ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vítovec** Jméno: **Josef** Osobní číslo: **421119**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Studijní obor: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Identifikační systém pro Národní filmový archiv

Název diplomové práce anglicky:

Identification system for National film archive

Pokyny pro vypracování:

Cílem diplomové práce bude analýza, návrh, implementace a testování speciálního identifikačního systému pro potřeby Národního filmového archivu. Identifikační systém zajistí generování a správu jedinečných perzistentních (trvalých) identifikátorů různých typů (s možností definice jejich struktury) pro odlišné typy objektů v různých systémech. Pro každý typ identifikátoru bude systém udržovat sadu povinných metadat.
Identifikátory bude možné přidělovat automatizovaně (přes API) i manuálně (přes webové rozhraní). Webové rozhraní identifikačního systému bude sloužit také pro administraci systému. Přes něj také bude možné importovat stávající identifikátory jiných systémů a pokračovat v jejich generování automatizovaně. Systém bude dále realizovat funkci tzv. resolveru, kdy bude na základě identifikátoru schopný uživatele přesměrovat na aktuální URL identifikovaného objektu.

Seznam doporučené literatury:

[1] HILSE, Hans Werner a Jochen KOTHE Implementing persistent identifier: overview of concepts, guidelines and reccomendations

Jméno a pracoviště vedoucí(ho) diplomové práce:

BcA. Jonáš Svatoš, Vedoucí Digitální laboratoře, Národní filmový archiv

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

Datum zadání diplomové práce: **07.09.2020** Termín odevzdání diplomové práce: **05.01.2021**

Platnost zadání diplomové práce: **19.02.2022**

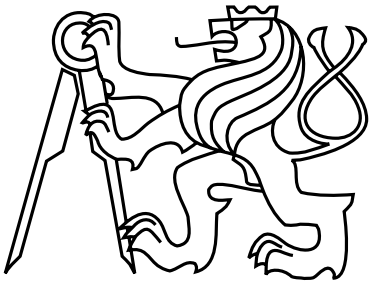
BcA. Jonáš Svatoš podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

.	
Datum převzetí zadání	Podpis studenta

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

Identifikační systém pro Národní filmový archiv

Bc. Josef Vítovec

Vedoucí práce: BcA. Jonáš Svatoš

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

5. ledna 2021

Poděkování

Zde bych rád poděkoval vedoucímu diplomové práce Bca. Jonáši Svatošovi za ochotu a rady během tvorby této práce. Dále potom PhDr. Ladislavu Cubrovi, Ph.D. za připomínky a pomoc při sběru požadavků. V neposlední řadě bych rád poděkoval všem členům projektu Videoarchiv, nez nichž by tato práce navznikla a samozřejmě také svým nejbližším včetně rodiny a přátel za trpělivost a podporu.

Prohlášení

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

V Praze dne 5. 1. 2020 .....

Abstract

This thesis focuses on the design and development of identification system that should be part of the National film archive technological infrastructure. It describes the whole process starting from analysis through design to implementation and testing. First, the thesis describes the area of persistent identification together with providing examples of existing identification schemas. Next it discusses the requirements and variety of technology with emphasis on sustainability and future development. Last but not least it covers the development and testing process.

Abstrakt

Diplomová práce se zaměřuje na návrh a vývoj identifikačního systému pro potřeby Národního filmového archivu. Identifikační systém zajišťuje generování a správu jednozančných perzistentních identifikátorů. Součástí systému je webové rozhraní s definovanými uživatelskými rolemi a k nim příslušnými operacemi. Práce nejprve představuje problematiku perzistentní digitální identifikace včetně již existujících identifikačních schémat. Následuje diskuze nad výběrem technologí a specifikace požadavků. Poslední fází je implementace a otestování systému.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Tato práce si klade za cíl vyvinout prototyp aplikace, která bude zajišťovat generování, uchovávání a správu jedinečných trvalých identifikátorů pro potřeby Národního filmového archivu. Identifikátory bude možné přidělovat z externích systémů automatizovaně přes API, příp. manuálně skrze webové rozhraní. Ke každému identifikátoru budou uchovávány přidružená metadata popisující identifikovaný objekt. Tyto metadata se budou lišit v závislosti na typu objektu – každý objekt je součástí tzv. archivního fondu, a zároveň podléhá určité úrovni archivního popisu.

Webové rozhraní pracuje s definovanými uživatelskými rolemi, na základě kterých bude možné identifikované objekty prohlížet, upravovat, popř. přímo vytvářet. Nedílnou součástí systému bude i tzv. resolver, což je služba, která na základě identifikátoru provede přesměrování na aktuální URL objektu.

Zpočátku se práce zabývá samotnou problematikou identifikace digitálních objektů včetně definice souvisejících pojmů, ze kterých se poté vychází při specifikaci požadavků. Následuje hrubý popis současného stavu v instituci a charakteristika již existujících schémat identifikace. Další kapitola se věnuje návrhu architektury a rozhodovacímu procesu při volbě použitých technologií. Nakonec dojde na popis samotné implementace a následnému testování.

1.1 Motivace

Internet dnes bezpochyby poskytuje nejjednodušší a nejdostupnější způsob sdílení digitálního obsahu. Často stačí pouze vložit URL adresu odkazu, který nám byl nasdílen, do libovolného prohlížeče, a v řádu několika málo sekund máme informace na obrazovce. Nejednou se ale každému z nás určitě stalo, že namísto kýženého obsahu se zobrazila pouze chybová hláška *Not Found*. V případě, že se jedná např. o URL obrázku, který jsme nalezli přes Google, není to takový problém – obrázek můžeme nejspíše snadno vyhledat znovu. Pokud je však na zdrojovém URL umístěn článek, který jsme citovali ve své závěrečné práci, a po dvou měsících již článek není dostupný, nejenže to vrhá na celou práci špatné světlo, zároveň se tím komplikuje celý proces sdílení odborných i neodborných informací. V kontextu paměťových institucí[1], jejichž posláním je mimo

[1] Paměťová instituce označuje knihovny, archivy, muzea, výzkumné ústavy, univerzity, jejichž cílem je ochrana a zpřístupňování dokumentů kulturního dědictví [2].

jiné ochrana a zpřístupňovaní kulturního dědictví, to potom znamená, že se tyto cíle nedaří dostatečně naplňovat.

1.2 Archivnictví

Jelikož se tato práce dotýká problematiky archivnictví, a samotný navrhovaný systém je pro archiv přímo určen, je nutné tento vědní obor alespoň zběžně představit a zadefinovat pojmy s ním související. S těmito pojmy se poté v diplomové práci dále pracuje, seznámení se s nimi je tedy nezbytným předpokladem k porozumění zbytku textu.

Archivnictví je obor lidské činnosti zaměřený na péči o archiválie jako součástí národního kulturního dědictví a plnící funkci správní, informační, vědecké a kulturní.

Podoba archivnictví i pojmy s ním související jsou v České republice dány zákonem o archivnictví a spisové službě.

1.2.1 Definice pojmů

Archiválie

Archiválií je takový dokument, který byl vzhledem k době vzniku, obsahu, původu, vnějším znakům a trvalé hodnotě dané politickým, hospodářským, právním, historickým, kulturním, vědeckým nebo informačním významem vybrán ve veřejném zájmu k trvalému uchování a byl vzat do evidence archiválií.

Archivní fond

Archivní fond nebo archivní sbírka (souhrnně archivní soubor) je základním prvkem v archivnictví, je definován legislativně a vymezuje se i fyzicky, případně jasnou evidenční vazbou u odděleně uložených, zejména elektronických archiválií. Každá archiválie nebo jejich skupina musí být součástí jednoho archivního fondu nebo jedné archivní sbírky, v jejímž rámci se také zpracovává a zpřístupňuje. [3]

Archivní popis

Archivní popis je výsledkem zpracování veškerých dostupných informací o archivním souboru do podoby informačního systému, který umožňuje zpřístupnění archiválií v reálném nebo virtuálním (digitálním) prostředí. Jedná se o informační celek, který se v průběhu jednotlivých archivních činností jako jsou evidence, zpracování, inventarizace a katalogizace archiválií neustále doplňuje, opravuje a prohlubuje v závislosti na tom, jak jsou zpřesňovány informace o příslušných archiváliích. [3]

Digitální knihovna

Pro potřeby této práce postačí, pokud si digitální knihovnu zadefinujeme jako systém sloužící primárně pro zpřístupnění archiválií popř. jejich derivátů.

1.2.2 Národní filmový archiv

Národní filmový archiv je specializovaný archiv zřízený jako příspěvková organizace Ministerstva kultury České republiky, a patří k deseti nejstarším a největším filmovým archivům na světě. Byl založen roku 1943 a v České republice se řadí k nejvýznamnějším paměťovým institucím; vedle plnění archivní role se podílí na zpřístupňování audiovizuálního kulturního dědictví veřejnosti, zabývá se vědeckou a publikační činností, prezentací filmového dědictví a rozvojem současné české kinematografie a pohyblivého obrazu.

V kontextu této práce se bude pro Národní filmový archiv využívat zkratka NFA.

Projekt Videoarchiv

Projekt Videoarchiv, formálně *Audiovizuální dílo mimo kontext kinematografie: dokumentace, archivace a zpřístupnění*, je výzkumný projekt financovaný z programu NAKI II Ministerstva kultury ČR[2]. Cílem projektu, probíhajícího pod vedením Národního filmového archivu, je vytvořit dlouhodobě udržitelnou strategii pro prezervaci a zpřístupnění audiovizuálních děl mimo kontext kinematografie. Pod pojmem audiovizuální dílo mimo kontext kinematografie si můžeme představit dílo z oblasti výtvarného umění s dominantními prvky pohyblivého obrazu, které svými charakteristikami nezapadá do představy o klasickém filmu. Mezi takováto díla můžeme řadit např. videoart, počítačové umění, intermediální a interdisciplinární díla atd.

V kontextu této práce projekt zmiňuji, jelikož byl jedním z hlavních impulsů pro vznik identifikačního systému. Jak je již zmíněno výše, díla, jimiž se Projekt Videoarchiv zaobírá, zasahují mimo kontext kinematografie. A jelikož NFA primárně takovým dílům pozornost nevěnuje, neexistuje v instituci potřebná technická infrastruktura pro jejich dlouhodobou identifikaci.

[2]Program na podporu aplikovaného výzkumu a vývoje národní a kulturní identity.

Kapitola 2

Analýza

Tato kapitola nejdříve vysvětluje pojem perzistentní identifikace a definuje obecné požadavky na systém spravující perzistentní identifikátory. Následuje sekce popisující současnou situaci v NFA z hlediska perzistentní identifikace.

2.1 Perzistentní identifikace

S pojmem perzistentní identifikace se lze nejpravděpodobněji setkat u identifikace digitálních informačních zdrojů v prostředí internetu. Některé z požadavků uvedených v následující sekci 2.1.1 je určitě možné aplikovat i na identifikátory fyzických objektů, nicméně v kontextu této práce se zaměřím pouze na problematiku digitální identifikace.

Běžné neperzistentní identifikátory často plní svůj účel po velmi omezenou dobu, ať už z důvodu změny samotného informačních zdroje nebo URL, na které je zdroj dostupný. Cílem perzistentní identifikace je tyto neduhy eliminovat a poskytnout způsob, jak zpřístupnit informační zdroje v dlouhodobém horizontu.

Na úvod je důležité říci, že samotný identifikátor, ať už má jakoukoliv syntax, ani systém spravující takové identifikátory, implicitně perzistencí nedisponuje. Perzistence lze dosáhnout až s trvalou koordinací a správou včetně nastavení příslušných procesů na straně instituce disponující takovými identifikátory [4, s. 41].

2.1.1 Požadavky na perzistentní identifikátory

Syntax

Syntax identifikátoru je dána použitým identifikačním schématem popř. systémem, který identifikátory přiděluje. Obecně však platí, že do struktury identifikátoru by se neměly promítnout žádné údaje o identifikovaném informačním zdroji jako jsou např. metadata. Rovněž je doporučováno se vyhnout jakýmkoliv slovům nesoucím význam. V průběhu času se může význam takového slova proměnit a způsobit tak nejasnosti při práci s identifikátorem, přičemž identifikátor jako takový by nikdy neměl být předmětem jakýchkoliv změn [4, s. 43]. Syntax identifikátoru by tedy měla být sémanticky neutrální, a pokud možno by měla obsahovat pouze alfanumerické znaky doplněná o omezený počet oddělovačů jako je např. pomlčka či dvojtečka.

Jednoznačnost

Jednoznačná identifikace je základním předpokladem pro identifikátory obecně. Pokud by identifikátor k informačnímu zdroji Ż nebyl, a identifikoval by např. dva různé zdroje, nebylo by možné dosáhnout požadavku přesměrování zmíněného níže v této sekci.

Perzistence

Perzistence je vlastnost zajišťující jednoznačnou identifikaci v dlouhodobém horizontu. V důsledku to znamená, že identifikátor, pokud je již jednou přidělen, nemůže být nikdy přidělen znovu, a to ani v případě, že původní zdroj zaniknul. Perzistenci lze zajistit na úrovni identifikačního systému vhodným nastavením pravidel při přidělování identifikátoru.

Směrování

Poslední, a z pohledu uživatele nejužitečnější vlastnost, je možnost směrování na aktuální URL informačního zdroje. To probíhá na základě identifikátoru, které je vložen včetně odpovídající domény a cesty k resolveru přímo do prohlížeče. Následuje standardní HTTP přesměrování do některé z digitálních knihoven, kde je daný zdroj zpřístupněn. Směrování musí být technicky vyřešeno tak, že i v případě, že došlo ke změně URL, uživatel bude přesměrován na správnou adresu.

2.1.2 Komponenty identifikačního systému

Technické požadavky na systémy perzistentní identifikace nepodléhají žádným standardům. Na základě požadavků na samotné identifikátory a jejich funkcionalitu je však možné technickou infrastrukturu rozdělit na následující čtyři komponenty [5, s. 198].

Generátor

Generátor je nástroj zajišťující tvorbu a přidělování identifikátorů. Generování by mělo probíhat v souladu se syntaxí danou identifikačním systémem a zároveň by mělo zajišťovat jednoznačnost a perzistenci generovaných identifikátorů.

Registr

Registrem se rozumí databáze obsahující identifikátory a k nim příslušné informace týkající se samotných informačních zdrojů jako jsou např. metadata. Registr si zároveň udržuje i adresy, na kterých jsou identifikované zdroje dostupné a na základě kterých probíhá směrování.

Resolver

Resolver neboli přesměrovávací služba zajišťuje směrování na aktuální URL informačních zdrojů na základě jejich identifikátoru. Aby bylo možné směrování realizovat je nutné, aby byl resolver propojen s registrem.

Evidenční služba

Evidenční služba má za úkol aktualizovat URL identifikovaných informačních zdrojů. Tato aktualizace může na základě sklízení, tzv. harvestingu, metadat z příslušných digitálních knihoven.

2.2 Současná situace v NFA

V současnosti v Národním filmovém archivu neexistuje žádná forma identifikace digitálních informačních zdrojů. Pokusím se tedy alespoň stručně nastínit situaci ohledně analogových identifikátorů. Centrálním systémem pro správu audiovizuálních materiálu je informační a katalogizační systém s názvem AIS. Jedná se o systém napsaný v jazyce COBOL, který nepracuje s relační databází v pravém slova smyslu. Relace mezi jednotlivými entitami jsou nahrazeny tzv. vazebními rejstříky. V systému AIS jsou sledovány tyto základní okruhy údajů:

- filmy – filmografické údaje o filmových titulech
- filmové materiály – technické údaje o filmových materiálech (negativ, duplikační pozitiv, kopie), které existují k filmovým titulům
- fotografie – technické údaje o fotografiích, které existují k filmovým titulům
- plakáty – technické údaje o plakátech, které existují k filmovým titulům
- osobnosti – jmenný seznam osobností

Co se týče identifikace – filmy, filmové materiály, fotografie a plakáty mají u každého záznamu identifikátor. Záznamy osobností jsou uloženy pouze v jednoduchém číselníku, tudíž žádným identifikátorem nedisponují. Ani u jedné ze zmíněných entit disponující vlastním identifikátorem však nemůžeme mluvit o perzistentním identifikátoru.

Film je identifikován řetězcem numerických znaků o délce sedm. Nejedná se však o náhodně generovaná číslice, ani o systém postupné inkrementace na základě přibývajících filmů. První tři znaky z řetězce totiž zároveň určují kategorii filmu. Každá kategorie potom spadá do některého z předem definovaných rozsahů. Pro představu uvádím několik příkladů:

- **001**0001 až **346**9999 – hraný film
- **360**0001 až **360**9999 – amatérský hraný film
- **006**0001 až **009**9999 – český a slovenský dokumentární film
- **350**0001 až **350**9999 – zahraniční dokumentární film
- **362**0001 až **362**9999 – amatérský dokumentární film
- **400**0001 až **400**9999 – animovaný film
- **364**0001 až **364**9999 – amatérský animovaný film

Důvodem pro rozhodnutí promítat filmografické údaje přímo do struktury identifikátoru byla nejspíše snaha umožnit rychlou orientaci uživatelů i při práci se samotnými identifikátory. Z uživatelského pohledu tento přístup určitě může mít své výhody, nicméně v dlouhodobém horizontu je ze své podstaty neudržitelný, a navíc nepřináší vlastnosti pro jednoznačnou a perzistentní identifikaci. V důsledku potom dochází i ke ztrátě na první pohled viditelné uživatelské přívětivosti. Jako problémové vlastnosti lze považovat následující:

1. Kategorie filmu nejsou navzájem disjunktními množinami – existují filmy, které lze zařadit do více než jedné kategorie. U takových filmů tedy není potom jisté, do jaké skupiny identifikátorů by měly patřit.
2. Omezený rozsah identifikátorů. Kategorie jako animovaný film příp. amatérský hraný film umožňují dle současných pravidel přidělení maximálního počtu 9999 identifikátorů. V případě překročení tohoto limitu by nejspíše bylo nutné zavést nový rozsah, příp. rozšířit ten stávající.
3. Před samotným přidělením identifikátoru je nutné vědět, do jaké kategorie film náleží. Až poté je možné na základě této kategorie identifikátor přidělit.
4. Vazba mezi strukturou identifikátoru a kategorií filmu. V případě, že dojde k chybnému identifikování kategorie filmu, na základě které se poté přidělí chybný identifikátor, bude nutné pro tento film identifikátor přidělit znovu. S přibývajícím časem navíc dopad tohoto chybného přidělení narůstá.

Identifikátor k fotografiím a plakátům vychází z identifikátoru příslušného filmu, resp. identifikátor filmu je přímo jejich součástí. Např. film s identifikátorem 0010971 obsahuje fotografie s identifikátory 0010971a01 až 0010971c53 a jeden plakát s identifikátorem 0010971a01. Oproti identifikátoru k filmu tedy navíc obsahují proměnlivou část, která je tvořena písmenem a číslem v rozsahu 0 až 99. Výhodou takového přístupu je z pohledu uživatele snadná identifikace filmu na základě fotografie či plakátu. Samotné identifikátory jsou však pro fotografie a plakáty totožné, což znamená, že ani v kontextu Národního filmového archivu se nelze spolehnout na jejich jednoznačnost.

2.3 Existující způsoby identifikace

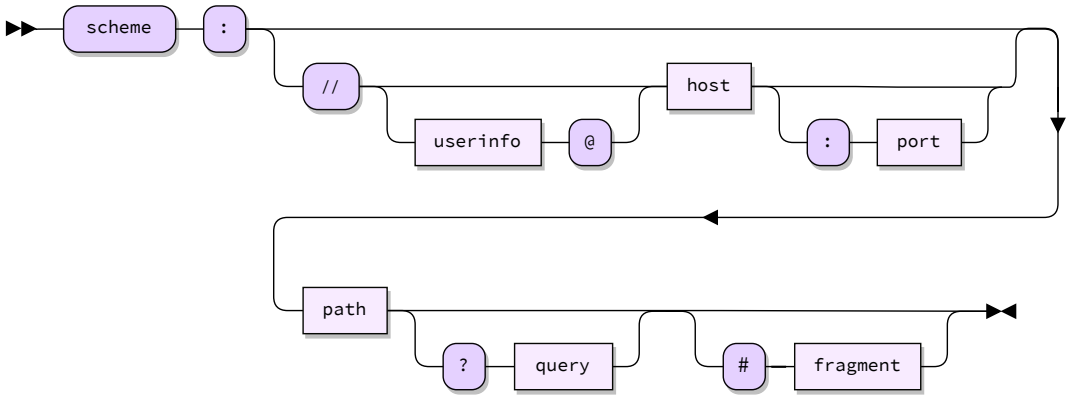
2.3.1 URI

Uniform Resource Identifier, zkráceně URI, v překladu potom „jednotný identifikátor zdroje“, je textový řetězec sloužící k identifikaci informačního zdroje. Tento zdroj může být jak abstraktní, tak fyzický [6]. Formát URI je vzhledem k šířce použití spíše volný. Příkladem URI mohou být např.:

- https://www.doi.org/factsheets/DOIHandle.html
- ftp://ftp.is.co.za/rfc/rfc1808.txt
- mailto:josef.vitovec@fel.cvut.cz

- tel:+1-212-555-1234
- urn:nbn:de:gbv:089-3321552
- info:doi/10.1000.10/123456

Hlavní složkou URI je tzv. schéma (https, ftp, mailto atd.), které blíže určuje význam a formátování zbytku identifikátoru. Samotné schéma může v některých případech být přímo i protokolem, jako je to např. v případě http, telnet, ftp. Obecná syntaxe URI je ilustrována na obrázku 2.1.



Obrázek 2.1: Diagram znázorňující syntax URI

URI může popisovat informační zdroj pouze z hlediska identity, pouze z hlediska umístění, nebo příp. může popisovat obojí současně. S URI bezprostředně souvisí URL a URN, jež jsou speciálními typy URI. Pro lepší ilustraci vztahů viz obrázek 2.2 Hranice mezi těmito pojmy jsou zdrojem častých diskuzí, a to nejspíše i kvůli nejasnostem ve specifikaci RFC[1] 3986 [7].

URL

Uniform Resource Locator, zkráceně URL a v překladu potom „jednotná adresa zdroje“, je textový řetězec sloužící k identifikaci adresy, na které je daný informační zdroj zpřístupněn. Každá URL je zároveň i URI, zpětně to však vždy platit nemusí. URL je využíváno nejčastěji schématem HTTP (využívající stejnojmenný protokol) pro přístup k webovým stránkám. V prohlížeči potom není nutné explicitně uvádět schéma, jelikož s HTTP prohlížeč pracuje automaticky. To samé platí pro port, který pokud není specifikován, je vždy 80. Příkladem URL může být např.:

https://example.com/path/to/resource/object.html

URL ze své podstaty závisí na doménovém jménu příp. přímo na IP adrese. U URL výše se konkrétně jedná o doménu *example.com*. Následuje cesta k informačnímu zdroji,

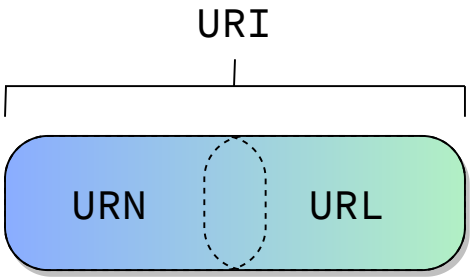
[1] Request for Comments – označení pro dokumenty, jejichž cílem je specifikace různých aspektů internetové komunikace jako jsou např. síťové protokoly, procedury, programy atd.

jejíž tvar často však není plně v moci organizace zpřístupňujícího daný zdroj. A to především z důvodu použití nejrůznějších technologií a frameworků, které mohou cestu předefinovat po svém. Takové cesty je sice možné přepsat s použitím tzv. *URL rewrite* mechanismu, nicméně pokud dojde k migraci z jednoho systému na druhý a staré URL adresy budou namapovány na nové, bez toho aniž bychom přímo vzájemně porovnali zdroje na staré a nové URL, nelze s jistotou říct, že staré URL identifikují stejné zdroje jako nové URL.

Z hlediska neperzistentní a nejednoznačné identifikace může v kontextu problémů zmíněných výše dojít k těmto situacím:

1. informační zdroj není dostupný
2. informační zdroj byl přesunut a je dostupný na jiné doméně
3. informační zdroj byl přesunut a je dostupný na stejné doméně avšak v rámci jiné cesty

Ačkoliv je URL z uživatelského hlediska zřejmě nejpohodlnější variantou identifikace, z těchto důvodů ho nelze považovat za stabilní způsob identifikace. Jak doménové jméno, tak samotná cesta a příp. i další části URL (query, fragment) podléhají neustálým změnám, které znemožňují spolehlivé použití v jiném než krátkodobém horizontu. Těmto změnám navíc z pohledu provozovatele nelze předejít ani vhodným nastavením interních procesů, jak se může na první pohled zdát.



Obrázek 2.2: Diagram ilustrující vztahy mezi URI, URL a URN

URN

Unified Resource Name, zkráceně URN a v překladu potom „jednotný identifikátor jména“, je trvalým identifikátorem, který je narozdíl od URL nezávislý na lokaci informačního zdroje. URN podléhá specifikaci RFC 1737 v rámci níž byly definovány funkční požadavky na tento identifikátor. Pro ilustraci uvádím některé z nich [8]:

- Globální rozsah: URN je identifikátor s globálním rozsahem, který v sobě nezahrnuje informaci o umístění zdroje. Všude na světě má stejný význam.
- Globální jedinečnost: Stejné URN nemůže být přiřazeno dvěma různým zdrojům.

- Trvalost: URN je trvalé bez ohledu na existenci samotného zdroje příp. vydávající autority, která URN přiřadila.
- Škálovatelnost: URN může být přiřazeno libovolnému zdroji.
- Nezávislost: Přidělování URN probíhá zcela za podmínek, které si určí vydávající autorita.
- Směrování: Pro URN, ke kterým existují odpovídající URL, musí existovat mechanismus, který zajistí překlad URN na URL.

Syntax URN vychází ze syntaxe URI a je definována specifikaci RFC 2141 takto:

<URN> ::= „urn:“ <NID> „:“ <NSS>

Schéma *„urn:“* je povinnou částí identifikátoru. Zkratka *NID* označuje globálně definovaný identifikátor jmenného prostoru mezi které řadíme např. ISBN (International Standard Book Number), NBN (National Bibliography Number), UUID (Universally Unique Identifier) aj. Zkratka *NSS* potom označuje specifický řetězec jmenného prostoru [9].

Na první pohled by se mohlo zdát, že URL a URN lze odlišit už na základě schématu. Nicméně to, jestli identifikátor identifikuje samotný objekt nebo pouze jeho lokaci často záleží spíše na autoritě, která daný identifikátor přiděluje, než na použitém schématu. Do budoucna se tedy spíše doporučuje použití obecnějšího označení URI namísto restriktivních URL a URN [6].

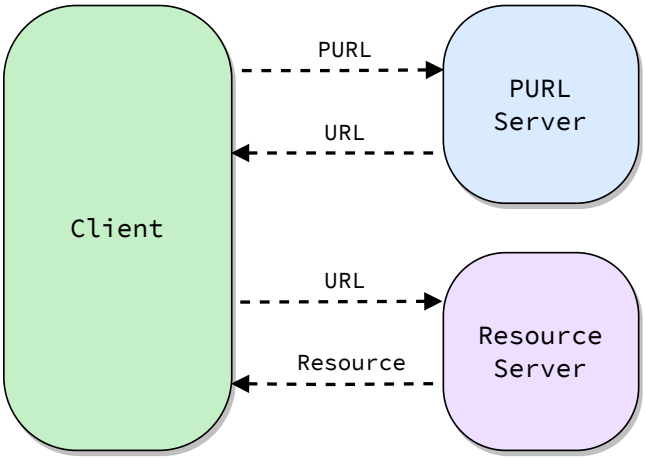
2.3.2 PURL

Persistent Uniform Resource Locator, zkráceně PURL a v překladu potom „trvalá jednotná adresa zdroje“ je typem trvalého identifikátoru. Narozdíl od URL však nesměřuje přímo na adresu identifikovaného informačního zdroje, ale nejdříve na směrovací službu, která si udržuje seznam odpovídajících URL k jednotlivým zdrojům. Na základě toho je vrácena aktuální URL, viz obrázek 2.3. Z pohledu uživatele dojde po zadání PURL do prohlížeče ke standardnímu směrování prostřednictvím HTTP.

2.3.3 Handle

Systém Handle je registr ve správě CNRI[2] umožňující přidělování trvalých identifikátorů, tzv. handlů, pro digitální informační zdroje. V kontextu programování pojmem handle rozumíme objekt bez vnitřní struktury, který slouží pouze jako reference na cílový objekt se složitější strukturou spravovaný jiným systémem. Příkladem může být práce se soubory, jež probíhá na úrovni operačního systému. Programu nepracuje přímo se soubory jako takovými, ale pouze s *file descriptory*, na jejichž základě potom probíhají všechny další operace s tímto souborem. Na stejném principu je postavený i Systém Handle – samotný identifikátor v sobě neobsahuje žádné informace o identifikovaném objektu, je provázaný pouze s jeho metadaty. Nezbytnou součástí systému je i resolver zajišťující směrování těchto identifikátorů.

[2]Corporation for National Research Initiatives



Obrázek 2.3: Diagram znázorňující princip PURL [1]

Handle je definován dle následující syntaxe:

<handle> ::= <prefix> „/“ <handle local name>

Prefix, tzv. *naming authority*, slouží k definici instituce přidělující identifikátor a je tvořen desetinnou číselnou notací. Znak tečky „.“ potom vyjadřuje hierarchii jmen v dané instituci. Zbylá část identifikátoru za lomítkem, tzv. *handle local name*, označována také jako *suffix*, identifikuje jméno konkrétního digitálního objektu.

Nutnou říci, že celý Systém Handle je vcelku komplexní a zahrnuje různé části včetně popisu autentizace nebo třeba definice protokolů týkajících se směrování na přidělující instituce. Samotné směrování identifikátorů je nezávislé na DNS a začíná u tzv. *Global Handle Registry*, což je služba spravovaná přímo CNRI. Tato služba obsahuje záznamy s příslušnými prefixy, na základě kterých se potom přistupuje do lokálních služeb (serverů), jež obsahují už konkrétní *handle local names* k jednotlivým prefixům. Výhodou této distribuované architektury je potom horizontální škálovatelnost, kdy s přibývajícími identifikátory přibývají i lokální služby zajišťující jejich dostupnost [10].

Handle lze zapsat přímo dle jeho syntaxe příp. je možné ho zapsat jako URI s použitím jmenného prostoru uvnitř schématu *info*. Handle 12.3456/78 je tedy možné zapsat jako info:hdl/12.3456/78. Další možností je zápis jako URL s využitím generického HTTP proxy serveru – https://hdl.handle.net/12.3456/78, kdy dojde přímo k přesměrování na aktuální umístění zdroje.

Systém Handle je v současnosti se zhruba 200 miliony směrovacích požadavků za měsíc hojně využíván, a to nejen v prostředí knihoven a univerzit. Pro zprovoznění identifikace k vlastním informačním zdrojům CRNI poskytuje software zdarma. Poplatek za registrační proces činí 50 USD, a za roční provoz služby se potom platí také 50 USD.

2.3.4 DOI

Digital Object Identifier, zkráceně DOI, je trvalým identifikátorem digitálního objektu založeném na Systému Handle. DOI se jako mezinárodní identifikační systém využívá

zejména ve sféře komerčních nakladatelů, a jako takový neukládá žádné požadavky na identifikovaný objekt. DOI využívá směrovací infrastrukturu Systému Handle, tudíž jakákoliv služba handle dokáže provést přesměrování na aktuální adresu informačního zdroje. Každý DOI je zároveň i handle. DOI však nabízí standardizaci a funguje spíše jako *framework* pro trvalou identifikaci zajišťující jak přesnou definici metadat, tak samotné přidělení DOI registrační agenturou a jeho následnou správu. Vzhledem k těmto službám je každé přidělení DOI zpoplatněno, směrování pro koncové uživatele je potom zdarma.

2.4 Specifikace požadavků

Součástí této sekce je popis funkčních a nefunkčních požadavků na systém. Sběr požadavků probíhal na základě konzultací se zaměstnanci NFA.

2.4.1 Funkční požadavky

F.1 Přihlášení

Pro přístup do aplikace je nutné přihlášení. Možnost přihlášení by měli mít pouze zaměstnanci NFA disponující emailovou adresou na doméně nfa.cz. S možností registrace pro ostatní uživatele mimo NFA se zatím nepočítá.

F.2 Uživatelské role

Aplikace by měla pracovat se třemi typy uživatelských rolí. Na základě těchto rolí je uživateli dán rozsah činností, které může vykonávat. Každý uživatel má přiřazenou právě jednu roli. Součástí aplikace by měly být následující role a k nim příslušné činnosti:

Role: **BASIC**

- přihlášení
- zobrazování detailů objektů identifikace
- zobrazování detailů fondů

Role: **CURATOR**

- činnosti role BASIC
- vytváření a editace objektů identifikace náležících fondu, ke kterému má daný uživatel přístup

Role: **ADMIN**

- činnosti role BASIC
- vytváření, editace a mazání fondů
- vytváření, editace a mazání úrovní popisů u jednotlivým fondů

- vytváření, editace a mazání digitálních knihoven a k nim příslušných instancí
- vytváření a úprava všech objektů identifikace
- deaktivace objektů identifikace
- import existujících identifikátorů s metadaty
- přiřazování uživatelských rolí uživatelům
- nastavování uživatelských přístupů k jednotlivým fondům

F.3 Generování jedinečných perzistentních identifikátorů

Základní funkcionalitou aplikace je možnost generovat identifikátory, a to jak přímo z aplikace, tak přes API. Aby bylo možné identifikátor vygenerovat je nutné splnit následující podmínky:

1. Musí existovat fond, pod který budou nově generované objekty identifikace spadat. Tento fond musí mít definovaný způsob generování – inkrementálně, příp. náhodně, požadovanou syntax identifikátoru a příznak určující, zda objekty identifikace podléhají hierarchii určené úrovní popisu a obsahují tudíž i odkaz na předka. Dále musí k danému fondu existovat alespoň jedna úroveň popisu.
2. Součástí požadavku musí být název úrovně popisu a dále metadata validní dle metadatového schématu daného právě požadovanou úrovní popisu.
3. Metadata a úroveň popisu musí být v rámci celého systému unikátní, jinak k vygenerování identifikátoru nedojde.
4. V případě, že bude u fondu povolené generování přes API, tak musí v systému existovat digitální knihovna a k ní příslušná instance s definovanými parametry. Rovněž je nutné pro digitální knihovnu vygenerovat API klíč, kterým se digitální knihovna při požadavcích bude autentizovat.

F.4 Přesměrování na aktuální URL záznamu

Klíčovou funkcionalitou systému je také směrování. Uvažujme, že aplikace bude běžet na doméně `id.nfa.cz`, a v systému zároveň existuje objekt s identifikátorem `nfa-va-a7e2b04fc9` a s definovanou URL, na které se daný objekt nachází. Přesměrování by mělo fungovat tak, že po zadání `id.nfa.cz/resolver/nfa-va-a7e2b0564fc9` by měl být uživatel přesměrován na aktuální URL objektu. V případě, že je k objektu přiřazeno více URL, přesměrování proběhne na tu adresu, která je označena jako výchozí.

F.5 Import existujících identifikátorů s metadaty

Uživatel s rolí *ADMIN* by měl mít, mimo samotného generování identifikátorů, možnost hromadně naimportovat již existující objekty identifikace z externích systémů. Tyto objekty by již disponovaly nějakým druhem identifikátoru a po jejich importu by s každým dalším vytvořeným objektem došlo, v případě, že by se jednalo o číselné

identifikátory založené na principu postupné inkrementace, k pokračování generování této řady. Pokud by se nejednalo o číselné po sobě jdoucí identifikátory, generování dalších identifikátorů by mělo standardně probíhat jako generování náhodných řetězců. Import by měl probíhat na základě nahrání souboru CSV, který bude obsahovat rovněž příslušná metadata. Před samotným importem bude nutné z pohledu uživatele s rolí *ADMIN* vytvořit fond, a k němu nastavit formát identifikátorů a specifikovat metadata. Po importu by mělo být potom možné pokračovat ve vytváření dalších objektů identifikace.

2.4.2 Nefunkční požadavky

N.2 Zabezpečení

Systém by měl být zabezpečený vůči standardním útokům jako jsou např. CSRF, XSS nebo SQL Injection.

Přístup do systému bude umožněn pouze zaměstnancům NFA s emailovou adresou na doméně nfa.cz. Každý uživatel bude moci vykonávat pouze takové operace, které mu umožňuje přidělená role.

N.3 Konzistence dat

Systém zajistí konzistenci dat i při souběžném zápisu.

Kapitola 3

Návrh

3.1 Architektura

Prvním krokem v návrhu je volba typu architektury. Architekturu lze definovat jako organizaci systému, či způsob definování jednotlivých komponent a jejich propojení. Jiná definice pojmem architektura rozumí množinu návrhových rozhodnutí, které musí být absolvovány v počáteční fázi projektu [11]. Architektur existuje celá řada, a ne všechny v kontextu naší aplikace dávají smysl. Pro webové aplikace se dnes nejčastěji používají dvě architektury – *layered architecture* a *microservice architecture*. Každá architektura má své specifické případy užití, které je potřeba vyhodnotit s ohledem na požadavky, dostupné zdroje a příp. budoucí rozvoj. V případě správně zvolené architektury lze předpokládat rychlejší vývoj nových funkcionalit.

3.1.1 Layered Architecture

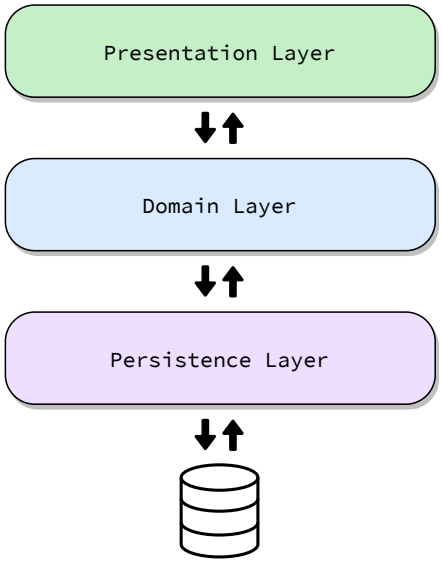
Pravděpodobně nejčastějším způsobem organizace aplikace a oddělení jednotlivých součástí systému je vícevrstvá architektura – *layered architecture*. Taková aplikace je rozdělená do několika horizontálních vrstev, z nichž každá plní svojí specifickou úlohu. Vzájemná komunikace potom vždy probíhá pouze mezi sousedními vrstvami. Tyto vrstvy jsou tedy navzájem logicky oddělené, a pokud jsou navíc oddělené i fyzicky, tj. jednotlivé vrstvy běží na samostatných, často virtuálních, serverech, mluvíme o tzv. *tiered architecture*. Hranice mezi těmito pojmy však není striktně definovaná, a často je možné narazit na obě varianty i v kontextu aplikací běžících na jednom stroji. Počet vrstev není touto architekturou pevně stanovený, nejčastěji se však v kontextu webových aplikací můžeme setkat se třemi vrstvami.

Vrstva s níž uživatel interaguje nejdříve se nazývá prezentační – *presentation layer*. Prezentační vrstva je zodpovědná za kontrolu HTTP requestů a vykreslování komponent uživatelského rozhraní pomocí HTML příp. JavaScriptu. O úroveň níže se nachází doménová vrstva – *domain layer*, jež obsahuje doménovou, často také označovanou jako business, logiku jako jsou např. různé výpočty, validace atd. Nejníže se nachází datová vrstva – *persistence layer*, zajišťující veškerou komunikaci s databází, a spravující data z ní získaná. Jednotlivé vrstvy a interakce mezi nimi jsou vyobrazené na obrázku 3.1.

Pokud tedy dojde ze strany uživatele k interakci s aplikací, pro ilustraci např. zobrazení výpůjček v knihovním systému, dojde k postupnému „probublání“ požadavku

od prezentační vrstvy až po vrstvu datovou zajišťující získání výpůjček z databáze. Tyto data se potom obdobným způsobem dostanou zpět až do nejvyšší vrstvy, kde se zpět prezentují uživateli.

Výhodou takového architektury je potom určitá nezávislost jednotlivých vrstev. Každou vrstvu je možné implementovat separátně, v případě, že dojde k zachování způsobu komunikace se sousedními vrstvami. Z hlediska testování je také výhodnější, pokud je striktně oddělená prezentační a doménová vrstva. Samotnou prezentační vrstvu není většinou úplně jednoduché otestovat, s čímž pomáhá právě přesun logiky do snadněji otestovatelné doménové vrstvy.



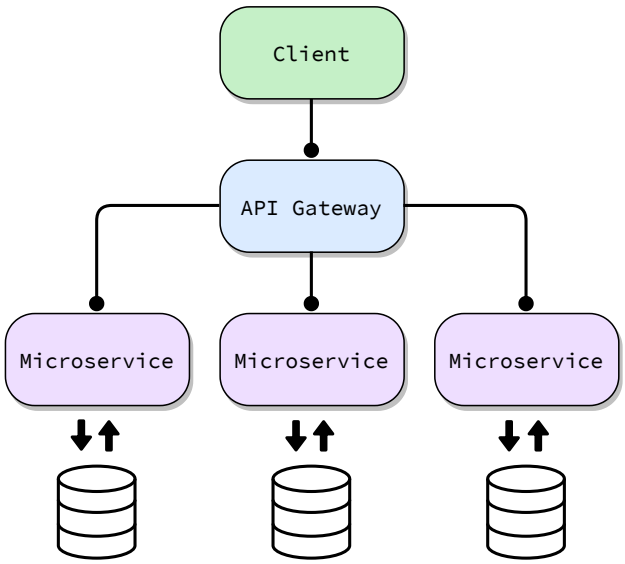
Obrázek 3.1: Diagram znázorňující vícevrstvou architekturu

3.1.2 Microservice Architecture

V poslední době se velkému zájmu těší architektura založená na mikroslužbách – *microservice architecture*. Základním principem této architektury je dekonstrukce monolitického systému na menší a nezávislé moduly, tzv. mikroslužby. Mikroslužby můžeme definovat jako součásti systému zodpovědné za specifickou oblast, což v kontextu již zmíněného knihovního systému může být např. správa uživatelů, správa výpůjček, správa knih atd. Jednotlivé mikroslužby mají definované rozhraní, nejčastěji postavené nad HTTP, prostřednictvím kterého mohou navzájem komunikovat. Mimo toto rozhraní jsou jednotlivé mikroslužby izolované a na sobě nezávislé, včetně databáze, kterou by v ideálním případě měla každá mikroslužba mít vlastní. Uživatel potom v rámci systému využívající mikroslužby přistupuje pouze na tzv. bránu API rozhraní – *API Gateway*, která zpřístupňuje jednotlivé mikroslužby v rámci jednoho přístupového bodu, viz obrázek 3.2.

Architektura mikroslužeb přináší výhody zejména v tom, že každá mikroslužba je logicky i technologicky izolovaná a nezávislá. Díky tomu je možné každou mikroslužbu

implementovat s použití různých technologií, pokud to situace vyžaduje. V kontextu softwarového vývoje může na každé mikroslužbě pracovat separátní vývojový tým dle svých vlastních procesů. Služby mohou být takto implementovány paralelně. Tím, že je proces vývoje kompletně oddělen včetně samotného nasazení, je možné dosáhnout lepší škálovatelnosti.



Obrázek 3.2: Diagram znázorňující architekturu microservice

3.1.3 Zvolené řešení

Každá architektura má své specifické případy užití, a ty mohou vyplývat jak ze samotného charakteru daného systému, tak i z nároků na infrastrukturu a operační tým, který bude za takový systém zodpovídat. I přesto, že architektura mikroslužeb přináší nesporné výhody v podobě modularity a možnosti technologické diverzity, je nutné mít na paměti, že se jedná o distribuovaný systém vnášející komplexitu do případného dalšího rozvoje a údržby systému [12]. A jelikož v současnosti, a nejspíše ani v blízké budoucnosti NFA nebude disponovat širokým technologickým zázemím, jeví se jako udržitelnější standardní vícevrstvá architektura.

3.2 Databáze

V této sekci bych se rád věnoval volbě databázového systému, který bude sloužit pro ukládání aplikačních dat. Vzhledem k povaze systému, který je založen na jednoznačně definovaných entitách a jejich vzájemných vazbách, se automaticky nabízí varianta relační databáze. Objekty identifikace by však měly obsahovat mimo jiné metadata ve formátu JSON, čímž vyvstává otázka, zda by nebylo výhodnější metadata ukládat zvlášť do nějaké dokumentové NoSQL databáze jako je např. MongoDB. Identifikovaný

objekt v relační databáze by pak byl propojený s metadatovým záznamem v MongoDB prostřednictvím cizího klíče.

V současnosti však existuje možnost ukládat JSON i v relačních databázích. Open-source databázový systém PostgreSQL, který bych na základě zkušeností s ním využil, disponuje typy JSON a JSONB. JSONB je vylepšená varianta typu JSON, která umožňuje indexování a narozdíl od typu JSON ukládá data v binárním formátu. To má za následek především rychlejší zpracování.

MongoDB oproti PostgreSQL dokáže pojmout stejná data s využitím poloviční kapacity. Jelikož však není v plánu pracovat s enormním objemem dat, PostgreSQL by měl v tomto ohledu dostačovat. Dle předpokladů je MongoDB obecně rychlejší, avšak pokud jsou data načtená v cache, PostgreSQL je dokáže vrátit rychleji [13]. S ohledem na výše zmíněné porovnání a zejména jednoduchost, se přikláním k PostgreSQL.

3.2.1 Logický datový model

Logický datový model je typ datového modelu popisující entity v systému a vztahy mezi nimi. Vychází z koncepčního modelu, na rozdíl od něj však již specifikuje pole u jednotlivých entit a taktéž upřesňuje typy vazeb. Model je však stále nezávislý na jakékoliv konkrétní platformě nebo databázovému systému [14, p. 9].

Logický datový model implementovaného systému lze nalézt na obrázku 3.3.

Fund

Entita *Fund* reprezentuje archivní fond, viz. sekce 1.2.1. Vyjma názvu obsahuje fond příznaky, zda je možné editovat objekty fondu skrze grafické uživatelské rozhraní nebo API, příp. zda jsou možné oba způsoby. Dalším příznakem je tzv. hierarchie, která určuje, jestli jsou identifikátory v rámci archivního popisu provázané. Poslední, neméně důležité pole, je reprezentací regulárního výrazu, který určuje podobu identifikátoru všech objektů identifikace náležící danému fondu.

User

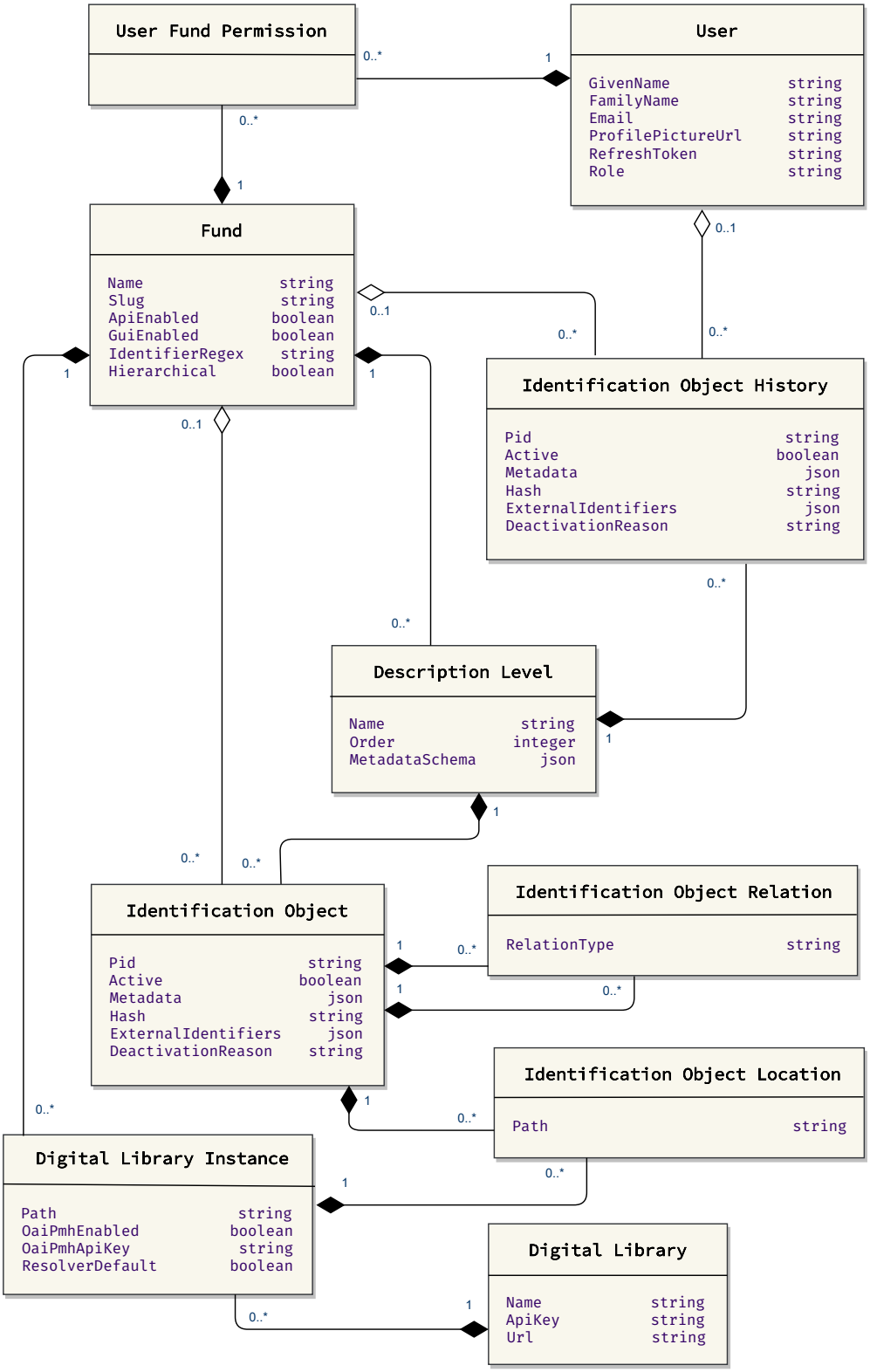
User představuje entitu, která je na základě přihlášení schopná pomocí uživatelského rozhraní interagovat se systémem. Mimo klasických polí jako jsou jméno, příjmení a emailová adresa, entita obsahuje uživatelskou roli pro odlišení úrovní přístupu a *refresh token*, který je uplatňován pro autentizaci, viz. sekce 3.4.3.

User Fund Permission

Entita sloužící jako dekompozice vztahu `M:N` mezi entitou *User* a *Fund*. Obsahuje tedy pouze odkazy na tyto entity.

Description Level

Description Level reprezentuje úroveň archivního popisu, viz. sekce 1.2.1. Úroveň archivního popisu je vztahu `N:1` vůči archivnímu fondu a její součástí je název, pořadí určující hierarchii archivního popisu a metadatové schéma. Toto metadatové schéma potom slouží pro validaci metadat objektu identifikace.



Obrázek 3.3: Logický datový model

Identification Object

Identification Object, neboli objekt identifikace, je základní jednotkou celého systému. Reprezentuje archivní objekt náležící k archivnímu fondu a podléhající určité úrovni archivního popisu. Entita obsahuje následující pole:

- perzistentní identifikátor generovaný na základě regulárního výrazu definovaného u příslušného fondu
- příznak, zda je objekt aktivní či naopak označený jako odstraněný (k samotnému fyzickému odstranění z databáze nedochází)
- metadata ve formátu JSON, která jsou validována proti schématu dané úrovně popisu
- hash sestávající se z metadat a úrovně popisu, slouží především k rychlému nalezení duplikátů při generování identifikátoru
- externí identifikátory ve formátu JSON, které však na rozdíl od metadat nejsou nijak validovány
- důvod deaktivace

Identification Object Relation

Entita sloužící jako dekompozice vztahu `M:N` mezi dvěma různými záznamy v rámci entity *Identification Object*. Obsahuje tedy pouze odkazy na tyto záznamy a typ vztahu, který mezi nimi existuje.

Digital Library

Entita *Digital Library* představuje digitální knihovnu, což může být jakýkoliv externí systém sloužící primárně pro zpřístupnění archiválií a jejich derivátů. Takováto digitální knihovna poté může sloužit jako zdroj objektů identifikace pro náš systém. Součástí této entity jsou pole popisující název knihovny, URL, na které je dostupná, a API klíč sloužící k autentizaci v rámci našeho systému při generování identifikátorů právě z těchto knihoven.

Digital Library Instance

Digital Library Instance označuje část digitální knihovny, která je součástí jednoho fondu. Digitální knihovna je často tvořena objekty z různých fondů, a toto je způsob, jak lze každý z fondů v jedné digitální knihovně zpracovávat individuálně. Instance digitální knihovny obsahuje příznaky, zda je možné sklízet záznamy pomocí protokolu OAI-PMH, a zda je tato instance výchozí instancí pro přesměrování. Dále je definována relativní cesta pro jednotlivé záznamy v kontextu digitální knihovny a příp. API klíč pro autentizaci na straně digitální knihovny při sklízení záznamů.

Obrázek 3.4: Fyzický datový model

Identification Object Location

Identification Object Location znázorňuje umístění objektu identifikace, resp. jeho reprezentace, v digitální knihovně. V rámci našeho systému se však nejedná přímo o entitu digitální knihovny, nýbrž o její instanci. *Identification Object Location* pak obsahuje pouze relativní cestu ke konkrétnímu objektu identifikace.

3.2.2 Fyzický datový model

Fyzický datový model je typ datového modelu s největší úrovní detailu. Model ilustruje jednotlivé entity a vztahy mezi nimi tak, jak jsou implementovány konkrétním databázovým systémem. Názvy jednotlivých tabulek a jejich polí včetně datových typů přesně odpovídají hodnotám v databázovém systému [14, p. 14].

Fyzický datový model implementovaného systému lze nalézt na obrázku 3.4.

3.3 API

Application Programming Interface, zkráceně API označuje rozhraní sloužící k programování aplikací. V zásadě se lze dnes setkat se třemi typy paradigmat – *command based*, *resource based* a *query based*.

V sekci 3.1 jsme na základě porovnání dvou typů architektur vybrali vícevrstvou architekturu, tzv. *layered architecture*. Vzhledem k tomu, že prezentační vrstva, tzv. frontend, poběží na samostatném virtuálním serveru a zvýající dvě vrstvy – doménová a perzistentní, tzv. backend, také, rád bych v této sekci diskutoval způsob propojení těchto dvou celků. *Command based* API by dávalo smysl spíše pro procedurálně orientovaný backend, proto se zaměřím na porovnání pouze *resource based* a *query based* API, a to konkrétně REST a GraphQL.

3.3.1 REST

Representational State Transfer neboli REST je architektonický styl používaný k implementaci webových služeb. Jak je již zmíněno v úvodu, REST je *resource based*, tzn. že každá operace probíhá na určitém zdroji. Pokud má být systém označován jako REST kompatibilní, tzv. RESTful, musí splňovat následující podmínky [15]:

1. **Client–server architektura**

   Důvodem této architektury je oddělení odpovědností mezi clientem a serverem.

2. **Bezstavovost**

   Každý request musí obsahovat všechny informace nutné k porozumění ze strany serveru.

3. **Cache**

   Každá odpověď musí obsahovat informaci, zda je zdroj cacheovatelný či ne.

4. **Jednotné rozhrani**

   Pro všechny clienty je definováno jednotné rozhraní, ať už jsou připojeni k jakémukoliv serveru.

5. **Vrstevnatost**

   Na straně serveru je možné skládat vrstvy poskytující služby jako je např. load balancer, cache atd.

Jednotné rozhraní zmíněné v bodě 4. potom podléhá následujícím omezením [15]:

1. **Identifikace zdrojů v requestu**

   Zdroj je možné identifikovat přímo z requestu např. na základě URI.

2. **Manipulace zdrojem na základě reprezentace**

   Zdrojem je možné manipulovat přímo pomocí jeho reprezentace. Způsoby manipulace jsou definovány na základě využívaného transportního protokolu. V případě HTTP se např. jedná o operace GET, POST, PUT, DELETE a PATCH.

3. **Samopopisné zprávy**

   Každá zpráva musí obsahovat informace, na základě kterých je možné zprávu zpracovat. V případě HTTP se může jednat např. o hlavičku `Content-Type` popisující formát reprezentace zdroje.

4. **HATEOAS**

   Hypermedia as the engine of application state zkráceně HATEOAS je princip, kdy server clientovi odkazy, pomocí kterých je client schopen objevit související zdroje.

Jelikož je REST pouze architektonickým stylem, nejsou jeho principy formálně vynutitelné. Drtivá většina API, která jsou označena jako RESTful, jsou tedy spíše jen jakousi variací na REST API využívající HTTP.

3.3.2 GraphQL

GrahpQL je open-source dotazovací jazyk a běhové prostředí pro realizaci těchto dotazů a vzniknul jako alternativa k REST pro tvorbu API. GraphQL je definováno schématem, které popisuje hierarchii datových typů a jejich polí. Schéma je dále tvořeno speciálními datovými typy definovaných jako dotazy, tzv. *Queries*, a mutace, tzv. *Mutations*, které může client vůči schématu exekuovat. GraphQL je inspirováno teorií grafů, kde každý typ definovaný schématem reprezentuje vrchol, a hrany spojující tyto vrcholy znázorňují vztahy mezi těmito typy.

Typ *Query* reprezentující operaci čtení definuje všechny přístupové body, tzv. *entry points*, kterých může client využít při dotazování proti definovanému schématu. Typ *Mutation* funguje na stejném principu, s tím rozdílem, že reprezentuje operaci zápisu.

Díky definovanému schématu včetně použitých typů, GraphQL při každém realizovaném dotazu vyžaduje přesnou specifikaci dat, která se z API mají vrátit. Navíc je možné v jednom dotazu specifikovat data i na základě definovaných vazeb.

3.3.3 REST vs GraphQL

Představme si jednoduchý datový mode složený z entit `Film` a `Director`, kdy entita `Film` obsahuje referenci na entitu `Director` v podobě `ID`. Na straně klienta máme k dispozici název filmu a chceme zjistit jméno a příjmení režiséra. V případě REST API by první volání vypadalo nejspíše nějak takto:

```
GET /api/film?title=markéta+lazarová
```

Následná odpověď ve formátu JSON:

```
{
  "id": 1,
  "title": "Markéta Lazarová",
  "year": 1967,
  "directorId": 1
}
```

Tímto requestem jsme získali `ID` režiséra, na jehož základě můžeme uskutečnit druhé volání API:

```
GET /api/director?id=1
```

A následná odpověď:

```
{
  "id": 1,
  "name": "František",
  "surname": Vláčil,
  "born": 1924
}
```

K tomu abychom získali na základě názvu filmu jméno a příjmení jeho režiséra, byly potřeba dva requesty. V případě GraphQL bychom nejdříve museli definovat následující schéma:

```
type Film {
  "id": ID,
  "title": String,
  "year": Int,
  "director": Director
}

type Director {
  "id": ID,
  "name": String,
  "surname": String,
  "born": Int,
```

```
  "films": [Film]
}

type Query {
  "film": Film
}
```

V kontextu tohoto schématu poté můžeme exekuovat následující dotaz:

```
query GetFilm {
  film {
    director {
      name
      surname
    }
  }
}
```

A následná odpověď:

```
{
  "data": {
    "film": {
      "director": {
        "name": "František",
        "surname": "Vláčil"
      }
    }
  }
}
```

S pomocí GraphQL se podařilo jedním requestem získat požadovaná data. Určitě by bylo možné i REST API navrhnout tak, aby bylo možné požadavek splnit za použití jediného requestu, flexibilita je nicméně určitě na straně GraphQL. S REST API se také pojí problém tzv. *over-fetchingu*, což znamená, že množství příchozích dat často bývá větší než je ve skutečnosti nutné. Lze si toho všimnout i na příkladu výše, kdy pro získání jména a příjmení režiséra bylo nutné skrze API zaslat všechna pole. Naproti tomu u GraphQL je nutné vždy explicitně uvádět požadovaná pole.

Mezi výhody REST API lze určitě zařadit cachování, které lze zajistit i u GraphQL, je však složitější na implementaci a jsou nutné dodatečné nástroje. Odpovědi v případě GraphQL jsou redukované na status kód `200` pro všechny validní odpovědi včetně chyb jako jsou např. nedostatečná oprávnění, validace atd. V případě chybného requestu např. na neexistující zdroj, přichází odpověď se status kódem `400`. Běžných uživatelské chyby je tedy nutné ošetřit přímo v implementaci.

Implementace GraphQL na straně serveru je určitě komplexnější než implementace REST API, v kontextu výhod zmíněných výše se však přikláním ke GraphQL.

3.4 Autentizace

Komunikace mezi klientem a serverem je zajištěna prostřednictvím HTTP. Tento protokol je ze své povahy bezstavový, tzn. že i všechny requesty z klienta na server přímo se stavem nepracují. Pokud bychom měli aplikaci, kde by se uživatel nejdříve přihlásil, a poté přešel např. na stránku s nastavením uživatelského profilu, bylo by znovu vyžadováno přihlášení, jelikož by server neměl tušení, že předtím již k přihlášení jednou došlo. Taková aplikace by z uživatelského pohledu nebyla moc přívětivá. Abychom takovým situacím předešli je nutné využít mechanismu, který zajistí, že server bude při každém requestu obeznámen, s jakým uživatelem pracuje. V zásadě existují dvě možnosti, jak toho dosáhnout – *session based autentizace* a *token based autentizace*.

Kromě autentizace mezi klientem se serverem, je nutné zajistit i autentizaci mezi serverem a externími systémy, v našem případě se jedná o tzv. digitální knihovny. Taková digitální knihovna by výměnou za metadata a úroveň popisu dostala od identifikačního systému perzistentní identifikátor.

3.4.1 Session based authentication

Session based authentication neboli autentizace založená na relaci, tzv. session, dochází ze strany serveru k vytvoření session vždy, když dojde k přihlášení uživatele. Session id poté bývá uložené na straně klienta v cookie přímo v prohlížeči uživatele. S každým dalším requestem klienta je zaslána i tato cookie obsahující session id. Server potom dokáže uživatele verifikovat na základě porovnání session id se session údaji uloženými přímo v databázi, a klientovi zaslat response včetně příslušného stavu, což může být např. uživatelský profil zmíněný výše. Z důvodu bezpečnosti dochází po určité době, kdy je klient neaktivní, k expiraci session, a to jak na straně serveru, tak na straně klienta expirací session cookie. Při další interakci je uživatel následně vyzván k opětovnému přihlášení.

3.4.2 Token based authentication

Token based authentication neboli autentizace využívající tokeny narozdíl od session based autentizace neudržuje stav na serveru, ale na klientovi. Jak již název napovídá, zakódovaný stav je udržován ve formě tokenu, tj. textového řetězce, a to nejčastěji v cookie, příp. v local storage. Token může mít různou podobu, v současnosti je nejpoužívanější typem tokenu JSON Web Token – JWT.

JWT

JSON Web Token je textový řetězec reprezentující kompaktní způsob přenosu dat ve formátu JSON mezi dvěma subjekty. Base64 zakódovaný JWT může vypadat třeba takto:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEzMT
IiLCJuYW1lIjoiSm9zZWYgVsOtdG92ZWMiLCJpYXQiOjE0MjI3N
zk1MTAsImVtYWlsIjoiam9zZWYudml0b3ZlY0BmZWwuY3Z1dC5j
eiJ9.B98yTFyXVh3Av9cS08yWSul2mfJ_7ISjGfRKuoe4rIY
```

Z hlediska struktury se token skládá ze tří částí oddělených tečkou – hlavičky (header), samotných dat (payload) a podpisu (signature). U tokenu výše jsou tyto části rozlišené barvami – hlavička je červeně, data růžově a podpis modře. Po dekódování části obsahující hlavičku dostaneme následující:

```
{
  "alg": "HS256",
  "typ": "JWT",
}
```

Z hlavičky lze vyčíst, že enkódovaný objekt je JWT podepsaný algoritmem HMAC SHA-256. Samotná data po dekódování potom vypadají takto:

```
{
  "id": "1312",
  "name": "Josef Vítovec",
  "email": "josef.vitovec@fel.cvut.cz",
  "iat": 1422779510,
}
```

Součástí těchto dat může být prakticky cokoliv, co není v rozporu formátu JSON. Navíc je možné přidat speciální pole, tzv. *claims*, která mají v kontextu tokenu určitý význam. V datech výše to je např. pole `iat`, které reprezentuje, kdy byl token vytvořen. Na základě toho je potom možné po nějaké době považovat token za nevalidní. JWT podléhá specifikaci RFC 7519, kde lze nalézt i všechny tato definovaná pole. Poslední složkou JWT je podpis, který je vytvořen na základě následujícího výpočtu výpočtu:

```
HMAC-SHA256(
  secret,
  base64urlEncode(header) + '.'
+
  base64urlEncode(payload)
)
```

Pomocí tohoto výpočtu lze i při přijetí tokenu ověřit, jestli hlavičky a data souhlasí s podpisem. A pokud hodnoty souhlasí, máme zaručeno, že token nebyl během svého přenosu nikým modifikován. Je však potřeba si uvědomit, že data nejsou mimo kódování nijak chráněna, a číst je tedy může kdokoliv.

Cílem JWT je poskytnout jednoduchý bezpečnostní token, jehož velikost ho nelimituje při použití v HTTP hlavičkách a zároveň je možné ho využít i jako *query* parametr v URI. V porovnání např. se SAML tokenem je JWT o poznání menší, a to i díky jednoduchému modelu, a použití úspornějšího JSON formátu namísto XML [16].

3.4.3 Zvolené řešení

JWT či spíše *token based authentication* obecně, narozdíl od *session based authentication*, nevyžaduje při ověření tokenu na straně serveru přístup do databáze. Veškerá data nutná

k verifikaci jsou již obsažená v samotném tokenu, na serveru je potřeba mít uložený pouze *secret key*, který slouží ke generování nových a ověřování příchozích tokenů. Díky tomu lze autentizace založená na tokenech vychází lépe z hlediska horizontální škálovatelnosti. S použitím tokenů je navíc jednodušší implementovat kompletní proces autentizace na jiném odděleném serveru a také kontrolovat autentizaci napříč vícero zařízeními.

Jako největší nevýhodu tokenů je možné považovat neschopnost invalidace, jelikož veškerá logika je uložena v tokenu samotném a tudíž neexistuje mechanismus, který by token dokázal zvenčí zneplatnit. JWT může mít definované pole `exp`, které znázorňuje expiraci tokenu. Pokud ale např. dojde k odhlášení uživatele, JWT bude platný i poté, a to až do doby definované právě v poli `exp`. Obecně je autentizace komplexní a v komunitě hojně debatované téma, a i přesto, že JWT dnes patří mezi velmi oblíbené způsoby autentizace, existují i názory, že oproti klasické session autentizaci nepřináší žádné výhody [17].

Jelikož ale věřím, že je možné nedostatky JWT minimalizovat, viz níže, a zároveň existuje teoretická šance, že by navrhovaný systém mohl být v budoucnu v kontextu NFA klíčový, vnímám z hlediska budoucího rozvoje vhodnější variantu za použití JWT.

Přihlášení

Jak je uvedeno v sekci 3.4.3, možnost přihlášení by měli mít pouze zaměstnanci NFA disponující emailovou adresou na doméně nfa.cz. Jelikož NFA využívá služeb Google Workspace[1], dříve známé jako G Suite, je možné pro přihlášení využít již existující identitu poskytovanou právě Googlem. Pro integraci tohoto řešení do aplikace je nutné vytvořit nového OAuth 2.0 klienta v konzoli Googlu. Poté již stačí pouze využít JavaScriptové knihovny poskytované Googlem nebo využít některého z řešení třetích stran, které však rovněž vychází z této knihovny [18].

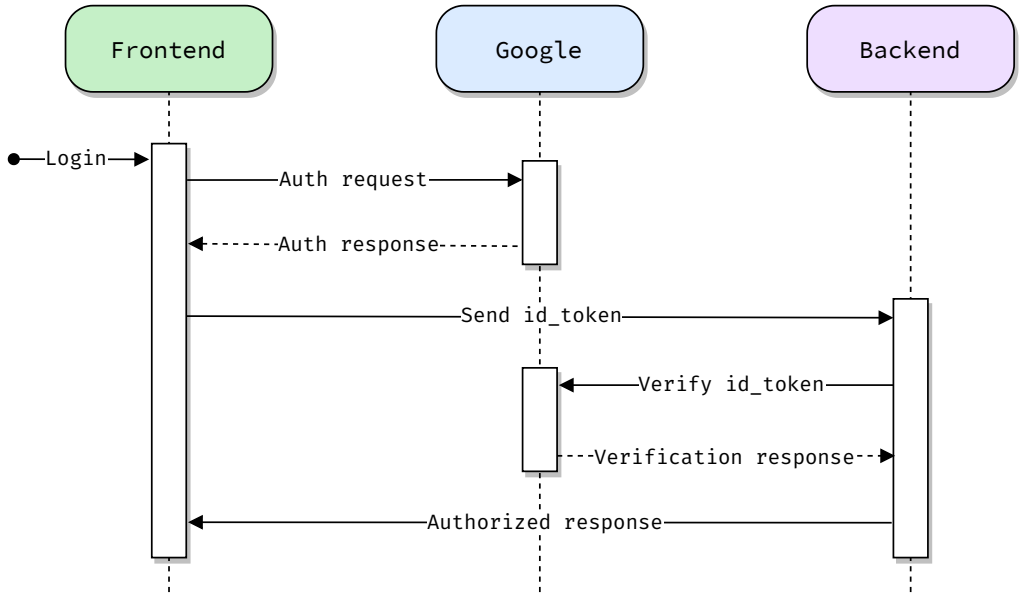
Z pohledu uživatele celý proces probíhá tak, že po kliknutí na tlačítko pro přihlášení dochází k otevření nového panelu se standardním přihlášením do systémů Google. Po úspěšném vyplnění jména a hesla se panel zavře a uživatel je autentizovaný.

Naše aplikace bude rozdělená na frontendovou a backendovou část, které navíc budou běžet na oddělených serverech. Nejdříve tedy ve frontendové části dojde k ověření uživatele ze strany Googlu, token získaný z odpovědi však bude nutné zaslat na backend, kde dojde k jeho verifikaci, a následnému vytvoření příp. aktualizaci uživatele v databázi. Až po tomto procesu budeme moct prohlásit uživatele v kontextu naší aplikace za autentizovaného. Celý tento proces je znázorněn na obrázku 3.5.

Autorizace requestů

Přihlášení popsané v předchozí sekci zajistí, že má do aplikace přístup pouze uživatel disponujícím emailovou adresou na doméně nfa.cz. Vzhledem k tomu, že by ani žádné další stránky neměly být veřejně přístupné, je potřeba zajistit autorizaci všech dalších requestů, které budou po přihlášení následovat. K tomu právě využijeme JWT přidělované naším backendem.

[1] https://workspace.google.cz



Obrázek 3.5: Sekvenční diagram znázorňující proces přihlášení

V sekci 3.4.2 je zmíněno, že za nevýhodu JWT lze označit nemožnost invalidace tokenu. Abychom předešli bezpečnostním rizikům, expirace JWT bude nastavena na 15 minut – takový token lze označit jako *access token*. Pokud bychom využívali pouze tento *access token*, znamenalo by to, že by se uživatel musel po každé čtvrthodině znovu přihlásit, což by rozhodně nebylo příliš uživatelsky přívětivé. Z toho důvodu aplikace bude využívat i tzv. *refresh token* s expirací 30 dní, který bude sloužit pro opětovné získání nového *access tokenu*. U refresh tokenu není vyžadováno, aby se jednalo o JWT, jelikož uživatelská data už jsou součástí JWT, který reprezentuje *access token*. Bude se tedy jednat o náhodně vygenerovaný textový řetězec. *Refresh token* je navíc nutní udržovat v databázi, jelikož je přidružený vždy ke konkrétnímu uživateli.

Zbývá ještě rozhodnout, jak by mělo docházet k výměně tokenů mezi backendem a frontendem a kde by měly být na straně uživatele uloženy. Pro každý token máme v zásadě tři možnosti:

1. paměť
2. cookies
3. local storage

Mezi hlavními kritérii, která určují k jaké možnosti se přiklonit, patří zejména odolnost vůči následujícím typům útoků:

- **Cross Site Scripting (XSS)**

  Zranitelnost, při které dochází ze strany útočníka k podstrčení vlastního kódu do dynamické webové aplikace. Útočník takto např. může získat citlivá data z prohlížeče uživatelů a na základě toho se za ně při komunikaci se serverem vydávat.

- **Cross Site Request Forgery (CSRF)**

  Zranitelnost, při které útočník donutí přihlášeného uživatele (často bez jeho vědomí) k vykonání určité škodlivé akce. Využívá toho, že většina aplikací má informace nutné k provedení autorizovaného požadavku uložené přímo v prohlížeči uživatele.

Pro *access token* zasílaný s každým requestem se jako nejvíce bezpečná zdá varianta uložení přímo v paměti. Uložení v cookie bez využití relativně nového atributu *SameSite*, by znamenalo největší náchylnost na CSRF útoky. Z hlediska XSS samospásná varianta neexistuje, jelikož už ze samotné povahy XSS má útočník aplikaci v moci a dokáže získat data ze všech uvažovaných zdrojů [19]. Nicméně je možné útočníkovi situaci, co nejvíce znesnadnit, a z toho vychází nejlépe uložení JWT přímo v paměti.

Refresh token token musí být uložen přímo v prohlížeči uživatele, v paměti by jeho existence pozbývala smyslu, jelikož by s každým zavřením okna prohlížeče zmizel, a bylo by tedy znovu nutné přihlášení. V prohlížeči se potom jako lepší jeví varianta uložení v cookie s atributem *HttpOnly*, který zajišťuje nedostupnost cookie z klientského JavaScriptu. Cookie obsahující *refresh token* není navíc zranitelná vůči CSFR, jelikož i pokud by se požadavek útočníkovi podařilo odeslat, vygenerovaný JWT by byl obsažený v těle odpovědi a k obnovení tokenu by tedy nakonec nedošlo.

Autentizace externích systémů

Pro autentizaci příchozích requestů z externích systémů, které jsou v kontextu našeho systému reprezentované digitálními knihovnami, bude nejjednodušší a nejefektivnější variantou použití API klíče. API klíč lze chápat jako vygenerovanou sekvenci alfanumerických znaků sloužících k ověření uživatele či systému na základě porovnání předloženého klíče s klíčem uloženým na straně provozovatele API. Klíč bude přítomen v hlavičce každého requestu.

Kapitola 4

Implementace

V této kapitole bych rád popsal některé z frameworků a knihoven, které jsem při implementaci identifikačního systému využil. Všechny tyto technologie jsou *open-source* dostupné skrze správce balíčků `npm`.

4.1 Server

4.1.1 Použité technologie

Node.js

Node.js je JavaScriptové *event-driven* běhové prostředí. *Event-driven* architektura spočívá zejména v existenci tzv. *event loopu*, neboli smyčky událostí příjímající uživatelské požadavky jako události, které jsou poté přidělovány jednotlivým vláknům. Na rozdíl od klasického JavaScriptu, který je určen pro spouštění přímo v prohlížeči, je Node.js naopak určené pro server.

Součástí Node.js je i balíčkovací systém npm, který byl rovněž využit při implementaci systému.

Express

Express je webový aplikační framework pro Node.js. Slouží převážně pro tvorbu webových aplikací a API. V současnosti se jedná o nejrozšířenější aplikační framework postavený nad Node.js.

TypeScript

TypeScript je programovací jazyk postavený nad JavaScriptem. Oproti JavaScriptu však zavádí statické typování a některé další funkcionality OOP jako např. třídy, rozhraní, moduly atd. Kód psaný v TypeScriptu je nutné kompilovat do JavaScriptu. Knihovny původně napsané v JavaScriptu je možné rozšířit o tzv. hlavičkové soubory, které definují rozhraní a typy dané knihovny. Na základě toho je možné potom využívat knihovnu v TypeScriptu i včetně silného typování.

TypeORM

TypeORM je knihovna zajišťující objektově relační mapování, což je stěžejní součást datové vrstvy zabezpečující transport a konverzi dat mezi aplikací a databází. Jak již název napovídá, knihovna podporuje TypeScript, a nabízí kompletní ORM funkcionality od generování migrací přes definici entit a jejích polí pomocí dekorátorů až po tvorbu vlastních *repositories*.

TypeGraphQL

TypeGraphQL je framework pro tvorbu GraphQL API. Tvorba GraphQL schématu je vcelku složitý proces – vyžaduje definici typů (včetně *queries* a *mutations*), rozhraní, enumů, unionů atd. Takto vytvořené schéma je často velmi složité na údržbu. Dojde ke změně ORM entity a je nutné celé schéma projít a zapracovat potřebné změny. Tento problém řeší právě TypeGraphQL, které umožňuje využít již existující ORM entity ke generování schématu. Umožňuje např. také definovat tzv. *resolvery*, což jsou třídy obsahující metody představující jednotlivé *queries* a *mutations*. Takto implementovaný systém je potom flexibilnější vůči příp. změnám a obecně udržitelnější do budoucna. TypeGraphQL také podporuje TypeScript a navíc je vcelku lehko integrovatelný i s TypeORM.

Jest

Jest je JavaScriptový framework sloužící k vytváření a spouštění testů. Díky existenci hlavičkových souborů je možné knihovnu využívat i v TypeScriptových projektech. Jest je vhodný pro unit i integrační testy a nabízí široké možnosti konfigurace včetně mockování, snapshotů atd. Testování je podrobněji popsáno v kapitole 5.

Apollo Server

Apollo Server je GraphQL server, který vzniká na základě GraphQL schématu, v našem případě, vygenerovaným frameworkem TypeGraphQL. Samotný server lze pak zaintegrovat jako middleware do Expressu. Součástí serveru je i webové testovací prostředí tzv. *GraphQL Playground*.

TypeDI

TypeDI je knihovna, na základě níž lze v prostředí TypeScriptu využívat *dependency injection*, a tím značně zpřehlednit kód a zajistit lepší testovatelnost.

randexp

Knihovna umožňující generovat náhodné řetězce na základě regulárního výrazu. V kontextu implementovaného systému je využita při generování identifikátorů.

jsonwebtoken

Knihovna zajišťující generování a ověřování JWT.

ajv

Knihovna sloužící pro validaci libovolného JSONu proti definovanému JSON schématu. V našem systému jsou pomocí JSON schématu definována povinná a volitelná pole pro metadata u jednotlivých úrovní popisu. Při generování identifikátoru pak dochází k validaci předložených metadat právě vůči tomu JSON schématu.

uuid

Knihovna umožňující generovat různé typy UUID. Náš systém toto UUID využívá např. pro generování *refresh tokenu*.

4.1.2 Entity

Entita je základním prvkem TypeORM reprezentující objekt vycházející z datového modelu. Definuje jednotlivá pole a jejich vlastnosti v rámci databázového systému. Na základě entit bývají často ORM frameworky generovány migrace.

Část jedné takové entity, konkrétně objekt identifikace, můžeme vidět na výpisu kódu 4.1. Entita je mapována přímo na tabulku `identification_object` a definuje dvě pole – `hash` a `fund`. Pomocí dekorátorů lze zajistit některá požadovaná omezení, jako např. unikátnost dvojice `hash` a `fund` v rámci tabulky. `Fund` je zároveň využit pro vyjádření vazby `M:1` s entitou `Fund` pomocí cizího klíče `fund_id`.

V kontextu našeho systému entity neslouží jen ORM, ale jsou rovněž využívány i jako typy pro GraphQL schéma (dekorátory `@ObjectType` a `@Field`). Tím se výrazně zjednodušuje údržba celého systému.

```
1  @Entity("identification_object")
2  @Unique(["fund", "hash"])
3  @ObjectType()
4  export class IdentificationObject extends AbstractEntity {
5
6      @Field()
7      @Column({ name: "hash", type: "varchar", length: 100 })
8      @Index()
9      hash: string;
10
11     @Field(() => Fund)
12     @ManyToOne(() => Fund, fund => fund.identificationObjects, {
13         onDelete: 'CASCADE',
14         lazy: true
15     })
16     @JoinColumn({ name: 'fund_id' })
17     fund: Fund;
18
```

Výpis kódu 4.1: Část entity IdentificationObject

4.1.3 Repositories

Repositories je označení pro třídy operující na datové vrstvě vícevrstvé architektury. Každá *repository* je tvořena metodami, které zajišťují základní operace nad danou entitou. Samotné metody jsou pak daným ORM „překládány“ do podoby SQL dotazu.

Ve výpisu kódu 4.2 je zobrazena *repository* k entitě Fund obsahující dvě asynchronní metody – `getByName` a `getBySlug`. Jelikož `FundRepository` dědí z `Repository<Fund>` lze využít i standardní metody definované TypeORM jako např. `create`, `find`, `findOne` nebo `save`.

```
1  @EntityRepository(Fund)
2  export class FundRepository extends Repository<Fund> {
3    public async getByName(name: string): Promise<Fund> {
4      return await this.findOne({
5        where: { name }
6      })
7    }
8
9    public async getBySlug(slug: string): Promise<Fund> {
10     return await this.findOne({
11       where: { slug }
12     })
13   }
14 }
```

Výpis kódu 4.2: FundRepository

4.1.4 Services

Jako *services* označujeme třídy implementující doménovou, nebo tzv. business, logiku. V případě výpisu kódu 4.3 se jedná o ověření příchozího Google JWT s využitím Google Auth Library. Dekorátor `@Service` potom slouží knihovně TypeDI, která je na základě tohoto dekorátoru schopna *service* injektovat např. do některého z *resolverů*.

```
1  @Service()
2  export class AuthService {
3
4    public async verifyJwtGoogle(token: string): Promise<TokenPayload> {
5      const ticket = await oauthClient.verifyIdToken({
6        idToken: token,
7        audience: config.GOOGLE_CLIENT_ID,
8      });
9
10     return ticket.getPayload();
11   }
12 }
```

Výpis kódu 4.3: AuthService

4.1.5 Middleware

V kontextu webového vývoje pod frameworkem Express, označujeme pojmem middleware funkci, která má přístup k request objektu, response objektu a další middleware funkci v pořadí. Prostřednictvím těchto funkcí je možné realizovat např. globální *error-handling* nebo třeba povolení CORS.

V rámci našeho systému se middleware využívá pro autentizaci. Pokud se jedná o běžný request z uživatelského rozhraní, tak by součástí takového requestu měla být hlavička `authorization` obsahující JWT. Jak je vidět ve výpisu kódu 4.4, pokud taková hlavička neexistuje vrací se clientovi chyba. V případě, že hlavička s tokenem existuje, dojde k ověření tokenu, a pokud je token validní, dojde k zavolání další middleware v pořadí, příp. přímo požadovaného endpointu.

```
1  export class CheckJwtMiddleware implements MiddlewareInterface<AuthContext> {
2
3    @Inject()
4    private authService: AuthService;
5
6    use = async ({ context }: ResolverData<AuthContext>, next: NextFn) => {
7
8      const token = this.authService.getTokenFromHeader(context.req.headers);
9      if (!token) {
10       throw new AuthenticationError("auth token not valid");
11     }
12
13     await this.authService.verifyJwt(token).then(data => {
14       context.jwtPayload = data;
15       return next();
16     }).catch(error => {
17       throw new AuthenticationError("auth token not valid");
18     });
19   }
```

Výpis kódu 4.4: CheckJwtMiddleware

4.1.6 Resolvers

Resolvers označují ve frameworku TypeGraphQL třídy, které sdružují *queries* a *mutations* náležící jedné entitě. Může se např. jednat o GraphQL reprezentace CRUD operací, jako je tomu ve výpisu kódu 4.5, často však bývají složitější. Každá tato metoda má definované parametry a návratový typ. Součástí metoda může být i dekorátor `@UseMiddleware` značící, že nejdříve request projde přes middleware. V tomto případě se konkrétně jedná o middleware zobrazený ve výpisu kódu 4.4.

```
@Service()
@Resolver()
export class FundResolver {

  @Query(() => Fund)
  @UseMiddleware(CheckJwtMiddleware)
  async fund(@Arg("id") id: number): Promise<Fund> {
    return await this.fundRepository.findOne(id);
  }

  @Mutation(() => Boolean)
  @UseMiddleware(CheckJwtMiddleware)
  async fundDelete(@Arg("id") id: number): Promise<Boolean> {
    const result = await this.fundRepository.delete({ id });

    return !!result.affected;
  }
}
```

Výpis kódu 4.5: FundResolver

4.2 Client

4.2.1 Použité technologie

React

React je JavaScriptová knihovna sloužící k tvorbě uživatelských rozhraní. Jedná se dnes o jedno z nejoblíbenějších řešení pro tzv. *single-page* aplikace.

Základní jednotkou je komponenta, pod kterou si můžeme představit množinu HTML elementů s definovanou funkcionalitou. Každá komponenta má pak svoje *props*, což si lze představit jako vlastnosti, a navíc může mít i definovaný *state*, neboli stav. *Props* slouží uvnitř komponenty pouze pro čtení, zatímco *state* lze přímo měnit.

Apollo Client

Apollo Client je knihovna zajišťující dotazování na GraphQL API včetně souvisejících činností jako je např. cachování.

axios

Knihovna axios funguje jako wrapper nad `XMLHttpRequest`, která využívá funkcionalitu z ES6 známou jako *Promise*.

antd

UI knihovna antd určená pro React poskytuje kompletní řešení pro tvorbu uživatelských rozhraní. Knihovna definuje vlastní *grid* systém včetně nejrůznějších předdefinovaných komponent.

4.2.2 Context

Context je datová struktura v knihovně *React* schopná sdílet data se všemi ostatními komponentami. Běžná komponenta může svůj stav sdílet pouze komponentám, která se ve stromu komponent nachází bezprostředně pod ní. Takovýto přístup může v důsledku vést k tzv. *prop drilling*, což je negativní jev, při kterém dochází k předávání dat mezi dvěma komponentami způsobem, kdy všechny komponenty mezi nimi pouze pasivně posílají data dále.

Context je ideální pro použití s daty, která se nemění příliš často (např. real-time). V našem případě, viz. výpis kódu 4.6 se jedná o autentizační data z JWT, na základě kterých se potom mění vykreslování pro jednotlivé uživatele.

```
1  const [login] = useMutation(LOGIN);
2
3  const clientId = process.env.REACT_APP_GOOGLE_CLIENT_ID
4
5  const onLoginSuccess = (googleResponse) => {
6    setJwt(googleResponse.getAuthResponse().id_token)
7
8    login().then(response => {
9      setJwt(response.data.login.jwt)
10
11     const jwtDecoded = jwtDecode(response.data.login.jwt);
12     setRole(jwtDecoded.role);
13     setFundsPermissions(jwtDecoded.funds);
14     setIsSignedIn(true);
15   }).catch(error => {
16     console.log(error)
17   })
18 }
```

Výpis kódu 4.6: AuthContext

4.2.3 Cachování

Jak je popsáno v sekci 3.3.3, za jednu z nevýhod GraphQL lze považovat cachování. Zatímco u REST API je možné využívat standardizované HTTP hlavičky, na poli GraphQL musíme spoléhat na řešení konkrétních knihoven.

V rámci klientské knihovny Apollo je ve výchozím nastavení implicitně definována in-memory cache. Pro obcházení cache definuje knihovna několik způsobů. Jeden z těch jednodušších na implementaci je součástí výpisu kódu 4.7. Umožňuje k definici

mutation připojit parametr `refetchQueries`, což je *query*, k jejímuž spuštění dojde bezprostředně po vykonání zmíněné *mutation*. Na jednu stranu se jedná o rychlý způsob, jak obejít cache, na druhou stranu `refetchQueries` vyžaduje síťový požadavek navíc. V kontextu našeho systému se však zatím jedná o dostačující řešení.

```
1  const result = await digitalLibraryUpdate({
2    variables: {
3      id: props.match.params.id,
4      digitalLibrary: {
5        name: values.name,
6        url: values.url,
7        apiKey: values.apiKey,
8      }
9    },
10   refetchQueries: [{ query: DIGITAL_LIBRARIES }]
11 })
```

Výpis kódu 4.7: Obcházení cache pomocí `refetchQueries`

Kapitola 5

Testování

Testování je nezbytnou součástí vývoje software, jehož cílem je minimalizovat počet chyb v systému. Čím dříve chybu objevíme, myšleno v souvislosti s fázemi softwarového vývoje, tím jednodušší bude i její odstranění. Softwarové testování lze kategorizovat na základě mnoha různých parametrů. Součástí této kapitoly bude zejména rozdělení dle úrovní testování na:

- unit testy
- integrační testy
- funkční testy

5.1 Unit testy

Úkolem unit testů je zajistit testování nejmenších částí kódu, tzv. jednotek. Tyto jednotky si lze představit jako funkce či metody. Cílem unit testů je mít testovaný kód co nejvíce izolovaný od zbytku aplikace, abychom se vyhnuli případným vnějším vlivům. Takovéto testy se potom většinou pouštějí před každým *buildem* aplikace, jelikož je jejich běh v porovnání s ním zanedbatelný.

5.2 Integrační testy

Integrační testy slouží pro komplexnější testování systému, při kterém jsou předmětem testování interakce mezi různými komponentami. V následujících dvou sekcích jsou zmíněny dva příklady integračních testů, které byly využity v implementovaném systému.

5.2.1 Databáze

Jedním z nejčastějších typů integračního testování je testování datové vrstvy. Abychom tyto testy odstínili od produkčních dat, je nutné vytvořit separátní databázi, která bude sloužit jen pro účely těchto testů. K zajištění izolace jednotlivých testů je pak potřeba udržovat databázi v konzistentním stavu.

V kontextu našeho systému se pod integračními testy databáze rozumí testování *repositories* zmíněných v sekci 4.1.3. Celý proces začíná už před samotným spuštěním testů, kdy dojde k *dropu* celé databáze včetně schématu. Schéma se poté znovu celé vytvoří na základě migrací. Před každou testovací sadou navíc probíhá vymazání všech dat z databáze, jak ilustruje výpis kódu 5.2. Těmito kroky je zajištěna absolutní nezávislost testů na aktuálním stavu dat v databázi.

```
1  import { DatabaseUtils } from "../utils";
2
3  beforeAll(async () => {
4      await DatabaseUtils.createConnection({ dropSchema: false })
5  })
6
7  afterAll(async done => {
8      await DatabaseUtils.close();
9      done();
10 })
11
12 beforeEach(async () => {
13     await DatabaseUtils.clear();
14 });
```

Výpis kódu 5.1: Obcházení cache pomocí `refetchQueries`

5.2.2 GraphQL

Dalším typem integračních testů je testování API. V našem konkrétním případě se jedná o GraphQL API. Jelikož se má jednat o integrační testování, při kterém by se měla otestovat samotná komunikace v rámci systému, neměli bychom se spokojit s *mockováním* celého rozhraní.

V kontextu GraphQL a našeho systému bylo pro tyto účely tedy nutné vytvořit testovací *Apollo Server*, proti kterému by se exekuovali *queries* a *mutations*. Jelikož je GraphQL z hlediska výstupu o něco méně různorodé než klasické REST API, bylo využito tzv. snapshotů, ve kterých je uložená očekávaná odpověď – v našem případě tedy výstup z API GraphQL. Test potom tedy spočívá v tom, že dochází k porovnání získané odpovědi s tou ze *snapshotu*. Příklad části takého testu je uveden ve výpisu kódu

```
1  const { mutate } = createTestClient(server);
2    const res = await mutate({
3      mutation: IDENTIFICATION_OBJECT_CREATE,
4      variables: {
5        identificationObject: identificationObject
6      },
7    });
8
```

```
9   expect(findOneMock).toHaveBeenCalled();
10  expect(res).toMatchSnapshot();
```

Výpis kódu 5.2: Testovací GraphQL server a využití *snapshotů*

5.3 Funkční testy

Posledním a nejkomplexnější typem z výše zmíněných jsou funkční testy. Funkční testování simuluje reálné chování uživatele v interakci se systémem. Do těchto testů jsou zapojeny všechny vrstvy aplikace. Vzhledem k jejich náročnosti na údržbu a i z hlediska nutných nástrojů a znalostí potřebných k vytvoření takovýchto testů, se využívají v omezené míře. V souvislosti s tímto systémem žádný takový test nevzniknul.

Kapitola 6

Závěr

Tato práce si kladla za cíl vyvinout prototyp aplikace zajišťující generování, uchovávání a správu jednoznačných perzistentních identifikátorů pro potřeby Národního filmového archivu. Jelikož výsledek práce bude dále užíván a rozvíjen v prostředí paměťové instituce, pokusil jsem se v úvodní kapitole také definovat několik pojmů souvisejících s archivnictvím jako jsou např. archivní fond či úroveň archivního popisu. Některé z těchto pojmů byly, myslím, nezbytné pro celkové pochopení dané problematiky.

Mimo samotné implementace byla práce z velké části analytická, kdy se v úvodní kapitole zaobírala samotnou problematikou digitální identifikace a definicí požadavků, které by měl systém, schopný perzistentní identifikace, naplňovat. V kontextu těchto požadavků byly představeny existující schémata identifikace včetně jejich způsobů užití, a výhod či nevýhod z nich plynoucích. Rovněž došlo i na shrnutí současného stavu v Národním filmovém archivu z hlediska perzistentní identifikace.

Kapitola návrh spočívala zejména v diskuzi nad existujícími technologickými řešeními a jejich využitelností v kontextu implementovaného systému. Ať už se jednalo o architekturu, nebo způsob autentizace, všechny představené technologie byly podrobeny důkladnému rozboru s důrazem na udržitelnost a budoucí možný rozvoj.

Lze říci, že implementovaný systém splňuje všechny hlavní požadavky, které na něj byly kladeny. Na druhou stranu se zatím stále jedná spíše o prototyp, než o plnohodnotnou aplikaci určenou pro produkční provoz. Touto diplomovou prací však moje fungování v rámci Národního filmového archivu nekončí, tudíž vývoj identifikačního systému bude dále pokračovat.

Literatura

[1] Bratkova, E. Síť trvalých identifikátorů informačních entit. 2005. Dostupné z: <https://sites.ff.cuni.cz>

[2] Národní knihovna ČR. Česká terminologická databáze knihovnictví a informační vědy. 2003. Dostupné z: <https://aleph.nkp.cz>

[3] MICHAL WANNER A KOL. *Základní pravidla pro zpracování archiválií*. Odbor archivní správy a spisové služby Ministerstva vnitra ČR, 2015. Dostupné z: <https://www.mvcr.cz/soubor/zakladni-pravidla-pro-zpracovani-archivalii-2015-cervene-vyznacenymi-zmenami.aspx>

[4] Hilse, H. W.; Kothe, J. *Implementing Persistent Identifiers: Overview of Concepts, Guidelines and Recommendations*. Consortium of European Research Libraries, 2006, ISBN 9789069845081. Dostupné z: <https://books.google.cz/books?id=H4R2QgAACAAJ>

[5] Cubr, L. *Autenticita a digitální informace [Authenticity and digital information]*. Dissertation thesis, 2017.

[6] Berners-Lee, T.; Fielding, R. T.; et al. Uniform Resource Identifier (URI): Generic Syntax. 2005, doi:10.17487/RFC3986. Dostupné z: <https://rfc-editor.org/rfc/rfc3986.txt>

[7] Miessler, D. What's the Difference Between a URI and a URL? 2020. Dostupné z: <https://danielmiessler.com/study/difference-between-uri-url/>

[8] Masinter, L. M.; Sollins, D. K. R. Functional Requirements for Uniform Resource Names. 1994, doi:10.17487/RFC1737. Dostupné z: <https://rfc-editor.org/rfc/rfc1737.txt>

[9] Moats, R. URN Syntax. 1997, doi:10.17487/RFC2141. Dostupné z: <https://rfc-editor.org/rfc/rfc2141.txt>

[10] Corporation for National Research Initiatives. HANDLE.NET (version 8) Technical Manual. 2015. Dostupné z: <http://hdl.handle.net/4263537/5043>

[11] Fowler, M. Design - Who needs an architect? *IEEE Software*, volume 20, no. 5, 2003: pp. 11–13. Dostupné z: <https://doi.org/10.1109/MS.2003.1231144>

[12] Fowler, M. Microservice Trade-Offs. 2015. Dostupné z: <https://martinfowler.com/articles/microservice-trade-offs.html>

[13] Pardi, F. A JSON use case comparison between PostgreSQL and MongoDB. 2018. Dostupné z: <https://portavita.github.io/2018-10-31-blog_A_JSON_use_case_comparison_between_PostgreSQL_and_MongoDB>

[14] Sparx Systems. Database Models: Integrated Conceptual, Logical and Physical Schemas with Live Connections. 2020. Dostupné z: <https://sparxsystems.com/resources/user-guides/15.2/model-domains/database-models.pdf>

[15] Fielding, R. T.; Taylor, R. N. *Architectural Styles and the Design of Network-Based Software Architectures*. Dissertation thesis, 2000.

[16] Jones, M.; Bradley, J.; et al. JSON Web Token (JWT). 2015, doi:10.17487/RFC7519. Dostupné z: <https://rfc-editor.org/rfc/rfc7519.txt>

[17] Slootweg, S. Stop using JWT for sessions. 2016. Dostupné z: <http://cryto.net/~joepie91/blog/2016/06/13/stop-using-jwt-for-sessions/>

[18] Google. Integrating Google Sign-In into your web app. 2020. Dostupné z: <https://developers.google.com/identity/sign-in/web/sign-in>

[19] De Ryck, P. Why avoiding LocalStorage for tokens is the wrong solution. 2020. Dostupné z: <https://pragmaticwebsecurity.com/articles/oauthoidc/localstorage-xss.html>