FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání diplomové práce

Název:	Analýza moderních softwarových architektur
Student:	Bc. Jakub Rathouský
Vedoucí:	Ing. Adam Vesecký
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

Cílem práce je nastudovat a popsat problematiku výběru architektury při vývoji softwarových aplikací, zejména webových služeb.

Autor práce vyjde z dostupné literatury a článků ohledně architektur Clean Architecture, Microservice architecture, Onion Architecture, MVC, MVVM a Hexagonal architecture. Jednotlivé architektury popíše a analyzuje jejich strukturu a případná omezení. Dále popíše a navrhne pomocí SI metodik prototyp webové služby, na které demonstruje použití jednotlivých architektur, včetně jejich předností a nedostatků, a to z hlediska implementace, rozšiřování, udržitelnosti a testování. Na závěr autor posoudí, jaké typy architektur jsou vhodné pro jednotlivé typy webových služeb.

Výstupem práce bude teoretický přehled o moderních softwerových architekturách a praktická ukázka jejich implementace. Implementační část bude včetně podrobné dokumentace veřejně dostupná na službě github.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 22. prosince 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Analýza moderních softwarových architektur

Bc. Jakub Rathouský

Katedra softwarového inženýrství
Vedoucí práce: Ing. Adam Vesecký

4. května 2021

Poděkování

Chtěl bych velmi poděkovat panu Ing. Adamovi Veseckému za výborné vedení při psaní této práce. Dále bych chtěl poděkovat Ladislavu Kožejovi za asistenci při výrobě grafického materiálu a mé rodině za podporu a trpělivost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. května 2021 .....

Odkaz na tuto práci

Rathouský, Jakub. *Analýza moderních softwarových architektur*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021. Dostupný také z WWW: ⟨`https://github.com/JakubRathousky/MasterThesis-ArchitecturePatterns`⟩.

Abstrakt

Zvolit vhodné architektonické řešení pro konkrétní aplikaci představuje náročný úkol, který vyžaduje, aby měl návrhář základní povědomí o běžně užívaných architektonických typech a vzorech na různých úrovních abstrakce, a dokázal je vhodně nakombinovat v konzistentní celek.

Cílem této práce bude čtenáři přiblížit populární architektonická řešení, jmenovitě architekturu mikroservis v oblasti provázanosti webových služeb, dále pak Clean architecture, Hexagonal architecture a Onion architecture v oblasti návrhu těchto jednotlivých služeb, a konečně, MVC a MVVM jakožto architektonické vzory v oblasti uživatelské interakce s těmito službami. Jednotlivé typy budou podrobně popsány a vysvětleny z hlediska jejich principů, architektonické struktury a možnostech použití.

Praktickým výstupem práce bude ukázkový prototyp několika propojených služeb, na kterých bude demonstrována příkladná implementace všech představených typů, a ty budou posléze vyhodnoceny z hlediska praktičnosti, udržitelnosti, snadnosti rozšiřování a automatizovaného testování.

Klíčová slova softwarová architektura, mikroslužby, clean architecture, onion architecture, hexagonal architecture, MVC, MVVM

Abstract

A resourceful architectural solution for a full-fledged application is a demanding task to carry out. The designer is required to be knowledgeable about the actual architectural types and patterns of various levels of abstraction, including the ability to assemble them into a consistent whole.

This thesis aims to walk the reader through a set of popular architectural solutions, namely: Microservice Architecture from the perspective of the composition of web services. Next, Clean Architecture, Hexagonal Architecture, and Onion architecture from the perspective of the design of those respective services. And finally, MVC along with MVVM as architectural patterns, from the perspective of user interaction with those services.

Principles, a structure, and possible use-cases of respective types will be thoroughly described and explained. The practical outcome of this thesis will be a prototype of a few connected services, that will demonstrate exemplary use of all introduced types.

Finally, these types will be evaluated by their practicality, sustainability, ease of extension, and automatized testing.

Keywords software architecture, microservices, clean architecture, onion architecture, hexagonal architecture, MVC, MVVM

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Během vývoje každé aplikace, ať už jde o malou službu nebo velkou aplikaci, strukturalizujeme složky, soubory i kód podle určitého vzoru. Tyto vzory utváří celkovou architekturu aplikace a mají zásadní dopad na její přehlednost, udržitelnost i rozšiřitelnost.

V dnešní době se vyskytujeme v informačním světě, ve kterém vznikají nejen nové technologie, postupy, ale i architektonické vzory rychleji, než je možné je všechny sledovat, pochopit a naučit se je. Přitom právě správný výběr architektury je klíčovým krokem pro zajištění bezproblémového budoucího rozvoje a jednoduché správy aplikace.

Noví programátoři, kteří právě vstupují do světa informačních technologií, se často ztrácejí v množství architektur a postupů, které aktuálně existují a vznikají. A to zejména ti, kteří se touto doménou chtějí zabývat - zde mluvíme o Solution Architektech.

V této práci budou vysvětleny nejpoužívanější architektury aktuální doby. Poukázat na silné a slabé stránky, popsat si jejich strukturu a na konkrétní ukázce demonstrovat jejich použití.

Kapitola 1

Cíl práce

Tato práce si klade za cíl rozklíčovat problematiku výběru architektury při vývoji softwarových aplikací a uvést čtenáře do kontextu jednotlivých architektur.

Budou popsány architektonické a návrhové principy, které se běžně používají při vývoji softwarových systémů, konkrétně: Clean Architecture, Microservice architecture, Onion Architecture, Hexagonal architecture, MVC, MVVM.

Jednotlivé architektury se popíší a zanalyzuje se jejich struktura a případná omezení. Dále se navrhne pomocí SI metodik prototyp webové služby, na které se demonstruje použití jednotlivých architektur, včetně jejich předností a nedostatků, a to z hlediska implementace, rozšiřování, udržitelnosti a testování.

Závěrem práce se posoudí, jaké typy architektur jsou vhodné pro jednotlivé typy webových služeb.

Kapitola 2

Analýza

Naše první kapitola se bude zabývat popisem jednotlivých architektur. Na úvod architektury si uvedeme základní popis architektury, po kterém bude následovat podrobnější, detailnější rozbor této architektury. Nakonec si u každé architektury uvedeme silné a slabé stránky.

2.1 Monolit a Microservices

2.1.1 Monolitická architektura

Monolitická architektura je označení pro silně provázaný systém po funkcionální stránce, dále po stránce dat, vstupů a uživatelského rozhraní [1]. Jedná se o jednu z nejčastěji zvolených architektur pro vývoj systému [2].

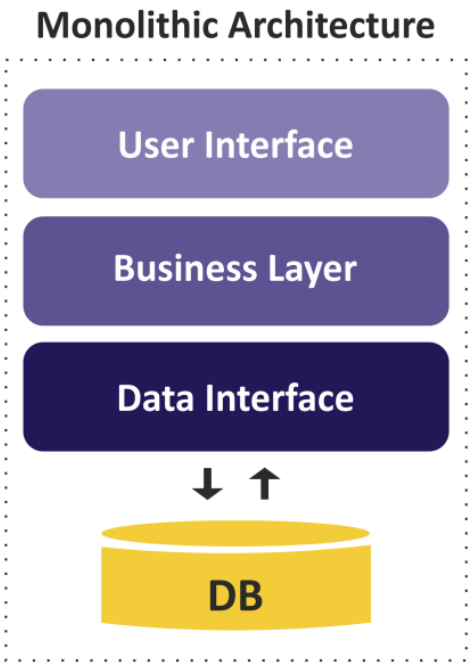
Tyto systémy jsou specifické tím, že i menší změna může vést k rozsáhlým úpravám v projektu [3]. To je z velké části způsobeno zakomponováním všech prvků aplikace do jedné silně provázané vrstvy.

Mezi tyto prvky patří například ukládání a získávání dat pomocí komunikace s databází, doménová logika aplikace, tzn. všechny funkce definující systém a komunikace s uživatelským rozhraním (v některých případech je dokonce uživatelské rozhraní zakomponované přímo v projektu).

Silná provázanost způsobuje, že přidávání funkčních požadavků si vyžádá velké množství úprav v navázaných komponentách napříč projektem, a to i funkcí, které s danou logikou nijak nesouvisí [3]. Z tohoto důvodu pro komplexní velké projekty nemusí být vhodné zvolit tuto architekturu. Vlastnost silné provázanosti způsobuje, podle některých zdrojů, že komplexita přidávání nových funkcí do systému s časem postupně narůstá [4].

To může vést až k extrému, že vývojář stráví delší dobu úpravou existujících funkcí v projektu než vývojem samotné nové logiky.

Prototypování monolitické aplikace je v porovnání s ostatními architekturami jednodušší, protože struktura projektu není tak komplexní a nevyžaduje tolik času pro návrh [5]. Pokud jde o jednorázový malý nebo střední pro-



Obrázek 2.1: Architektura monolitické aplikace

jekt, u kterého se nezamýšlí intenzivní budoucí rozvoj, může být tato volba vhodná. Navíc, noví vývojáři nemusí v rámci zapracování do projektu trávit příliš mnoho času zkoumáním infrastruktury a široké škály technologií, neboť u monolitické architektury je vše obsaženo v jediném projektu.

2.1.1.1 Výhody a nevýhody

Nyní si pojďme shrnout některé výhody a nevýhody monolitických aplikací. Monolitická aplikace je vhodná v následujících případech:

1. vhodný pro implementaci nových modulů - pokud s modulem začínáme, může být vhodné umístit všechen kód do jednoho balíčku či složky pro získání přehledu o doméně, rychlé iterování a základní vývoj
2. mnoho nástrojů je zaměřeno na vývoj monolitických aplikací
3. snadné znovupoužití kódu, jelikož všechen kód najdeme uvnitř jediného projektu
4. jednoduchá konfigurace a snadné nasazení projektu
5. snadné ladění (debugging) a řešení problémů
6. nasazuje se a spouští pouze jedna aplikace

Co se týče nevýhod, můžeme poukázat na následující:

1. komplexnost a provázanost rychle narůstá s novými požadavky
2. během slučování větví ve verzovacím systému bude docházet k častějším konfliktům
3. vývojová prostředí mívají s velkými monolitickými aplikacemi výkonnostní problém
4. dlouho se sestavují - sestavuje se celá aplikace
5. delší proces nasazení aplikace do produkce - stejně jako samotné sestavení, zde navíc ještě probíhá kontrola testu a samotné nasazení aplikace
6. pokud se vyskytnou kritické chyby na produkci, je potřeba produkt opravit a opětovně nasadit jako celek
7. celý produkt je spuštěn jako jeden proces - při selhání jednoho modulu se ukončí všechny a tím i celá aplikace
8. není možné horizontálně škálovat vybrané moduly, jediná možnost je škálovat aplikaci vertikálně tj. navýšit paměťové a výkonnostní prostředky daného procesu
9. těžkopádná migrace technologií - pokud chceme konkrétní modul zmigrovat na novou technologii, je potřeba zásah do velkého množství kódu

2.1.2 Microservices

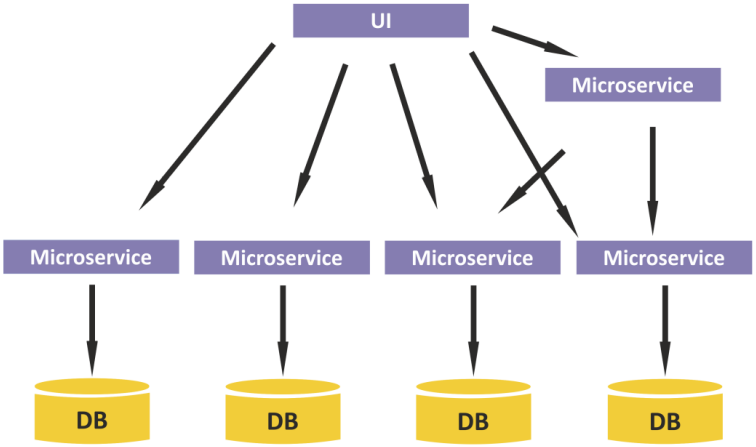
Pojem „Microservices“ nebo „mikroslužby“ vychází původem ze spojení „micro web services“, které poprvé zaznělo na konferenci „Web Services Edge“ o Cloud computing v roce 2005 [6]. Autorem je uveden Dr. Peter Rogers.

Postupem let se pojmenování architektury měnilo, až v roce 2011 získala architektura název, pod kterým se v dnešní době reprezentuje. V tomto období se pořádala konference pro architekty zabývající se návrhem struktury aplikací. Diskutovaly se zde různé architektury, se kterými jednotlivé společnosti aktuálně pracují a experimentují. Společnosti jako Amazon a Netflix byly průkopníky této architektury, které ji mezi prvními zaintegrovaly do svých projektů a o tyto zkušenosti se podělili s ostatními [7].

V posledních letech se tato architektura začala popularizovat, což vedlo k širokému rozšíření používání.

I přes složitější klíčový prvotní návrh a nastavení architektury jsou mikroslužby díky své izolovanosti schopné řešit mnoho problémů velkých i menších projektů, u kterých se čeká rozsáhlý budoucí rozvoj. Je vhodná i pro systémy které mají nerovnoměrné zatížení funkcí a je potřeba systém vhodně škálovat. Příkladem nerovnoměrné zatíženosti jsou funkce pro přihlášení uživatele a

funkce pro získání dat z katalogu zboží. Mezitím, co požadavek na přihlášení vykoná uživatel jednou za návštěvu aplikace, na přehled zboží se vrací opakovaně.



Obrázek 2.2: Architektura mikroslužeb

2.1.2.1 Popis

Principem Microservices je rozdělení domény do samostatných služeb takových, aby každá služba řešila pouze jednu funkční oblast a byla nezávislá[8]. To znamená, že nepotřebuje ke své funkci žádné přímé volání jiných služeb. Tyto služby lze navíc samostatně nasazovat, není tedy potřeba vždy nasazovat celý systém najednou.

Výsledná aplikace vypadá jako množina malých služeb. Do této množiny se mohou přidávat, resp. odebírat, služby podle potřeby. Díky této vlastnosti se systém snadno škáluje, navíc na rozdíl od monolitické architektury můžeme horizontálně škálovat pouze potřebné zatížené služby a ne celý systém.

2.1.2.2 Použití

Při vytváření nové domény a jejím prototypování je v pořádku začít s monolitickou aplikací a teprve postupem času přejít na architekturu mikroslužeb [5].

Při zakládání projektu ještě nemusíme mít zcela jasnou představu o doméně a všech prvcích, které v aplikaci budeme potřebovat. Takové změny by mohly mít velký dopad na návrh našich služeb v průběhu času.

To znamená, že zvolit mikroslužby v rané fázi vývoje projektu nemusí být ideální řešení a mohlo by vést ke znatelnému zpomalení týmu v době, kdy chce vyvinout první funkční prototyp. Navíc hrozí riziko vytvoření systému s nevhodně definovanou množinou služeb, s čímž se pojí ruku v ruce udržitelnost systému do budoucna.

Investice do mikroslužeb má smysl v případě, kdy aplikace může benefitovat ze škálování jednotlivých služeb, jejich nezávislého nasazení nebo při potřebě použít více technologií vhodných k řešení daného problému. Kapacita týmu jde rovněž ruku v ruce škálování mikroslužeb. Pokud máme mikroslužby, a tým se příliš rozroste, tak ho můžeme rozdělit na dva nebo více menších a přerozdělit mezi ně určitou podmnožinu služeb. Jednodušeji se tak dělí zodpovědnosti mezi týmy.

2.1.2.3 Výhody a nevýhody

Největší výhodou mikroslužeb je rozdělení logiky na menší samostatné celky. Rozdělení umožňuje rozčlenit vývojáře do týmů starající se o danou funkcionalitu (službu), takže může probíhat nezávislý vývoj na více službách paralelně.

Zrychlení procesu celého vývoje pramení ze samotné podstaty architektury. Vývojář pracuje na své službě, po potřebných úpravách ji sestaví, spustí se testy a ta se poté případně nasadí do produkce. Pokud dojde k chybě při sestavení nebo výsledkem spuštěného testu, vývojář ví, kde chybu hledat, a po opravě opět zkusí projekt sestavit a podrobit testům. Právě díky rozdělení na malé služby se může vývojář starat pouze o svou službu a na ostatní nemusí brát zřetel. Díky tomu nemusí pokaždé spouštět sestavení a otestování celé aplikace, čímž se ušetří čas a prostředky při vývoji.

Další výhoda plynoucí z rozdělení je nezávislost na programovacím jazyku a platformě. Služby se mohou implementovat v jazycích, které jsou pro danou funkcionalitu nejvhodnější a tím zefektivnit fungování systému. Například, pokud systém obsahuje službu, která řeší statistické úkony, analyzuje nebo zpracovává velké množství dat, lze tuto službu implementovat v jazyce Python. Další službu pracující s citlivými daty můžeme napsat v Java nebo C#. [9]

Malá provázanost systému umožňuje snadné upravování a rozšiřování projektu. Pokud je potřeba, je možné celou službu přepsat bez toho, aby ostatní služby věděly, že se něco takového děje.

V neposlední řadě již zmíněné škálování aplikace je důležitou výhodou mikroslužeb.

Výhody mikroslužeb jsou patrné. V některých případech ale dokážou zmíněné výhody zkomplikovat jiné části vývoje.

Aplikace navržena touto architekturou je jednoduše testovatelná. Rozdělení služeb umožňuje otestovat a pokrýt větší množství řádků kódu testy. Problém je ale v počtu těchto funkcí, oproti například monolitické aplikaci obsahuje mnohonásobně více funkcí. To je způsobeno tím, že si například perzistentní vrstvu musí každá služba implementovat sama, dále jsou zde navíc funkce zajišťující komunikaci s dalšími službami spolu s následným mapováním přijatých dat a tak dále. Zajištění pokrytí funkcionalit mikroslužeb testy je časově náročnější. Tento čas je částečně kompenzovaný tím, že vývojář při sestavení nové verze pouští pouze testy pro danou službu.

Při vývoji funkcionality, která vyžaduje zásah do více služeb, narážíme na problém rozdělení služeb do separátních týmů. Tento typ funkcionalit je organizačně složitější a musí být správně naplánován. Vývojář během zpracování požadavku musí spolupracovat s patřičným týmem, který o změnách v jejich službě musí vědět, a případně upravit nebo dopsat testy pokrývající danou oblast.

I v monolitické aplikaci se může stát, že funkcionalitu implementuje více týmů. Je zde ale rozdíl týkající se nasazení těchto funkcionalit. Jak bylo již v sekci o monolitické architektuře 2.1.1 rozebráno, monolitická aplikace je jeden celek, který obsahuje vždy všechny implementované funkce pospolu, a při nasazení aplikace se vždy nasadí ve stejnou chvíli, zatímco mikroslužby jsou rozděleny a jejich nasazení je nezávislé na sobě. Pokud tedy nastane zmíněná potřeba implementace funkcionality skrze více služeb, tak je potřeba správně naplánovat nasazení odpovídajících verzí těchto služeb.

Stejně jako jsme si uvedli u monolitické architekury stručný seznam výhod a nevýhod, i zde zakončíme tuto kapitolu rychlou rekapitulaci [10][11].

Výhody

1. kompaktní a snadno pochopitelný kód - rozdělením aplikace na malé služby zajistíme, že každá služba bude mít relativně malou množinu funkcionalit
2. rychlejší sestavení - sestavují se jednotlivé služby a nikoliv celá aplikace
3. rychlejší vyhodnocení unit testů
4. rychlejší nasazení na produkci
5. nezávislost/izolace služeb - pokud dojde k chybě v nějaké ze služeb u její nové verze, stačí danou službu nahradit předchozí verzí, ostatní služby mohou fungovat nezávisle dál
6. možnost škálovat jednotlivé služby
7. selhání jedné služby nezpůsobí pád celé aplikace
8. lehká adaptace nových technologií - možnost vytvoření nebo převedení jednotlivých služeb na jiné/nové technologie
9. možnost nezávislého paralelního vývoje jednotlivých částí aplikace
10. zpětná kompatibilita - je možné mít nasazeno více verzí stejné služby, přičemž ostatní služby, které s ní komunikují, mohou používat starší verzi adresy (endpoint), dokud se neprovede jejich aktualizace

Nevýhody

1. není jednoduché navrhnout správné rozvržení služeb
2. přidaná složitost způsobena technologiemi použitými pro podporu fungování distribuovaných systémů (například Message broker, služby pro agregované logování a jiné)
3. kód, který se využívá ve více službách najednou, se musí extrahovat do separátních sdílených knihoven
4. některá vývojová prostředí nemají plnou podporu pro distribuované aplikace - složité spouštění a ladění několika služeb najednou během vývoje
5. vývoj funkcionality vyžadující změny v jiných službách je komplikovaný
6. hledání a reprodukování vzniklých chyb je těžší než u monolitu
7. potřeba synchronizovat spolupráci služeb a ukládání do více databází najednou za cílem udržení konzistence dat

2.1.2.4 Na co si dát pozor

Prvotní návrh je klíčový ke správnému fungování celého systému. Špatně navržený systém se může stát rychle nepřehledný a tím se všechny výhody mikroslužeb stanou nevýhodami systému.

Technická dokumentace pokrývající přehled služeb a jejich metod je u mikroslužeb nutností. Kvůli vysokému množství těchto služeb a funkcí v systému je složité dohledat konkrétní službu a metodu implementující logiku, kterou potřebujeme. Časová investice vložená do zdokumentování systému se v budoucnu zužitkuje během dalšího rozšiřování.

Logování by ve službách nemělo chybět. Například při pádu aplikace napsané monolitickou architekturou dojde k výpadku celého systému a z výpisu se často daří dohledat důvod pádu systému. Dále se u monolitické architektury nemůže stát, že by se zavolala metoda v kódu, a ta by nevrátila odpověď na dané volání. To ale neplatí u mikroslužeb - zde se může stát, že komunikace mezi službami selže z různých důvodů. Tedy, jejich stav může být ovlivněn jak běhovým prostředím, tak celou infrastrukturou. Každá služba tvoří potenciální bod selhání.

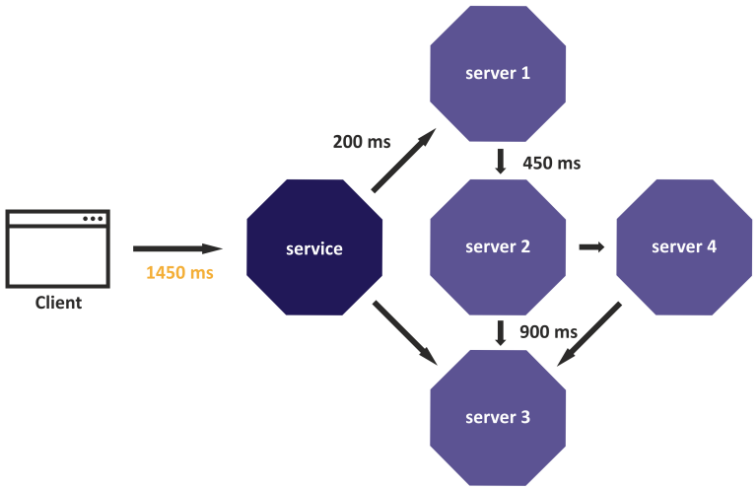
Při využití monitorování, neboli logování a hlídání stavu služeb (Health Check), docílíme toho, že budeme mít přehled o stavu jednotlivých služeb a budeme dostatečně rychle informováni o případném pádu služby.

Health Check slouží k monitorování stavu jednotlivých služeb a detekci případných výpadků. V případě výpadku služby by měl být nastaven proces, který se pokusí danou službu zapnout.

V případě, kdy služba potřebuje komunikovat s databází, která obsahuje data potřebná ke splnění daného úkolu, je potřeba si dát pozor na to, o jaká

data jde. Pokud se jedná například o databázi obsahující konfigurační parametry nebo jiné sdílené údaje, může do této databáze přistoupit jakákoliv služba napřímo. Pokud ale obsahuje data, která jsou nějakým způsobem vázána k doméně, to znamená, že je nad nimi nějaká aplikační logika jako například filtrování nebo kontrola přístupu, je zapotřebí přístup k databázi omezit. Toto omezení je zapříčiněno tím, že při změně modelu nebo databáze musí všechny služby změnu registrovat a reagovat na ni. Další komplikací je zajištění konzistence. Ideální řešení je mít nad takovou databází službu, která umožňuje přístup k datům pomocí jejího rozhraní. Při jakékoliv změně DB se poté mění jen odpovídající služba a ostatní nejsou změnou nijak brzděné. Tento přístup odpovídá principu mikroslužeb a nízké provázanosti.

Komunikace v monolitu probíhá v rámci jednoho procesu, u mikroslužeb požadavky mohou být odeslané napříč několika servery a komunikace přes API requesty je tak omezena rychlostí sítě.



Obrázek 2.3: Synchronní komunikace mezi službami

To má za následek, že i například jednoduchý dotaz na data, která se musí agregovat z více služeb najednou, může trvat neúměrně dlouho. Alternativní varianty, zahrnující asynchronní komunikaci, jsou například Event-driven komunikace nebo posílání zpráv přes message brokera.

Rozdělení rozhraní služeb na více typů vede na problém při implementaci, kdy se vývojář musí správně rozhodnout, zdali se jedná o resource (např. API pro klienta) nebo o funkci (slouží pro komunikaci mezi službami).

2.1.2.5 Návrhové vzory

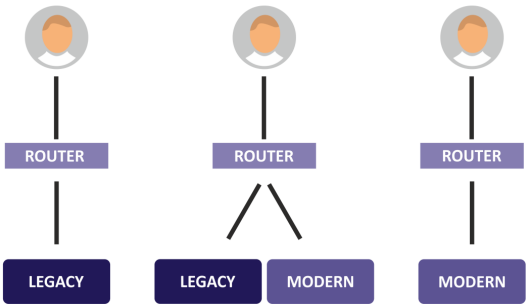
S mikroslužbami jsou spojené návrhové vzory, které si kladou za cíl usnadnit jejich používání.

V přehledu níže je výpis několika návrhových vzorů a jejich účel. Některé z nich si přiblížíme více do hloubky [12].

- Database per Service - popisuje princip přiřazení databází pod samostatné služby k zajištění nízké provázanosti
- Client(server)-side Discovery - použité pro routování požadavků na dostupné služby
- *Strangler* - zajištění hladkého přechodu při migraci na mikroslužby z jiné architektury
- Circuit breaker - prevence kaskádového pádu služeb při výskytu chyby
- API Gateway - přístup k jednotlivým službám přes jednotnou doménu
- Monitorování:
  - Log aggregation - agregace logu z více služeb pro smysluplné popsání stavu procesu
  - Health Check - detekce stavu služeb
- Hostování služeb - způsoby nasazení služeb:
  - Single Service per Host
  - Multiple services per Host

Je běžné, že systémy mikroslužeb vznikají migrací z jiné architektury, například monolitické. Migrace mohou být velmi složité a je důležité si uvědomit, že při přechodu nemusíme všechny služby vyvíjet od znova. Pokud programovací jazyk, ve kterém je funkcionalita napsána, slouží dobře pro implementaci daného požadavku a funkcionalitu lze jednoduše extrahovat, tak poté je možné danou funkci vyjmout a uzavřít do nové samostatné služby. Tím se dá ušetřit mnoho práce a času.

Migrace je většinou časově náročná a trvá nějaké delší období uvést aplikaci do stavu, aby byla schopná provozu. Pro lineární přechod a neomezení stávajících funkcí se může použít návrhový vzor **Strangler**.



Obrázek 2.4: Strangler - převedení stávající aplikace (Legacy) na novou (Modern)

Tento vzor zajišťuje koexistenci nové a staré verze aplikace. Aplikace existuje na stejné stránce, dotazy z ní přichází na router, který zajišťuje přesměrování požadavku na správnou službu. Pokud router zaznamená, že služba odpovídající za danou funkci již existuje v novém systému, přepošle dotaz na nové rozhraní. Pokud daná funkce v novém systému naprogramována nebyla, přepošle ji standardně do starého systému.

Tímto způsobem lze zajistit inkrementálnímu přechod na novou verzi systému. Jakmile jsou funkce dovyvinuty, je stará aplikace zrušena a všechny dotazy budou směřovány na novou verzi aplikace. Tomuto postupu se říká *Transform* a skládá se tedy ze 3 kroků:

1. Transform – zahájení prací na nové verzi aplikace
2. Coexist – postupné nahrazování služeb staré verze službami novými
3. Eliminate – zrušení staré verze aplikace, jakmile jsou všechny služby převedeny

Návrhový vzor Gateway nabízí zapouzdření všech služeb pod jednu adresu. Stránka se tak jeví jako jedna služba, a po zpracování dotazu klienta pomocí Gateway dojde k přeposlání na konkrétní službu. Další velká výhoda tohoto vzoru je například možnost kontroly práv k jednotlivým službám na jednom místě.

O pravděpodobnosti pádu služeb v architektuře mikroslužeb byla již zmínka v kapitole 2.1.2.4. Při pádu služby je důležité mít nastavený správný proces zotavení. Rozdělením služeb do skupin podle zatíženosti a důležitosti můžeme dosáhnout toho, že i když k nějakému pádu služby dojde, jiné klíčové služby budou stále aktivní, čímž se dá zabránit nejhoršímu, jako například nedostupnost celého systému. Tomuto rozdělení se říká *Bulkhead pattern*. Jinou variantou je vzor *Sidekick/Sidecar*, ten hovoří o rozdělení služeb do vlastních procesorových kontejnerů, které fungují naprosto izolovaně, například pomocí Dockeru.

Pokud jde o hostování služeb, máme na výběr ze dvou možností: „Jedna služba na Jeden OS“ a „Více služeb na Jeden OS“. Přibližme si nyní klady a nevýhody obou způsobů.

Jedna služba na jeden OS

- služby jsou více izolované
- lepší škálování a správa závislostí
- lze využít Hypervisor - software, firmware nebo hardware, který vytváří a spravuje virtuální stroje [13].
  - Hypervisor dokáže redukovat hardwarové prostředky, protože je dokáže simulovat nad jedním hostem
  - Hypervisor může být naopak přítěží, pokud cena za jeho správu spotřebuje více prostředků, než kolik ušetří virtualizací

Více služeb na jeden OS

- méně strojů ke správě
- snazší automatizace
- služby na sobě nejsou plně nezávislé, musí spolu sdílet stejné knihovny a funkce prostředí (například ve stejném prostředí nemůžeme mít dvě služby napsané v Javě a zajistit, aby každá běžela na jiné verzi)
- chyby vzniklé prostředím jsou hůře reprodukovatelné
- horší škálování – nemůžeme horizontálně škálovat konkrétní službu, ale vždy cluster několika služeb, které spolu sdílí prostředí

2.2 Architektury služeb

2.2.1 Clean Architecture

Pojem Clean Architecture zavedl Robert C. Martin v roce 2012. Tato architektura si klade za cíl rozdělení systému do jednotlivých vrstev, kde každá z nich odpovídá za jednotlivé přidělené funkce. Snaží se řešit běžné problémy informačních systémů jako například předcházet nepřehlednosti struktury projektu, nedodržení nízké provázanosti tříd, duplikace kódu nebo nedodržení SOLID principů. Zmíněné příklady nastávají u komplexních velikých i menších systémů, a po nějaké době vychází lépe systém vytvořit znovu, než pokračovat v podpoře stávajícího. Rozdělení do zmíněných vrstev má za cíl zjednodušit systém a jeho vývoj. Na každou vrstvu jsou kladena pravidla, která každá vrstva musí splňovat.

Jde konkrétně o tyto body:

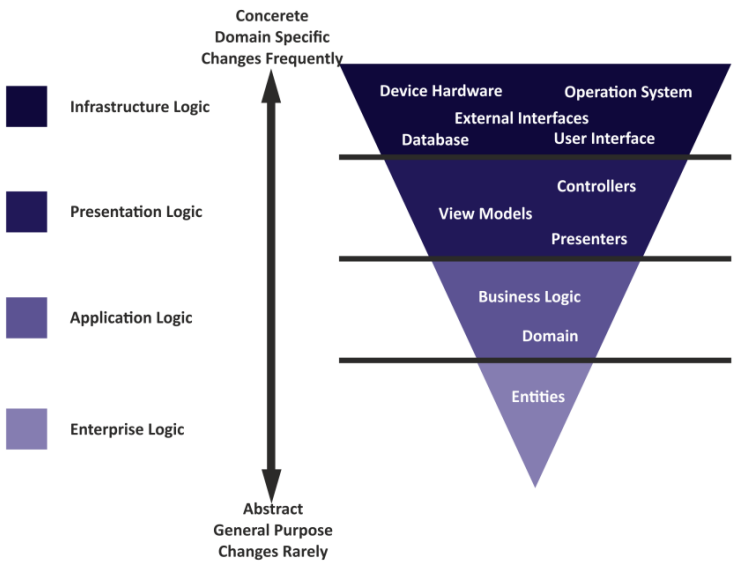
1. mělo by být možné ji samostatně testovat nezávisle na ostatních vrstvách
2. měla by být nezávislá na uživatelském rozhraní
3. měla by být nezávislá na datových zdrojích
4. měla by být nezávislá na rozhraních třetích stran
5. měla by respektovat hranice své domény

2.2.1.1 Popis

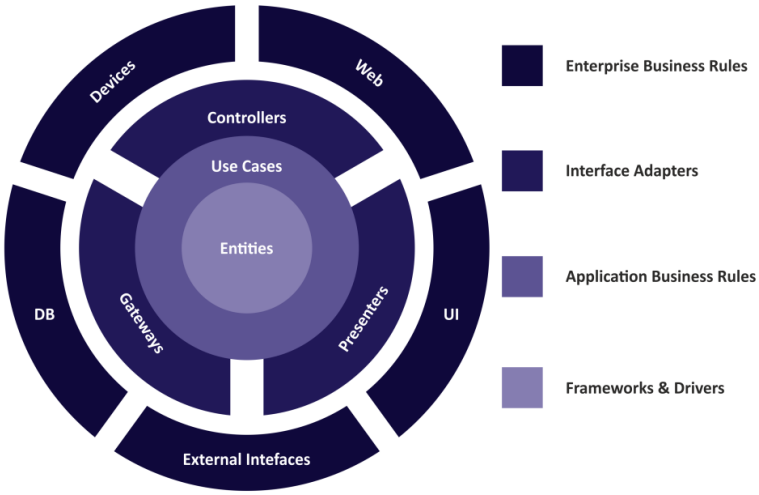
Architektura popisuje rozdělení systému do samostatných nezávislých vrstev. Konkrétní počet a pojmenování vrstev je variabilní, v seznamu níže jsou vypsané některé kombinace, které se veřejně objevují. Vrstvy jsou seřazeny podle pořadí, na levé straně je vrstva nejvyšší úrovně a na pravé straně nejnižší úrovně. Závislosti jsou od vrstvy vyšší úrovně k nižší úrovni. Z toho plyne, že vrstva nižší úrovně neví nic o vrstvách, které jsou nad ní.

1. Infrastructure, Presentation, Application a Enterprise.
2. UI, Presentation, Use case a Entity
3. (Persistance, Interface, Presentation ve stejné vrstvě), Application a Domain

V této práci se zaměříme na rozdělení č. 1.



Obrázek 2.5: Clean architektura - rozdělení vrstev 1



Obrázek 2.6: Clean architektura - rozdělení vrstev 2

2.2.1.2 Enterprise vrstva

Tato vrstva obsahuje všechny datové modely a entity, které tvoří základ business logiky. Každá z entit by měla být klíčová pro popis domény daného systému.

Pro představu si uvedeme příklad na systému zabývajícím se rezervací letenek.

Pro tento systém jsou entity „letenka“ a „termín“ objekty popisující daný business model a neobejdeme se bez nich. Patří tedy do této vrstvy.

Naopak například třída, která reprezentuje vybrané filtry uživatele při vyhledávání, není specifická pro zvolenou doménu a do této vrstvy nepatří.

Zapsání entit by mělo být co nejvíce generické, resp. abstraktní.

Entity by u sebe neměly mít žádné informace o logice nebo o způsobu ukládání do databáze jako databázové anotace nebo vazbu na framework použitý pro ukládání. Změna frameworku/knihovny pro ukládání dat není změna v business logice, proto by neměla existovat reference na daný framework/knihovnu v této vrstvě. Naopak, pokud rozšíříme doménu nabízení „letenek“ o „jízdenky“, nebo přidáme k entitě „termín“ kapacitní omezení, jedná se o úpravy business modelu a jeho pravidel.

Častou chybou je implementace databázové vrstvy tak, že se ji aplikační logika přizpůsobuje.

2.2.1.3 Application vrstva

Aplikační vrstva je jádrem celého systému. Odehrává se zde veškerá business logika, která je klíčová pro splnění cílů aplikace.

Je důležité, aby tato vrstva byla naprosto samostatná a neměla žádné provázanosti s vrstvami Enterprise ani Presentation. Tyto vrstvy mají větší

tendenci se měnit a jejich změna by neměla vynucovat změnu vrstev nižší úrovně. V případě výměny těchto vrstev, například z důvodu převedení desktopové aplikace na aplikaci webovou, by změna měla proběhnout zcela bez vědomí této vrstvy o daném procesu.

Izolování těchto vrstev vyžaduje využití rozhraní. Stanoví se jasné a specifické rozhraní, které musí vrstva Enterprise splňovat. Toto rozhraní poté implementuje konkrétní třída starající se o získání dat. Při běhu aplikace jsme schopni si pomocí rozhraní tuto třídu získat a komunikovat s ní. Vrstva Application tím dokáže získat potřebná data a rozhraní zajistí, že data mohou přicházet jak z MongoDB, tak MSSQL, a Application vrstvu to nijak nemění.

Služby jsou uspořádané do logických celků, které odpovídají jednotlivým scénářům (anglicky Use Case), které si aplikace stanovila. Vývojář by měl být schopný v krátké době najít místo zabývající se daným scénářem a detekovat všechny služby, které se na vykonání této logiky podílí.

Žádost na vykonání scénáře přijde z vrstvy Presentation. Služba v Application vrstvě odpovědná za vykonání scénáře získá potřebná data z vrstvy Enterprise. Vykoná potřebné kroky ke splnění daného cíle a vrátí výsledek vrstvě Presentation. Tím, že vrstvy Presentation a Application jsou oddělené, je vrstva Application schopná daný úkol splnit bez ohledu na to, jestli požadavek přišel z mobilního zařízení nebo webového.

Krom plnění jednotlivých scénářů se také stará o hlídání práv a business pravidel. V případě zmíněné letecké společnosti se jedná například o hlídání věku pasažéra - zde může být nastaveno pravidlo, že příliš mladý cestující musí mít doprovod starší osoby a sám si letenku koupit nemůže.

2.2.1.4 Presentation vrstva

V této třídě dochází k definování výstupního formátu dat a komunikace s klientem.

Využívají se zde prezentační návrhové vzory jako MVVM, MVC, MVP a jiné.

Hlavním cílem této vrstvy je fungovat jako most mezi uživatelem a business logikou. Uživatel od systému vyžaduje splnění nějaké akce. Obvykle pomocí uživatelského rozhraní odešle požadavek. Tento požadavek odchytí prezentační vrstva, která na jeho základě zavolá odpovídající službu obsluhující daný scénář. Pokud uživatel v dotazu poslal i nějaká data, prezentační vrstva zařídí konverzi na formát, se kterým aplikační vrstva pracuje. Po vyhodnocení dotazu službou konvertuje prezentační vrstva navrácená data na tvar srozumitelný klientem a odpověď odešle.

2.2.1.5 Infrastructure vrstva

Poslední uvedená vrstva se stará o veškeré nastavení prostředí jako napojení frameworků nebo konfigurace databáze. Slouží jako opěrný bod, který podpo-

ruje celou aplikaci. Neobsahuje většinou mnoho kódu, protože ten je napsaný buď v napojených frameworcích nebo knihovnách.

Implementuje rozhraní, které umožňuje komunikaci s touto vrstvou.

2.2.2 Onion Architecture

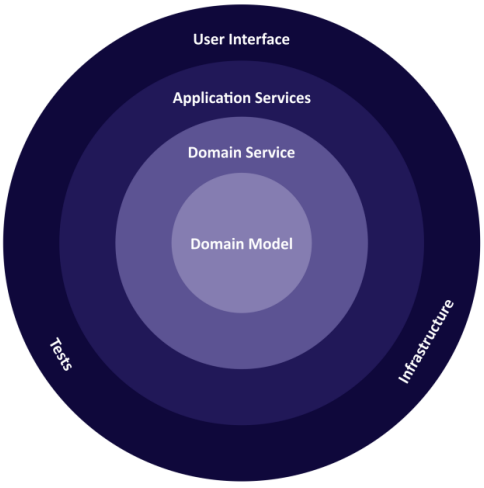
Jeffrey Palermo použil výraz „Onion architecture“, když se snažil se světem sdílet svůj pohled na DDD neboli Domain Driven Design [14].

Hlavní otázku, kterou se snažil tímto principem zodpovědět, byla „Jak implementovat business logiku, pokud jí není umožněno pracovat s datovou vrstvou“. Domain Driven Design spolu s principem obrácení toku závislostí (anglicky Dependency Inversion Principle) tvoří základ pro Onion architecture. Avšak použití DDD není nutnou podmínkou pro implementaci této architektury, stejně jako použití Onion architecture nenutí vývojáře zvolit DDD.

Z názvu architektury „Onion“ je patrný způsob vrstvení systému. Přirovnání k jednotlivým vrstvám cibule, kde se ve středu nachází jádro celého systému neboli Application Core. Toto jádro postupně nabalují další vrstvy rozšiřující vrstvu pod sebou.

Jednotlivé vrstvy stejně jako u Clean architecture se liší mezi jednotlivými výklady, v této práci použijeme interpretaci objevitele této architektury, tedy Jeffreyho Palerma. Autor prezentuje tyto vrstvy:

1. Domain Model
2. Domain Services
3. Application Services
4. Outer layer (testy, uživatelské rozhraní, infrastruktura)



Obrázek 2.7: Onion architektura

Domain Model je vrstvou nejnižší úrovně a Outer layer je vrstvou nejvyšší úrovně. Trojici Domain Model, Domain Services a Application Services označuje zmíněné aplikační jádro a to z důvodu, že v těchto vrstvách se nachází veškerá business logika potřebná k plnění cílů aplikace.

Modely Onion architecture a Clean architecture si jsou velmi podobné, a to z důvodu, že se snaží řešit stejný problém. Oba cílí na snížení provázanosti, složitosti a zvýšení životnosti systému. Často jsou tyto dvě architektury označovány za totožné anebo implementací jedna druhé.

2.2.2.1 Domain Model

Domain Model vrstva je podobná vrstvě Enterprise 2.2.1.2 z kapitoly Clean Architecture 2.2.1. Obsahuje entity a třídy, které definují doménovou strukturu systému.

Jedná se o vrstvu nejnižší úrovně a tedy nemá přímou referenci na žádnou jinou vrstvu.

2.2.2.2 Domain Services

Vrstva druhé úrovně. Může napřímo používat funkce z první vrstvy, ale k jiným vrstvám přístup nemá. Domain Services není úplnou kopií vrstvy Application 2.2.1.3 o které mluví Clean Architecture 2.2.1, jedná se spíše o podmnožinu zmíněné vrstvy. Najdeme zde třídy které popisují doménové procesy a vztahy mezi nimi.

2.2.2.3 Application Services

Předposlední vrstva, tedy Application Services, doplňuje vrstvu Domain Services a obsahuje konkrétní implementaci scénářů. Spolu dohromady tvoří dříve referencovanou Application vrstvu 2.2.1.3.

Z této vrstvy můžeme napřímo využívat funkcí a vlastností jak Domain Services tak Domain Model vrstev.

2.2.2.4 Outer Layer

Vrstva nejvyšší úrovně, kde se nachází všechny součásti systému, které se přímo nepodílí na definování business logiky, ale slouží jako podpora sytému. Konkrétně jsou v této vrstvě umístěny například testy, napojené frameworky, komunikace s rozhraním třetích stran, komunikace s UI nebo databáze.

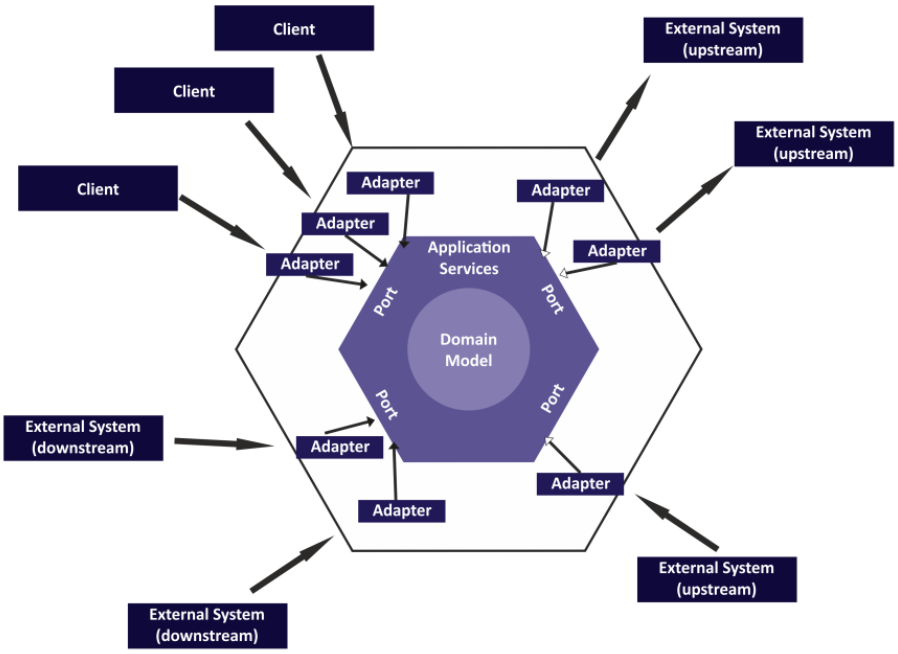
Často se můžeme setkat se situací, kdy na frameworcích a databázích staví značná část architektury systému. Tento způsob implementace systému není správný, protože zbytek systému je silně provázaný se zvolenými technologiemi a přestává být adaptivní na změny. Izolace infrastrukturních prvků do vnější vrstvy je velmi častý problém mnoha existujících aplikací.

Outer Layer vrstva může využívat funkcí všech ostatních vrstev napřímo, ale o její existenci žádná jiná vrstva neví.

Způsob, jak umožnit systému využívat funkce poskytované touto vrstvou, spočívá v tom, že vnitřní vrstvy definují rozhraní, které musí splňovat jejich požadavky, aby mohly správně fungovat. Třídy v této vrstvě poté konkrétní rozhraní implementují a definují v sobě požadované funkce. Služby si poté od prostředí vyžádají třídu, která toto rozhraní implementuje, a použije ji.

2.2.3 Hexagonal architecture

Architektura známá také jako „Ports and adapters“ byla představena v roce 2005 autorem Alistarem Cockburnem. Systém je rozdělen do dvou vrstev, **vnitřní** a **vnější**. Jádro aplikace je označeno jako „Application and Business Logic“. Toto jádro je **vnitřní** vrstvou a má na svých hranicích takzvané porty. Do těchto portů se pomocí adaptéru mohou připojovat další komponenty, které se nachází v **vnější** vrstvě. Co jsou komponenty, co obsahují a jak napojení funguje, si popíšeme později.



Obrázek 2.8: Hexagonální architektura

Začněme porovnáním fungování Hexagonal Architecture oproti tradiční vrstvené architektuře.

2.2.3.1 Popis

Pro připomenutí, tradiční vrstvené architektury se většinou skládají ze tří vrstev a to *Controller* pro komunikaci s klientem, *Service* vykonávající logiku aplikace a vrstvu *Persistence* pro manipulaci s daty a jejich ukládáním. Směr komunikace většinou jde zleva doprava, neboli Controller zavolá vrstvu *Service* a občas při požadavku na jednoduché zobrazení dat dokonce vrstvu *Persistence*. *Service* volá Persistence vrstvu, aby získala data pro své výpočty nebo předala data k uložení. Datové modely bývají většinou sdílené skrze všechny vrstvy a tím i frameworky a knihovny, které odpovídají za jejich činnost.

Představme si situaci, kdy je naše aplikace napsána pomocí konkrétního programovacího jazyka, například C#. Aplikace je nasazena na serveru s operačním systémem Windows. Během vývoje jsme použili mnoha knihoven implementující funkce, které v systému vyžadujeme.

Nyní přijde požadavek, že musíme migrovat naši aplikaci na nový server obsahující operační systém Linux. Během migrace aplikace zjistíme, že některé knihovny nejsou multiplatformní a nefungují na Linuxovém jádře. Tyto funkce (a tím tedy danou knihovnu) jsme ale použili napříč systémem v různých funkcích. Po nalezení vhodné náhrady musíme daná místa přeprogramovat, aby nově využívaly alternativní knihovnu.

Podobná situace může nastat například i v případě, že knihovna přestane být aktuální a již není podporována nebo při změně typu databáze během vývoje (například kvůli nízkému výkonu). I v takovém případě může nastat situace, kdy bude nutné knihovnu ze systému odstranit a případně místo ní zaintegrovat knihovnu s podobnou funkcionalitou.

Hexagonal architecture poskytuje zázemí k vyřešení tohoto problému, hlavní myšlenkou je oddělení logiky (jádra) aplikace od jakékoliv vazby na specifické technologické podrobnosti, které se mohou kdykoliv v průběhu vývoje a životnosti aplikace změnit, a to právě pomocí adaptérů a portů.

Na začátku kapitoly jsme se dozvěděli, že ze základu se Hexagonal architecture skládá ze 4 základních prvků:

1. jádro obsahující aplikační a business logiku
2. porty
3. adaptéry
4. komponenty

V jádře aplikace se vyskytuje jen kód sloužící k řešení business scénářů a to ve formě pouhých tříd, rozhraní a jiných struktur specifických pro daný jazyk. V jádře by se naopak nemělo vyskytovat používání frameworků, knihoven třetích stran a jiných prvků, které nejsou specificky vyžadovány business logikou.

Na „hranicích“ logického jádra se vyskytují porty. Porty slouží pro připojení komponent pomocí adaptéru.

Obrázek 2.9: Adaptér a port

2.3 Architektonické návrhové vzory

Poslední dva architektonické vzory, které si v této kapitole popíšeme, jsou MVC a MVVM.

Oba patří do skupiny Prezentačních návrhových vzorů, jejich účelem je prezentace dat uživateli a redukce závislostí v prezentační vrstvě aplikace.

Jak již bylo zmíněno, tyto návrhové vzory by se měly používat pouze v prezentační vrstvě.

2.3.1 MVC

Tento prezentační návrhový vzor vymyslel Trygve Reenskaug, když pracoval na programovacím jazyku Smalltalk v roce 1979[15].

Originální název je *Model - View - Controller*, ale při psaní knihy *Design Patterns: Elements of Reusable Object-Oriented Software* vznikla pro název zkratka MVC.

Stejně jako u předchozích probíraných architektur i u MVC se konkrétní implementace architektury liší, například u některých implementací komunikace probíhá pouze skrze Controller a View o existenci Modelu neví. V této sekci se budeme věnovat tradičnímu návrhu MVC vzoru.

2.3.1.1 Historie

Dříve byly aplikace relativně malé a snažily se plnit dvě funkcionality:

1. definice zobrazení uživatelského rozhraní, hierarchické uspořádání prvků a reakce na uživatelských vstupů

2. implementace logiky, která nemohla být jednoduše zakomponovaná do uživatelského rozhraní

Logika a zobrazení (View) v těchto případech byly silně provázané a

při vývoji jsme si museli dát pozor na změnu ve View, stejně jako v logice aplikace. Nebylo možné je vyvíjet paralelně v různých týmech.

Zdrojem problému byla tedy silná provázanost mezi View a logikou aplikace. K vyřešení tohoto problému musíme rozdělit tyto dvě části takovým způsobem, aby byly na sobě nezávislé. Tím vznikl princip nazývající se *Separovaná prezentace* (Separated Presentation) a jeho cílem je vytvořit hranici mezi doménovými objekty a těmi objekty, které se používají v uživatelském rozhraní.

Při aplikaci tohoto principu dostaneme dvě skupiny: prezentační část a logická část. Prezentační část je tvořena dvěma prvky a to: View a Controller. Do druhé skupiny zabývající se logikou patří prvek Model.

Tímto vznikl návrhový vzor MVC.

2.3.1.2 Model

Model je řídící jednotkou celého vzoru. Zpracuje data, aplikační logiku a business pravidla. Neví o existenci Controlleru nebo View.

2.3.1.3 View

View se stará o veškeré zobrazení dat uživateli pomocí uživatelského rozhraní. Obsahuje logiku, která reaguje na změnu stavu modelu.

V některých variantách MVC se setkáme s tím, že View nepracuje napřímo s Modelem. Tradičně je ovšem aplikovaný například návrhový vzor Observer, který zajišťuje, aby se View dozvědělo o změnách v Modelu a provedlo patřičné úpravy zobrazení.

2.3.1.4 Controller

Controller zajišťuje správné vyhodnocení následující akce na základě vstupu uživatele.

Na základě vstupu bude Controller informován o vstupu uživatele a na základě typu interakce provede patřičnou akci, která je pro danou událost specifikována.

V neposlední řadě sleduje všechny modifikace v Modelu a případně patřičným způsobem upraví View.

2.3.1.5 Výhody

Jednou z největších výhod je rychlý vývoj. Rozdělením prezentační vrstvy na zmíněné menší části získáme možnost paralelního vývoje týmu, lze tedy současně pracovat na View, Modelu i Controlleru.

Vizuální prvky aplikace se mění mnohem častěji než data a business logika. Rozdělení pomocí MVC umožňuje nezávislé změny uživatelského rozhraní bez nutného zásahu do jiných komponent. Navíc Model je snadné použít opakovaně. Pokud potřebujeme stejná data zobrazit v různých formátech, stačí nahradit View za jiné a použít stejný Model.

MVC umožňuje vyvíjet prezentační vrstvu pomocí test-driven přístupu vývoje.

2.3.2 MVVM

Druhý prezentační návrhový vzor je MVVM neboli Model-View-ViewModel. Svou strukturou si je velmi podobný s MVC. MVVM je vzor použitý k rozdělení a strukturování objektů do tří separátních skupin.

1. Model - obsahuje data aplikace
2. View - zajišťuje zobrazení dat a elementů k interakci s uživatelem
3. ViewModel - zajišťuje transformaci dat modelu na tvar, který je vhodný pro zobrazení

2.3.2.1 Model

Model je základní stavební blok tohoto vzoru. Slouží jako kontejner pro data a informace zobrazené uživateli. Jako příklad Modelu můžeme opět použít entitu „letenka“, která obsahuje položky jako destinace, čas odletu a délku letu. Důležité je si uvědomit, že Model slouží pouze jako kontejner na data a neobsahuje žádné další chování navíc, to znamená, že neobsahuje business logiku, nemodifikuje data a nevolá žádné služby. Také se nestará o způsob, jak jsou data reprezentována uživateli, nestará se o barvu, velikost ani podbarvení textu.

Business logika je většinou uchovávána mimo Model ve speciální třídě, která nad daným modelem operuje a modifikuje jej.

Udržet Model ve stavu, který odpovídá reprezentaci reálného světa a nutných dat pro reprezentaci domény není lehké ani po stránce samotných proměnných dané entity. Když se vrátíme k příkladu s letenkou, tak zmíněná konečná destinace je reprezentována pomocí reference. Abychom tuto referenci dokázali získat, musí mít model proměnnou s identifikátorem odkazujícím na tato data.

Ve většině případů se nedá Model implementovat jako čistý kontejner jen s daty zobrazených uživateli, a hledá se ideální kompromis. Špatný návrh Modelu vede k vyšší provázanosti a horší udržitelnosti systému. Je tedy potřeba se při návrhu a implementaci zamyslet nad strukturou Modelu a snažit se tam dát opravdu jen to potřebné.

2.3.2.2 View

View v MVVM je jedinou částí, kterou koncový uživatel vidí a se kterou může jistým způsobem interagovat. Funguje primárně pro zobrazení dat uživateli pomocí uživatelského rozhraní.

Jako vstup pro zobrazení přijímá model a následně reprezentuje data v modelu na formát vhodný pro koncového uživatele. Například pokud v modelu

máme datum narození, můžeme chtít uživateli vypsat vypočtený věk a nebo podbarvit text na základě nějaké podmínky.

Zobrazení textu není jedinou funkcí, kterou View vykonává. Slouží pro detekci uživatelského vstupu a na jeho základě dochází k manipulaci s daným Modelem, oproti klasickému View (z MVC).

2.3.2.3 Výhody

Poněvadž si je MVVM velmi podobný s MVC, i jejich výhody jsou do jisté míry totožné. Paralelní vývoj a testování jednotlivých částí, znovupoužitelnost View i Modelu a v neposlední řadě flexibilní výměna View nad stejnými daty.

Kapitola 3

Návrh

Pro demonstraci použití architektur rozebraných v předešlé kapitole naimplementujeme ukázkovou aplikaci zabývající se rezervací filmů a knih.

Abychom mohli dojít k rozhodnutí o architektuře, uvedeme si seznam několika uživatelských požadavků, které budeme od aplikace vyžadovat. Požadavky jsou rozřazeny do dvou skupin. První skupina obsahuje funkční požadavky, to jsou požadavky na chování a funkce aplikace. Druhá skupina obsahuje nefunkčních požadavků, tedy požadavky na technologickou stránku aplikace, jako například databáze a architektura.

Z důvodu, že aplikace má fungovat jako ukázka pro implementaci architektur, mají webové aplikace pouze limitovaný výčet funkcí. Dále systém nebere v potaz rozdíl mezi uživateli a zabezpečení jednotlivých sekcí aplikace. Požadavky na aplikaci jsou vypsané v tabulce níže (zkratka P značí funkční požadavek, N značí požadavek nefunkční):

1. P1 - výpis zásilek s novými zásobami
    a) Uživatel bude mít možnost si zobrazit seznam naskladněných zásilek s novým zbožím
2. P2 - synchronizace skladu s katalogy filmů a knih
    a) Informace o naskladnění nových filmů a knih se přenese do nabídky filmů a knih
3. P3 - evidence nové zásilky
    a) Uživatel může zadat informaci o naskladnění nového zboží
4. P4 - výpis knih a filmů dostupných na skladu
    a) Uživatel si může zobrazit přehled dostupných filmů a knih
5. P5 - přehled o rezervování a dostupnosti knih a filmů

   a) Uživatel u výpisu knih a filmů uvidí, kolik kusů je dostupných pro zvolené datum rezervace

6. P6 - rezervace knih a filmů

   a) Uživateli bude umožněno zadat datum do kdy chce knihy rezervovat, vybrat si množství knih a filmů, a provést jejich rezervaci

7. P7 - přehled o rezervacích

   a) Uživateli bude umožněno nahlédnout na přehled všech rezervací

8. P8 - ukončení rezervace

   a) Uživatel může ukončit z přehledu rezervací platnost zvolené rezervace a tím opět zpřístupnit zboží k dalšímu rezervování

9. N1 - komunikace s archivem autorů

   a) systém bude komunikovat se službou poskytující informace o autorech

10. N2 - škálovatelnost systému

    a) systém bude připraven na potřebu škálování

Na základě těchto požadavků můžeme dedukovat existenci následujících domén:

1. katalog autorů - simulace externí služby s autory (N1)
2. katalog knih (P2, P4)
3. katalog filmů (P2, P4)
4. centrum skladového hospodářství (P1, P2, P3)
5. rezervace knih a filmů (P5, P6)
6. statistické přehledy (P7, P8)

3.1 Architektura

Po specifikování domén, definující naši aplikaci, můžeme přistoupit k návrhu architektury. Aplikaci tvoří 5 domén a jedna externí služba. Na základě nefunkčního požadavku N2 ohledně připravenosti na možnost škálování jednotlivých funkcí se nabízí implementace pomocí mikroslužeb. Napomáhá tomu fakt, že se systém skládá z několika domén.

Jednotlivé domény aplikace se rozdělí do samostatných služeb. Každá z těchto služeb by měla reprezentovat nějaký z architektonických vzorů zanalyzovaných v dřívější kapitole.

Služby v rámci systému budou mezi sebou komunikovat pomocí asynchronních komunikačních kanálů. Se službou simulující externí archiv autorů se bude komunikovat pomocí synchronních komunikačních kanálů. Toto rozdělení nám umožní demonstrovat oba způsoby komunikace mezi službami.

Celkovou strukturu aplikace tedy utvoří množina služeb a sdílených knihoven, tento styl odpovídá architektuře mikroslušeb.

Pro uskutečnění zmíněné asynchronní komunikace se zavede message broker, konkrétně *RabbitMQ*. *RabbitMQ* je široce rozšířený a běžně používaný message broker a z tohoto důvodu byl zvolen pro tuto ukázku. Na jeho popis a výhody se zaměříme v sekci o technologiích 3.2.

Při bližším prozkoumání požadavků si můžeme všimnout, že se systém skládá ze 3 větších logických celků a to:

1. sklad
2. rezervace
3. přehledy

Pro tyto celky se vytvoří webové aplikace, které zastřeší logiku daného celku. Pro izolování těchto celků mezi sebou se použije jednotného rozhraní pro každou ze služeb a to pomocí *Ocelot Gateway*. Stejně jako *RabbitMQ* i *Ocelot Gateway* je často používaná technologie, více si řekneme opět v sekci o technologiích 3.2.

Následující tabulka vyobrazuje propojení zmíněných *Gateway* zastřešující svou část logiky a množiny služeb v systému.

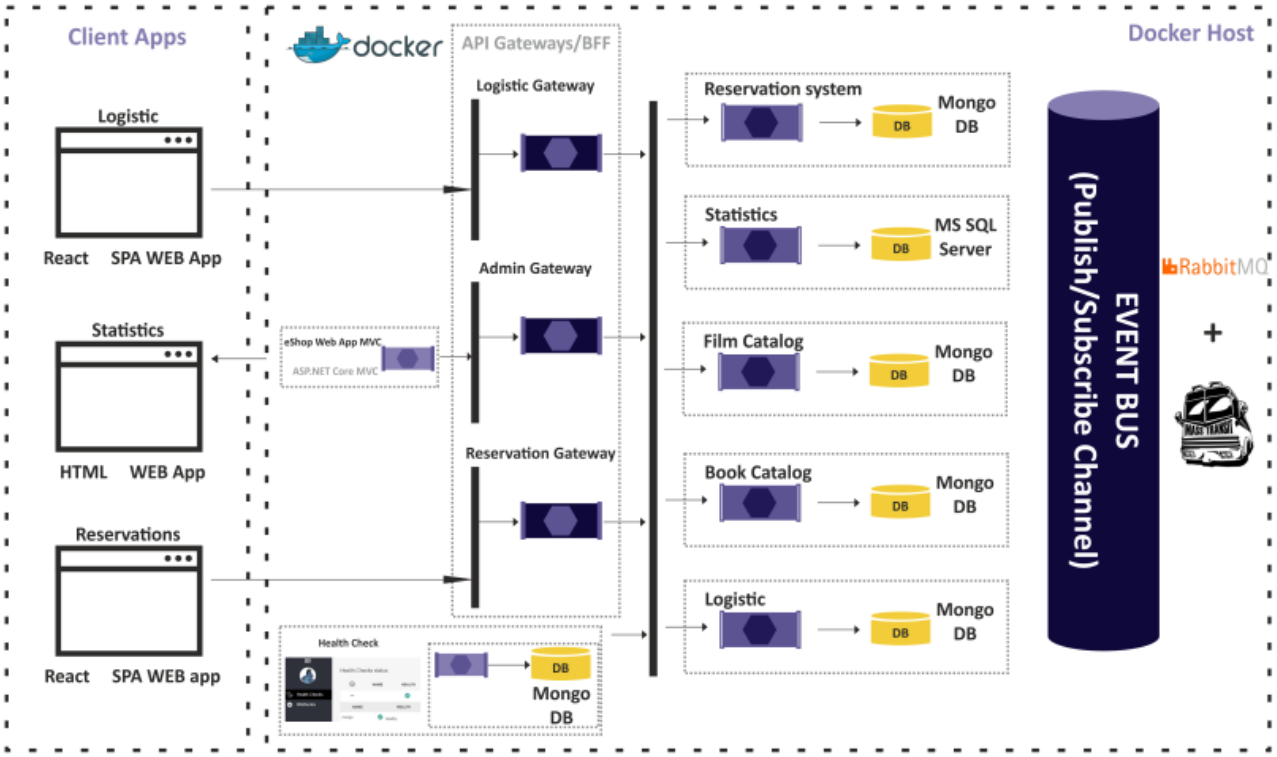
Tabulka 3.1: Pokrytí služeb pomocí jednotných rozhraní

	rezervace	sklad	přehledy
katalog autorů			x
katalog filmů	x		
katalog knih	x		
sklad. hospodářství		x	
rezervační systém	x		x
statistické přehledy			x

Protože každá služba manipuluje s daty, bude pro každou ze služeb vytvořena vlastní databáze, do které služby budou vkládat svá data. Ostatní služby k těmto databázím nebudou moci přistoupit napřímo, pro získání potřebných dat budou volat odpovídající službu.

Všechny služby včetně Ocelon Gateway, databází i RabbitMQ poběží v Docker kontejnerech pro jednodušší nasazení a škálování.

Celkový návrh architektury projektu je zachycena na obrázku 3.1.



Obrázek 3.1: Architektura projektu

3.2 Technologie

V této sekci si řekneme něco o technologiích, které během implementace použijeme, popíšeme jejich funkce a případné alternativy.

3.2.1 C# a .Net

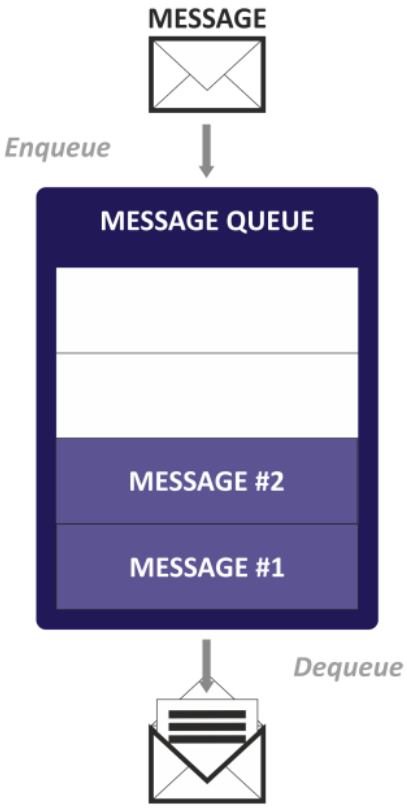
C# je programovací jazyk vyvinutý společností Microsoft. Aplikace napsané pomocí jazyka C# se spouští v prostředí *.Net*.

C# je moderní, objektově orientovaný, typový a imperativní programovací jazyk. Jeho základy jsou podobné jazyku C a Java[16].

.Net je multiplatformní, open-source platforma pro tvorbu širokého spektra aplikací. Platforma umožňuje vývoj ve více jazycích než pouze v C#, podporuje například i F# a Visual Basic[16].

3.2.2 RabbitMQ

Jedná se o open source *Message-Queue* framework[17]. Tento druh softwaru se také vyskytuje pod názvem *Message Broker* nebo *správce fronty*[18].



Obrázek 3.2: Fronta zpráv

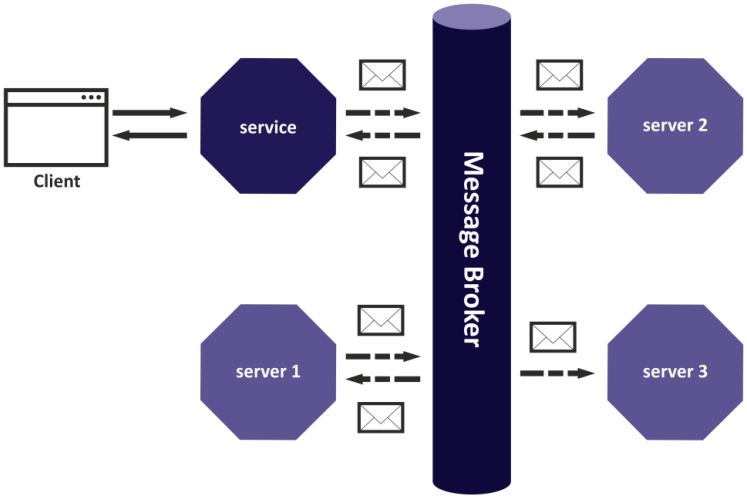
Využívá *publish-subscribe* model ke směrování dat mezi uzly, co zprávy vysílají (publisher) a těmi, co data přijímají (consumer). Model *publish-subscribe* je identický s modelem *producer-consumer*.

Obrázek 3.3: Producer-Consumer model

Poskytuje webové rozhraní pro monitorování jednotlivých front a jejich správu.

3.2.2.1 Popis

Message Broker obsahuje fronty, ke kterým aplikace/služby mohou přistoupit a pomocí nich rozesílat zprávy. Zprávy mohou obsahovat jakýkoliv typ informace, mohou se využít pro rozesílání akcí, událostí i příkazů [18].



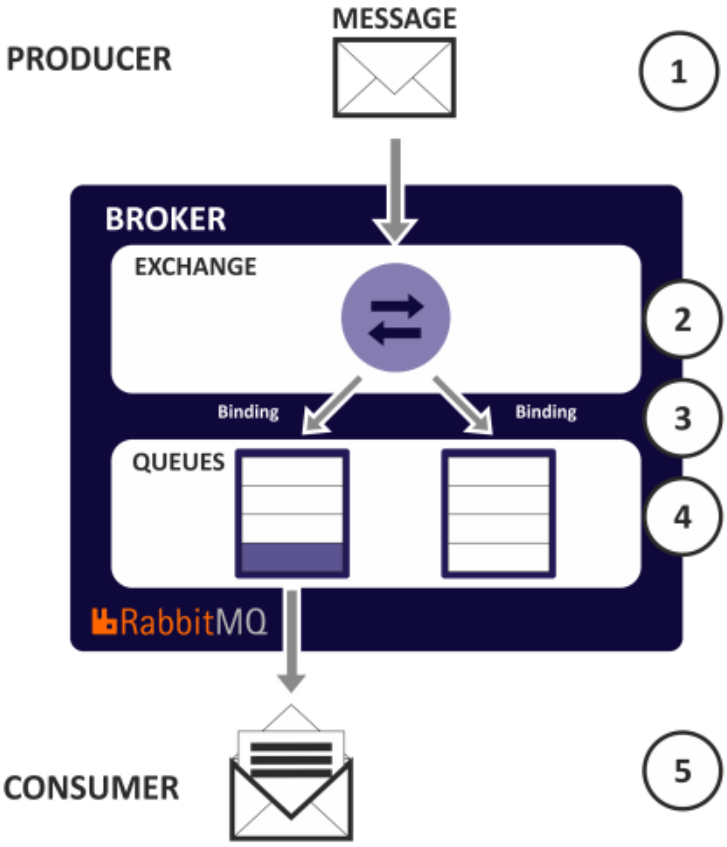
Obrázek 3.4: Komunikace pomocí sdíleného kanálu

Kromě informování, že došlo k události v jedné službě, se používají také k posílání dat mezi účastníky komunikace. Řekněme, že vznikla nová entita a potřebujeme o jejím vytvoření informovat okolí, můžeme jí pomocí Message Brokera rozeslat ostatním účastníkům.

RabbitMQ ukládá zprávy dokud nejsou někým přijaty/zkonzumovány. Jakmile někdo potvrdí přijetí zprávy, je tato zpráva odebrána z fronty. To je velkou výhodou tohoto druhu komunikace, protože zajišťuje, že i když je konzument dané zprávy v dané chvíli zaneprázdněn anebo neaktivní, tak jakmile je konzument schopen zprávu odbavit, bude na něj daná zpráva čekat a neztratí se. U synchronní komunikace by ve stejném scénáři došlo buď k timeoutu, opakovanému zkoušení o navázání komunikace nebo nejhůře ztracení zprávy samotné.

Rozesílání zpráv při vzniku událostí není jediná funkce, kterou Message Broker nabízí. *RabbbitMQ* dokáže sloužit i jako takzvaný *Load balancer*. To znamená, že dokáže zajistit rovnoměrnou distribuci zpráv mezi konzumenty a snaží se zabránit zahlcení konzumenta. Výsledkem je rychlejší a efektivnější zpracování zpráv a využití prostředků.

Rozesílání zpráv se v *RabbitMQ* realizuje pomocí takzvaného *Exchange* prvku. Zprávy nejsou odesílány přímo do fronty, místo toho producent zpráv publikuje danou zprávu do *Exchange* a ta je zodpovědná za směrování zpráv do příslušných front pomocí směrovacích klíčů.

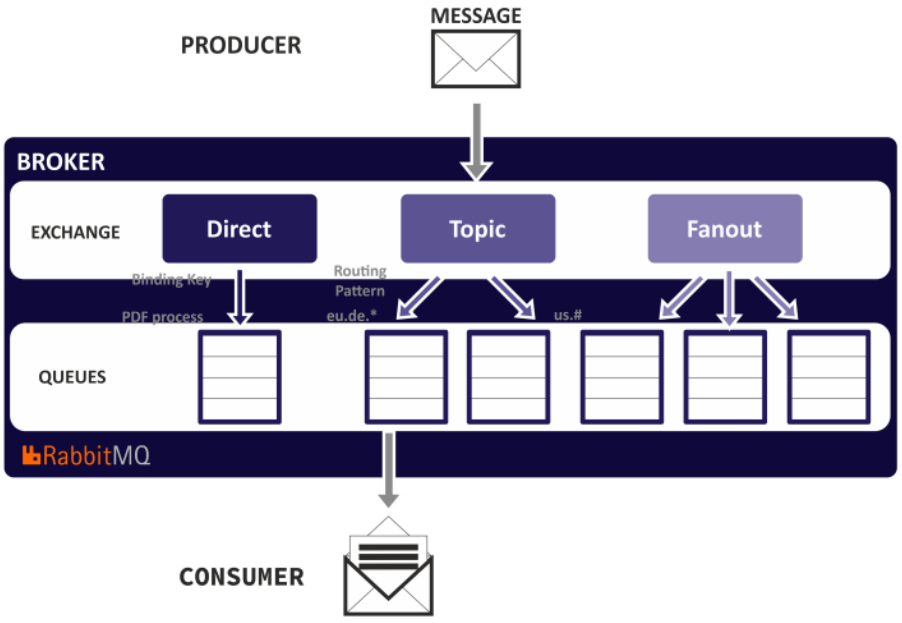


Obrázek 3.5: Průběh zpracování zprávy v RabbitMQ

Postup při publikování nové zprávy je následující:

1. producent publikuje zprávu
2. *Exchange* přijme zprávu
3. na základě atributů zprávy zařídí její přesměrování do odpovídající fronty
4. zpráva zůstane ve frontě až do okamžiku, než je nějakým konzumentem přijata
5. konzument přijme zprávu a ta se odebere z dané fronty

Existují čtyři typy *Exchange* prvku: základní, který se používá, pokud není specifikováno jinak, se nazývá *Direct*. Dále existují *Topic*, *Fanout* a *Headers*. Každý typ je specifický tím, jakým způsobem provádí směrování zpráv do front na základě uvedeného *Routing Key*.



Obrázek 3.6: Exchange typy u RabbitMQ

3.2.2.2 Direct Exchange

Jedná se o nejjednodušší směrovací způsob [19]. Zpráva je zařazena do fronty, jejíž klíč (identifikátor) je uveden ve zprávě.

Tento typ je vhodný například pokud chceme v rámci *Exchange* rozlišit zprávy, protože pro každý klíč je vytvořena vlastní fronta.

Uveďme si funkci na příkladu, kdy máme jednu *Exchange* a v ní dvě fronty: `rezervovat_knihu_queue` a `rezervovat_film_queue`. Pokud publikujeme zprávu s klíčem **rezervovat_knihu**, je tato zpráva zařazena do první zmíněné fronty. Naopak pokud by obsahovala klíč **rezervovat_film**, byla by zařazena do druhé fronty. Dále v případě, že by zpráva obsahovala jiný klíč než uvedený v příkladu, zpráva by byla zahozena. Poslední situace je, když máme dvě fronty se stejným klíčem. V takovém případě je zpráva broadcastována do obou front.

3.2.2.3 Topic Exchange

Tento typ je velmi podobný předešlému typu. Největší rozdíl je ve způsobu provedení směrování zpráv. Směrování se provádí buďto na základě klíče obsahující *Wildcard* nebo na základě vzoru.

Zprávy jsou přesměrovány do jedné nebo více front podle toho, jestli zadaný vzor odpovídá klíči dané fronty. Vzor se skládá z několika částí, jednotlivé části jsou od sebe odděleny tečkou. Krom textových názvů klíčů lze použít speciální znaky * a #.

Na následujícím příkladu si ukážeme konkrétní použití. Definujeme si klíče pro jednotlivé fronty:

1. common.logs.info
2. reservationSystem.bookCatalog.creation
3. reservationSystem.bookCatalog.reservations
4. reservationSystem.filmCatalog.reservations
5. reservationSystem.authorCatalog.creation

Pokud bychom nyní poslali zprávu a využívali typ *Exchange Topic* se vzorem *common.logs.info*, zpráva by byla přesměrována pouze do 1. fronty, protože žádná jiná nesplňuje uvedený vzor.

Pokud bychom místo toho použili *reservationSystem.*.reservations*, zpráva by byla přesměrována do fronty 3 a 4. Znak * tedy umožňuje nespecifikovat konkrétní hodnotu na dané pozici v klíči.

Poslední příklad vzoru pro směrování je *reservationSystem.bookCatalog.**, zde bude zpráva zařazena do front 2 a 3 a to z důvodu, že znak „#“ označuje všechny klíče, které mají stejný prefix klíče až k danému znaku a na konci klíče již nezáleží.

3.2.2.4 Fanout Exchange

Fanout funguje na principu *Broadcastu*. Jakmile přijme zprávu, rozešle jí všem bez ohledu na to, jaký klíč má daná zpráva specifikovaný.

Vhodné použití tohoto typu je například v případě, kdy chceme všechny uzly informovat o nějaké významné události.

3.2.2.5 Headers Exchange

Posledním typem je *Headers Exchange*, který také nevyužívá klíčů.

Pro své routování využívá jiného mechanismu a to atributů v hlavičce příchozích zpráv. Mezi *Exchange* a frontou/frontami je nastaven speciální argument *x-match*, který může nabývat hodnoty „all“ anebo „any“. Na jednotlivých frontách se nastaví pravidla ve tvaru Key=Value. Tyto pravidla specifikují, jaké atributy a hodnoty musí příchozí zpráva obsahovat v hlavičce, a argument *x-match* určuje, jestli musí být splněny všechny stanovené atribut,y anebo stačí alespoň jeden pro to, aby příchozí zpráva byla zařazena do dané fronty. Je potřeba si dávat pozor, aby uživatelský klíč neobsahoval prefix „x-“, protože takové klíče jsou ignorovány.

Pro lepší pochopení si uvedeme příklad i u tohoto typu. Řekněme, že máme 4 fronty. Pro frontu A,B a C musí hlavička zprávy obsahovat všechny vypsané atributy. Fronta D se použije pokud aspoň jeden ze dvou uvedených atributů bude splněn.

1. Fronta A - loguje vznik nových knih
    a) x-match=all
    b) catalog=book
    c) type=new
    d) action=log
2. Fronta B - loguje vymazání filmu z katalogu
    a) x-match=all
    b) catalog=film
    c) type=removed
    d) action=log
3. Fronta C - vypisuje údaje aktualizovaných knih
    a) x-match=all
    b) catalog=book
    c) type=update
    d) action=output
4. Fronta D - je informována, že dochází k výpisu anebo odebrání prvku z katalogu
    a) x-match=any
    b) type=removed
    c) action=output

3.2.2.6 Výhody a nevýhody

Přiblížíme si nyní výhody a nevýhody *RabbitMQ* [20].

Výhody

1. rychlost front
2. spolehlivost
3. flexibilní směrování
4. posílání zpráv přes širokou škálu protokolů
5. podpora mnoha jazyků [21]
6. obsahuje uživatelské rozhraní pro monitorování a správu

7. open-source
8. velké množství pluginů a tutoriálů

Nevýhody

1. neumožňuje dávkovat zprávy, odesílají se jednotlivě
2. konfigurační soubory napsány v jazyce Elang
3. není k dispozici historie zpráv

3.2.2.7 Alternativy

Mezi alternativní Message Brokery patří například Kafka, Kinesis, ZMQ, ActiveMQ, RocketMQ, AWS SQS a Redis.

Kafka je distribuovaný Message Broker, je tolerantní při vzniku chyb a je horizontálně škálovatelný. Využívá principu *rozděl a panuj* a díky tomu dosahuje vysoké dostupnosti a vysoké škálovatelnosti, stačí přidat nový server v průběhu běhu aplikace. Hlavní výhodou oproti *RabbitMQ* je možnost přečíst si historické zprávy, které jsou již zpracovány. Tato doba uchování zprávy se dá nastavit v konfiguraci Apache Kafka.

Redis je primárně databáze, kterou lze využít i jako Message Brokera. Oproti *RabbitMQ* nezaručuje doručení zpráv a je pomalejší na poli doručování velkých zpráv. Výhodou je, že může zároveň sloužit jako databáze i cache. Dále, s přibývajícími konzumenty a počtem zpráv se nezpomaluje tak výrazně jako RabbitMQ, což má za následek, že dokáže zpracovat až milion zpráv za vteřinu, oproti *RabbitMQ* s padesáti tisíci [22][23].

3.2.3 MassTransit

MassTransit je open-source framework, který přidává nadstavbu pro komunikaci pomocí zpráv a Message Brokera. Hlavním cílem je přidat konzistentní využívání těchto brokerů tím, že přidává vrstvu abstrakce nad Message Brokerem.

Tato abstrakce umožňuje využívání širokého spektra front nezávisle na tom, jakého poskytovatele jsme si zvolili, a tím snížit provázanost celého systému. Můžeme využívat například RabbitMQ zmíněný v dřívější kapitole a v průběhu života aplikace zaměnit za jiného bez nutnosti přepisování kódu.

3.2.3.1 Výhody

Paralelní zpracování přidané tímto frameworkem umožňuje, že konzumenti mohou asynchronně zpracovávat zprávy a tím maximalizovat efektivnost a rychlost komunikace [24].

Pokud je uzel nedostupný, Message broker jako RabbitMQ udržuje informace o zprávách pro následné doručení, jak jsme si již řekli v předešlé kapitole.

MassTransit navíc zařizuje, aby aplikace po výpadku plynule zahájila spojení s frontou z druhé strany.

Další výhodou je **jednotná logika serializace** objektů pro komunikaci pomocí fronty. Může se stát, že každý broker přijímá zprávy v jiném formátu, RabbitMQ například pracuje s *byte* jednotkami. MassTransit se stará o to, aby vývojáři mohli předávat objekty ve specifické formě pro C# jazyk, to znamená staticky typované objekty, a on následně provede konverzi na typ, který fronta očekává. Toto je jedna z klíčových vlastností a výhod, umožňující nízkou provázanost systému.

Zprávy mohou být zašifrované pomocí AES-256, čímž se zaručí jejich bezpečné přeposílání mezi službami.

Framework se stará o komunikaci samotnou, ta obsahuje jak potvrzování přijetí zprávy tak naopak její chybový stav při nedoručení. Framework si udržuje dvě fronty: *skipped queue* a *error queue*, do kterých dává zprávy, které nedošly uživateli a nebo během jejich zpracování došlo k chybě. K těmto frontám lze jednoduše přistoupit a rychle zjistit, jaký je stav zpráv.

Stejně jako RabbitMQ i MassTransit obsahuje monitorovací nástroje, které lze použít pro sledování průtoku zpráv službou, její zatížení a výsledný výkon.

Pokud je potřeba naplánovat odeslání zprávy do budoucna, MassTransit tuto funkci umožňuje.

Poslední zmíněnou výhodou je ulehčení testování. In-memory fronty, které dokáže tento framework simulovat, ulehčují proces unit testování. K dispozici je knihovna vyvinuta touto společností, která obsahuje logiku pro spuštění i deaktivaci těchto front, takže během testování se stačí soustředit čistě na testování logiky aplikace a oprostit se od věcí týkající se infrastruktury.

3.2.4 Docker

Docker je nástroj, který je používán k automatizovanému nasazování aplikace pomocí kontejnerů, tak, že aplikace dokáže běžet efektivně a nezávisle na prostředí, ve kterém se reálně nachází.

Kontejner je kus softwaru, který obsahuje závislosti potřebné ke spuštění aplikace [25]. V tomto kontejneru se nachází izolovaná aplikace, která zde běží nezávisle na ostatních aplikacích v dalších kontejnerech. Docker eliminuje kopírování operačního systému do každého kontejneru. Tato vlastnost způsobuje, že Docker je méně náročný na zdroje (operační i úložné) i přes to, že poskytuje stejné funkce jako virtuální stroj. Podrobnější porovnání je v kapitole 3.2.4.1.

Docker se snaží řešit problém ohledně využívání frameworků, prostředí a jiných zdrojů. Tím, že každá aplikace běží ve vlastním kontejneru, může si do toho kontejneru nastavit prostředky a technologie, které jsou pro danou aplikaci potřebné. Stejným způsobem si mohou nastavit prostředky i další aplikace ve svých kontejnerech a nedochází k žádnému střetu verzí a technologií [26].

Řekněme například, že máme historickou aplikaci napsanou v technologii .Net verze 4, ale nové části systémů resp. jiné služby implementujeme pomocí moderního multiplatformního .Net Core. Mezi těmito dvěma technologiemi je nekompatibilita.

Jedním z řešení je mít jednotlivé části systému rozdělené na více strojích, tím nedochází ke sdílení prostředků. Toto řešení je ale neefektivní a vyžaduje vlastnění více hostovacích strojů.

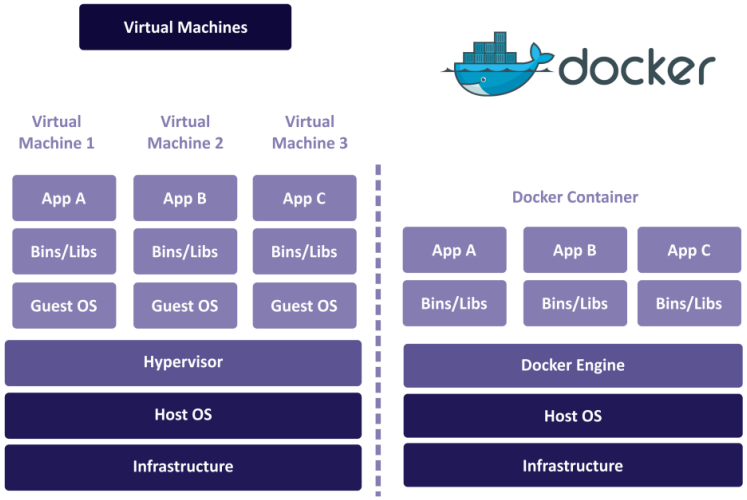
Řešení druhé je vytvoření virtuálního stroje a do něj vložit vše potřebné pro danou aplikaci. Tímto jsme schopni zajistit, aby celá aplikace byla hostována na stejném stroji a aby nedocházelo ke kolizi infrastrukturních prvků - v našem případě jde o prostředí .Net.

Třetí řešení je využít technologie Docker, jednotlivé části systému uzavřít do vlastního kontejneru a do nich nastavit prostředí, které je potřeba. Výsledek je tedy podobný, téměř identický na pohled, s řešením číslo dvě. Na rozdíly v těchto dvou řešeních se podíváme v následující kapitole.

3.2.4.1 Virtuální stroj a Docker

Tato sekce se zabývá rozdílem mezi Virtuálním strojem a Dockerem. Oba na první pohled vypadají velmi podobně, rozdíl je právě ve způsobu sdílení operačního systému, jak již bylo nastíněno v předešlé sekci.

Docker redukcí kopírování operačního systému nedosahuje jen menšího využití zdrojů, ale také tím ovlivňuje samotné zapínání a nastavování, neboli *Boot-up time*, stroje samotného. I ten je odlišný a to ve prospěch Dockeru.



Obrázek 3.7: Porovnání virtuálního stroje a Dockeru

Největší výhodou je však přenosnost zařízení. Zatímco Docker je velmi lehce přenositelný mezi jednotlivými platformami a jednoduše konfigurova-

telný pro spuštění na novém stroji, tak virtuální stroje se potýkají s mnohými problémy.

3.2.4.2 Jak funguje Docker

Docker se skládá z několika částí. Základem všeho je *Docker Engine*, který slouží k sestavení a spuštění kontejnerů na hostovacím zařízení, neboli na stanici, kde chceme aplikaci spustit [26].

Využívá Client-Server architektury, klient rozhraní nainstalované na hostovací stanici komunikuje s *Docker Daemon* (server) a řídí sestavení, spouštění a distribuci kontejnerů. Server neboli *Docker Daemon* je persistetní proces, který běží na pozadí, a spravuje kontejnery na daném hostu. Je to samostatně funkční a výkonný proces, který se stará o připojení k síti, správu obrazů operačních systémů, úložiště atd... Server kontroluje, jakým způsobem je klient vytvářen a že vše funguje dle předpokladů. Komunikace mezi Serverem a Klientem probíhá pomocí Rest API rozhraní. Tento fakt umožňuje, že je možné s tímto rozhraním komunikovat a naprogramovat si nad ním nadstavbu [26].

Komunikovat a zadávat příkady Dockeru lze pomocí konzolového příkazového řádku nebo pomocí nějakého grafického rozhraní jako například *Docker Hub*[27].

Ve chvíli, kdy uživatel zadá příkaz pomocí Docker klienta, ať už pomocí konzole nebo grafického nástroje, tento příkaz klient zvaliduje, přeloží a vyhodnotí. Na základě požadavku začne komunikovat se serverem, který interaguje s operačním systémem.

Většinou nechceme psát jednotlivé příkazy pro sestavení a spuštění opakovaně, obzvlášť pokud se náš systém skládá z mnoha služeb, které mají vlastní kontejnery. Tento problém řeší speciální Docker soubor - *Docker File*. Tento speciální soubor umožňuje zapsat běžné příkazy, které jsou k dispozici v terminálu. Pokud tedy potřebujeme například sestavit jednotlivé aplikace, vypublikovat z nich knihovny, jenž jsou následně využívány službami, tak tento celý proces se zde dá zapsat a při spouštění služeb se provede automaticky na začátku [26].

Docker-Compose je nadstavba Dockeru. Je to aplikace, která využívá soubor Docker-Compose typu yml, neboli obsah tohoto souboru je zapsán pomocí YAML jazyka. Používá se pro orchestraci, konfiguraci a spuštění jak jednokontejnerové tak i více-kontejnerové aplikace. Konfiguruje se zde například na jakém portu kontejner bude přístupný, jaké adresářové adresy bude využívat pro načítání a ukládání dat nebo jaký systémový obraz má použít [28].

3.2.4.3 Komponenty Dockeru

Docker se skládá se čtyř základních prvků. První komponentou je dvojice Klient-Server. O této dvojici jsme již mluvili v předchozí sekci, takže ji zde přeskočíme.

Druhou komponentou jsou *Docker Images* občas také označeny jako *Snapshots*. Jedná se o obraz systému v danou chvíli. Na rozdíl od obrazů používaných ve virtuálním stroji nemohou být po vytvoření modifikovány. Pokud potřebujeme obraz opravit, musíme jej smazat a vytvořit znovu. Tato vlastnost se může zdát jako nevýhodou Dockeru, ale opak je pravdou. Tato vlastnost totiž zaručuje, že po nastavení a vytvoření image, jakmile vše funguje jak potřebujeme, bude takto fungovat vždy stejně. Nelze jej modifikovat a způsobit rozbití image. Po nakonfigurování *Image* jej lze sdílet s ostatními lidmi pomocí registrů. Uživatel, který si stáhne vypublikovaný obraz, si spustí nový kontejner s naším obrazem a nejenže bude fungovat stejně jako u uživatele, co obraz publikoval, bude obsahovat i ta samá data, aplikace, nastavení a nakonfigurované závislosti [29].

Dále je zde komponenta reprezentující kontejnery (běžně se využívá anglický tvar Container). Jedná se o softwarový balíček obsahující aplikaci a jejich závislosti. Funguje nezávisle na ostatních a lze jej spouštět samostatně. Kontejnery běží izolovaně na stejné infrastruktuře hostovacího zařízení a mohou tak sdílet operační systém s dalšími kontejnery. Pokud tedy dva různé kontejnery, a zde se nemusí jednat o dvě různé aplikace, ale například dvakrát spuštěnou stejnou aplikaci, využívají stejný operační systém, mohou si jej sdílet navzájem a tím se nemusí duplicitně nahrávat do paměti. Zde narážíme na téma, které jsme probírali v porovnání oproti virtuálnímu stroji v sekci 3.2.4.1 [26].

A poslední komponentou je Docker registr. Registr jsou open-source server-side služby používané pro hostování a distribuci systémových obrazů (*Image*). Pomocí tohoto volně dostupného registru můžeme například sdílet naše *Images* s ostatními členy týmu nebo znovu použít v dalších projektech. Registry jsou jak veřejné tak soukromé. Ve veřejném registru existuje mnoho systémových obrazů vytvořených komunitou a lze zde najít již hotová řešení problémů, které si stačí stáhnout a spustit.

3.2.5 Ocelot GateWay

Jak je již z názvu zjevné, funkcí Ocelot Gateway je zajištění jednotného bodu přístupu do aplikace. Jedná se o knihovnu specifickou pro .Net technologie a nejčastější využití je v případě mikroslužeb[30].

Pokud aplikace obsahuje více služeb, které operují pod jim přiděleným portem, pak pro přístup k těmto službám potřebujeme dané přidělené porty znát. Pokud bychom tyto porty neznali, nevěděli bychom na jakou adresu delegovat dotazy[30].

Ocelot Gateway se stará o to, aby se služby tvářily jako celek vystupující pod jednou adresou. To má hned dvě výhody, ta první je, že nemusíme na klientovi znát porty našich služeb, ale pro všechny dotazy používáme jednotnou adresu. Druhá výhoda vyplývá z té první - pokud by se časem změnil port nějaké služby, museli bychom tuto změnu reflektovat na všech místech, kde

se adresa využívala. Ale protože při použití Gateway aplikace s porty nemusí pracovat, nemusí zmíněnou změnu portů služeb řešit.

3.2.5.1 API Gateway

API Gateway implementuje vzor, o kterém jsme mluvili v kapitole ohledně mikroslužeb, konkrétně sekci návrhových vzorů 2.1.2.5.

Pro zopakování, tento vzor se snaží řešit problém s přístupem k jednotlivým službám z uživatelské aplikace. Pomocí vytvoření jednoho bodu přístupu, tedy *Gateway*, probíhá komunikace mezi uživatelskou aplikací a jednotlivými službami. Tento bod funguje jako fasáda, která ukrývá komplexitu a rozdělení služeb před přistupujícími klienty. V praxi se jedná o další službu, která deleguje dotazy na odpovídající služby.

Kromě ukrytí komplexnosti aplikace může být využita i pro kontrolu práv přístupu k daným službám na jednom místě a tím opět docílit menší provázanosti kódu a redukce opakovaného kódu[30].

3.2.5.2 Výhody

Krom zmíněné agregace požadavků pomocí jednoho bodu přístupu a ověřování práv, případně přihlašování, nabízí Ocelot GateWay ještě mnoho dalších výhod. Umožňuje například funkci vyhledávání služeb (v angličtině *Service Discovery*), cachování dotazů, Load Balancing a jiné [31].

Dále umožňuje posílání dotazů pomocí WebSocket technologie, tato funkcionalita je však teprve v aktivním vývoji a její funkcionalita je limitována.

Podporuje využívání Kubernetes, změnu konfigurace jednotlivých Gateway během běhu aplikace a napojení případného dalšího middlewaru mezi služby a klientské aplikace.

3.3 Doména

3.3.1 Katalog filmů a knih

Katalogy mají na starosti správu a ukládání informací o knihách a filmech. Při vytváření entit komunikují se službou s autory pro zjištění identifikátoru na základě registračního čísla. Data budou uložena pomocí dokumentové databáze MongoDB, která se hodí na systémy, u kterých nepotřebujeme vyhledávání na základě relačních vazeb. Protože zmíněné služby budou hlavně ukládat data a případně je vyhledávat na základě identifikátoru v jedné tabulce, je tato databáze vhodnou volbou.

Tyto dvě služby a simulovaná externí služba autorů, jako jediné nebudou mít uživatelské rozhraní.

Interakce s nimi budou dvojího typu. Varianta první je pomocí vystaveného webového rozhraní, neboli API, které bude umožňovat přímé volání služby

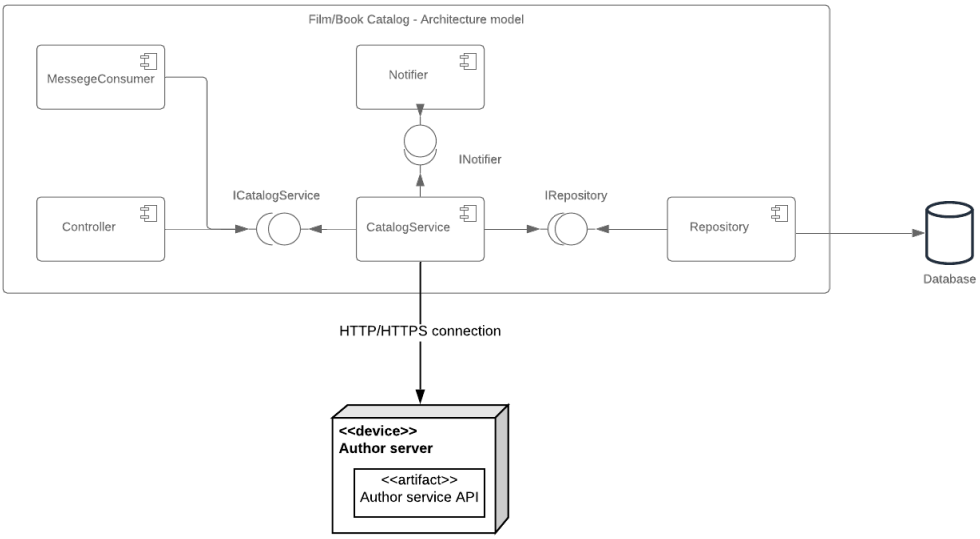
jak z uživatelského rozhraní tak jiných služeb. Druhý typ komunikace bude pomocí nepřímého zasílání zpráv mezi službami.

Architektura těchto služeb je velmi jednoduchá, bude se skládat z pěti částí a to z webového API, třídy zpracovávající příchozí zprávy od jiných služeb, třídy pro odesílání zpráv, jednotky vykonávající logiku aplikace a datové vrstvy pro ukládání dat.

Záštitu nad databází zajistí třída implementovaná podle návrhového vzoru *Repository*. Tento návrhový vzor slouží k přidání abstrakce při přístupu k databázi. Třída implementuje *IRepository* rozhraní, které bude používat *CatalogService* nezávisle na tom, zda-li toho rozhraní implementuje třída s přístupem k MongoDB databázi nebo jiné.

CatalogService obdobně implementuje rozhraní *ICatalogService*. Toto rozhraní bude používat *Controller* starající se o zpracování webových dotazů a *MessageConsumer* zpracovávající komunikaci pomocí zpráv. V rámci své logiky bude *CatalogService* informovat ostatní služby o založení nové knihy, resp. filmu, pomocí notifikační třídy *Notifier* s rozhraním *INotifier*. Abstrakce nad notifikační třídou opět umožní volnost zvolení konkrétní implementace notifikační technologie.

Na následujícím diagramu je zachycen návrh doménových tříd.



Obrázek 3.8: Návrh architektury katalogu filmů/knih

Protože se jedná o jednoduchý katalog na evidenci knih a zboží, je doména tvoře pouze dvěma objekty (2 třídy v katalogu knih a zbylé 2 v katalogu filmů). Objekt transakcí slouží pro historickou evidenci naskladňování, která se zároveň dá využít pro podporu správné komunikace a synchronizace mezi skladem a katalogy.

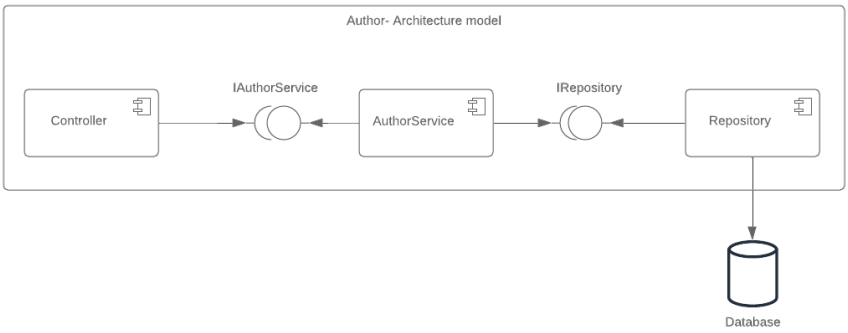
Pro komunikaci s ostatními službami budou vystaveny dva typy zpráv *BookCreated* a *BookSupplyTransactionConfirmation*, resp. *FilmCreated* a *FilmSupplyTransactionConfirmation*. První typ zprávy informuje o vzniku nové entity a bude obsahovat data ohledně vytvořené entity, pro případ, že by některá služba měla tyto data zájem. Druhá zpráva bude sloužit jako potvrzení, že entita byla na základě zprávy úspěšně uložena.

Návrh těchto dvou služeb odpovídá Clean architektuře i Onion architektuře.

3.3.2 Simulovaná externí služba autorů

Jak bylo zmíněno, tato služba má sloužit jako simulace externí služby za účelem ukázky synchronní komunikace.

Architektura je koncipována obdobně jako u katalogů, výjimkou jsou části týkající se zasílání zpráv. Jako externí služba nemůže být napojena na stejný komunikační kanál a služby s ní musí komunikovat napřímo.



Obrázek 3.9: Návrh architektury katalogu autorů

Ve chvíli, kdy služba pro evidenci knih nebo filmů obdrží žádost pro vytvoření odpovídajícího typu entity, odešlou požadavek na ověření autora. Tento požadavek bude odchycen webovým rozhraním, *AuthorService* ji zvaliduje, zpracuje a následně jako odpověď odešle identifikátor autora, za předpokladu, že byl autor nalezen.

Doménový model pro tuto službu je jednoduchý. Obsahuje pouze jednu tabulku autorů s identifikátorem, jménem a registračním číslem, podle kterého služby vyhledávají autora.

3.3.3 Rezervační systém

Rezervační systém je první zmíněnou službou, která bude obsahovat server i webové uživatelské rozhraní.

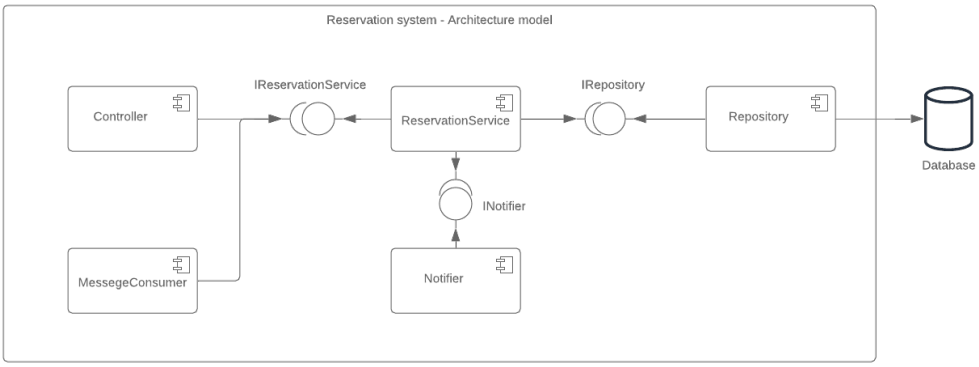
S ostatními službami bude komunikovat pomocí zpráv zaslaných pomocí sdíleného komunikačního kanálu.

3.3.3.1 Server

Server rezervačního systému se implementuje pomocí Onion architektury.

Pro připomenutí, Onion architektura se skládá ze 4 vrstev, z toho *Domain Model* a *Domain Service* tvoří jádro. Další vrstvou je *Application* obsahující logiku aplikace a poslední vrstvou, takzvanou *Outer*, je vrstva obsahující infrastrukturní prvky, jako například databázi a webové rozhraní.

Tohoto rozdělení se bude držet i server rezervačního systému. Jádro aplikace bude obsahovat doménové entity a jejich validační podmínky. Následuje *Application* vrstva, obsahující logiku aplikace. Tato vrstva definuje rozhraní, které mohou využít vrstvy v poslední vrstvě pro připojení nebo implementaci požadované logiky týkající se infrastruktury. Mezi tyto rozhraní patří rozhraní pro přijímání a odesílání zpráv pomocí komunikačního kanálu a ukládání dat do databáze.



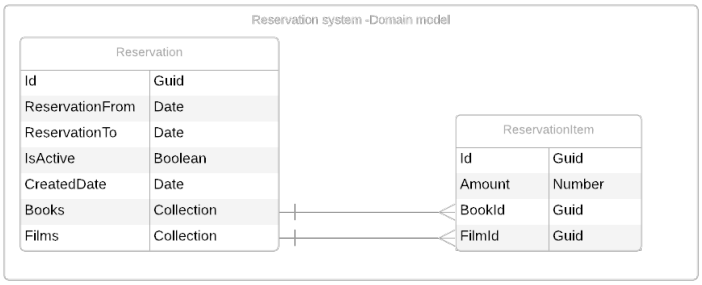
Obrázek 3.10: Návrh architektury rezervačního systému

Z návrhu architektury aplikace si lze všimnout blízká podobnost s návrhem architektury pro katalog filmů a knih. Jediná odlišnost je zde v tom, že rezervační systém nebude komunikovat se žádnou službou napřímo.

Podobnost také plyne z toho, že pro služby byla použita stejná architektura. Struktura se může lišit počtem tříd v *Application* vrstvě, počtem Controllerů, různorodosti infrastrukturních prvků nebo doménou služby, ale princip propojení prvků uvnitř služby zůstává stejný. Tento fakt napomáhá zorientování se v projektu, návrhy služeb jsou cíleně takto rozvrženy pro demonstraci podobnosti, ač se služby zabývají úplně jinou doménou.

Doménu v této službě tvoří uzavřené rezervace a položky rezervací. Rezervace si drží kolekci rezervovaných knih a filmů včetně množství u každého

prvku. Rezervace také eviduje, zdali je ještě platná a období platnosti. Rezervaci je možné předčasně ukončit.



Obrázek 3.11: Doménový model rezervačního systému

Při vytvoření rezervace bude vyslána notifikační zpráva o vytvoření rezervace pomocí sdíleného komunikačního kanálu.

3.3.3.2 Klient

Klientská aplikace bude implementovaná pomocí MVVM architektury.

Uživatelské rozhraní zobrazí přehled všech knih a filmů. Po zadání data se zjistí dostupnost knih v intervalu aktuálního data a zadaného data.

Uživatel si navolí knihy a filmy, které si pro tento interval chce rezervovat a po stisknutí tlačítka bude odeslán požadavek na server, kde se provede následná rezervace.

Komunikace nebude probíhat napřímo se serverem, ale půjde přes gateway. Bez gateway by musela aplikace komunikovat se třemi službami: filmový katalog, knižní katalog a rezervační systém. Pomocí gateway bude komunikovat pouze s jedincem, který následně zařídí přesměrování.

3.3.4 Logistika

Logistický sklad má na starosti naskladňování knih a filmů. Data na server bude umožněno ukládat pomocí uživatelského rozhraní.

Při obdržení žádosti o naskladnění nových zásob, proběhne uložení tohoto požadavku do databáze a odeslání zpráv pomocí komunikačního kanálu pro zaevidování nových knih a filmů.

Sklad při ukládání přiřadí zboží unikátní identifikátor, tento identifikátor bude sloužit pro párování žádosti o zaevidování s potvrzením o zaevidování.

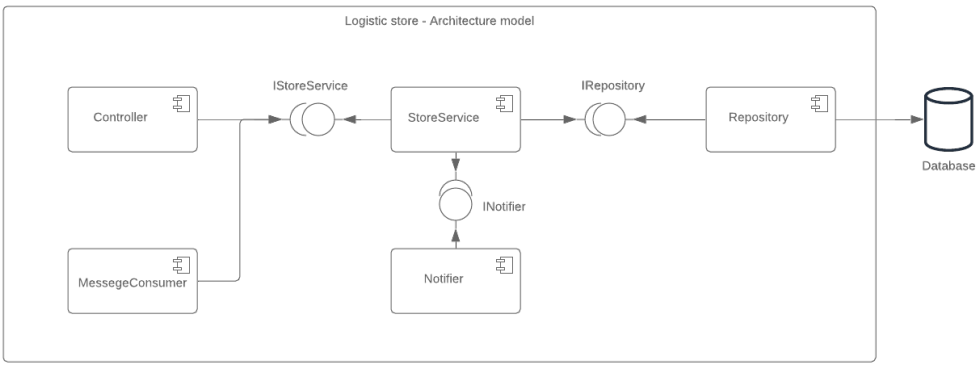
3.3.4.1 Server

Server rezervačního systému se implementuje pomocí Clean architektury.

Opět pro připomenutí, Clean architektura se skládá také ze čtyř vrstev, *Enterprise*, *Application*, *Presentation* a poslední vrstvou je *Infrastructure*.

Architektura služby je tedy navrhnuta tak, aby odpovídala tomuto rozdělení. Strukturou se moc neliší od předchozí Onion architektuře.

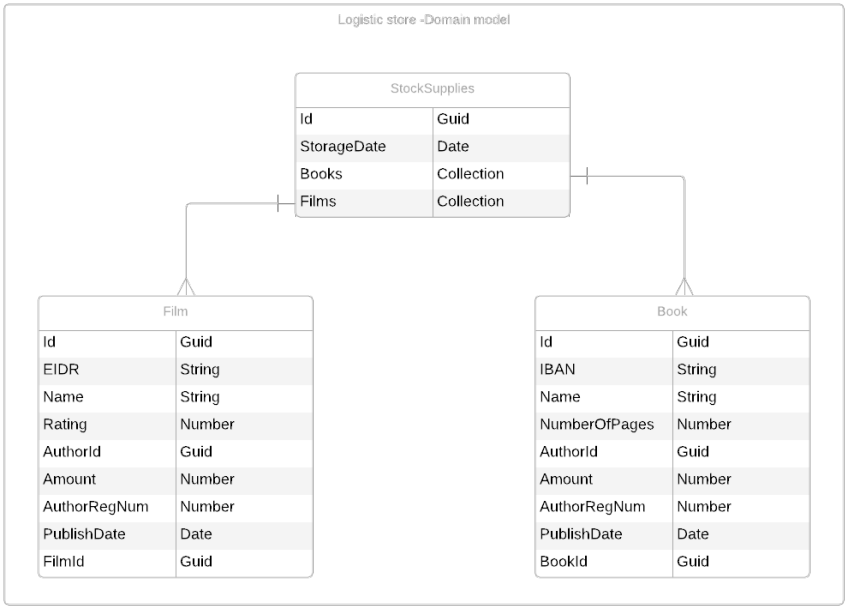
Enterprise vrstva, neboli také nazývaná Doménová vrstva, bude obsahovat entity. Po doménové vrstvě následuje *Application* vrstva, obsahující logiku aplikace. Zde opět budou definovány rozhraní, které implementují třídy z poslední vrstvy. Nakonec je zde vrstva prezentační a vrstva infrastruktury.



Obrázek 3.12: Návrh architektury logistického skladu

Podobnost mezi Clean Architecture a Onion Architecture je tak velká, že návrhy služeb na základě architektur jsou totožné. Jediné rozdíly jsou terminologické pojmenování, které na návrhu architektury a domény nejsou vidět.

Doménu této služby tvoří zásilka se zbožím. Konkrétně se u zásilky bude evidovat datum naskladnění, tzn. zaevidování požadavku a kolekce knih a filmů. Služba k položkám filmů a knih přidá identifikátor pojmenovaný *Id*, který předvyplní, a identifikátor *FilmId*, resp. *BookId*, sloužící pro rozeznání již uložených filmů a knih.



Obrázek 3.13: Doménový model logistického skladu

Po zaevidování požadavku na naskladnění zboží, provede služba uložení tohoto požadavku a pomocí komunikačního kanálu odešle žádost o naskladnění daného zboží. Katalogové služby požadavek zachytí a odešlou odpověď obsahující dvojici identifikátoru. První odpovídá předgenerovanému identifikátoru logistickou službu a druhý odpovídá tomu, pod kterým je evidován v daném katalogu. Po obdržení potvrzení logistický sklad vyhledá pomocí prvního klíče dané zboží a nastaví mu příznak, tzn. hodnotu druhého identifikátoru.

Tímto procesem bude mít logistický sklad přehled o zaevidovaných požadavcích v případě, že by během požadavku došlo k neošetřené chybě.

3.3.4.2 Klient

Klientská aplikace bude také implementovaná pomocí MVVM architektury.

Uživatelské rozhraní na úvodní stránce zobrazí přehled všech zaevidovaných požadavků o naskladnění.

Uživatel bude mít možnost přejít na formulář, kde vyplní zboží pro naskladnění a jeho množství. Po odeslání požadavku je přesměrován na přehled žádostí.

Ani u této klientské aplikace nebude komunikace probíhat napřímo, ale půjde přes gateway. V tomto případě sice nebude muset komunikovat s více různými službami, ale pokud by tomu tak v budoucnu bylo, nebude se muset tato změna řešit na straně klienta.

Kdybychom v klientské aplikaci chtěli například přidat možnost výběru autora knih/filmů pomocí našeptávače a ne napsáním registračního čísla, museli bychom do klientské aplikace později doprogramovat druhý datový zdroj a logiku kolem něj. Jakmile je zavedena gateway, tak přidáme akorát přesměrování do konfigurace a na klientské straně novou metodu, která bude žádat o data.

3.3.5 Statistický systém

Poslední službou reprezentující doménu v našem systému bude aplikace zabývající se výstupy dat. Služba nebude obsahovat žádné složité výstupy, protože to není záměrem této práce.

Místo toho si na službě ukážeme, že lze pro každou službu použít jiné technologie a to konkrétně jiný druh databázového připojení. Oproti ostatním službám bude využívat relační databázi, do které bude ukládat data vzniklá na jiných službách, neboli bude vytvářet kopie dat uložené do jiné databázové struktury pro efektivnější vyhledávání a provádění databázových operací.

Vystavené webové rozhraní bude sloužit pro získávání jednoduchých datových přehledů a pomocí připojení na sdílený komunikační kanál bude odposlouchávat zprávy o vytváření a editaci prvků v jednotlivých službách. Na tyto zprávy nebude nijak reagovat, pouze je zpracuje a promítne dané změny do své databáze.

Statistický systém bude obsahovat uživatelské rozhraní implementující MVC architekturu.

3.3.5.1 Server

Implementace tohoto systému bude uskutečněna pomocí hexagonální architektury. Pro vývoj této služby se použije Domain-Driven design, pomocí kterého se stanoví jaké funkce (*User case*) doména bude mít. Systém bude rozdělen do čtyř prvků, *Domain* a *Application* budou tvořit základ systému. *Domain* obsahuje, stejně jako u předchozích architektur, doménové entity a *Application* bude obsahovat definici rozhraní použitých ve službě a implementaci zmíněných *Use case* scénářů.

Prvky *WebApi* a *Infrastructure* implementují rozhraní, které *Application* vrstva vystavila. Toto propojení odpovídá propojení Adapter-Port, kde Port je vystavené rozhraní aplikační vrstvou a Adapter je konkrétní implementace, kterou zařídí externí součásti služby. Jako externí prvky systému se bere vše, co nesouvisí se základem systému a není důležité pro definici procesů aplikace.

Obrázek 3.14: Adapter a port

Jednotlivé části systému budou implementované jako jednotlivé projekty, které se mezi sebou propojí. Aplikační vrstva a doménová budou napojené napřímo, k aplikační vrstvě se poté připojí pomocí Adaptérů ostatní části systému jako webové rozhraní pro přijímání požadavků klienta nebo databáze.

Tyto externí části budou se systémem komunikovat pomocí modelů, které definuje aplikační vrstva, a budou spouštět vyhodnocení zmíněných scénářů, které aplikační vrstva implementuje.

3.3.5.2 Klient

Klientská uživatelská aplikace umožní nahlédnout na data uložená v databázi, jako například přehled knih, filmů a aktivních rezervací, které zde mohou být ukončeny.

Pro komunikaci se systémem použije speciální *Gateway* zastřešující logiku klientské aplikace statistické služby. Krom komunikace se serverem umožní i komunikaci se službou autorů pro získání detailu autora, v našem případě název autora.

3.3.6 Health Check

Health Check je služba zabývající se monitorováním stavu aplikací.

Služba bude komunikovat s ostatními službami a zjišťovat jejich stav, tento stav bude graficky zobrazen pomocí webového rozhraní.

Kapitola 4

Realizace

V této kapitole se zaměříme na konkrétní implementaci architektur, pro které jsme navrhli architekturu v předešlé kapitole.

Na úvod kapitoly se podíváme na celkovou strukturu ekosystému naší aplikace implementované pomocí mikroslužeb a na prvky společné pro všechny služby.

Začněme tím, že si povíme o nástrojích použitých při implementaci. Jako vývojové prostředí byl použit textový editor *Visual Studio Code* [32]. Jedná se o editor, který je zadarmo, fungující na bázi přidávání balíčků s funkcionalitami, které komunita uživatelů může vytvářet a sdílet.

Důvod, proč jsme si zvolili zrovna toto vývojové prostředí, je to, že nabízí mnoho rozšíření, které usnadňují vývoj aplikací. Krom rozšíření je zde i velká výhoda v tom, že *Visual Studio Code* dokáže spravovat několik terminálových oken najednou, které je možné si pojmenovat, lépe se mezi nimi orientovat a rychle mezi nimi přepínat. Tato vlastnost se obzvlášť hodí, když potřebujeme mít spuštěných více aplikací najednou a sledovat výstupy jednotlivých aplikací. Mezi použité rozšíření patří:

1. oficiální balíček **C#** [33] pro umožnění psaní .Net aplikací a jejich spouštění přímo z *Visual Studio Code*

2. **Docker** [34], toto rozšíření je obzvlášť užitečné, protože umožňuje spravování Docker containerů přímo z editoru

3. rozšíření **MongoDB for VS Code** [35] umožňuje sledování a interakci s MongoDB databázemi

4. aplikace si mezi sebou sdílí knihovny pomocí NuGet repositáře, k rychlému vyhledávání a aktualizaci balíčků z něj slouží **NuGet Package Manager** [36]

5. **Tabnine Autocomplete** [37] je asistent pro našeptávání kódu za pomoci umělé inteligence, dokáže detekovat opakující se vzory a nabízí je během psaní

Pro verzování textu této práce a zdrojových kódů praktické části se použil verzovací nástroj Git. Repositář, do kterého se práce verzovala, je hostován službou Github. Zde se nachází i podrobný popis projektu a návod na spuštění aplikace na lokálním zařízení, aby byla práce veřejně přístupná pro veřejnost.

Jak bylo již zmíněno v předchozích kapitolách, pro implementaci dokumentové databáze se zvolilo MongoDB a pro relační databázi MSSQL (Microsoft SQL Server). Pro zavedení komunikačního kanálu mezi službami se použila kombinace MassTransit a RabbitMQ. Pro vytvoření Gateway rozhraní se použila technologie Ocelot Gateway. Finálně celá práce běží v Docker containerech.

Aplikace napsané technologií .Net se musí nejdříve sestavit a následně spustit pomocí příkazu v terminálu, respektive pomocí vývojového prostředí. Následující příkazy pro sestavení a spuštění aplikace se musí vyvolat nad složkou, která obsahuje typ souboru s koncovkou *.csproj*[38].

```
dotnet build
dotnet run
```

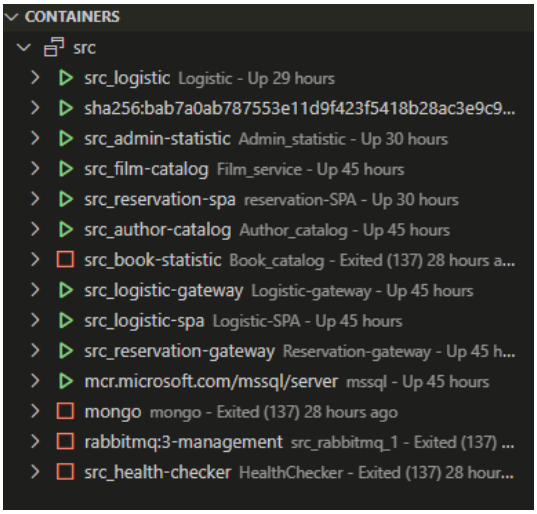
Obrázek 4.1: Sestavení a spuštění projektu

Poněvadž je ale systém nakonfigurován, aby fungoval v Docker containerech, je možné sestavit a spustit všechny potřebné části najednou místo jednotlivých částí pomocí příkazu 4.2 viz níže[28].

```
docker-compose build
docker-compose up
```

Obrázek 4.2: Příkaz pro sestavení a spuštění docker containerů

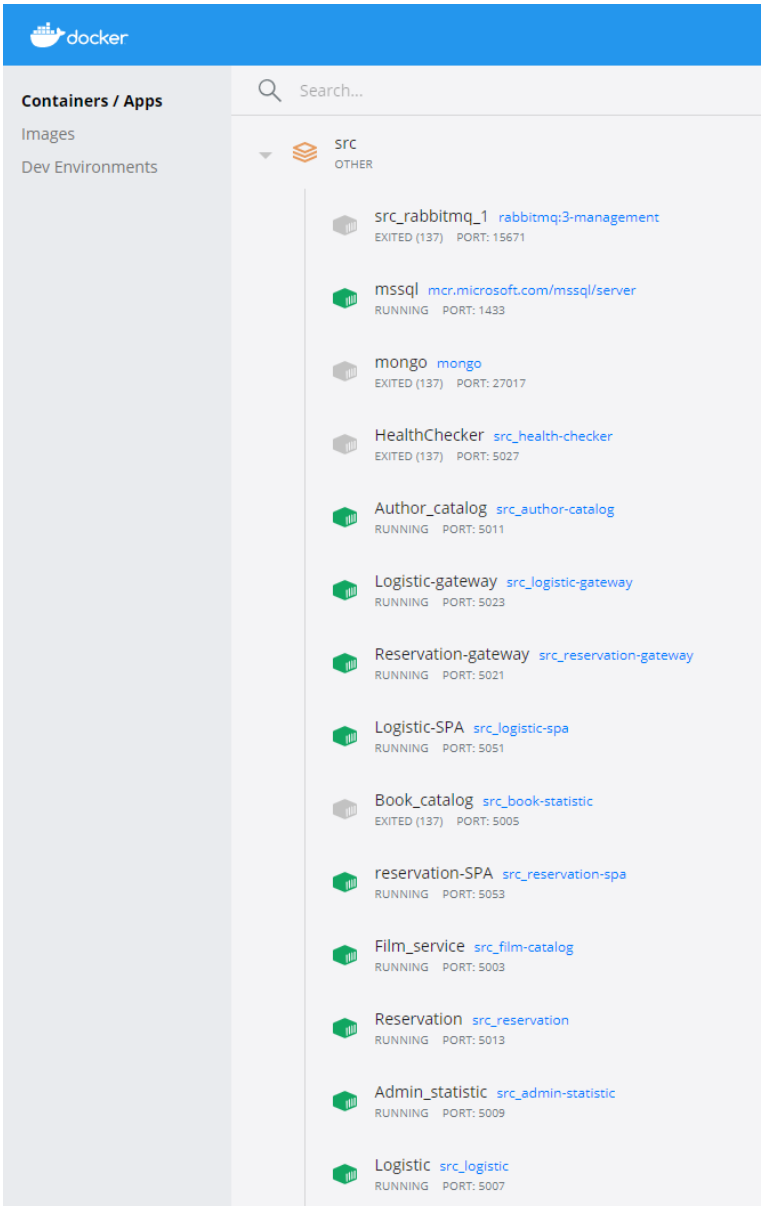
Po vyvolání příkazů výše si lze ve *Visual Studio Code* prohlédnout stav containerů, který výpadá následovně.



Obrázek 4.3: Zobrazení containerů ve Visual Studio Code

Containery se zelenou šipkou se úspěšně spustily, naopak ty s červeným čtvercem se z nějakého důvodu nespustily. Rozšíření umožňuje připojit konzoli ke containeru a nahlédnout do stavového logu, který může obsahovat informaci, proč nedošlo k úspěšnému spuštění.

Stejné informace dokáže získat i za pomocí samotné okenní Docker aplikace, která se nainstaluje spolu s *Docker Deamon*, o kterém se mluvilo v předchozí kapitole 3.2.4.2.



Obrázek 4.4: Zobrazení containerů v Docker Hub

Zobrazení z Docker aplikace vypadá podobně, aktivní containery jsou zelené, a ty, co se nespustily nebo nejsou aktuálně zapnuty, jsou šedé.

Co se týče konfiguračního souboru, který se stará o nastavení jednotlivých conteinerů, vystihneme zde pouze pár základních informací.

Protože každá aplikace včetně komunikačního kanálu a databází běží izolovaně ve vlastním conteineru, je zapotřebí nakonfigurovat síť, jejíž budou containery součástí, a přes kterou budou spolu komunikovat.

```
networks:
  localnetwork:
    driver: bridge
    driver_opts:
      com.docker.network.enable_ipv4: "true"
    name: localnetwork
    ipam:
      driver: default
      config:
          - subnet: 172.28.1.0/16
```

Obrázek 4.5: Nastavení lokální sítě pro Docker containery

Ukázka kódu zachycuje konfiguraci použitou pro naší aplikaci. Lokální síť jsme pojmenovali *localnetwork*, název použijeme pro referencování sítě při zařazování containerů do této sítě. Pokud container nemá specifikovanou síť, je zařazen do subnetu sítě hostitele, což znamená, že je přístupný z hostovacího zařízení, ale containery mezi sebou dostupné nejsou.

V další části ukázkového kódu se povoluje použití IPv4 protokolu a v sekci IPAM (zkratka IP address management) definujeme subnet, který bude síť používat. Pokud bychom nespecifikovali subnet, nemohli bychom staticky přiřadit adresy jednotlivým containerům. To by mělo za následek ten, že by se containery sice inicializovaly ve stejném subnetu, ale s dynamicky přidělenými adresami.

To by nebyl problém, pokud by náš systém měl zavedený například Service discovery systém, pomocí kterého bychom mohli zjistit adresy jednotlivých služeb. Poněvadž naše ukázka tento systém neimplementuje, jsou adresy nastaveny staticky, abychom s nimi mohli pracovat v jednotlivých službách.

Ještě se podívejme na ukázku nastavení jedné z našich služeb, například katalogu knih.

```
book-catalog:
  container_name: Book_catalog
  build: ./Services/ResSys.BookCatalog/src/ResSys.BookCatalog.Service/
  restart: always
  ports:
    - 5005:80
  depends_on:
    - mongo
    - rabbitmq
  networks:
    localnetwork:
      ipv4_address: 172.28.1.4
```

Obrázek 4.6: Docker nastavení služeb

První řádek opět specifikuje název containeru pro následné referencování, druhý řádek odkazuje na adresu, kde se nachází *Dockerfile* soubor starající se

o sestavení a spuštění dané služby. Sekce s porty je pole, do kterého lze zapsat množinu dvojic portů. První z dvojice označuje port, na kterém bude container naslouchat při dotazech z hostovacího zařízení. Druhý z portů představuje port používaný uvnitř Docker sítě.

Za sekcí portů následuje definice závislostí na ostatní containery. Do sekce se specifikuje množina containerů, které musí být aktivní a běžet, než se Docker pokusí tento container pustit. V našem případě říkáme, že katalog knih se má spustit, jakmile je spuštěn container s Mongo databází a komunikační kanál RabbitMQ.

Poslední sekce konfiguruje nastavení sítě containeru, zde si lze všimnout, že container má využít síť *localnetwork*, neboli síť, kterou jsme v předchozí ukázce 4.5 definovali. Navíc zde specifikuje fixní statickou adresu, kterou má v rámci subnetu použít. Docker tuto adresu nastaví containeru při spuštění.

Pomocí tohoto konfiguračního souboru lze konfigurovat mnohem více věcí, jako například přístupové údaje k databázi, specifikování cesty pro soubory a podobně.

4.1 Struktura projektu

Krom služeb, které se implementovaly, obsahuje struktura aplikace navíc jeden projekt. Tento projekt obsahuje konfigurační metody, rozhraní a pomocné metody, které využívají všechny nebo většina služeb. Tyto opakující se kódy byly vyjmuty a sjednoceny do jednoho projektu typu knihovna.

Z knihovny jsme po sestavení publikovali takzvaný NuGet balíček. Jedná se o balíček zdrojových kódů, který lze sdílet pomocí nástroje NuGet. NuGet je mechanizmem vymyšlený společností Microsoft, pomocí něj je možné veřejně verzovat balíčky, nastavovat závislosti na jiné balíčky a podobně. Repositář je veřejný, avšak je možné si vytvářet i soukromé repositáře, například i na lokálním stroji.

Pro naši ukázku jsme tedy mohli jít i cestou vytvoření vlastního lokálního repositáře, do kterého bychom sestavené balíčky vložili. Tato cesta má nevýhodu tu, že po sestavení Docker containerů je tento repositář nedostupný. Situaci je poté nutné řešit například tím, že v rámci sestavení containeru vytvoříme nový sdílený prostor, do kterého by pokaždé byly balíčky překopírovány. Tento proces ale zdržuje celkový proces sestavení containerů a z tohoto důvodu se zvolila varianta veřejného repositáře poskytnutý společností Microsoft, který je přístupný i ze samotných containerů. Repositář je dostupný na adrese *https://www.nuget.org/*.

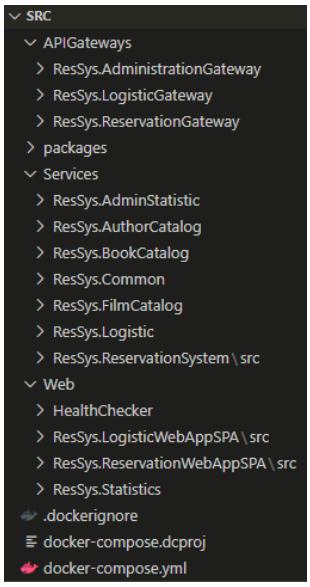
Projekty obsahují již dříve zmíněný soubor s koncovkou *.csproj*, do tohoto souboru se definují závislosti na jiné projekty a právě i NuGet balíčky.

```
<PropertyGroup>
        <TargetFramework>net5.0</TargetFramework>
</PropertyGroup>
<ItemGroup>
        <PackageReference Include="ResSys.Common" Version="1.1.2" />
        <PackageReference Include="ResSys.ReservationSystem.Contracts" Version="1.0.3" />
        <ProjectReference Include="\..\.....ResSys.ReservationSystem.Data.csproj" />
</ItemGroup>
```

Obrázek 4.7: Orientační ukázka csproj souboru

Struktura *csproj* souboru obsahuje definování nástroje pro sestavení souborů a zmíněné nastavení závislostí. V ukázce si lze všimnout referencování dvojího typu. První typ využívá NuGet repozitářů, speciálně zde vyžadujeme balíček pojmenovaný *ResSys.Common* a *ResSys.ReservationSystem.Contracts* včetně specifikované verze. Druhý typ odkazuje přímo na jiný projektu.

Celková struktura projektu poté odpovídá obrázku níže. V kořeni se vyskytuje *docker-compose* soubor starající se o sestavení a konfiguraci containerů a pak složky pro jednotlivé typy aplikací. Konkrétně máme tedy tři typy aplikací - ty, co slouží jako *Gateway*, ty, co tvoří služby, a poslední skupinou jsou webová uživatelská rozhraní. Poslední nezmíněná složka obsahuje sestavené NuGet balíčky na jednom místě. Jednotlivé projekty dále obsahují rozdělení na podprojekty, rozdělení se liší a jé závislé na typu architektury použité pro implementaci.



Obrázek 4.8: Struktura projektu

Tabulka níže obsahuje výčet síťového nastavení jednotlivých služeb. Při spuštění služeb pomocí *docker-compose* jsou služby dostupné na adrese

http://localhost:port, kde port odpovídá vnějšímu portu uvedenému v tabulce.

Tabulka 4.1: Přehled síťového nastavení služeb v Dockeru

	IP adresa	Vnější port	Vnitřtní port
RabbitMQ	172.28.1.1	15672	15672
Statistické přehledy	172.28.1.2	5009	80
Katalog autorů	172.28.1.3	5011	80
Katalog knih	172.28.1.4	5005	80
Katalog filmů	172.28.1.5	5003	80
Logistický sklad	172.28.1.6	5007	80
Rezervační systém	172.28.1.7	5013	80
Health Checker	172.28.1.8	5027	80
MSSQL DB	172.28.1.13	1433	1433
Mongo	172.28.1.10	27017	27017
Rezervační Gateway	172.28.1.11	5021	80
Logistická Gateway	172.28.1.12	5023	80
Statistická Gateway	172.28.1.15	5025	80
Logistická UI Web SPA	172.28.1.51	5051	80
Rezervační UI Web SPA	172.28.1.53	5053	80

Než se vrhneme na implementaci konkretních služeb, zbývá zmínit poslední technologii, kterou využívá většina služeb. Technologie se nazývá *Swagger* a slouží pro generování popisu webového rozhraní služby [39]. Součástí technologie je interaktivní webové rozhraní, které nejen zobrazuje informace o REST rozhraní dané služby, ale navíc umožňuje zasílání dotazů na zvolené rozhraní. Tato vlastnost se hodí při testování webového rozhraní služby.

Alternativním nástrojem, který se použil i během vývoje praktické části, je *Postman*. Nástroj slouží pro testování webových rozhraní a nabízí služby jako například vytváření kolekcí souvisejících adres a metod webových rozhraní s předdefinovanými hodnotami. Lze zde tedy mít pro každou službu vytvořenou záložku s uloženými metodami a zasílat testovací dotazy pomocí jednoho nástroje. Navíc tento nástroj umožňuje naimportovat popis rozhraní generované pomocí technologie *Swagger*.

4.2 Katalog autorů

Nyní již k samotné implementaci služeb. Začneme nejjednodušší službou a to katalogem autorů.

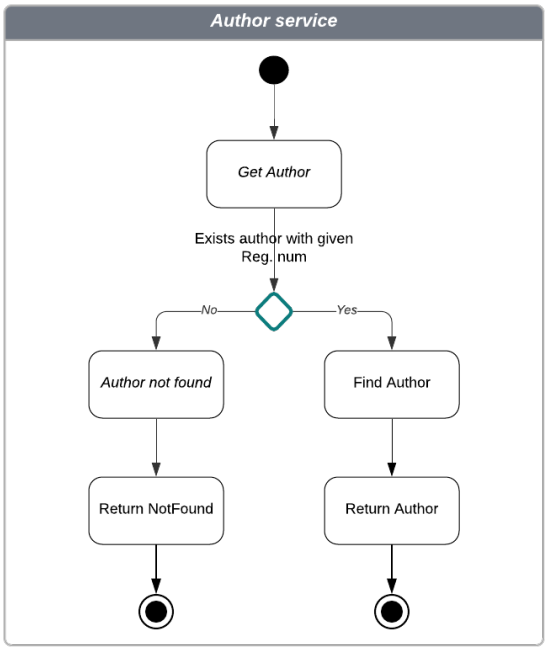
Služba má na starosti jedinou činnost a to je vyhledávání ve svém katalogu na základě přijatého registračního čísla autora anebo podle identifikátoru autora.

Přesto že v naší ukázce používáme pouze tyto dvě operace, webové rozhraní služby má implementované všechny základní CRUD operace. Popis rozhraní obsahuje následující tabulka.

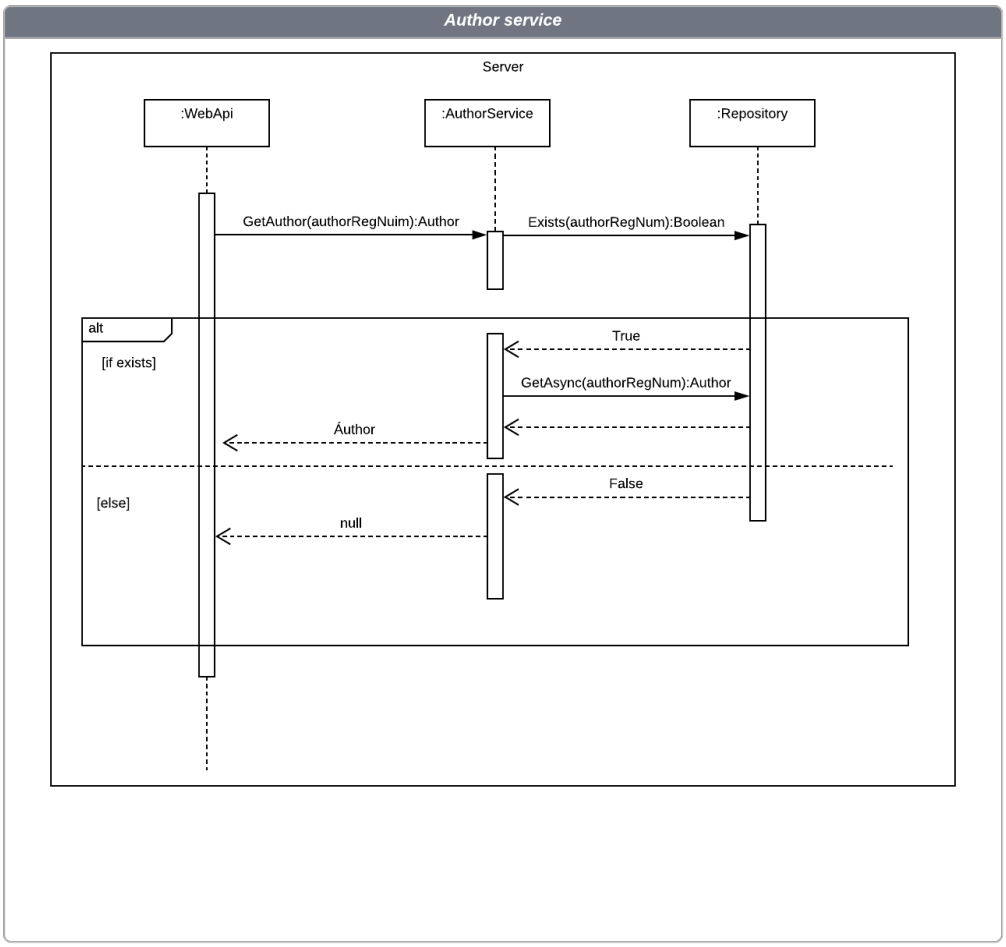
Tabulka 4.2: Metody webového rozhraní katalogu autorů

URI	Metoda	Popis
/authors	GET	Výpis všech autorů
/authors/{authorRegNum}	GET	Vrátí autora s registračním číslem
/authors	POST	Uložení nového autora
/authors/{id}	PUT	Editace autora s identifikátorem
/authors/{id}	DELETE	Vymazání autora s identifikátorem

Mezi nejsložitější operaci této služby patří vyhledání autora. Proces vyhledávání popisují následující dva diagramy.



Obrázek 4.9: Scénář vyhledání autora



Obrázek 4.10: Průběh zpracování požadavku na nalezení autora

Data jsou ukládaná do databáze MongoDB nacházející se v separátním Docker containeru, pro připojení k této databázi se využívá knihovna *Mongo.Driver*, které stačí poskytnout adresu, kde se MongoDB server nachází, a název databáze, se kterou má pracovat. Oba tyto údaje jsou konfiguračně nastavitelné v souboru *appsettings.json*. Tento soubor je načten prostředím při spuštění a lze do něj přistupovat pro čtení konfiguračních parametrů.

4.3 Katalog filmů a knih

Služby pro evidenci filmů a knih jsou totožné až na doménovou entitu, kterou spravují. Jejich architektura a implementační detaily jsou téměř stejné, z tohoto důvodu zde popíšeme implementaci pouze jedné.

Začneme opět popisem webového rozhraní, následující tabulka obsahuje výpis dostupných metod. Tabulka je vztažena ke katalogu filmů, katalog knih má metody obdobné, jen routa adresy obsahuje „books“ místo „films“.

Tabulka 4.3: Metody webového rozhraní katalogu filmů a knih

URI	Metoda	Popis
/films	GET	Výpis všech filmů
/films/{id}	GET	Vrátí film, kterému patří daný identifikátor
/films/{id}	PUT	Editace filmu s daným identifikátorem

Z výpisu dostupných metod si lze všimnout, že službě chybí metoda typu *POST* a *DELETE*. To je z důvodu, že chceme zaručit zakládání a mazání filmů, resp knih, pouze na základě zaslaných zpráv přes komunikační kanál.

4.3.1 Nastavení komunikačního kanálu

Nyní se pojďme podívat na implementaci *MassTransit* a *RabbitMQ*, pomocí kterých se služba připojuje ke komunikačního kanálu. Prvním krokem je stáhnout knihovnu *MassTransit* z *NuGet* repositáře. Tato knihovna obsahuje konfigurační metodu *AddMassTransit* zajišťující definici a nastavení pro danou službu. Metodu je potřeba zavolat při inicializaci projektu a to konkrétně v metodě *ConfigurateServices* v souboru *Startup.cs*. V základu obsahuje soubor *Startup.cs* dvě metody, první je *ConfigureServices*, to je metoda starající se o nastavení služeb, konfiguraci vkládání referencí a závislostí (*Dependency injection*). Z tohoto důvodu vkládáme konfiguraci *MassTransit* právě sem. Druhou metodou je *Configure* a ta obsahuje konfiguraci například pro middleware a routování (endpointů).

Celá konfigurace ve zmíněném souboru je znázorněna na následující ukázce kódu.

```
services.AddMassTransit(x =>
{
    x.AddConsumers(type.Assembly);
    x.UsingRabbitMq((context, configurator) =>
    {
        var configuration = context.GetService<IConfiguration>();
        var serviceSettings = configuration
                .GetSection(nameof(ServiceSettings)).Get<ServiceSettings>();
        var rabbitMQSettings = configuration
                .GetSection(nameof(RabbitMQSettings)).Get<RabbitMQSettings>();
        configurator.Host(rabbitMQSettings.Host);
        configurator.ConfigureEndpoints(
            context,
            new KebabCaseEndpointNameFormatter(serviceSettings.ServiceName, false)
        );
    });
});

services.AddMassTransitHostedService();
```

Obrázek 4.11: Konfigurace MassTransit

První krok, který při nastavení děláme, je specifikování takzvaných konzumentů zpráv. Ke konzumentům se vrátíme později. Programu říkáme, aby registroval všechny konzumenty, které se nachází v projektu, který se spouští. I v případě, že se náš projekt skládá z více subprojektů, tento příkaz by fungoval, protože říká, aby prošel všechny zdrojové kódy, které jsou v rámci tohoto projektu. Každá služba si musí tuto konfiguraci dělat sama.

Dalším krokem je specifikování, jaký druh komunikačního kanálu budeme používat a jeho nastavení. V našem případě používáme *RabbitMQ* a proto zvolíme metodu *UsingRabbitMq*. *MassTransit* umožňuje i jiné poskytovatele, v takovém případě bychom zvolili jinou konfigurační metodu. V metodě *UsingRabbitMq* nastavíme připojení ke komunikačnímu kanálu, pro toto nastavení využíváme data z konfiguračního souboru *appsettings.json*, do kterého jsme definovali adresu a port služby. Jakmile získáme dané konfigurační nastavení, pomocí metody *Host* nastavíme cestu. Dále už jen nastavíme název endpointu aby odpovídal názvy dané služby a službu zaregistrujeme.

Z důvodu, že tento kód se opakuje u téměř každé služby, je vyjmut a dán do separátní knihovny *ResSys.Common*. Knihovny služby importují a mohou z ní využívat funkce jako představené nastavení technologie *MassTransit*.

Vraťme se nyní ke konzumentům zpráv a jejich definici. Vytvoření a registrace konzumenta zpráv je jednoduchá. Registraci jsme si již ukázali, zbývá si tedy ukázat jejich definice. Knihovna *MassTransit* obsahuje definici generického rozhraní *IConsumer*, jemuž se specifikuje jaký typ zprávy má přijímat. Pokud chceme vytvořit konzumenta, stačí pokud třída implementuje zmíněné rozhraní a metodu *Consume*. Metoda *Consume* obsahuje jeden argument a to

je obsah přijaté zprávy. Argument naplňuje mechanizmus technologie *MassTransit* a my se tedy nemusíme starat o přijímání a zpracování zprávy jako takové, ale až finálních dat.

```
public class SupplyFilmConsumer : IConsumer<SupplyFilm>
...
...
public async Task Consume(ConsumeContext<SupplyFilm> context)
{
var msg = context.Message;
...
...
```

Obrázek 4.12: MassTransit konzument

Při registraci konzumentů metoda *AddConsumers*, která byla v předchozí ukázce 4.11, prochází všechny sestavené třídy a hledá ty, které implementují právě toto rozhraní.

MassTransit umožňuje krom přijímání zpráv i jejich odesílání. To je taky velmi jednoduché. V rámci této knihovny je k dispozici rozhraní *IPublishEndpoint* s definovanou metodou *Publish*. Pro odeslání zprávy stačí zavolat tuto metodu a jako parametr jí dát zprávu, kterou chceme odeslat.

```
private readonly IPublishEndpoint publishEndpoint;
...
...
await publishEndpoint.Publish(new FilmUpdated(...));
```

Obrázek 4.13: Odeslání zprávy do komunikačního kanálu

Každá služba odesílající zprávy má ještě vedlejší knihovnu s názvem ve formátu *ResSys.XXX.Contracts*. V těchto knihovnách se nachází přehled zpráv, které služba odesílá. Pokud služba chce přijímat zprávy od nějaké jiné, naimportuje si danou knihovnu a pro jednotlivé zprávy vystaví konzumenty. Tímto je i zaručena kompatibilita formátu a datové struktury zpráv. Navíc služba nemůže odesílat modifikovanou zprávu bez toho, aby se to projevilo v dané knihovně. Takže cílové služby si pouze aktualizují verzi knihovny, pokud nastane změna.

Výčet zpráv, se kterými služba pracuje je následující, opět vztažena ke katalogu filmů, katalog knih má výčet obdobný:

1. příchozí
    a) SupplyFilm - ResSys.Logistic.Contracts - informace o naskladnění zboží
2. odchozí

a) FilmCreated - ResSys.FilmCatalog.Contracts - informace o vytvoření nového filmu

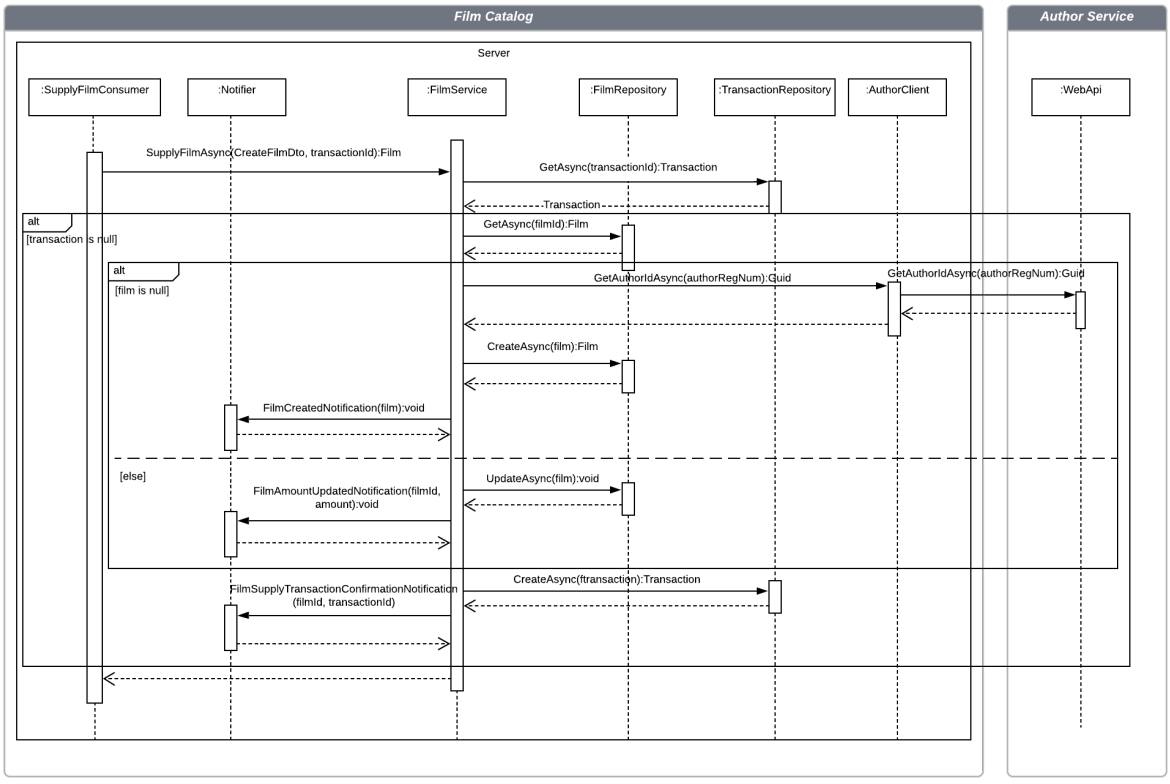
b) FilmUpdated - ResSys.FilmCatalog.Contracts - informace o aktualizaci informací filmu

c) FilmAmountUpdated - ResSys.FilmCatalog.Contracts - informace o naskladnění nových kusů existujícího filmu

d) FilmSupplyTransactionConfirmation - ResSys.FilmCatalog.Contracts - informace o přijetí a zpracování požadavku na naskladnění nového zboží

4.3.2 Architektura

Hlavní logika těchto služeb tkví v naskladnění nového zboží. Tento proces vyžaduje koordinaci všech částí, ze kterých se služby skládají. Tento proces je znázorněn na následujícím diagramu.



Obrázek 4.14: Proces naskladnění nových zásob

Celý proces začíná tím, že konzument odchytí zprávu o naskladnění zboží. Pro ošetření stavu, kdy jednu zprávu odchytí více instancí služby je zde za-

vedena kontrola, že transakce se stejným identifikátorem ještě nebyla zpracována.

Dalším krokem je zjistit, zda-li jde o navýšení zásob anebo o naskladnění úplně nového zboží. K tomu použijeme repositář filmů, resp. knih, abychom zjistili informaci o existenci filmu s daným EIDR (unikátní identifikátor filmů), resp. v případě knih jde o IBAN. Pokud není žádná taková entita nalezena, považujeme zprávu jako žádost o naskladnění nového zboží a v takovém případě musíme zjistit identifikátor autora. K tomu použijeme třídu *AuthorClient* starající se o komunikaci s touto službou. Na rozdíl od komunikace pomocí komunikačního kanálu se jedná o komunikaci synchronní, to znamená, že voláme danou službu napřímo a čekáme na její odpověď.

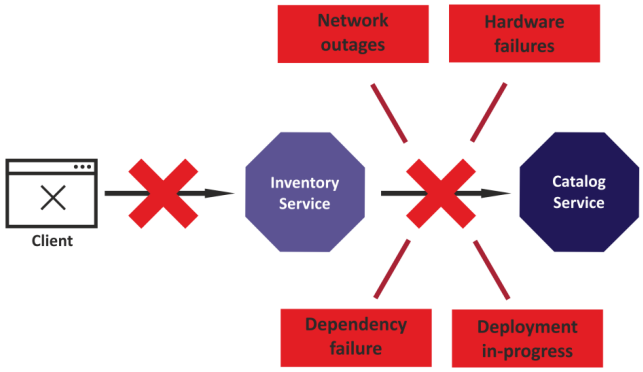
Abychom mohli komunikovat takto napřímo, musíme mít definovanovanou třídu typu *HttpClient*. Tuto konfiguraci provádíme v souboru *Startup.cs* v metodě *AddClientSynchronize*. V rámci konfigurace nastavujeme i opakování dotazu při selhání a *Circuit breaker*.

```
.AddTransientHttpErrorPolicy
(
  builder => builder.Or<TimeoutRejectedException>().WaitAndRetryAsync
  (
    5,
    retryAttemp =>
       TimeSpan.FromSeconds(Math.Pow(2, retryAttemp))
       + TimeSpan.FromMilliseconds(jitterer.Next(0, 10000)),
    onRetry: (outcome, timespan, retryAttempt) =>
    {
        var serviceProvider = services.BuildServiceProvider();
        serviceProvider.GetService<ILogger<AuthorClient>>()?
        .LogWarning(
        \$"Dealying for {timespan.TotalSeconds}s, next attempt in {retryAttempt}");
    }
))
```

Obrázek 4.15: Opakování dotazu při selhání

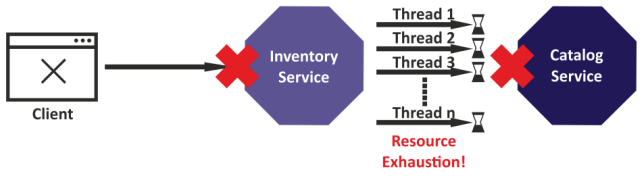
Konkrétně jsme nastavili exponenciální opakování při selhání, neboli při prvním selhání se čeká 2 vteřiny, po jejich uplynutí se dotaz zkusí znovu a pokud opět selže, čekáme 4 vteřiny a tak dále. Informaci o opakování dotazu můžeme evidovat (logovat) využitím metody *onRetry*.

Selhání dotazu může proběhnout z několika důvodů a je vhodné dotaz zkusit zopakovat než celý proces zahodit a proces ukončit například chybovou hláškou nebo informováním o neúspěšném navázání kontaktu s danou službou.



Obrázek 4.16: Vznik chyby během synchronní komunikace

Circuit breaker návrhový vzor slouží k redukci zátěže služby při dotazování externích služeb. Nejdříve si povězme, co by se mohlo stát, kdybychom to implementované neměli. Opakování dotazů nám přidává šanci, že po nějaké době bude služba dostupná a dotaz se k ní nakonec dostane. Každopádně toto opakování se vztahuje ke každému dotazu, který se pošle, není tedy rozdíl pokud zde čeká několik dotazů anebo několik stovek. Pokud bychom tuto situaci neřešili, mohlo by se nám stát, že po nějaké době, kdyby služba byla delší dobu nedostupná, by se vyčerpaly všechny naše zdroje a to by mohlo vést i k samotnému pádu/zkolabování služby, která čeká na výsledek.



Obrázek 4.17: Vyčerpání zdrojů při mnohonásobném dotazování

Vraťme se tedy k implementaci *Circuit breaker* návrhového vzoru. Jeho implementací docílíme toho, že pokud se detekuje selhávání dotazů, dojde k zablokování odesílání dotazů na tuto službu globálně. *Circuit breaker* čeká určitý časový interval po jehož uplynutí povolí průchod jednoho dotazu. Pokud tento dotaz opět selže, obvod se opět zablokuje. Pokud dotaz proběhne v pořádku, obvod se sepne a je možné se službou opět normálně komunikovat.

V našem systému využíváme existující implementace v .Net frameworku a ta vypadá následovně.

```
.AddTransientHttpErrorPolicy(
  builder => builder.Or<TimeoutRejectedException>().CircuitBreakerAsync(
    3,
    TimeSpan.FromSeconds(15),
    onBreak: (outcome, timespan) =>
    {
        var serviceProvider = services.BuildServiceProvider();
        serviceProvider.GetService<ILogger<AuthorClient>>()?
        .LogWarning(\$"Opening the circuit for {timespan.TotalSeconds}s...");
    },
    onReset: () =>
    {
        var serviceProvider = services.BuildServiceProvider();
        serviceProvider.GetService<ILogger<AuthorClient>>()?
        .LogWarning(\$"Closing the circuit...");
    }
))
```

Obrázek 4.18: Implementace Circuit breaker

Definujeme zde, že po 3 neúspěšných dotazech dochází k zablokování (otevření) obvodu. Další parametr udává jak dlouho má být obvod zablokovaný, v našem případě se jedná o 15tivteřinový interval. Po uplynutí tohoto intervalu se obvod opět sepne a pokusí se provést jeden testovací dotaz.

4.4 Rezervační systém

Tato služba se skládá z klientské webové části implementované pomocí MVVM architektury 2.3.2 a serverové části navržené pomocí Onion architektury 2.2.2.

Stejně jako předchozí služba obsahuje napojení na komunikační kanál, konfigurace probíhá takřka totožně a proto jí zde nebudeme opakovat.

Jediný větší rozdíl je zde rozdělení zodpovědnosti a komunikace mezi jednotlivými části serverové části.

4.4.1 Server

Struktura serverové části odpovídá návrhu z předešlé kapitoly, ale než se vrhneme na implementační detaily, opět si zde na úvod uvedeme přehled metod webového rozhraní.

Tabulka 4.4: Metody webového rozhraní rezervačního systému

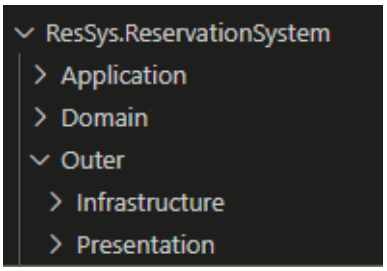
URI	Metoda	Popis
/reservation	GET	Výpis všech rezervací
/reservation/{id}	GET	Vrátí rezervaci s identifikátorem
/reservation/book/{id}/{date}	GET	Počet aktivních rezervací knihy
/reservation/film/{id}/{date}	GET	Počet aktivních rezervací filmu
/reservation/	POST	Uložení nové rezervace
/reservation/{id}	DELETE	Ukončení rezervace rezervace

Z metod je patrné, že systém přijímá požadavky na vytváření nových rezervací pomocí webového rozhraní. Jedná se o nejsložitější scénář služby a jeho průběh si ukážeme později na diagramu.

Krom komunikace pomocí webového rozhraní je služba napojena i na komunikační kanál. Na rozdíl od katalogových služeb žádné zprávy nepřijímá, nemá tedy implementované žádné konzumenty. Do komunikačního kanálu odesílá dvě zprávy.

1. odchozí
    a) ReservationCreated - ResSys.ReservationSystem.Contracts - informace o vytvoření nové rezervace
    b) ReservationDeleted - ResSys.ReservationSystem.Contracts - informace o ukončení rezervace

Po seznámení se se zapojením služby do celkové komunikace našeho ekosystému mikroslužeb, si pojďme přiblížit strukturu projektu a rozdělení jednotlivých částí.



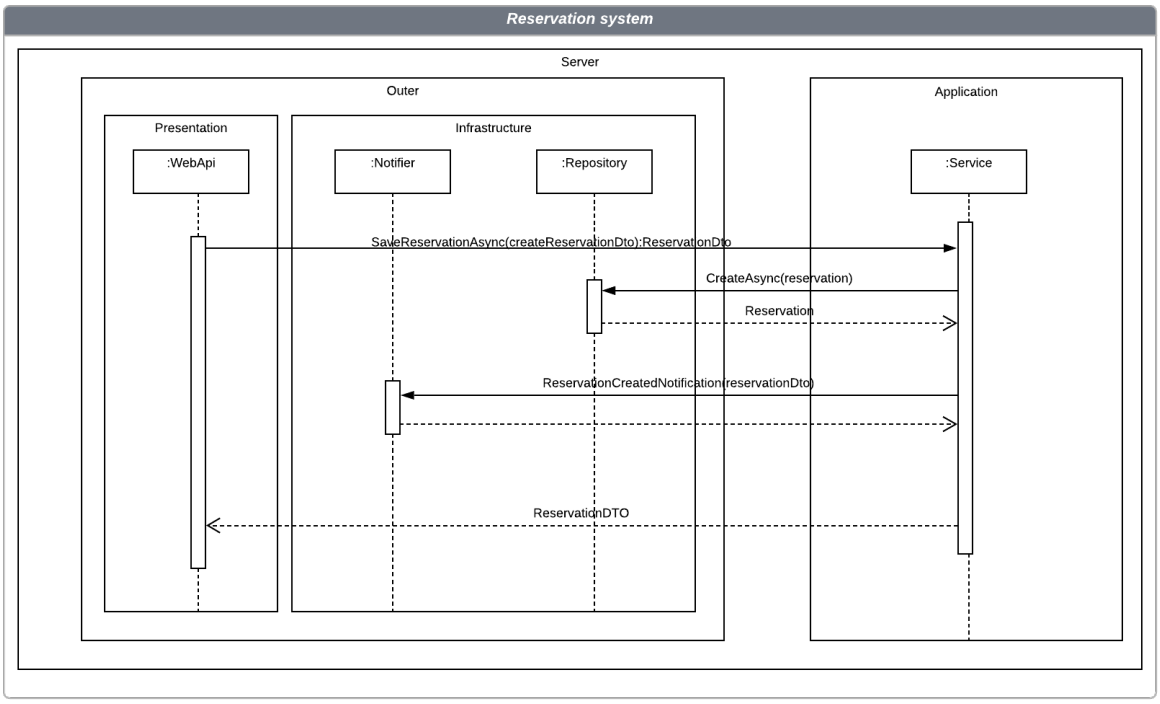
Obrázek 4.19: Struktura projektu rezervačního systému

Z návrhu jsme rozdělili systém do tří větších celků a to do doménové, aplikační a „vnější“ (outer).

Doménová část obsahuje definované entity tvořící doménu rezervačního systému. Struktura a propojení entit odpovídá vytvořenému doménovému návrhu. Aplikační část obsahuje složku s definovanými rozhraními,
které potřebuje pro vykonávání své logiky. V našem případě obsahuje definici

svého rozhraní, rozhraní notifikačního klienta a rozhraní definující získávání dat perzistentně uložených do databáze pomocí návrhového vzoru *Repository*. Kromě složky s definicemi rozhraní také definuje třídu vykonávající aplikační logiku a takzvaný „Mapper“ starající se o převod doménových entit na modely používaných při komunikaci mezi vrstvami aplikace. Poslední celek obsahuje definici webového rozhraní a položky infrastruktury. V části infrastruktury se poté nachází definice databázového připojení a notifikační klient, starající se o zasílání zpráv při vzniku nové rezervace.

Komunikaci těchto celků si můžeme prohlédnout na následujícím diagramu reprezentující scénář zakládání rezervací.



Obrázek 4.20: Proces vytváření nové rezervace

Jednotliví účastníci tohoto scénáře odpovídají podprojektům v rezervačním systému. Je vidět, že i přes to, že *WebApi*, *Notifier* a *Repository* patří do stejné kategorie Outer, jsou ještě rozděleny do podkategorií *Presentation* a *Infrastructure*, která se navíc rozpadají na další dva podprojekty.

Aplikační projekt a databázový projekt zabývající se persistencí dat (*Repository*) jsou jedinými projekty s referencí na doménovou část, ta se v této ukázce nevyskytuje. Zbylé projekty včetně databázového mají ještě referenci na aplikační projekt. Je tedy vidět, že i přes to, že aplikace při vykonávání logiky komunikuje s prvky z infrastruktury, tak na ně nemá žádnou referenci.

Toho jsme dosáhli pomocí abstrakce docílené definováním rozhraní.

Tyto rozhraní pro zopakování má definované aplikační projekt. Projekty v infrastruktuře toho rozhraní implementují a zaregistrují se, že dané rozhraní implementují.

Představme si v rychlosti jak takový proces funguje na příkladu s notifikační službou. Aplikační vrstva definuje rozhraní *INotifier*, ve kterém specifikuje nutnost na implementaci dvou metod. Tyto dvě metody odpovídají zprávám, které služba rozesílá komunikačním kanálem.

```
public interface INotifier
{
        Task ReservationCreatedNotification(ReservationDto reservationDTO);
        Task ReservationDeletedNotification(Guid id);
}
```

Obrázek 4.21: Definice rozhraní pro notifikační službu

V části projektu zabývající se infrastrukturou vytvoříme projekt *ResSys.ReservationSystem.Notify*, který se bude starat o implementaci konkrétního notifikačního nástroje, v našem případě *MassTransit*.

Krom metod pro konfiguraci tohoto nástroje obsahuje i samotnou třídu, starající se o zasílání zpráv pomocí *MassTransit*. Tato třída implementuje rozhraní INotifier, definované v aplikačním projektu.

```
public class Notifier : INotifier
```

Obrázek 4.22: Implementace rozhraní INotifier

Aby se tato implementace využívala, je potřeba, aby byla zaregistrována do nástroje pro vkládání referencí (*Dependency injection*).

```
public static IServiceCollection AddReservationNotifier(this IServiceCollection services)
{
        services.AddScoped<INotifier, Notifier>();
        return services;
}
```

Obrázek 4.23: Registrace implementace

A aby naše třída v aplikační části projektu tuto implementaci získala, musí si požádat pomocí konstruktoru o implementaci specifikovaného rozhraní, jenž tam vloží .Net runtime při vytváření instance dané třídy.

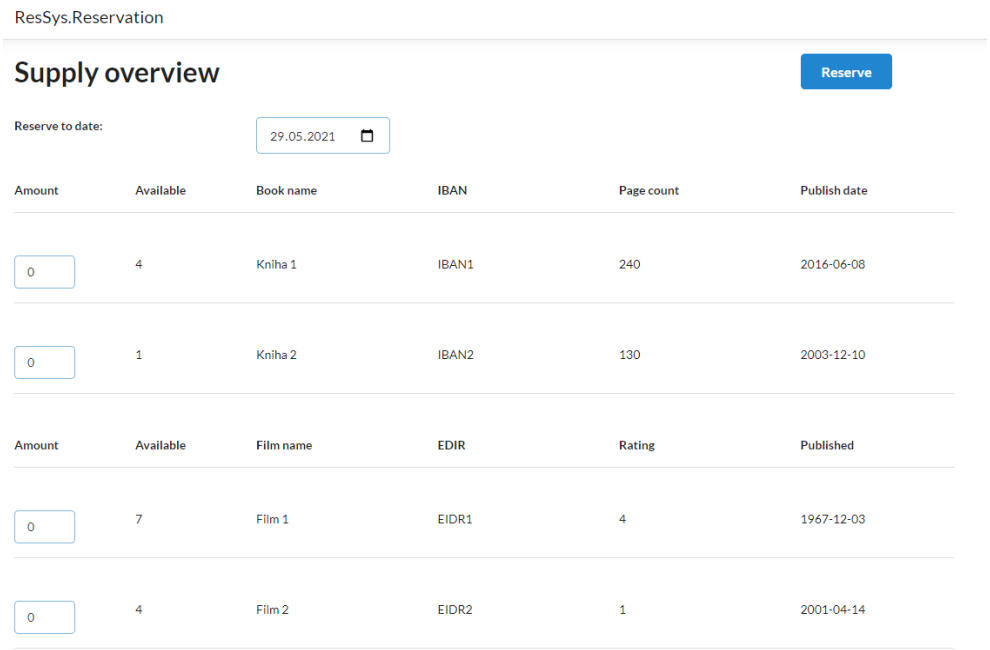
```
public ReservationService(
    IReservationRepository reservationRepository,
    INotifier msgNotifier)
{
        this.reservationRepository = reservationRepository;
        this.msgNotifier = msgNotifier;

        if (this.reservationRepository == null)
        throw new ArgumentNullException("Missing reference for IReservationRepository");
        if (this.msgNotifier == null)
        throw new ArgumentNullException("Missing reference for INotifier");
}
```

Obrázek 4.24: Požádání o implementaci pomocí konstruktoru

V tomto případě krom implementace rozhraní *INotifier* žádáme i o implementaci rozhraní *IReservationRepository* obstarávající komunikaci s databází. Dále si lze všimnout, že kontrolujeme, zdali jsme požadované implementace dostali. V případě, že ne, je zde velká pravděpodobnost, že by funkce třídy nefungovaly správně a je potřeba na tento fakt upozornit. Tento problém může nastat například v případě, že jsme zapomněli registrovat implementační třídu.

4.4.2 Klient

Klientská část aplikace měla podle návrhu splňovat architekturu MVVM 2.3.2.



Obrázek 4.25: Formulář pro rezervování zboží

V našem případě jsme pro implementaci tohoto vzoru použili knihovnu React, která se stará o vykreslování uživatelského rozhraní. Ve zkratce MVVM stojí za první z písmen V, neboli View.

Tím, že jsme implementovali i komunikaci se serverovou stranou rezervačního systému v našem uživatelském rozhraní, získali jsme navíc zkratku VM, neboli ViewModel, a rozšířili celkovou zkratku na VVM. Poslední znak ze zkratky získáme tím, že přidáme správu aktuálního stavu aplikace.

Rezervační systém obsahuje formulář, pomocí kterého uživatelé mohou rezervovat knihy a filmy. Tím, že uživatel může zvolit více knih a filmů, musí aplikace evidovat, které možnosti již zvolil a případně jaké množství si vybral. Navíc kromě volby zboží je zde i nutnost zadat datum do kdy si chce zboží zapůjčit. Abychom udržovali tyto informace o stavu formuláře pohromadě, definovali jsme v aplikaci model nazývající se *State*. *State* se stará o stav komponenty, v našem případě formuláře, a při jeho změně dochází k přegenerování vzhledu formuláře.

Pro komunikaci se serverem se používá knihovna *Axios*, jako adresu jsme specifikovali adresu rezervační Gateway, která zastřešuje komunikaci s rezervačním systémem, katalogem filmů a katalogem knih. Stejně jako na serverové části i zde jsme aplikovali návrhový vzor *Repository* a obalili tím komunikaci pomocí *Axios* knihovny.

4.5 Logistický sklad

Logistický sklad funguje pro naskladňování nového zboží. Pro tyto účely bylo vytvořené uživatelské rozhraní s formulářem pro vkládání dat. Uživatel tedy pomocí uživatelského rozhraní vyplní obsah nové dodávky zboží a po stisknutí tlačítka odešle požadavek na server, kde ho zpracuje služba.

Naskladnění zboží a synchronizace jsou jediné funkce, které rezervační systém nabízí, tomu odpovídá i seznam metod webového rozhraní serveru.

Tabulka 4.5: Metody webového rozhraní logistického skladu

URI	Metoda	Popis
/synchronize	GET	Vyvolá akci synchronizace s katalogy
/supply	GET	Vrátí přehled naskladnění
/supply	POST	Uložení nové dodávky zboží

Metoda pro naskladnění nového zboží funguje intuitivně. Záznam je uložen do databáze a jednotlivým filmům a knihám vygeneruje transakční identifikátor, ten použije při komunikaci se službami a zpětně namapování při potvrzení uložení daného produktu. Katalog filmů a knih je informován o nové dodávce pomocí komunikačního kanálu a zpráv typu: *SupplyFilm* a *SupplyBook*. Služba odchytává zprávy typu *BookSupplyTransactionConfirmation* sig-

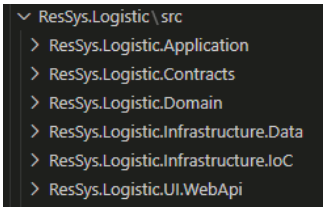
nalizující potvrzení uložení knihy a obdobně pro film zprávu typu *FilmSupplyTransactionConfirmation*.

1. příchozí
    a) FilmSupplyTransactionConfirmation - ResSys.FilmCatalog.Contracts - potvrzení o vytvoření
    b) BookSupplyTransactionConfirmation - ResSys.BookCatalog.Contracts - potvrzení o vytvoření
2. odchozí
    a) SupplyFilm - ResSys.Logistic.Contracts - zpráva o naskladnění nového filmu
    b) SupplyFBook - ResSys.Logistic.Contracts - zpráva o naskladnění nové knihy

Druhá metoda webového rozhraní spouští mechanizmus pro synchronizaci zboží, které se nepovedlo dříve uložit. Služba vyhledá všechno zboží, u kterých nemá potvrzení o vytvoření a pošle požadavek o jejich založení.

4.5.1 Server

Struktura serverové části je podobná rezervačnímu systému.



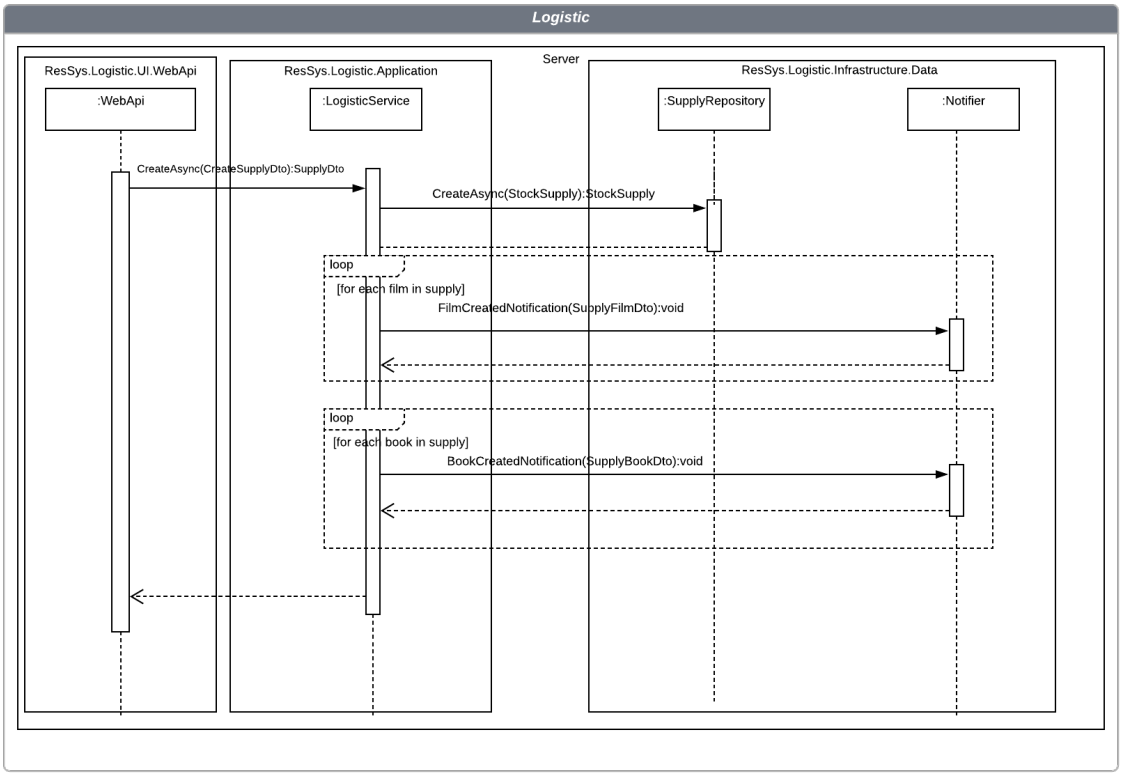
Obrázek 4.26: Struktura projektu logistické služby

Napojení na komunikační kanál je opět totožné jako u předchozích služeb, nebudeme se jím tedy zabývat.

Jediným větším rozdílem oproti předchozí službě je ten, že rezervační služba má kromě notifikačního klienta i konzumenta zpráv a také to, že zajištění vkládání referencí bylo extrahováno do části infrastruktury. Konzument zpráv je klasifikován obdobně jako webové rozhraní anebo prvek infrastruktury. Z tohoto důvodu byla jeho implementace zařazena do infrastruktury, konkrétně části zabývající se daty.

Z obrázku 4.26 lze vidět podobnost s Onion architekturou, kterou implementovala předchozí služba. Jednotlivé složky z obrázku odpovídají samostatným projektům typu knihovna, které jsou mezi sebou provázané.

Provázanost je velmi podobná stejně jako struktura v porovnání s Onion architekturou. Jediný rozdíl tvoří projekt *ResSys.Logistic.Infrastructure.IoC*, který musí mít referenci na všechny části projektu, kromě doménové. Je to z důvodu, že zajišťuje konfiguraci a registraci služeb do nástroje pro vkládání referencí. To může udělat pouze za předpokladu, že má přístup ke všem rozhraním a jejich implementacím. Extrakce tohoto chování do separátní části projektu má výhodu, že ostatní části se nemusí zajímat o to, jakým způsobem je vkládání řešeno, aby mohly zaregistrovat svá rozhraní. Nevýhoda je naopak taková, že při vzniku nového rozhraní musíme upravovat jinou část projektu.



Obrázek 4.27: Proces naskladnění nových zásob

Diagram 4.27 vyobrazuje proces ukládání nové zásilky. V rámci tohoto procesu se validují jednotlivé položky knih a filmů, po zvalidování se objekt uloží do objektové databáze MongoDB. Entity produktů mají v této databázi kombinaci dvou identifikátorů, předgenerované Id a poté *FilmId*, resp. *BookId*.

Pokud záznam nemá doplněný druhý identifikátor, předpokládá se, že nedošlo k jeho uložení, protože tento identifikátor se doplňuje z potvrzovací zprávy odpovídajícího katalogu. První identifikátor představuje identifikátor transakce, který se posílá spolu s daty k uložení. Jakmile odpovídající služba

uloží přijatá data, odesílá potvrzení obsahující právě dvojici prvního a druhého identifikátoru. Tento proces jsme si ukázali dříve na diagramu 4.14.

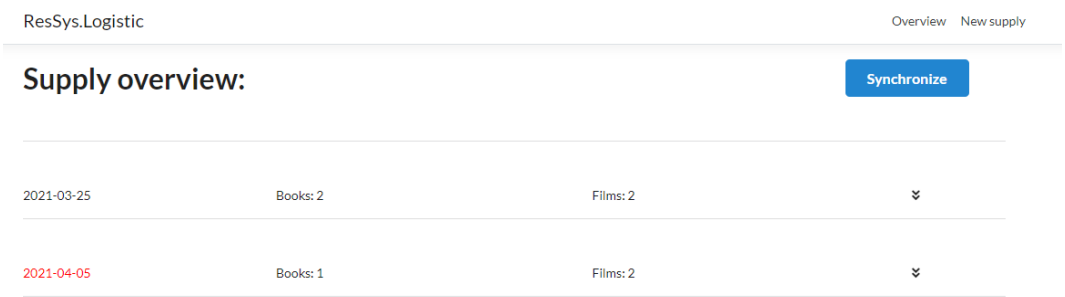
Metoda vyvolávající synchronizaci dat funguje také na základě nedoplněných identifikátorů. Mechanizmus projde uložená data a vyhledává ty, které mají druhý identifikátor prázdný. Pro tyto položky odešle zprávu o naskladnění, jako kdyby právě přišel požadavek z webového rozhraní. Katalogy si hlídají jaké transakce již zpracovaly, nemůže tedy dojít ke zduplikování dat.

Jakmile server obdrží potvrzení o naskladnění, vyhledá zboží s *Id* odpovídající *TransactionId* obdržené v potvrzovací zprávě a této položce nastaví druhý identifikátor. Tímto je proces naskladnění ukončen.

4.5.2 Klient

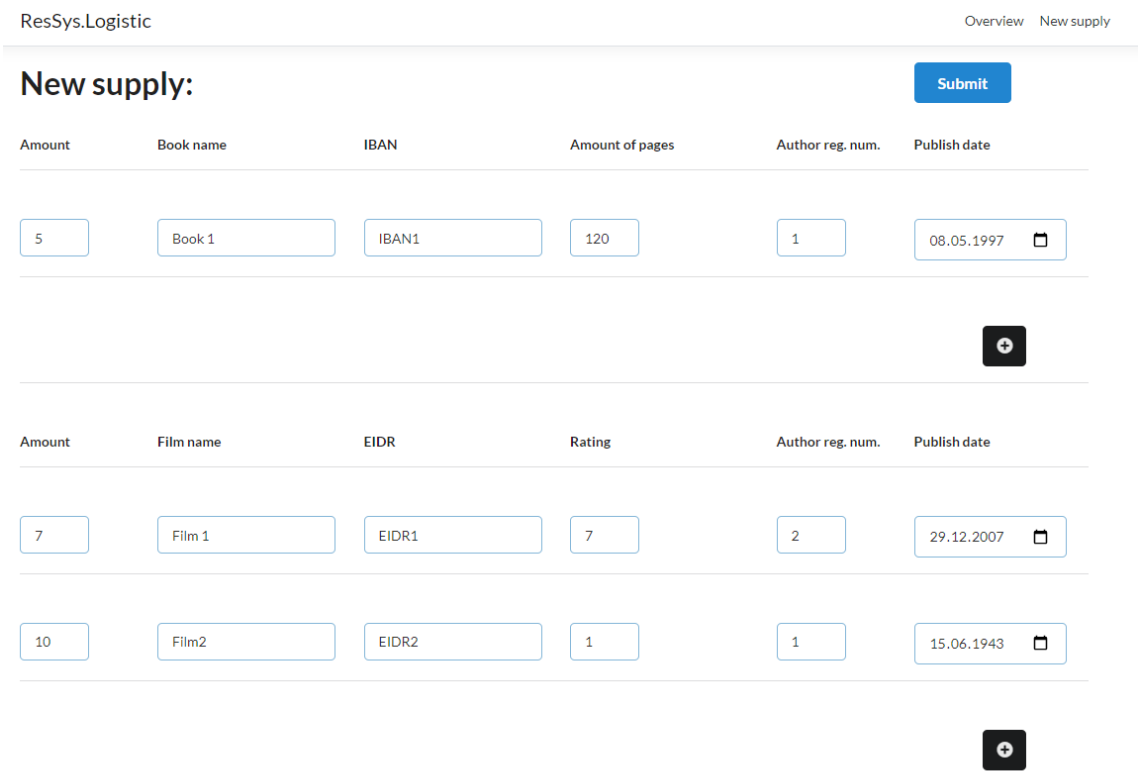
Klientská aplikace využívá stejného principu jako služba rezervační. Vyvinula se také na základě MVVM architektonického vzoru za pomocí technologie React. I zde se pro komunikaci se server stranou používá knihovna *Axios*. V případě logistické webové aplikace se využívá jiná gateway pro směrování dotazů než v předešlém případě. Zde se jedná o rezervační gateway, která přesměrovává všechny dotazy pouze na serverovou stranu logistické služby.

Na úvodní stránce si může uživatel zobrazit přehled dříve uložených zásob včetně informace, jestli se uložily všechny položky. Pokud některá z položek nebyla úspěšně uložena, je datum zobrazeno červenou barvou. Pokud uživatel bude chtít zahájit synchronizaci pro aktualizaci těchto položek, nachází zde i tlačítko, které po stisknutí zavolá serverovou metodu pro zahájení synchronizace.



Obrázek 4.28: Přehled vytvořených požadavků na naskladnění zboží

Další záložka obsahuje formulář, kam uživatel může vyplnit neomezený počet knih a filmů, a po potvrzení formuláře odešle požadavek o jeho uložení.



Obrázek 4.29: Formulář na vytvoření požadavku naskladnění zboží

Webová aplikace neobsahuje rozsáhlé validace a ošetření krajních případů užití, protože slouží záměrně za účelem ukázky implementace.

4.6 Statistická služba

Poslední nezmíněná služba se taktéž skládá ze serverové a klientské aplikace. Pro implementaci serverové části se zvolila Hexagonal architektura 2.2.3.

Se serverovou částí komunikuje uživatelská webová aplikace, navržená na základě MVC architektury, přes rezervační Gateway. Tato brána zaštiťuje dotazy na službu autorů, rezervací a právě statistickou službu.

V rámci práce jsme se zaměřili primárně na implementaci architektury a z těchto důvodů služba nenabízí žádné pokročilé statistiky.

Pro vyhodnocování výstupů potřebuje mít služba přístup k datům z jednotlivých služeb, agregace těchto dat pomocí komunikace s jednotlivými daty by mohla být pomalá a navíc by se mohla vracet v nevhodném formátu. Z tohoto důvodu se v statistické službě implementovala relační databáze, do které si služba ukládá vzniklé knihy, filmy i rezervace. Aby mohla dosáhnout

této funkcionality, musí být připojena na komunikační kanál a odposlouchávat zprávy týkající se vzniku a úpravy produktů nebo rezervací.

Relační databáze se zde zvolila za účelem demonstrace, že jednotlivé služby mohou používat jiné databáze na základě toho, co je pro jejich funkcionalitu nejvýhodnější. Relační databáze nám dovolují na rozdíl od dokumentových databází vytvářet relace mezi daty a rychleji vyhledávat provázaná data.

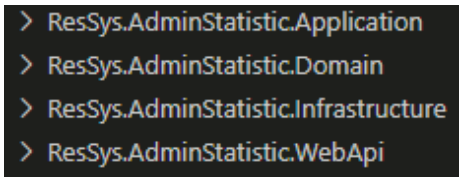
Pro komunikaci se serverovou službou je vystaveno jednoduché webové rozhraní, obsahující pouze 2 metody.

Tabulka 4.6: Metody webového rozhraní statistické služby

URI	Metoda	Popis
/statistic/monthDistribution	GET	Počty vytvořených rezervací v měsíci
/statistic/reservations	GET	Přehled aktivních rezervací

4.6.1 Server

Hexagonální architektura si zakládá na co nejmenší provázanosti systému. Střed celého systému je doménový model, který obaluje aplikační vrstva. Zbytek systému je tvořen takzvanými moduly, které se připojují k systému pro doplnění a rozšíření funkcionality.



Obrázek 4.30: Struktura projektu

Aplikační vrstva obsahuje definici rozhraní stejně jako u ostatních architektur. Je zde ale velký rozdíl v registraci závislostí. Oproti předchozím službám se zde využívá *AutoFac* [40] knihovna, která se snaží co nejvíce snížit provázanost mezi rozhraním a jeho implementací.

V předchozích případech jsme museli přímo v kódu definovat, jaké rozhraní je implementované kým. Pokud vzniklo nové rozhraní, museli jsme tuto vazbu přidat a pokud došlo k vymazání nebo úpravě rozhraní, museli jsme změnu reflektovat i v nastavení referencí.

Pro nastavení závislostí pomocí *AutoFac* nemusíme nic takového dělat. V kódu se definuje třída, která dědí od *Autofac.Module* nacházející se v *Autofac* knihovně. V naši definované třídě poté nastavíme, jaké jmenné prostory chceme sledovat.

```
public class ApplicationModule : Autofac.Module
{
        protected override void Load(ContainerBuilder builder)
        {
                        builder.RegisterAssemblyTypes
                        (typeof(ApplicationException).Assembly)
                        .AsImplementedInterfaces()
                        .InstancePerLifetimeScope();
                }
}
```

Obrázek 4.31: Registrace modulu AutoFac

V ukázce 4.31 definujeme, že chceme zaregistrovat všechny nalezené implementace rozhraní ve jmenném prostoru odpovídající aplikační vrstvě. Vyhledávání podle jmenného prostoru vypadá tak, že pokud máme například jmenný prostor *ResSys.Statistics*, zaregistruje se nám vše, co obsahuje tento prefix, jako například *ResSys.AdminStatistic.Infrastructure.Repository* nebo *ResSys.AdminStatistic.Infrastructure.MassTransit*. Protože by některé jmenné prostory mohly obsahovat více tříd, než bychom chtěli procházet a registrovat, tak tento nástroj umožňuje přidat doplňující podmínky, které musí třída splnit, aby byla zaregistrována.

Krom zmíněného zaregistrování referencí slouží tento nástroj i pro nakonfigurování nástrojů, které do systému přidáváme. Tato vlastnost je pro hexagonální architekturu klíčová, protože každý přidaný modul do systému si s sebou nese i postup jak se nakonfigurovat. V takovém případě pak stačí daný modul pouze zaregistrovat do konfiguračního souboru využívající *AutoFac* a systém si tento modul dokáže sám najít a spustit proces nastavení.

```
{
    "defaultAssembly": "ResSys.AdminStatistic.Infrastructure",
    "modules": [
        {
            "type": "ResSys.AdminStatistic.Infrastructure.Modules.ApplicationModule",
            "properties": {}
        },
        {
          "type": "ResSys.AdminStatistic.Infrastructure.EntityFrameworkDataAccess.Module",
           "properties": {
               "ConnectionString":
                  "Server=172.28.1.13,1433;Database=AdminStat;User=SA;Password=Your_password123"
           }
        },
        {
           "type": "ResSys.AdminStatistic.Infrastructure.ServiceBus.MassTransit.Module",
           "properties": {
                "Host": "amqp://guest:guest@172.28.1.1:5672",
                "ServiceName": "AdminStat_Service"
           }
        }
    ]
}
```

Obrázek 4.32: Konfigurace AutoFac

Konfigurace 4.32 obsahuje nastavování všech modulů, ze kterých se statistická služba skládá. Vidíme, že je zde konfigurace *MassTransit*, *MSSQL* databáze a aplikačního jádra, který se také považuje za modul.

Připojení k databázi je provedeno pomocí ORM *Entity Frameworku*. Pro připojení ke komunikačnímu kanálu je opět implementováno pomocí *MassTransit RabbitMQ*.

4.6.2 Klient

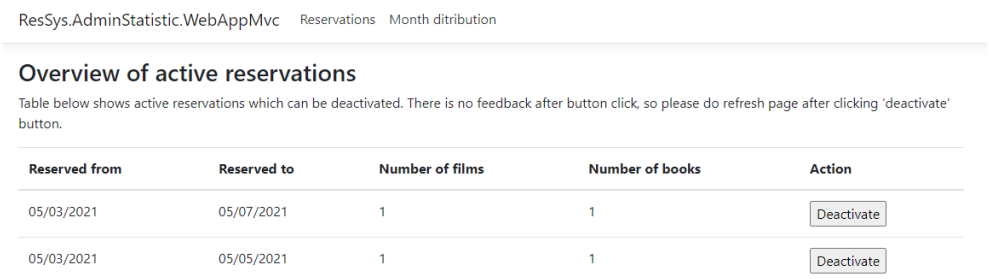
Při vývoji klientské aplikace pro statistické výstupy jsme se snažili držet prezentačního návrhového vzoru MVC.

Pro implementaci jsme zvolili technologii Razor stránek, které jsou primárně generované na serverové straně. Kompletní architektura je tvořena třemi prvky, View reprezentuje vzhled stránky, do kterého lze pomocí Razor dynamicky generovat obsah. Obsah lze generovat například na základě dat, které do View pošleme. Data se do View přenášejí pomocí Modelu.

Scénář je tedy takový, že Controller odchytí dotaz na server a zavolá aplikační logiku, jejíž chování jsme si představili v předchozí sekci. Jakmile nám aplikační vrstva vrátí výsledek operace, převedeme výsledek na Model, neboli tvar, který očekává klientská vrstva. Jakmile máme vytvořený Model, zavoláme metodu *View* a do parametru vložíme vytvořený Model. Metoda

View na základě kritérii vyhledá námi definované View a to využije námi poskytnutého Modelu pro vygenerování svého obsahu.

Vygenerovaný obsah se poté vrací uživateli, kterému se výsledek zobrazí v prohlížeči.



Obrázek 4.33: Ukázka statistické webové aplikace

4.7 Infrastruktura

4.7.1 Ocelot Gateway

Všechny brány jsou implementovány stejně. Ukážeme si tedy způsob implementace na jedné z nich.

V našem projektu jsme *Ocelot Gateway* bránu implementovali tak, že jsme pro ní vytvořili úplně nový projekt, pro každou z bran jeden, a do těchto projektů se zaintegrovala knihovna *Ocelot* verze 17.0.0.

Konfigurace brány samotné je poté jednoduchý proces. Musíme zaregistrovat, že se má *Ocelot Gateway* použít pro řešení směrování příchozích požadavků. Registrace se vykonává ve dvou krocích, prvním krokem je přidat *Ocelot Gateway* do kolekce služeb metodou *AddOcelot*, aby byla během běhu aplikace dostupná druhým krokem je přidání do množiny používaného middlewaru metodou *UseOcelot*. Obě metody jsou součástí zmiňované knihovny, která se do projektu zaintegrovala.

Konfigurace směrování dotazů se provádí v konfiguračním souboru, typicky pojmenovaném *ocelot.json*. Nesmíme zapomenout přidat tento soubor do obsahu, který si prostředí při zapínání aplikace načte do paměti.

```
{
"Routes": [
      {
        "UpstreamPathTemplate": "/api/books",
        "UpstreamHttpMethod": [
        "GET","POST"
        ],
        "DownstreamPathTemplate": "/books",
        "DownstreamScheme": "http",
        "DownstreamHostAndPorts": [
        {
                        "Host": "172.28.1.4",
                        "Port": 80
                }
        ]
      }
    ]
}
```

Obrázek 4.34: Ocelot - Ukázka nastavení směrovacích pravidel

Na příkladu 4.34 vidíme ukázkové nastavení směrování. *UpstreamPathTemplate* v kombinací *UpstreamHttpMethod* specifikují pro jakou příchozí adresu a použité metody má provést přesměrování.

Pokud brána obdrží dotaz odpovídající adrese a metodě uvedené v konfiguraci, přesměruje tento dotaz na adresu uvedenou v *DownstreamPathTemplate*. Pro přesměrování je potřeba specifikovat i adresu cílové služby a port, na kterém služba naslouchá.

4.7.2 HealthCheck

Sledování stavu jednotlivých služeb je zapotřebí, obzvlášť když se počet služeb rozroste nad počet, který se dá rozumně kontrolovat manuálně.

V rámci této práce jsme implementovali systém, který poskytuje sama firma *Microsoft*, a nazývá se *HealthCkecksUI*. Konkrétně se implementovala verze knihovny 5.0.1.

Služba v našem systému mikroslužeb se nazývá *HealthChecker* a jediná její funkce je registrace funkcí z knihovny do kolekce služeb a middlewaru. A to je vše, co je potřeba, aby systém na dotazování začal fungovat, služba začne být aktivní a začne posílat dotazy. Aby věděla, na jaké adresy má služba dotazy zasílat, musíme do souboru *appsettings.json* vypsat jednotlivé adresy a názvy služeb.

```
{
    "HealthChecksUI": {
        "HealthChecks": [
        {
                "Name": "FilmCatalog",
                "Uri": "http://172.28.1.5/health"
            },
        {
                "Name": "BookCatalog",
                "Uri": "http://172.28.1.4/health"
        }
      ]
    }
}
```

Obrázek 4.35: HealthCheck - výpis adres pro kontrolu zdraví

Z ukázky 4.35 si lze všimnout, že každá služba musí mít implementované rozhraní pro */health*, které vrací údaje o stavu služby. Nemusí se nutně jednat o tento název, název definují vývojáři a může být i jiný, každopádně pojmenování z ukázky se bere jako konvence. *Microsoft* se již postaral i o implementaci rozhraní a v rámci knihovny *HealthChecksUI* poskytuje metodu *MapHealthChecks*, pomocí které se nastavuje adresa rozhraní i typ výstupu při dotazu.

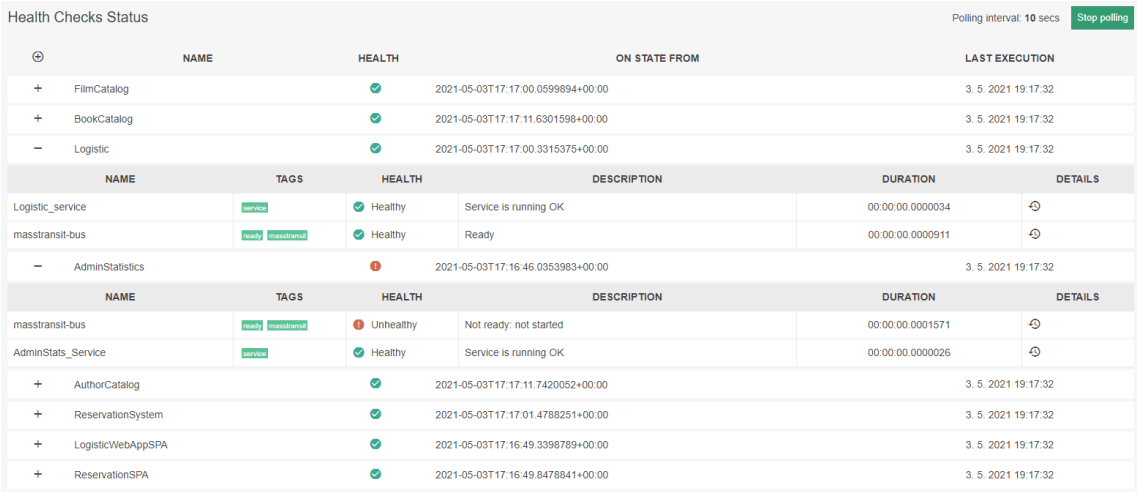
```
{
  endpoints.MapHealthChecks("/quickHealth", new HealthCheckOptions()
        {
                Predicate = _ => false
        });
        endpoints.MapHealthChecks("/health/services", new HealthCheckOptions()
        {
                Predicate = reg => reg.Tags.Contains("service"),
                ResponseWriter = HealthChecks.UI.Client
                    .UIResponseWriter.WriteHealthCheckUIResponse
        });
        endpoints.MapHealthChecks("/health", new HealthCheckOptions()
        {
                ResponseWriter = HealthChecks.UI.Client
                    .UIResponseWriter.WriteHealthCheckUIResponse
        });
}
```

Obrázek 4.36: HealthCheck - konfigurace rozhraní a odpovědí

V naší aplikaci jsme zvolili pro ukázku tři různé adresy a tím i výstupy 4.36. První způsob vrací pouze jednoduchou informaci, zda-li je vše v pořádku anebo pokud existuje nějaký problém. Druhý způsob vrací informace pouze o službách, jejichž název obsahuje slovo *service*. Tímto filtrem by neprošlo

například *MongoDB* nebo *MassTransit*. Poslední způsob nemá specifikované žádné filtry a při dotazu na tuto adresu dostaneme detailní informace o stavu služby a všech jejích částí.

V praxi se často kombinuje více adres v rámci jednoho procesu ověřování stavu služby. Například kombinace prvního a třetího způsobu nám dá rychlou informaci o tom, zdali je služba plně funkční a při případné potřebě zjištění konkrétních detailů se využije adresa pro detailní výpis.



Obrázek 4.37: HealthCheckUI

Dokonce i samotné grafické rozhraní 4.37 je součástí řešení od společnosti *Microsoft*.

Kapitola 5

Testování

Poslední částí implementačního procesu je pokrýt aplikaci automatizovanými testy. Povíme si o způsobu testování jednotlivých částí aplikace, jaké technologie jsme zvolili pro vytvoření testů a ukážeme si nějaké konkrétní ukázky.

Způsob psaní testů se liší na základě zvoleného nástroje pro testování. Pro testování *.Net* aplikací existuje řada testovacích nástrojů, ze kterých si můžeme vybrat. Mezi nejznámější a nejpoužívanější nástroje patří tyto: *xUnit*, NUnit a *MSTest*. Pro otestování našich aplikací jsme zvolili nástroj *xUnit*.

Testovat aplikaci lze po několika stránkách. Můžeme testovat chování jednotlivých metod v systému, neboli *Unit tests*, nebo můžeme spojit testování více metod dohromady například pro otestování vyhodnocení nějakého scénáře, tento způsob se nazývá *Integration tests*. Dalšími typy jsou například: testování uživatelského rozhraní, *End-to-end tests*, testování přístupnosti nebo *Load tests* testující chování systému při zátěži.

Pro otestování naší ukázky použijeme první dva zmíněné typy a to jsou *Unit tests* a *Integration tests*. *Unit Tests* použijeme pro otestování jednotlivých metod, jejich chování a zda-li na základě vstupu vracejí správná data. *Integration tests* použijeme pro vyhodnocování dotazů na webové rozhraní.

Technologie *xUnit* nám umožňuje označit část kódu a ten následně spouštět v testovacím prostředí. Také poskytuje nástroje pro kontrolu vyhodnocení metod pomocí *Assert*. *Assert* funguje na bázi toho, že si nastavíme, jaký očekáváme výsledek a poskytneme mu reálný výsledek. Pokud se tyto hodnoty liší, tetovací prostředí ukončí proces testování a označí výsledek jako chybný.

Část kódu můžeme označit jako testovací dvěma způsoby. Knihovna *xUnit* využívá dvou klíčových označení *Fact* a *Theory*. Rozdíl mezi nimi je ten, že *Fact* se používá pro jednorázové testování daného kódu. Oproti tomu *Theory* umožňuje otestování stejného kódu na základě množiny kombinací vstupních dat. Pokud použijeme klíčové slovo *Theory* je kód vyhodnocený jako chybný za předpokladu, že aspoň jedna z uvedených kombinací vstupních dat selhala.

5.1 Testy služeb

5.1.1 Jednotkové testy

Při psaní jednotkových (*Unit*) testů se zaměřujeme primárně na otestování konkrétní jedné funkcionality (metody) a ne celkového řetězce akcí, který metoda spouští. Pokud metoda využívá jiné objekty a jejich funkce, jsou většinou tyto funkce takzvaně „mockované“. Toto označení popisuje proces, kdy specifikujeme chování, které se má při zavolání specifické metody vykonat. Například v ukázce 5.3 specifikujeme chování objektu *readOnlyRepository* při zavolání metody *GetAllAsync*. Metoda, která by za normálních okolností provedla dotaz na databázi a případné mapování objektů, se v tomto případě vůbec nezavolá, místo toho testovací prostředí zajistí podvržení reálného zavolání výsledkem, který jsme specifikovali.

Abychom mohli tento „mockovací“ proces použít, musíme využít dostupných knihoven zaměřených na tento typ funkcionality. V rámci testování jsme použili dva nástroje *Moq* a *NSubstitute* pro účely demonstrace jejich použití.

Substituční nástroj *NSubstitute* se snaží cílit na co největší uživatelskou přívětivost, aby programátoři ztráceli co nejméně času psaním testů a měli více prostoru na vývoj[41]. V rámci testování jsme použili verzi 4.2.0 staženou pomocí *NuGet* nástroje.

Pro nahrazení třídy a její následnou simulaci stačí jeden řádek kódu 5.1. Vlastně zde specifikujeme pouze to, že chceme vytvořit náhradníka pro specifikovaný typ objektu, v tomto případě pro *IBookReadOnlyRepository*.

```
Substitute.For<IBookReadOnlyRepository>()
```

Obrázek 5.1: NSubstitute nahrazení

Moq vytváří substituty velmi podobně, pro samotné vytvoření stačí také pouze jeden řádek 5.2. Následně specifikuje chování pro potřebné metody a to je vše.

```
var mock = new Mock<IBookReadOnlyRepository>();
...
mock.GetByIdAsync(1).Returns(book);
```

Obrázek 5.2: Moq nahrazení

Jakmile máme specifikované a vytvořené substituty za potřebné třídy, stačí specifikovat, jaké očekáváme chování při zavolání specifické metody nahrazeného objektu. Můžeme zde nastavit jakékoliv chování volaného objektu a to od bezproblémového vyhodnocení až po vyvolání výjimky (*Exception*). Ukázka 5.3 obsahuje příklad, jak lze specifikovat chování objektu *readOnlyRepository* při zavolání metody *GetAllAsync*.

```
[Fact]
public async void Get_all_books()
{
        var getUseCase = new GetBooksUseCase(
        readOnlyRepository
        );

        Book book = new Book()
        {
                Id = Guid.Parse("A3A12B10-21B3-46C5-92CE-C888CF9856D0"),
                IBAN = "IBAN1",
                Name = "Kniha 1",
                Description = "description 1",
                NumberOfPages = 10,
                AuthorId = Guid.Parse("A3A12B10-21B3-46C5-92CE-C888CF9856D0"),
                AuthorRegNum = 1,
                Amount = 5,
                PublishDate = DateTime.Parse("2021-04-24")
        };

        readOnlyRepository
            .GetAllAsync()
            .ReturnsForAnyArgs(new List<Book>() { book });

        BooksResult result = await getUseCase.Execute();

        Assert.Equal(result.Count(), 1);
        var bookRes = result.ElementAt(0);
        Assert.Equal(bookRes.Id, book.Id);
        Assert.Equal(bookRes.IBAN, book.IBAN);
        Assert.Equal(bookRes.Name, book.Name);
        Assert.Equal(bookRes.Description, book.Description);
        Assert.Equal(bookRes.NumberOfPages, book.NumberOfPages);
        Assert.Equal(bookRes.AuthorId, book.AuthorId);
        Assert.Equal(bookRes.AuthorRegNum, book.AuthorRegNum);
        Assert.Equal(bookRes.Amount, book.Amount);
        Assert.Equal(bookRes.PublishDate, book.PublishDate);
}
```

Obrázek 5.3: Unit test ukázka typu Fact

Ukázka 5.3 testování správného vyhodnocení scénáře pro získání seznamu knih. V tomto případě jde o jednorázové otestování a využilo se testu typu *Fact*. Na začátku metody inicializujeme třídu scénáře v statistické službě implementující Hexagonal architekturu za pomocí Domain Driven Designu.

```
[Fact]
[Theory]
[InlineData("C4A12R71-21B3-46C5-92CE-C888CF9856D0", "IBAN1", "Kniha 1",
        "description 1", 10, "A3A12B10-21B3-46C5-92CE-C888CF9856D0", 1, 5, "2021-04-24")]
[InlineData("A3A12B10-21B3-46C5-92CE-C888CF975364", "IBAN2", "Kniha 2",
        "description 1", 10, "A3A12B10-21B3-46C5-92CE-C888CF985999", 1, 5, "1998-01-15")]
public async void Save_books(string bookId ... string publishDate)
{
        var datetime = DateTimeOffset.Parse(publishDate);
        var bookIdGuid = Guid.Parse(bookId);
        var authorIdGuid = Guid.Parse(authorId);

        var useCase = new SaveBookUseCase(
        writeOnlyRepository,
        readOnlyRepository
        );

        BookResult book = await useCase.Execute(bookIdGuid ... datetime);

        Assert.Equal(book.Id, bookIdGuid);
        Assert.Equal(book.IBAN, IBAN);
        Assert.Equal(book.Name, name);
        Assert.Equal(book.Description, description);
        Assert.Equal(book.NumberOfPages, numberOfPages);
        Assert.Equal(book.AuthorId, authorIdGuid);
        Assert.Equal(book.AuthorRegNum, authorRegNum);
        Assert.Equal(book.Amount, amount);
        Assert.Equal(book.PublishDate, datetime);
}
```

Obrázek 5.4: Unit test ukázka typu Theory

Testování pomocí *Theory* využívá *InlineData* anotace 5.7. Test se spustí pro každý řádek dat, který je pomocí anotace k metodě přidán, pokud se pro některou kombinaci dat vyhodnotí ukládání špatně, je test celkově vyhodnocen záporně.

5.1.2 Integrační testy

Aplikaci jsme tedy pokryly jednotkovými testy a následně jsme si pověděli o způsobu testování komunikačního kanálu. Poslední testování, o kterém si povíme, jsou integrační testy. V rámci integračního testování jsme testovali chování celkové funkcionality služby a spolupráce jednotlivých částí.

Integrační test začíná v bodě odeslání dotazu na server a následným otestováním výsledku. V těchto testech se nepoužívá „mockování“ jednotlivých částí a odpovídá to tedy reálnému chování služby jako celku.

Dotazy se neodesílají na reálně běžící služby, místo toho vytváříme simulující kopii serveru. K tomu slouží *TestServer*, nástroj vytvořený společností

Microsoft vytvořený pro simulaci běhu serveru v paměti.

Pro simulaci a testování webového serveru napsaného v *.Net* stačí vytvořit nový testovací server a konfigurovat mu patřičné údaje 5.5, jako jsou například soubory, které si má při spouštění načíst. Instance objektu vytvořená inicializací *TestService* třídy obsahuje metodu pro vytvoření klienta, kterého můžeme při testování použít pro odesílání dotazů 5.6. Můžeme zkoušet zavolání několika adres za sebou anebo klidně pouze jednu a následně testovat návratové hodnoty odpovědí.

```
var webHostBuilder = new WebHostBuilder()
    .UseStartup<Startup>()
    .ConfigureAppConfiguration(
      (builderContext, config) =>
        {
            IHostingEnvironment env = builderContext.HostingEnvironment;
            config.AddJsonFile("autofac.json")
            .AddEnvironmentVariables();
        })
        .ConfigureServices(services => services.AddAutofac()
    );
server = new TestServer(webHostBuilder);
client = server.CreateClient();
```

Obrázek 5.5: Vytvoření testovacího serveru

```
var response = await client.PostAsync("api/Synchronize", content);

response.EnsureSuccessStatusCode();

var responseString = await response.Content.ReadAsStringAsync();
JObject result = JsonConvert.DeserializeObject<JObject>(responseString);
Assert.Equals(result["numberOfItems"].Value<int>(), 5);
```

Obrázek 5.6: Integrační test synchronizace zboží

5.2 Testy infrastruktury

Všechny naše služby připojené na komunikační kanál využívají kombinaci technologii *MassTransit* a *RabbitMQ*. V kapitole zabývající se výhodami použití *MassTransit* a důvody, proč ho používat, jsme zmiňovali abstrakci při práci s komunikačním kanálem jako hlavní výhodu. Při testování můžeme této vlastnosti využít taktéž.

Tím, že v aplikaci komunikujeme pouze s *MassTransit* nakonfigurovaným, aby na pozadí používal *RabbitMQ*, má výhodu v tom, že vždy budeme testovat pouze správné chování frameworku *MassTransit*, protože jsme naprosto odstíněni od komunikace s konkrétním *Message Brokerem*.

Pro testování frameworku lze využít nástrojů vytvořených samotnými jeho tvůrci. Testovací nástroje jsou již součástí stejné knihovny a není potřeba stahovat žádné další doplňkové knihovny. Nástroje obsahují funkcionalitu pro vytvoření komunikačního kanálu běžícího uvnitř výpočetní paměti a tím simulovat funkci reálné komunikace a posílání zpráv.

```
var harness = new InMemoryTestHarness();
var repository = new Mock<ISaveBookUseCase>();
var consumerHarness = harness.Consumer<BookCreatedConsumer>(() =>
    new BookCreatedConsumer(repository.Object));

await harness.Start();
try
{
        BookCreated book = new BookCreated(Guid.NewGuid(), "IBAN1",
        Guid.NewGuid(), "Kniha1", "Popis", 2, 15, 1, DateTime.Now, Guid.NewGuid());

        await harness.InputQueueSendEndpoint.Send<BookCreated>(book);

        Assert.True(await harness.Consumed.Any<BookCreated>());
        Assert.True(await consumerHarness.Consumed.Any<BookCreated>());
}
finally
{
        await harness.Stop();
}
```

Obrázek 5.7: Unit test ukázka typu Theory

Proces začíná tím, že si vytvoříme metodu pro testování pomocí *xUnit* stejně jako v předešlé kapitole. V metodě jsme si vytvořili pomocí *InMemoryTestHarness* objekt reprezentující virtuální komunikační kanál. Ke kanálu musíme připojit jednotlivé konzumenty, které chceme testovat.

Pomocí vystaveného endpointu *InputQueueSendEndpoint* zasíláme jednotlivé zprávy a následně testujeme, zda-li se zpráva zpracovala, kdo jí zpracoval a jaký byl případně výsledek zpracování.

Kapitola 6

Souhrn výstupů

Poslední kapitola práce se zabývá porovnáním naimplementovaných architektur na základě několika faktorů.

6.1 Testování

Začneme porovnáním, jak se jednotlivé architektury testují, protože zde není žádný významný rozdíl.

Všechny architektury dosahují vysoké abstrakce a nízké provázanosti. Tyto vlastnosti velmi napomáhají psaní testů. Většina tříd má pouze pár referencí na jiné objekty, nejvíce jich mají třídy v aplikační vrstvě, takže při psaní testů nebylo třeba trávit dlouhou dobu „mockováním“ chování navázaných objektů ani vytvářet složité mechanizmy nastavující prostředí.

Jediný větší rozdíl se vyskytl u Hexagonal architektury. Napojení modulů má na starosti knihovna *AutoFac* a při psaní integračních testů jsme museli specifikovat jakou aktuální konfiguraci má zvolit. Ostatní architektury mají jasně definované reference na implementace rozhraní.

6.2 Implementace

Na sekci testování navážeme porovnáním složitosti implementace jednotlivých architektur.

Úvodem bych rád zapojil i architekturu mikroslužeb. Úsilí vynaložené pro implementaci ekosystému mikroslužeb je výrazně vyšší než implementace pomocí monolit architektury. Vývojář během vývoje nesčetněkrát narazí na neznalost jednotlivých technologií potřebných pro propojení jednotlivých služeb a jejich následné nasazení, konkrétněji například Docker a RabbitMQ. V rámci jednotlivých služeb si navíc vývojář musí dát pozor na úskalí, které by u Monolit aplikace nepotkal, jako například nastavení CORS, aby fungovala delegace dotazu na jednotlivé služby. V neposlední řadě, pokud chceme zachovat kon-

zistenci zpráv, které se posílají a přijímají, nesmíme zapomenout vypublikovat novou verzi knihovny s definicí jednotlivých zpráv.

Mikroslužby tedy obsahují mnoho nových úskalí, které se při implementaci stejného produktu monolitickou architekturou řešit nemusí. Na druhou stranu přinesly i výhody na poli rozšířitelnosti a přehlednosti projektů a kódů. K tomu se dostaneme v dalších sekcích této kapitoly.

Co se týče implementace ostatních architektur, tak implementace Onion architektury a Clean architektury je téměř totožná. Jak bylo již možné vytušit z návrhu jednotlivých služeb, rozložení zodpovědnosti mají velmi podobné, liší se zde převážně jen pravidla pro pojmenování jednotlivých částí nebo pojmenování jednotlivých vrstev. Po stránce obtížnosti je nejtěžší si uvědomit zařazení funkcionality do správné vrstvy aplikace. Samotné propojení částí je pak velmi jednoduché.

Největším rozdílem je opět Hexagonal architektura. Zde bylo rozdělení logiky oproti Clean a Onion architektuře jednodušší. Je to z velké části díky tomu, že jádro aplikace tvoří pouze doménové entity a logika aplikace. Každá další logika, ať jde o webového rozhraní, komunikační notifikátory nebo databáze jsou stejným způsobem rozděleny do samostatných uzavřených modulů a připojeni k jádru pomocí adaptérů a portů. Takže, co není aplikační logika, je vlastní modul připojující se k systému, nemusí se zde řešit, jestli to je prvek infrastruktury nebo kohokoliv jiného a k jakým třídám a funkcím má právo přistupovat .

6.3 Rozšiřování

Po stránce rozšiřování si vedou všechny architektury velmi dobře. V rámci všech architektur se pro novou logiku vytvoří nové rozhraní definované v aplikační vrstvě a následně se rozhodne kde tato funkcionalita bude naimplementována.

Zda-li se jedná o aplikační logiku, může být u Onion a Clean architektur těžké rozhodnout správné umístění do odpovídající třídy. U Hexagonální architektury je to trochu jednodušší. Poněvadž je aplikován Domain Driven Design, vytváří se pro každou novou logiku nová třída vykonávající tuto logiku.

Pokud vytváříme logiku mimo aplikační jádro, například definice nového emailového agenta, v Hexagonal architektuře vytvoříme nový modul. U ostatních architektur musíme opět rozhodnout o správném umístění.

V obou případech nesmíme zapomenout zaregistrovat novou implementaci do nástroje vkládání referencí. Zde jsou na tom lépe Onion a Clean architektura, protože specifikují přidání reference na jednom konkrétním místě. U Hexagonální architektury se může stát, že budeme muset přidat registraci do více konfiguračních souborů, podle toho jaké kombinace nástrojů podporujeme.

V rámci rozšiřování nabízí Hexagonal architektura vysokou znovupoužitelnost modulů v jiných projektech. Modul v sobě obsahuje všechny potřebné věci, aby se mohl zapojit a byl plně funkční. Lze tento modul tedy vzít a zapojit do jiného projektu pomocí adaptéru a modul se postará o své nakonfigurování.

Co se týče rozšiřovaní monolitické aplikace oproti mikroslužbám, je zde velké usnadnění v tom, že je aplikace jeden celek. Nemusíme řešit, zda-li máme nastavené správné reference na sdílené knihovny, jestli máme jejich poslední verzi nebo jestli aplikace správně přijímá zprávy z komunikačního kanálu.

Na druhou stranu je zde problém kolizí, které jsem v rámci vyhotovení praktické části nemohl detekovat, a rozdíl v přehlednosti.

Pokud potřebujeme ekosystém mikroslužeb rozšířit způsobem, že přidáme novou službu, musíme nejen naprogramovat logiku, kterou potřebujeme, ale také službu přidat do sytému hlídání zdraví, zaregistrovat do patřičných *Gateway* pro delegaci požadavků, přidat do komunikačního kanálu a podobně. Pokud ale jde o rozšíření stávající logiky, je zde znatelný rozdíl ve prospěch mikroslužeb. Rozdělení zodpovědnosti mezi služby jasně určuje místo, kam se logika má dopsat.

6.4 Přehlednost

Na poli přehlednosti kódu jednoznačně vede Hexagonální architektura. Rozdělení na moduly jasně ohraničuje hranice funkcionality a její komunikaci s okolím. Je tedy snadné detekovat s kým funkcionalita komunikuje a i přehlednost samotné funkcionality je vysoká, protože je všechna na jednom místě.

V kategorii přehlednosti vyhrávají i mikroslužby nad monolitickou architekturou. Rozdělení na malé služby s jasnou funkcionalitou napomáhají vyhledávání funkcionalit a jejich interakci s okolím. Také díky rozdělení funkcionality mezi více služeb, viz. například náš příklad s naskladněním zboží 4.27, je množství kódu v jednotlivých metodách znatelně menší a přehlednější, než kdyby se vyskytovalo vše na jednom místě.

Závěr

Práce si kladla za cíl předat základní povědomí o architektonických typech a vzorech, jednotlivé architektury popsat, představit ukázkovou implementaci a nabídnout čtenáři porovnání implementovaných řešení.

V úvodní části práce zabývající se analýzou jsme si architektury představili, popsali si jejich základní strukturu, cíl, kterého se snaží dosáhnout, a krátký přehled o tom, kdy se vyplatí implementovat jakou architekturu.

Jakmile jsme si představili jednotlivé architektonické vzory, definovali funkcionalitu ukázkové aplikace, jež jsme následně zanalyzovali, dekomponovali na jednotlivé domény a ty přiřadili jednotlivým službám. Pro každou z domén jsme vybrali jeden architektonický typ, podle kterého jsme navrhli implementaci dané služby a komunikaci mezi jednotlivými částmi.

Na část návrhu navázala samotná implementace práce, ve které jsme si představili jednotlivé technologie použité pro implementaci, jejich konfiguraci a jakým způsobem napomáhají funkci celého systému. Pro každou službu jsme si ukázali specifickou část její funkcionality a pomocí digramu znázornili komunikaci s dalšími službami.

Nakonec jsme podrobili jednotlivé služby automatizovaným testům a popsali jsme si použité nástroje.

Na závěr jsme architektury porovnali na základě faktorů vyplývajících z implementační ukázky.

Na základě praktické části jsme došli k závěrům, že pokud se systém skládá z mnoha domén, je lepší se vydat cestou mikroslužeb a rozdělit zodpovědnost jednotlivých domén mezi samostatné služby. Dále se ukázalo, že Hexagonální architektura přináší vysokou přehlednost jednotlivých modulů, kterou doprovází vysoká režie za nastavení správného chování systému. Hexagonální architekturu se tedy vyplatí použít u větších, více komplikovaných aplikací nebo služeb využívající široké spektrum technologií. Pro menší aplikace a služby je dostatečné využít architektury Clean nebo Onion.

Vývoj aplikací šel za posledních několik let mílovými kroky dopředu. To, co se považovalo před lety za moderní a jedinečný způsob vývoje, je v dnešní době

překonáno novými metodikami, jež jsou opět unikátní a přináší nový pohled na vývoj samotný. Moderní architektury nám přinášejí možnost vývoje velkých škálovatelných aplikací, které jsou nezávislé na běžícím prostředí a odolné vůči výpadkům a jiným dříve kritickým nedostatkům.

Otázkou zůstává, za jak dlouho přijde nový způsob vývoje, který opět zanechá všechny pro nás nyní moderní postupy daleko za sebou a co to bude pro nás znamenat?

Bibliografie

1. TALEND. *What is monolithic architecture? Definition and examples* [online] [cit. 2020-04-23]. Dostupné z: https://www.talend.com/resources/monolithic-architecture/.
2. CHAN, Mike. *Monolithic Software: Microservices vs. Monoliths: What's the Right Architecture for your Software* [online] [cit. 2020-02-17]. Dostupné z: https://www.thorntech.com/microservices-vs-monoliths-whats-right-architecture-software/.
3. AKHTAR, Jahid. *Microservices Introduction (Monolithic vs. Microservice Architecture)* [online] [cit. 2020-02-17]. Dostupné z: https://dzone.com/articles/microservices-1-introduction-monolithic-vs-microse.
4. PACKT. *Understanding problems with the monolithic architecture style* [online] [cit. 2020-02-19]. Dostupné z: https://subscription.packtpub.com/book/web_development/9781785887833/1/ch01lvl1sec9/understanding-problems-with-the-monolithic-architecture-style.
5. OMELCHENKO, Igor. *Monolithic vs. Microservices - the choice that defines the whole development process.* [Online] [cit. 2020-02-19]. Dostupné z: https://clockwise.software/blog/monolithic-architecture-vs-microservices-comparison/.
6. RODGERS, Peter. *Service-Oriented Development on NetKernel- Patterns, Processes Products to Reduce System Complexity* [online] [cit. 2020-02-19]. Dostupné z: https://web.archive.org/web/20180520124343/http://www.cloudcomputingexpo.com/node/80883.
7. MAUERSBERGER, Laura. *Microservices: What They Are and Why Use Them* [online] [cit. 2020-02-19]. Dostupné z: https://www.leanix.net/en/blog/a-brief-history-of-microservices.

8. MERSON, Paulo. *Principles for Microservice Design: Think IDEALS, Rather than SOLID* [online] [cit. 2020-02-19]. Dostupné z: https://www.infoq.com/articles/microservices-design-ideals/.
9. NKENGSA, Brice. *I've heard I can use any language for each microservice, is that correct?* [Online] [cit. 2020-02-21]. Dostupné z: https://medium.com/the-andela-way/ive-heard-i-can-use-any-language-for-each-microservice-is-that-correct-6bc63ca910db.
10. SOLACE. *Microservices Advantages and Disadvantages: Everything You Need to Know* [online] [cit. 2020-04-04]. Dostupné z: https://solace.com/blog/microservices-advantages-and-disadvantages.
11. INC., Cloud Academy. *Advantages and Disadvantages of Microservices Architecture* [online] [cit. 2020-04-04]. Dostupné z: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/.
12. RICHARDSON, Chris. *Microservice Architecture pattern. What are microservices?* [Online] [cit. 2020-02-21]. Dostupné z: https://microservices.io/patterns/microservices.html.
13. SOLARWINDS WORLDWIDE, LLC. *What Is a Hypervisor? Hypervisor Definition, Types, and Examples* [online] [cit. 2020-04-04]. Dostupné z: https://www.dnsstuff.com/what-is-hypervisor.
14. SEMENOV, Serge. *Stem in Onion Architecture or Fallacy of Data Layer* [online] [cit. 2020-03-15]. Dostupné z: https://medium.com/@sergiis/stem-in-onion-architecture-or-fallacy-of-data-layer-9923f398f215.
15. MAINA, Duncan. *Origins of Model View Controller. MVC (model view controller) is a...* [Online] [cit. 2020-02-27]. Dostupné z: https://medium.com/@duncandevs/origins-of-model-view-controller-d685528857ce.
16. MICROSOFT. *.NET programming languages* [online] [cit. 2020-04-01]. Dostupné z: https://dotnet.microsoft.com/languages.
17. RABBITMQ. *Messaging that just works* [online] [cit. 2020-04-04]. Dostupné z: https://www.rabbitmq.com/.
18. CLOUDAMQP. *Part 1: RabbitMQ for beginners - What is RabbitMQ?* [Online] [cit. 2020-04-04]. Dostupné z: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html.
19. BEQIROVSKI, Fatiha. *RabbitMQ Exchange Types* [online] [cit. 2020-04-04]. Dostupné z: https://medium.com/trendyol-tech/rabbitmq-exchange-types-d7e1f51ec825.
20. RABBITMQ. *What can RabbitMQ do for you* [online] [cit. 2020-04-05]. Dostupné z: https://www.rabbitmq.com/features.html.

21. RABBITMQ. *Messaging that just works* [online] [cit. 2020-04-05]. Dostupné z: https://www.rabbitmq.com/devtools.html.

22. LOGZ.IO. *Kafka vs. Redis: Log Aggregation Capabilities and Performance* [online] [cit. 2020-03-05]. Dostupné z: https://logz.io/blog/kafka-vs-redis/.

23. PEDAMKAR, Priya. *RabbitMQ vs Redis* [online] [cit. 2020-03-05]. Dostupné z: https://www.educba.com/rabbitmq-vs-redis/.

24. MASSTRANSIT. *What does MassTransit add to the transport?* [Online] [cit. 2020-04-05]. Dostupné z: https://masstransit-project.com//understand/additions-to-transport.html.

25. DOCKER. *What is a Container?* [Online] [cit. 2020-03-12]. Dostupné z: https://www.docker.com/resources/what-container.

26. DOCKER. *Docker overview* [online] [cit. 2020-04-06]. Dostupné z: https://docs.docker.com/get-started/overview/.

27. DOCKERHUB. *Docker Hub* [online] [cit. 2020-04-06]. Dostupné z: hub.docker.com.

28. DOCKER. *Overview of Docker Compose* [online] [cit. 2020-04-06]. Dostupné z: https://docs.docker.com/compose/.

29. STACKIFY. *Docker Image vs Container: Everything You Need to Know* [online] [cit. 2020-04-06]. Dostupné z: https://stackify.com/docker-image-vs-container-everything-you-need-to-know/.

30. CONSULTING, Altkom Software. *Building API Gateways With Ocelot* [online] [cit. 2020-04-05]. Dostupné z: https://altkomsoftware.pl/en/blog/building-api-gateways-with-ocelot/.

31. DZONE. *Ocelot: The API Gateway Framework for .NET* [online] [cit. 2020-04-05]. Dostupné z: https://dzone.com/articles/ocelot-the-api-gateway-framework-for-net.

32. MICROSOFT. *Visual Studio Code - Code Editing* [online] [cit. 2020-04-23]. Dostupné z: https://code.visualstudio.com/.

33. MICROSOFT. *C programming with Visual Studio Code* [online] [cit. 2020-04-23]. Dostupné z: https://code.visualstudio.com/docs/languages/csharp.

34. MICROSOFT. *Docker - Visual Studio Marketplace . Extensions for Visual Studio family of products* [online] [cit. 2020-04-23]. Dostupné z: https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker.

35. MONGODB. *MongoDB for VS Code* [online] [cit. 2020-04-23]. Dostupné z: https://www.mongodb.com/products/vs-code.

36. JMROG. *NuGet Package Manager - Visual Studio Marketplace . Extensions for Visual Studio family of products* [online] [cit. 2020-04-23]. Dostupné z: `https://marketplace.visualstudio.com/items?itemName=jmrog.vscode-nuget-package-manager`.

37. TABNINE. *Code Faster with AI Code Completions* [online] [cit. 2020-04-23]. Dostupné z: `https://www.tabnine.com/`.

38. MICROSOFT. *Project documentation - Project* [online] [cit. 2020-04-01]. Dostupné z: `https://docs.microsoft.com/en-us/project/`.

39. SOFTWARE, SmartBear. *API Documentation Design Tools for Teams* [online] [cit. 2020-04-23]. Dostupné z: `https://swagger.io/`.

40. AUTOFAC. *Welcome to Autofac's documentation!* [Online] [cit. 2020-04-23]. Dostupné z: `https://autofaccn.readthedocs.io/en/latest/`.

41. NSUBSTITUTE. *NSubstitute: A friendly substitute for .NET mocking libraries* [online] [cit. 2020-04-23]. Dostupné z: `https://nsubstitute.github.io/`.

Příloha A

Seznam použitých zkratek

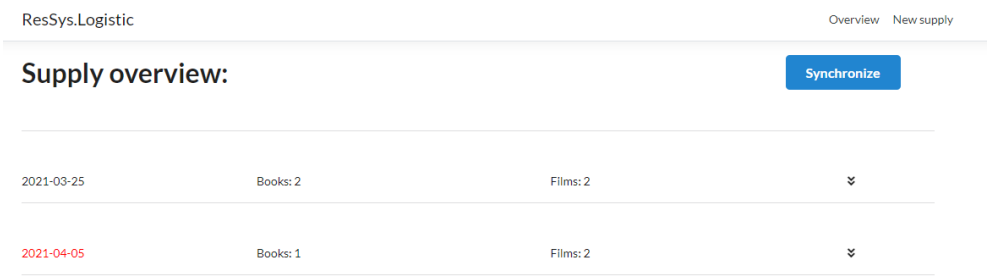
UI User Interface

API Application Programming Interface

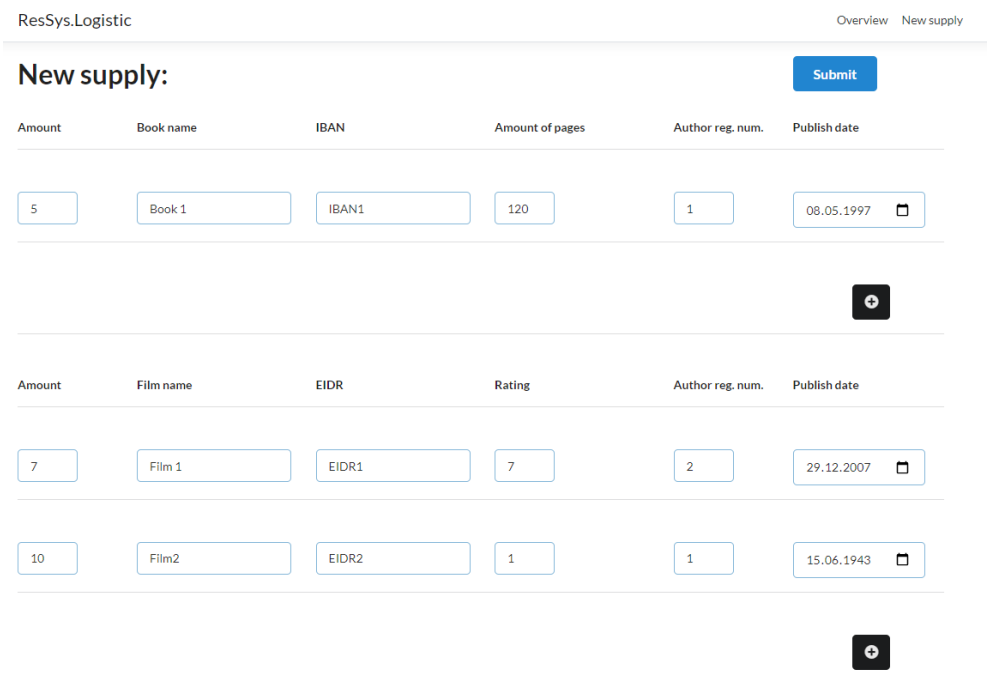
ORM Object–relational mapping

Příloha B

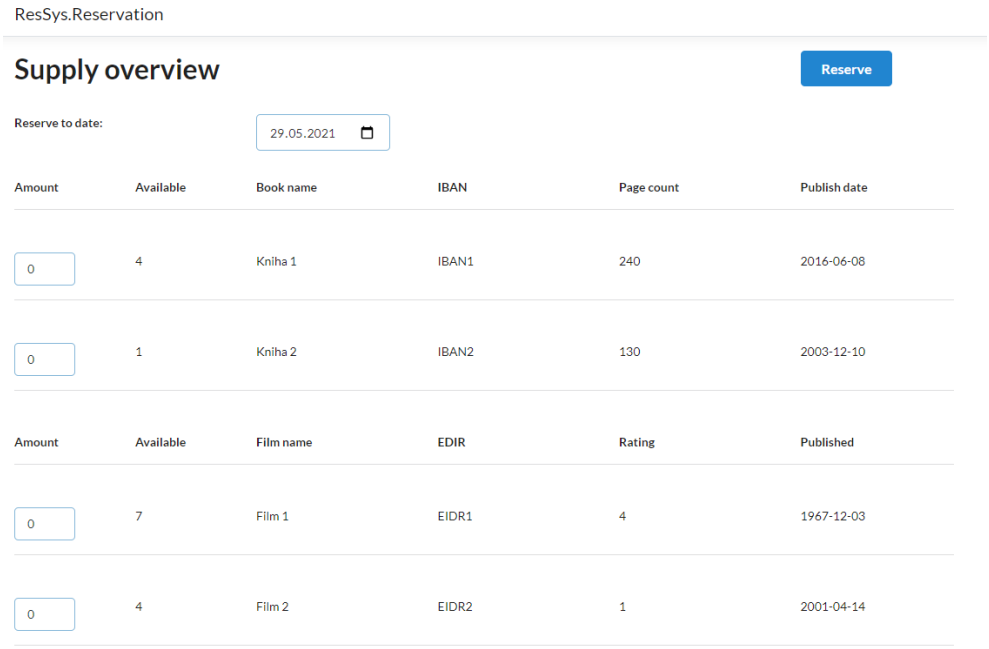
Snímky obrazovek



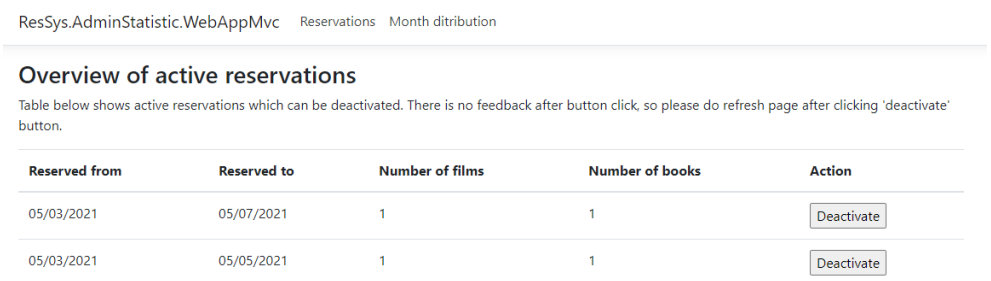
Obrázek B.1: Přehled zásilek



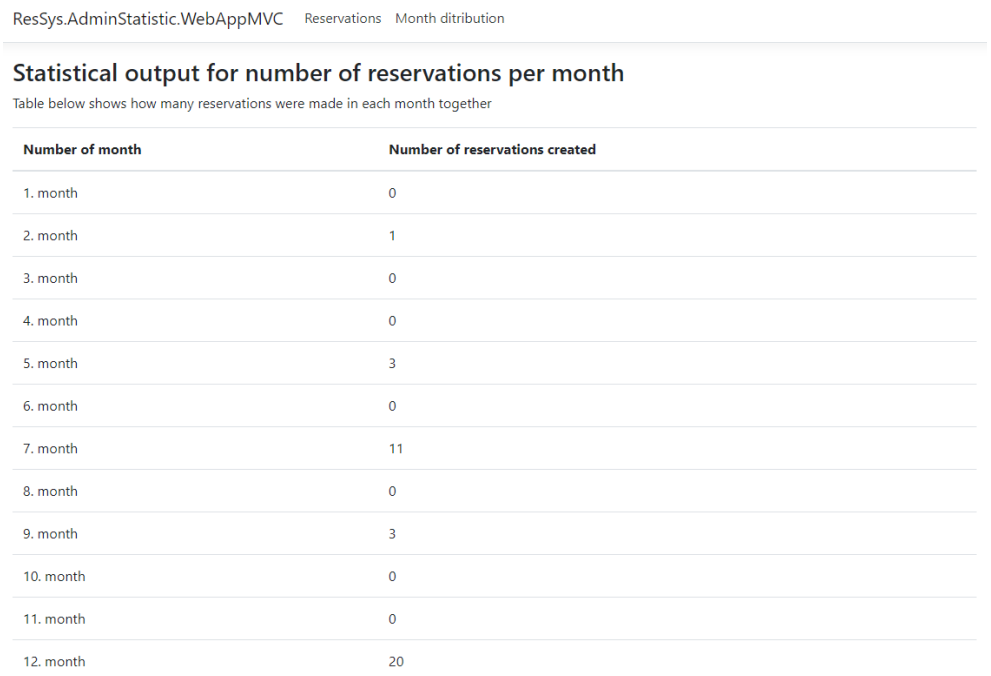
Obrázek B.2: Vytvoření nové zásilky



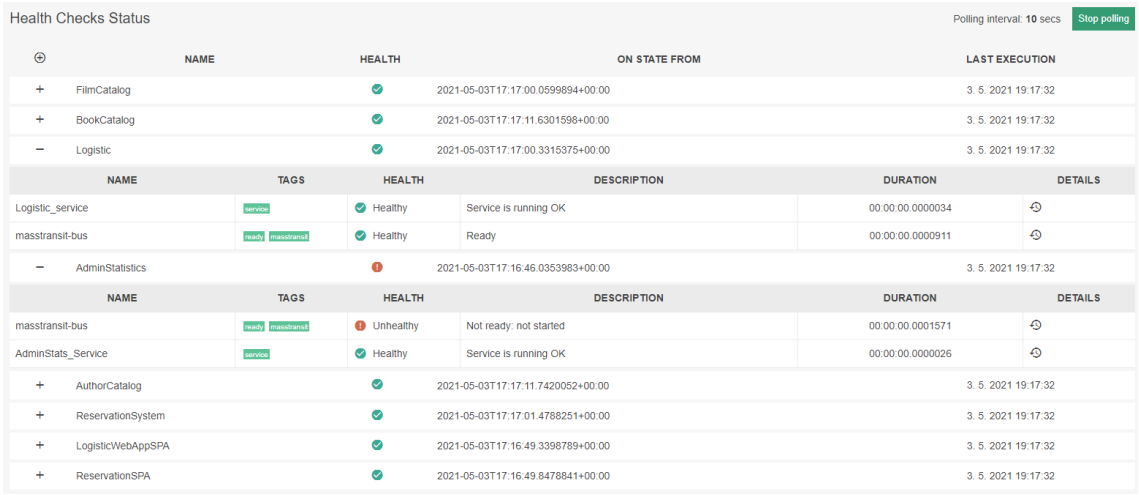
Obrázek B.3: Rezervační systém



Obrázek B.4: Přehled aktivních rezervací



Obrázek B.5: Statistické výstupy



Obrázek B.6: Přehled zdraví služeb

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `data`
  - `src` ..... adresář se zdrojovými kódy aplikace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF