha najít jádrové hodnoty. Co vlastně znamená statek? Pod statkem si lze představit například chov zvířat, dlouhé zelené pláně v širokém okolí domu, vyšší bezpečnost, odpočinek a klid, výchovu dětí poblíž přírody místo městského ruchu atd. Tímto postupem a zodpovídáním dalších prohlubujících otázek jsou nalezeny jádrové hodnoty.

V neposlední řadě je nutné se podívat na věci, které už ostatní lidé dávno udělali. V uvedeném příkladu se statkem je potřeba zhlédnout statky sousedů statkářů a vzít si vše podstaté z jejich rad a zkušeností. Tedy co by udělali lépe, kdyby oni stavěli svůj statek od základů. Rady ostatních statkářů aplikujeme i do klientova statku, aby nabýval vyšších hodnot. [6, 8]

Celý *model* tvoří zmíněný statek. Na statku stojí dům, stáje, stodola atd. Tyto budovy jsou *domény*. Zároveň jednotlivé *domény* tvoří *ohraničený kontext*. Dům (*doména*) se dá rozdělit na jednotlivé části (*subdomény*), a to například na kuchyň, obývací pokoj, terasu atd. Uvedené názvy (kuchyň, obývací pokoj, terasa atd.) jsou tvořeny *doménovým expertem* a promítají se tak do *všudepřitomného jazyka*, respektivě jako termíny do sdíleného slovníku. [6, 8]

2.2.1 Všudypřítomný jazyk (Ubiquitous Language)

Všudypřítomný jazyk se užívá z jednoho prostého důvodu. Na mnoha projektech mají členové týmu svoji vlastní terminologii. Proto může docházet mezi týmy ke „komunikačnímu šumu“. Jeden člen může mít na mysli roli „Běžného uživatele“ a druhý člen myslí „Návštěvníka“. [6, 7]

Všudypřítomný jazyk tedy představuje jednotnou komunikace mezi zúčastněnými lidmi na projektu. Speciální termíny v projektu jsou zaznamenány ve sdíleném slovníku. Termíny do slovníku zapisují *doménový experti* i s jejich přesným zněním. Od analytika přes grafika, frontend vývojáře, backend vývojáře, tester, manager, doménového experta atd. všichni používají stejnou terminologii a znají termíny ze slovníku. *Všudypřítomný jazyk* by se měl odrazit hlavně ve funkční specifikaci, dokumentaci projektu, kódu, testovacích scénářích, UML diagramech atd. [6, 7]

Všudypřítomný jazyk vnáší do projektu určitou výhodu. Jakmile přijde do projektu nový člen, jakýkoliv stávající člen týmu je schopen mu plně vysvětlit, jak celý *model* funguje. Tento společný jazyk by měli pochopit i netechnicky zaměření lidé.

2.2.2 Ohraničený kontext (Boundary context)

Ohraničený kontext je u projektů dalším důležitým bodem. Důvodem jeho zavádění je, že na velkých projektech dochází k dezinformacím. Různé skupiny programátorů mohly řešit daný problém v jiné části aplikace/*domény*, což následně způsobuje nedostatek komunikace. Někdy není jasné, kde má daný *model* hranice, tudíž není jasný rozsah celého *modelu domény*. K tomu slouží právě *ohraničený kontext*, který stanovuje hranice modelu. [6, 8]

Ohraničený kontext explicitně definuje hranice, ve kterých se model aplikuje. Výrazy ze slovníku *všudypřítomného jazyka* mají význam pouze a jen v daném kontextu. V týmu to

znamená, že jeden tým bude pouze dělat na jednom *modelu* (neboli *doméně*) – například správě smluv – a nebude tak rozšiřovat *model* o věci, které tam nepatří. Nadále by se tento *ohraničený kontext* měl odrazit do kódu i databáze, kde lze říci, že tahle sekce patří pouze tomuto *modelu*. [6, 8]

2.2.3 Kontextová mapa (Context map)

Při vývoji se stává, že daný *model* začne zasahovat do *modelu* druhého. Z důvodu absence globálního pohledu nebo neuvědomění si *ohraničeného kontextu* se zavádí *kontextová mapa*. [6, 8]

Kontextová mapa vyjadřuje vztahy mezi jednotlivými *ohraničenými kontexty/modely*. Měla by popsat místa, kde se jednotlivé *modely* spojují a znázornit tak veškerou komunikaci mezi *modely*. Nadále by se mělo vyznačit, co je sdíleno mezi *modely*, případně izolovat nebo zapouzdřit *model*. Prvně je doporučeno zmapovat existující, a pak provádět teprve změny na *modelu*. Nadále jsou vyjmenovány různé relace *kontextové mapy*. [6, 8]

Sdílené jádro (Shared Kernel)

Ohraničené kontexty mohou dojít ke sdílenému jádru. Sdílené jádro vyplývá z průniku mezi dvěma *modely*. V tomto případě se explicitně vyjádří hranice *submodelu*, a to za podmínky, že s tím souhlasí ostatní týmy. Jádro by se mělo udržet malé. Jakékoliv změny by neměly probíhat bez konzultace s týmy, které jádro sdílí. [6, 8]

Týmy zákazníků a dodavatelů (Customer/Supplier teams)

Týmy zákazníků a dodavatelů fungují na bázi vztahu. Tým dodavatelů reprezentuje například tým upstream, který provádí změny a na kterém jsou závislé jiné týmy – například tým downstream, který reprezentuje tým zákazníků. Bez vystavení potřebných funkcionalit, které potřebuje tým downstream jakožto zákazník, není možnost pracovat a čeká se na tým upstream jako dodavatele. Z tohoto důvodu by se měl udržovat čistý vztah mezi týmem zákazníků a dodavatelů, což znamená, že potřeby zákazníků by se měly promítnout do priorit dodavatelů a jejich následného plánování. [6, 8]

Konformista (Conformist)

Zde se jedná o vazbu týmů upstream/downstream podobně jako u předchozího odstavce. Tým upstream nemá motivaci poskytnout týmu downstream to, co potřebuje. Pomocí *všudypřítomného jazyka*, který sdílí, donutí týmy komunikovat mezi s sebou. [6]

Otevřená služba pro hosty (Open Host Service)

Otevřená služba pro hosty nabízí *ohraničený kontext* speciální služby, kterou může každý použít. Každý si tak může vytvořit vlastní integraci. Otevřená služba pro hosty se použije tehdy, pokud je integrace svázána s ostatními systémy. Dále je aplikována, pokud je implementace integrace příliš pracná. [6, 8]

Veřejný jazyk (Published language)

Zde se jedná o návaznost Otevřené služby pro hosty v předchozím odstavci. Tato část není nikde dokumentována a je potřeba ji dokumentovat.

Oddělené cesty (Separate ways)

Pokud dvě sady funkcí nemají žádný významný vztah, lze je od sebe oddělit. *Ohraničený kontext* po tomto verdiktu prohlašuje, že nemá žádné spojení. Oddělení přináší výhodu vývojářům, kteří nacházejí jednoduchá a specializovaná řešení. [6, 8]

Antikorupční vrstva (Anticorruption layer)

Antikorupční vrstva převádí *doménový model* do jiného. Oba tyto modely jsou zcela odděleny. Příkladem může být integrace staršího systému bez nutnosti převzít *doménové modely*. Ve starých systémech nemá smysl modelovat do DDD, takže se ponechá tak, jak je. [6, 8]

Velká koule bláta (Big ball of mud)

Když se zkoumá existující softwarový systém a je zde snaha jej pochopit, tak lze nalézt odlišné *modely* aplikované v rámci definovaných hranic. Často jsou také *modely* smíšené a nekonzistentní. Poté je potřeba vytvořit velkou hranici kolem tohoto nepořádku a označit ji jako velkou bahenní kouli. Zde není a nevzniká snaha použít nějaké sofistikovanější modelování. [6]

2.2.4 Průběžná integrace (Continuous integration)

Průběžná integrace se uvádí pod zkratkou CI. Jedná se o sadu nástrojů a procesů, které urychlují a zjednodušují vývoj. Mezi přední CI patří například Jenkins a CircleCI. Hlavní výhodou je ušetření času a následně i finančních prostředků. Vše začíná u Gitu. [6, 11]

Git

Nejdříve je potřeba definovat, co je to Git, přestože není hlavním prvkem CI. Je třeba jej vysvětlit pro dokreslení celého komplexu. Git je distribuovaný systém správy verzí. Slouží pro

verzování aplikace/kódu. Výhodou Gitu je, že mnoho vývojářů může pracovat na stejném projektu s rozdílným kódem. Každý vývojář si vytvoří vlastní větev, kde může vyvíjet novou funkčnost aplikace, opravit chybu, testovat atd. Novou větev si lze představit jako odbočku, kde je daný kód modifikován z původní větve, a to bez vlivu na původní větev. Tento kód je pak připojován do hlavní větve, která se díky CI může postupně nasadit do produkce. Další podstatnou výhodou Gitu je kontrola kódu vývojářem před sloučením do hlavní větve, tzv. „Pull request“ někdy i „Merge request“. Záleží na variantě Gitu, kdo jej poskytuje (Github, BitBucket, Gitlab). Pull request vede k odhalení chyb v aplikaci předem a poskytuje lepší zpětnou vazbu pro vývojáře. Dále skrývá mnoho dalších vylepšení, které usnadní vývojáři práci. Jedním z nich je tzv. „Git hook“, který spustí úlohu na CI.

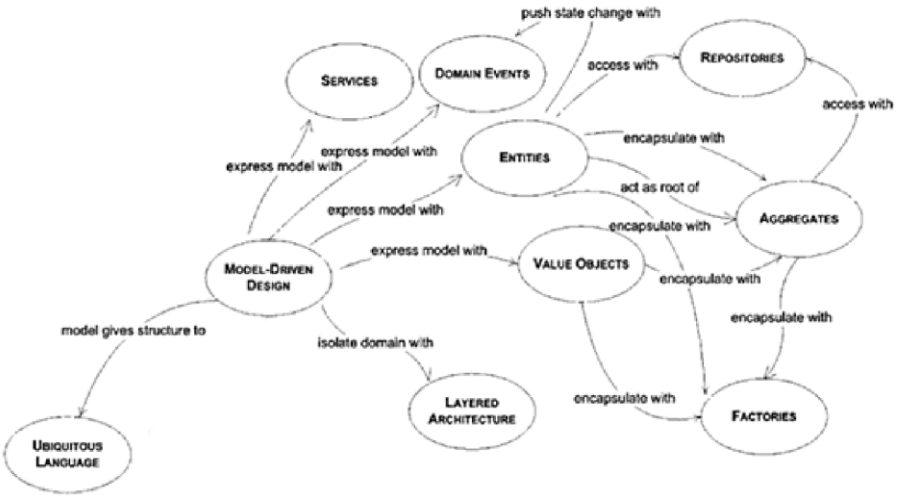
Zmíněná úloha spustí soubor procesů na CI. Nejdříve provede stažení kódu z Gitu, poté vykoná kontrolu dependency a jejich stažení. Jedná se o knihovny třetích stran, kdy celý proces probíhá přes Gradle nebo Maven na platformě Java. Po stažení dependency se realizuje samotné sestavení aplikace. V neposlední řadě se spouští testy, které se vyhodnocují. Následuje kompilace nové verze aplikace. Úplně posledním krokem je odeslání sestavené aplikace.

2.2.5 Implementace nástroje pro strategický návrh

Doménoví a techničtí experti spolu úzce spolupracují při návrhu *modelu*. Navrhování *modelu* probíhá pomocí skupinové kreativní techniky, tzv. brainstormingu. Experti se sejdou v jedné místnosti a diskutují o daných problémech, zaznamenávají důležité vlastnosti *domén*. Na základě těchto informací probíhá modelování, a to za přítomnosti obou skupin. *Technický expert* načrtne, co a jak bylo pochopeno od *doménového experta*. Na základě náčrtků a předělávání *modelu* dochází obě skupiny k nejlepšímu *modelu*. Pro lepší vyjádření myšlenek používají *techničtí experti* UML jazyk. Hojně jsou také používány diagramy případu užití (Use case diagram), diagramy aktivit (Activity diagram), sekvenční diagramy (Sequence diagram). [5, 6]

2.3 Nástroje pro taktický návrh (Tactical design tools)

Nástroje pro taktický návrh se zabývají podrobnostmi implementace. Obecněji lze říci, že se starají o *model* uvnitř *ohraničeného kontextu*. Mezi nástroje spadají například *servisy*, *agregáty*, *entity*, *repositáře*, *továrny*. Narozdíl od strategického designového nástroje se u nástrojů pro taktický návrh očekává, že dojde k změně v produktu. Nástroj pro taktický návrh a jeho struktura jsou znázorněny na obrázku číslo 8. [6, 7, 9]



Obrázek 8 - Diagram nástroje pro taktický návrh [6]

2.3.1 Modelově řízený návrh (Model driven design)

Modelově řízený návrh se někdy uvádí pod zkratkou MDD. Modelově řízený návrh využívá *model* k popisu byznys řešení, nezávisle na platformních omezení. *Model* tak reprezentuje *doménu*. Tento *model* je vnesen do UML diagramů, testů, programovacího kódu, funkční specifikace atd. Jakákoliv změna *modelu* musí být promítnuta do reprezentací. Všechny reprezentace pomáhají ujistit již známou vlastnost nebo odhalit nové vlastnosti *domény*. [6, 7, 9]

UML diagramy fungují spíše jako komunikační prostředek, protože vylepšují komunikaci mezi *doménovými experty* a *technickými experty*. [6]

MDD poskytuje sadu vzorů, podle kterých se modelují dané znalosti o *doméně*. Vzory reflektují specifické chování objektů. Na obrázku číslo 8 jsou zachyceny vzory. Jak je z obrázku patrné, dané vzory mohou projít vývojem, kde se buď slučují, přeskupují nebo rozdělují při modelování. [6, 9]

2.3.2 Servisy (Service)

Servisy nabízí pouze službu nebo jinými slovy rozhraní, poskytující operace. Servisy vykonávají pouze operace nad jednotlivými objekty. Jsou bezstavové. Servisy lze rozeznat podle slovesa a nelze je jednoznačně přiřadit k podstatnému jménu. Je potřeba se zamyslet, zda patří vážně do *domény*, nebo řeší pouze technickou část aplikace. [6]

2.3.3 Doménové události (Domain events)

Doménová událost je událostí v doméně, která slouží jako sledovací nástroj pro *doménový experty*, kteří se následně o tyto události starají. Ne všechny události jsou stejně důležité a sám *doménový expert* stanoví, které události budou upřednostněny a sledovány. Doménová událost je neměnná a měla by obsahovat následující atributy:

- čas vzniku,
- záznam nějaké události,
- identitu entity, která událost vyvolala.

Na základě *doménových událostí* se dobře hledají chyby v aplikaci. V jiných případech je *doménový expert* schopen dobře analyzovat celý proces a navrhnout tak lepší *model*. [6]

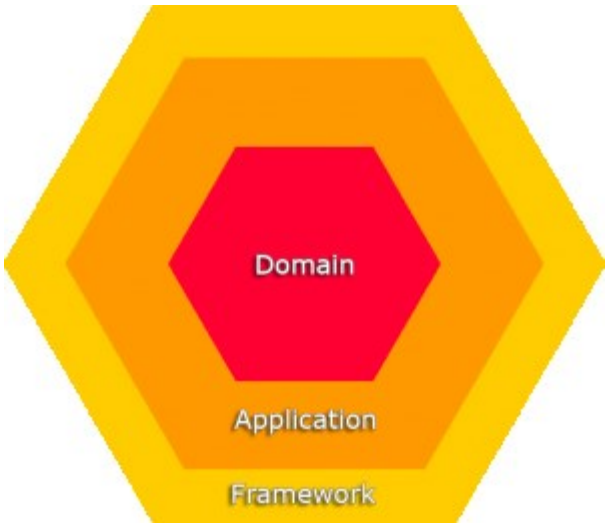
2.3.4 Vrstvená architektura (Layered architecture)

Vrstvená architektura rozděluje aplikaci do vrstev. Jedná se odizolování jednotlivých částí do svých vrstev, což přináší řadu výhod. Mezi výhody patří především přehlednější struktura aplikace, aplikace jednoduší na práci, vytvoření plnohodnotné aplikace atd. [6, 9, 12]

Vrstvená architektura se dělí do těchto čtyř vrstev:

1) **Uživatelské rozhraní („User Interface Layer“)** slouží pro interakci uživatele se systémem.
2) **Aplikační vrstva („Application Layer“)** řídí transakce, koordinuje běh aplikace a v neposlední řadě vytváří a přistupuje k doménovým objektům.
3) **Doménová vrstva („Domain Layer“)** popisuje stav a chování *domény*.
4) **Infrastrukturní vrstva („Infrastructure Layer“)** podporuje všechny ostatní vrstvy skrze repositáře, frameworky atd.

Klíčem je izolace vrstev. Někdy se používá šestiúhelníková architektura („Hexagonal Architecture“), lidově řečeno architektura cibule. Šestiúhelníková architektura je znázorněna na obrázku číslo 9. [6]



Obrázek 9 - Šestiúhelníková architektura [13]

2.3.5 Entity

Entita je objekt, který je jednoznačně identifikovatelný podle unikátního ID. Nese konzistentní a proměnlivé hodnoty. *Entita* je perzistentní jako řádek v databázi. Převážná většina entit obsahuje byznys data. [6, 10]

2.3.6 Objekty reprezentující hodnotu (Value object)

Objekty reprezentující hodnotu se někdy uvádí pod názvem Doménové primitivy. Objekt reprezentující hodnotu je objekt, který obsahuje atributy, ale nemá oproti *entitě* unikátní identitu. Jedná se o rozšíření datových typů a řešení byznys logiky se skrývá uvnitř těchto speciálních objektů. S objekty reprezentující hodnotu by se mělo zacházet jako s neměnnými, což znamená, že pomocí konstruktoru se vytvoří daný objekt a po modifikaci by se objekt měl zahodit a vytvořit znovu přes konstruktor. Objekt by měl být vláknově bezpečný. Níže je vidět standardní kód z vývoje. [6, 9]

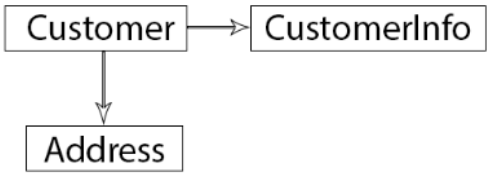
```
public class User {
        private String name;
        private String email;
        private String mobile;
        ...
}
```

Výše zmíněný kód nesplňuje doménovou primitivu. Datový typ String není dostačující a je třeba dodefinovat datové typy jako Name, Email, PhoneNumber. V těchto datových typech je řešena jejich logika, validace a další. Níže je uvedena správná ukázka doménové primitivu.

```
public class User {
        private Name name;
        private Email email;
        private PhoneNumber mobile;
        ...
}
```

2.3.7 Agregát (Aggreagete)

Agregát je seskupení nějakých *entit*, které dodržují *ohraničený kontext*. *Agregát* efektivně snižuje počet vazeb mezi *entitami*. *Agregát* mívá kořenovou *entitu* a je konzistentní, což znamená, že případné změny se promítnou všude v daném *ohraničeném kontextu*. *Agregát* je použit ještě předtím, než se *entita* stane příliš složitou. Pro přístup k jednotlivým *entitám* je jedině přes kořenovou *entitu*. K jednotlivým metodám se předává pouze *agregát* neboli také kořenová *entita*. Příkladem je zákazník (Customer), který je kořenovou *entitou*, ke které se přidává *entita* zákazník informace (CustomerInfo) a adresa (Address). Tento příklad je možné vidět níže na obrázku číslo 10. [6, 10]



Obrázek 10 - Příklad agregátu. Zdroj: autor

2.3.8 Repositář (Repository)

Repositář slouží pro ukládání, aktualizaci, mazání *entit* nebo *agregátů* do databáze nebo jiného uložiště. Repositář je možné také využít pro zpětné získávání *entit* a *agregátů*. V případě *agregátu* složeného z více *entit* by měl vést k ukládání pouze přes *agregát*. [6, 10]

2.3.9 Továrny (Factory)

Továrny slouží pro vytváření *agregátů* a *entit*. Pokud je *agregát* nebo *entita* příliš složitá, pak se vše předává továrně. Pokud *entita* potřebuje ještě jiné zdroje, pak i ty jsou také předány továrně, neboť zodpovědnost již připadá na ni, nikoliv na *entitu*. Jiné zdroje reprezentují například informace z třetí strany. Při implementaci kódu je dobré použít návrhový vzor *tovární metoda* (Factory method). [6, 10]

2.3.10 Implementace nástroje pro taktický návrh

Doménový a *techničtí experti* spolu úzce spolupracují při návrhu modelu. Navrhování modelu probíhá pomocí skupinové kreativní techniky, tzv. brainstormingu. Někteří *doménoví* a *techničtí experti* používají při brainstormingu nálepky. Nálepky jsou od sebe barevně odlišeny, aby bylo jasné, k čemu patří. Po zhlédnutí nálepek je zřetelné, že modrá nálepka patří do událostí, žlutá do *agregátů* a *entit*, zelená do rolí atd. Tyto nálepky jsou postupně lepeny na bílou tabuli, čímž se zaznamenávají jednotlivé iterace mezi nimi. Příklad užití nálepek bude demonstrován na sledování zboží. Při sledování zboží je zaznamenáváno několik událostí: doručovatel dostal žádost, doručovatel akceptoval žádost, doručovatel vyexpedoval zboží, doručovatel doručil zboží. Zmíněné události se nalepí na tabuli modrou nálepkou. Již zde je vidět sekvenční průběh událostí. Obdobně je „nálepkový postup“ aplikován na *entity* a *agregáty*, jejichž nálepky mají žlutou barvu. Důležité je poznamenat, že nejdříve jsou řešeny zajímavé události, které jsou následně „obaleny“ *agregáty*, *entitami*, uživatelskými rolemi, příkazy atd. Mezi zajímavé události například patří doručovatel dostal žádost, který brán jako z hlavních bodů událostí. Po vymodelování všech těchto událostí a příslušných věcí se začíná hledat výše zmíněný *ohraničený kontext*. Tato technika je použita pro modelování softwaru firmy XDeliverer v kapitole 5.2 [6, 11]

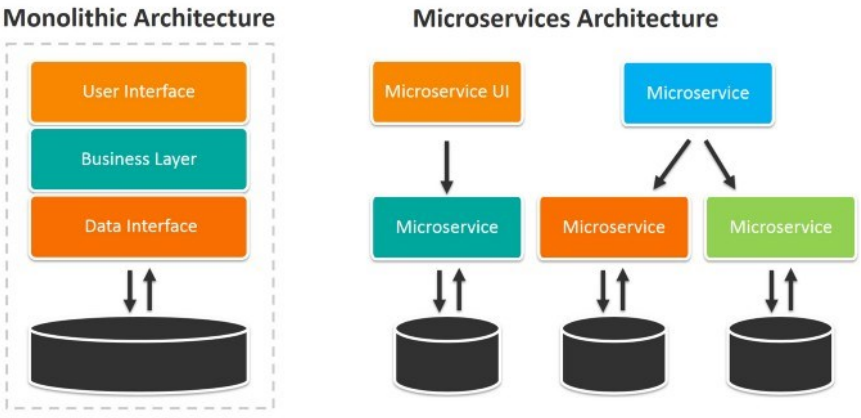
3 ARCHITEKTURA

Následujících podkapitolách jsou rozebrány dvě nejdůležitější architektury, kterými se tato diplomová práce zabývá, a to: monolitická architektura a mikroservisní architektura. V dnešní době stále převažují monolitické aplikace a bude tomu i tak nadále. Mikroservisní aplikace mají smysl pouze u velkých projektů, které tomu odpovídají svými nároky. Dobrým příkladem mohou být banky, firma Amazon a obecně řečeno velké korporátní systémy.

3.1 Monolitická architektura

Monolitická architektura je implementována jako jedna aplikace. Aplikace běží na jedné platformě a je sestavena z vrstev uživatelského rozhraní, byznys vrstvy a data rozhraní. Uživatelské rozhraní slouží k interakci člověka s počítačem. Byznys vrstvu tvoří sada pravidel, která je dána od klienta v jeho funkční specifikaci. Data rozhraní je manipulace s daty od uložení, mazání, načtení atd. [15]

Výhodou tohoto přístupu je, že je snadný na implementaci a řízení skrze tým. Monolitická architektura je jednoduchá na pochopení a otestování. Její nevýhoda spočívá v obtížné údržbě při velikostním nárůstu aplikace a v náročné přenositelnosti aplikace mezi různými platformami. Dalším omezením monolitické aplikace je škálovatelnost, kterou lze dělat jen do určité míry. Jedna aplikace je napsána na jedné platformě. Vždy je sestavena jen jedna verze aplikace, a ta je následně i nasazena na server. Srovnání monolitní architektury a mikroservisní architektury, která je blíže popsána v následující kapitole, lze vidět na obrázku číslo 11. [15]



Obrázek 11 - Monolitní architektura vs Mikroservisní architektura [16]

3.2 Mikroservisní architektura

Myšlenkou mikroservisní architektury je rozdělit aplikaci do malých servis, které vzájemně komunikují mezi sebou, namísto vytvoření jediné monolitické aplikace. Oproti monolitické aplikaci je zde využíván jiný přístup k databázi. Monolitické aplikace mají svoje vlastní databáze napříč celou aplikací. Narozdíl od toho u mikroservisní architektury má každá mikroservisa vlastní databázi. [15]

Z kapitoly pojednávající o DDD jsou čtenáři známy veškeré základní pojmy, tedy co je *doména*, *subdoména*, *ohraničený kontext* atd. Domain-driven design tvoří v architektuře mikroservis významnou roli, a to že dobře odizoluje dané *ohraničené kontexty domény*. *Doména* tvoří mikroservisu, která má na starost jednu určitou oblast aplikace.

Hlavní výhodou mikroservisní architektury je škálovatelnost. Jednotlivé mikroservisy je možné spustit vícekrát a tím umožňují zvládat větší zátěže. Vše funguje pouze pod podmínkou, pokud je správně vymodelovaná doména.

Mezi přední firmy, které užívají mikroservisy patří: Amazon, Netflix, Twitter, PayPal, EBay, Target, LinkedIn atd.

3.3 Porovnání monolitické vs mikroservisní architektury

Níže jsou uvedeny výhody a nevýhody monolitické a mikroservisní architektury.

Monolitická aplikace

Výhody:

- Jednoduchá na vývoj, řízení a nasazení.
- Jednoduchá na pochopení a následné otestování.
- Vhodné pro malé nebo střední aplikace.
- Do určité míry možná škálovatelnost.

Nevýhody:

- Každou verzí aplikace roste i velikost kódu. Při delším vývoji se stává kód hůře udržitelným.
- Platforma / Framework omezení.
- Zpomaluje se vývoj s postupem rozrůstající aplikace.
- Při velké zátěži zpomaluje rychlost obsluhy jednotlivých požadavků.

- Zdroje nejsou optimalizovány („Load balancing“).
- Škálovatelnost je obtížnější s rostoucí aplikací.
- Velikost aplikace může zpomalit spouštění.
- Při každé aktualizaci se nasazuje celá aplikace.

Mikroservisa

Výhody

- Snáze pochopitelné jako jedna mikroservisa.
- Rychlejší vývoj a lehčí pro udržitelnost.
- Nezávislé nasazování mikroservis.
- Snazší škálovatelnost mikroservis.
- Možnost psaní každé mikroservisy na vlastní platformě.
- Každá mikroservisa vlastní instanci databáze. Výhoda tkví v tom, že databázové schéma může být přizpůsobeno k dané mikroservise.
- Podpora agilního vývoje. Jeden tým se stará o jednu mikroservisu.

Nevýhody

- Hůře pochopitelné jako celek.
- Architektura mikroservis je distribuovaný systém a je potřeba implementovat mechanismus pro komunikaci mezi jednotlivými mikroservisemi. Ať už přes zasílání zpráv nebo RPC („Remote procedure call“).
- Co mikroservisa, to vlastní databáze. Problém nastává v dodržení konsistence dat napříč mikroservisemi. Když například mikroservisa A aktualizovala informace o zákazníkovi, pak je potřeba promítnout tyto změny i do mikroservisy B.
- Testování mikroservisní aplikace je složitější. Mikroservisy jsou mezi sebou různě provázány. Testují se pouze samostatné, nezávislé mikroservisy. Komunikace s ostatními mikroservisami se odizoluje a připraví se falešné odpovědi od ostatních mikroservis.
- Závislost mezi mikroservisami. Jedna mikroservisa čeká na novější verzi jiné mikroservisy. Příkladem může být změna datové struktury uživatele, kterému jsou

přidány informace o trvalém bydlišti. Na tuto změnu čeká jiná mikroservisa, než se změna nasadí a bude s ní moci pracovat. Tyto změny je potřeba řádně proplánovat.

- Složitější nasazování. Každá instance mikroservisy musí být nakonfigurována a monitorována. Je zde problém složitější konfigurace průběžné integrace („Continuous integration“).

4 SPRING FRAMEWORK

Spring framework vznikl jako framework nad JavaEE (Java Enterprise Edition), který působí především v oblasti webových služeb. Působí od roku 2002 a prošel velkým vývojem. Spring framework se rozděluje do tří sekcí: Spring Boot, Spring Cloud a Spring Cloud Data Flow. [17]

4.1 Spring Boot

Spring Boot pomáhá k rychlému sestavení aplikace s minimem konfigurace. Programátor se může zaměřit jen na byznys implementaci a nemusí řešit do detailu technickou část například: zpracování požadavků, obalení odpovědi, správa databáze atd. Samozřejmě v některých případech musí krajně řešit i technickou část. Jinak veškeré technické zpracování programátorovi obstarává Spring Boot. Výhodou je zejména jednoduchá konfigurace zabezpečení a bohatá podpora SQL a NoSQL. Dalším kladem je také podpora serverů Tomcat, Jetty atd. Další předností je možnost stažení jen potřebných modulů, a ne celého Spring Frameworku, podpora nástrojů jako metriky, trasování a aktuálního stavu aplikace. Spring Boot je bohatě podporován v populárních IDE (Spring Tool Suite, IntelliJ IDEA a NetBeans). Předchůdcem Spring Boot byl Spring, kde veškeré beany a konfigurace byly prováděny v XML souboru. Se Spring Bootem přišla jednoduchá konfigurace skrze anotace a application.yml. Konfigurační soubor application.yml v sobě nese veškerou konfiguraci aplikace. Výhodou tohoto konfiguračního souboru je, že může provést jakékoliv změny bez opakovaného nasazení aplikace. Pro aplikování změn stačí pouze restartovat aplikaci s novým nastavením v application.yml. [17]

Součástí Spring Boot je tzv. Spring Initializr. Spring Initializr je webové rozhrání, které slouží pro jednoduší založení a konfiguraci startovacího projektu. Jsou zde nabídky:

- **Projekt** – Maven nebo Gradle. Oba slouží jako nástroj pro sestavení aplikace a stažení dependecy tří stran.
- **Jazyk** – Probíhá volba programovacího jazyku mezi Java, Kotlin a Groovy.
- **Spring boot –** Volba verze Spring Boot.
- **Projekt metadata –** Vyplňují se veškerá metadata o projektu. Včetně verze programovací jazyku a finálního sestavení aplikace do Jar nebo War.
- **Závislost** – Sem spadají veškeré moduly od Springu. Každý modul má na starost jednu vlastnost. Například sem spadá Spring Data JPA, který udržuje data perzistentní a využívá k tomu framework Hibernate. Dále jsem patří například Spring Security, Spring Web Starter atd.

Základní roli ve Spring Boot hraje Inversion of Control a Dependency Injection. Nadále jsou zmíněny níže důležité výrazy a anotace.

Inversion of Control

Inversion of Control (IoC) lze také jinak nazvat obracené řízení. Inversion of Control je návrhový vzor, který umožní uvolnit vazby mezi komponentami. V praxi to znamená, že v klasickém programování se vytvoří třída, která je v několika směrech závislá na jiné třídě, což vede k obtížné změně kódu a následně i údržbě. V neposlední řadě značně komplikuje testování kódu. Inversion of Control uvedené problémy odstraňuje pomocí Dependency Injection (DI) a dodává danou instanci třídy externě. [18]

Dependecy Injection

Dependency Injection, ve zkratce DI, je mechanismus, kdy dochází k tomu, že se do vybrané třídy vkládá další potřebná instance třídy. Mezi jednotlivé typy DI patří: Property inject, Constructor inject, Setter inject. [18]

Property inject – Využívá java reflexe, a tak Property inject dosadí odpovídající instanci.

```
@Autowired
private UserService userService;
```

Constructor inject – Do konstruktoru jsou dosazeny odpovídající instance.

```
private final UserRepository userRepository;
@Autowired
public UserService(UserRepository userRepository){
        this.userRepository = userRepository;
}
```

Setter inject – Do setteru methody je vložena odpovídající instance.

```
private UserRepository userRepository;
@Autowired
```

```
public void setUserRepository(UserRepository userRepository){
        this.userRepository = userRepository;
}
```

Nyní už je známo, co je IoC a DI a je možné tak rozvést výrazy bean, IoC kontejner a důležité anotace, které jsou úzce spjaty se Spring frameworkem.

Bean

Bean je objekt, který je zodpovědný za nějakou funkcionalitu. Příkladem může být ukládání dat do databáze, dekodér hesla atd. Každá beana žije v IoC kontejneru po celou dobu běhu aplikace. Beana v IoC kontejneru může být reprezentována podle návrhového vzoru buď jako jedináček, nebo jako prototyp. Jedináček (Singleton) zastupuje pouze jednu instanci v celé aplikaci. Kdykoliv je beana žádána ze Spring kontextu, pak je zpět obdržená stejná instance, která byla vytvořena při spuštění aplikace. U varianty prototyp (prototype) je to obdobné jako u jedináčka, ale s tím rozdílem, že pokud je žádáno o daný objekt, zpět je obdržena nová instance třídy. [18]

IoC kontejner/Spring kontext

Ve Spring frameworku je IoC kontejner často označován také jako aplikační kontext nebo Spring kontext. IoC kontejner uchovává beany a je zaváděn při startu aplikace. V celé aplikaci existuje pouze jeden IoC kontejner. [18]

Spring anotace

- *@Bean* – Anotace *@Bean* vytváří beanu, kde návratová hodnota tvoří typ beany. Název beany je odvozen od názvu metody, pokud není definováno explicitně parametrem *name* u anotace *@Bean.* Dále parametr *scope* u anotace @Bean definuje beanu jako jedináčka nebo prototyp.
- *@Autowired* – Anotace *@Autowired* injektuje beany z IoC kontejneru.
- *@ComponentScan* – Anotace *@ComponentScan* proskenuje dané balíčky, kde hledá anotace *@Controller, @Service, @Repository, @Configuration, @Component* nad třídou. Třídy jsou jednotlivě skenovány a zavedeny do Spring kontextu.
- *@A další.*

Interceptor

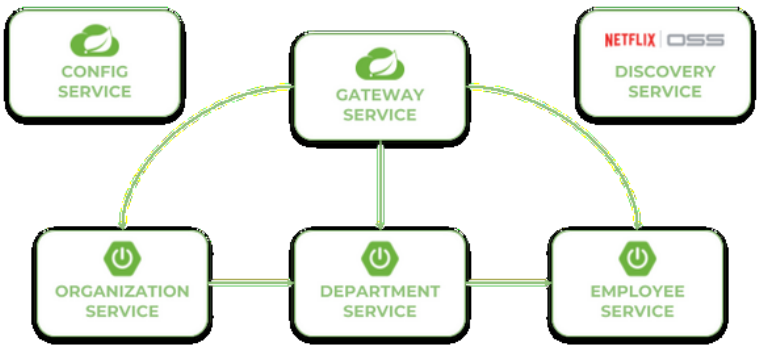
Interceptor je objekt, který může provést operaci s požadavkem od klienta ještě předtím, než dorazí ke kontroléru. V jiném případě může také provést operaci s odpovědí od kontroléru ke klientovi.

4.2 Spring Cloud

Spring Cloud slouží k vybudování distribuovaných systémů a je ilustrován na obrázku číslo 12. Spring Cloud nabízí jednoduchý model pro nejběžnější distribuované vzory. Spring Cloud je postaven na platformě Spring Boot. Spring Cloud model obsahuje Gateway, Discovery, Config a vlastní servisy. Tyto jmenované mikroservisy jsou rozebrány níže. Spring Cloud podporuje OAuth2 zabezpečení, distribuční sledování, Load Balancing, volání se vzájemně mezi mikroservisemi atd. OAuth2 zabezpečení je rozebráno později. S vlastní implementací mikroservis ve Spring frameworku prorazila americká firma Netflix. [17, 19, 20]

Spring Cloud poskytuje Hystrix, který je součástí každé mikroservisy. Hystrix v sobě má mechanismus Circuit Breaker, který poskytuje odolnost vůči poruchám v mikroservisách. Circuit Breaker místo stálého doručování požadavků v krátkých intervalech na specifickou mikroservisu, která následně zapříčiní rostoucí zátěž mikroservisy, nechá chvíli mikroservisu zotavit. [17, 19, 20]

Další součástí každé mikroservisy je Ribbon. Ribbon se stará o „load balancing“. Má na starosti rovnoměrné zatížení mikroservisy. Dynamicky detekuje prostředí na základě integrací s discovery. Ribbon dokáže detekovat, které mikroservisy jsou spuštěny a může také označit mikroservisy, které nejsou spuštěny. Na základě této detekce předchází poruchám. [17]



Obrázek 12 - Spring Cloud rozvržení [21]

Discovery mikroservisa

Discovery mikroservisa se v implementaci od Netlifx uvádí jako Eureka. Discovery mikroservisa registruje všechny mikroservisy ve své zóně, aby mezi sebou mohly vzájemně komunikovat. To řeší problém napevno zadaným hostname a portem. [20]

Gateway mikroservisa

Gateway mikroservisa se v implementaci od Netflix uvádí jako Zuul. Gateway slouží jakožto jeden vstupní bod do distribuovaného systému zvenčí. Veškeré požadavky zvenčí proudí pouze přes mikroservisu Gateway. Díky Gateway je vyřešen problém s Cross-Origin Resource Sharing (CORS). Gateway jakožto vstupní bod obslouží požadavek a přes Gateway se vrací odpověď zpět. Tím pádem nedochází k zaměňování domén. Gateway má k dispozici směrování, které obsahuje cestu a id-mikroservisy. Gateway se na základě požadavku, který přijde, podívá do směrování a nalezne shodu cest. Po shodě cest získá odpovídající id-mikroservisy. Celý požadavek s id-mikroservisy přepošle mikroservise Discovery, která informaci přepošle koncové mikroservise pod id-mikroservisy. [20]

Config mikroservisa

Config mikroservisa zajišťuje externalizovanou konfiguraci v distribuovaném systému na straně serveru a klienta. Jinými slovy Config mikroservisa dokáže stáhnout za běhu změny konfigurace z daného zdroje, jako je například Git, a poslat je odpovídající mikroservise. Cílová mikroservisa za běhu dokáže změnit konfiguraci. [22]

4.3 Spring Cloud Data Flow

Spring Cloud Data Flow umožňuje vytvářet a organizovat datové pipelines[1] pro běžné případy použití, jako je příjem dat, analýza v reálném čase a import / export dat. Také umožňuje jednoduché provázání systémů mezi sebou. Spring Cloud Data Flow podporuje zpracování dat v reálných proudech nebo dávkách. Dále provozuje řídící panel, kde je možné nalézt metriky, kontroly stavu a vzdálenou správu. V neposlední řádě podporuje Cloud Foundry, Kubernetes atd. [17, 23]

[1] Pipelines je sada instrukcí, kde vstup jednoho toku dat přechází do další instrukce.

5 XDELIVERER

Následující podkapitoly se zabývají aplikováním DDD a mikroservisní architektury pro fiktivní firmu XDeliverer. Rozsah zadání odpovídá velikosti diplomové práci, a to pro celkové zjednodušení, snadnější pochopení celého postupu a lehčí orientaci v projektu. Frontend část je zahrnuta pouze do fáze analýzy a návrhu. V následujících kapitolách je jsou využity znalosti z DDD, mikroservisní architektury, Spring Bootu a Spring Cloudu.

5.1 Zadání

Fiktivní firma XDeliverer se zabývá přepravou zásilek. Firma má několik poboček po celé České republice, které přijímají zásilky a následně přepravují mezi bodem/městem/vesnicí A a bodem/městem/vesnicí B. Firma funguje na podobné bázi jako je firma Česká pošta, PPL nebo DHL. Fiktivní firma XDeliverer má potencionál prorazit v zahraničí a rozšířit se tak celosvětově. Očekává velkou zátěž na informační systém (dále jako IS) s kterou je potřeba počítat do budoucna. XDeliverer vytvořila konkurz na vývoj softwaru jako IS s webovými stránkami.

Od IS se očekává správa nad objednávkami, zákazníky, sledování aktuální polohy zásilky atd. Vytvoření objednávky probíhá přes vyplněný formulář od zákazníka na webové stránce firmy nebo zadáním do informační systému zaměstnancem XDeliverer na počítači pobočky X-Pointu. X-Point je pracovní název pro pobočku. Na X-Pointu pracuje zaměstnanec, který vytváří nové objednávky, přijímá platby nebo přijímá a validuje zásilku. Pokud zákazník vytvářel objednávku přes webové stránky, tak má možnost platby přes platební bránu nebo může uhradit objednávku až na pobočce. Vygenerovaný QR kód, který slouží pro zjednodušení manipulaci a identifikování zásilky, nalepí na zabalenou zásilku buď zákazník, nebo zaměstnanec XDeliverer. QR kód je generován informačním systémem. Po přijetí zásilky je balíček přepraven ke koncovému bodu, a to kurýrem firmy.

Objednávka obsahuje veškeré informace o uživateli, adrese doručení, platbě, dále informace o zásilce a aktuální stav objednávky. Informace o zásilce obsahuje nejen údaje o rozměrech, ale i také o poloze aktuální lokace a QR kód.

5.2 Fáze analýzy a návrhu

Během této fáze probíhá fiktivní brainstorming, na jehož základě je udělána analýza a návrh pomocí barevných lístečků, viz obrázek číslo 13.

Legenda:

- Žluté lístečky – *entity a agregáty.*
- Zeleno-žluté lístečky – uživatelské role.
- Oranžové lístečky – příkazy.
- Růžové lístečky – události.
- Zelené lístečky – *domény*.
- Modré lístečky – *ohraničený kontext*.

Obrázek 13 - DDD modelování. Zdroj: autor

Prvotní analýza a návrh

Při modelování je postupováno následovně. Nejprve jsou zachyceny důležité události domény, mezi které patří: registrace zásilky, platba za přepravu zásilky, přijetí zásilky na X-Point a její zaevidování, transport zásilky, doručení zásilky.

K jednotlivým doménovým událostem se připojily entity a agregáty:

- Registrace zásilky – Zákazník, Zásilka, Doručovací adresa.
- Platba za přepravu zásilky – Faktura, Platba.
- Přijetí zásilky na X-Point – Aktuální lokace.

V jednotlivých doménových událostech vystupují role:

- Registrace zásilky – Zákazník, Zaměstnanec X-Pointu.
- Platba za přepravu zásilky – Zákazník, Zaměstnanec X-Pointu.
- Přijetí zásilky na X-Point – Zaměstnanec X-Pointu.
- Transport zásilky – Kurýr.
- Doručení zásilky – Kurýr.

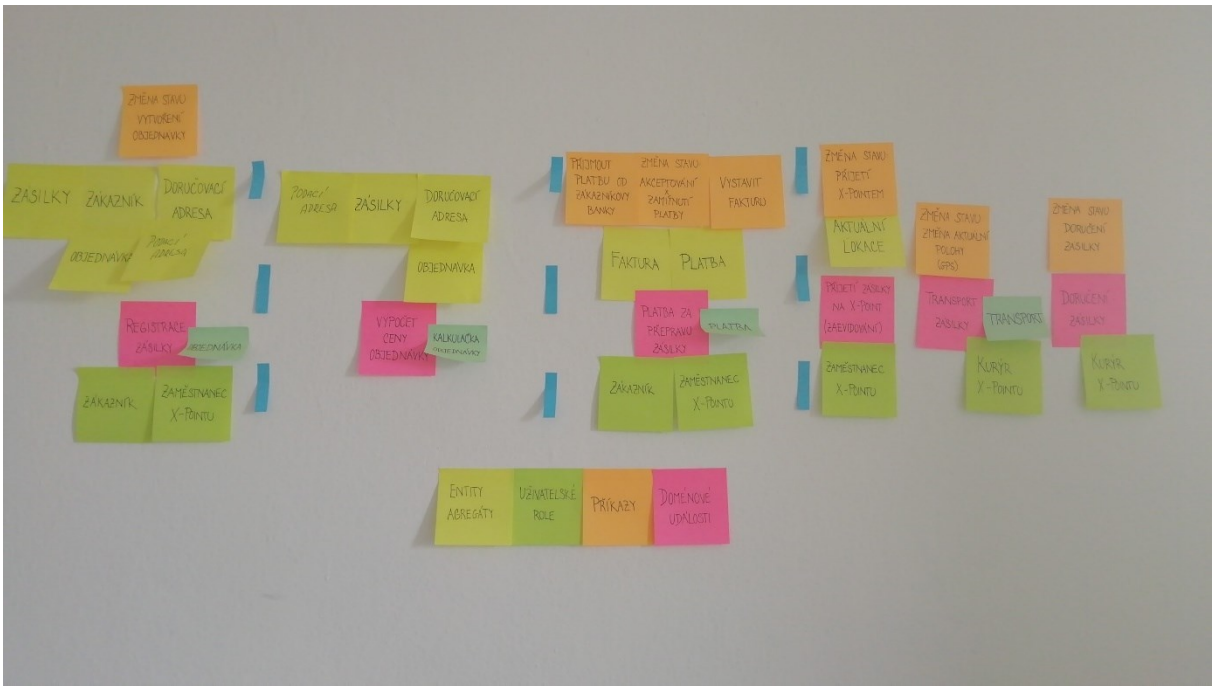
K jednotlivým doménovým událostem se vykonávají příkazy:

- Registrace zásilky – Změna stavu zásilky: vytvoření zásilky.
- Platba za přepravu zásilky – Změna stavu zásilky: akceptování nebo zamítnutí platby.
- Přijetí zásilky na X-Point – Změna stavu zásilky: přijetí zásilky na X-Point.
- Transport zásilky – Změna stavu zásilky: změna aktuální polohy zásilky.
- Doručení zásilky – Změna stavu zásilky: doručení zásilky nebo nezastižení koncového zákazníka.

V neposlední řadě je určen *ohraničený kontext* a přiřazen název *domény*. První *doména* pojmenováno jako Kontrakt obstarává pouze věci spojené se zásilkou, a to vytváření, smazání, modifikace zásilky včetně evidence balíků. Druhá *doména* s názvem Platby nese zodpovědnost za platby a informace o nich. Dále obstarává komunikaci s platební bránou a zpracovává odpověď z platební brány. Třetí *doména* Evidence akceptuje zásilku, popřípadě upraví nesprávné hodnoty zásilky, co se rozměrů týče. K zásilce navíc přidává informace o aktuální poloze. Poslední *doména* Dodávka je spojena s předáním kurýrovi a následnou dopravou k cílovému zákazníkovi.

Úpravy modelu

Po absolvování dalšího fiktivního brainstormingu je potřeba upravit *model*. Je potřeba nalézt lepší názvy vystihující *domény*, *entity*, *agregáty* atd. nebo přidat novou *doménu*, *entity* atd. Na obrázku číslo14 je upravený model XDeliverer.



Obrázek 14 - Přemodelovaný model XDeliverer. Zdroj: autor

Domény:

Kontrakt

Doména je přejmenována na Objednávka. Je přidán nový *agregát* Objednávka, pod nějž spadají informace o zákazníkovi, doručovací adrese, podací adresa a zásilkách. *Entita* zásilka byla přejmenována na Zásilky, jelikož objednávka může obsahovat více zásilek.

Kalkulačka objednávky

Nově vytvořena *doména* Kalkulačka objednávky. *Doména* poskytuje funkcionalitu výpočet ceny objednávky na základě zásilek, podací adresy a doručovací adresy.

Platby

Doména je přejmenována na Platba ke konsistenci názvů.

Evidence

Doména je zrušena a její vlastnosti se přesunuly do *domény* Dodávka. Jiné *domény* už částečně obsahovaly vlastnosti *domény* Evidence. Například přijetí zásilky se promítne v *doméně* Objednávka. Aktuální lokace spadá více pod *doménu* Dodávka.

Dodávka

Doména přejmenována na Transport. *Doména* zavádí *entitu* aktuální poloha.

5.2.1 Doménový slovník

V rámci každé domény je veden slovník. V tomto slovníku jsou nalezeny jednotlivé výrazy včetně popisku, které se vyskytují v dané *doméně*. Slovník slouží pro lepší komunikaci mezi *doménovými* a *technickými experty*. Jednotlivé výrazy jsou používány v rámci komunikace, dokumentace atd. a nejsou zaměňovány s jinými výrazy. V rámci *domén* se mohou vyskytnou duplicitní názvy, ale s jiným popiskem, což nelze považovat za chybu. V jiné *doméně* může mít stejný termín jinou specifikaci.

Objednávka

- **Zákazník –** Osoba, která si objedná přepravu zásilky z bodu A do bodu B. Zákazník je role uživatele.
- **X-Point –** X-Point je jiný název pro pobočku XDeliverer.
- **Zaměstnanec X-Pointu –** Zaměstnanec pracující na X-Pointu. Zaměstnanec X-Pointu je role uživatele. Tato osoba může vytvářet objednávky na základě žádosti od zákazníka.
- **Objednávka –** Objednávka může být vytvořena na základě formuláře od zákazníka nebo zaměstnance X-Pointu. Objednávka je *agregát* obsahující informace o zákazníkovi, ceně, měně, doručovací adrese, kolekci zásilek, stavu objednávky, datumu vytvoření a poslední modifikace.
- **Zásilky –** Zásilky jsou věci, která jsou přepravovány. Zásilka je *entita* obsahující informace o šířce, výšce, délce a váze zásilky. Jedna objednávka může obsahovat více zásilek.
- **Podací adresa –** Podací adresa je místo, kde jsou dané zásilky přijaty. Může se jednat o pobočku X-Pointu nebo bydliště zákazníka atd.
- **Doručovací adresa –** Doručovací adresa je adresa, kam je daná zásilka doručena. Doručovací adresa je součástí objednávky.

Kalkulačka objednávky

- **Objednávka –** Objednávka obsahuje zásilky pro výpočet konečný ceny.
- **Zásilka –** Zásilka je věc, která je přepravována. Zásilka je entita obsahující informace šířka, výška, délka a váha zásilky. Jedna objednávka může obsahovat více zásilek.
- **Podací adresa –** Podací adresa je místo, kde jsou dané zásilky přijaty. Může se jednat o pobočku X-Pointu nebo bydliště zákazníka atd. Podací adresa je součástí objednávky a je zohledněna v ceně.

- **Doručovací adresa** – Doručovací adresa je adresa, kam je daná zásilka doručena. Doručovací adresa je součástí objednávky a je zohledněna v ceně.

Platba

- **Zákazník –** Osoba, která je zodpovědná za platbu objednávky.
- **X-Point –** X-Point je pracovní název pro pobočku XDeliverer.
- **Zaměstnanec X-Pointu –** Zaměstnanec pracující na X-Pointu. Zaměstnanec X-Pointu je role uživatele. Zaměstnanec přijímá platby typu hotovost nebo platby kartou na pobočce.
- **Objednávka –** Objednávka je vytvořena na základě formuláře od zákazníka nebo zaměstnance X-Pointu. Objednávka obsahuje informace o zákazníkovi, ceně, měně, doručovací adrese, kolekci zásilek, stavu objednávky, datumu vytvoření a poslední modifikace.
- **Platba –** Platba je *entita*, která vede informace o objednávce, zákazníkovi, ceně, měně, datum zaplacení, poslední modifikaci, typu platby, platební informaci a úspěchu platby.
- **Platební informace –** Informace při platbě online.
- **Typ platby –** Typ platby je informace o platbě, a to jakou možností byla provedena. Mezi typy platby spadá: hotovost, bankovní převod, online platba a platba kartou na X-Pointu.
- **Hotovost** – Platba objednávky na X-Pointu.
- **Bankovní převod** – Připsání určité částky na učet XDeliverer firmy**.**
- **Online platba –** Platba přes platební bránu.
- **Platba kartou na X-Pointu –** Platba přes terminál na pobočce X-Pointu.
- **Zásilka –** Zásilka je věc, která je přepravována.

Transport

- **Zákazník –** Osoba, která je zodpovědná za platbu objednávky.
- **Zaměstnanec X-Pointu –** Zaměstnanec pracující na X-Pointu. Zaměstnanec X-Pointu přijímá zásilky na X-Point.
- **Kurýr** – Osoba, která je zodpovědná za přepravu zásilky.
- **Aktuální lokace** – Aktuální lokace obsahuje zeměpisnou šířku a výšku, kde se aktuálně nachází zásilky.

5.3 Obecná implementace XDeliverer

V následujících podkapitolách je pojednáváno zvolených technologiích, jsou zde vysvětleny možné způsoby komunikace mezi mikroservisemi, je pojednáno o zabezpečení pomocí OAuth2 protokolu a také je vysvětlena struktura mikroservis v rámci tohoto projektu.

5.3.1 Technologická část

Veškeré mikroservisy jsou napsány v programovacím jazyku Java 8 za pomocí Spring frameworku. Novější Java 8 není volena z důvodu nekompatibility s některými knihovnami a z důvodu obtížnějšího řešení. Je využívána databáze PostgreSQL běžící na Google Cloud Platform. Pro sestavení aplikace a stažení závislostí je využit Gradle. Zabezpečení je prováděno OAuth 2 protokolem za pomocí JWT tokenu. Níže je výčet důležitých použitých dependecy[2].

Spring Boot:

- **spring-boot-starter-actuator** – Actuator slouží pro poskytnutí informací o dané mikroservise. Poskytuje údaje o tom, jestli daná mikroservisa běží, seznam aktuálních bean, informace z logů, využití zdrojů atd. Při přidaní této závislosti se automaticky přidává cesta */actuator*, kde se získají dané informace. Actuator je důležitý pro Admin mikroservisu, která implementuje Spring Boot Admina. Spring Boot Admin sbírá informace o daných mikroservisách a dělá o nich přehled.
- **spring-boot-starter-web** – Dependency pro zjednodušení vytváření webů, včetně REST (Representational State Transfer) aplikací využívajících Spring MVC (Model View Controller). Používá Tomcat jako výchozí vložený kontejner.
- **spring-boot-starter-security –** Security poskytuje API (Application Programming Interface) pro základní zabezpečení aplikace.
- **spring-boot-starter-data-jpa –** Zjednodušuje práci s ukládáním dat do DB pomocí JPA (Java Persistence API) s Hibernatem. Hibernate je ORM (Object Relational Mapping) framework. U ORM[3] se pracuje s objekty, než aby využíval přímo databázi.
- **spring-boot-starter-test –** Poskytuje Api pro napsání základních JUnit testů se Spring boot kontextem.

Spring Cloud:

- **spring-cloud-starter-netflix-zuul –** Poskytuje API pro nastavení gateway mikroservisu.

[2] Dependency jsou závislosti nebo knihovny třetích stran.
[3] ORM je mapování databázových tabulek na entity tříd.

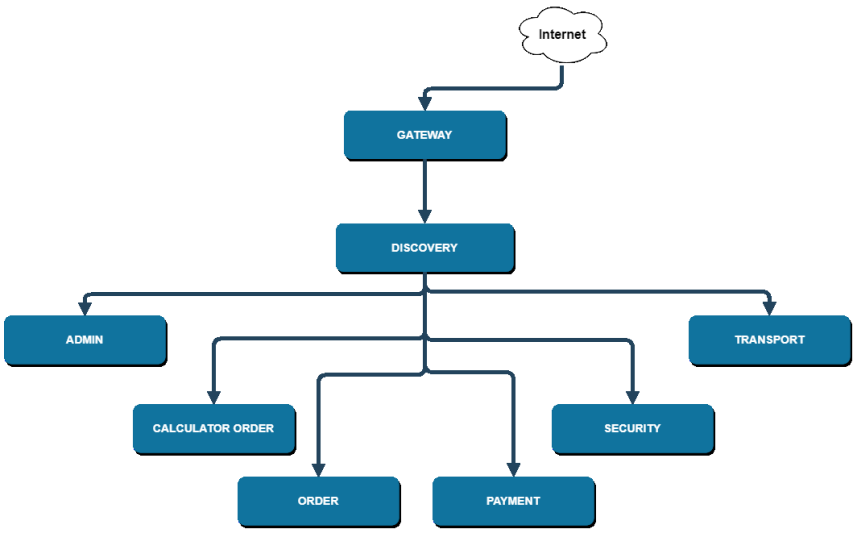
- **spring-cloud-starter-netflix-eureka-server** – Poskytuje API pro nastavení discovery mikroservisy.
- **spring-cloud-starter-netflix-eureka-client –** Poskytuje API pro správnou komunikaci s discovery mikroservisou.
- **spring-cloud-starter-openfeign** – Poskytuje API pro komunikace mezi mikroservisami.
- **spring-cloud-starter-oauth2 –** Poskytuje API pro zabezpečení protokolem OAuth 2.
- **spring-cloud-gcp-starter-sql-postgresql** – Poskytuje API pro komunikaci s PostgreSQL databází.

Ostatní:

- **Liquibase –** Je nástroj pro efektivní vytváření databázových schémat a změn v nich. Liquibase je nezávislý na cílové DB (PostgreSQL, MySQL atd.).
- **Lombok** – Je plugin pro zjednodušené psaní datových tříd. Pomocí anotací vygeneruje odpovídající metody třídy. Například anotace *@Getter* vygeneruje všechny veřejné gettery k privátním atributům. Lombok má velkou výhodu ve výrazném zpřehlednění a zkrácení zápisu kódu.

5.3.2 XDeliverer mikroservisní architektura

XDeliverer aplikace se skládá z devíti mikroservis. Na obrázku číslo 15 je grafické znázornění mikroservis. Později je vysvětlena komunikace mezi mikroservisami, zabezpečení pomocí protokolu OAuth 2 a v kapitole 5.4 jsou vysvětleny jednotlivé mikroservisy.



Obrázek 15 - Diagram mikroservis firmy XDeliverer. Zdroj: autor

5.3.3 Komunikace mezi mikroservisemi

Komunikace mezi mikroservisami je obecně problém, jelikož se jedná o distribuovaný systém a každá mikroservisa běží ve své vlastní instanci. Komunikace mezi nimi probíhá asynchronně. Pro komunikaci mezi mikroservisami jsou možné implementace: Feign Client nebo Google Cloud Pub/Sub. V diplomové práci je využita implementace Feign Client.

Google Cloud Pub/Sub

Google Cloud Pub/Sub je služba poskytovaná Google Cloud Platformou. Tato služba funguje na základě „Message queue“. Zprávy z jedné mikroservisy jsou posílány do služby Google Cloud Pub/Sub. Google Cloud Pub/Sub si jej zařadí do fronty, ze které jsou postupně odebírány a posílány k cílovým mikroservisám. Tento mechanismus garantuje minimálně jedno doručení. Problém nastává v případě, kdy některé zprávy mohou být doručeny vícekrát k dané mikroservise a může tak dojít k nekonzistenci dat. Na tento problém je třeba myslet při implementaci a ošetřit jej.

Feign Client

Feign Client je jednodušší implementace oproti Google Cloud Pub/Sub. Feign Client provolává REST API druhé mikroservisy. Feign Client negarantuje doručení zprávy. Co se týče implementace, pak stačí pouze nakonfigurovat a sestavit rozhraní REST API pro daný zdroj mikroservisy. Níže je ukázka kódu konfigurace Feign Clienta.

```
@Configuration
@EnableFeignClients("com.xdeliverer.payment.client.order")
public class FeignConfig {
    private static final String AUTHORIZATION = "Authorization";
    @Bean
    public RequestInterceptor oauth2FeignRequestInterceptor() {
        return template -> {
        if (SecurityContextHolder.getContext().getAuthentication() != null) {
OAuth2AuthenticationDetails authenticationDetails = (OAuth2AuthenticationDetails)
                SecurityContextHolder.getContext().getAuthentication().getDetails();
```

```
        template.header(AUTHORIZATION,
                        String.format("%s %s", authenticationDetails.getTokenType(),
                                authenticationDetails.getTokenValue()));
                    }
                };
        }
}
```

Anotace *@Configuration* zavede třídu do Spring kontextu. Anotace *@EnableFeignClients* aktivuje Feign Clienta a jako parametr uvede název balíčku, kde má hledat vytvořené REST API rozhraní jednotlivých mikroservis. Uvnitř třídy FeignConfig je beana oauth2FeignRequestInterceptor. Tento interceptor přibalí do hlavičky požadavku informace o tokenu, který je potřebný pro zpracování další mikroservisou. Níže je ukázka rozhraní REST API na mikroservisu, se kterou je potřeba komunikovat.

```
@FeignClient(name = "order-service", path = "/api/order")
public interface OrderClient {

        @GetMapping("/{id}")
        Order getOrderById(@PathVariable("id") UUID id);

        @PutMapping("/{id}/state")
        Order updateOrderState(@PathVariable("id") UUID id, @RequestBody OrderState state);
}
```

Anotace *@FeignClient* oznamuje existenci rozhraní REST API. Parametr *name* specifikuje cílenou mikroservisu podle id. Specifikování pouze id mikroservisy ušetří problém s url a portem, kde daná mikroservisa běží. Informace, kde koncová mikroservisa běží, si obstará sám Feign Client z discovery mikroservisy. Také tato anotace aktivuje load balancing od Ribbonu. Parametr *path* specifikuje koncovou cestu, pro který je REST API vystaveno. Uvnitř rozhraní se pouze specifikují odpovídající signatury metody pro REST API pro volání koncového bodu mikroservisy.

5.3.4 Zabezpečení pomocí OAuth 2 protokolu

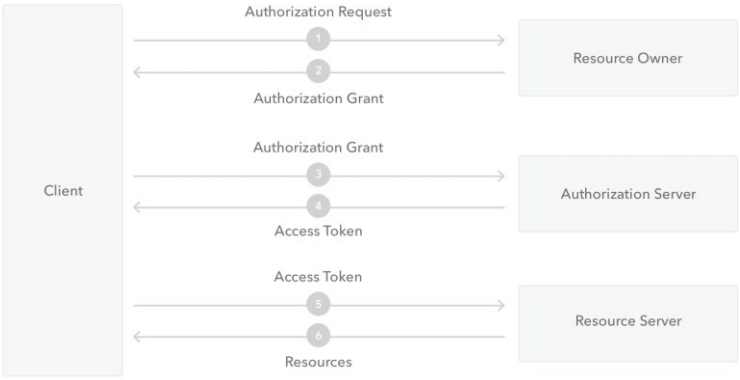
Veškeré mikroservisy jsou zabezpečeny OAuth 2 protokolem, který je blíže popsán níže.

OAuth 2 protokol

OAuth 2 je authorizační protokol, který už je téměř standardem pro zabezpečení REST API. V dnešní době je na mnoho webových aplikací, které využívá ověřování pomocí třetí strany.

Oauth2 role:

- **Client –** *Clienta* tvoří převážně webová aplikace, která má přístup ke zdrojům na *resource serveru* s právy *resource ownera*.
- **Resource owner –** *Resource owner* je uživatel, který vlastní zabezpečené zdroje (data). *Resource owner*, který je schopen přidělit nebo zamítnout přístup k daným zdrojům.
- **Authorization server –** *Authorization server* přiděluje přístupový token, a to poté co úspěšně proběhla autentizace od uživatele (*resource owner*) a autorizace.
- **Resource server –** *Resource server* je server se zdroji (daty). Je schopen na základě přístupového tokenu vydat odpovídající zdroje.



Obrázek 16 - OAuth 2 princip [26]

Na obrázku číslo 16 je znázorněn základní princip OAuth 2. V následujícím textu je postup demonstrován z reálného života na příkladu s přihlášením se do služby/aplikace Spotify pomocí Facebook účtu. Spotify je služba pro streamování hudby a Facebook je sociální síť. *Clienta* zde tvoří Spotify. *Resourcce owner* je uživatel za PC. *Authorization server* je Facebook a *resource server* je také Facebook.

Spotify jako první vyzve *resource ownera* autorizačním požadavkem. *Resource ownerovi* vyskočí okno s přihlášením do Facebooku. R*esource owner* se přihlásí a následně udělí oprávnění, která aplikace žádá. Nazpět je Spotify zasláno autorizační oprávnění. Spotify vezme autorizační oprávnění a pošle ho *authorization serveru* Facebooku. *Authorization server* vygeneruje přístupový token a pošle ho zpět Spotify. Spotify vezme přístupový token a zašle ho *resource serveru* od Facebooku. Tento server na základě rozsahu oprávnění poskytne data. Někdy je *authorization server* a resource *server* stejný, není podmínkou, aby byly odděleny. Poznámka: přihlašovací údaje Facebooku se nikdy nedostanou do aplikace Spotify.

Authorization grant type:

- **Authorization Code** – Autorizační kód je použit při autorizaci na Facebooku. Kde klient žádal o oprávnění uživatele.
- **Implicit –** Podobný princip jako u Authorization Code. Implicit grant type se používá u SPA (Single Page Application), které jsou napsány v JavaScriptu. Při přesměrování uživatele na jinou stránku kvůli autorizaci je možné přibalit aktuální stav aplikace. Poté je přesměrován zpět do SPA a může pokračovat tam, kde skončil.
- **Password –** Uživatel rovnou zasílá přihlašovací údaje pro své ověření na straně autorizačního serveru a následné získání přístupového tokenu.
- **Client Credentials –** Používá se, když jedna aplikace komunikuje s druhou aplikací. Například když se ve tři hodiny ráno pouští automatická úloha na jedné z aplikací a potřebuje zabezpečeně komunikovat s ostatními aplikacemi.

U všech autorizačních požadavků je povinné vyplnit *authorization grant type*, pokud není implicitně volen výchozí nějaký *grant type*. Dále je povinné vyplnit *client id*. Client id zavedená identifikace na straně *authorization serveru*.

Ukázka grant type password

V rámci přístupu k REST API je potřeba se ověřit. Níže je zobrazeno, jak by daný požadavek měl vypadat.

```
Request Headers:
        Authorization:"Basic eGRlbGl2ZXJlcmZmZmY6ZGVsaXZlcmVy"
        ...
Request Body:
        grant_type:"password"
        password:"password"
        username:"customer@example.com"
        client_id:"xdeliverer"
```

Authorization je základní přihlášení k serveru, který obsahuje přihlašovací jméno a heslo. *Grant_type* je typ udělení. *Username* a *password* jsou přihlašovací údaje klienta. *Client_id* je id klienta neboli aplikace. Zpět přijde odpověď s přístupovým tokenem, kterým se dále uživatel prokazuje u *resource serveru*. Spolu s přístupovým tokenu přichází další informace, a to doba trvaní tokenu, typ tokenu atd.


```
{
   "access_token": "...“
   "token_type": "bearer",
   "refresh_token": "...",
   "expires_in": 43199,
   "scope": "read write",
   "user": {
      "id": "808bc4b3-415d-47ea-9e3a-3ff715ddd99e",
      "firstname": "Jakub",
      "lastname": "Žufánek",
      "email": "customer@example.com",
      "phone": "123456789",
```

```
    "role": "CUSTOMER",
    "username": "customer@example.com"
  },
  "jti": "6d656d8f-4cc0-4d6f-b577-fb472467b892"
}
```

JWT (JSON Web Token)

JWT má výhodu, že přenáší důležité informace ve svým tokenu. JWT využívá asymetrické kryptografie. Symetrická kryptografie není zvolena z důvodu bezpečnosti. Asymetrická kryptografie je zde využívána k ověření tokenu. Jestli daný token je vydán důvěryhodným *authorization serverem* a token není pouze podvrhem. *Authorization server* vloží důležitá data vloží do JWT, který podepíše svým privátním klíčem a pošle. Mikroservisa, která JWT přijala ověří podpis veřejným klíčem *authorization serveru*.

JWT je složen ze tří částí oddělených tečkami. Jednotlivé části jsou kódovány base64.

A.B.C

Oblast A obsahuje hlavičku obsahující typ tokenu a algoritmus. V oblasti B jsou vložená důležitá data. Někdy jsou tato data označována jako claims. Oblast C je podpis *authorization serverem*.

Zpracování JWT mikroservisou probíhá následovně:

1) Verifikace, zda JWT obsahuje tři tečky.
2) Rozdělení na hlavičku, data, podpis.
3) Dekódování hlavičky a získání algoritmu.
4) Zkontrolování podpisu.
5) Zpracování dat.

XDeliverer implementace OAuth 2

Client/Aplikace prvně musí získat JWT, aby získal přístup ke zdrojům. *Client* vyšle požadavek s *grant typem password* na Gateway na ověření u *authorization serveru*. Gateway vezme

požadavek a přepošle jej na mikroservisu Security. Mikroservisa Securita je *authorization server* odpovídající OAuth2. Zde klient získá JWT, který mu je zpět poslán přes Gateway. Pomocí JWT je umožněno klientovi přistoupit ke svým zdrojům. V tomto případě k REST API.

Client pošle požadavek o přistoupení ke zdroji objednávek. Požadavek přijde na Gateway. Gateway, jelikož tvoří vstupní bod pro zdroje, tak jsou zde všechny požadavky ověřovány a též mají validní JWT. Poté pomocí směrování přepošle data odpovídají mikroservise. V tomto případě zašle mikroservise Order. Mikroservisa Order validuje JWT znovu. Poté si zde vytáhne potřebná data z JWT pro zpracování požadavku. V případu této implementace jsou to informace o uživateli. Po zpracování požadavku pošle zpět odpověď opět přes Gateway.

5.3.5 Struktura mikroservis

V každé mikroservise v kořenovém adresáři lze nalézt build.gradle, který obsahuje informace o potřebných dependency, konfiguraci sestavení aplikace, verzi Java atd. Dále zde je settings.gradle soubor. Tento soubor obsahuje informace o názvu projektu a případném připojení dalších lokálních dependency.

Každá mikroservisa má následující formát kořenového balíčku com.xdeliverer.[název mikroservisy]. Z kořenového balíčku se připojují tyto baličky:

- **kořenový balíček** – V kořenovém baličku je uvedena třída s metodou *main* a s anotací *@SpringBootApplication*, která startuje kontext Spring Boot.
- **configuration** – Balíček obsahuje třídy, které konfiguruje chování aplikace nebo zavádí různé beany.
- **controller –** Balíček kontrolér obsahuje kontroléry, který vystavují odpovídající REST API mikroservisy.
- **dto –** Balíček obsahuje DTO (Data Transfer Object). DTO jsou použity při zpracování požadavku REST kontroléry.
- **client.[název jiné mikroservisy]** – Obsahuje rozhraní pro Feign Clienta dané mikroservisy.
- **client.[název jiné mikroservisy].dto** – Obsahuje DTO objekty odpovídající rozhrání Feign Clienta dané mikroservisy.
- **domain –** Obsahuje další balíčky, které spadají do konceptu DDD.
- **domain.entity** – Obsahuje *entity* a *agregáty* určité domény mikroservisy.
- **domain.factory –** Obsahuje *továrny* pro vytváření *entit* a *agregátů*.
- **domain.repository –** Obsahuje JPA *repositáře* pro ukládání *entit* a *agregátů* do DB.

- **domain.service –** Obsahuje *servisy domény*.

V *src/main/resources* je konfigurační soubor application.yml a složka db. Application.yml slouží pro konfiguraci springu, připojení DB atd. Složka *db/changelog* slouží pro Liquibase.

složka db/changelog obsahuje:

- **db.changelog-master.yaml –** Kořenový soubor pro Liquabase. Zde jsou uvedeny definice datových typů pro jednotlivé DB a připojení následujících dvou souborů.
- **db.changelog-1.0.yaml –** Soubor s definovaným databázových schématem.
- **db.changelog-default-data.yaml –** Soubor s vloženými daty do DB.

5.4 XDeliverer mikroservisy

V následujících podkapitolách jsou rozebrány důležité vlastnosti a konfigurace jednotlivých mikroservis. V rámci neopakování stejných pasáží jsou zmíněny konfigurace pouze jednou. Například konfiguraci komunikace s Discovery mikroservisou, která je popsána pouze u první podkapitoly Gateway mikroservisa, nebude dále znovu rozebírána, i když je používána i v ostatních mikroservisách.

5.4.1 Gateway mikroservisa

Gateway slouží jakožto vstupní bod do distribuovaného systému. Gateway funguje jako filtr na požadavky a pouští dovnitř pouze autorizované požadavky ke zdrojům (k jednotlivým mikroservisám). Požadavky, které nejsou ověřeny, vyzve k autorizaci. Autorizační požadavky jsou přeposílány na mikroservisu Security.

Třída GatewayApplication

Třída GatewayApplication obsahuje hlavní metodu *main* a tyto anotace:

- *@EnableDiscoverClient* – Anotace konfigurace komunikaci s Discovery mikroservisou.
- *@EnableZuulProxy* – Anotace spouští konfiguraci služby gateway.
- *@SpringBootApplication* – Spouští kontext Spring Boot aplikace.

Třída GatewayConfiguration

Třída dědí od ResourceServerConfigurerAdapter, která umožňuje konfiguraci serveru se zdroji. Navíc třída obsahuje anotace:

- *@Configuration* – Zavede třídu do Spring kontextu.
- *@EnableResourceServer* – Anotace provede konfiguraci *resource serveru*.

Přepsaná metoda *configure* slouží pro filtrování http požadavků. Na základě shody cesty se vzorem je požadavek buď přijat, nebo zamítnut.

```
@Override
public void configure(final HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .antMatchers("/**")
                .authenticated();
}
```

Konfigurační soubor application.yml

Níže v konfiguračním souboru je uveden název aplikace gateway, který slouží jako informace pro mikroservisu Discovery a Admin. Dále je zde port, na kterém mikroservisa naslouchá.

```
spring:
  application:
    name: gateway
server:
  port: 8080
...
```

Dále zde nalezneme směrování. Všechny požadavky s cestou /*ouath/*** budou přeposílány mikroservise Security, která nese id oauth-service.

```
...
zuul:
```

```
  routes:
    oauth:
      path: /oauth/**
      service-id: oauth-service
...
```

Nastavení veřejného klíče pro rozšifrování JWT při zabezpečení OAuth 2.

```
...
security:
  oauth2:
    resource:
      jwt:
        key-value: |
          -----BEGIN PUBLIC KEY-----
          YIvcE43Y7JyFYqnPr/e2gIU+zTOC98WypaT8OjTVwkyg0gqn2hiH2xcapT2YOS5Z…
          -----END PUBLIC KEY-----
...
```

Konfigurace pro komunikaci s Discovery mikroservisou. *LeaseRenewalIntervalInSeconds* nastavuje časový limit pro navázání spojení s Discovery, pokud časový limit vyprší, aplikace se ukončí. *RegistryFetchIntervalSeconds je* perioda, která udává, jak často je posílán ping na Discovery. *DefaultZone* je zóna ve které má Gateway mikroservisa hledat Discovery mikroservisu.

```
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10
  client:
    registryFetchIntervalSeconds: 5
    serviceUrl:
      defaultZone: ${EUREKA_SERVICE_URL:http://localhost:9999}/eureka/
```

Atribut *include* s parametrem *"*"* zpřístupní všechny konečné body Actuatoru, které slouží pro poskytnutí informací o mikroservise. Informace sbírá mikroservisa Admin, která dává přehled nad všemi mikroservisami.

```
management:
  endpoints:
    web:
      exposure:
        include: "*"
```

5.4.2 Discovery mikroservisa

Discovery slouží pouze pro detekci a registrování ostatních mikroservis k sobě. Jedná se o centrální prvek pro komunikaci mezi mikroservisami.

Třída DiscoveryApplication

Třída DiscoveryApplication obsahuje hlavní metodu *main* a tyto anotace:

- *@EnableEurekaServer* – Anotace aktivuje discovery server s odpovídající konfigurací.

5.4.3 Security mikroservisa

Security mikroservisa poskytuje základní správu nad uživateli, vydávání JWT. Je zde implementován autorizační server v rámci OAuth 2 a jejich příslušných služeb.

Třída SecurityApplication

Třída SecurityApplication obsahuje hlavní metodu *main* a tyto anotace:

- *@EnableGlobalMethodSecurity(prePostEnabled = true)* – Anotace aktivuje interceptor pro ověření přístupu k danému REST koncovému bodu.

Třída AuthServerOAuth2Config

Třída dědí od AuthorizationServerConfigurerAdapter. Třída obsahuje anotaci:

- *@EnableAuthorizationServer* – Anotace spustí a provede konfiguraci autorizačního serveru OAuth 2.

Přetížené metody:

- *void configure(final AuthorizationServerSecurityConfigurer oauthServer)* – Metoda kontroluje příchozí tokeny.
- *void configure(final ClientDetailsServiceConfigurer clients)* – Metoda ukládá do kontextu *clienta* s definovanými hodnotami. Tyto informace nadále slouží pro ověřování u OAuth 2 konkrétně s *grant typem password*.

Beany:

- *tokenStore* – Pouze vytváří instanci *JwtTokenStore,* která zpracovává příchozí JWT a získá z nich důležité informace, která ale neukládá.
- *accessTokenConverter* – Získá informace z JWT.
- *passwordEncoder* – Vytváří instanci *BCryptPasswordEncoder*, která slouží pro hashovaní hesel s funkcí BCrypt.

Třída OAuthResourceServerConfig

Třída pouze konfiguruje věci spojené s *resource serverem* a nastavuje *tokenServicu* do konfigurace.

Třída XDelivererAccessTokenConverter

Třída dědí od DefaultUserAuthenticationConverter. Nadále přepíše metodu:

- *Authentication extractAuthentication(Map<String, ?> claims)* – Metoda má na starost extrahovat uživatele, práva atd. z claims JWT.

Třída XDelivererPrincipal

Třída pouze nese informace o uživateli. Tento objekt je dál použit v kontroléru, kde rámci jednoho koncového bodu je možnost získat informace o tom, kdo je právě ověřen. Dále přistupuje ke zdroji dat.

Konfigurační soubor application.yml

Níže je uvedena konfigurace k databázi, kde je potřeba zadat přihlašovací údaje a uživatelské jméno a heslo. Nadále je potřeba uvést url ve formátu *jdbc:[driver]//[hostname nebo ip adresa]/[jméno databáze]*.

```
spring:
  ...
  datasource:
    username: xdeliverer-security
    password: ...
    url: jdbc:postgresql://35.234.148.255:5432/xdeliverer-security
...
```

V rámci připojení ke Google Cloud Platform k SQL DB je potřeba upřesnit tyto informace.

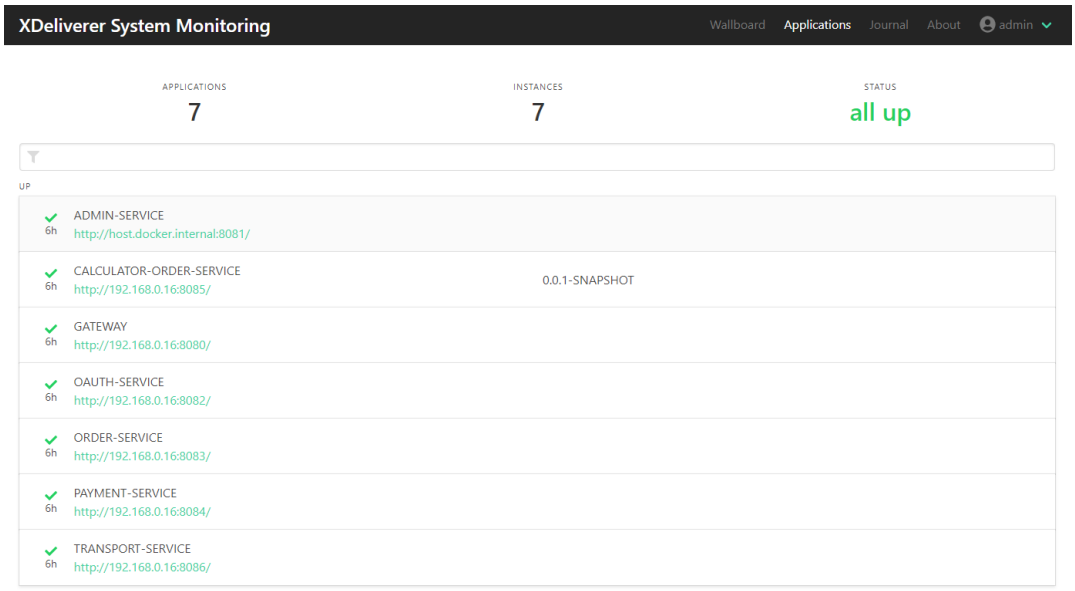
```
spring:
  ...
  cloud:
    gcp:
      sql:
        instance-connection-name: xdeliverer-248911:europe-west2:xdeliverer
        database-name: xdeliverer-security
      project-id: xdeliverer-248911
...
```

U hibernate je potřeba definovat *ddl-auto*, kde varianta *validate* pouze validuje strukturu tabulky a sloupců, a to zda existují nebo ne. Pokud neexistují, hibernate vyhodí výjimku. *Database-platform* určuje cílovou DB.

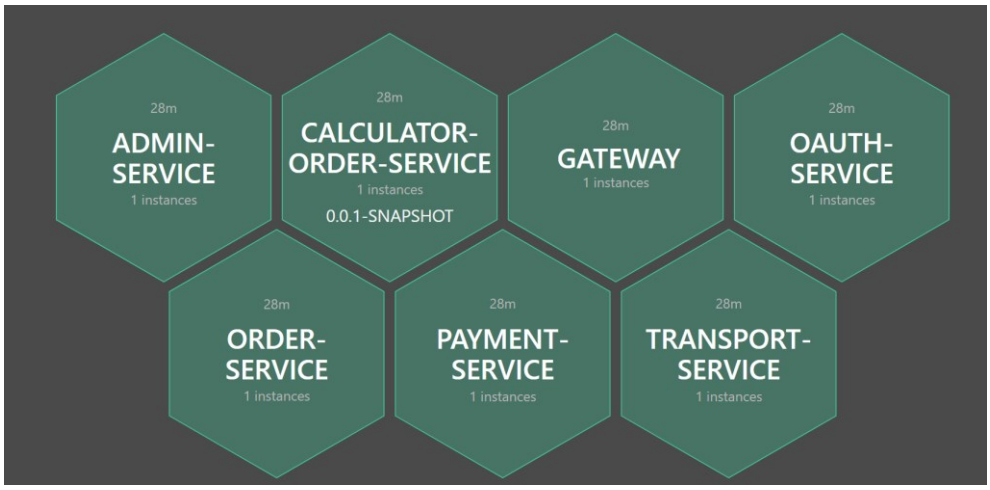
```
  jpa:
    hibernate:
      ddl-auto: validate
      database-platform: org.hibernate.dialect.PostgreSQL9Dialect
```

5.4.4 Admin mikroservisa

Admin mikroservisa implementuje Spring Boot Admin. Spring Boot Admin obstarává přehled informací o aktuálních běžících mikroservisách. Na obrázku číslo 17 a 18 je vidět přehled mikroservis. Kde u jednotlivých mikroservis je možné zpozorovat jejich stav, doba běhu, url adresa a verze.

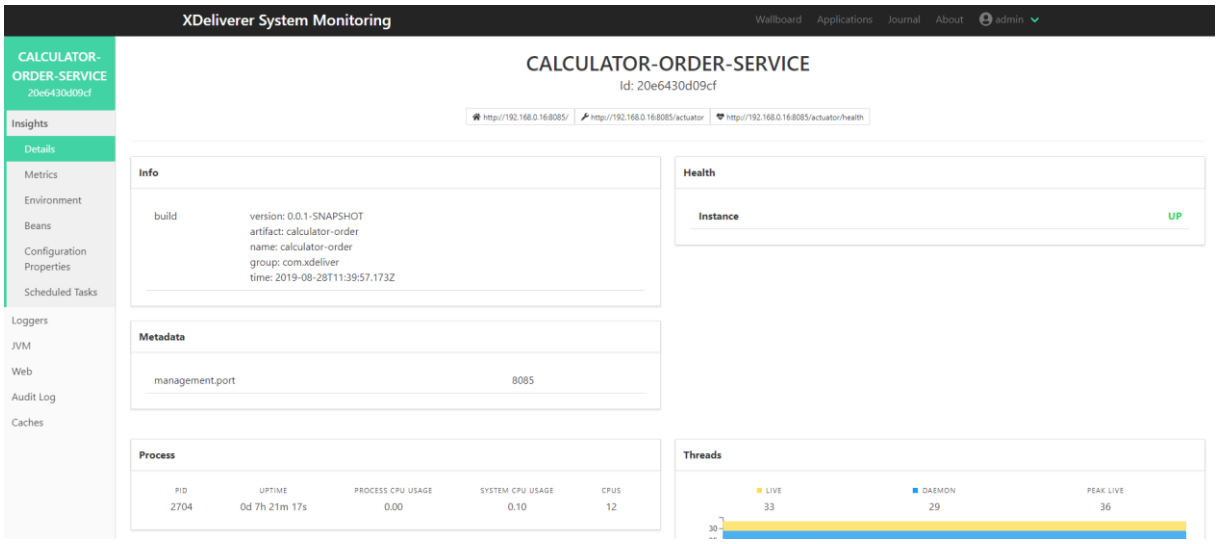


Obrázek 17 - Přehled aktuálně běžících mikroservis. Zdroj: autor



Obrázek 18 - Přehled mikroservis firmy XDeliverer. Zdroj: autor

Spring Boot Admin nadále poskytuje u jednotlivých mikroservis podrobnější informace. Mezi podrobnější informace mikroservisi se řadí: informace o sestavení, aktuální stav, url, využití systémových zdrojů. Pod jednotlivými sekcemi je možnost nahlédnout do: aktuálně běžících bean, aktuálně nastavené mikroservisy, vystavených konečných bodů atd.



Obrázek 19 - Detailní přehled informací o mikroservise. Zdroj: autor

5.4.5 Ostatní mikroservisy

Níže je popsáno, která mikroservisa je za co zodpovědná a zmíněna jejich byznys logika. Implementace a konfigurace se od sebe už moc neliší po technické stránce příliš neliší. V neposlední řadě je potřeba zmínit vlastní knihovna *oauth-resource-server-autoconfigure,* která poskytuje autokonfiguraci skrze OAuth2 pro jednotlivé mikroservisy.

Order mikroservisa

Order mikroservisa spravuje objednávky a zásilky a k tomu vystavené příslušné REST API. V každé objednávce jsou vložené informace o zásilkách. Zákazník přijde vytvoří si svoji objednávku a přidá zásilky do objednávky. Při vytvoření zásilky je stav objednávky *CREATED.* Na základě přidávání zásilek do objednávky je voláno rozhraní REST API od *Calculator order* pro výpočet ceny. Nadále ke každé zásilce lze vygenerovat QR kód, který se nalepí na zásilku pro jednoduší identifikaci zásilky.

Calculator order mikroservisa

Calculator order mikroservisa poskytuje funkcionalitu pro výpočet ceny objednávky. Do výpočtu ceny objednávky je zahrnuta vzdálenost mezi podací a doručovací adresou, váha jednotlivých zásilek. Jednotlivé váhy zásilek spadají do váhové kategorie, která je ohodnocena cenou.

Payment mikroservisa

Payment mikroservisa se stará o platby. Vede informace o platbách, včetně typu platby atd. Nadále komunikuje s mikroservisou Order skrze informace o objednávce. Po úspěšné platbě je

změněn stav objednávky na *SUCCESSFUL_PAYMENT* nebo neúspěšné platbě na *FAIL_PAYMENT*.

Transport mikroservisa

Transport mikroservisa slouží ke sledování zásilky. Po přijetí zásilky zaměstnancem X-Pointu je zaevidována aktuální lokace pobočky do DB a změněn stav objednávky na *ACCEPTED_PACKET*. Po převzetí zásilek kurýrem je sledována poloha kurýra až k dodací adresa zákazníka.

Pomocná dependence oauth-resource-server-autoconfigure

Tato dependence slouží jako autokonfigurace pro mikroservisy Order, Payment, Transport atd. Konfiguruje zabezpečení resource server, který je součástí OAuth2. Následně také pracuje s JWT a získává z něj informace o uživateli.

6 ZÁVĚR

Cílem diplomové práce bylo vytvořit podnikový informační systém postavený na moderní architektuře založené na mikroslužbách (microservice architecture) v podobě několika autonomních spolupracujících komponent.

Dle autora bylo dosaženo všech cílů, které byly očekávány v úvodu práce. Autor si zde rozšířil zkušenosti s návrhem mikroservisní architektury pomocí Domain-driven designu a nabyl hlubších znalostí se Spring frameworkem, konkrétně se Spring Bootem a Spring Cloudem.

Při průběhu zpracování se naskytly problémy s komunikací mezi mikroservisami. Komunikace mezi mikroservisami probíhá asynchronně a je potřeba na toto myslet při implementaci kódu. Jedno z možných řešení je anotace *@Transaction* nad konečným bodem v REST kontroléru. Anotace *@Transaction* při výskytu výjimky udělá rollback nad posledními příkazy do DB. Toto řešení bylo použito v diplomové práci. Jako druhé řešení se nabízí použití Google Cloud Pub/Sub, které zaručuje doručení požadavku k dané mikroservise alespoň jednou. Zde je potřeba ošetřit konzistenci dat při doručení více zpráv stejného požadavků.

Mezi další problémy lze zařadit vypršení času na čekání odpovědi od jiné mikroservisy. Řešení se nabízí pouze ve zvýšení těchto časů.

Mikroservisní architekturu může autor doporučit pro korporátní systémy nebo systémy u kterých lze očekávat velký nárůst požadavků. V jiném případě je lepší využít monolitickou architekturu s vyžitím návrhu podle Domain-Driven designu. Domain-driven design zajistí správný návrh aplikace pro budoucí přechod na mikroservisní architekturu, pokud si to okolnosti vyžádají.

Výhodou/nevýhodou mikroservisní architektury spočívá v duplicitě kódu. Datové třídy v rámci komunikace mezi mikroservisami si vývojář může upravit k obrazu svému. Vývojář může zanechat pouze atributy, které se používají v dané mikroservise. Toto autor řadí mezi výhodu. Nevýhodou je, že duplicitní kód špatně reaguje na změny, což se týká například výše zmíněných datových tříd. Datové třídy je potřeba přizpůsobit k daným změnám ve všech mikroservisách, kde se vyskytují.

Mikroservisní architektura je v rámci návrhu složitý proces a je potřeba dodržet veškeré Domain-driven designové prvky. Je důležité nepodcenit fázi analýzy a návrhu. Autorovi se často stávalo, že při implementaci kódu zvažoval, zda danou problematiku řeší mikroservisa/doména A nebo B. Řešením je přesné stanovení *ohraničeného kontextu*.

Při implementaci mikroservis řeší mnoho technických částí Spring Cloud za vývojáře. Z dokumentace Spring Cloudu je potřeba správně pochopit konfiguraci jednotlivých mikroservis. Konfigurace je v některých částech obtížná.

Jedno z budoucích rozšíření aplikace je o frontendovou část, ať už se jedná o část webového rozhraní nebo formou mobilní aplikace. Nadále by se mohla rozšířit sada jednotkových a integračních testů. Dále také by mohlo dojít k zavedení mikroservis do Google Cloud Platform nebo jiného cloudového řešení.

7 POUŽITÁ LITERATURA

[1] **MARTINŮ, Jiří a Petr ČERMÁK.** *Metodiky vývoje software* [online]. [cit. 2019-08-21]. Dostupné z: http://mvso.cz/wp-content/uploads/2018/02/Metodiky-v%C3%BDvoje-software-studijn%C3%AD-text.pdf

[2] *Metodika vývoje softwaru*. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-08-21]. Dostupné z: https://cs.wikipedia.org/wiki/Metodika_v%C3%BDvoje_softwaru

[3] *Agilní vývoj: Úvod.* Zdroják – o tvorbě webových stránek a aplikací [online]. Dostupné z: https://www.zdrojak.cz/clanky/agilni-vyvoj-uvod/

[4] *Agilní vývoj: Scrum.* Zdroják - o tvorbě webových stránek a aplikací [online]. Dostupné z: https://www.zdrojak.cz/clanky/agilni-vyvoj-scrum/

[5] **EVANS, Eric.** *Domain-driven design: tackling complexity in the heart of software.* Boston: Addison-Wesley, c2004. ISBN 03-211-2521-5.

[6] **EVANS, Eric.** *Domain--Driven Design Reference* [online]. [cit. 2019-08-21]. Dostupné z: http://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf?fbclid=IwAR12WYE6PF-mb0FvtCWUMKTilDVXb1EG5IBcPXH56i1WgaIuUjTW78RDN1M

[7] *Eric Evans — Tackling Complexity in the Heart of Software.* In: Youtube [online]. [cit. 2019-08-21]. Dostupné z: https://www.youtube.com/watch?v=dnUFEg68ESM. Kanál uživatele Domain-Driven Design Europe.

[8] *DDD Strategic Design in under 15 minutes.* In: Youtube [online]. [cit. 2019-08-21]. Dostupné z: https://www.youtube.com/watch?v=Evers5npkmE. Kanál uživatele Alpha Code.

[9] *4. DDD Tactical Design in 14 minutes Part #1.* In: Youtube [online]. [cit. 2019-08-21]. Dostupné z: https://www.youtube.com/watch?v=WZb-FPmiuMY. Kanál uživatele Alpha Code.

[10] *4. DDD Tactical Design in 14 minutes Part #2.* In: Youtube [online]. [cit. 2019-08-21]. Dostupné z: https://www.youtube.com/watch?v=3n3OcAIlXjk. Kanál uživatele Alpha Code.

[11] *6. Implementing Strategic Domain Driven Design.* In: Youtube [online]. [cit. 2019-08-21]. Dostupné z: https://www.youtube.com/watch?v=b6D_NTgzmhs. Kanál uživatele Alpha Code.

[12] *Continuous integration pro tvorbu mobilních aplikací* [online]. [cit. 2019-08-21]. Dostupné z: https://www.ackee.cz/blog/continuous-integration-pro-tvorbu-mobilnich-aplikaci/

[13] **BATIA, Naresh.** *6. Domain-Driven Design - Layered Architecture* [online]. [cit. 2019-08-21]. Dostupné z: https://archfirst.org/domain-driven-design-6-layered-architecture/

[14] **BIEL, Marcus**. *Hexagonal Architecture* [online]. [cit. 2019-08-21]. Dostupné z: https://marcus-biel.com/hexagonal-architecture/

[15] **KHARENKO, Anton.** *Monolithic vs. Microservices Architecture* [online]. [cit. 2019-08-21]. Dostupné z: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

[16] **JŮZA, Petr.** *Mikroservisy všude kam se podíváš* [online]. [cit. 2019-08-21]. Dostupné z: https://medium.com/openwise-tech-blog/mikroservisy-kam-se-podivas-6d81f6fb1f66

[17] *Pivotal Software, Inc.* [online]. [cit. 2019-08-21]. Dostupné z: https://spring.io/

[18] **KUNČAR, Petr.** *Spring – IoC Kontejner* [online]. [cit. 2019-08-21]. Dostupné z: https://www.itnetwork.cz/java/pokrocile/spring-ioc-kontejner

[19] *Spring Cloud* [online]. [cit. 2019-08-21]. Dostupné z: https://spring.io/projects/spring-cloud

[20] **SCHIMANDLE, Tim.** *Spring Cloud – Bootstrapping* [online]. [cit. 2019-08-21]. Dostupné z: https://www.baeldung.com/spring-cloud-bootstrapping

[21] **MIŃKOWSKI, Piotr.** *Quick Guide to Microservices With Spring Boot 2.0, Eureka, and Spring Cloud* [online]. [cit. 2019-08-21]. Dostupné z: https://dzone.com/articles/quick-guide-to-microservices-with-spring-boot-20-e

[22] *Spring Cloud Config* [online]. [cit. 2019-08-21]. Dostupné z: https://cloud.spring.io/spring-cloud-config/reference/html/

[23] *Spring Cloud Data Flow Reference Guide* [online]. [cit. 2019-08-21]. Dostupné z: https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/

[24] **RV, Rajesh.** *Spring Microservices*. Packt Publishing Limited, 2016. ISBN 1786466686.

[25] *Atlassian* [online]. [cit. 2019-08-25]. Dostupné z: https://wac-cdn.atlassian.com/dam/jcr:8e2b794f-0d62-4b16-ab23-7810bfd4ce66/Group%205.png?cdnVersion=532

[26] *OAuth 2.0 Authorization Framework* [online]. [cit. 2019-08-25]. Dostupné z: https://auth0.com/docs/protocols/oauth2