FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Automatizované testování aplikací s mikroservisní architekturou
Student:	David Hospodka
Vedoucí:	Ing. Martin Komárek
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

1. Nastudujte problematiku automatizovaného testování z odborné literatury a pokuste se zjistit aktuální přístupy velkých a středních technologických firem.
2. Popište specifika testování mikroservisní architektury a rozdíly oproti testování monolitické architektury.
3. Kompletně otestujte referenční aplikaci společnosti Stratox na prodej jízdenek a popište technologie, které jste k testování použil. Dokumentujte všechny nalezené chyby.
4. Na základě nastudovaných přístupů a osobní zkušenosti navrhněte doporučení pro efektivní testování aplikací postavených na mikroservisní architektuře.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 10. února 2024 v Praze.

Bakalářská práce

AUTOMATIZOVANÉ TESTOVÁNÍ APLIKACÍ S MIKROSERVISNÍ ARCHITEKTUROU

David Hospodka

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Martin Komárek,
16. května 2024

Odkaz na tuto práci: Hospodka David. *Automatizované testování aplikací s mikroservisní architekturou.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Chtěl bych poděkovat především mému vedoucímu, panu Ing. Martinu Komárkovi za vedení této bakalářské práce, za pomoc při vymýšlení tématu a za průběžné konzultace. Dále bych chtěl poděkovat panu Ing. Robinu Vávrovi za seznámení s testovanou demo aplikací a také za rady a přátelský přístup při konzultacích této práce.

Prohlášení

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

V Praze dne 16. května 2024

Abstrakt

Bakalářská práce se zabývá problematikou automatizovaného testování v mikroservisním prostředí, představuje jednotlivé způsoby testování a uvádí, jaké jsou rozdíly oproti testování monolitické architektury. Veškeré poznatky doplňuje aktuálním přístupem různých technologických společností. V praktické části popisuje tvorbu automatizovaných testů pro demo aplikaci společnosti Stratox a v závěru představuje autorovo vlastní doporučení pro automatizované testování mikroservisní architektury.

Klíčová slova automatizované testování, mikroservisní architektura, unit testy, contract testy, integrační testy, component testy, end-to-end testy, load testy

Abstract

Bachelor thesis deals with the issue of automated testing in a microservices environment, introduces the different ways of testing and shows what are the differences compared to testing a monolithic architecture. It complements all the knowledge with the current approach of different technology companies. In the practical part, it describes the creation of automated tests for a demo application of Stratox and concludes with the author's own recommendations for automated testing of microservices architecture.

Keywords test automation, microservice architecture, unit tests, contract tests, integration tests, component tests, end-to-end tests, load tests

Seznam zkratek

API	Application Programming Interface
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
URL	Uniform Resource Locator
DSL	Domain Specific language
UI	User Interface
AI	Artificial Inteligence
QA	Quality Assurance
IT	Informační technologie

Úvod

Bakalářská práce se zabývá automatizovaným testováním softwaru postaveném na mikroservisní architektuře. Mikroservisní architektura je moderní softwarová architektura, jejíž hlavní princip spočívá v rozdělení softwaru do menších, na sobě nezávislých komponent zvaných mikroservisy. Díky tomuto rozdělení na sobě mohou být týmy, které jednotlivé mikroservisy spravují, více nezávislé, a to vede k agilnějšímu vývoji a snadnějšímu nasazování. Tato architektura však přináší nové výzvy mimo jiné i v oblasti testování.

Automatizované testování se stalo klíčovým prvkem v oblasti udržení kvality softwaru a minimalizace rizik spojených s nasazováním softwaru. Tato práce se hlouběji zabývá problematikou automatizovaného testování v mikroservisním prostředí.

Cílem této práce je nejen analyzovat klíčové způsoby testování mikroservisní architektury, ale také poskytnout praktické nástroje a doporučení pro vývojáře a testery.

Kapitola 1
Teoretická část

1.1 Od monolitické k mikroservisní architektuře

Mikroservisní a monolitická architektura jsou dvě různé architektury v oblasti vývoje softwaru, které určují jakým způsobem je software vyvíjen, udržován, nasazován a také testován. Monolitická architektura je starší a více tradiční architektura. Je definována tím, že strukturuje celý systém jako jednotnou nasaditelnou jednotku. Tato architektura má zpočátku vývoje několik výhod. Mezi ně patří:

- Jednoduchý vývoj - vývojová prostředí a ostatní nástroje vývojářů se zaměřují na vývoj jednotných aplikací.
- Snadné provedení radikálních změn aplikace - je snadné změnit kód, databázové schéma, provést build a nasadit.
- Přímočaré testování - Je jednoduché psát end-to-end testy, které testují celý chod aplikace.
- Přímočaré nasazení - jednotně spustitelná aplikace se snadno nasazuje.

S postupem času a s rostoucí velikostí aplikace se však výhody této architektury vytrácí. Aplikace se stává příliš komplexní a nikdo z vývojářů už jí kompletně nerozumí. Výsledkem toho je velice pomalý vývoj a složité opravování chyb. Testování takové aplikace se stává velice komplexním a obtížným. Chyba v jedné části aplikace může navíc ohrozit chod celého systému. [1]

Všechny výše zmíněné problémy se pokouší řešit mikroservisní architektura. Ta je definována jako architektonický styl, který dekomponuje aplikaci do jednotlivých servis. Tyto servisy jsou volně provázané a komunikují spolu výhradně pomocí API. Jednotlivé servisy nazýváme mikroservisy[1]. Velikost jednotlivých mikroservis není přesně definována. Vývoj každé by však měl být řízen jedním menším týmem [2]. V praxi často spravuje jeden tým více mikroservis zároveň. Tato architektura přináší několik výhod:

- Umožňuje průběžné doručování softwaru u velkých komplexních aplikací.
- Jednotlivé servisy jsou malé a snadno udržovatelné. Jsou samostatně nasaditelné a dají se samostatně škálovat.
- Jednotlivé týmy jsou na sobě více nezávislé.
- Umožňuje pro každou servisu zvolit jiné technologie.
- Lépe izoluje chyby. [1].

Z pohledu automatizovaného testování je testování jednotlivých mikroservis jednodušší, protože jsou menšího rozsahu. Testování celé aplikace je však komplexnější. Více o tom v dalších kapitolách.

1.2 Automatizované testování

Automatizace testování softwaru popisuje jakýkoli proces, který zahrnuje použití samostatných softwarových nástrojů k testování vyvíjeného softwaru. Tyto nástroje využívají skriptované sekvence k přezkoumání a ověření produktů s podstatně menším zásahem člověka než tradiční testovací techniky. [3] Tradiční přístup k testování softwaru probíhal vždy až po dokončení vývoje manuálním způsobem. Tento způsob testování je však velice neefektivní. Lidské testování je oproti tomu automatizovanému extrémně pomalé. Testování až po dokončení vývojového cyklu je důležité ale samo o sobě nedostačující. Pro kvalitní vývoj softwaru je nutné testovat aplikaci už během vývoje. Ačkoliv jsou tyto fakta v komunitě vývojářů známa už delší dobu, aktuální průzkumy ukazují, že pouhých 18 % společností využívá převážně automatizovaného testování a pouhé 3 % mají proces testování plně automatizovaný[4]. Neefektivní manuální testování tedy hraje napříč společnostmi stále velkou roli.

Hlavními důvody, proč se vývojáři často vyhýbají psaní automatizovaných testů, je menší atraktivita psaní jejich kódu oproti kódu produkčnímu. Dalšími nejčastěji zmiňovanými důvody v tomto průzkumu [5] mezi vývojáři je malé množství času, který jim je umožněn věnovat psaní automatizovaných testů a chybějící kvalitně zavedený testovací proces v rámci organizace.

Osobně si myslím, že atraktivitu psaní automatizovaných testů může zvednout povědomí o tom, jak správně tyto testy psát. Po nabrání jisté praxe nezabere psaní těchto testů mnoho času. Pokud si bude vývojář vědom výhod, které mu tyto testy přináší a pozná kvalitu, kterou přinesou jeho práci, bude mít mnohem větší zájem se psaní automatizovaných testů věnovat. Ostatně i průzkum zmíněný v [6] ukazuje, že psaní testů zabere vývojářům mnohem méně času, než si ve skutečnosti myslí.

Podobné platí i pro celé organizace. Psaní automatizovaných testů jistě v úvodu zpomalí vývoj, neboť bude potřeba věnovat prostředky na jejich řádné proškolení v této oblasti a také jim vyčlenit čas, který budou psaní testů věnovat. V dlouhodobém měřítku se však tento čas vrátí, neboť v důsledku

automatizované testy zredukují počet chyb, které bude potřeba v budoucnu opravit a celkově zlepší kvalitu dodávaného softwaru. Tím zajisté dosáhnou organizace úspory peněz. Stejných závěrů došel experiment z roku 2016. [7]

Aby testování bylo efektivní, je potřeba znát různé typy automatizovaných testů a specifika architektury testované aplikace.

1.3 Způsoby automatizovaného testování v mikroservisním prostředí

V této kapitole bude popsána zavedená teorie ohledně testovacích způsobů, které se aplikují při testování mikroservisní architektury. Následující typy testů ověřují správnou funkcionalitu aplikace:

- **Unit testy**: testují malou část kódu. Např. jednu funkci.
- **Contract testy**: testují dodržování contractu mezi dvěma mikroservisami.
- **Integrační testy**: testují, že aplikace dokáže komunikovat s externími systémy (databází, jinou mikroservisou apod.). Mohou také testovat spolupráci jednotlivých vrstev testované aplikace.
- **Component testy**: akceptační testy pro jednotlivé servisy.
- **End-to-end testy**: akceptační testy pro celou aplikaci. [1]

Dále budou v této práci popsány **load testy**, které zkoumají funkcionalitu aplikace pod větší zátěží.

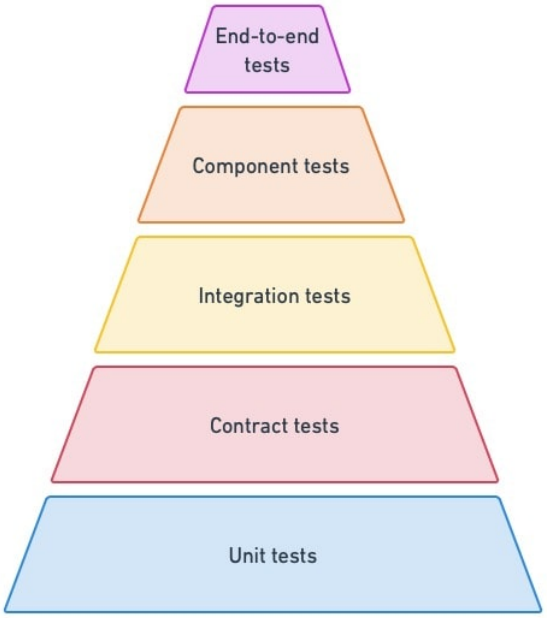
Rozdělení testů dle jejich rozsahu je dnes standard. Nemusí to však být nutně jediný možný způsob kategorizace. Např. společnost Google dělí své testy do tří kategorií následujícím způsobem:

- **Malé testy**: běží na jednom procesu.
- **Střední testy**: běží na jednom zařízení.
- **Velké testy** : běží kdekoliv.

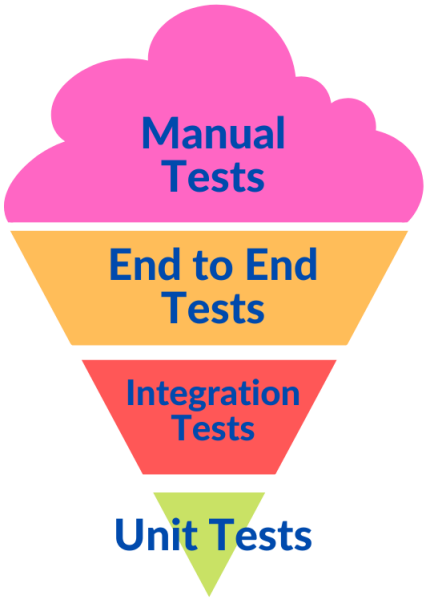
Důvodem pro toto rozdělení jsou dvě pro Google nejdůležitější vlastnosti testů: rychlost a determinismus. Menší testy jsou vždy rychlejší a determinističtější než velké testy. [8]

Co se týče množství testů, které je pro jednotlivé kategorie obecně doporučováno, již několik let panuje všeobecná shoda na takzvané testovací pyramidě, kterou zachycuje obrázek 1.1.

Toto rozdělení říká, že nejvíce by mělo být při testování vytvořeno unit testů. Při postupu pyramidou nahoru nabývají testy rozsahu a stávají se složitějšími. Takových testů by se mělo vytvářet méně a méně. [1]



Obrázek 1.1 Testovací pyramida [9]



Obrázek 1.2 Zmrzlinový pohár [10]

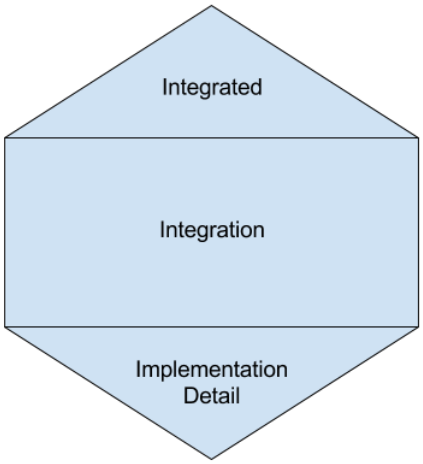
V praxi je však často k vidění jiný přistup, popsaný jako testovací zmrzlinový pohár. Tento přístup zachycuje obrázek 1.2

V takovém rozložení hraje největší roli pomalé a neefektivní manuální tes-

tování. Při sestupu pohárem dolů k jednodušším a rychlejším testům, jejich počet ubývá. Toto rozložení je také označováno jako testovací anti-pattern. [8]

Nejčastěji bývá tedy doporučována již zmíněná testovací pyramida. Google se podle ní také řídí a dokonce uvádí, že unit testy tvoří 80 % všech napsaných testů. End-to-end testy oproti tomu tvoří pouhých 5 %. [8]

Pro zajímavost uveďme i přístup Spotify, ve kterém se u testování mikroservisní architektury řídí podle tzn. včelí plástve, která je vyobrazena na obrázku 1.3. Včelí plástev říká, že ve Spotify se nejvíce zaměřují na integrační



Obrázek 1.3 Včelí plástev [11]

testy. Snaží se naprosto limitovat množství integrovaných testů, které definují jako testy, u kterých úspěšné doběhnutí závisí na jiném systému. Snaží se i omezit množství testů, které testují konkrétní implementaci (např. právě unit testy). Jejich cílem je tedy co nejvíce izolovat testování konkrétní mikroservisy a důkladně otestovat integrační body.

obrázek 1.4 zachycuje, jak takový běžný integrační test ve společnosti Spotify funguje. Test spustí databázi pomocí technologie TestContainers a naplní ji nějakými daty. Nastartuje testovanou mikroservisu a pošle na ni request. Následně zkontroluje důsledky zaslaného requestu. Výhodou takového testu je, že i pokud kompletně změníme implementaci operace, kterou testujeme, test úspěšně doběhne. Na druhou stranu pokud test doběhne neúspěšně, bude složitější zjistit, kde je problém. Tento test také poběží výrazně déle, než běžný unit test. [11]

```java
@BeforeClass
public static void setupClass() throws Exception {
  startPostgresFromTestContainers();
  seedDatabaseWithTestComponent();
  startService();
}

@AfterClass
public static void tearDownClass() {
  stopService();
  stopPostgresFromTestContainers();
}

@Test
public void shouldReturnComponent() throws Exception {
  final Request request = Request.forUri("/components/testcomponent");
  final Response<ByteString> response = serviceHelper.serviceClient()
      .send(request).toCompletableFuture().get();

  assertThat(response.status(), is(OK));

  final ComponentModel component = fromJson(
      response.payload().get().utf8(), ComponentModel.class);

  assertThat(component.component(), is("testcomponent"));
  assertThat(component.org(), is("testorg"));
  assertThat(component.repo(), is("testrepo"));
}
```

Obrázek 1.4 Ukázka integračního testu Spotify [11]

1.4 Rozdíly oproti testování monolitické architektury

Hlavní charakteristikou této architektury je její distribuovanost. Celá aplikace se skládá z mnoha jednotlivých mikroservis, a vývoj každé může být řízen jiným týmem. Správná strategie testování takové architektury je testovat každou mikroservisu izolovaně zvlášť. Tím je dosaženo toho, že si testování své mikroservisy může každý tým řídit sám a testuje tedy pouze tu část aplikace, které nejvíce rozumí. V praxi jsou často využívány techniky jako mocking, stubbing a další specializované nástroje, které izolaci testovacího procesu umožní. Důsledkem toho bude celý proces testování rychlejší, jednodušší a kvalitnější. [12]

Jednou z největších výzev této architektury je komunikace jednotlivých mikroservis skrze API. Testování takové komunikace je výrazně složitější, než testování v monolitické architektuře, kde pouze dochází k provolání metod jiných modulů. V mikroservisní architektuře si producent neboli mikroservisa, která implementuje nějaké API, není vědoma potřeb svých konzumentů, neboli mikroservis, které toto API využívají. Pokud nějaký producent upraví své API, tak ačkoliv ho důkladně otestuje, nemůže mít garanci, že nerozbil funkcionalitu nějakého konzumenta. [13] Jak tedy garantovat správnou inter-

akci jednotlivých servis a zároveň zachovat izolovanost testovacího procesu? Řešením jsou integrační a contract testy. Více o nich v dalších kapitolách.

Dalším rozdílem je výrazně složitější vývoj end-to-end testů v mikroservisním prostředí. Tento typ testů vyžaduje pro své spuštění nasazení celé aplikace. To může být v praxi složité. Proto je vhodné tento typ testů v mikroservisním prostředí minimalizovat. Novým typem testování v tomto prostředí jsou naopak tzn. component testy, které testují každou mikroservisu izolovaně zvlášť a měly by garantovat správnou funkčnost každé jednotlivé mikroservisy. Více bude tento typ testování představen v dalších kapitolách.

Testování business logiky je v mikroservisním prostředí jednodušší, protože jednotlivé mikroservisy jsou menšího rozsahu, a tak se funkcionalita každé z nich testuje jednodušeji.

Celkově je testování mikroservisní architektury komplexnější, neboť vyžaduje znalost více způsobů automatizovaných testů. To vyžaduje větší znalosti ohledně automatizovaného testování všech členů týmu.

1.5 Unit testy

Unit testy jsou naprostým základem testování jakéhokoliv softwaru. Jedná se o typ testů, které testuje jednotlivé metody, třídy nebo dokonce několik tříd spolupracujících dohromady. Toto testování probíhá v izolaci a má několik výhod:

- Unit testy jsou rychlé. Obvyklá doba běhu je několik milisekund.
- Jsou jednoduše ovladatelné. Je snadné změnit vstup, který testovacím metodám vkládáme a upravit výstup, který očekáváme.
- Jednoduše se vytváří. K testování většinou využíváme nějaký testovací framework a k naprogramování jednoho testu nám obvykle stačí napsat pár řádků kódu.

Oproti tomu mají však i několik nevýhod:

- Unit testy netestují realitu. Skutečné systémy se skládají z mnoha tříd a jejich vzájemná interakce může způsobit odlišné chování od toho, které testujeme v izolaci.
- Některé typy chyb unit testy nezachytí. Některé chyby vznikají až po vzájemné interakci tříd a takové chyby tyto testy neumí zachytit. [6]

Unit testy jsou dnes od vývojářů standardně vyžadovány. Podobně to má i společnost Google, kde každá změna kódu musí obsahovat také patřičné testy. Ve společnosti je tedy naprosto legitimní změny kódu při code-review zamítnout, pokud patřičné testy neobsahují. [8]. Ve společnosti Picnic jsou

unit testy také nejpreferovanějším typem testů a každý vývojář je nucen k produkčnímu kódu dodat sadu unit testů, které garantují celkovou funkcionalitu vyvíjené featury. V důsledku toho se mohou členové QA týmu více soustředit na testování celkové funkcionality dané componenty. [14]

Jako metriku pro zkoumání obsáhlosti testovacího procesu je možné použít tzn. line coverage nebo branch coverage. Line coverage říká, kolek procent řádků testovaného kódu bylo za běhu testu vykonáno. Branch coverage bere v potaz jednotlivé podmínky a cykly, které způsobují, že se program pod jiným vstupem může větvit a chovat různě. Tato metrika udává kolik % větví vykonávaného kódu testy obsáhnou. Z této definice je tedy zřejmé, že 100% branch coverage implikuje 100% line coverage. Opačná implikace neplatí. [6] Téměř všechny moderní vývojová prostředí nabízí výpočet těchto hodnot při spouštění testů a pro každého vývojáře/testera to může být velice praktický nástroj. Při používání těchto metrik, je důležité mít na paměti, že ani jedna nemluví o kvalitě testů, pouze o množství kódu, který testy projde. Existuje však dnes několik empirických důkazů o tom, že vyšší hodnota těchto metrik společně s množstvím testů alespoň lehce koreluje s nižší chybovostí softwaru. [6]

Jak již bylo zmíněno základem unit testů je, že testují jednotlivé třídy izolovaně. V produkčním kódu má však téměř každá třída závislost na nějaké jiné třídě. Tyto závislosti jsou nahrazovány tzn. mocky. Mock umí vracet předem nastavené hodnoty a zároveň umožňuje runtime verifikaci provolání nějakých metod. [2] Používání mocků je pro kvalitní unit testy naprosto základní a každý vývojář by měl být s tímto konceptem při psaní testů seznámen.

1.5.1 Technologie pro psaní unit testů

1.5.1.1 JUnit

je testovací framework, který umožňuje psát unit testy pro programovací jazyk Java. V současné době je rozšířen jinými nástroji, aby umožňoval i psaní jiných typů testů: např. integrační testy nebo component testy. V ekosystému Javy je JUnit považován za naprostý standard. [2]

1.5.1.2 Mockito

je nejpoužívanější mockovací framework v ekosystému Javy a umožňuje vytvářet mockované objekty, které jsou potřeba pro kvalitní izolaci unit testů. Mockito je v ekosystému Javy jednou z nejpoužívanějších knihoven celkově, a tak je možné tuto knihovnu také považovat za standard v testování. [2]

1.5.1.3 AssertJ

je knihovna, která velice zlepšuje čitelnost vytvořených unit testů. Tato knihovna totiž obsahuje několik assertů, které je možné v testech použít a díky tomu

bude kód čitelnější. Dobrá čitelnost testů je velice důležitá. V budoucnosti bude pravděpodobně právě vyvíjena funkcionalita někým upravována, či refaktorována a tím pádem bude potřeba upravit i již napsané unit testy. Tím, že budou testy dobře čitelné, tedy ulehčíme práci sobě i našim kolegům.

1.6 Contract testy

Aplikace postavená na mikroservisní architektuře je distribuovaný systém, ve kterém se každé jednotlivé mikroservise neustále vyvíjí její API. Pro každý tým je esenciální psát takové testy, které dohlédnou na to, že jejich mikroservisa korektně interaguje s jejími závislostmi a jejími klienty. [1] To nás přivádí ke contract testům.

Komunikace mezi jednotlivý mikroservisami může mít různou podobu. Může se jednat o synchronní komunikaci skrze REST protokol, nebo např. asynchronní komunikaci přes výměnu zpráv. Jakákoliv interakce mezi dvěma mikroservisami představuje contract. Jakožto vývojář nějakého API, je pro vás klíčové, abyste nenarušil potřeby někoho, kdo vaše API konzumuje. Stejné platí i naopak. Je nesmírně důležité aby API, které vaše mikroservisa konzumuje bylo stabilní. Je potřeba automatizovaně testovat dodržování vzniklého contractu mezi dvěma mikroservisami z obou stran. [1]

Jedním způsobem jak otestovat dodržování nějakého contractu, by bylo spustit obě mikroservisy, vyvolat komunikaci mezi nimi a zkontrolovat očekávaný výstup. K tomuto přístupu, by však bylo potřeba mikroservisy společně spouštět. V rámci jednoho týmu by bylo nutné spouštět i mikroservisu, která může spadat pod vývoj jiného týmu. Bylo by navíc potřeba spustit i další závislosti, které tyto mikroservisy mají. Jednalo by se v podstatě o E2E testování. Množství takových testů je však dobré v testovacím procesu redukovat na naprosté minimum. Na této úrovni testů je stále vhodné uchovat testování v izolaci. Tím bude možné se zaměřit pouze na chování testované mikroservisy a testování bude rychlejší a jednodušší. Vhodným přístupem je tedy tzn. **consumer-driven contract testování.** [1]

Consumer contract je automatizovaný test na straně producera, tedy toho, kdo ve vzájemném vztahu vystavuje nějaké API. Při asynchronní komunikaci je v roli producera ten, kdo je odesílatelem asynchronní zprávy. Tento test kontroluje, že vystavené API, nebo asynchronní zpráva, má takovou podobu, kterou očekává consumer. Pro REST endpoint bude pravděpodobně contract testem kontrolováno, že:

- Má očekávanou HTTP metodu s konkrétní URL.
- Akceptuje očekávané hlavičky.
- Akceptuje konkrétní tělo.
- Vrací odpověď s konkrétním kódem, hlavičkami a tělem. [1]

Pro asynchronní komunikaci bude contract test kontrolovat, že jednotlivé zprávy:

- Mají správnou strukturu a správný formát.
- Consumer je schopný správně parsovat přijímané zprávy.

Fyzicky je contract nějaký dokument, který popisuje komunikaci mezi dvěma mikroservisami za pomocí DSL (domain specific language). Může být napsán v různých jazycích na základě technologií, které jsou využívány. Typicky se může jednat o JSON, YAML, Javu apod. [15]. Contract je vložen do repozitáře producera, kde je poté využit automatizovaným testem. Tento automatizovaný test prověří, že je tento contract dodržen ze strany producera.

Cílem těchto testů není důsledně otestovat business logiku. To mají na starost jiné kategorie testů. Cílem je především ověřit, že to, co jedna servise od druhé očekává, ta druhá skutečně poskytuje. Pokud úspěšně doběhnou na straně producera všechny contract testy od jeho consumerů, může si tým na straně producera být jistý, že v důsledku vývoje nenarušil očekávání nějakého jiného týmu.

Osobně si myslím, že contract testy jsou velice důležité pouze pokud je celá aplikace většího rozsahu. U menších aplikací je z mého pohledu konzistentnost API možné udržovat i bez těchto automatizovaných testů. Pokud jste však součástí nějaké větší organizace a vaše aplikace je tvořena desítkami různých mikroservis, která je každá spravována vlastním týmem, je tento druh testování z mého pohledu nedocenitelný. Obrovská výhoda je izolovanost, kterou opět přináší do celého testovacího procesu. Testy jsou jednoduché na vytvoření a rychle běží, což jsou velice ceněné vlastnosti v jakékoliv fázi vývoje. K jejich efektivnímu využití je však nutné nastavit si správnou politiku napříč organizací. Jednotlivé týmy musí být s konceptem contract testů seznámeni a musí být schopni si efektivně vkládat jednotlivé contracty do svých repozitářů (např. skrze pull request) [1].

1.6.1 Technologie pro psaní contract testů

1.6.1.1 Pact

je open-source code-first nástroj pro vytváření contract testů. Je dostupný pro mnoho programovacích jazyků a je samozřejmě zdarma. [16] V ekosystému Java se jedná standardní knihovnu, kterou lze nainstalovat např. skrze Maven.

1.6.1.2 Pact flow

je komerční nástroj, který umožňuje automatizovat celý proces contract testování. Contracty, které testy produkují jsou nahrávány na vzdálený server a následně z něj načteny při spouštění na straně producera. Využití tohoto nástroje tedy ulehčí od manuálního přeposílání contractů napříč repozitáři. Pro velké a komplexní aplikace se jedná o velice praktický nástroj. [17]

1.7 Integrační testy

Integrační testy kontrolují komunikaci s externími systémy. Mohou však také testovat dohromady více vrstev testované aplikace. Cílem tohoto typu testování je tedy ověřit např. správnou komunikaci s databází, jinou mikroservisou, nebo systémem pro výměnu asynchronních zpráv.

Pokud je testována komunikace s nějakým infrastrukturním systémem, jako je např. databáze, je postup při testování poměrně přímočarý:

1. Spustíme testovanou mikroservisu.
2. Spustíme externí systém.
3. Vyvoláme komunikaci mezi testovanou mikroservisou a externím systémem.
4. Otestujeme výstup.

Abychom dosáhli naprosté automatizace při tomto testování a samotný test spouštěl tedy i samotný externí systém použijeme technologii zvanou TestContainers. Ta bude více představena v závěru této kapitoly.

Zajímavější je testování komunikace mezi dvěma mikroservisami. Je potřeba opět myslet na izolování testované mikroservisy. Je dobré se vyhnout spouštění jiné mikroservisy, protože to by z testu dělalo spíše end-to-end test. [1] Proto je vhodné při testování komunikace mezi různými mikroservisami mockovat celé API externí mikroservisy. V automatizovaném testu se tedy za využití speciálních nástrojů nastaví různé scénáře, při kterých fiktivní API vrátí různá data. Takový test bude mít následující vlastnosti:

- Poběží izolovaně bez závislosti na jiné mikroservise.
- Ověří, že jsme korektně provolali API mockované servisy.
- Ověří, že námi testovaná mikroservisa dokáže správně zpracovat odpověď mockované mikroservisy.

Zde už by mělo být zřejmé, jak moc se tyto integrační testy doplňují s již popsanými contract testy. Contract testy zaručí, že si mikroservisy navzájem dodávají to, co očekávají. Izolované integrační testy naopak zaručí, že je testovaná mikroservisa schopna data do externích servis správně odeslat, přijmout je a případně s nimi dále pracovat.

1.7.1 Technologie pro psaní integračních testů

1.7.1.1 TestContainers

je open source framework, který umožňuje v našich testech jednoduše spouštět instance databází, systému pro výměnu zpráv nebo čehokoliv, co je schopné běžet v docker containeru. [18]

1.7.1.2 WireMock

je open source nástroj pro mockování API. Může být spuštěn jako samostatný server nebo v rámci virtuálního stroje Java. V rámci mockování API je považován za standard. [19]

1.8 Component testy

V mikroservisním prostředí je componentou každá jednotlivá mikroservisa. Tento typ testování tedy testuje celkovou funkcionalitu nějaké mikroservisy v izolaci. K nahrazení všech závislostí testované mikroservisy se použijí již zmíněné mockovací nástroje. Izolace testované mikroservisy bude mít několik pozitivních důsledků:

- Test bude jednodušeji spustitelný a konfigurovatelný.
- Test poběží rychleji.
- Testovaná funkcionalita nebude záležet na chování jiné mikroservisy.
- Bude možné snadněji konfigurovat různé testovací scénáře. [20]

V tomto způsobu testování už je vhodné k mikroservise přistupovat jako k černé skříňce a testovat celkově různé operace, které mikroservisa poskytuje. Ke spouštění infrastrukturních závislostí můžeme použít již popsanou technologii TestContainers. K nahrazení externích mikroservis již také popsanou technologii WireMock. Component testy rozdělujeme na dva typy.

In-process component test spustí aplikaci v operační paměti společně s testem. Takový typ testů často spustí i in-memory náhrady za externí systémy (např. databáze). Většina vývojových frameworků poskytuje různé možnosti, jak tohoto typu testování dosáhnout. Tento typ testů je výrazně jednodušší na vyvinutí, je rychlejší a snadněji se spouští. Na druhou stranu se opět oddaluje od reality. [1]

Out-of-process component test spustí celou mikroservisu tak, jak by se reálně spouštěla po nasazení a testuje její operace zvenčí. V mikroservisním prostředí by tedy test pravděpodobně spustil docker container s testovanou mikroservisou, všechny její infrastrukturní služby a mockoval by pouze externí mikroservisy, které aplikace využívá. Tento typ testování je mnohem blíže realitě. Na druhou stranu je vývoj takových testů výrazně složitější. [1]

Při testování mikroservisy zaměřující se na tvorbu rezervací, by component testy mohly testovat např. tyto operace:

- Vytvoř novou rezervaci.
- Načti existující rezervaci.
- Uprav existující rezervaci.

- Smaž existující rezervaci.

Jedná se zkrátka o sadu testů, které pokud doběhnou úspěšně, měly by vývojářskému týmu dodat velký pocit jistoty ohledně naplnění funkčních požadavků testované mikroservisy.

1.8.1 Technologie pro psaní component testů

Na tento typ testování se hodí použít dohromady všechny doposud zmíněné nástroje. Zejména TestContainers a WireMock. Je dobré podotknout, že konkrétní nastroje se zde už budou často lišit dle technologií, zvolených k vývoji aplikace. V Java Spring Boot by se např. dala použít anotace @SpringbootTest, díky které test spustí celou aplikaci a umožní na ní testovat různé operace.

1.9 End-to-End testy

End-to-end testy testují celou aplikaci dohromady. Nacházejí se na vrcholu dříve zmíněné testovací pyramidy a mělo by jich být v našem testovacím procesu tvořeno opravdu malé množství. Jejich vývoj je totiž velice náročný. Jejich spouštění vyžaduje nasazení všech jednotlivých mikroservis a jejich infrastrukturních závislostí. V praxi často nasazení nějaké mikroservisy selže a to dělá testy nespolehlivými. [1]

Dobrou strategií pro návrh end-to-end testů je navrhovat tzn. **user journey testy**. User journey test simuluje cestu skutečného uživatele v aplikaci. Místo toho, aby bylo vytvářeno několik různých end-to-end testů pro vytvoření, upravení a zrušení rezervace, je vhodnější vytvořit jeden end-to-end test, který postupně provede všechny tři operace zároveň. Tento způsob výrazně sníží množství testů, které je potřeba napsat a zkrátí celkovou dobu běhu. [1]

End-to-end testy však mají i několik pozitiv. Mezi ty patří např. tyto vlastnosti:

- Jako jediný způsob testování mohou naprosto reflektovat realitu.
- Jeden test může otestovat mnoho funkcionalit dohromady.
- Mohou sloužit jako dobrý podklad pro doložení funkcionality celého systému.

1.9.1 Technologie pro psaní end-to-end testů

1.9.1.1 Karate

Karate je open-source nástroj, který kombinuje API testování, mockování, performance testování a testování uživatelského prostředí. Syntaxe tohoto nástroje je poměrně jednoduchá a může ho proto snadno využívat i někdo s menším technickým vzděláním. [21]

1.9.1.2 Robot Framework

Robot framework je další open-source nástroj, který slouží k automatizovanému testování. Má jednoduchou syntaxi a lidsky snadno čitelná klíčová slova, díky čemuž je opět snadno využitelný i mezi méně technologicky vzdělanými jedinci. [22]

1.10 Load testy

Load testing je způsob testování, ve kterém se aplikace testuje pod větší zátěží, která by mohla nastat v reálném světě. Většinou se tento typ testování aplikuje v závěru vývoje, před tím, než je aplikace uvedena na trh. V agilním prostředí je však samozřejmě dobré provádět load testy průběžně během vývoje. Hlavním výstupem tohoto způsobu testování může být:

- Vyhodnocení výstupu aplikace.
- Rychlost zpracování a přenosu dat.
- Maximální počet souběžných uživatelů.
- Doba odezvy příkazu.

Load testování tedy dodá vývojářskému týmu sebevědomí ohledně zpřístupnění aplikace většímu počtu uživatelů. Může identifikovat pokulhávání či pomalé načítání systému pod větší zátěží, což jsou často vlastnosti, které mohou odradit budoucí zákazníky. [23]

S rostoucím počtem digitálních aplikací v lidských životech roste i očekávání kvality, a pokud aplikace v produkci spadne, může to mít velice drahé důsledky. Dle [24] se cena v IT odvětví za minutu, kdy je systém nedostupný, pohybuje okolo 5 600 dolarů.

1.10.1 Technologie pro load testing

1.10.1.1 Gatling

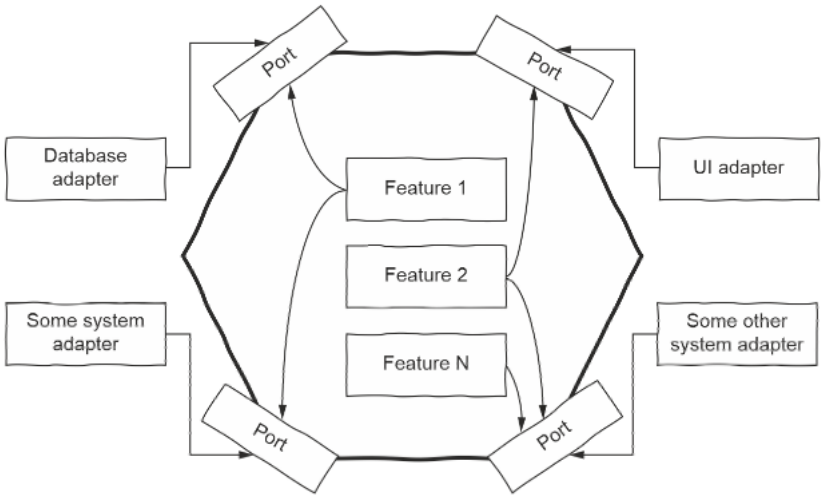
Gatling je load a performance testing nástroj. Skládá se z open-source verze a placené enterprise verze. Nástroj generuje detailní reporty a performance metriky, které zahrnují např. dobu odpovědi nebo počet chyb. Tyto reporty mohou pomoci rychle identifikovat problémy s výkonem. [25]

1.11 Návrh softwaru pro lepší testovatelnost

Softwarové systémy jsou někdy složitě testovatelné z důvodu jakým byli navrhnuty. V této sekci budou popsány některé myšlenky, které stojí za jednodušším automatizovaným testováním softwaru.

1.11.1 Oddělení infrastrukturního kódu od doménového kódu

doména je ta část kódu, ve které leží jádro softwaru. Tedy ta část, ve které se nachází veškerá businessová pravidla, logika, entity, služby apod. Infrastrukturní kód je tou častí, která se stará o externí závislosti. Tedy např. ta část kódu, která komunikuje s databází nebo s jinými mikroservisami. V praxi, pokud se doménový kód a infrastrukturní kód mixuje dohromady, stává se software mnohem složitěji testovatelným. K oddělení odpovědností může pomoci hexagonální architektura.



Obrázek 1.5 Hexagonální architektura [6]

Obrázek 1.5 zachycuje hexagonální architekturu. Uvnitř hexagonu se nachází aplikace a veškerá businessová logika. Tento kód se tedy týká bussinesové logiky a funkčních požadavků aplikace. V určitý moment bude však tato část kódu muset komunikovat s externím světem. K tomu využije na obrázku zachycené porty. Porty jsou rozhraní, která definují, co všechno umí konkrétní adaptér. Adaptéry jsou implementacemi daných portů, které už mají ke konkrétní infrastruktuře velice blízko a přímo s ní komunikují. Získaná data z infrastruktury poté zasílají zpět aplikaci ve formátu, který je definován portem. [6]

Jak může taková architektura zlepšit testovatelnost? Pokud doménové třídy závisí pouze na portech, je možné jednoduše testovat veškeré chování doménové logiky, bez komunikace s externími systémy díky mockování využívaných portů. Správnou funkcionalitu adaptérů pak otestují zvlášť integrační testy. [6]

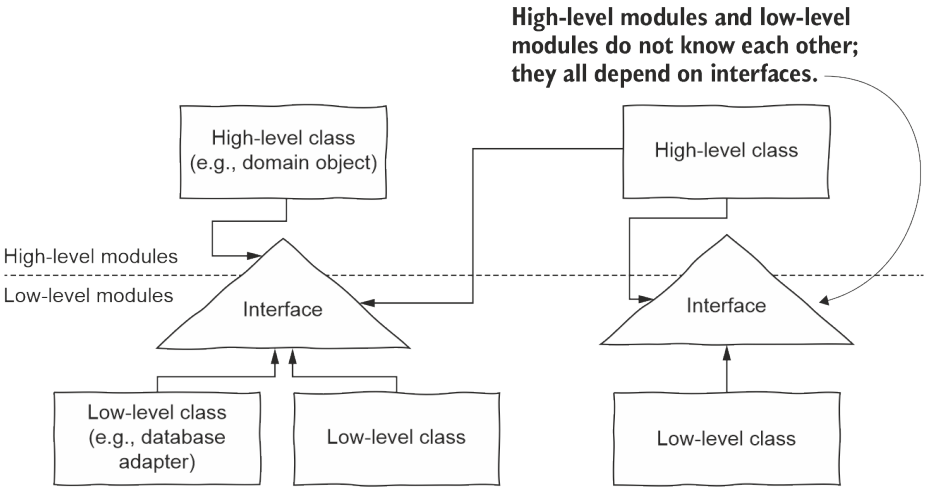
1.11.2 Dependency injection a ovladatelnost

Na úrovni jednotlivých tříd je důležité, aby byly jednotlivé třídy dobře kontrolovatelné (tedy aby šlo jednoduše kontrolovat, co testovaná třída dělá) a pozorovatelné (tedy aby šlo pozorovat, jak se testovaná třída chová a jaký je její výstup). Pro kontrolovatelnost musí být třída navržena tak, aby šly její závislosti na jiných třídách snadno nahradit mocky. Toho se dá dosáhnout tak, že si třída neinicializuje své závislosti sama, ale jsou jí předány (např. pomocí konstruktoru). To má za důsledek mnoho výhod:

- Umožní se tím mockovat závislosti během testování.
- Všechny závislosti jsou tak více explicitní.
- Je tím dosaženo lepšího oddělení starostí, protože se třída nemusí sama starat o inicializaci závislostí.
- Třída je snadněji rozšiřitelná.

V praxi se k dosažení vkládání závislostí využívá návrhový vzor zvaný dependency injection. Většina velkých vývojových frameworků tento návrhový vzor využívá. [6]

Dalším vzorem, který ulehčí testování je tzn. dependency inversion principle. Tento návrhový vzor říká, že třídy vysoké úrovně (např. doménové třídy) by neměly záviset na detailech tříd nízké úrovně (např. komunikace s databází). Měly by záviset na abstrakci těchto tříd nízké úrovně.



Obrázek 1.6 Dependency inversion principle [6]

Obrázek 1.6 zachycuje, jak vypadá tento návrhový vzor. Jednotlivé abstrakce představují rozhraní, na kterých závisí třídy vyšší úrovně. Třídy nižší úrovně pak tyto rozhraní implementují. Jak pomůže tento návrhový vzor při

testování? Mockování tříd, které implementují nějaké rozhraní je výrazně jednodušší. Rozhraní jsou většinou jednoduchá a přímočará a při testování tak není potřeba rozumět složité implementaci tříd. [6]

1.11.3 Pozorovatelné třídy

Aby se daly jednotlivé třídy snadno testovat, je potřeba, aby jejich chování bylo snadno pozorovatelné. Pokud třída dělá interní změny svého stavu, ale nenabízí cestu, jak tyto změny v testu kontrolovat, bude testování takové třídy výrazně složitější. Samotný kód automatizovaného testu bude pak také hůře čitelný a udržovatelný. Každá třída by tedy měla nabízet gettery a jednoduché metody pro kontrolu svých vnitřních stavů. [6]

1.12 Exloratory testing

Část testování, kterou stále stojí za to provádět manuálně a není zahrnuta v dříve zmíněné testovací pyramidě, je tzn. exploratory testing. Exploratory testing je způsob testování, ve kterém se tester snaží manuálně objevit veškeré nedostatky testovaného softwaru. Tester při tomto způsobu testování nepracuje podle předem zadaného scénáře, ale snaží se k objevení chyb uplatnit svou lidskou kreativitu a intuici. Objevené chyby a nedostatky může následně pokrýt automatizovanými testy, aby se v budoucnu nemohly znovu opakovat. [26]

1.13 Automatizované testování pomocí umělé inteligence

Umělá inteligence a strojové učení se začínají promítat do mnoha sektorů ekonomiky a měnit mnoho aspektů běžného lidského života. Protože automatizované testování softwaru v praxi často vyžaduje mnoho manuálních zásahů člověka, začala se objevovat snaha o využití umělé inteligence právě v oboru automatizovaného testování. [27]

V poslední době tak vzniklo mnoho výzkumů, zaměřující se na využití umělé inteligence v oblasti automatizovaného testování. Ukazuje se, že AI může dobře sloužit k těmto testovacím činnostem:

1. Generování testovacích případů.
2. Generování testovacích dat.
3. Tvorba testovací specifikace. [28]

v [29] autoři předpovídají, že automatizované testování pomocí AI se stane samostatnou části průmyslu a bude hrát v IT hlavní roli. Předpokládají, že nahradí roli dnešních testerů a QA inženýrů, kteří budou pouze vylepšovat a

monitorovat práci AI. Na základě výzkumu autoři dále předpokládají, že AI celkový proces testování velice zkvalitní.

Kapitola 2
Praktická část

2.1 Demo aplikace

V praktické části bakalářské práce byla kompletně otestována demo aplikace společnosti Stratox. Tato aplikace je postavena na mikroservisní architektuře a slouží k demonstraci výhod této architektury pro klienty společnosti. Aplikace simuluje nákup dopravních jízdenek pro autobusovou nebo vlakovou dopravu.

Celá aplikace se skládá z vícero mikroservis napsaných v Java Spring Boot, které spolu komunikují skrze systém pro výměnu zpráv Apache Kafka.

2.1.1 Jednotlivé mikroservisy

Testovány byly pouze některé z mikroservis, protože některé spadaly pod vývoj jiných studentů v rámci jejich bakalářských prací. Celou aktuální architekturu zachycuje obrázek 2.1. Níže jsou popsaný jednotlivé mikroservisy, které prošly testováním.

2.1.1.1 BL - component

Jak již z názvu vyplývá tato mikroservisa obsahuje veškerou business logiku zajišťující rezervaci jízdenky. Jednotlivé rezervace ukládá do PostGreSQL databáze. Skrze Apache Kafka komunikuje s API A MailSender mikroservisami.

2.1.1.2 API - component

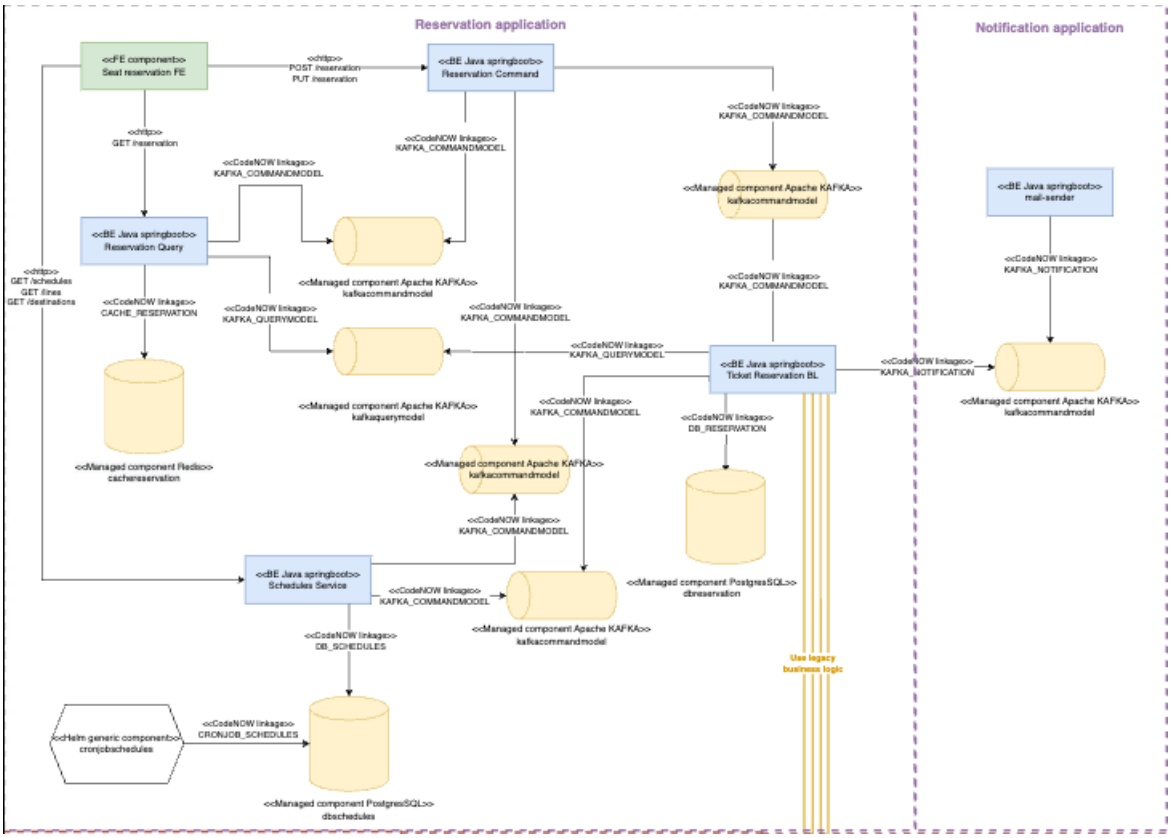
API mikroservisa, která vystavuje veškeré end-pointy týkající se rezervace jízdenek. Jednotlivé operace cashuje do Redis databáze a requesty skrze Kafku asynchronně přeposílá BL - componentě.

2.1.1.3 MailSender - component

Mikroservisa, která zasílá email uživateli při vytvoření či zrušení rezervace. Komunikuje s BL - componentou skrze Apache Kafka.

2.1.1.4 Sent reservation FE

Componenta s React.js aplikací, která tvoří UI celé aplikace. Uživatel zde může tvořit rezervace a spravovat již existující rezervace. Testování zde probíhalo pouze v rámci end-to-end testů.



Obrázek 2.1 Demo aplikace

2.2 Proces testování

Při testování bylo postupováno dle všech nastudovaných znalostí. Testování probíhalo pro každou mikroservisu co nejvíce izolovaně. Na každou mikroservisu byly aplikovány všechny popsané způsoby testování. V závěru byl vytvořen jeden společný end-to-end test pro nasazenou beta verzi aplikace a také jeden load test pro otestování chování aplikace pod větší zátěží.

2.2.1 Unit testy

Pro unit testy byly použity technologie popsané v teoretické části, tedy JUnit, Mockito a AssertJ. Každá mikroservisa byla pokryta unit testy alespoň na hranici 75 % line coverage.

2.2.1.1 Ukázkový unit test

Obrázek 2.2 Ukázka hlavičky unit testu

```
@ExtendWith(MockitoExtension.class)
public class ReservationServiceImplTest {
    @InjectMocks
    ReservationServiceImpl reservationService;

    @Mock
    TracingHelper tracingHelper;

    @Mock
    MailNotificationProducer mailNotificationProducer;

    @Mock
    NewReservationProducer newReservationProducer;

    @Mock
    CancelReservationProducer cancelReservationProducer;

    @Mock
    ReservationViewProducer reservationViewProducer;

    @Mock
    ReservationRepository reservationRepository;

    ...
```

Obrázek 2.2 zachycuje hlavičku běžného unit testu. Nad deklarací samotné třídy, která reprezentuje všechny unit testy pro konkrétní třídu produkčního kódu (v tomto případě třídy ReservationServiceImpl), je použita anotace @ExtendWith(MockitoExtension.class). Tato anotace nastaví pro tento unit test využití Mockito rozšíření, které umožní v této třídě používat mocky.

V těle třídy je dále deklarovaná proměnná ReservationServiceImpl s anotací @InjectMocks. S touto anotací se vždy deklaruje třída, která bude testována.

Anotace zařídí, že veškeré závislosti této třídy budou nahrazeny mocky.

Ostatní proměnné této třídy jsou deklarovány s anotací @Mock. Ta zaručí, že závislost na této třídě bude v testované třídě nahrazena jejím mockem.

Před popisem samotného testu je vhodné se podívat na produkční kód, jehož funkcionalitu test ověřuje.

Obrázek 2.3 Ukázka implementace

```
@Override
public void getLastNReservations(Integer numberOfReservations)
throws ExecutionException, InterruptedException {
    Pageable limit =
    PageRequest.of(0, numberOfReservations, Sort.Direction.DESC, "date");

    Page<ReservationEntity> reservationEntities =
        reservationRepository.findAll(limit);

    List<ReservationDTO> reservationDTOS = new ArrayList<>();
    for (ReservationEntity entity : reservationEntities) {
        reservationDTOS.add(reservationEntityToReservationDTO(entity));
    }
    reservationViewProducer
        .reservationViewResponse(new ReservationViewDTO(reservationDTOS));
}
```

Účelem této metody, znázorněné na obrázku 2.3, je provolat repozitář s rezervacemi a získat patřičné rezervace. Tyto rezervace následně konvertovat do patřičných objektů pro přenos dat a ty v jednom poli předat producentovi. Obrázek 2.4 znázorňuje, jak vypadá unit test pro výše popsanou metodu.

V ukázce kódu je možné vidět anotaci @Test, která se používá pro označení každého jednotlivého unit testu. V těle metody jsou následně připraveny dvě rezervace a entita Page, která je těmito rezervacemi naplněna. Následně je využit framework Mockito, pomocí kterého je nastavena entita Page jako hodnota, kterou má vracet metoda findAll na mockované entitě reservationRepository. Poté se přejde k provolání samotné metody, která je testována a tedy metodě getLastReservations na entitě reservationService. V závěru je ověřeno, že došlo k provolání správné metody se správnými argumenty.

2.2.2 Contract testy

Další kategorií testů byly Contract testy. K těm byla použita technologie Pact popsaná v teoretické části. Všechny mikroservisy v této architektuře spolu komunikují asynchronně pouze pomocí systému pro výměnu zpráv Apache Kafka.

Obrázek 2.4 Ukázka unit testu

```
@Test
public void getLastNReservationsTest()
throws ExecutionException, InterruptedException
{
    ReservationEntity reservationEntity1 =
        new ReservationEntity(UUID.randomUUID(), "David", "Hospodka",
        "david@email.cz", "seat1", "train1",
        new Date(), ReservationStatus.ACTIVE);

    ReservationEntity reservationEntity2 =
        new ReservationEntity(UUID.randomUUID(), "Marek", "Novak",
        "Marek@email.cz", "seat2", "train1",
        new Date(), ReservationStatus.ACTIVE);

    Page<ReservationEntity> page =
        new PageImpl<>(List.of(reservationEntity1, reservationEntity2));
    when(reservationRepository.findAll(any())).thenReturn(page);

    reservationService.getLastNReservations(2);

    ArgumentCaptor<ReservationViewDTO> reservationViewDTOArgument =
        ArgumentCaptor.forClass( ReservationViewDTO.class );

    verify(reservationViewProducer, times(1))
        .reservationViewResponse(reservationViewDTOArgument.capture());
    assertThat(reservationViewDTOArgument.getValue().getReservations().size())
        .isEqualTo(2);
}
```

V rámci tohoto typu testů bylo tedy potřeba správně ověřit dodržování asynchronního contractu mezi jednotlivými mikroservisami. Je vhodné podotknout, že aplikace je velice malého rozsahu, a tak zde dodržování contractů bylo zřejmé i bez automatizovaných testů. Tyto testy však mohou sloužit tvůrcům aplikace k demonstračním účelům, a také do budoucna, pokud se bude aplikace rozšiřovat.

2.2.2.1 Ukázkový contract test

Obrázek 2.5 ukazuje jak vypadá contract test na straně consumera.

V tomto testu jsou definovány veškeré atributy, které konzument očekává

Obrázek 2.5 Contract test na straně consumera

```
/**
* Defines the contract
* @param builder
* @return
*/
@Pact(consumer = "ReservationDTOConsumer")
MessagePact reservationPact(MessagePactBuilder builder) {
    PactDslJsonBody jsonBody = new PactDslJsonBody();
    return builder.expectsToReceive("Valid ReservationDTO")
        .withMetadata(Map.of(KEY_CONTENT_TYPE, JSON_CONTENT_TYPE))
        .withContent(LambdaDsl.newJsonBody(lambdaDslJsonBody -> {
            lambdaDslJsonBody.uuid("id");
            lambdaDslJsonBody.stringType("firstName", "David");
            lambdaDslJsonBody.stringType("lastName", "Hospodka");
            lambdaDslJsonBody.stringType("email", "david@email.cz");
            lambdaDslJsonBody.stringType("seatId", "seat1");
            lambdaDslJsonBody.stringType("trainId", "train1");
            lambdaDslJsonBody.stringType("status", "ACTIVE");
            lambdaDslJsonBody.datetime("date", "yyyy-MM-dd'T'HH:mm:ss:SSS");
            }).build())
        .toPact();
}
```

od dodávaného objektu ReservationDTO. U data je specifikován i jeho formát. Spuštění takového testu vygeneruje JSON contract, který byl následně vložen do repozitáře producera. Test na straně producenta zachycuje obrázek 2.6

Při spouštění tohoto testu se vezme objekt, který vytvoří pomocná metoda getCorrectReservationDTOObject a porovná se s objektem, který je vytvořen na základě JSON contractu. Pokud by nějaké pole objektu chybělo, nebo mělo špatný typ, test by neprošel.

Obrázek 2.6 Contract test na straně producera

```
@TestTemplate
@PactVerifyProvider("Valid ReservationDTO")
String verifySimpleMessageEvent() throws JsonProcessingException {
    Map<String, Object> metadata = Map.of(
            KEY_CONTENT_TYPE, JSON_CONTENT_TYPE
    );
    ReservationDTO reservationDTO  = getCorrectReservationDTOObject();
    return objectMapper.writeValueAsString(reservationDTO);
}

/**
* Correctly build the object you produce here
* @return
*/
ReservationDTO getCorrectReservationDTOObject() {
    return ReservationDTO.builder()
        .id(UUID.randomUUID())
        .firstName("John")
        .lastName("Smith")
        .seatId("seat1")
        .trainId("train1")
        .date(new Date())
        .email("John.Smith@email.com")
        .status(ReservationStatus.ACTIVE)
        .build();
}
```

Pokud by tento typ testů byl využíván v praxi na aplikaci, ve které se dynamicky vyvíjí API jednotlivých mikroservis, musely by týmy, které API využívají mít na paměti udržování těchto testů aktuálních. Vygenerované contracty zde byly napříč repozitáři nakopírovány manuálně. V praxi by se pro automatizaci tohoto procesu mohl využít nástroj Pact Flow popsaný v teoretické části.

2.3 Integrační testy

Integrační testy musely otestovat validní komunikaci s Apache Kafka, PostgreSQL databází a in-memory databází Redis. Pro otestování byly použita technologie TestContainers, která umožňuje v rámci testu spouštět skrze Docker externí systémy. Integrační test si tedy zpočátku spustil systém, který testoval

a následně otestoval komunikaci.

2.3.0.1 Přidaný consumer pro účely testování

Pro otestování komunikace s Apache Kafka bylo zapotřebí přidat consumery, kteří by zachycovaly odesílané zprávy. Díky tomu bylo možné otestovat, že odeslaná zpráva do Apache Kafka skutečně odešla.

Obrázek 2.7 Přidaný consumer pro testování producentů

```
@Getter
@Service
@Profile("test")
public class CancelReservationResponseConsumer {

    private CountDownLatch latch = new CountDownLatch(1);

    private ReservationDTO lastPayload;

    @KafkaListener
    (topics = "${spring.kafka.reservation.topic.cancel-reservation-response}",
    containerFactory="cancelReservationResponseKafkaListenerContainerFactory")
    public void mailReservationListener
    (@Payload ReservationDTO reservationDTO, Acknowledgment ack) {
        latch.countDown();
        lastPayload = reservationDTO;
        ack.acknowledge();
    }

    public void reset() {
        latch = new CountDownLatch(1);
        lastPayload = null;
    }
}
```

Obrázek 2.7 zachycuje, jak vypadal přidaný consumer. Je vhodné si povšimnout anotace @Profile(”test”), která zaručí, že tato třída je použita pouze při testování a nebude mít žádný vliv na produkční kód. Tato třída tedy přijímá objekty, které do Kafky poslala ta stejná mikroservisa. V důsledku toho je možné v automatizovaném testu využít tohoto consumera a zkontrolovat odeslaný obsah. Pro testování komunikace s Kafkou je tak možné použít pouze jednu mikroservisu a testovací proces tak může zůstat izolovaný. Zajímavá je v této třídě proměnná latch typu CountDownLatch, která slouží ke kontrole

příjmu nějaké zprávy. Proměnná lastPayload slouží k uložení přijaté zprávy. Obě tyto proměnné jsou viditelné pomocí anotace @Getter a můžeme k nim tedy v testu zvenčí přistupovat. Metoda mailReservationListener je zavolána při přijmutí zprávy z Apache Kafka a nejprve zavolá metodu countDown na proměnné latch a poté do proměnné lastPayload uloží přijatou zprávu. V závěru informuje Kafku o přijmutí zprávy pomocí metody acknowledge na objektu Acknowledgment.

2.3.0.2 Ukázkový integrační test

Obrázek 2.8 zachycuje hlavičku integračního testu, ve kterém je výše zmíněný přidaný konzument využit. Tento test dále využívá anotace @SpringBootTest, @Profile a @TestContainers, které se do ukázky z důvodu velikosti nevešly. V tomto testu je zajímavé použití anotace @Container a proměnné Kafka, ve které si pomocí technologie TestContainers připravíme Docker kontejner s Apache Kafka. Následně v metodě overrideProperties konfigurujeme spouštěnou aplikaci, aby pro komunikaci s Apache Kafka použila adresu tohoto kontejneru. V testu dále deklarujeme několik proměnných pro testování komunikace s Apache Kafka.

Samotný test zachycuje obrázek 2.9. V tomto testu je otestovaná funkcionalitu producera CancelReservationProducer za použití přidaného consumera cancelReservationResponseConsumer. V testu se nejprve vytvoří objekt s rezervací, který se následně odešle do Kafky skrze testovaného producera. Následně test využije dříve zmíněnou proměnnou latch na objektu cancelReservationConsumer a počká pomocí ní na přijetí odeslané zprávy. Maximální doba čekání je 10 vteřin. V závěru test ověří, že přijatý objekt má všechny hodnoty, které byly do Kafky odeslány.

Obrázek 2.8 Hlavička integračního testu pro Apache Kafka

```
public class CancelReservationITest {

    @Autowired
    @Qualifier("reservation")
    private KafkaTemplate<String, UUID> kafkaTemplate;

    @Autowired
    private CancelReservationConsumer cancelReservationConsumer;

    @Autowired
    private CancelReservationProducer cancelReservationProducer;

    @Autowired
    private CancelReservationResponseConsumer cancelReservationResponseConsumer;

    @Container
    private static final KafkaContainer KAFKA =
            new KafkaContainer(DockerImageName
                .parse("confluentinc/cp-kafka:6.2.1"));

    /**
     * Set up test to use the newly created kafka container.
     * @param registry
     */
    @DynamicPropertySource
    static void overrideProperties(final DynamicPropertyRegistry registry) {
        registry
            .add("spring.kafka.reservation.bootstrap-servers",
                KAFKA::getBootstrapServers);
        registry
            .add("spring.kafka.notification.bootstrap-servers",
                KAFKA::getBootstrapServers);
    }

    ...

}
```

Obrázek 2.9 Test producera pro Apache Kafka

```
@Test
    public void testProducer() throws ExecutionException, InterruptedException {
        //Wait a bit for kafka to setup properly.
        Thread.sleep(1000);
        UUID id = UUID.randomUUID();
        Date date = new Date();
        ReservationDTO reservationDTO =
            new ReservationDTO(id, "David",
            "Hospodka", "David@email.cz", "Seat1", "Train1",
            date, ReservationStatus.ACTIVE);

        cancelReservationProducer.cancelReservationResponse(reservationDTO);
        boolean consumed = cancelReservationResponseConsumer
                           .getLatch().await(10, TimeUnit.SECONDS);
        assertThat(consumed).isTrue();
        assertThat(cancelReservationResponseConsumer
            .getLastPayload().getId()).isEqualTo(id);
        assertThat(cancelReservationResponseConsumer
            .getLastPayload().getFirstName()).isEqualTo("David");
        assertThat(cancelReservationResponseConsumer
            .getLastPayload().getLastName()).isEqualTo("Hospodka");
        assertThat(cancelReservationResponseConsumer
            .getLastPayload().getEmail()).isEqualTo("David@email.cz");
        assertThat(cancelReservationResponseConsumer
            .getLastPayload().getSeatId()).isEqualTo("Seat1");
        assertThat(cancelReservationResponseConsumer
            .getLastPayload().getTrainId()).isEqualTo("Train1");
        assertThat(cancelReservationResponseConsumer
            .getLastPayload().getDate()).isEqualTo(date);
        assertThat(cancelReservationResponseConsumer
            .getLastPayload().getStatus()).isEqualTo(ReservationStatus.ACTIVE);
    }
```

2.3.1 Component testy

V component testech byly testovány klíčové operace každé mikroservisy. Jednalo se tedy o operace typu vytvoř rezervaci, načti rezervaci a zruš rezervaci.

2.3.1.1 Ukázkový component test

Tyto testy se strukturou velice podobají předchozím integračním testům. Pomocí anotace @SpringBootTest spustí celou aplikaci a pomocí technologie TestContainers spustí externí závislosti konkrétní mikroservisy.

Obrázek 2.10 ukázka component testu

```
@Test
public void cancelReservationAcceptanceTest() throws InterruptedException {

    ...

    kafkaTemplate.send("cancel-reservation", id);

    boolean consumed = cancelReservationConsumer
        .getLatch().await(10, TimeUnit.SECONDS);
    assertThat(consumed).isTrue();

    Thread.sleep(100);
    assertThat(reservationRepository
        .findById(id).get().getStatus()).isEqualTo(ReservationStatus.CANCELED);

    boolean responseConsumed = cancelReservationResponseConsumer
        .getLatch().await(10, TimeUnit.SECONDS);
    assertThat(responseConsumed).isTrue();

    ...

    boolean mailNotificationConsumed = mailNotificationConsumer
        .getLatch().await(10, TimeUnit.SECONDS);
    assertThat(mailNotificationConsumed).isTrue();
}
```

Obrázek 2.10 zachycuje component test pro BL - componentu, který testuje zrušení konkrétní rezervace. Některé části kódu jsou pro přehlednost z ukázky vyňaty. Tato componenta vystavuje své API pouze skrze Apache Kafka, proto je pro otestování operace nutné skrze objekt kafkaTemplate odeslat do Kafky ID rušené rezervace. Test následně ověří všechny důležité aspekty operace, tedy že zpráva z Kafky byla přijata, že v databázi došlo ke zrušení rezervace, že byla do Kafky odeslána odpověď a také, že byla do Kafky odeslána zpráva pro odeslání informačního emailu. Pomocí poměrné jednoduchého kódu tak bylo otestováno mnoho funkcionalit.

2.3.2 End-to-end testy

V rámci end-to-end testování byl vyvinut jeden velký end-to-end test, který registruje nového uživatele, provede přihlášení, vytvoří novou rezervaci, rezervaci si zobrazí a zruší ji. Test se tedy snaží odpovídat tzn. user-journey popsané v teoretické části této práce. K testování byl využit testovací framework Karate.

2.3.2.1 Ukázkový test

Test kontroloval mnoho funkcionalit, a tak se skládal z více různých feature souborů. To je způsob, jakým se v Karate frameworku rozdělují jednotlivé testy.

Obrázek 2.11 ukázka end-to-end testu

```
@LoginUser
@parallel=false
Feature: Login user
  Background:
    * def base_url = Java.type('karate.ConfigLoader').getRegisterUrl()

  Scenario:
    * def register =
        call read('classpath:reservations/common/RegisterUser.feature')
    * driver base_url
    * waitForText('h1', 'Sign in to your account')
    * input("//input[@name='username']", register.username)
    * delay(200)
    * input("//input[@name='password']", register.password)
    * delay(200)
    * mouse('.pf-m-primary').click()
```

Obrázek 2.11 zachycuje část end-to-end testu, která přihlašuje registrovaného uživatele. Test nejprve provolá jinou část testu, která registruje nového uživatele. Tato část testu vrátí objekt register, který obsahuje informace o registrovém uživateli, tedy jméno a heslo. Následně vyplní těmito údaji příslušná pole a přihlásí uživatele kliknutím na tlačítko. Za povšimnutí stojí syntaxe technologie Karate, která je lidsky velice snadno čitelná.

2.3.3 Load testy

V rámci load testů byly otestovány jednotlivé REST endpointy API componenty. Performance test zkoušel vytvořit mnoho rezervací v jeden moment a

následně je zrušit. K testování byly použita technologie Karate a technologie Gatling. Obě tyto technologie jsou více popsané v teoretické části.

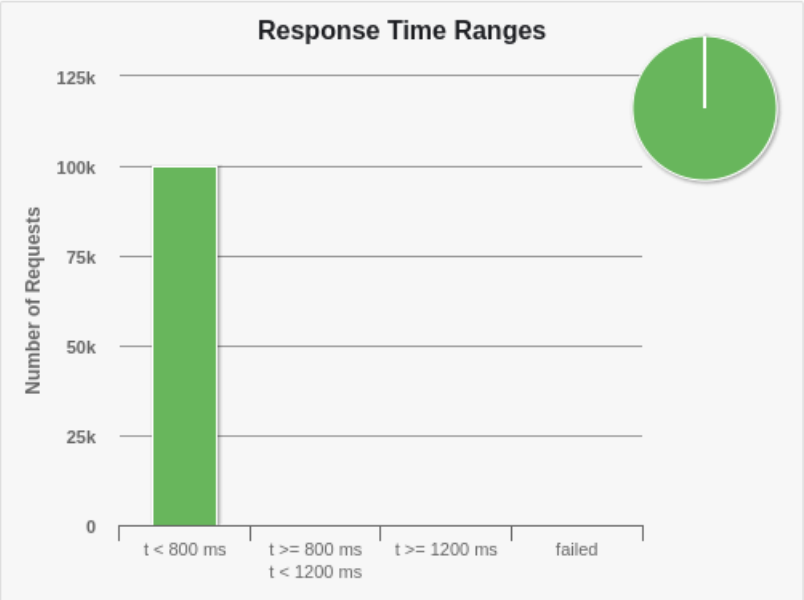
Obrázek 2.12 ukázka load testu

```
class PerformanceTest extends Simulation {
  val protocol = karateProtocol()

  val create = scenario("Create reservation")
    .exec(karateFeature("classpath:performance/CreateAndDeleteReservation.feature"))

  setUp(
    create.inject(
      atOnceUsers(100000)
    ).protocols(protocol)
  )
}
```

Obr. 2.12 zachycuje vytvořený load test. Test využívá feature soubor vytvořený pomocí technologie Karate, který zašle request pro vytvoření a následně zrušení rezervace. Pomocí technologie Gatling se provede tato operace stotisíckrát najednou a tím je simulováno mnoho uživatelů. Technologie Gatling následně vytvořila report, jehož část zachycuje obrázek 2.13.



Obrázek 2.13 Ukázka reportu s výsledky load testu API componenty

Na obrázku 2.13 je zobrazeno jak dlouho trvalo API componentě odpovědět na zaslané requesty, kterých bylo celkem 100 000. Z výsledků je snadno pozorovatelné, že na všechny requesty odpověděla API componenta téměř okamžitě. To je důsledek dobře zvolené architektury, kdy na požadavek klienta odpovídá API componenta velice rychle a logiku řeší až následně BL componenta. Další důvodem je také jednoduchost testovaných operací. Jedná se o základní operace s entitami a ty nezaberou mnoho času. Přesto byla během tohoto testování objevena jedna chyba, které je popsaná níže ve výčtu chyb.

2.3.4 Nalezené chyby

1. U API componenty neprobíhala validace vstupu u REST endopointů.
2. U žádné mikroservisy neprobíhala validace vstupu u příchozích objektů z Apache Kafka.
3. Předmět informačního emailu po zrušení rezervace je ”nová rezervace”.
4. Aplikace vrací kód 204 místo 404 při rušení neexistující rezervace.
5. BL - componenta přijímá objekty z Apache Kafka sekvenčně a to má při simulaci mnoha uživatelů za důsledek velice pomalé zpracování.

Kapitola 3
Závěr

Cílem práce bylo nastudovat problematiku automatizovaného testování v mikroservisním prostředí. Dále také porovnat jaké jsou rozdíly oproti testování monolitické architektury. Celá problematika byla studována z odborné literatury a vědeckých článků. Následně byla nastudovaná teorie popsána v teoretické části práce. Pozornost byla věnována především jednotlivým způsobům testování mikroservisní architektury a uvedená teorie byla doplňována přístupy k problematice různých technologických společností. Vždy byly také uvedeny technologie a nástroje, které jsou vhodné pro konkrétní typ testování. Rozdíly oproti testování monolitické architektury byly popsány ve speciální kapitole. Cílem praktické části bylo otestovat demo aplikaci společnosti Stratox, ke které chyběly téměř všechny automatizované testy. V praktické části tak byly vyvinuty všechny typy automatizovaných testů, které byly uvedeny v teoretické části a nalezené chyby byly reportovány tvůrcům aplikace. V závěru bylo dodáno autorovo vlastní doporučení pro testování mikroservisní architektury.

3.1 Výsledek testování demo aplikace

Výčet všech nalezených chyb je popsaný v závěru praktické části této práce. Je důležité podotknout, že aplikace byla vyvinuta kvalitně a tak nalezených chyb nebylo mnoho. Kromě chyb ohledně validace vstupu nebyla žádná chyba kritická a jednalo se spíše o detaily. Je důležité si uvědomit, že aplikace je zatím poměrně malého rozsahu, a proto se chybám dalo snadněji předejít. Automatizované testy by měly do budoucna zařídit, aby žádná z již existujících funkcionalit nebyla někým při vývoji rozbita. Bylo by zároveň vhodné, kdyby se psaní testů stalo součástí práce všech budoucích vývojářů této aplikace a s novou funkcionalitou přibývaly i nové testy.

3.2 Autorovo doporučení pro automatizované testování mikroservisní architektury

V závěru bych rád dodal mé vlastní doporučení pro testování mikroservisní architektury. Osobně si myslím, že všechny kategorie testů jsou důležité a žádná by se neměla vynechat. V reálném světě však často nejsou prostředky a čas na důkladné testování, a tak si dokáži představit následující přizpůsobení testovacího procesu.

Unit testy by dle mého názoru měly být běžnou součástí práce každého vývojáře, který by ke svému kódu měl vždy dodat sadu unit testů. Zejména ty části kódu, které obsahují business logiku a složitější operace by měly být pokryty mnoha unit testy. Na spíše infrastrukturní kód je možné použít pouze integrační testování.

Contract testování považuji za nutné až u aplikací většího rozsahu. U menších aplikací s pár mikroservisami se dodržování jednotlivých contractů dá praktikovat pouze na základě vzájemné komunikace či kvalitně vedené dokumentace. U větších aplikací, kde na aplikaci spolupracuje několik nezávislých týmu, je však dle mého názoru tento typ testování nezbytný.

Integrační testování je v mikroservisním prostředí nutné a mělo by se vždy automatizovaně testovat, že aplikace dokáže správně komunikovat s ostatními mikroservisami či externími systémy jako je databáze apod.

Component testování je myslím také nutné a každá klíčová operace, kterou testovaná mikroservisa implementuje, by měla být otestována component testem. Do budoucího vývoje dávají tyto testy záruku, že nebyla rozbita žádná existující funkcionalita. Je důležité nezapomenout na izolovanost těchto testů a všechny externí závislosti nahradit mocky či speciálními nástroji.

Množství end-to-end testů bych minimalizoval a snažil se vyvinout pouze malé množství testů, které garantují funkčnost celého systému dohromady. Uživatelské prostředí bych nicméně automatizovaně testoval pomocí nástrojů určených k end-to-end testování izolovaně.

Příloha A

Report s chybami

A.1 API - COMPONENT

1. Neprobíhá validace vstupu u žádného requestu.
   - Jak reprodukovat: např. poslat request pro vytvoření nové rezervace. Jednotlivé atributy rezervace můžeme nechat prázdné.
   - Očekávaný výsledek: Atributy by se měly validovat a měli bychom obdržet status code 4XX.
   - Aktuální výsledek: Rezervace projde a dostaneme status code 200
2. Při pokusu o smazání neexistující rezervace je vráceno 204 místo 404.
   - Jak reprodukovat: pošlete request pro smazání rezervace, které neexistuje.
   - Očekávaný výsledek: status code 404.
   - Aktuální výsledek: status code 204.

A.2 BL - COMPONENT

1. Subject mailu po zrušení rezervace je: “New Reservation”.
   - Jak reprodukovat: Zrušte nějakou existující rezervaci.
   - Očekávaný výsledek: MailNotification objekt poslaný do Kafky má subject “Canceled Reservation”.
   - Aktuální výsledek: MailNotification objekt poslaný do Kafky má subject “New Reservation”.

2. Nevalidují se příchozí zprávy z kafky.
   - Jak reprodukovat: poslat do Kafky jakýkoliv objekt, který bl componenta očekává.
   - Očekávaný výsledek: Atributy objektů, které přijdou do komponenty z kafky budou validovány.
   - Skutečný výsledek: Atributy validovány nejsou.
3. Zprávy z Kafky jsou přijímány sekvenčně.
   - Jak reprodukovat: poslat do Kafky více zpráv najednou.
   - Očekávaný výsledek: Zprávy z Kafky budou přijímány paralelně.
   - Skutečný výsledek: Zprávy jsou zpracovávány pomalu sekvenčně.

A.3 Mail Sender - COMPONENT

1. Nevaliduje se MailControllerRequestDTO objekt při post request o zaslání mailu.
   - Jak reprodukovat: poslat post request s prázdnými atributy.
   - Očekávaný výsledek: status code 4XX.
   - Aktuální výsledek: status code 200.
2. Nevalidují se příchozí zprávy z Kafky.
   - Jak reprodukovat: poslat do kafky jakýkoliv objekt, který bl componenta očekává.
   - Očekávaný výsledek: Atributy objektů, které přijdou do komponenty z kafky budou validovány.
   - Skutečný výsledek: Atributy validovány nejsou.

Bibliografie

1. CHRIS, Richardson. *Microservices Patterns*. First. Shelter Island: MANNING, 2018. ISBN 978-1-61729-454-9.
2. BUENO, Soto; ANDY, Gumbrecht; JASON, Porter. *Testing Java Microservices*. First. Shelter Island: MANNING, 2018. ISBN 978-1-61729-289-7.
3. *Kompletní průvodce automatizací testování softwaru - zaptest* [online]. ZAPTEST, 2024 [cit. 2024-01-23]. Dostupné z: `https://www.zaptest.com/cs/kompletni-pruvodce-automatizaci-testovani-softwaru`.
4. *Test automation statistics - research aimultiple* [online]. RESEARCH AIMULTIPLE, 2024 [cit. 2024-01-03]. Dostupné z: `https://research.aimultiple.com/test-automation-statistics/`.
5. STRAUBINGER, P.; FRASER, G. A Survey on What Developers Think About Testing. In: *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, s. 80–90. Dostupné z DOI: `10.1109/ISSRE59848.2023.00075`.
6. ANICHE, Maurício. *Effective software testing*. First. Shelter Island: MANNING, 2022. ISBN 978-1-63343-993-1.
7. KUMAR, Divya; MISHRA, K.K. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science*. 2016, roč. 79, s. 8–15. ISSN 1877-0509. Dostupné z DOI: `https://doi.org/10.1016/j.procs.2016.03.003`. Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.
8. TITUS, Winters; TOM, Manshreck; WRIGHT, Hyrum. *Software engineering at Google*. First. O'REILLY, 2020. ISBN 978-1-492-08279-8.
9. *Testing Strategies For Microservices* [online]. semaphoreci, 2022 [cit. 2024-04-25]. Dostupné z: `https://semaphoreci.com/blog/test-microservices`.

10. *Testing Pyramid and Testing Ice Cream Cone — What is the difference?* [online]. semaphoreci, 2023 [cit. 2024-04-25]. Dostupné z: https://k-hartanto.medium.com/testing-pyramid-and-testing-ice-cream-cone-what-is-the-difference-6ddde3876c20.
11. *Testing of Microservices*. ATSPOTIFY, 2018. Dostupné také z: //engineering.atspotify.com/2018/01/testing-of-microservices/.
12. *Testing Microservices — Strategies and Processes for Enterprises* [online]. xenonstack, 2024 [cit. 2024-04-17]. Dostupné z: https://www.xenonstack.com/blog/microservices-testing.
13. *Microservices Testing: Challenges, Strategies, and Tips for Success* [online]. codefresh, 2024 [cit. 2024-04-17]. Dostupné z: https://codefresh.io/learn/microservices/microservices-testing-challanges-strategies-and-tips-for-success/.
14. *Reinventing the QA process* [online]. picnic, 2024 [cit. 2024-04-19]. Dostupné z: https://blog.picnic.nl/reinventing-the-qa-process-25854fee51f3.
15. *Testing Microservices - Contract Tests* [online]. SOFTWAREMILL, 2023 [cit. 2024-03-31]. Dostupné z: https://softwaremill.com/testing-microservices-contract-tests/.
16. *Introduction* [online]. DOCS PACT, 2022 [cit. 2024-03-31]. Dostupné z: https://docs.pact.io/.
17. *Contract testing for teams* [online]. Pact Flow, 2024 [cit. 2024-03-31]. Dostupné z: https://pactflow.io/features/.
18. *Unit tests with real dependencies* [online]. TestContainers, 2024 [cit. 2024-04-07]. Dostupné z: https://testcontainers.com/.
19. *Overview* [online]. WireMock, 2024 [cit. 2024-04-07]. Dostupné z: https://wiremock.org/docs/overview/.
20. *Testing Strategies in a Microservice Architecture* [online]. martinFowler, 2014 [cit. 2024-04-18]. Dostupné z: https://martinfowler.com/articles/microservice-testing/.
21. *Karate* [online]. karatelabs, 2024 [cit. 2024-04-18]. Dostupné z: https://karatelabs.github.io/karate/.
22. *Robot Framework* [online]. robotframework, 2024 [cit. 2024-04-18]. Dostupné z: https://robotframework.org/.
23. *Load Testing* [online]. loadninja, 2022 [cit. 2024-04-19]. Dostupné z: https://loadninja.com/load-testing/.
24. *Average Cost of Downtime per Industry* [online]. pingdom, 2023 [cit. 2024-04-19]. Dostupné z: https://www.pingdom.com/outages/average-cost-of-downtime-per-industry/.

25. *Comprehensive Reporting* [online]. gatling, 2024 [cit. 2024-04-19]. Dostupné z: https://gatling.io/.

26. *A Strategy to Testing Microservices* [online]. semaphoreci, 2022 [cit. 2024-04-25]. Dostupné z: https://konghq.com/blog/learning-center/microservices-testing-guide.

27. KING, Tariq M.; ARBON, Jason; SANTIAGO, Dionny; ADAMO, David; CHIN, Wendy; SHANMUGAM, Ram. AI for Testing Today and Tomorrow: Industry Perspectives. In: *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. 2019, s. 81–88. Dostupné z DOI: 10.1109/AITest.2019.000-3.

28. KHALIQ, Zubair; FAROOQ, Sheikh Umar; KHAN, Dawood Ashraf. Artificial Intelligence in Software Testing : Impact, Problems, Challenges and Prospect. *CoRR*. 2022, roč. abs/2201.05371. Dostupné z arXiv: 2201.05371.

29. HOURANI, Hussam; HAMMAD, Ahmad; LAFI, Mohammad. The Impact of Artificial Intelligence on Software Testing. In: *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. 2019, s. 565–570. Dostupné z DOI: 10.1109/JEEIT.2019.8717439.

Obsah příloh

- `readme.txt` ..... stručný popis obsahu média
- `src`
  - `impl` ..... zdrojové kódy jednotlivých mikroservis s testy
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF