Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Cloudová aplikace pro generování PDF z HTML s využitím AWS

Daniel Bechný

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Daniel Bechný

Informatika

Název práce

Cloudová aplikace pro generování PDF z HTML s využitím AWS

Název anglicky

Cloud application for generating PDFs from HTML using AWS

Cíle práce

Hlavním cílem práce je vytvoření cloudové aplikace, určené k vygenerování PDF souboru z vložené URL, analýza a porovnání různých řešení této problematiky a volba nejvhodnějšího.

Metodika

První variantou je aplikace vytvořená pomocí programovacího jazyka Kotlin a frameworku SpringBoot – provozována na AWS s využitím tradičních služeb. Další řešení spočívá ve využití serverless architektury pro danou aplikaci.

Tyto dvě různá řešení budou po otestování mezi sebou porovnány a z analýzy výsledků vzejde doporučení k implementaci vybraného řešení.

Tato metodika diplomové práce poskytuje rámcovou strukturu a návod pro provedení výzkumu a implementace cloudové aplikace, která využívá AWS služeb k generování PDF souborů z HTML obsahu. V rámci práce se provádí analýza požadavků, navrhuje se architektura aplikace. V závěru jsou shrnuty dosažené výsledky a navrhována možná rozšíření a vylepšení aplikace

Doporučený rozsah práce

60-80 stran

Klíčová slova

AWS, Kotlin, NodeJS, Serverless, SpringBoot, Softwarová architektura

---

Doporučené zdroje informací

Cloud Amazon Web Services – https://docs.aws.amazon.com
Dokumentace frameworku NodeJs – https://nodejs.org
Dokumentace frameworku Spring – https://docs.spring.io
Dokumentace jazyka Kotlin – https://kotlinlang.org

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 4. 9. 2023

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 3. 11. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 27. 03. 2024

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Cloudová aplikace pro generování PDF z HTML s využitím AWS" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 27. 3. 2024 _____

Poděkování

Rád(a) bych touto cestou poděkoval(a) panu doktoru Brožkovi, za vedení diplomové práce, své rodině, přítelkyni a kamarádům za podporu

Cloudová aplikace pro generování PDF z HTML s využitím AWS

Abstrakt

Tato práce se zabývá problematikou implementace aplikace pro tvorbu PDF z HTML obsahu. V této práci jsou představeny 2 rozdílné způsoby implementace řešení. První varianta řešení nabízí klasický model, kdy se jedná o Java aplikaci, na technologii SpringBoot, která je spuštěna tzv. on-premise.

Ve druhé variantě se pak jedná o totožnou aplikaci, která je postavena na technologii s názvem AWS Lambda, tudíž jako serverless. Jako běhové prostředí těchto lambda funkcí je zvoleno nodeJs.

Po představení těchto variant je provedeno srovnání daných řešení, diskuse kladů a záporů a možných budoucích problémech těchto řešení. Je odhadnuta cena jednotlivých řešení a je vybráno doporučení pro implementaci.

Klíčová slova: PDF, HTML, AWS, Cloud, Serverless, Lambda, Kotlin, NodeJs, IaaC, Softwarový vývoj

Cloud application to generate PDF from HTML using AWS

Abstract

This thesis considers implementation of an application that generates PDF from provided HTML content which is usually passed as a link to a website. In this thesis are being 2 different variants of a solution introduced. The first solution is a classical model which refers to a Java application that is built upon a SpringBoot framework and is running as an on-premise.

The second variant of a solution offers the same application but built with AWS cloud, especially AWS Lambda functions. This implementation is serverless. These lambdas are using nodeJS runtime.

Keywords: PDF, HTML, AWS, Cloud, Serverless, Lambda, Kotlin, NodeJs, IaaC, Software development

Obsah

1 Úvod

V době, kdy jsem působil jako softwarový inženýr pro technologickou společnost, která se specializuje na vývoj technologií pro školy, univerzity a obecně oblast vzdělávání se objevil požadavek na funkcionalitu generování PDF. V tomto případě se jednalo o generování PDF z webové aplikace na odhalování plagiátorství, tudíž z tzv. reportu o plagiátorství, který měl podobu HTML stránky. Z určitých důvodů nebylo možno v tomto případě využít zabudovanou funkcionalitu webových prohlížečů, která umožňuje webovou stránku exportovat přímo u uživatele. Požadavek na export do PDF formátu vznikl zejména kvůli možnosti ukládání, archivování a tisknutí těchto reportů o plagiátorství příslušnými univerzitami a institucemi.

Tato diplomová práce představuje aplikované řešení tohoto požadavku, včetně zkoumání různých variant implementace a zpracování. První přistupuje k tradičnějšímu řešení prostřednictvím on-premise aplikace s využitím Kotlinu a Spring Bootu. Tato aplikace je pak provozována na serverové instanci od AWS (EC2[1]). Ve skutečnosti zejména proto, že se v praxi jednalo o rozšíření existují webové aplikace na odhalování plagiátorství, která byla tímto způsobem vystavěna téměř celá. Druhá varianta využívá cloudové služby AWS[2] Lambda spolu s NodeJS, aby poskytla škálovatelnou a efektivní alternativu.

Cílem této práce je poskytnout hluboký vhled do obou přístupů, s přihlédnutím k jejich výhodám, nevýhodám a praktickým implikacím, které byly klíčové při výběru finálního řešení. V následujících kapitolách se budu věnovat technickým aspektům obou variant, jejich výkonnostním charakteristikám, a diskutovat o tom, jak každé z řešení ovlivňuje celkovou udržitelnost a rozšiřitelnost v kontextu vzdělávacích aplikací.

[1] Elastic Compute Cloud – Serverová instance instance na AWS
[2] Amazon Web Services

2 Cíle a metodika

2.1 Cíl práce

Cílem této diplomové práce je navrhnout a implementovat cloudovou aplikaci pro efektivní generování PDF dokumentů z předaných URL na straně serveru. Budou představeny 2 způsoby řešení, které budou mezi sebou porovnána, otestována, implementována a následně bude vybráno jedno, které bude doporučeno pro implementaci.

Před rozhodnutím proběhne důsledná komparace jednotlivých kladů a záporů, a s přihlédnutím k business potřebám dané implementace.

Specifické cíle:

- Prozkoumat a analyzovat existující řešení pro generování PDF z HTML na straně serveru.
- Návrh on-premise aplikace na EC2
- Návrh serverless aplikace s využitím AWS lambda funkce.
- Implementace obou prototypů a demonstrace funkčnosti.
- Porovnání pozitiv a negativ obou řešení, včetně cenového odhadu obou řešení.
- Zhodnocení bezpečnosti, udržitelnosti, škálovatelnosti a rozšiřitelnosti navrhovaných řešení.
- Doporučení pro implementaci.

2.2 Metodika

Metodika této práce zahrnuje kombinaci teoretického a praktického výzkumu, který byl proveden při vývoji softwaru, o němž pojednává tato práce. V první fázi se jedná o seznámení se s požadavky a nároky na tuto aplikaci. Provedení rešerše dostupných řešení pro generování PDF obsahu a jejich porovnání.

Vznikly 2 prototypy aplikace, jeden implementován jako aplikace nad JVM[3] s použitím programovacího jazyka Kotlin a technologie SpringBoot. Druhý prototyp implementován jako nodeJS aplikace využívající AWS lambdu. Obě aplikace budou

[3] Java Virtual Machine

nasazeny do cloudového prostředí AWS, on-premise varianta na EC2 instanci a serverless jako AWS lambda. Následně je provedena komparace těchto řešení, analýza možností škálování a nákladů.

Teoretický základ práce je vytvořen na základě přehledu literatury a analýzy služeb poskytujících použité technologie.

Praktická část se pak zaměřuje na implementaci, nasazení a testování těchto aplikací s využitím poznatků nabytých v teoretické části. Tento proces zahrnuje kromě podrobného popisu implementace samotné logiky taktéž sestavení a nasazení takto vystavěných aplikací. Praktická část se taktéž zabývá řešením škálovatelnosti aplikace a paralelizace v případě souběžných požadavků na vygenerování PDF.

3 Teoretická východiska

3.1 Cloud computing

3.1.1 Definice cloud computingu

Cloud computing je obecný termín pro poskytování hostovaných výpočetních služeb a IT zdrojů přes internet. Uživatelé mohou pronajímat služby, jako je výpočetní výkon, úložiště a databáze od poskytovatele cloudu, čímž odpadá nutnost nákupu, provozu a údržby fyzických datových center a serverů. (Yasar, a další, 2023)

Cloud computing je možný díky virtualizaci. Virtualizace umožňuje vytvořit simulovaný, digitální "virtuální" počítač, který se chová, jako by to byl fyzický počítač s vlastním hardwarem. Technický termín pro takový počítač je virtuální stroj. Při správné implementaci jsou virtuální počítače na stejném hostitelském počítači navzájem odděleny, takže spolu vůbec neinteragují a soubory a aplikace z jednoho virtuálního počítače nejsou pro ostatní virtuální počítače viditelné, přestože jsou na stejném fyzickém počítači.

Virtuální počítače také efektivněji využívají hardware, na kterém jsou umístěny. Poskytovatelé cloudových služeb tak mohou nabídnout využití svých serverů najednou velkému počtu zákazníků. (Cloudflare, 2023)

3.1.2 Historie cloud computingu

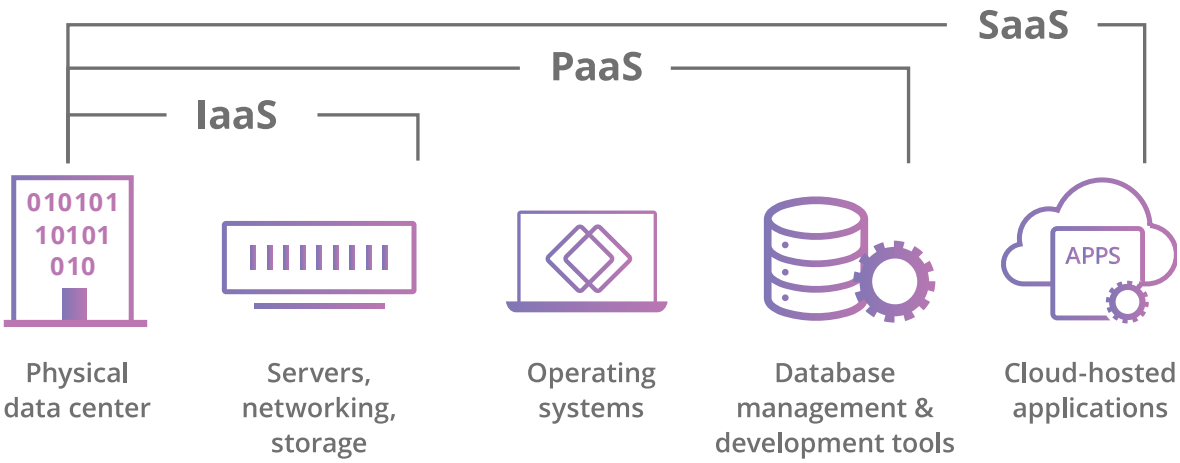
Historie Cloud computingu se začala psát před téměř 50 lety. Za duchovního otce této myšlenky je považován John McCarthy, profesor zprestižní americké univerzity MIT, který v roce 1961 jako první prezentoval myšlenku sdílení počítačových technologií ve stejné logice jako například sdílení elektrické energie.

Po nástupu tranzistorů, procesorů apočítačů se kpojmu vrátila především společnost Amazon. Té nevyhovovalo, že využívá pouze 10% své kapacity výpočetní techniky a zbytek leží ladem pro případy nárazového využití (špiček).

Vznikla tak první komerční služba cloud computing - Amazon Web Services (AWS) v roce 2006.
O rok později se připojil Google a IBM a řada univerzit začala pracovat na vědeckých / komerčních programech založených na Cloud Computing.

Od roku 2009 je Cloud Computing vnímán jako klíčová́ budoucí technologie. Mezi své nejdůležitější technologie ji zařadili přední výrobci ve světě ICT. (Bezpalec, 2015)

3.1.3 Služby cloud computingu



Obrázek 1- služby cloud computingu (Cloudflare, 2023)

3.1.3.1 Software as a service (SaaS)

Software jako služba (SaaS) je model poskytování softwaru v cloudu, kdy poskytovatel cloudu vyvíjí a udržuje software cloudové aplikace, poskytuje automatické aktualizace softwaru a zpřístupňuje software svým zákazníkům prostřednictvím internetu na základě platby za používání. Poskytovatel veřejného cloudu spravuje veškerý hardware a tradiční software, včetně middlewaru, aplikačního softwaru a zabezpečení. Zákazníci SaaS tak mohou výrazně snížit náklady, nasazovat, rozšiřovat a aktualizovat podniková řešení rychleji než při údržbě lokálních systémů a softwaru a s větší přesností předvídat celkové náklady na vlastnictví. (Oracle, 2024)

3.1.3.2 Infrastructure as a service (IaaS)

V tomto modelu se jedná o pronájem serverů a úložišť od poskytovatele cloudových služeb. Poté bývá takto pronajatá infrastruktura využita k provozování vlastního softwaru. (Cloudflare, 2023)

Ve skutečném světe to můžeme přirovnat k tomu, že si osoba pronajímá pozemek na kterém může vybudovat dům, bude si ale muset vlastními náklady zajistit materiál, nářadí a pracovní sílu.

3.1.3.3 Platform as a service (Paas)

V tomto modelu společnosti neplatí za hostované aplikace; místo toho platí za věci, které potřebují k vytvoření vlastních aplikací. Poskytovatelé PaaS nabízejí přes internet vše potřebné pro vývoj aplikace, včetně vývojových nástrojů, infrastruktury a operačních systémů. (Cloudflare, 2023)
Lze přirovnat k pronájmu všech nástrojů a nářadí, které je potřebné k postavení domu.

3.1.3.4 Function as a service (FaaS)

Také známé jako serverless computing. Jde o rozložení cloudové aplikace na ještě menší komponenty, které běží pouze tehdy, když jsou potřeba.
Serverless stále běží na serverech, podobně jako všechny tyto modely cloud computingu, ale nazývají se "serverless" (bezserverové), protože neběží na dedikovaných strojích. (Cloudflare, 2023)
Odpadá tak nutnost starání se o server jako takový, jeho konfiguraci, zabezpečení, prostředí, operační systém.

3.1.4 Výhody a nevýhody cloud computingu

3.1.4.1 Výhody

Žádné opakované investice do infrastruktury – je třeba zohlednit fakt, že zejména serverová infrastruktura se obvykle obnovuje ihned, jak skončí podpora od výrobce.

Žádné skryté výdaje.

Není potřeba správa a údržba – u Cloud computingu odpadá péče o serverovou část infrastruktury.

Možnost dynamicky měnit kapacitu – jako jedna z velkých předností Cloud computingu se často uvádí možnost dynamicky měnit kapacitu služeb, která je zejména u větších poskytovatelů takřka neomezená. (Bezpalec, 2015)

Škálování je možné provádět tzv. on demmand. To znamená, že je možné škálovat infrastrukturu dle aktuálního provozu v dané aplikaci. Odpadá tak nutnost držet infrastrukturu dimenzovanou tak, aby zvládla špičkovou zátěž i v čase, kdy na aplikaci je provoz výrazně menší.

Rychlost nasazení – Zejména u některých aplikací a infrastrukturních služeb, které musí být rychle nasazeny je obrovskou výhodou už samotná rychlost nasazení Cloud computingu. (Bezpalec, 2015)

Aktualizace – Aktulizace infrastruktury je typicky dodávána poskytovatelem cloudu, odpadá tak nutnost složitých migrací na nové operační systémy aj.

SLA – Service Level Agreement neboli smluvně garantovaná dostupnost služeb.

Bezpečnost – Ačkoliv využití cloudu samo o sobě bezpečnost infrastruktury nebo aplikace nezvyšuje, pokud jsou dodrženy obecně přijímané standardy, je nižší pravděpodobnost opomenutí některých bezpečnostních prvků. Jelikož služby, které zabezpečují aplikaci jsou kupovány od dodavatele cloudu, je minimalizováno riziko na zanesení chyby při implementaci takovýchto služeb.

3.1.4.2 Nevýhody

- **Žádné či malé úspory z rozsahu**
- **Někdy vyšší ceny, než za realizaci svépomocí** – cena u Cloud computingu může být samozřejmě ivyšší než cena při realizaci svépomocí, zejména pak zdlouhodobého hlediska. To se týká zejména pronájmu infrastruktury, kde existuje určitá přirážka za to, že je možné dynamicky měnit její kapacitu i za to, že poskytovatel Cloud computingu ve formě SLA garantuje obvykle poměrně vysokou (nejčastěji 99,9%) dostupnost, za jejíž nedodržení je sankcionován (obvykle slevou na měsíčním/ročním paušálu). (Bezpalec, 2015)

Data jsou uložena na infrastruktuře třetí strany – Data fyzicky neleží na interním datovém úložišti. To může být v některých případech značná nevýhoda, jelikož například zadání projektu neumožní využití cloudového poskytovatele.

Uživatelská data v internetu – Vzhledem k povaze cloudu, je vždy nutný přenos dat od zákazníka nebo uživatele k poskytovateli cloudu. Toto odpadá například při provozování interního systému, pouze ve vlastní síti.

Volba hardware a software je limitována nabídkou poskytovatelů cloudu – Při návrhu aplikace je tudíž nutné brát v potaz možnosti a nabídku služeb zvoleného cloudového poskytovatele a přizpůsobit jim návrh aplikace. (Bezpalec, 2015)

Vendor lock-in – Vystavění aplikace na službách konkrétního poskytovatele cloudu může znamenat naprostou závislost na tomto poskytovateli. V případě zdražení nebo změny smluvních podmínek může být velmi nákladné aplikaci upravit tak, aby ji bylo možné jednoduše přenést k jinému poskytovateli služeb.

3.2 Docker

Docker představuje otevřenou platformu navrženou pro vývoj, distribuci a provoz aplikací. Jeho hlavním přínosem je možnost oddělení aplikací od infrastruktury, což umožňuje zrychlení procesu dodávání softwaru. Docker nabízí nástroje umožňující efektivní správu infrastruktury podobným způsobem, jakým se spravují aplikace. Implementací metodik Dockeru pro distribuci, testování a nasazení kódu lze podstatně zkrátit časový interval mezi vývojem kódu a jeho nasazením do produkčního prostředí.

Docker umožňuje balit a spouštět aplikaci v izolovaném prostředí zvaném kontejner. Izolace a bezpečnost umožňují současně spouštět mnoho kontejnerů na daném hostiteli. Kontejnery jsou minimalistické a obsahují vše potřebné k běhu aplikace, takže není nutné spoléhat na to, co je nainstalováno na hostiteli. Kontejnery mohou být sdíleny a použity jinde, tudíž unifikují běhové prostředí aplikace. (Docker, 2024)

3.3 AWS[4]

Amazon Web Services nabízí širokou sadu globálních cloudových produktů včetně výpočetní kapacity, úložiště, databází, analýzy, sítí, mobilních služeb, nástrojů pro vývojáře, nástrojů pro správu, IoT, bezpečnosti a podnikových aplikací: na vyžádání, dostupné během několika sekund, s cenami podle skutečného využití. Od datových skladů po nástroje pro nasazování, adresáře po doručování obsahu, je dostupných přes 200 služeb AWS. Nové služby lze rychle zprovoznit bez počátečních fixních výdajů. To umožňuje podnikům, start-upům, malým a středním firmám a zákazníkům ve veřejném sektoru získat stavební bloky, které potřebují k rychlé reakci na měnící se obchodní požadavky.
Infrastruktura cloudového prostředí AWS je postavena kolem AWS regionů a dostupnostních zón. AWS Region je fyzická lokace ve světě, kde se nachází více dostupnostních zón. Dostupnostní zóny se skládají z jednoho nebo více samostatných datových center, každé s redundantním napájením, sítěmi a konektivitou, umístěnými v oddělených zařízeních. Tyto dostupnostní zóny nabízí možnost provozovat produkční aplikace a databáze, které jsou vysoce dostupné, odolné vůči chybám a škálovatelné větší mírou, než by bylo možné v případě jediného datového centra. (AWS, 2024)

3.3.1 Lambda

FaaS, tedy function as a service je označení pro výpočetní model, který v portfoliu služeb AWS zastupuje Lambda funkce.

Lambda je platforma pro výpočty bez nutnosti administrace. Uživatelé nemusí konfigurovat, spouštět ani monitorovat serverové instance. Není nutné instalovat operační systémy ani prostředí pro programovací jazyky. Uživatelé nemusí řešit otázky škálování či odolnosti proti chybám a není potřeba žádat ani rezervovat kapacitu dopředu.
Nejdůležitějším konceptem platformy Lambda je funkce Lambda, nebo stručně funkce.
Kontextové informace specifikují výkonné prostředí (jazyk, požadavky na paměť, časový limit a IAM[5] roli) a také odkazují na funkci, kterou chcete exekuovat. Kód a metadata jsou

[4] Amazon Web Services
[5] Identity Access Management – služba AWS na správu oprávnění

trvale uloženy v AWS a později lze na ně odkazovat pomocí jména nebo ARN[6]. (AWS, 2014)
Lambda funkce neuchovává persistentní stav, po skončení exekuce je funkce zahozena, tudíž k uchování dat je potřeba využít S3[7] nebo databáze.
Každá funkce Lambda běží ve svém vlastním kontejneru. Když je funkce vytvořena, kód je nasazen do nového kontejneru a kontejner poté spuštěn na clusteru strojů spravovaném AWS. Před zahájením běhu funkcí je každému kontejneru přidělena potřebná kapacita RAM a CPU. Jakmile funkce dokončí svůj běh, alokovaná RAM na začátku se vynásobí dobou, po kterou funkce běžela. Zákazníci jsou poté účtováni na základě přidělené paměti a doby běhu, kterou funkce potřebovala k dokončení. (Serverless, 2023)

3.3.2 EC2[8]

Amazon Elastic Compute Cloud (Amazon EC2) představuje rozsáhlou platformu pro výpočetní služby, která zahrnuje více než 750 typů instancí. Tyto instance nabízejí různé konfigurace procesorů, úložišť, síťových řešení, operačních systémů a modelů nákupu, což umožňuje uživatelům optimalizovat výběr v souladu s požadavky jejich specifických pracovních zátěží. AWS je významným inovátorem v oblasti poskytování cloudových služeb, neboť jako první podporuje procesory Intel, AMD a Arm, a nabízí instance EC2 Mac na vyžádání a poskytuje 400 Gbps připojení. Platforma je rovněž ceněna za poskytování nejefektivnějšího poměru ceny a výkonu pro trénink strojového učení a za nabídku instancí. (Amazon Web Services, 2024)

3.4 Technologie implementace

V této kapitole budou představeny technologie, jež jsou použity v praktické části této práce. Bude přiblížena historie, důvody vzniku a jednotlivé aspekty technologií používané pro implementaci služby.

[6] Amazon Resource Name – identifikátor konkrétní instance AWS služby
[7] Simple Storage Service – souborové úložiště AWS
[8] Elastic Compute Cloud

3.4.1 Java

Ačkoliv Java není použita přímo pro implementaci, pochopení tohoto jazyka je nutné pro využití Kotlinu v práci.
Java je objektově orientovaný programovací jazyk vyvinut Jamesem Goslingem a jeho kolegy ve společnosti Sun Microsystems v začátcích 90 let. Na rozdíl od konvenčních jazyků, které byly určeny k překladu do nativního (zdrojového) kódu nebo byly interpretovány ze zdrojového kódu za běhu, je Java určena ke kompilaci do bajtkódu, který je následně spuštěn pomocí JVM[9].

Jazyk jako takový se syntakticky inspiruje v jazycích C a C++, ale disponuje jednodušším objektovým modelem a méně nízko-úrovňovými možnostmi. Java je pouze vzdáleně příbuzná JavaScriptu, jelikož syntaxe obou jazyků vychází z jazyku C.
Java vznikla jako projekt s názvem Oak, který vytvořil James Gosling v červenci roku 1991. Goslingovým cílem bylo implementovat virtuální stroj a jazyk, který by měl povědomou syntaxi z jazyka C, ale zároveň by byl univerzálnější a jednodušší než C nebo C++. První veřejná implementace byla Java 1.0 v roce 1995.
Zavázala se příslibem „Napiš jednou, spusť kdekoliv“ s runtimy zdarma na populárních platformách.
V roce 1997 společnost Sun oslovila normalizační orgán ISO / IEC JTC1 a později Ecma International, aby Javu formalizovala, ale brzy z procesu ustoupila.
Java zůstává de facto proprietárním standardem, který je řízen procesem Java Community Process. Sun poskytuje většinu svých implementací Javy zdarma, přičemž tržby generují specializované produkty, jako je Java Enterprise System. Sun rozlišuje mezi svou Software Development Kit (SDK) a Runtime Environment (JRE), což je podmnožina SDK, přičemž hlavní rozdíl spočívá v tom, že v JRE kompilátor není přítomen.
V roce 2009 koupila firmu Sun společnost Oracle a tím i Javu. (Bechný, 2021)

3.4.1.1 JVM

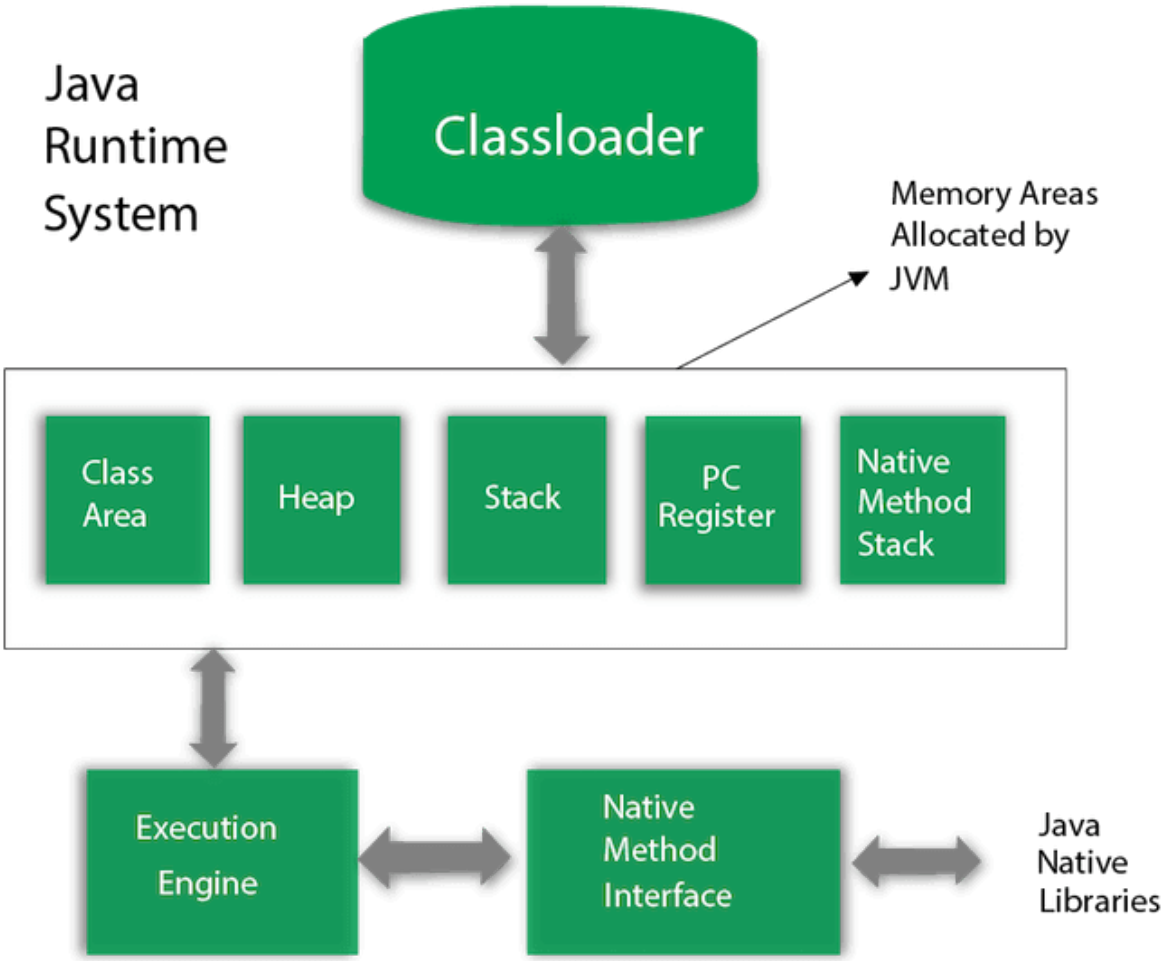
Java Virtual Machine představuje základní kámen platformy Java. Je to technologie, která je zodpovědná za její nezávislost na hardwaru a operačním systému, malou velikost zkompilovaného kódu a schopnost chránit uživatele před škodlivými programy.

[9] Java Virtual Machine

JVM je abstraktní výpočetní stroj. Podobně jako reálný výpočetní stroj má soubor instrukcí a během provozu manipuluje s různými paměťovými oblastmi. Implementace programovacího jazyka pomocí virtuálního stroje je poměrně běžná; nejznámější virtuální stroj může být P-Code stroj UCSD Pascalu. První prototypová implementace JVM byla provedena ve společnosti Sun Microsystems, Inc., kde emulovala instrukční sadu JVM v softwaru hostovaném na přenosném zařízení. Současné implementace společnosti Oracle emulují JVM na mobilních, desktopových a serverových zařízeních, ale JVM nepředpokládá žádnou konkrétní technologii implementace, hostitelský hardware nebo operační systém. JVM není inherentně interpretována a může být může být také implementována kompilací instrukční sady do instrukční sady CPU.

JVM nezná programovací jazyk Java, zná pouze konkrétní binární formát, formát souboru třídy. Soubor třídy obsahuje instrukce JVM (bytekódy) a symbolickou tabulku, stejně jako další pomocné informace. Z důvodu bezpečnosti JVM klade silné syntaktické a strukturální omezení na kód v souboru třídy. Nicméně, jakýkoli jazyk s funkcionalitou, která může být vyjádřena v termínech platného souboru třídy, může být hostován JVM. (Oracle, 2024)

Princip vnitřní architekruy JVM popisuje přiložený obrázek.



Obrázek 2 - Architektura JVM (Javatpoint, 2021)

3.4.2 Kotlin

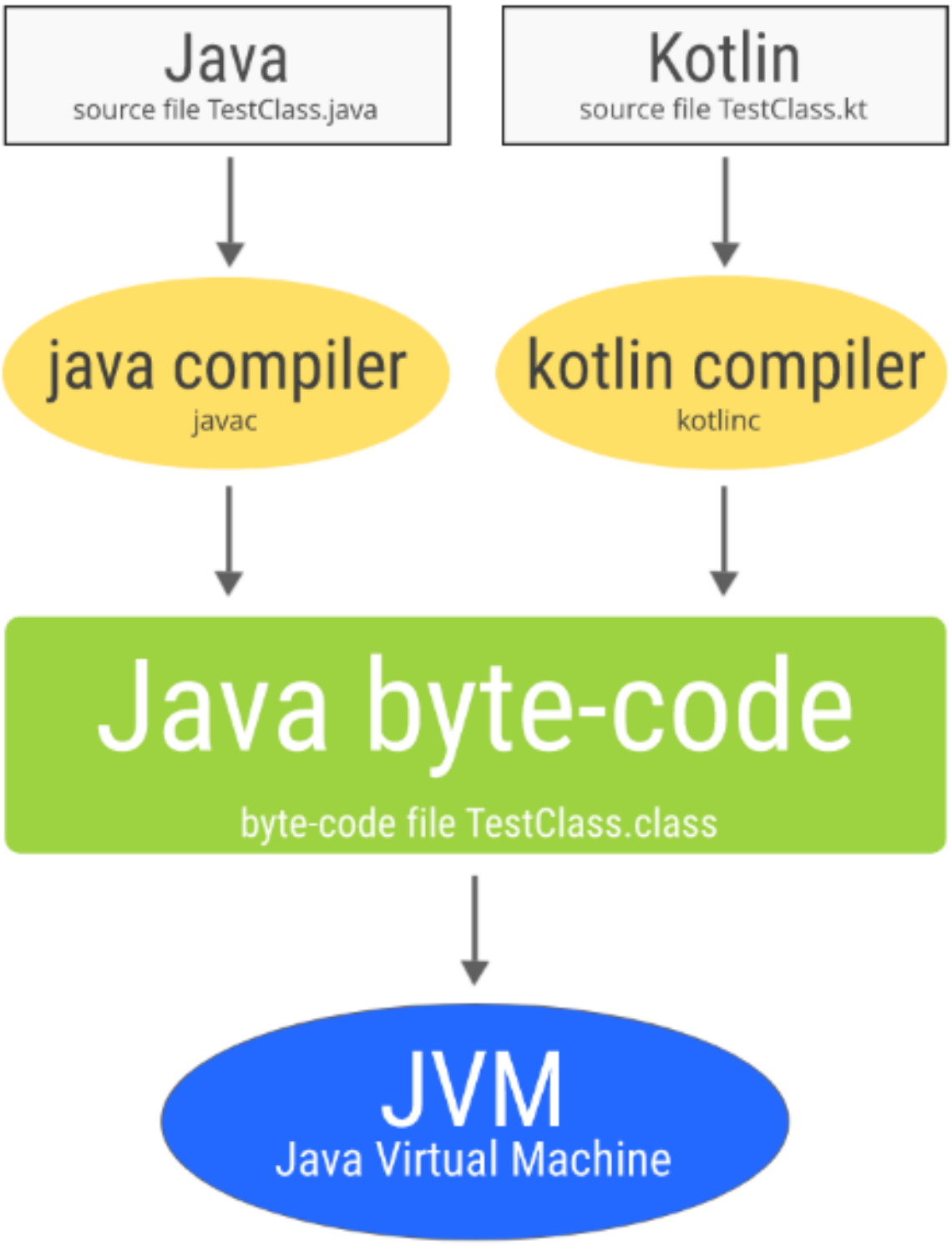
V předchozí kapitole byl popsán jazyk Java a zejména technologie JVM, která umožňuje implementaci jiných programovacích jazyků než Java nad technologií JVM.
V praktické části této práce je využíván jazyk Kotlin, který taktéž spadá do tzv. JVM jazyků.

Kotlin je staticky typovaný programovací jazyk vyvíjený firmou JetBrains. Snaží se napravit časté problémy, které se vyskytují v jiných jazycích, proto je navržen tak, aby byl snadno čitelný a aby svým návrhem předcházel chybám. Vznikl s cílem být plně interoperabilní s jazykem Java, ale oproti jazykům jako Scala, Ceylon nebo Clojoure není akademickým

počinem. Přestože i Java se vyvíjí a doplňuje jazyk o populární a užitečné konstrukty, Kotlin jich stále nabízí víc, než Java 11, 17 nebo i 21.
Snaží se být pragmatický, stručný a bezpečný. Jazyk Kotlin je nejen objektově orientovaný, ale také funkcionální. Obsahuje tedy třídy a rozhraní, ale i funkce vyšších řádů, lambda výrazy nebo líné vyhodnocení. Vývoj Kotlinu započal roku 2010. Poté, co v roce 2016 vyšla verze 1.0, netrvalo společnosti Google ani rok, aby přijala Kotlin jako oficiálně podporovaný jazyk pro vývoj Android aplikací. V počátku sloužil Kotlin především jako alternativa k Javě a běžel na JVM. Experimentální podpora multiplatformního vývoje byla zahájena v roce 2017 s možností sdílení kódu pro JVM a JavaScript. Společnost JetBrains také oznámila, že pracuje na projektu Kotlin/Native — překladači Kotlinu do nativních binárních kódů. V tuto chvíli je možné v Kotlinu programovat mobilní, webové i desktopové aplikace. (Dlabal, 2022)

Za základní variantu Kotlinu by se dal považovat Kotlin/JVM. Jde o jeho nejrozšířenější podobu, která se používá především pro vývoj serverových aplikací a aplikací pro Android. Průzkum firmy JetBrains z roku 2020 ukázal, že 65 % respondentů využívá Kotlin k vývoji Android aplikací, 48 % k vývoji Kotlin/JVM serverových aplikací a pro jiné typy aplikací využívá Kotlin/JVM 15 % dotazovaných. Další anketa, taktéž provedená firmou JetBrains, z roku 2021 odhalila, že 49 % dotazovaných programují v Kotlinu mobilní Android aplikace a 67 % serverové aplikace. (Dlabal, 2022)

Na obrázku je znázorněn proces kompilace Kotlinu do Java bajtkódu. Jak z grafu vyplývá, Kotlin lze používat v jednom projektu souběžně s třídami napsanými v jazyce Java.



Obrázek 3 - Kompilace Kotlinu a Javy (Markovic, 2020)

Tomu je využito například v operačním systému Android, jehož části jsou implementovány v těchto různých jazycích.

3.4.2.1 Správa závislostí - Maven

Maven, slovo jidiš znamenající akumulátor znalostí, začal jako pokus zjednodušit procesy sestavení v projektu Jakarta Turbine. Existovalo několik projektů, z nichž každý měl své vlastní soubory pro sestavení Ant, které byly všechny mírně odlišné. JAR soubory byly ukládány do systému CVS[10].
Byl vyjádřen požadavek na standardní způsob sestavení projektů, jasnou definici toho, co projekt obsahuje, snadný způsob publikování informací o projektu a způsob sdílení JAR souborů napříč několika projekty. Výsledkem je nástroj, který lze nyní použít pro sestavení a správu jakéhokoli projektu založeného na Javě. (Apache Maven, 2024)

3.4.2.2 Správa závislostí – Gradle

Gradle je open-source nástroj pro sestavování, který je schopen vývoje jakéhokoli druhu softwaru. Tento nástroj vyvinula skupina vývojářů vedená Hansem Dockterem, Szczepanem Faberem, Adamem Murdochem, Lukem Daleym, Peterem Niederwieserem, Dazem DeBoerem a Rene Gröschkem. Jedná se o automatizační nástroj založený na Apache Ant a Apache Maven. Tento nástroj je schopen sestavovat aplikace podle průmyslových standardů a podporuje řadu jazyků, včetně Groovy, C++, Java, Scala a C. Gradle je také schopen řídit vývojové úkoly od kompilace a balení po testování, nasazení a publikování.

Ve srovnání s Ant a Maven je Gradle nejstabilnějším nástrojem. Tento nástroj byl původně vydán na konci roku 2007 jako alternativa k předchůdcům, která je nejen nahradila, ale také překonala jejich nedostatky. (Geeks for Geeks, 2023)

3.4.3 Spring

Spring usnadňuje vytváření podnikových aplikací v jazyce Java. Nabízí vše potřebné pro využití jazyka Java v podnikovém prostředí s podporou alternativních jazyků na JVM, jako jsou Groovy a Kotlin, a s flexibilitou pro vytváření různých druhů architektur v závislosti na potřebách aplikace.
Spring podporuje širokou škálu aplikačních scénářů. V rozsáhlých podnicích aplikace často existují dlouhou dobu a musí běžet na JDK a aplikačním serveru, jejichž cyklus aktualizace

[10] Concurrent Versioning Systém – Systém na verzování obsahu

není pod kontrolou vývojářů. Jiné mohou běžet jako samostatný jar s vestavěným serverem, možná v cloudovém prostředí. Další mohou být samostatné aplikace (například dávkové nebo integrační workloads), které nepotřebují server.

Spring je open source. Má velkou a aktivní komunitu, která poskytuje nepřetržitou zpětnou vazbu na základě rozmanité škály reálných použití. (Spring, 2024)

Předtím, než bude věnována pozornost frameworku Spring a SpringBoot, je nutné nejprve prozkoumat některé koncepty programování, které tento framework pomáhá vyřešit a tím zjednodušit vývoj aplikací.

3.4.3.1 IoC / Dependency injection

Tato kapitola se zabývá implementací principu Inverze řízení (IoC) v rámci Spring Frameworku. Injektáž závislostí (Dependency Injection, DI) je specializovanou formou IoC, při které objekty definují své závislosti (tj. další objekty, se kterými spolupracují) pouze prostřednictvím argumentů konstruktoru, argumentů tovární metody nebo vlastností, které jsou nastaveny na instanci objektu po jeho vytvoření nebo vrácení z tovární metody. Kontejner IoC poté tyto závislosti injektuje v okamžiku vytváření beanu. Tento proces je opačný (odtud název, Inverze řízení) k situaci, kdy bean sám kontroluje vytváření nebo lokalizaci svých závislostí použitím přímé konstrukce tříd. (Spring, 2024)

3.4.3.2 Spring Boot

Spring Boot je nadstavba Spring Frameworku, která řeší nedostatek Spring frameworku a to je absence servlet kontejneru a http serveru. Implementuje tedy veškeré funkcionality Spring Frameworku a navíc má v sobě zabudovaný servlet kontejner (Tomcat), který by si vývojáři jinak museli nastavovat ručně. Tento kontejner při spuštění projektu automaticky vytvoří a nakonfiguruje server, ke kterému se budou moci připojovat uživatelé přes HTTP protokol. (Network, 2024)
Mezi nejdůležitější výhody Spring boot frameworku patří zejména autokonfigurace, kdy není nutné trávit množství času konfigurací frameworku a schopnost vytvářet samostatně stojící aplikace, které je možné spustit jednoduše a samostatně.

3.4.4 Tomcat

Jako referenční implementace Java Servlet a Java Server Pages (JSP) byl Tomcat zahájen ve společnosti Sun Microsystems, která později v roce 1999 darovala zdrojový kód Apache Software Foundation.
Od té doby přispělo k produktu mnoho dobrovolníků ze Sun a dalších organizací, což vedlo k jeho označení jako projektu nejvyšší úrovně Apache. V současnosti je Apache Tomcat široce využíván mnoha společnostmi, jelikož implementuje mnoho specifikací Jakarta EE, jako jsou:

- Java Servlet,
- JavaServer Pages,
- Java Expression Language,
- Java WebSockets.

(Fol, 2024)

Tomcat je považován za webový server spíše než za aplikační server, protože funguje jako webový server a kontejner Servletů. Neposkytuje plnou sadu funkcí Jakarta EE, což však nutně nemusí být nevýhodou.

3.4.5 Javascript

JavaScript je jedno-vláknový, objektově orientovaný programovací jazyk. JavaScript je jedním ze tří hlavních technologií pro web. (další jsou HTML a CSS). Obvykle JavaScript provádí kód v prohlížeči, tedy na straně uživatele, jsou však nástroje, které umožňují jeho využití jinde. (Procházka, 2019)

3.4.5.1 Historie

Javascript vznikl reakcí na vývoj Java appletů, tedy mikro programů, které umožňovaly spustit Javu v prohlížeči.
To dobou ve společnosti Netscape Communications programátor Brendan Eichl vytvořil speciální jazyk pro prohlížeč Netscape. Prohlížeč podle něj měl mít svůj vlastní skriptovací jazyk, který mohl sloužit jako jednoduchý nástroj pro vytváření jednoduchých dynamických

prvků do webových stránek. Jeho původní plán byl napsat „Scheme" pro prohlížeč. První verze JavaScriptu byla v roce 1995 přidána do Netscape Navigator 2. (Js pro vývojáře, 2021)

Javascript se k dnešnímu dni stal standardem pro vývoj webových stránek. Společnosti Microsoft například nadstavbu jazyka nazvanou TypeScript, protože JavaScript neobsahuje typovou kontrolu a jedná se o volně dynamicky typovaný jazyk. Dnes již okolo JavaScriptem existuje celá řada technologií a frameworků, které vytváří bohatý ekosystém okolo tohoto jazyka. Následující výčet zahrnuje pouze ty, které budou použity v implementační části práce a nemá ambici zahrnout desítky různých frameworků použitých ve světě JavaScriptu.

3.4.5.2 NodeJs

Node.js je open-source a multiplatformní prostředí běhu JavaScriptu. Jedná se o oblíbený nástroj pro téměř jakýkoli typ projektu.
Jako asynchronní, na událostech založené prostředí běhu JavaScriptu, je Node.js navrženo pro vývoj škálovatelných síťových aplikací.
Node.js je ve svém designu podobné a ovlivněné systémy jako je Event Machine v Ruby a Twisted v Pythonu.
Node.js používá V8 JavaScript engine, jádro Google Chrome, mimo prohlížeč. To umožňuje Node.js dosahovat vysokého výkonu.
Aplikace Node.js běží v jednom procesu bez nutnosti vytvářet nové vlákno pro každý požadavek. Node.js poskytuje sadu asynchronních I/O primitiv ve své standardní knihovně, která brání blokování kódu JavaScriptu a obecně jsou knihovny v Node.js psány s využitím neblokujících paradigmat, čímž se blokování stává výjimkou, nikoli pravidlem.

Když Node.js provádí I/O operaci, jako je čtení ze sítě, přístup k databázi nebo souborovému systému, místo blokování vlákna a plýtvání cykly CPU čekáním, Node.js obnoví operace tehdy, když přijde odpověď.
To umožňuje Node.js zvládat tisíce současných spojení s jediným serverem bez nutnosti správy konkurence vláken, což bývá významným zdrojem chyb.
Výhodou Node.js je znalost JavaScriptu frontendovými vývojáři, to jim umožnňuje psát kód na straně serveru navíc ke kódu na straně klienta, aniž by se museli učit zcela odlišný jazyk. (NodeJS, 2024)

3.4.5.3 Správa balíčků

npm znamená Node Package Manager, tedy manažer balíčků pro Node.js.

Společnost npm, Inc. byla založena v roce 2014 a v roce 2020 byla akvizicí převedena pod společnost GitHub.
npm představuje klíčovou součást komunity JavaScriptu a podporuje jedno z největších vývojářských ekosystémů na světě.
Pod npm můžeme chápat několik věcí:

- npm je správce balíčků pro Node.js. Byl vytvořen v roce 2009 jako open-source projekt s cílem usnadnit vývojářům JavaScriptu sdílení balíčkovaných modulů kódu.
- npm Registry je veřejná sbírka balíčků open-source kódu pro Node.js, webové aplikace, mobilní aplikace, roboty, routery a nesčetné další potřeby komunity JavaScriptu.
- npm je příkazový řádek, který vývojářům umožňuje instalovat a publikovat tyto balíčky.

(npm, Inc, 2024)

3.4.6 Generování PDF

Vzhledem k potřebe generovat PDF na straně serveru jsou možnosti výrazně omezené. Nelze tak využít některých JavaScriptových knihoven pro generování PDF na straně uživatele, které využívají prohlížeč uživatele.

Po zvážení implementační náročnosti, zkušeností vývojářského týmu a testování jednotlivých řešení byla pro tuto službu vybrána knihovna Playwright, které se bude věnovat následující kapitola.

3.4.6.1 Playwright

Playwright je open-source knihovna pro automatizaci testování prohlížečů a web scraping, vyvinutá společností Microsoft a spuštěná 31. ledna 2020, která se od té doby stala populární mezi programátory a webovými vývojáři.

Playwright umožňuje automatizaci úkolů v prohlížečích Chromium, Firefox a WebKit pomocí jednotného API. To umožňuje vývojářům vytvářet spolehlivé end-to-end testy, které jsou schopny běžet také v headless režimu. Playwright podporuje programovací jazyky jako JavaScript, Python, C# a Java, přestože jeho hlavní API bylo původně napsáno v Node.js. Podporuje všechny moderní webové funkce včetně zachytávání síťové komunikace a více kontextů prohlížeče a poskytuje automatické čekání na zobrazení elementu, což snižuje nespolehlivost testů. (Microsoft, 2024)

Playwright byl oznámen společností Microsoft v lednu 2020. Byl vyvinut týmem inženýrů, kteří dříve pracovali na podobných projektech, jako je Puppeteer v Google.
Běhové prostředí @playwright/test bylo vydáno později jako součást snahy poskytnout komplexnější řešení pro testování založené na prohlížeči. Jeho vývoj byl do značné míry založen na potřebě mít specializovaný runner, který může využít plný potenciál Playwright API a učinit end-to-end testování robustnějším a přímočařejším. (Schiemann, 2020)

4 Vlastní práce

V této sekci práce je nastíněn skutečný problém a jeho navrhované řešení. Implementace se bude zabývat implementací samostatně stojící cloudové služby, pro generování PDF dokumentů. Budou představeny 2 různé architektury, které se opírají o cloud od AWS. Pro první variantu bude použito označení „on-premise“, dále jen OP. Ve smyslu této aplikace se nejedná o skutečné OP řešení, provozováno na skutečném fyzickém serveru, ale o využití pronajatého serveru, služby EC2 od AWS, která je popsána výše v teoretické části.
Na konci je provedena komparace vybraných řešení a na základě analýzy dat a vlastností jednotlivých způsobů implementace je zvoleno vhodné řešení pro konkrétní případ.

4.1 Popis řešeného problému

Požadavek na implementaci systému na generování PDF vznikl jako business požadavek vycházející od zákazníků používající systém pro detekci plagiátorství, který je integrován do různých LMS[11] systémů nebo je k dispozici přes API[12]. Dále je představen zevrubný popis obdobného systému. Z pravidla se jedná o komplexní webovou aplikaci, která přijímá text v různých formátech. Student nebo učitel využijí rozhraní pro odevzdávání prací v různých podporovaných formátech nebo vloží text přímo, toto bude dále označováno jako dokument. Po vložení je vstupní dokument validován, zda se jedná o podporovaný formát nebo není jinak poškozen. Pokud je tato validace úspěšná, dokument je přijat ke zpracování a je vrácena odpověď, že dokument bude zpracován.
Následně je dokument zařazen do fronty a dostává přiděleno ID zpracování. V okamžiku, kdy je dokument přebrán ke zpracování, dochází k další úpravě textu dokumentu. Ten je rozčleněn do celků, jenž se nazývají věty, ačkoliv tyto věty nemusí odpovídat skutečné větě tak, jak ji chápeme v lingvistickém smyslu. Tento proces je podobný procesu tokenizace používané v LLM[13]. Tyto větné celky jsou následně porovnány s databází, která obsahuje tzv. otisk internetu a jiné zdroje, typicky akademické články či již dříve odevzdané práce.

Pokud je nalezena shoda – podobnost, je to zaznamenáno do metadat odevzdaného dokumentu. Výsledný soubor s metadaty je následně uložen a na dotaz je vrácen zpět

[11] Learning Management System
[12] Application Programming Interface
[13] Large Language Models – označení pro neuronové sítě zabývající se zpracováním jazyka

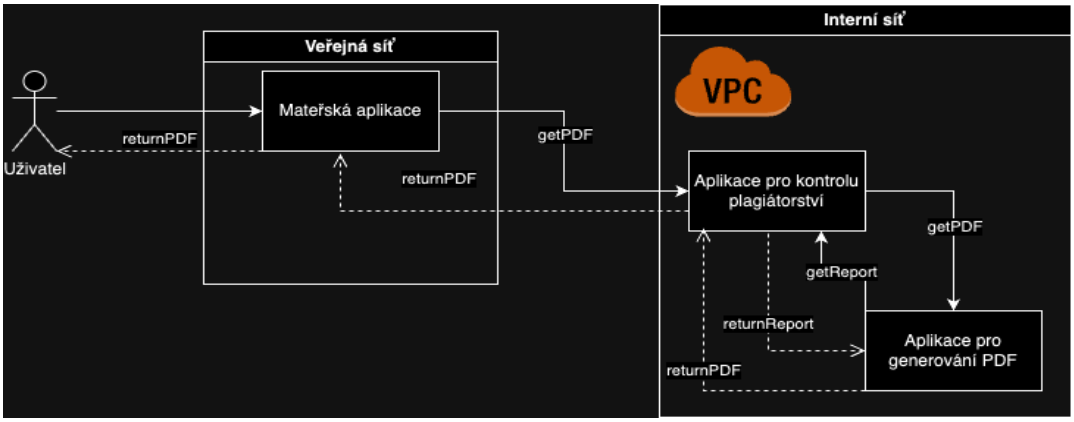
žadateli ve formě HTML stránky. Tento výsledek bývá označován jako report o plagiátorství.

Systém se vyznačuje výkonovými špičkami, kdy dochází k odevzdání značného množství souborů za krátký časový úsek. Typicky se tak děje ke konci akademických semestrů či jiných deadlinů.

Instituce mohou v případě podezření na plagiátorství zahájit s autorem práce řízení o plagiátorství a tento report může sloužit jako jeden z důkazních materiálů. Proto vzniká požadavek na možnost archivace v neměnném a snadno přenositelném formátu.

4.2 Diagram systému

Výše popsané zasazení služby do existujícího systému je názorně popsáno přiloženým diagramem.



Obrázek 4 - Diagram komunikace (vlastní zpracování)

Uživatel žádá „mateřskou aplikaci“, která běží ve veřejné síti a je tedy dostupná z internetu. Ta zajišťuje prvotní autentikaci uživatele. Tato aplikace se následně autentikuje aplikaci pro kontrolu plagiátorství. Pokud je požadavek ověřen, je zahájeno generování PDF z adresy, kterou poskytuje aplikace pro kontrolu plagiátorství. Po vygenerování je PDF vráceno postupně přes tyto aplikace až k uživateli, který na začátku celý proces spustil.

4.2.1 Analýza požadavků na systém

Nejdříve je potřeba stanovit funkční a nefunkční požadavky na systém.

4.2.2 Funkční požadavky

V tomto případě je definice funkčních požadavků triviální. Jelikož se jedná o aplikaci, jenž je zamýšlena k integraci do existujícího systému a rozšiřuje jeho funkcionalitu, je stanovení funkčních požadavků zřejmé:

Aplikace bude generovat PDF z HTML zadaného jako URL
Aplikace lokalizuje PDF do jazyka dle příchozího požadavku, je-li k dispozici.

Aplikace nemá vlastní front-end, ani API, které je určeno k veřejnému použití, proto jsou možnosti z hlediska koncového uživatele omezené a jsou řízeny systémy, do níž je tato aplikace integrována.

4.2.3 Nefunkční požadavky

V případě nefunkčních požadavků již jejich stanovení není triviální, jako v případě funkčních. Od aplikace se očekává, v poměrně rychlém (v řádech sekund, což bylo stanoveno jako přijatelná doba čekání na PDF dokument, odvíjí se též od velikosti dokumentu) čase obsloužení paralelních požadavků na lišící se velikosti souborů.

- Doba odezvy
  - Definována v řádech jednotek vteřin pro obvykle velké soubory. Medián velikosti souboru odevzdávaného do systému jsou ~2 strany textu.
- Škálovatelnost
  - Možnost škálovat aplikaci během špičkového provozu.
- Dostupnost
  - Dostupnost se řídí SLA[14] „mateřského“ systému.
- Bezpečnost
  - Aplikační bezpečnost je řešena pomocí VPC[15]. Aplikační endpoint je dostupný pro volání pouze vnitřní sítě systému, do kterého je aplikace

[14] service-level agreement
[15] Virtual Private Cloud

implementována. V praxi dochází k vrácení PDF klientovi, kterým je „mateřská“ aplikace, která dále přebírá zodpovědnost za autentikaci žadatele a vrácení PDF souboru. Službu tudíž může volat pouze mateřská aplikace, tudíž služba na generování PDF „dědí“ bezpečnostní nastavení z mateřské aplikace. Jinými slovy, pouze specifická aplikace smí zavolat endpoint této aplikace, veškerá sanitizace a očištění requestu probíhá o úroveň výše.

4.3 On-premise verze

Pro OP řešení aplikace je jako jazyk pro implementaci zvolen Kotlin. Využívá se frameworku SpringBoot. Aplikace je distribuována jako docker kontejner, který je spuštěn na EC2 instanci.

4.3.1 Výběr technologie

Pro tuto implementaci je zvolen programovací jazyk Kotlin společně s frameworkem Spring Boot. Jedná se o standardní kombinaci, která je běžně využívána v Java ekosystému. Jelikož služba je implementována do aplikací běžících v Javě, je jednoznačnou volbou JVM technologie, kvůli standardizaci.
Kotlin byl vybrán jako modernější alternativa Javy, potenciální nástupce, jak je popsáno v teoretické části této práce.

Spring Boot pak zajistí spuštění a provoz webového serveru bez nutnosti extenzivní konfigurace.

Pro část sestavení je zvolen Gradle, který na rozdíl od Mavenu umožňuje využití tzv. Gradle wrapperu, což je technologie, která sebou přináší zabalený Gradle engine. Není tudíž nutné instalovat Gradle do prostředí serveru, ale je distribuován rovnou s aplikací. Odpadá tedy nutnost instalace Gradlu do OS[16].

4.3.2 Kód aplikace

Projekt pro OP verzi má následující strukturu:

[16] Operační Systém

```
├── build.gradle.kts
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
├── settings.gradle.kts
└── src
    ├── main
    │   ├── kotlin
    │   │   └── cz
    │   │       └── bechny
    │   │           ├── deploy
    │   │           │   └── Deployment.kt
    │   │           └── pdfgenerator
    │   │               ├── PdfGeneratorApplication.kt
    │   │               ├── ServletInitializer.kt
    │   │               ├── controller
    │   │               │   └── PdfGeneratorController.kt
    │   │               └── tools
    │   │                   └── PdfGenerator.kt
    │   └── resources
    │       ├── application.properties
    │       ├── static
    │       └── templates
    └── test
        └── kotlin
            └── cz
                └── bechny
                    └── pdfgenerator
                        └── PdfGeneratorApplicationTests.kt
```

4.3.2.1 Sestavení

Pro sestavení aplikace je použit Gradle wrapper. Ten je možné spusit z kořenového adresáře aplikace. Proces sestavení začíná příkazem:

```
./gradlew clean bootWar
```

Tento příkaz spustí gradle wrapper a sestaví aplikaci tak, jak je definováno v souboru, nacházejícím se taktéž v kořenovém adresáři, **build.gradle.kts**.
Tento soubor obsahuje definici průběhu sestavení aplikace a taktéž závislosti potřebné ke spuštění.

```
1. import org.jetbrains.kotlin.gradle.tasks.KotlinCompile
2.
3. plugins {
4.        war
5.        id("org.springframework.boot") version "3.2.2"
6.        id("io.spring.dependency-management") version "1.1.4"
7.        kotlin("jvm") version "1.9.22"
8.        kotlin("plugin.spring") version "1.9.22"
9. }
10.
11. group = "cz.bechny"
12. version = "0.0.1-SNAPSHOT"
13.
14. java {
15.       sourceCompatibility = JavaVersion.VERSION_17
16. }
17.
18. repositories {
19.       mavenCentral()
20. }
21.
22. val playwrightVersion = "1.17.1"
23. val cdkVersion = "2.89.0"
24. val springBootVersion = "3.2.2"
25.
26. dependencies {
27.       implementation("org.springframework.boot:spring-boot-starter-
web:$springBootVersion")
28.       implementation("org.jetbrains.kotlin:kotlin-reflect")
```

```
29.       implementation("com.microsoft.playwright:playwright:$playwrightVersion")
30.
31.       implementation("software.amazon.awscdk:aws-cdk-lib:${cdkVersion}")
32.
33.       providedRuntime("org.springframework.boot:spring-boot-starter-
tomcat:$springBootVersion")
34.       testImplementation("org.springframework.boot:spring-boot-starter-
test:$springBootVersion")
35. }
36.
37. tasks.withType<KotlinCompile> {
38.       kotlinOptions {
39.                 freeCompilerArgs += "-Xjsr305=strict"
40.                 jvmTarget = "17"
41.       }
42. }
43.
44. tasks.withType<Test> {
45.       useJUnitPlatform()
46. }
47.
```

(Vlastní zpracování)

Řádek 4 definuje, že se jedná o WAR, tedy Web Application Resource. Tím při sestavení vzniká soubor war místo obvyklého jar, které je běžné pro java aplikace. Tento balíček obsahuje veškerý kód potřebný pro provoz webové aplikace (serveru). Jelikož je aplikace psána v jazyce Kotlin, je nutné určit, do jaké verze bytekódu bude zdrojový kód zkompilován. V tomto případě nastavujeme verzi kompatibility pro Javu verze 17. V době vydání aplikace je jednalo o poslední LTS[17] verzi Javy.

Následuje deklarace verzí balíčků, které tato aplikace využívá. Je zde možné nastavit verzi frameworku Spring Boot a Playwright, který zajišťuje tvorbu PDF.

4.3.3 Kód aplikace

Samotný kód aplikace se nachází v adresáři src/main/kotlin/cz/bechny/pdfgenerator/.

[17] Long Term Support

Najdeme zde soubor **PdfGeneratorApplication.kt**. Jedná se o vstupní soubor této aplikace. Definuje v sobě pomocí anotace *SpringBootApplication* třídu, která slouží jako hlavní třída pro načtení kontextu Spring Boot aplikace. Přu spuštění aplikace pomocí příkazu:

```
./gradlew bootRun
```

dojde k načtení a spuštění právě této třídy.

4.3.3.1 Web server

Po nastartování aplikace je k dispozici http server na nakonfigurovaném portu.
Tento port je v základu nastaven na hodnotu 8080. Tuto hodnotu je ovšem možné změnit a jednou z variant je nastavení vlastnosti *server.port* v souboru *application.properties* ve složce *resources*.
Tento soubor slouží k nejrůznějším konfiguracím aplikace. Je nutné mít na paměti to, že možností, jak nastavit port je ve skutečnosti více a mají různé pořadí vyhodnocení. Pokud např. bude port definován při spuštění, jako argument příkazu v příkazové řádce, bude mít vyšší prioritu než port nakonfigurovaný v aplikačních vlastnostech.

Po nastartování webového serveru je otevřen endpoint na adrese (Hodnoty mezi znaky < a > budou vyplněny skutečnými daty):

<adresa_webového_serveru>/pdf

To je adresa pro přístup k této službě. Endpoint očekává pracuje s http metodou GET, která má minimálně jeden povinný query parametr. Tímto povinným parametrem je parametr „url“. Tímto parametrem dojde k předání odkazu na stránku, která má být službou zpracována a převedena do PDF. Minimální požadavek by pak vypadal takto:

<adresa_webového_serveru>/pdf/?url=<url>

Endpoint obsahuje i další parametry, které jsou nepovinné. Je to řešeno tím způsobem, že mají nastavenu předdefinovanou hodnotu, která se použije v případě, kdy nejsou obdrženy v požadavku. Jedná se o parametry, které jsou v originální aplikaci použity k personalizaci a lokalizaci reportu o plagiátorství.

Kompletní GET požadavek pak může vypadat takto:

<adresa_webového_serveru>/pdf/?url=<url>&locale=<locale>&timezoneId=<timezone Id>&filename=<filename>

Příklad takového požadavku:

```
http://localhost:43925/pdf?url=https%3A%2F%2Fcs.wikipedia.org%2Fwiki%2F%C4%8Cesk%C3%A1_zem%C4%9Bd%C4%9Blsk%C3%A1_univerzita_v_Praze&fileName=soubor.pdf&locale=cs_CZ&timezoneId=UTC
```

Jelikož třída *PdfGenerationController* je anotována jako @RestController a obsahuje metodu *generatePdf()*, která nese anotaci @GetMapping, je zajištěno zpracování tohoto požadavku, rozdělení query parametrů a namapování na hodnoty jednotlivých proměnných, které jsou dále předány do třídy PdfGenerator metodě createPdf(). Ta je popsána níže, nicméně tato metoda zajistí vygenerování pdf souboru, který předá zpět na *PdfGenerationController*.

Pokud je tato operace úspěšná, endpoint vyrobí odpověď na požadavek. Do hlavičky odpovědi jsou přidány informace o typu obsahu, jménu souboru a typu odpovědi. To zajistí, že klientovi, který tento endpoint volá se vrátí již vytvořený soubor PDF.

Vlastní kód endpointu:

```
@GetMapping("/pdf")
fun generatePdf(
servletRequest: HttpServletRequest,
@RequestParam url: String,
@RequestParam locale: String = "en-US",
@RequestParam timezoneId: String = "UTC",
@RequestParam fileName: String = "pdfFile.pdf"
): ResponseEntity<ByteArrayResource> {
val generatedPdf: ByteArray?
try {
generatedPdf = PdfGenerator().createPdf(url, locale, timezoneId)
} catch (ex: Exception) {
println(ex)
return ResponseEntity.status(HttpServletResponse.SC_INTERNAL_SERVER_ERROR)
.body(ByteArrayResource("Error occurred while generating PDF.".toByteArray()))
 }

if (generatedPdf != null) {
val headers = HttpHeaders()
headers.add("Content-Disposition", "attachment; filename=$fileName")
headers.add("Content-Type", "application/pdf")
return ResponseEntity.ok()
.headers(headers)
.body(ByteArrayResource(generatedPdf))
 }
return ResponseEntity.status(HttpServletResponse.SC_INTERNAL_SERVER_ERROR)
.body(ByteArrayResource("Error occurred while generating PDF.".toByteArray()))
 }
}
```

Pokud cokoliv v průběhu generování PDF selže, klient, který službu volal dostane zpět odpověď s chybovým kódem 500.

4.3.3.2 PDF generátor

Ve třídě PdfGenerator se nachází metoda createPdf(). Právě tato metoda je zavolána webovým endpointem a dostává v parametrech předány hodnoty z požadavku.

Dojde k provolání knihovny Playwright, která v sobě nese ořezaný prohlížeč. V tomuto prohlížeči jsou pak předány informace z originálního požadavku a dojde k nastavení příslušných hodnot, které určují, jakým způsobem bude vykreslena stránka. Ta je poté načtena a pomocí zabudované funkcionality Chromium na generování PDF je vytvořen PDF soubor, který je následně vrácen endpointu, který jej zpracuje způsobem, který je popsán výše. Pokud při tomto zpracování dojde k chybě, je vyhozena výjimka, která je následně taktéž zpracována do odpovědi na požadavek.

Metoda na vytvoření PDF (vlastní zpracování):

```
fun createPdf(url: String, locale: String, timeZoneId: String): ByteArray? {
Playwright.create().use { playwright ->
val browser = playwright.chromium().launch()
val page = browser.newPage(
Browser.NewPageOptions().setIgnoreHTTPSErrors(true).setLocale(locale).setTimezoneId(timeZoneId
)
)
page.navigate(url)
return page.pdf(Page.PdfOptions().setDisplayHeaderFooter(true))
}
}
```

4.3.4 Nasazení

Tato služba je nasazována pomocí CDK s využitím jazyka Kotlin.
Nasazuje se přímo na instanci, která musí mít přichystané běhové prostředí pro JVM, nainstalovaný Tomcat a zajištěny veškeré potřebné konfigurace.

4.4 Serverless verze

Tato služba je napsána v jazyce Javascript a nasazena do cloudového prostředí AWS s využitím AWS Cloud Development Kit (CDK), což je infrastruktura jako kód (IaC) framework umožňující definovat cloudové zdroje pomocí známých programovacích jazyků. Klíčovou součástí architektury této služby je využití AWS Lambda, což je výpočetní služba, která umožňuje běh kódu bez potřeby spravovat servery. Tato architektura poskytuje

významné výhody v oblasti škálovatelnosti a efektivity nákladů, jelikož umožňuje automatické škálování v závislosti na množství příchozích požadavků, a zároveň umožňuje platbu pouze za skutečně využité výpočetní zdroje. Tento přístup eliminuje potřebu předem alokovat zdroje a spravovat fyzické nebo virtuální servery, což vede k redukci operačních nákladů a zjednodušení vývojového cyklu.

Využitím AWS CDK lze infrastrukturu služby definovat deklarativně v programovacím jazyce, což zvyšuje efektivitu vývoje díky možnosti využití existujících znalostí programovacího jazyka a nástrojů. CDK poskytuje bohatý soubor abstrakcí a komponent, které zjednodušují definici a konfiguraci zdrojů AWS včetně AWS Lambda funkcí. Nasazení a správa serverless aplikace, která je napsána v JavaScriptu a deployována pomocí AWS CDK do AWS Lambda, představuje velmi moderní a přístup k vývoji a provozu aplikací.

4.4.1 Kód aplikace

Nejdřív představme strukturu projektu:

```
├── app
│   ├── index.js
│   ├── lambda_source.zip
│   ├── node_modules
│   │   ├── lambdafs
│   │   ├── playwright-aws-lambda
│   │   └── playwright-core
│   ├── package-lock.json
│   └── package.json
├── deployment
│   └── pdfGenAppCdk
│       ├── README.md
│       ├── bin
│       ├── cdk.json
│       ├── cdk.out
│       ├── jest.config.js
│       ├── lib
│       ├── node_modules
│       ├── package-lock.json
│       ├── package.json
│       ├── test
│       └── tsconfig.json
└── out
    └── production
        └── pdfGenApp
```

Složka s názvem app představuje zdroj pro výkonný kód aplikace. Soubor index.js je soubor, který je načten při spuštění Lambda funkce a je spuštěna jeho funkce index.handler. To je funkce, která je spuštěna při každé jedné exekuci lambda funkce.

Struktura repozitáře je oproti OP verzi složitější s větším množstvím souborů. Vyplývá to zejména z odlišných filozofií použitého jazyka. Ve stejné složce se pak již klasicky nachází package.json, package-lock.json a složka node_modules, obsahující potřebné balíčky. Mírnou zvláštností je soubor ZIP. Ten vznikne po sestavení projektu, a právě ten je následně použit k nahrání kódu, kdy je rozbalen do prostředí Lambdy v cloudu.

Složka deployment pak v sobě nese soubory potřebné k fungování CDK. JavaScriptové soubory obsahující definici infrastruktury se pak nacházejí ve složkách bin a lib, kdy ve složce bin je iniciální soubor pro CDK, který volá pdf_gen_app_cdk-stack.js, který v sobě již obsahuje AWS zdroje.

4.4.2 Kód Lambdy

Na rozdíl od OP verze není potřeba nastavovat webový server a zabývat se programováním API endpointů. Níže, v sekci s vysvětlením CDK kódu bude tomuto věnována pozornost. Díky tomu odpadá značná část komplexity kódu a je možné se věnovat přímo logice.

```
const playwright = require('playwright-aws-lambda')

const {S3Client, PutObjectCommand} = require('@aws-sdk/client-s3')
const {requestPresigner, getSignedUrl} = require('@aws-sdk/s3-request-presigner')
const s3 = new S3Client()

exports.handler = async (event) => {
    let browser     let pdfOutput     const bucketName = ' pdfgeneratorbucketpdf'

console.log(event)


    try {
        let context         browser = await playwright.launchChromium({headless: false})
        context = await browser.newContext({
            ignoreHTTPSErrors: true, timezoneId: event.timezoneid,
        })

        const page = await context.newPage()
        await page.goto(event.url, {timeout: 10000})
        pdfOutput = await page.pdf({
            printBackground: true,
            scale: 1.2,
```

```
            margin: {top: "50px", bottom: "50px"},
            displayHeaderFooter: true,
            headerTemplate: '<div></div>',              footerTemplate: '<span
class="pageNumber" style="width:100%; text-align:center; font-size:7px;"/>'         })
    } finally {
        if (browser) {
            await browser.close()
        }
    }

    let pdfUrl

    try {
        const command = new PutObjectCommand({
            Bucket: bucketName, Key: event.fileName, Body: pdfOutput,
        })
        pdfUrl = await getSignedUrl(s3, command, {expiresIn: 3600})
    } catch (err) {
        console.error(`Uploading PDF to the bucket: "${bucketName}" failed`, err)
        throw err     }


    console.log(`Signed URL: ${pdfUrl}`)
    return {data: pdfUrl}
}
```

4.4.3 Deployment

Následující kód ukazuje, jak může vypadat popis stacku pomocí AWS CDK.
Tento stack se skládá ze dvou hlavních zdrojů. Je jimi S3 bucket, který slouží jako úložiště hotových dokumentů, které jsou poté nabízeny ke stažení. Dalším zdrojem je samotná Lambda funkce, které je nastaveno běhové prostředí, velikost operační paměti, a je do ní nahrán kód aplikace. Na konci je vytvořena tzv. function URL. Jedná se o veřejnou URL adresu, na kterou je možné přistoupit a tím spustit Lambda funkci. V tomto případě tímto nahradíme konfiguraci a nastavování webového http serveru, jak bylo zmíněno výše. Pomocí GET http požadavku na tuto adresu, se správnými query parametry dojde k předání těchto parametrů do vstupní události Lambda funkce. V těchto query parametrech je pak obsaženy metadata, včetně URL ze které je žádoucí vygenerování PDF.

V tomto demonstrativním případě je URL bez autentizace. To rozhodně není doporučený postup, jelikož je tím docíleno toho, že je Lambda funkce vystavená do internetu, a kdokoliv se znalostí tohoto URL ji může libovolně spouštět. Takovéto zneužití může vést k dramatickému nárustu finančních nákladů na AWS. Proto se doporučuje nikdy neotvírat podobné zdroje do internetu. V případě použití autentizačního typu NONE se předpokládá implementace zabezpečení ve vnitřní logice Lambda funkce. V tomto případě by bylo vhodné např. opatřit požadavek na Lambda funkci před-sdíleným autorizačním tokenem v hlavičce požadavku a na straně Lambda funkce implementovat validaci tohoto tokenu.
Ve skutečné implementaci této služby není nutné tento problém řešit, jelikož Lambda funkce nemá veřejnou URL a je spouštěna kódově z mateřské aplikace. Tato aplikace má pak patřičnou IAM roli, která zajišťuje oprávnění spustit tuto Lambdu. Taktéž se nachází ve stejném VPC, tudíž komunikace se službou na generování PDF probíhá po vnitřní sítí a pouze z rozsahu předem známých a nastavených IP adres.

Kód vlastního zpracování:

```
const cdk = require('aws-cdk-lib')
const s3 = require('aws-cdk-lib/aws-s3')
const lambda = require('aws-cdk-lib/aws-lambda')

class PdfGenAppCdkStack extends cdk.Stack {
  /**
   * @param {cdk.App} scope
   * @param {string} id
   * @param {cdk.StackProps=} props
   */   constructor(scope, id, props) {
    super(scope, id, props);

      new s3.Bucket(this, 'pdfGeneratorS3Bucket', {
          bucketName: 'pdfgeneratorbucketpdf',
          publicReadAccess: false,
      })

      new lambda.Function(this, 'pdfGeneratorFunction', {
          runtime: lambda.Runtime.NODEJS_16_X,
          handler: 'index.handler',
          memorySize: 128,
          code: lambda.Code.fromAsset('../../app/'),
      }).addFunctionUrl({authType: lambda.FunctionUrlAuthType.NONE})

  }
}

module.exports = { PdfGenAppCdkStack }
```

4.5 Kritéria pro srovnání a vyhodnocení

V této části je prezentováno komparativní hodnocení dvou přístupů k implementaci IT infrastruktury – on-premise a serverless architektury. Hodnocení je strukturováno s ohledem na pečlivě vybraná kritéria, která reflektují klíčové aspekty technických, ekonomických a výkonnostních faktorů. Cílem je poskytnout objektivní a vyvážený pohled na každé řešení, jehož relevantnost a aplikovatelnost jsou zkoumány v kontextu specifik a potřeb moderního podnikového prostředí.

Analýza zahrnuje nejen bezprostřední výhody a limitace obou modelů, ale i jejich dlouhodobý dopad na organizace, které je zavádějí. Pozornost je věnována také

uživatelskému prostředí a praktické správě, což jsou faktory, jež mohou významně ovlivnit finální rozhodnutí o nasazení.

Výsledkem bude shromáždění důkazů a závěrů, které budou sloužit jako základ pro formulaci doporučení určených pro potenciální aplikace v praxi. Závěry budou představovat hodnotné informace pro rozhodování o adopci jednoho nebo druhého řešení v rámci různých podnikatelských strategií a cílů.

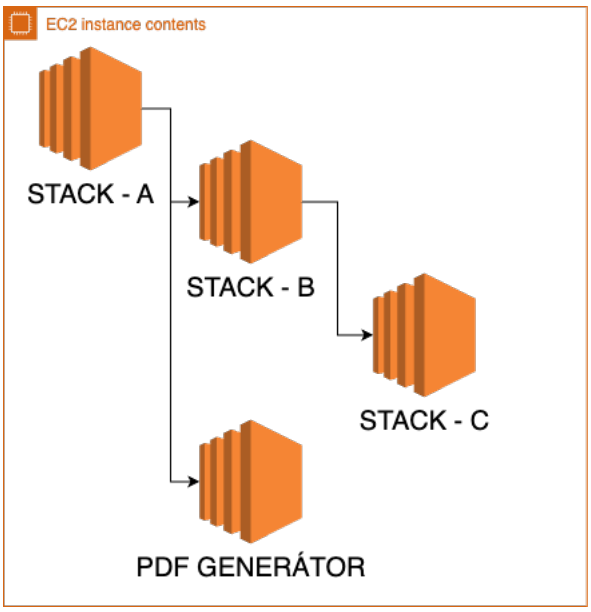
Konkrétní kritéria pro hodnocení:

1. **Technická specifikace a architektura**
    - Flexibilita a škálovatelnost řešení
    - Kompatibilita s existujícími systémy a technologiemi
2. **Ekonomická efektivita**
    - Celkové náklady na vlastnictví (Total Cost of Ownership, TCO)
    - Předpovědi a modelování nákladů (Cost Forecasting)
3. **Výkonnostní metriky**
    - Latence a doba odezvy aplikace
    - Odolnost vůči chybám a dostupnost služby (fault tolerance and availability)
4. **Uživatelská zkušenost a správa**
    - Uživatelské rozhraní a uživatelská přívětivost
    - Jednoduchost správy a údržby
5. **Integrační a vývojové možnosti**
    - Podpora pro kontinuální integraci a kontinuální doručování (CI/CD)
    - Rozsah a dostupnost vývojářských nástrojů a knihoven
6. **Rizika a návrh jejich řešení**
    - Potenciální rizika spojená s implementací a provozem.

4.5.1 Technická specifikace a architektura

4.5.1.1 OP

Při nasazování do systému, který je popsán v kapitole 3.1 je zvažována varianta OP nasazení z toho důvodu, že tento systém, ačkoliv je provozován kompletně v cloudovém prostředí, tak je provozován pomocí tzv. serverové farmy. Tudíž pomocí různých stacků jednotlivých EC2 instancí v AWS. Nabízí se tudíž možnost nasazení jako rozšíření této aplikace o další službu v podobě stacku EC2 instancí na generování PDF.



Obrázek 5 - Diagram EC2 stack (Vlastní zpracování)

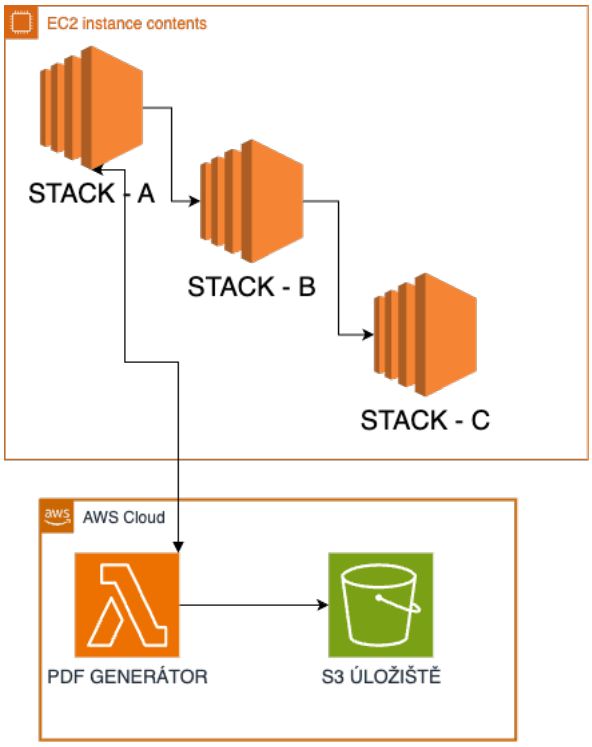
Takto nasazená aplikace je rozšířením již stávající infrastruktury a stávajícího systému. Je proto možné využít a aplikovat všechny existující a zavedené postupy a technologie pro správu Java aplikací, které jsou již provozovány na konkrétním systému. Takto navržené řešení snižuje nároky na údržbu a správu infrastruktury. Je nutné podotknout, že nijak nesnižuje nároky na údržbu, aktualizace a zabezpečení PDF generátoru samotného. V tomto řešení je stále nutné počítat se správou samotného softwarového prostředí serveru, na kterém je aplikace provozována. Umožňuje ale využití již zavedených a standardních postupů, které jsou aplikovány na ostatní stacky ve službě.

Kompatibilita tohoto řešení s ostatními je zajištěna použitím standartní http protokolu. Stack A na obrázku 5 s PDF Generátorem komunikuje pomocí http requestu a PDF je následně vráceno v odpovědi na tento http požadavek.

Toto řešení je horizontálně i vertikálně škálovatelné. Vertikálně pomocí nahrazení serverové instance za výkonnější, horizontálně pak pomocí umístění load balanceru před PDF generátor stack a zvýšením počtu instancí.

4.5.1.2 Serverless

Implementace serverless řešení do stejného systému je odlišná. Místo vlastního stacku se jedná o přidání AWS Lambda funkce.



Obrázek 6 - Diagram Lambda stacku (Vlastní zpracování)

V tomto případě odpadá velká část údržby infrastruktury. Tato zodpovědnost je přenesena na AWS. Pří vytvoření Lambdy je vybráno běhové prostředí, které je průběžně aktualizováno cloudovým poskytovatelem. Je nutné zdůraznit, že neodpadá nutnost zabezpečení takové aplikace.

Škálovatelnost je v tomto případě téměř neomezená a technologicky naráží pouze na technické limity zvoleného poskytovatele cloudových služeb. Velmi jednoduché je

zálohování vygenerovaných PDF s využitím služeb S3 bucketu. Díky tomuto řešení odpadají starosti o vertikální škálování, protože u Lambda funkce se konfiguruje pouze přidělená operační paměť a ta má vliv na rychlost a výkon zpracování jednoho požadavku, neovlivní však paralelní zpracování vícero požadavků. Toto horizontální škálování je pak řešeno spuštěním další instance Lambda funkce, se stejnými parametry, které je v režii AWS. Je možné ho konfigurací omezit.

Kompatibilita se stávajícím systémem je zde řešena stejně jako v OP verzi, tudíž za využití http požadavků. V odpovědi na takovýto požadavek se místo PDF vrací odkaz k jeho stažení z S3 úložiště. Tím odpadá náročné udržování hotového souboru v RAM.

4.5.2 Ekonomická efektivita

Ekonomická efektivita představuje klíčový faktor v rozhodovacím procesu při výběru vhodné infrastruktury. Tato kapitola poskytuje analýzu celkových nákladů na vlastnictví (TCO) a zkoumá metody pro předpovědi a modelování nákladů. Rozbor TCO zahrnuje nejen pořizovací cenu zařízení a softwaru, ale i dlouhodobé operativní náklady spojené s jejich používáním, údržbou a aktualizacemi. Porovnání TCO on-premise a serverless řešení odhalí finanční dynamiku a ukáže dlouhodobé ekonomické dopady každého přístupu.

V této části budou modelovány náklady na infrastrukturu pro různé scénáře využití služby pomocí nástroje AWS pricing calculator. (AWS, 2024)

Scénáře pro modelování nákladů vychází z odhadů využití služby implementované ve výše popsaném systému. Bude modelován scénář nízkého, obvyklého a extrémního využití. Jak je popsáno v předchozích kapitolách, služba, do které je zamýšlena implementace nemá konstantní rozložení využití a požadavků, ale požadavky přichází obvykle ve špičkách. Modely ukazují měsíční náklady na provoz služby (Nezahrnují náklady na lidský kapitál, vybavení vývojářů atd.)

První varianta scénáře bude obsahovat 500 požadavků na PDF denně po dobu jednoho měsíce. Druhá varianta, též střední, obsahuje 3000 požadavků denně a následně třetí varianta s dotací 10 000 požadavků.

Všechny následující výpočtu jsou platné k měsíci Březnu roku 2024. V této kapitole se jedná o shrnutí výsledků. Veškeré výpočty jsou přiloženy k této práci ve formátu JSON.

4.5.2.1 OP

Jelikož v prezentovaném systému dochází k zátěži ve špičkách, a to jak během dne, tak během měsíce, je obtížené rezervovat instance na dlouhou dobu dopředu. EC2 instance musí být připraveny v ASG[18] k okamžitému nasazení při zvýšené zátěži.
Při výpočtu uvažujeme o instanci r5a.4xlarge, která nabízí 16 jader CPU, 128GB RAM a síťovou kartu až 10 Gbit/s. Toto bude jednotka instance, která má standardní hodinovou cenu 0,958 USD.

Je nutné vzít v potaz, že tento výpočet není konečný a jedná se o kvalifikovaný odhad. Při nasazení služby do provozu je třeba kontinuální monitoring a empirické a kontinuální upravování parametrů s ohledem na rozdíl mezi předpokládanou (odhadovanou) a skutečnou zátěží na službu.

4.5.2.1.1 Nízká zátěž

Nízká zátěž počítá s 500 požadavky na PDF denně. V tomto scénáři je počítáno s jednou modelovou instancí. Ta zabezpečí dostupnost kdykoliv, ale neškáluje.

V této variantě s jednou instancí uvažujeme o rezervované EC2 instanci, s rezervací na jeden rok. Tyto instance jsou obecně levnější než instance na vyžádání.

V tomto modelu je celková cena odhadována na **416,10 USD**, dle aktuálního kurzu 9 761,21 Kč.

4.5.2.1.2 Střední zátěž

1 instance zvládne provoz paralelně několika aplikací na generování PDF. V tomto scénáři je v ASG minimum 1 instance a maximálně 5 při špičkovém využití. Průměrný denní provoz

[18] Auto scaling group

čítá 3000 požadavků. Neočekávaně nadstandardní provoz při OP variantě povede k dlouhé prodlevě při generování PDF, případně time-outu požadavků.

Při této koncepci vychází měsíční náklady na **792,72 USD**. Převedeno aktuálním středním kurzem se jedná o 18,592.45 Kč měsíčně.

4.5.2.1.3 Extrémní zátěž

V extrémní zátěži je počítáno s 10 000 požadavky denně. Ke škálování je tedy třeba minimálně 1 a maximálně X instancí.
Pokud by platila přímá úměra dostaneme rovnici:

$$x = \frac{I1 * P2}{P1}$$

Rovnice 1- Škálování při extrémní zátěži (Vlastní zpracování)

Kdy I1 je počet maximálních instancí při střední zátěži, P2 je počet odhadovaný počet požadavků při extrémní zátěži a P1 je počet požadavků při střední zátěži.
Dostáváme:

$$x = \frac{5 * 10\,000}{3000}$$

Tato rovnice vychází z předpokladu použití 5 instancí při škálování pro střední zátěž a předpokládá platnost přímé úměry, jelikož je extrémně obtížně odhadnout přesný potřebný výpočetní výkon pro variabilní zátěž a variabilní velikost generovaných souborů.
Z této rovnice plyne, že X = 16,7. Za tohoto předpokladu by bylo třeba ve špičce přesně 16,7 instancí.

Jelikož ale přímá úměra v tomto případě neplatí a požadavky jsou rozprostřeny nerovnoměrně, dává nám tato rovnice podklad k odhadnutí počtu instancí tak, aby byla zachován poměr mezi kvalitou služby a její cenou.

Špička nastává obvykle ve dnech po odevzdávání většího množství studentských prací. K tomuto dochází zejména ve dnech před konci měsíce. Vzhledem k tomuto rozložení bude počítána cena pro 70% špičkových instancí X, tedy ~ 12.

S takto stanovenou koncepcí dostáváme cenový odhad **1 307,16** USD, tedy 30,658.13 Kč.

4.5.2.2 Serverless

U serverless Lambdy je škálování značně efektivnější a jednodušší a taktéž cenový odhad, protože se platí pouze za skutečně spotřebovaný čas běžící Lambda funkcí a nikoliv za tzv. stale time (Tedy čas, kdy funkce nevykonává žádnou činnost, ale existuje a je k dispozici).

Jelikož tato architektura služby nedrží vygenerované soubory v paměti a ve streamu nevrací klientovi, nýbrž ukládá výsledky do S3 úložiště, je nutné ke každé ceně připočítat cenu za S3 bucket. Jelikož budou rozdíly ve velikosti dat zanedbatelné z hlediska ceny, bude připočtena částka za úložiště pro nejdražší scénář.

Lambda funkce je konfigurována s x86 architekturou běhového prostředí, 2 GB RAM a průměrnou délkou běhu 15s.

V lambda funkci je nutné zohlednit „Provisioned concurrency“, tedy kolik instancí dané funkce je přichystáno a nastartováno k okamžitému zpracování požadavku. Toto řešení výrazně snižuje odezvu a eliminuje problém zvaný „cold start“. V nízké zátěži se s tímto nepočítá. Ostatní varianty pak mají přichystáno 10 instancí funkce. Vzhledem k povaze aplikace, kdy Lambda funkce generuje PDF soubor, což je samo o sobě časově náročná činnost, není tato metrika klíčová pro kvalitu navrhovaného systému. Jinak by tomu bylo v případě, pokud by se jednalo o funkce s nutností velmi rychlé odezvy, jako je např. odeslání push notifikace aj.

4.5.2.2.1 Nízká zátěž

Při 500 požadavcích denně s výše definovanou konfigurací lambdy je výsledkem zanedbatelná cena **7,6 USD**. Převedeno na koruny to je 178 Kč měsíčně.

4.5.2.2.2 Střední zátěž

Při stejných parametrech lambda funkce, ale s použitím provisioned concurrency na 50 instancí funkce, které jsou k dispozici ve špičkovém čase vychází cena na **90,66** USD měsíčně, což je 2126 Kč.

4.5.2.2.3 Vysoká zátěž

Náklady na vysokou zátěž s použitím provisioned concurrency činní **197.17** USD, tedy 4 624,1 Kč. I před připočtením nákladů na úložiště je patrné, že se jedná z hlediska ekonomické efektivity o cenově výrazně výhodnější variantu ve všech jednotlivých scénářích.

4.5.2.2.4 S3 úložiště

S3 úložiště pro vygenerované PDF ve standardní konfiguraci s předpokladem minimálně 70 GB dat do internetu (tedy stažení PDF koncovým uživatelem) přidává v tomto modelovém případě ke každé Lambdě cenu **7.96** USD neboli 186.69 Kč.

4.5.2.3 Výkonostní metriky

Latence a doba odezvy aplikace jsou klíčovými metrikami pro hodnocení výkonnosti jak on-premise, tak serverless architektur.
V případě této aplikace je teoreticky možné dosahovat nižší latence za použití dedikovaných EC2 instancí. To právě díky tomu, že odpadá nutnost řešení cold startů, tedy inicializace aplikace jako takové.
V modelové aplikaci by se snadno mohlo stát, že dojde k vyčerpání dostupné operační paměti, pokud by bylo do služby posláno více požadavků, než je schopná ve stejném čase procesovat. Aby se předešlo tomuto problému, lze uvažovat o implementaci tzv. poolingu. V tomto případě browser poolingu, jelikož virtuální prohlížeč je ta část, kde dochází ke generování PDF. Browser pooling je podrobněji rozebrán níže.

Na druhé straně, serverless architektury, kde se výpočetní zdroje automaticky škálují v reakci na požadavky aplikace, mohou vykazovat variabilní latenci. To je způsobeno "cold start" problémy, když je funkce spouštěna po periodě nečinnosti, což vyžaduje čas na inicializaci prostředí. Avšak, v případě konzistentního provozu a optimalizace lze tyto efekty minimalizovat, a serverless architektura pak může poskytovat srovnatelnou, ne-li lepší dobu odezvy.
Pokud je třeba snížit latenci ještě více, je možné velmi jednoduše tuto službu rozšířit do více AWS regionů, tudíž ji fyzicky dostat blíže k uživatelům.

V generátoru PDF, popisovaném v této práci, však není nízká doba odezvy primární metrikou. Důležitější je čas, za který je PDF vygenerováno, a nabídnuto ke stažení koncovému uživateli. To se výrazně liší velikostí požadovaného souboru, kterou můžeme jen velmi těžko ovlivnit. Proces generování může být významně ovlivněn velikostí disponibilní paměti RAM. Ta je velmi jednoduše konfigurovatelná v prostředí serverless.
V OP prostředí to vyžaduje nasazení celé nové instance EC2.

Hledisko dostupnosti služby je v tomto případě obdobné, jelikož OP varianta běží též na cloudu AWS, tudíž SLA je definováno poskytovatelem cloudu. V případě aplikace, která by vyžadovala kritickou kontrolu nad celým systémem by bylo nutné zvažovat implementaci a nasazená vlastní infrastruktury, nikoliv cloudové.

4.5.2.4 Uživatelská zkušenost a správa

Jedním z důležitých faktorů, které je nutné zvažovat při implementaci služby v podnikovém prostředí je i zkušenost zaměstnanců nebo společnosti s konkrétními technologiemi. Pokud se nabízí více vhodných technologických alternativ, je nutné přihlédnout i k faktoru znalosti a zkušenosti implementačního týmu. To může velmi pozitivně ovlivnit náklady na vývoj a provoz aplikace.

OP řešení může organizacím umožnit větší kontrolu, ale také vyžaduje, aby měly vlastní tým schopný spravovat a podporovat infrastrukturu. To zahrnuje pravidelné aktualizace, patch management, zálohování a obnovu, což může být nákladné a časově náročné.
Serverless architektura může tyto úkoly zjednodušit tím, že řadu správních povinností přesune na cloudového poskytovatele. Tím se redukuje potřeba věnovat čas a zdroje rutinním úkolům, což umožňuje personálu soustředit se na vývoj a inovace. Přestože může dojít k omezení kontroly, serverless model nabízí dynamiku a flexibilit.

Výběr mezi on-premise a serverless řešeními by měl být vedle technických a ekonomických aspektů zvážen i s ohledem na uživatelskou zkušenost a potřeby správy. Volba by měla být založena na důkladné analýze současných a budoucích potřeb organizace, schopnostích týmu a očekávaných trendech ve vývoji softwaru a technologií.

4.5.2.5 Integrační a vývojové možnosti

OP infrastruktury mohou představovat výzvy v rychlosti a pružnosti kontinuální integrace a doručování kvůli potřebě konfigurace a údržby serverů. Naproti tomu serverless architektury nabízí inherentní výhody pro CI/CD díky své vysoké škálovatelnosti a flexibilitě, které jsou základem cloudových služeb. Bez nutnosti správy serverů mohou týmy rychle iterovat a nasazovat kód, což je ideální pro agilní vývojové prostředí.
V případě on-premise řešení závisí dostupnost vývojářských nástrojů a knihoven na interních zdrojích organizace a jejích IT politikách. Serverless platformy, jako je AWS Lambda, poskytují široký výběr integrovaných nástrojů a knihoven, které jsou neustále aktualizovány a spravovány poskytovatelem cloudových služeb. Tato bezproblémová integrace s širokou paletou cloudových služeb může značně urychlit vývoj a snížit čas potřebný pro uvedení aplikace na trh.

V OP prostředích mají organizace plnou kontrolu nad instalovanými nástroji a verzemi knihoven, což umožňuje vývojářům plně přizpůsobit vývojové prostředí specifickým potřebám projektu. Tato flexibilita vede k možnosti využití speciálních nástrojů, které jsou možná pro cloudové služby nedostupné nebo neslučitelné. Nicméně, s touto kontrolou přichází také odpovědnost za aktualizaci, konfiguraci a údržbu těchto nástrojů, což může být časově náročné.

Serverless architektury nabízejí integraci s širokým ekosystémem cloudových služeb, které často zahrnují nejnovější vývojářské nástroje a knihovny dostupné "out-of-the-box". Poskytovatelé jako AWS, Azure, nebo Google Cloud pravidelně aktualizují své prostředí, aby zajistili vývojářům přístup k nejmodernějším technologiím. Dále, serverless platformy mají obvykle integraci s řadou CI/CD nástrojů, které automatizují testování a nasazení, což zvyšuje rychlost vývoje a zjednodušuje správu verze a závislostí. Tato vysoká míra automatizace a integrace může výrazně zrychlit vývojové cykly a snížit potřebu technické údržby. On-premise řešení mohou vyžadovat nastavení vlastních Git serverů nebo integraci s externími službami, zatímco serverless platformy často přicházejí s předkonfigurovanými službami, které se přímo propojují s cloudovým prostředím. Na závěr, při rozhodování o vývojářských nástrojích a knihovnách by měly být vzaty v úvahu celkové náklady na správu

a provoz, možnost rozšiřování a integrace s ostatními službami, a jak tyto aspekty ovlivňují celkový čas a náklady na vývoj softwaru.
Serverless architektura může nabídnout rychlou a efektivní cestu k vývoji, zatímco on-premise řešení může být lepší volbou pro organizace, které potřebují větší kontrolu a specifické konfigurace.
V této práci je tento rozdíl ve verzích patrný z části, která popisuje kód a nasazení. Serverless architektura v tomto případě velmi zjednodušuje jak kód aplikace, tak její nasazení.

4.5.2.6 Rizika a návrh řešení některých

Identifikace a mitigace rizik jsou kritickými aspekty správy IT infrastruktury, zvláště v kontextu implementace a běžného provozu. Tato kapitola je zaměřena na rozpoznání specifických výzev, kterým mohou čelit on-premise a serverless architektury, a představuje strategie pro jejich řešení. Průzkum těchto rizik je klíčový pro zajištění stability a bezpečnosti systému.

4.5.2.6.1 OP rizika

U OP řešení, jedním z klíčových rizik je vytížení webového serveru, které může být způsobeno náhlým nárůstem provozu nebo distribuovaným útokem odmítnutí služby (DDoS[19]). K mitigačním opatřením patří použití technik browser pooling, což znamená správu skupiny prohlížečů, které jsou připraveny na zpracování příchozích požadavků. Tato metoda pomáhá regulovat zatížení serveru a udržovat rychlost odezvy.

Navrhovaná řešení:

- **Zavedení Load Balancing**:

  Distribuce požadavků mezi více serverů pro vyrovnání zatížení. AWS nabízí více druhů load-balancerů, s velkým množstvím dostupných konfigurací.

- **Implementace Rate Limiting**:

  Omezení počtu požadavků, které může jednotlivý uživatel nebo IP adresa provést v

[19] Distributed Denial of Service

daném časovém okně. V tomto případě není možné využít IP adresu, jelikož koncovou službu volá “mateřská aplikace”. Ta ale zná přihlášeného uživatele, proto je možností implementovat omezení na úrovni uživatelského účtu. V aktuální podobě toto v aplikaci implementováno není a je to tudíž jedním z doporučení pro budoucí vylepšení aplikace, viz. závěr práce.

- **Vybudování Redundantní Infrastruktury**:

Vytvoření záložních systémů, které mohou převzít provoz v případě selhání primárního systému.

4.5.2.6.2 Serverless

Serverless architektura, například AWS Lambda, čelí riziku přetížení při vysoké konkurenci funkcí, které mohou být také cílem DDoS útoků. AWS Lambda a podobné platformy obvykle nabízí nástroje pro nastavení limitů konkurence (concurency), které mohou pomoci předcházet těmto problémům a chránit systém před přetížením.

Navrhovaná řešení:

- **Konfigurace Concurrency Limits**:

  Nastavení limitů pro současně běžící instance funkce k omezení zátěže na infrastrukturu.

- **Implementace API Gateway**:

  Využití API brány, která může poskytnout další vrstvu ochrany proti nadměrnému provozu a zneužití. API Gateway umožňuje analyzovat a řídit příchozí požadavky efektivněji, než běžný HTTP server.

Pro obě architektury je také klíčové mít robustní monitorovací a upozorňovací systémy, které mohou detekovat a reagovat na anomální chování v reálném čase. Kombinací preventivních

a reaktivních strategií může organizace efektivně zvládnout potenciální rizika a zabezpečit bezpečnost a dostupnost svých služeb.

4.5.2.6.3 Technika Browser pooling

V této části bude navržena možná implementace techniky browser pooling pro OP architekturu za využití open source projektu Apache Commons Pool (Apache Commons, 2024).

Browser pooling je technika používaná v softwarovém vývoji a web scrapingu, kde je předem nastavena skupina webových prohlížečů nebo jejich instancí na serveru, které jsou připraveny zpracovat příchozí požadavky nebo úlohy. Tento přístup může efektivně snižovat zátěž a latenci, zvláště při vysokém počtu uživatelských požadavků, a zvyšuje odolnost proti výpadkům a kolísání výkonu.

Implementace začíná importováním zvolené poolingové knihovny do projektu.

```
implementation group: 'org.apache.commons', name: 'commons-pool2', version: '2.12.0'
```

Následně je vytvořen konfigurační soubor *pdf-generator.properties* ve složce *resources*

Ten může vypadat následovně:

```
print.report.poolConfig.maxTotal=1
print.report.poolConfig.minIdle=0
print.report.poolConfig.maxIdle=1
print.report.poolConfig.maxWait=20000
```

Hodnoty, které jsou zde uváděný jsou pouze výchozí a při skutečném nasazení by měly být změněny pomocí CI/CD nástroje při sestavení aplikace tak, aby odpovídaly prostředí, do kterého je nasazováno.

Jak je patrné z názvů položek, určují maximální celkový počet prohlížečů, které mohou být vytvořeny, minimální a maximální počet „idle“ (tedy čekajících) prohlížečů a maximální dobu, po kterou aplikace bude čekat na uvolnění prohlížeče.

Tyto položky je následně nutné dostat do aplikace. Nabízí se využití konfiguračních Bean z frameworku Spring. To ilustruje následující kód:

```
import org.springframework.beans.factory.annotation.Value
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.PropertySource

@Configuration @PropertySource("classpath:pdf-generatorproperties", ignoreResourceNotFound =
false)
class ReportPrinterConfiguration {

  @Bean
  fun maxTotal(@Value("\${print.report.poolConfig.maxTotal}") maxTotal: Int?): Int? {
    return maxTotal
 }

  @Bean
  fun minIdle(@Value("\${print.report.poolConfig.minIdle}") minIdle: Int?): Int? {
    return minIdle
 }

  @Bean
  fun maxIdle(@Value("\${print.report.poolConfig.maxIdle}") maxIdle: Int?): Int? {
    return maxIdle
 }

  @Bean
  fun maxWait(@Value("\${print.report.poolConfig.maxWait}") maxWait: Long?): Long? {
    return maxWait
 }

}
```

Následně je potřeba dostat tyto konfigurace ke třídě, která zajišťuje vygenerování PDF.

```
import cz.bechny.pdfgenerator.configuration.PdfGeneratorConfig
...
@Autowired
fun poolConfigMinIdle(minIdle: Int) {
  this.browserPoolMinIdle = minIdle
}
```

Následně je vytvořen pool prohlížečů za pomocí těchto konfigurací:

```
  @PostConstruct
  fun init() {
    val poolConfig: GenericObjectPoolConfig<Browser?> = GenericObjectPoolConfig<Browser?>()
    //Includes both active and idle Browsers (it is the sum total of Browsers)
    poolConfig.maxTotal = browserPoolMaxTotal
    //The minimum number of created Browsers that should be kept in the pool at all times.
    poolConfig.minIdle = browserPoolMinIdle
    //The maximum idle Browsers (maxIdle) are Browsers that are ready to be used (but are
currently unused).
    poolConfig.maxIdle = browserPoolMaxIdle
    //Maximum wait time for getting browser instance in millis
    poolConfig.setMaxWait(Duration.ofMillis(browserPoolMaxWait))
    browserPool = GenericObjectPool(BrowserFactory(), poolConfig)
}
```

Tato třída musí samozřejmě nejdříve inicializovat tyto konstanty:

```
  lateinit var browserPool: ObjectPool<Browser>
  private var browserPoolMaxTotal = 0
  private var browserPoolMinIdle = 0
  private var browserPoolMaxIdle = 0
  private var browserPoolMaxWait: Long = 0
```

Tím je hotovo, pro konkrétní vytvoření PDF se již neinstancuje celý prohlížeč při volání, ale zapůjčuje se již předvytvořený k tomuto účelu z poolu.

```
val browser: Browser? = browserPool.borrowObject()
```

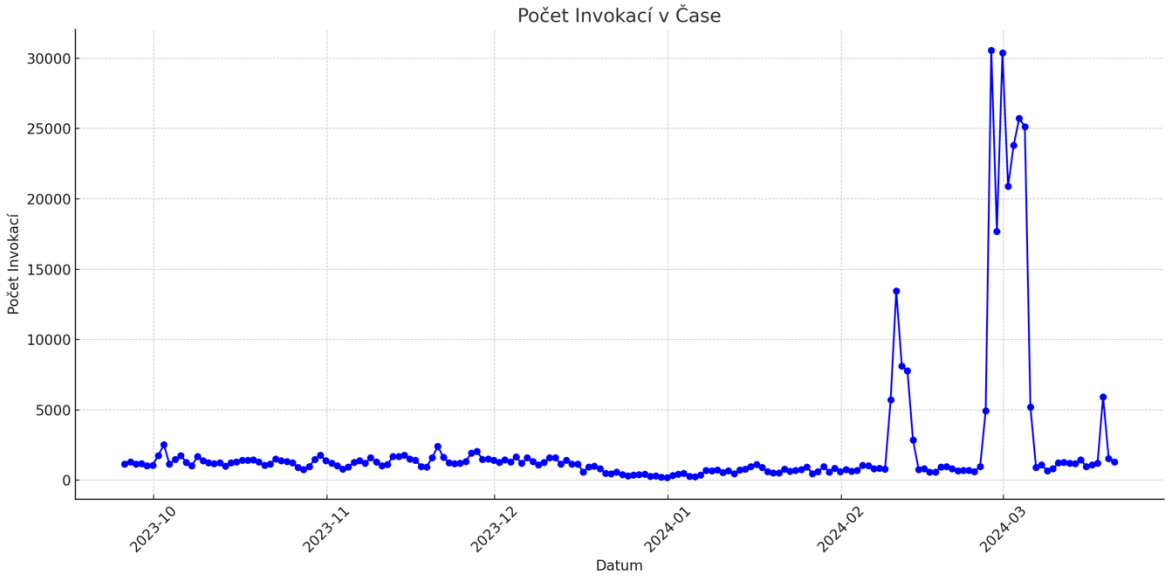
Následně pokračuje exekuce stejným způsobem, jako předtím. Tato technika vyžaduje implementaci a udržování další knihovny a zvyšuje komplexitu aplikace. V serverless variantě toto není vůbec potřeba, jelikož každá Lambda funkce má vlastní browser, a běží v odděleném prostředí. Při zpracování více požadavků je vytvořena další instance též funkce.

Výsledky a diskuse

4.6 Analýza požadavků v čase

Následující kapitola ukazuje distribuci požadavků ze skutečného systému, ve kterém je obdobná služba, jako ta popisována v této práci, implementována. Tato data pochází ze služby SafeAssign™ od společnosti Anthology.

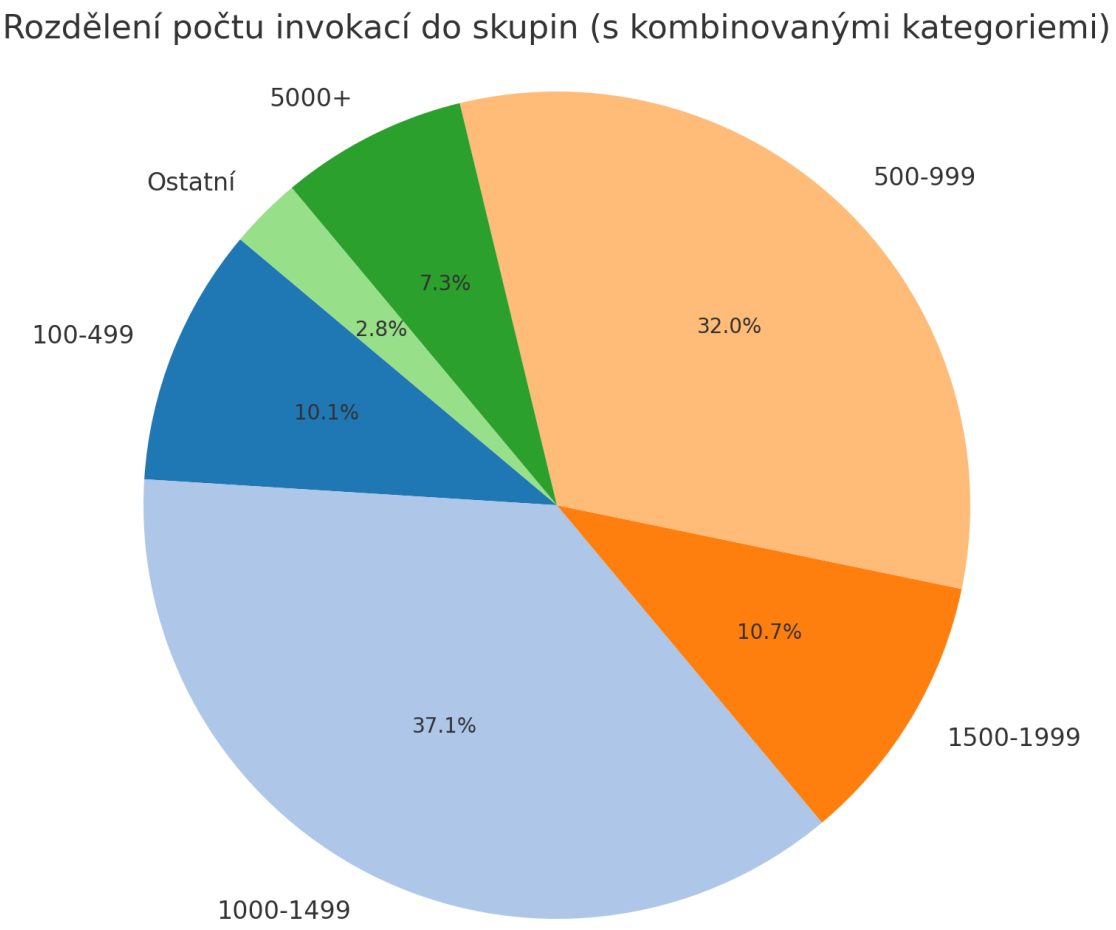
Tento graf ukazuje vývoj počtu dotazů na PDF během školního semestru za jednotlivé dny.



Obrázek 7- Graf počtu dotazů za semestr (vlastní zpracování)

Z grafu je patrné, že je potřeba počítat se špičkovým provozem ke konci akademického semestru a tomu přizpůsobit možnosti škálování infrastruktury navrhované aplikace.

Pro zjištění standardního provozu byla data kategorizována a rozdělena do skupin dle počtu požadavků. Tento způsob dává informaci o tom, kolik požadavků na PDF služba zpracovává nejčastěji.



Obrázek 8 - Graf s kategorizovanými požadavky (Vlastní zpracování)

Nejčastěji je tedy denní požadavek na 500–1500 PDF souborů denně.
Z těchto dat jednoznačně vyplývá, že při zvolení tradiční OP varianty bude nutné velmi často náročně škálovat a zároveň platit strojový čas, který bude ve velké míře nevyužitý.

Pokud by kapacita serverů v OP variantě měla být využitá, bylo by nutné aplikační prostředí generátoru PDF virtuálně oddělit, např. pomocí technologie Docker a využít dostupný výpočetní výkon na provoz jiné aplikace či služby. Avšak tuto variantu nelze doporučit, jelikož výrazně zvyšuje komplexitu tohoto řešení a mohla by negativně ovlivnit stabilitu a chybovost PDF generátoru, například při přetížení serveru.

4.6.1 Výkonnost

Aplikace byla testována lokálně, byla generována stránka Wikipedie o ČZU dostupná na:

https://cs.wikipedia.org/wiki/%C4%8Cesk%C3%A1_zem%C4%9Bd%C4%9Blsk%C3%A1_univerzita_v_Praze (březen, 2024).

Na obě verze aplikace byl zaslán 10x stejný GET požadavek na vygenerování PDF. Pomocí programu Insomnia pak byla měřena doba potřebná na zpracování požadavku.

```
curl --request GET \
  --url
'http://localhost:43925/pdf?url=https%3A%2F%2Fcs.wikipedia.org%2Fwiki%2F%25C4%258Cesk%25C3%25A1_zem%25C4%259Bd%25C4%259Blsk%25C3%25A1_univerzita_v_Praze&fileName=soubor.pdf&locale=cs_CZ&timezoneId=UTC' \
  --header 'User-Agent: insomnia/8.6.1'
```

Výsledky měření:

http GET	**OP**	**Lambda (1 GB)**	***Rozdíl (s)***
1	1.8	2.58	*0.78*
2	1.82	2.61	*0.79*
3	1.75	2.51	*0.76*
4	1.85	2.51	*0.66*
5	1.73	2.48	*0.75*
6	1.7	2.5	*0.8*
7	1.71	2.56	*0.85*
8	1.8	2.67	*0.87*
9	1.75	2.43	*0.68*
10	1.81	2.4	*0.59*
	17.72	**25.25**	***7.53***

Tabulka 1- OP vs Lambda 1 GB (vlastní zpracování)

Dle předpokladu vykázala OP verze aplikace v měření lepší výsledky. Zpracování bylo v průměru o 0,75 sekundy rychlejší. To je v průměru o 30% lepší rychlost zpracování. Nelze vynechat informaci, že OP verze vrací PDF jako soubor přímo ke stažení, zatímco odpovědí lambdy je časově omezené URL ke stažení souboru. V tomto testu byla využita Lambda funkce s RAM pamětí 1 GB.

Test bude opakován s Lambdou, s operační pamětí 2 GB.

http GET	**OP**	**Lambda (1 GB)**	***Rozdíl (s)***
1	1,80	1,50	*-0,30*
2	1,82	1,55	*-0,27*
3	1,75	1,49	*-0,26*
4	1,85	1,46	*-0,39*
5	1,73	1,46	*-0,27*
6	1,70	1,54	*-0,16*
7	1,71	1,77	*0,06*
8	1,80	1,46	*-0,34*
9	1,75	1,78	*0,03*
10	1,81	1,50	*-0,31*
	17,72	**15,51**	***-2,21***

Tabulka 2 - OP vs Lambda 2 GB (vlastní zpracování)

Výsledky potvrzují hypotézu, že rychlost zpracování požadavku Lambda funkcí je ovlivněna velikostí dostupné operační paměti, dokud ji prohlížeč běžící ve funkci dokáže využít. V druhém testu vykázala Lambda funkce v průměru o 0,22 vteřin lepší rychlost zpracování požadavku. To je o 14,2% lepší výsledek než OP verze aplikace.
Tento test zanedbává cold start problém, a využívá již instanciovanou funkci. V případě cold startu tak bude první požadavek zpracováván déle než vykazuje měření.

4.6.2 Doporučení pro implementaci

Při porovnání dvou různých architektonických přístupů k vývoji aplikace, kde jeden byl založen na Apache Tomcatu provozovaném na EC2 instancích a druhý využíval serverless model s AWS Lambdou, byly identifikovány klíčové rozdíly s významnými implikacemi pro implementaci.
AWS Lambda ukázala výrazné přednosti v oblastech škálovatelnosti, nákladové efektivity a údržby. Tyto faktory byly stanoveny jako klíčové pro implementaci aplikace, zejména cena hraje významnou roli. S touto serverless službou jsou náklady úzce spojeny s reálným využitím, na rozdíl od tradičního EC2 modelu, kde je nutné platit za instance neustále, i když nejsou využívány. Serverless architektura díky automatickému škálování rychle reaguje na

změny v zátěži, což eliminuje potřebu manuálního škálování a umožňuje aplikaci efektivněji reagovat na požadavky uživatelů.
OP architektura může v některých případech nabídnout nižší latenci, což je proto, že odpadá problém s „cold starty“. Nicméně v kontextu zkoumaného projektu, který nespočívá na kritických real-time požadavcích, je tato výhoda OP architektury považována za relativně nevýznamnou. V této práci bylo prokázáno, že jakékoliv potenciální výhody OP architektury shazuje dramatický cenový rozdíl oproti využití serverless varianty.

Na základě výše uvedených zjištění je doporučeno přijmout serverless architekturu pro tento konkrétní projekt a obdobné projekty s podobnými potřebami. Přestože může existovat zvyková překážka při přechodu od známého prostředí EC2 k serverless paradigmatu, přínosy serverless architektury, jako jsou nižší provozní náklady, menší nároky na správu a vysoká škálovatelnost, převažují nad potenciálními výhodami EC2 pro tuto kategorii aplikací. Zvolení serverless řešení přinese větší agilitu a efektivitu při řízení zdrojů a snížení celkových vlastnických nákladů.

4.7 Návrh možných rozšíření a vylepšení

V aplikaci zůstává prostor pro upgrade a zlepšování funkcionality. Služba navržená a popsaná v této práci představuje první produkční verzi, kdy bylo nutné zodpovědět, podložit a rozhodnout zásadní otázky pro provoz a funkci aplikace jako takové. Zvolené serverless řešení je navrženo tak, že skýtá prostor pro jednoduché rozšíření funkcionality. Pro OP variantu je výrazné zlepšení popsáno v kapitole 3.5.2.6.3, následující návrhy, ačkoliv mohou být aplikovatelné pro obě varianty, se budou vztahovat k serverless variantě implementace, jelikož byla vyhodnocena jako výhodnější a lepší, a proto byla zvolena.

4.7.1 Caching

Aby bylo možné ještě více snížit zátěž na generování PDF, bylo by vhodné implementovat způsob mechanismu cache. To s sebou přináší několik výzev, jako zajištění invalidace cache, vracení aktuálního obsahu a jiné. Ve konkrétním prostředí, kde je tato aplikace v produkci implementována a generuje PDF reporty o plagiátorství, tak je nutné na základě dotazu na

konkrétní report porovnávat, kdy došlo k jeho změně vůči tomu, kdy pro něj bylo vytvořeno PDF, pokud vůbec. To by umožnilo servírovat uživateli PDF přímo z S3 bucketu a vyhnout se tak invokaci Lambda funkce pro generování.

Tento přístup může přinést vyšší nároku na úložiště, zejména cenu S3 bucketu, proto by měl být doplněn o mazání starých neaktuálních reportů z trvalého úložiště. To vychází z předpokladu, že reporty jsou validní, a tedy potřebné pouze po omezený časový úsek, tedy například po dobu řízení o plagiátorství, do obhájení závěrečných prací a jiné. I pokud budou již hotové PDF odstraněny z úložiště a bude obdržen požadavek, pouze dojde k vygenerování standardní cestou.
Toto řešení eliminuje i zahlcení aplikace pomocí dotazů například z UI, kdy se nevaliduje, jestli již byl konkrétní požadavek zpracován či nikoliv.

4.7.2 Velké soubory

S takto navrženou aplikací se může stát, že pokud požadované PDF bude příliš velké, nemusí dostačovat kapacita RAM Lambda funkce. Paměť jako takovou je velmi jednoduše možné navýšit až do velikosti 10240 MB (AWS, 2024), to ale nese i navýšení ceny služby. Taktéž je možné narazit na strop, kdy už nebude možné RAM zvyšovat, ale generování příliš velkých souborů bude stále pomalé.

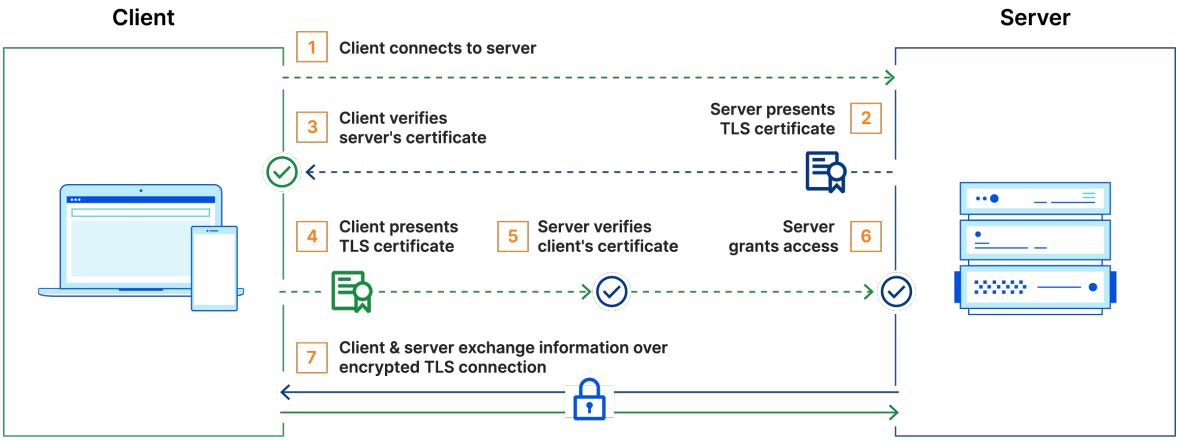
Jako řešení je proto možné stanovit FUP[20], kdy aplikace ještě před tím, než zahájí samotné generování PDF souboru zjistí velikost požadovaného obsahu a pokud bude přesahovat předem stanovenou velikost, požadavek odmítne.

4.7.3 Podepsání požadavků

Aktuální řešení opírá svou bezpečnost o uzavřený ekosystém a přenáší zabezpečení na mateřskou aplikaci. Pokud by ale došlo k MITM[21] útoku, spoofingu nebo by byla kompromitována mateřská aplikace, je generátor PDF nezabezpečen. Návrh na řešení tohoto problému pak obsahuje využití mechanismů service-to-service autorizace. V tomto případě

[20] Fair User Policy
[21] Man-in-the-middle

je možné využití před sdíleného tajemství, jako například OTP[22] nebo implementací mTLS[23].



Obrázek 9 - mTLS (Cloudflare, 2024)

[22] One-time-password
[23] Mutual Tansport Layer Security

5 Závěr

Tato práce se zaměřuje na strategický výběr architektury pro aplikaci. Zahrnuje komplexní teoretický základ, empirickou analýzu a praktický vývoj, měření výsledků v konkrétním prostředí a jejich analýzu.

To společně vedou k informovanému rozhodnutí o implementaci, které je podpořeno taktéž kalkulací nákladů na pořízení a provoz této aplikace.

Práce začíná rozsáhlou literární rešerší, která se zabývá současnými technologickými trendy a jejich aplikací v reálném světě, poskytuje podklady pro zvolení optimálního řešení a stanovuje zásadní kritéria pro hodnocení dostupných možností. Následně empirická část práce analyzuje data a zkušenosti získané z testování prototypů, jejichž porovnání přináší cenné postřehy pro finální výběr implementace. Detailní návrh implementace je pak představen s důrazem na technické, ekonomické a provozní aspekty, které jsou nezbytné pro úspěšnou realizaci aplikace.

Další hodnotou práce je přínos v podobě návrhů na vylepšení jednotlivých řešení, které byly identifikovány během analytické fáze. Tento přístup nejen zvyšuje praktickou aplikovatelnost výsledků práce, ale také otevírá prostor pro další inovace. Ve finální fázi je práce završena vyhodnocením nasazení vybrané aplikace v produkčním prostředí softwaru určeného pro školství. Využití reálných provozních dat poskytuje autentický pohled na funkčnost a efektivitu implementace, a umožňuje vyvodit praktické doporučení pro nasazení podobných systémů v podobných podmínkách. Výsledky tak představují zdroj informací pro rozhodování o technologickém směřování v kontextu návrhu enterprise aplikací.

5.1 Seznam použitých zdrojů

Yasar, Kinza, Chai, Wesley a Bigelow, J. Stephen. 2023. TechTarget CloudComputing deifnition. *TechTarget.* [Online] TechTarger, 2023. https://www.techtarget.com/searchcloudcomputing/definition/cloud-computing.

Cloudflare. 2023. what is cloud. *Cloudflare.* [Online] Cloudflare, 2023. https://www.cloudflare.com/en-gb/learning/cloud/what-is-the-cloud/.

Oracle. 2024. What is SaaS. *Oracle.* [Online] Oracle, 2024. https://www.oracle.com/cz/applications/what-is-saas/.

Bezpalec, Pavel. 2015. *Nové trendy v elektronických komunikacích Cloud Computing.* Praha : České vysoké učení technické v Praze, 2015. 000-00-000-0000-0.

AWS. 2024. Amazon Web Services Cloud. *AWS.* [Online] Amazon, 2024. https://docs.aws.amazon.com/.

—. 2014. AWS Lambda – Run Code in the Cloud. *AWS.* [Online] AWS, 2014. https://aws.amazon.com/blogs/aws/run-code-cloud/.

Serverless. 2023. AWS Lambda, The Ultimate Guide. *Serverless.* [Online] Serverless, Inc., 2023. https://www.serverless.com/aws-lambda.

Docker. 2024. Docker overview. *Docker.* [Online] 2024. https://docs.docker.com/get-started/overview/.

Amazon Web Services. 2024. AWS EC2. *AWS.* [Online] Amazon Web Services, 2024. https://aws.amazon.com/ec2/.

Bechný, Daniel. 2021. *Implementace frameworku Allure do testů v Javě.* místo neznámé : Univerzita Hradec Králové, 2021. https://theses.cz/id/kivnkp/.

Oracle. 2024. Java Virtual Machine Specification. *Oracle docs.* [Online] Oracle, 2024. https://docs.oracle.com/en/java/javase/21/vm/java-virtual-machine-technology-overview.html#GUID-982B244A-9B01-479A-8651-CB6475019281.

Javatpoint. 2021. JVM (Java Virtual Machine) Architecture. *Javatpoint.* [Online] 2021. https://www.javatpoint.com/jvm-java-virtual-machine#jvminternalarch.

Dlabal, Matouš. 2022. *Vývoj mobilních aplikací pomocí technologie kotlin multiplatform.* místo neznámé : ČVUT, 2022. 108929.

Markovic, Ban. 2020. Process of compiling Android app with Java/Kotlin code. *Medium.* [Online] 2020. https://medium.com/@banmarkovic/process-of-compiling-android-app-with-java-kotlin-code-27edcfcce616.

Procházka, Šimon. 2019. *Vývoj webové aplikace* . místo neznámé : VÚT, 2019.

Js pro vývojáře. 2021. Vznik a historie JavaScriptu. *Javascript pro vývojáře.* [Online] 26. 6 2021. https://js.provyvojare.cz/zaklad/vznik-javacriptu.

NodeJS. 2024. Node.js. *NodeJS.* [Online] OpenJS Foundation, 2024. https://nodejs.org/.

npm, Inc. 2024. npmjs. *npmjs.* [Online] npm, Inc, 2024. https://www.npmjs.com/.

Spring. 2024. Introduction to the Spring IoC Container and Beans. *Docs Spring.* [Online] VMware, 2024. https://docs.spring.io/.

—. 2024. Spring IO. *Spring IO.* [Online] VMware, 2024. https://spring.io/.

Network, IT. 2024. Úvod do Spring Boot frameworku v Javě Zdroj: https://www.itnetwork.cz/java/spring-boot/zaklady/uvod-do-spring-boot-frameworku-pro-javu. *Itnetwork.* [Online] 2024. https://www.itnetwork.cz/java/spring-boot/zaklady/uvod-do-spring-boot-frameworku-pro-javu.

Fol, Pavel. 2024. What is Apache Tomcat. *JRebel.* [Online] Perforce, 2024. https://www.jrebel.com/blog/what-is-apache-tomcat.

Apache Maven. 2024. Maven what is it? *Maven.* [Online] 2024. https://maven.apache.org/.

Geeks for Geeks. 2023. Introduction to Gradle. *Geeks for Geeks.* [Online] Sanchhaya Education Private Limited, 2023. https://www.geeksforgeeks.org/introduction-to-gradle/.

Microsoft. 2024. Playwright docs. *Playwright docs.* [Online] 2024. https://playwright.dev/docs.

Schiemann, Dylan. 2020. Microsoft Announces Playwright Alternative to Puppeteer. *InfoQ.* [Online] 30. 1 2020. https://www.infoq.com/news/2020/01/playwright-browser-automation/.

AWS. 2024. AWS Pricing Calculator. *AWS Pricing Calculator.* [Online] Amazon Web Services, 2024. https://calculator.aws/#/.

Apache Commons. 2024. Apache Commons Pool. *Apache Commons.* [Online] The Apache Software Foundation, 2024. https://commons.apache.org/proper/commons-pool/.

Cloudflare. 2024. Mutual TLS. *Cloudflare.* [Online] Cloudflare, 2024. https://www.cloudflare.com/en-gb/learning/access-management/what-is-mutual-tls/.

6 Seznam obrázků, tabulek, grafů a zkratek

6.1 Seznam obrázků

6.2 Seznam tabulek

6.3 Seznam rovnic

Přílohy

Kód pro OP verzi: https://github.com/dennnyb/pdfGeneratorApp

Kód serverless verze: https://github.com/dennnyb/pdfGeneratorApp-onprem

Estimace nákladů AWS:


```
{
          "Name": "My Estimate",
          "Total Cost": {
                    "monthly": "2819.37",
                    "upfront": "0.18",
                    "12 months": "33832.62"
          },
          "Metadata": {
                    "Currency": "USD",
                    "Locale": "en_US",
                    "Created On": "22/03/2024",
                    "Legal Disclaimer": "AWS Pricing Calculator provides only an estimate of
your AWS fees and doesn't include any taxes that might apply. Your actual fees depend on a
variety of factors, including your actual usage of AWS services.",
                    "Share Url":
"https://calculator.aws/#/estimate?id=43f246810c82d993e606d319ff683ca6f8b26bd5"
          },
          "Groups": {
                    "Services": [
                              {
                                        "Service Name": "Amazon EC2 ",
                                        "Description": "Extreme",
                                        "Region": "US East (N. Virginia)",
                                        "Status": "",
                                        "Service Cost": {
                                                  "monthly": "1307.16",
                                                  "upfront": "0.00",
                                                  "12 months": "15685.87"
                                        },
                                        "Properties": {
                                                  "Tenancy": "Dedicated Instances",
                                                  "Operating system": "Linux",
                                                  "Workload": "Daily, (Workload days: Monday,
Tuesday, Wednesday, Thursday, Friday, Baseline: 1, Peak: 12, Duration of peak: 3 Hr 30 Min)",
                                                  "Advance EC2 instance": "r5a.4xlarge",
                                                  "Pricing strategy": "On-Demand",
                                                  "Enable monitoring": "disabled",
                                                  "DT Inbound: Not selected": "0 TB per month",
```

```
                                        "DT Outbound: Internet": "70 GB per month",
                                        "DT Intra-Region:": "0 TB per month"
                                }
                        },
                        {
                                "Service Name": "Amazon EC2 ",
                                "Description": "Middle",
                                "Region": "US East (N. Virginia)",
                                "Status": "",
                                "Service Cost": {
                                        "monthly": "792.72",
                                        "upfront": "0.00",
                                        "12 months": "9512.64"
                                },
                                "Properties": {
                                        "Tenancy": "Dedicated Instances",
                                        "Operating system": "Linux",
                                        "Workload": "Daily, (Workload days: Monday,
Tuesday, Wednesday, Thursday, Friday, Baseline: 1, Peak: 5, Duration of peak: 3 Hr 30 Min)",
                                        "Advance EC2 instance": "r5a.4xlarge",
                                        "Pricing strategy": "On-Demand",
                                        "Enable monitoring": "disabled",
                                        "DT Inbound: Not selected": "0 TB per month",
                                        "DT Outbound: Internet": "20 GB per month",
                                        "DT Intra-Region:": "0 TB per month"
                                }
                        },
                        {
                                "Service Name": "Amazon EC2 ",
                                "Description": "Low",
                                "Region": "US East (N. Virginia)",
                                "Status": "",
                                "Service Cost": {
                                        "monthly": "416.10",
                                        "upfront": "0.00",
                                        "12 months": "4993.20"
                                },
                                "Properties": {
                                        "Tenancy": "Shared Instances",
                                        "Operating system": "Linux",
                                        "Workload": "Consistent, Number of instances:
1",
                                        "Advance EC2 instance": "r5a.4xlarge",
                                        "Pricing strategy": "1yr  No Upfront",
                                        "Enable monitoring": "disabled",
                                        "DT Inbound: Not selected": "0 TB per month",
```

```
                                        "DT Outbound: Not selected": "0 TB per
month",
                                        "DT Intra-Region:": "0 TB per month"
                                }
                        },
                        {
                                "Service Name": "AWS Lambda",
                                "Description": "Low",
                                "Region": "US East (N. Virginia)",
                                "Status": "",
                                "Service Cost": {
                                        "monthly": "7.60",
                                        "upfront": "0.00",
                                        "12 months": "91.20"
                                },
                                "Properties": {
                                        "Architecture": "x86",
                                        "Invoke Mode": "Buffered",
                                        "Number of requests": "500 per day",
                                        "Amount of ephemeral storage allocated": "512
MB"
                                }
                        },
                        {
                                "Service Name": "AWS Lambda",
                                "Description": "Middle",
                                "Region": "US East (N. Virginia)",
                                "Status": "",
                                "Service Cost": {
                                        "monthly": "90.66",
                                        "upfront": "0.00",
                                        "12 months": "1087.92"
                                },
                                "Properties": {
                                        "Architecture": "x86",
                                        "Invoke Mode": "Buffered",
                                        "Number of requests": "3000 per day",
                                        "Concurrency": "50",
                                        "Amount of ephemeral storage allocated": "512
MB",
                                        "Time for which Provisioned Concurrency is
enabled": "30 hours",
                                        "Number of requests for Provisioned
Concurrency": "50 per month"
                                }
                        },
```

```
                        {
                                "Service Name": "AWS Lambda",
                                "Description": "Extreme",
                                "Region": "US East (N. Virginia)",
                                "Status": "",
                                "Service Cost": {
                                        "monthly": "197.17",
                                        "upfront": "0.00",
                                        "12 months": "2366.04"
                                },
                                "Properties": {
                                        "Architecture": "x86",
                                        "Invoke Mode": "Buffered",
                                        "Number of requests": "10000 per day",
                                        "Concurrency": "50",
                                        "Time for which Provisioned Concurrency is
enabled": "30 hours",
                                        "Number of requests for Provisioned
Concurrency": "100 per month",
                                        "Amount of ephemeral storage allocated": "512
MB"
                                }
                        },
                        {
                                "Service Name": "S3 Standard",
                                "Region": "US East (N. Virginia)",
                                "Status": "",
                                "Service Cost": {
                                        "monthly": "1.66",
                                        "upfront": "0.18",
                                        "12 months": "20.10"
                                },
                                "Properties": {
                                        "S3 Standard storage": "70 GB per month",
                                        "S3 Standard Average Object Size": "2 MB",
                                        "PUT, COPY, POST, LIST requests to S3
Standard": "10000",
                                        "GET, SELECT, and all other requests from S3
Standard": "10000"
                                }
                        },
                        {
                                "Service Name": "Data Transfer",
                                "Region": "US East (N. Virginia)",
                                "Status": "",
                                "Service Cost": {
```

```
                                        "monthly": "6.30",
                                        "upfront": "0.00",
                                        "12 months": "75.60"
                                },
                                "Properties": {
                                        "DT Inbound: Not selected": "0 TB per month",
                                        "DT Outbound: Internet": "70 GB per month"
                                }
                        }
                ]
        }
}
```